数字电路与逻辑设计

单周期 CPU 设计

学院：数据科学与计算机学院
专业：软件工程
班别：教务二班软工 4 班
姓名：郑佳豪
学号：16305204
时间：2018 年 7 月 8 号

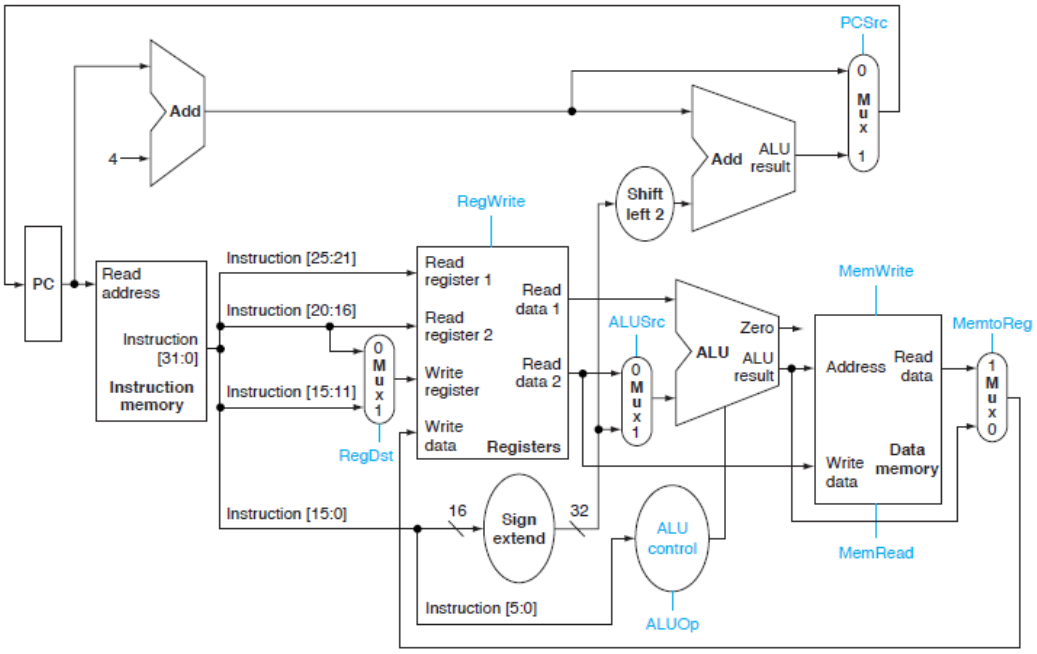
目录

单周期 CPU

概念

单周期 CPU 指的是一条指令的执行在一个时钟周期内完成，然后开始下一条指令的执行，即一条指令用一个时钟周期完成。电平从低到高变化的瞬间称为时钟上升沿，两个相邻时钟上升沿之间的时间间隔称为一个时钟周期。

工作原理



根据上图，我们知道单周期 CPU 在处理指令时，一般需要经过以下几个步骤：

- 取指令(IF)：根据程序计数器(PC)中的指令地址，从存储器中取出一条指令，同时 PC 根据指令字长度自动递增产生下一条指令所需要的指令地址，但遇到地址转移指令时，则控制器把转移地址送入 PC，当然得到的地址需要处理才送入 PC。
- 指令译码(ID)：对从 PC 中取到的指令进行译码，产生相应的操作控制信号。
- 指令执行(EXE)：根据指令译码得到的操作控制信号，执行相应的动作。
- 存储器访问(MEM)：所以需要访问存储器的操作都在此进行，把数据写入到存储器中数据地址所指定的存储单元或者从存储器中得到数据地址单元中的数据。
- 结果写回(WB)：指令执行的结果或者访问存储器中得到的数据写回相应的目的寄存器中。

MIPS 指令

介绍

MIPS(Microprocessor without Interlocked Pipeline Stages)，是一种采取精简指令集(RISC)的处理器架构，于 1981 年出现，由 MIPS 科技公司开发并授权，广泛被使用在许多电子产品、网络设备、个人娱乐设备和商业设备中。

关于 MIPS 的更详细的信息，请查阅 https://en.wikipedia.org/wiki/MIPS_architecture。

指令类型

在 MIPS 架构中，指令被分为三种类型：R 型、I 型和 J 型。三种类型的指令的最高 6 位均为 6 位的 opcode 码。从 25 位往下，

- R 型指令用连续三个 5 位二进制码来表示三个寄存器的地址，然后用一个 5 位二进制码来表示移位的位数（如果未使用移位操作，则全为 0），最后为 6 位的 function 码（它与 opcode 码共同决定 R 型指令的具体操作方式）；
- I 型指令则用连续两个 5 位二进制码来表示两个寄存器的地址，然后是一个 16 位二进制码来表示的一个立即数二进制码；
- J 型指令用 26 位二进制码来表示跳转目标的指令地址(实际的指令地址应为 32 位，其中最低两位为 00，高四位由 PC 当前地址决定)

MIPS 使用的是大端存储模式，关于大端存储的更详细信息，请查阅 https://en.wikipedia.org/wiki/Endianness#Big-endian

R 类型

Type	-31-		Format(bits)			-0-
R	opcode(6)	rs(5)	rt(5)	rd(5)	shamt(5)	funct(6)

I 类型

Type	-31-		Format(bits)	-0-
I	opcode(6)	rs(5)	rt(5)	immediate(16)

J 类型

Type	-31-	Format(bits)	-0-
J	opcode(6)	address(26)	

上述类型指示图中的字母简写含义如下：

- opcode: 6 位操作码
- rs: 第 1 个源操作数寄存器，寄存器地址编号为 00000-11111，即 00-1F
- rt: 第 2 个源操作数寄存器或目的操作数寄存器，寄存器地址编号为 00-1F
- rd: 目的操作数寄存器，寄存器地址编号为 00-1F
- shamt: 位移量(Shift Amount)，在移位指令中用于指定移动的位数
- funct: 为功能码，在 R 类型指令中用来指定指令功能
- immediate: 16 位立即数，用于无符号的逻辑操作数、有符号的算术操作数、数据加载和数据保存指令的数据地址字节偏移量、分支指令中相对 PC 的有符号偏移量

常见运算指令

算术运算指令

- add rd, rs, rt

000000	rs(5)	rt(5)	rd(5)	Reserved

  功能：rd = rs + rt。reserved 为预留部分，即未用，一般填“0”
- addi rt, rs, immediate

000001	rs(5)	rt(5)	immediate(16)

  功能：rt = rs + (sign-extend)immediate；对 immediate 符号扩展，进行加运算
- sub rd, rs, rt

000010	rs(5)	rt(5)	rd(5)	Reserved

  功能：rt = rs + (sign-extend)immediate；对 immediate 符号扩展，进行加运算

逻辑运算指令

- ori rt, rs, immediate

010000	rs(5)	rt(5)	immediate(16)

  功能：rt = rs | (zero-extend)immediate；对 immediate 零扩展，进行或运算
- and rd, rs, rt

010001	rs(5)	rt(5)	rd(5)	reserved

  功能：rd = rs & rt；逻辑与运算
- or rd, rs, rt

010010	rs(5)	rt(5)	rd(5)	reserved

  功能：rd = rs | rt；逻辑或运算

移位指令

- sll rd, rt, shamt

011000	未用	rt(5)	rd(5)	shamt	reserved

  功能：rd = rt << (zero-extend)shamt，左移(zero-extend)shamt 位

比较指令

- slt rd, rs, rt　　带符号数

011100	rs(5)	rt(5)	rd(5)	reserved

  功能：rd = rs < rt ? 1 : 0

存储器读/写指令

- sw rt, immediate(rs) 写存储器

100110	rs(5)	rt(5)	immediate(16)

  功能：memory[rs+ (sign-extend)immediate] = rt
- lw rt, immediate(rs) 读存储器

100111	rs(5)	rt(5)	immediate(16)

  功能：rt = memory[rs + (sign-extend)immediate]

分支指令

- beq rs, rt, immediate

110000	rs(5)	rt(5)	immediate(偏移量，16)

  功能：if(rs = rt) pc = pc + 4 + (sign-extend)immediate <<2;

- bne rs, rt, immediate

110001	rs(5)	rt(5)	immediate(16)

  功能：if(rs != rt) pc = pc + 4 + (sign-extend)immediate <<2 else pc = pc + 4
- bgtz rs, immediate

110010	rs(5)	00000	immediate(16)

  功能：if(rs>0) pc = pc + 4 + (sign-extend)immediate <<2　else pc ←pc + 4

跳转指令

- j addr

111000	addr[27..2]

  功能：pc = {(pc+4)[31..28], addr[27..2], 0, 0}，无条件跳转。

停机指令

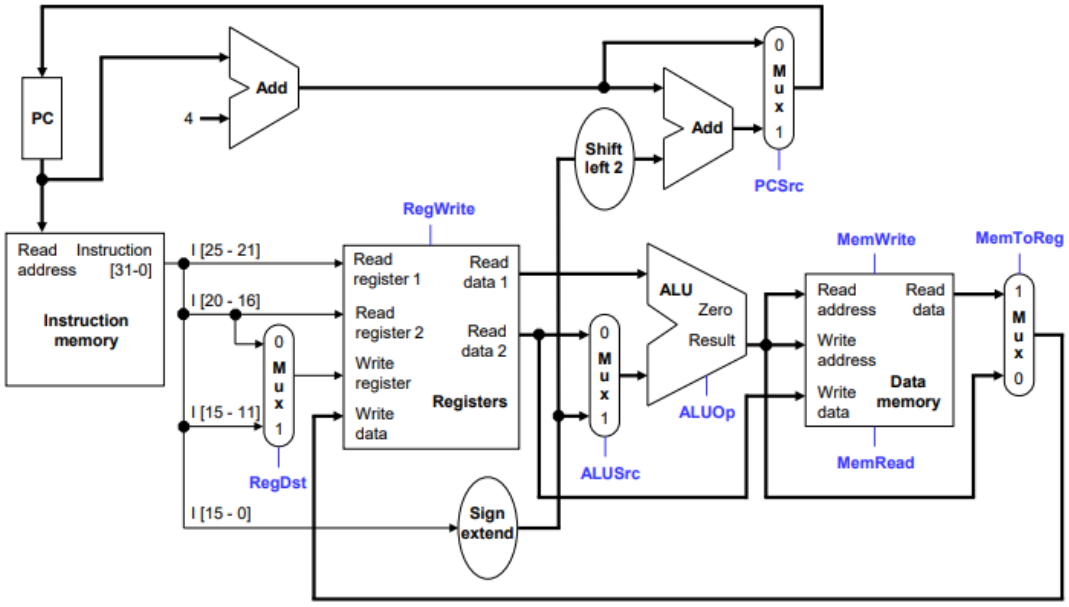
- halt

111111	00000000000000000000000000(26)

  功能：停机，pc 保持不变。

设计思路

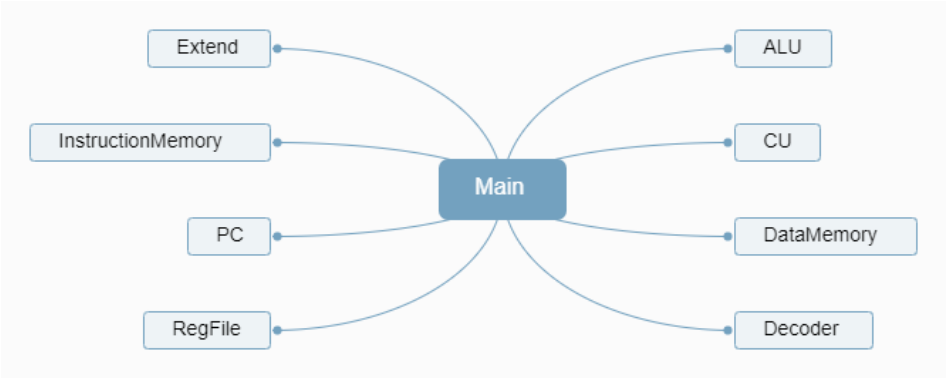
数据通路



上图对我们编写 Verilog 代码，实现单周期 CPU 是至关重要的。下面是对各控制信号的说明：

控制信号	状态 0	状态 1
RESET	初始化 PC 为 0	PC 接收新地址
ALUSRC	来自寄存器 data2 输出，如 addu、subu、or、and、bne	来自 sign 或 zero 扩展的立即数，如 addiu、ori、sw、lw
REGWRITE	不写入寄存器，如 beq、sw、halt	写入寄存器，如 add、addi、sub、ori、or、and、sll、lw
MEMTOREG	来自 ALU 输出，除 lw 以外	来自数据存储器输出，如 lw
MEMREAD	读数据存储器，如 lw	输出高阻态
MEMWRITE	写入数据存储器，如 sw	无操作
EXTSEL	立即数零扩展，如 ori	立即数符号扩展，如 addi、sw、lw、beq
REGDST	写寄存器组寄存器的地址，如 addi、ori、lw	写寄存器组寄存器的地址，如 add、sub、and、or、sll
PCSRC	PC=PC+4，如 add、addi、sub、ori、or、and、sw、sll、lw、beq(zero=0)	PC=PC+4+符号扩展后的立即数，如 beq(zero=1)
ALUOP [2:0]	选择 ALU 功能	

模块设计



参照上面提及到的数据通路，我将项目拆分为上图所示的几个部分。Main 为顶层模块，用来将其余八大模块联系。

- CU：控制数据路线
- ALU：负责算术逻辑运算
- DataMemory：数据存储器
- Decoder：负责按照 MIPS 指令要求解析指令
- Extend：对立即数进行零扩展和符号扩展
- InstructionMemory：指令存储器
- PC：负责程序要处理指令的提供
- RegFile：寄存器堆

指令宏

为了开发的方便，我在项目开发使用了宏，即将易输错的机器码，使用别名代替，从而减少项目中隐藏的错误发生的概率。

```
// Define directives used in development
`define OP_R_DEV 6'b000000

// Op
`define OP_ADD 6'b000000      // Add
`define OP_ADDI 6'b001000     // Add Immediate
`define OP_AND 6'b000000      // And
`define OP_BEQ 6'b000100      // Branch on Equal
`define OP_BGTZ 6'b000111     // Branch on Greater Than Zero
`define OP_BNE 6'b000101      // Branch on Not Equal
`define OP_HALT 6'b111111     // Halt
`define OP_J 6'b000010        // Jump
`define OP_LW 6'b100011       // Load Word
`define OP_OR 6'b000000       // Or
`define OP_ORI 6'b001101      // Or Immediate
`define OP_SLL 6'b000000      // Shift Left Logical
`define OP_SLT 6'b000000      // Set on Less Than
`define OP_SUB 6'b000000      // Sub
`define OP_SW 6'b101011       // Save Word

`define FUNCT_ADD 6'b100000
`define FUNCT_SUB 6'b100011
`define FUNCT_AND 6'b100100
`define FUNCT_OR 6'b100101
`define FUNCT_SLL 6'b000000
`define FUNCT_SLT 6'b101010

// ALU
`define ALU_ADD 3'b000
`define ALU_AND 3'b010
`define ALU_CMPS 3'b101
`define ALU_CMPU 3'b100
`define ALU_FROM_DATA 1'b0
`define ALU_FROM_IMMD 1'b1
`define ALU_FROM_SA 1'b1
`define ALU_OR 3'b011
`define ALU_SLL 3'b110
`define ALU_SUB 3'b001

// PC
`define PC_ABS_JMP 2'b10
`define PC_HALT 2'b11
`define PC_NEXT_INS 2'b00
`define PC_REL_JMP 2'b01

// Reg
`define REG_FROM_ALU 1'b0
`define REG_FROM_DATAMEMORY 1'b1
`define REG_FROM_RD 1'b1
`define REG_FROM_RT 1'b0

// Extend
`define EXT_SIGN 1'b1
`define EXT_ZERO 1'b0
```

Control Unit

功能介绍

控制数据通路的路线，即控制其余各模块的数据通路。

代码实现

```
`timescale 1ns / 1ps
`include "Constants.v"

module CU(
  input [5:0] Opcode,
  input [5:0] Funct,
  input Zero,
  input Sign,
  output ALUSrcA,
  output reg ALUSrcB,
  output RegDst,
  output MemToReg,
  output ExtSel,
  output MemRead,
  output MemWrite,
  output reg RegWrite,
  output reg [1:0] PCSrc,
  output reg [2:0] ALUOp
);
  // ALUSrcA
  assign ALUSrcA = (Opcode == `OP_SLL && Funct == `FUNCT_SLL) ? `ALU_FROM_SA :
`ALU_FROM_DATA;
  // ALUSrcB
  always@(*) begin
    case(Opcode)
      `OP_ADDI, `OP_LW, `OP_SW, `OP_ORI: ALUSrcB = `ALU_FROM_IMMD;
      default: ALUSrcB = `ALU_FROM_DATA;
    endcase
  end
  // MemToReg
  assign MemToReg = Opcode == `OP_LW ? `REG_FROM_DATAMEMORY : `REG_FROM_ALU;
  // MemRead
  assign MemRead = Opcode == `OP_LW ? 0 : 1;
  // MemWrite
  assign MemWrite = Opcode == `OP_SW ? 0 : 1;
  // RegWrite
  always@(*) begin
    case(Opcode)
      `OP_SW, `OP_BEQ, `OP_BNE, `OP_BGTZ, `OP_J, `OP_HALT: RegWrite = 0;
      default: RegWrite = 1;
    endcase
  end
  // RegDst
  assign RegDst = Opcode == `OP_LW || Opcode == `OP_ADDI || Opcode == `OP_ORI ?
`REG_FROM_RT : `REG_FROM_RD;
  // ExtSel
  assign ExtSel = Opcode == `OP_ORI ? `EXT_ZERO : `EXT_SIGN;
  // PCSrc
  always@(*) begin
    case(Opcode)
      `OP_BEQ: PCSrc = Zero == 1 ? `PC_REL_JMP : `PC_NEXT_INS;
      `OP_BNE: PCSrc = Zero == 1 ? `PC_NEXT_INS : `PC_REL_JMP;
      `OP_BGTZ: PCSrc = Sign == 0 && Zero == 0 ? `PC_REL_JMP : `PC_NEXT_INS;
      `OP_J: PCSrc = `PC_ABS_JMP;
      `OP_HALT: PCSrc = `PC_HALT;
      default: PCSrc = `PC_NEXT_INS;
    endcase
  end
  // ALUOp
  always@(*) begin
    case(Opcode)
      `OP_R_DEV: begin
        case(Funct)
          `FUNCT_ADD: ALUOp = `ALU_ADD;
          `FUNCT_SUB: ALUOp = `ALU_SUB;
          `FUNCT_AND: ALUOp = `ALU_AND;
          `FUNCT_OR: ALUOp = `ALU_OR;
          `FUNCT_SLL: ALUOp = `ALU_SLL;
          `FUNCT_SLT: ALUOp = `ALU_CMPS;
          default: ALUOp = `ALU_ADD;
        endcase
      end
      `OP_ORI: ALUOp = `ALU_OR;
      `OP_BEQ, `OP_BNE, `OP_BGTZ: ALUOp = `ALU_SUB;
      default: ALUOp = `ALU_ADD;
    endcase
  end
endmodule
```

PC

功能介绍

PC 负责为处理器提供要处理的下一指令的地址，在本项目中我将 PC 模块拆分为两个模块，一个模块 PCHelper 根据正在执行的指令，计算下一条指令的地址，另个模块 PC 则将 PCHelper 计算的地址在时钟上升沿或重置下降沿时作为下一周期执行的指令。

代码实现

```
`timescale 1ns / 1ps
`include "Constants.v"

module PC(
  input CLK,
  input Reset,
  input [31:0] NewPC,
  output reg [31:0] pc
);

  always@(posedge CLK or negedge Reset) begin
    pc <= Reset == 0 ? 0 : NewPC;
  end

endmodule

module PCHelper(
  input [31:0] pc,
  input [15:0] Immediate,
  input [25:0] Address,
  input [1:0] PCSrc,
  output reg [31:0] NewPC
);

  initial begin
    NewPC = 0;
  end

  // Extend 16-bit immediate value to 32-bit immediate value
  wire [31:0] ExtImmediate = {{16{Immediate[15]}}, Immediate};

  always@(*) begin
    case(PCSrc)
      `PC_NEXT_INS: NewPC <= pc + 4;
      `PC_REL_JMP: NewPC <= pc + 4 + (ExtImmediate << 2);
      `PC_ABS_JMP: NewPC <= {pc[31:28], Address, 2'b00};
      `PC_HALT: NewPC <= pc;
    endcase
  end

endmodule
```

Decoder

功能说明

将从 InstructionMemory 得到的指令进行解析，输出 opcode、rs、rt、rd、shamt、funct、immediate、address。

输入说明

输入	说明
INS [31:0]	欲进行解析的指令

输出说明

输出	说明
OPCODE [5:0]	6 位操作码
FUNCT [5:0]	功能码
SHAMT [4:0]	位移量 Shift Amount
RS [4:0]	第 1 个源操作数寄存器
RT [4:0]	第 2 个源操作数寄存器或目的操作数寄存器
RD [4:0]	目的操作数寄存器
IMMEDIATE [15:0]	16 位立即数
ADDRESS [25:0]	地址

代码实现

```
`timescale 1ns / 1ps

/* Instruction formats
 * Instructions are divided into three types: R, I and J. Every instruction
 * starts with a 6-bit opcode. In addition to the opcode, R-type instructions
 * specify three registers, a shift amount field, and a function field; I-type
 * instructions specify two registers and a 16-bit immediate value; J-type
 * instructions follow the opcode with a 26-bit jump target.
 *
 * https://en.wikipedia.org/wiki/MIPS_architecture#Instruction_formats
 */

module Decoder(
  input [31:0] Ins,
  output [5:0] Opcode,
  output [5:0] Funct,
  output [4:0] Shamt,
  output [4:0] rs,
  output [4:0] rt,
  output [4:0] rd,
  output [15:0] Immediate,
  output [25:0] Address
);

  assign Opcode = Ins[31:26];
  assign Funct = Ins[5:0];
  assign Shamt = Ins[10:6];
  assign rs = Ins[25:21];
  assign rt = Ins[20:16];
  assign rd = Ins[15:11];
  assign Immediate = Ins[15:0];
  assign Address = Ins[25:0];

endmodule
```

InstructionMemory

功能说明

根据当前的 PC 地址，得到对应地址的指令。

输入说明

输入	说明
ADDRESS [31:0]	执行读取的指令地址

输出说明

输出	说明
INS [31:0]	输出读取出来的指令

代码实现

```
`timescale 1ns/1ps

module InstructionMemory(
  input [31:0] Address,
  output reg [31:0] Ins
);

  reg [7:0] ROM [0:99];


///Here you should add your own path to the ROM.txt file on your own computer.
  // in the path.
  initial begin
    $readmemh ("E:/Users/Code/Single-Cycle-CPU/test/ROM.txt", ROM);
  end

  // MIPS Architecture uses the Big-Endian order.
  always@(*) begin
    Ins[31:24] = ROM[Address];
    Ins[23:16] = ROM[Address + 1];
    Ins[15:8] = ROM[Address + 2];
    Ins[7:0] = ROM[Address + 3];
  end

endmodule
```

RegFile

功能说明

为其余模块提供读取和写入寄存器的功能。

输入说明

输入	说明
CLK	时钟信号
RESET	复位信号
REGWRITE	使能信号，为 0 时不写入，为 1 时写入数据
READREG1	读数据地址输入端
READREG2	读数据地址输入端
WRITEREG	写数据地址输入端
WRITEDATA	写入数据

输出说明

输出	说明
READDATA1 [31:0]	读数据输出端
READDATA2 [31:0]	读数据输出端

代码实现

```
`timescale 1ns / 1ps

module RegFile(
  input CLK,
  input Reset,
  input RegWrite,
  input [4:0] ReadReg1,
  input [4:0] ReadReg2,
  input [4:0] WriteReg,
  input [31:0] WriteData,
  output [31:0] ReadData1,
  output [31:0] ReadData2
);

  integer index;
  reg [31:0] regFile[1:31];

  assign ReadData1 = ReadReg1 == 0 ? 0 : regFile[ReadReg1];
  assign ReadData2 = ReadReg2 == 0 ? 0 : regFile[ReadReg2];

  always@(negedge CLK or negedge Reset) begin
    if (Reset == 0) begin
      // The for statement using here would increase unneccessary units'usage.
      // However, to make the code more developer-friendly, I used this kind
      // of statement.
      for (index = 1; index < 32; index = index + 1) begin
        regFile[index] <= 0;
      end
    end
    else if (RegWrite == 1 && WriteReg != 0) begin
      regFile[WriteReg] <= WriteData;
    end
  end

endmodule
```

DataMemory

功能说明

提供读取和写入数据的功能。

输入说明

输入	说明
CLK	时钟信号
ADDRESS [31:0]	要操作的内存地址
WRITEDATA [31:0]	来自寄存器要写入的数据
MEMREAD	控制信号为 0，进行读操作
MEMWRITE	控制信号为 1，进行写操作

输出说明

输出	说明
READDATA [31:0]	输出读操作的结果

代码实现

```
`timescale 1ns / 1ps

module DataMemory(
  input CLK,
  input [31:0] Address,
  input [31:0] WriteData,
  input MemRead,
  input MemWrite,
  output [31:0] ReadData
);

  reg [7:0] RAM [0:60];

  assign ReadData[7:0] = MemRead == 0 ? RAM[Address + 3] : 8'bz;
  assign ReadData[15:8] = MemRead == 0 ? RAM[Address + 2] : 8'bz;
  assign ReadData[23:16] = MemRead == 0 ? RAM[Address + 1] : 8'bz;
  assign ReadData[31:24] = MemRead == 0 ? RAM[Address] : 8'bz;

  always@(negedge CLK) begin
    if (MemWrite == 0) begin
      RAM[Address] <= WriteData[31:24];
      RAM[Address + 1] <= WriteData[23:16];
      RAM[Address + 2] <= WriteData[15:8];
      RAM[Address + 3] <= WriteData[7:0];
    end
  end

endmodule
```

ALU

功能说明

ALU 模块接收寄存器的数据和控制信号的输入，将运算后的结果输出。

输入说明

输入	说明
ALUOP [2:0]	控制信号，用于运算功能的选择
READDATA1 [31:0]	来自寄存器 rs 的数据
READDATA2 [31:0]	来自寄存器 rt 的数据

输出说明

输出	说明
ZERO	当运算结果为 0，输出为 1，否则输出为 0
SIGN	运算结果最高位，0 为正数，1 为负数
RESULT [31:0]	运算结果

ALU 功能

ALUOP [2:0]	功能	描述
000	Y = A + B	加
001	Y = A – B	减
010	Y = A & B	与
011	Y = B | A	或
100	Y = A < B ? 1 : 0	无符号判断 A 与 B
101	Y = (A < B && A[31] == B[31]) || (A[31] == 1 && B[31] == 0) ? 1 : 0	有符号判断 A 与 B
110	Y = B << A	B 左移 A 位

代码实现

```
`timescale 1ns / 1ps
`include "Constants.v"

module ALU(
  input [2:0] ALUOp,
  input [31:0] ReadData1,
  input [31:0] ReadData2,
  output reg [31:0] Result,
  output Zero,
  output Sign
);

  initial begin
    Result = 0;
  end

  assign Zero = Result == 0 ? 1 : 0;
  assign Sign = Result[31];

  always@(*) begin
    case(ALUOp)
      `ALU_ADD: Result = ReadData1 + ReadData2;
      `ALU_SUB: Result = ReadData1 - ReadData2;
      `ALU_AND: Result = ReadData1 & ReadData2;
      `ALU_OR: Result = ReadData1 | ReadData2;
      `ALU_SLL: Result = ReadData2 << ReadData1;
      `ALU_CMPU: Result = ReadData1 < ReadData2 ? 1 : 0;
      `ALU_CMPS: Result = (ReadData1 < ReadData2 && ReadData1[31] == ReadData2[31]))
                          || (ReadData1[31] == 1 && ReadData2[31] == 0)
                          ? 1 : 0;
      default: Result = 8'h00000000;
    endcase
  end

endmodule
```

Extend

功能说明

将一个 16 位立即数扩展到 32 位。

输入说明

输入	说明
IMMEDIATE [15:0]	输入的 16 位立即数
EXTSEL	控制信号为 1，进行符号扩展；为 0，进行零扩展

输出说明

输入	说明
IMMEDIATE [32:0]	扩展后的 32 位立即数

代码实现

```
`timescale 1ns / 1ps

module Extend(
  input [15:0] Immediate,
  input ExtSel,
  output [31:0] ExtImmediate
);

  assign ExtImmediate[15:0] = Immediate[15:0];
  assign ExtImmediate[31:16] = ExtSel == 1 && Immediate[15] == 1 ? 16'hFFFF : 16'h0000;

endmodule
```

顶层 CPU 模块

功能说明

顶层模块是 CPU 的综合模块，能将各个子模块联系，从而实现 CPU 功能。

代码实现

```
`timescale 1ns / 1ps

module CPU(
  input CLK,
  input Reset
);

  wire [31:0] pc;
  wire [31:0] NewPC;
  wire [15:0] Immediate;
  wire [25:0] Address;
  wire [1:0] PCSrc;
  wire [31:0] Ins;
  wire [5:0] Opcode;
  wire [5:0] Funct;
  wire [4:0] Shamt;
  wire [4:0] rs;
  wire [4:0] rt;
  wire [4:0] rd;
  wire [31:0] ALUResult;
  wire [31:0] RAMOut;
  wire [4:0] WriteReg;
  wire [31:0] RegWriteData;
  wire [31:0] RegReadData1;
  wire [31:0] RegReadData2;
  wire [31:0] ExtImmediate;
  wire Zero;
  wire Sign;
  wire ALUSrcA;
  wire ALUSrcB;
  wire MemToReg;
  wire RegWrite;
  wire MemRead;
  wire MemWrite;
  wire RegDst;
  wire ExtSel;
  wire [2:0] ALUOp;
  wire [31:0] ALUReadData1;
  wire [31:0] ALUReadData2;

  PC PC(
    .CLK(CLK),
    .Reset(Reset),
    .NewPC(NewPC),
    .pc(pc)
  );

  PCHelper PCHelper(
    .pc(pc),
    .Immediate(Immediate),
    .Address(Address),
    .PCSrc(PCSrc),
    .NewPC(NewPC)
  );

  InstructionMemory InstructionMemory(
    .Address(pc),
    .Ins(Ins)
  );

  Decoder decoder(
    .Ins(Ins),
    .Opcode(Opcode),
    .Funct(Funct),
    .Shamt(Shamt),
    .rs(rs),
    .rt(rt),
    .rd(rd),
    .Immediate(Immediate),
    .Address(Address)
  );
```

```
assign WriteReg = RegDst == `REG_FROM_RT ? rt : rd;
  assign RegWriteData = MemToReg == `REG_FROM_ALU ? ALUResult : RAMOut;

  RegFile RegFile(
    .CLK(CLK),
    .Reset(Reset),
    .RegWrite(RegWrite),
    .ReadReg1(rs),
    .ReadReg2(rt),
    .WriteReg(WriteReg),
    .WriteData(RegWriteData),
    .ReadData1(RegReadData1),
    .ReadData2(RegReadData2)
  );

  Extend Extend(
    .Immediate(Immediate),
    .ExtSel(ExtSel),
    .ExtImmediate(ExtImmediate)
  );

  assign ALUReadData1 = ALUSrcA == `ALU_FROM_DATA ? RegReadData1 : {{27{1'b0}}, Shamt};
  assign ALUReadData2 = ALUSrcB == `ALU_FROM_DATA ? RegReadData2 : ExtImmediate;

  ALU ALU(
    .ALUOp(ALUOp),
    .ReadData1(ALUReadData1),
    .ReadData2(ALUReadData2),
    .Result(ALUResult),
    .Zero(Zero),
    .Sign(Sign)
  );

  DataMemory DataMemory(
    .CLK(CLK),
    .Address(ALUResult),
    .WriteData(RegReadData2),
    .MemRead(MemRead),
    .MemWrite(MemWrite),
    .ReadData(RAMOut)
  );

  CU CU(
    .Opcode(Opcode),
    .Funct(Funct),
    .Zero(Zero),
    .Sign(Sign),
    .ALUSrcA(ALUSrcA),
    .ALUSrcB(ALUSrcB),
    .MemToReg(MemToReg),
    .ExtSel(ExtSel),
    .MemRead(MemRead),
    .MemWrite(MemWrite),
    .RegDst(RegDst),
    .RegWrite(RegWrite),
    .PCSrc(PCSrc),
    .ALUOp(ALUOp)
  );

endmodule
```

仿真文件

功能说明

用于实例化 CPU 模块，给定初始的 CLK 和 Reset 信号。

代码实现

```
`timescale 1ns/1ps

module Main_tb();
  reg CLK = 1;
  reg Reset = 1;

  always begin
    #5;
    CLK = ~CLK;
  end

  initial begin
    #1;
    Reset = 0;
    #1;
    Reset = 1;
  end

  CPU CPU(
    .CLK(CLK),
    .Reset(Reset)
  );

endmodule
```

功能测试

测试文件

详细测试文件说明，请查看 test 文件夹的 README.docx 文件。

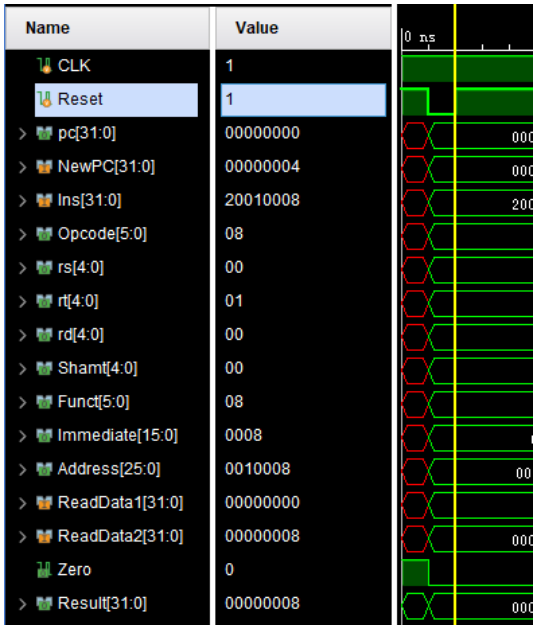
```
Address      Instruction    Description

0x00000000   20 01 00 08    // addi $1, $0, 8
0x00000004   34 02 00 02    // ori $2, $0, $2
0x00000008   00 41 18 20    // add $3, $2, $1
0x0000000C   00 62 28 23    // sub $5, $3, $2
0x00000010   00 A2 20 24    // and $4, $5, $2
0x00000014   00 82 40 25    // or $8, $4, $2
0x00000018   00 08 40 40    // sll $8, $8, 1
0x0000001C   15 01 FF FE    // bne $8, $1, -2
0x00000020   00 41 30 2A    // slt $6, $2, $1
0x00000024   00 C0 38 2A    // slt $7, $6, $0
0x00000028   20 E7 00 08    // addi $7, $7, 8
0x0000002C   10 E1 FF FE    // beq $7, $1, -2
0x00000030   AC 22 00 04    // sw $2, 4
0x00000034   8C 29 00 04    // lw $9, 4
0x00000038   1D 20 00 01    // bgtz $9, 4
0x0000003C   FC 00 00 00    // halt
0x00000040   20 09 FF FF    // addi $9, $0, -1
0x00000044   08 00 00 0E    // j 0x00000038
```

功能验证

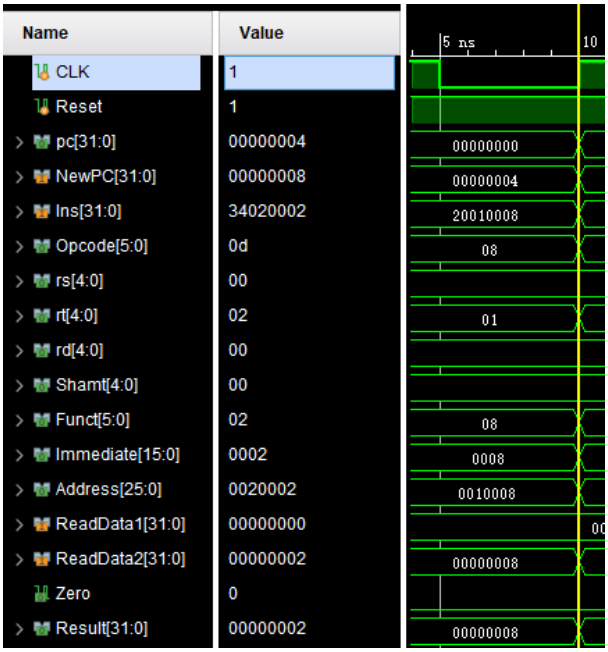
addi $1, $0, 8

可以看到：PC 为：00000000，Instruction 为：20010008，表示与立即数的加法运算，其中寄存器分别是$1，$0，立即数是 8，所以 ReadData1 是 0，ReadData2 是 8，ALUResult 是 8。



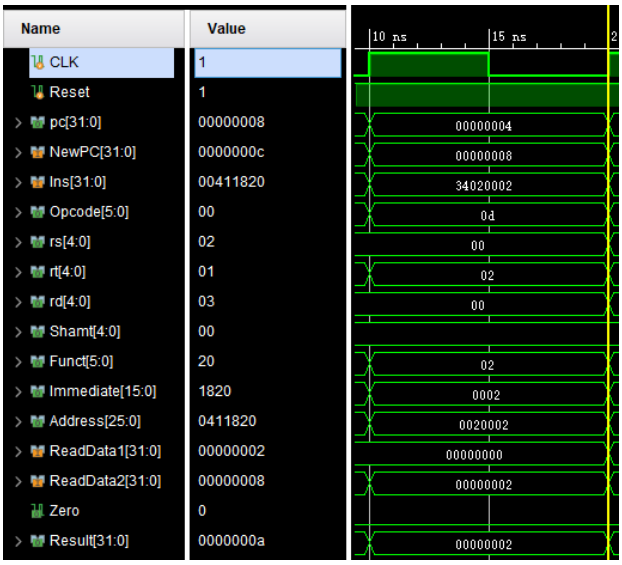
ori $2, $0, $2

可以看到：PC 为：00000004，Instruction 为：34020002，表示与立即数的或运算，其中寄存器分别是$2，$0，$2，立即数是 2，所以 ReadData1 是 0，ReadData2 是 2，ALUResult 是 2。



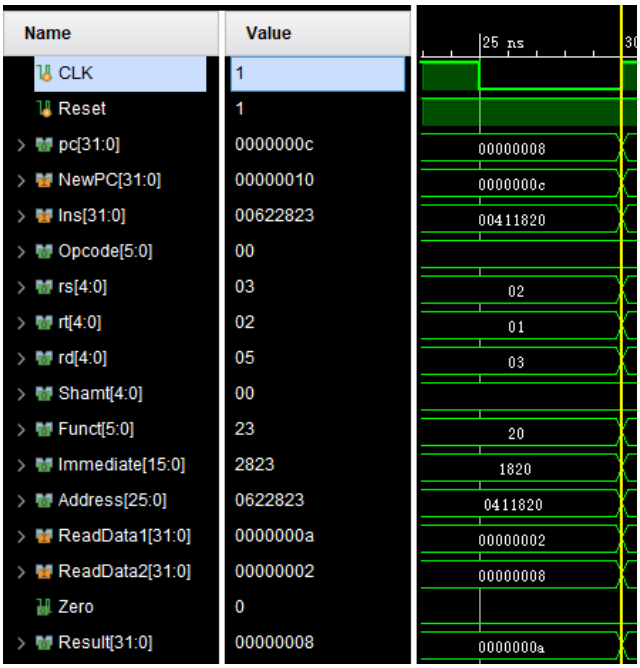
add $3, $2, $1

可以看到：PC 为：00000008，Instruction 为：00411820，表示加法运算，其中寄存器分别是$3，$2，$1，所以 ReadData1 是 2，ReadData2 是 8，ALUResult 是 10。



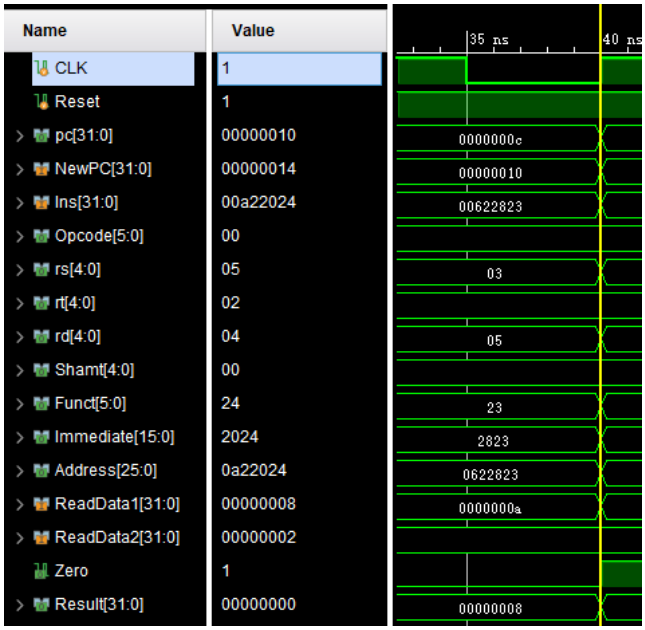
sub $5, $3, $2

可以看到：PC 为：0000000C，Instruction 为：00622823，表示减法运算，其中寄存器分别是$5，$3，$2，所以 ReadData1 是 10，ReadData2 是 2，ALUResult 是 8。



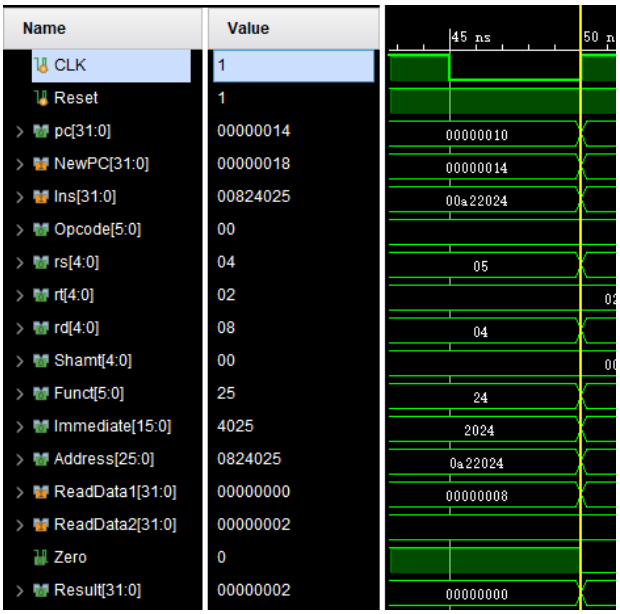
and $4, $5, $2

可以看到：PC 为：00000010，Instruction 为：00a22024，表示与运算，其中寄存器分别是$4，$5，$2，所以 ReadData1 是 8，ReadData2 是 2，ALUResult 是 0。



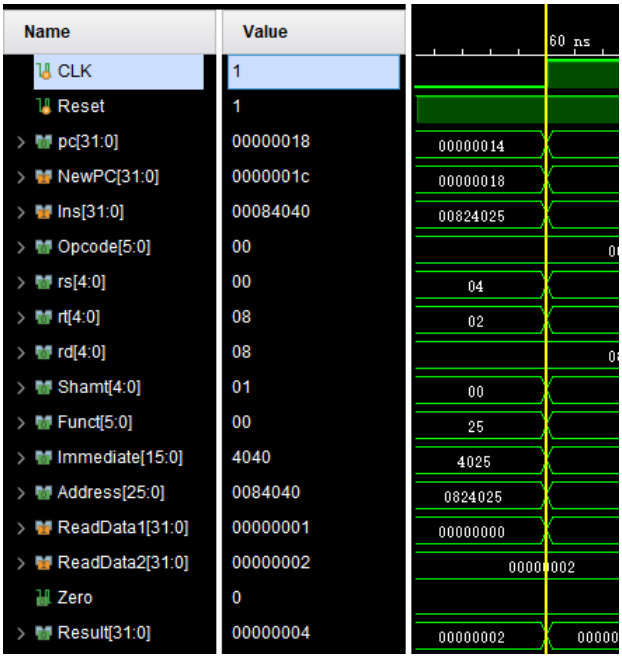
or $8, $4, $2

可以看到：PC 为：00000014，Instruction 为：00824025，表示或运算，其中寄存器分别是 $8，$4，$2，所以 ReadData1 是 0，ReadData2 是 2，ALUResult 是 2。



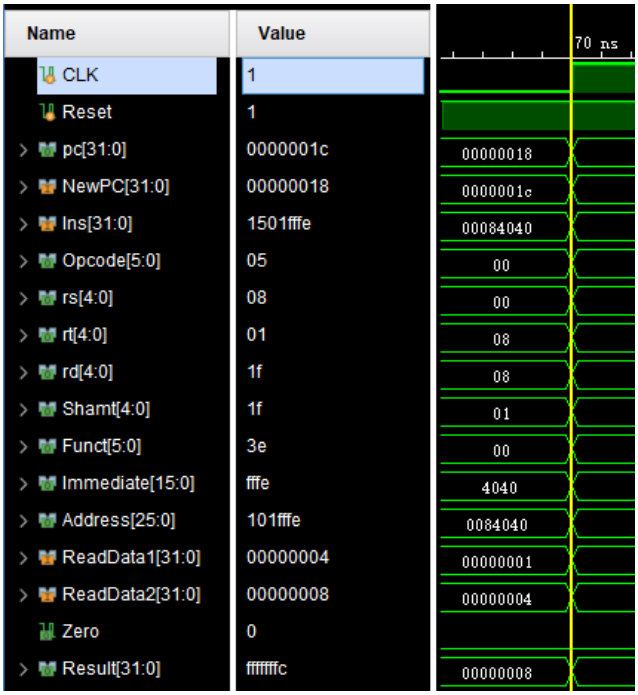
sll $8, $8, 1

可以看到：PC 为：00000018，Instruction 为：00084040，表示左移位运算，其中寄存器是 $8，$8，左移位数是 1*2=2，所以 ReadData1 是 1，ReadData2 是 2，ALUResult 是 4。



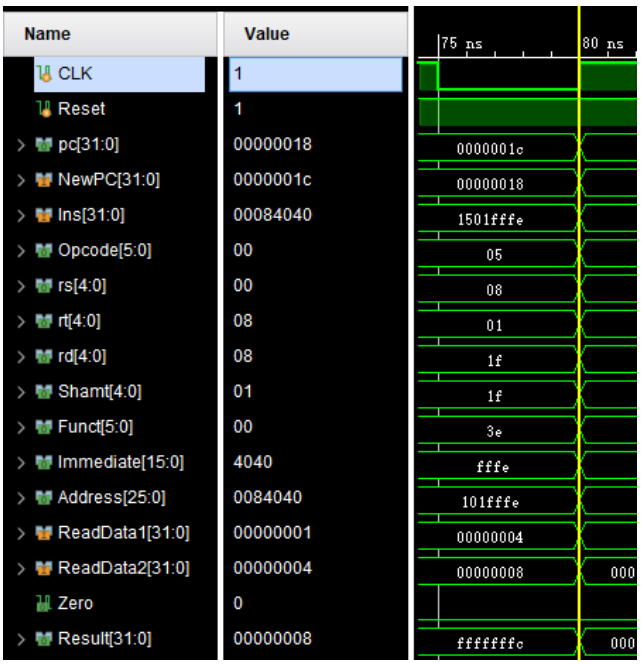
bne $8, $1, -2

可以看到：PC 为：0000001C，Instruction 为：1501FFFE，表示分支指令，由于 Zero 为 0，故将跳转至 00000018。



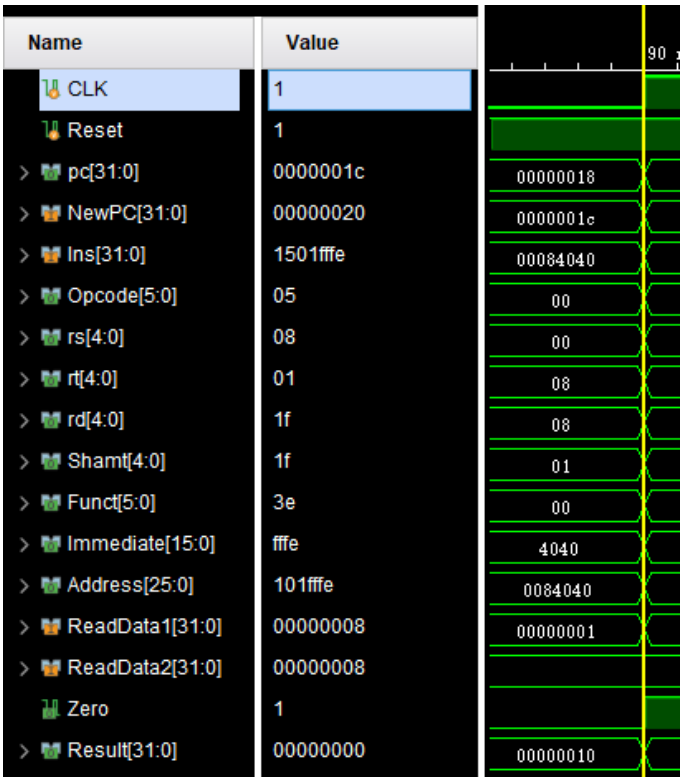
sll $8, $8, 1

可以看到：PC 为：00000018，Instruction 为：00084040，表示左移位运算，其中寄存器是 $8，左移位数是 1，所以 ReadData1 是 1，ReadData2 是 4，ALUResult 是 8。



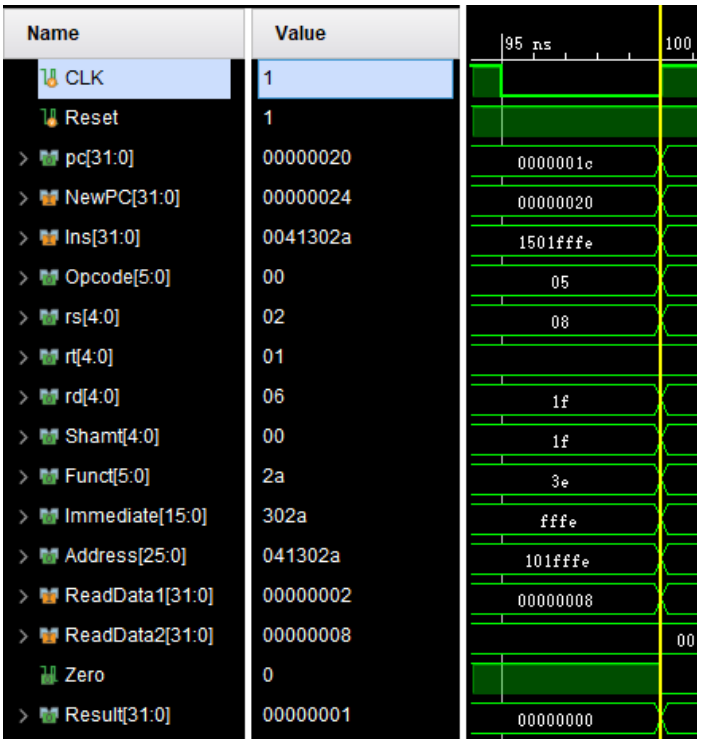
bne $8, $1, -2

可以看到：PC 为：0000001C，Instruction 为：1501FFFE，表示分支指令，由于 Zero 为 1，故将跳转至 00000020。



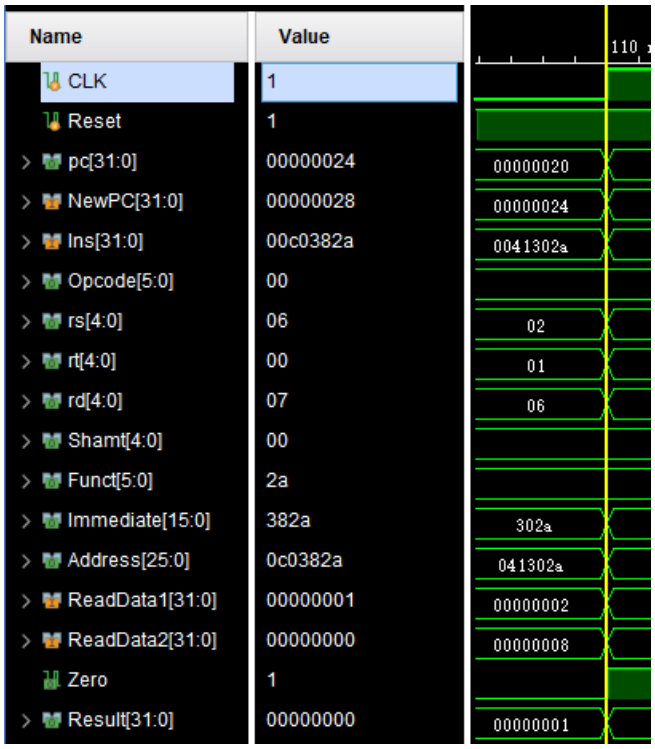
slt $6, $2, $1

可以看到：跳转完成后，PC 为：00000020，Instruction 为：0041302A，表示比较运算，其中寄存器分别是$6，$2，$1，由于 rs<rt，故 ALUResult 为 1



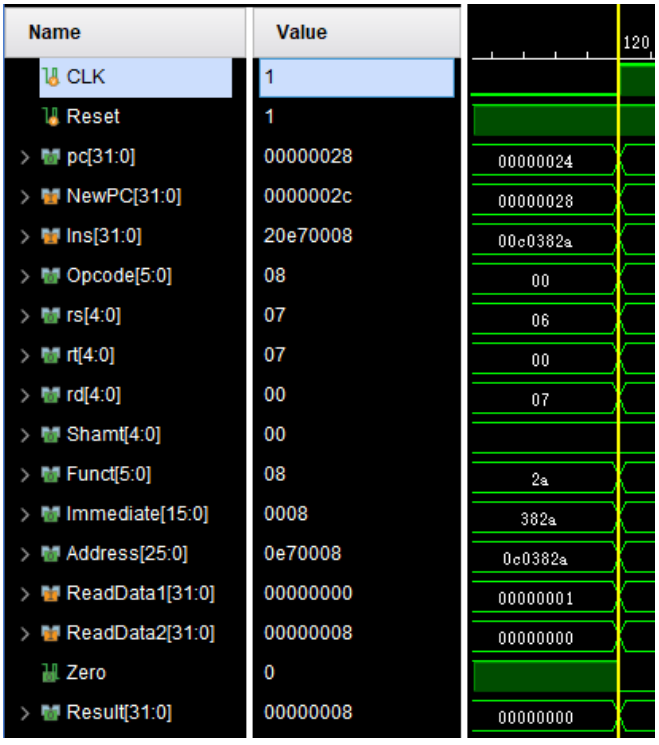
slt $7, $6, $0

可以看到：PC 为：00000024，Instruction 为：00C0382A，表示比较运算，其中寄存器分别是$7，$6，$0，由于 rs>rt，故 ALUResult 为 0



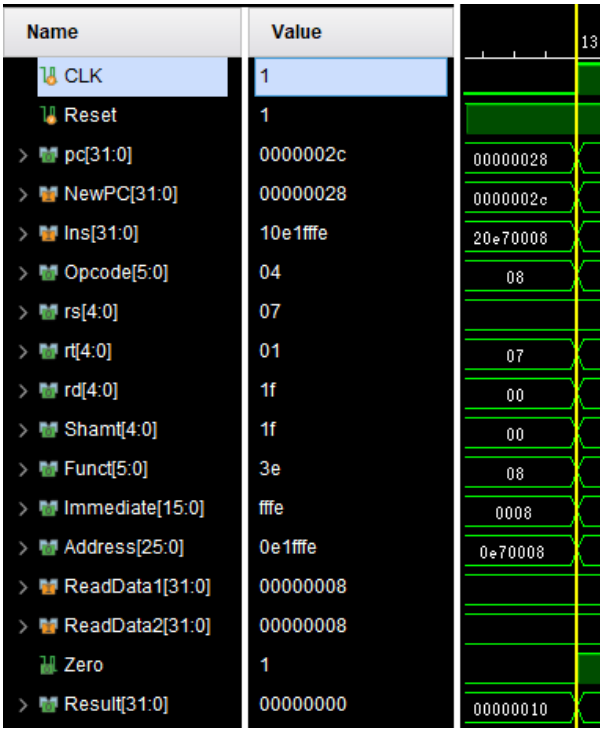
addi $7, $7, 8

可以看到：PC 为：00000028，Instruction 为：20E70008，表示与立即数的加法运算，其中寄存器分别是$7，立即数是 8，所以 ReadData1 是 0，ReadData2 是 8，ALUResult 是 8。



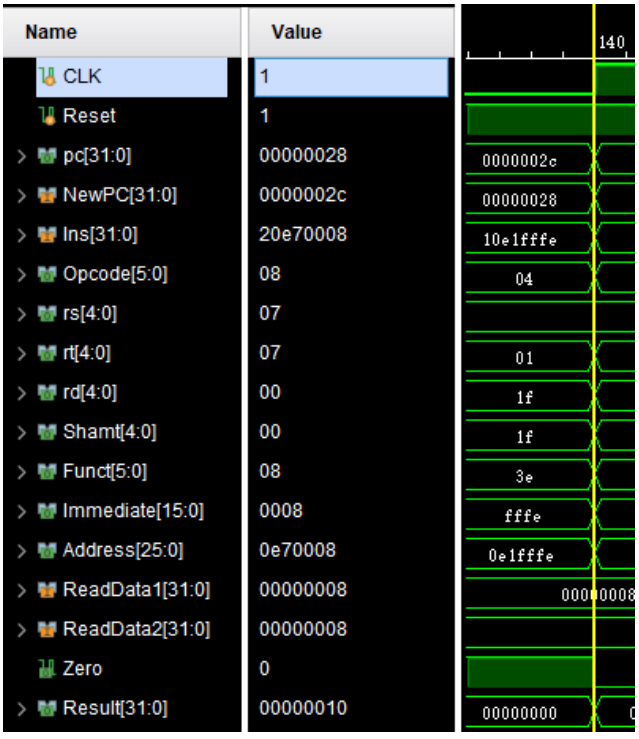
beq $7, $1, -2

可以看到：PC 为：0000002C，Instruction 为：10E1FFFE，表示分支指令，由于 Zero 为 1，故将跳转至 00000028。



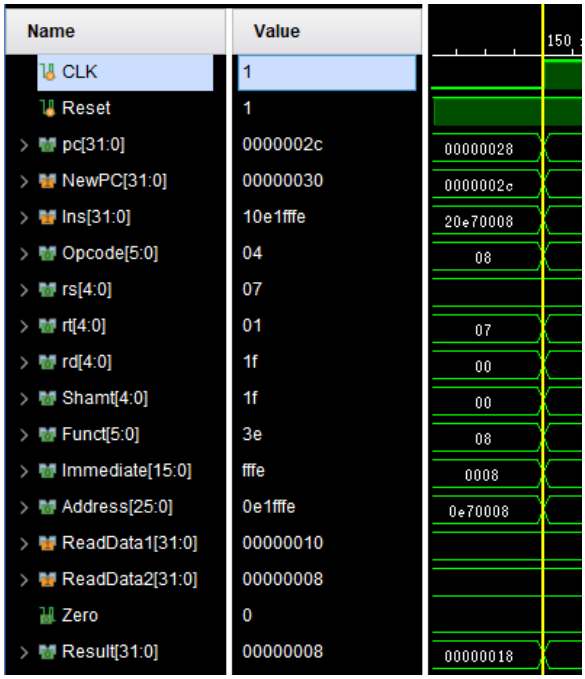
addi $7, $7, 8

可以看到：PC 为：00000028，Instruction 为：20E70008，表示与立即数的加法运算，其中寄存器分别是$7，立即数是 8，所以 ReadData1 是 8，ReadData2 是 8，ALUResult 是 16。



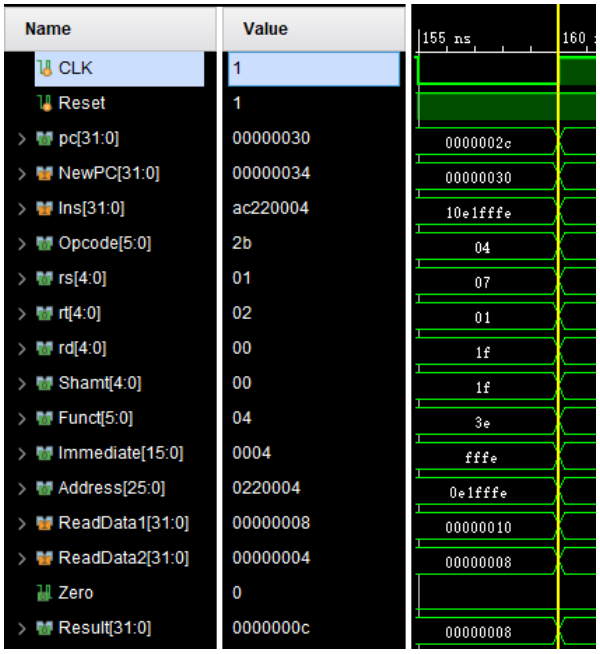
beq $7, $1, -2

可以看到：PC 为：0000002C，Instruction 为：10E1FFFE，表示分支指令，由于 Zero 为 0，故将跳转至 00000030。



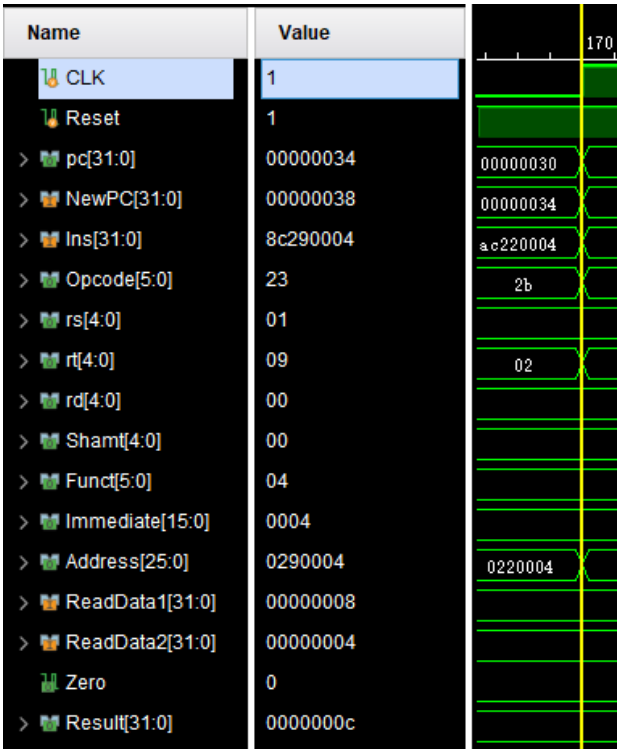
sw $2, 4($1)

可以看到：PC 为：00000030，Instruction 为：AC220004，表示写入寄存器指令，将$2 内容写入至$5。



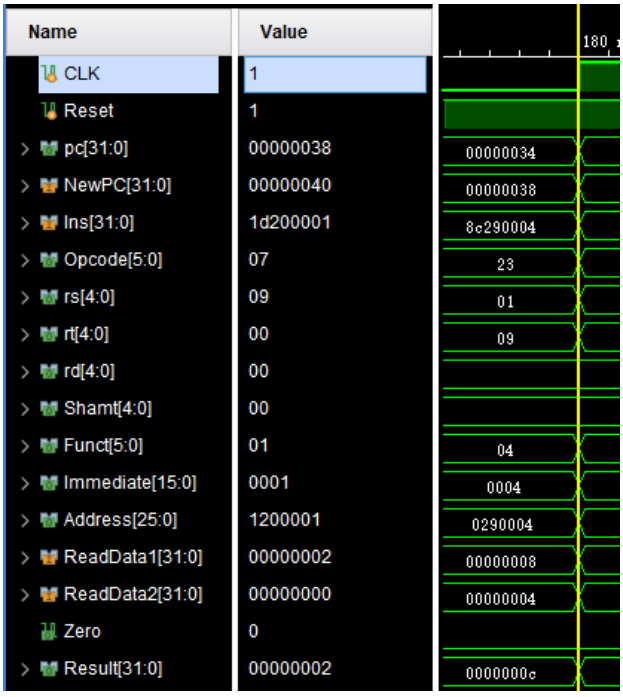
lw $9, 4($1)

可以看到：PC 为：00000034，Instruction 为：8C290004，表示读取寄存器指令，将$5 的内容读取至$9 里面。



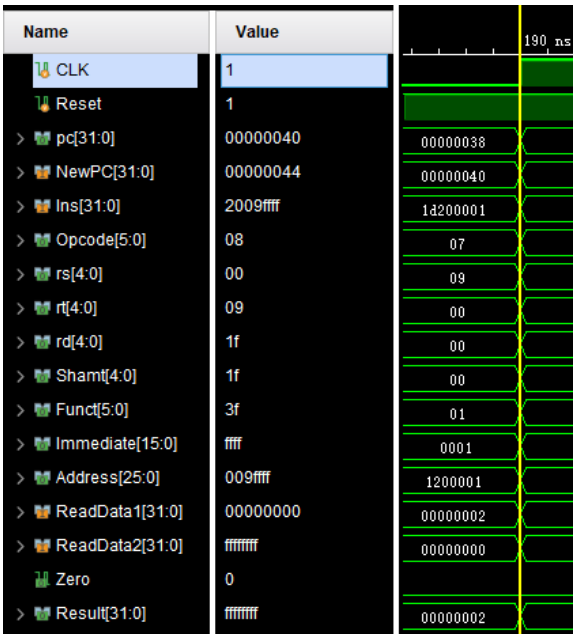
bgtz $9, 1

可以看到：PC 为：00000038，Instruction 为：1D200001，表示分支指令，由于 $9>1，故将跳转至 00000040。



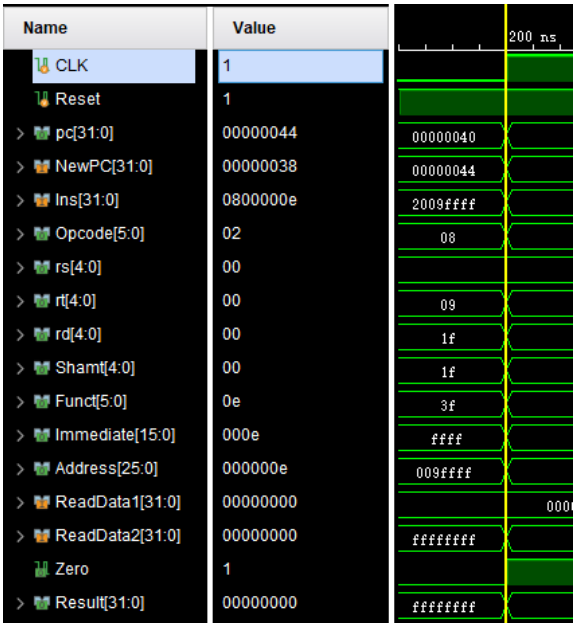
addi $9, $0, -1

可以看到：PC 为：00000040，Instruction 为：2009FFFF，表示与立即数的加法运算，其中寄存器分别是$9，$0，立即数是-1，所以 ReadData1 是 0，ReadData2 是-1，ALUResult 是-1。



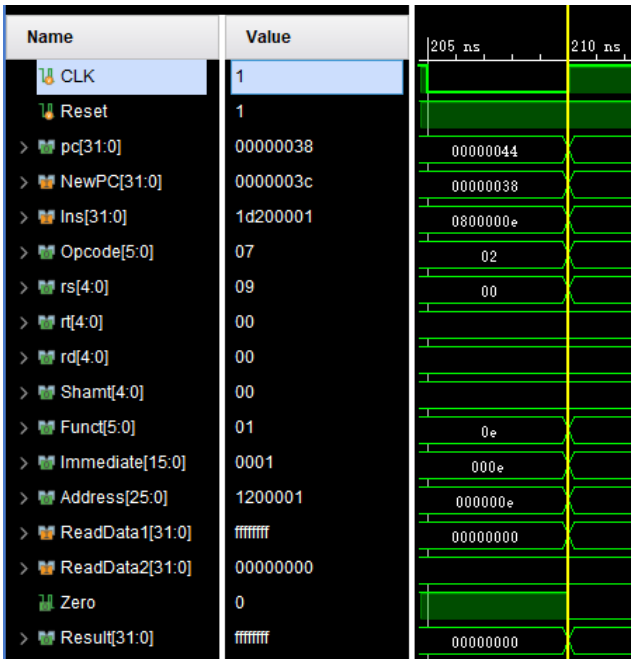
j 0x00000038

可以看到：PC 为：00000044，Instruction 为：0800000E，表示无条件跳转指令，故将跳转至 00000038。



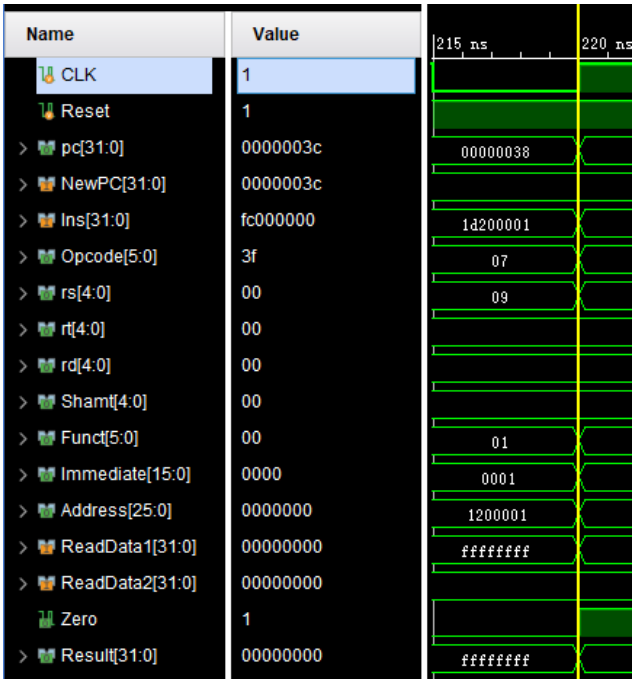
bgtz $9, 4

可以看到：PC 为：00000038，Instruction 为：1D200001，表示分支指令，由于$9<4，故将跳转至 0000003C。



halt

可以看到：PC 为：0000003C，Instruction 为：FC000000，表示停机指令，故程序执行结束。



寄存器内存检查

步骤	$0	$1	$2	$3	$4	$5	$6	$7	$8	$9
1	0	0	0	0	0	0	0	0	0	0
2	0	8	0	0	0	0	0	0	0	0
3	0	8	2	0	0	0	0	0	0	0
4	0	8	2	10	0	0	0	0	0	0
5	0	8	2	10	0	8	0	0	0	0
6	0	8	2	10	0	8	0	0	0	0
7	0	8	2	10	0	8	0	0	2	0
8	0	8	2	10	0	8	0	0	4	0
9	0	8	2	10	0	8	0	0	4	0
10	0	8	2	10	0	8	0	0	8	0
11	0	8	2	10	0	8	0	0	8	0
12	0	8	2	10	0	8	1	0	8	0
13	0	8	2	10	0	8	1	0	8	0
14	0	8	2	10	0	8	1	8	8	0
15	0	8	2	10	0	8	1	8	8	0
16	0	8	2	10	0	8	1	16	8	0
17	0	8	2	10	0	8	1	16	8	0
18	0	8	2	10	0	8	1	16	8	0
19	0	8	2	10	0	8	1	16	8	2
20	0	8	2	10	0	8	1	16	8	2
21	0	8	2	10	0	8	1	16	8	-1
22	0	8	2	10	0	8	1	16	8	-1
23	0	8	2	10	0	8	1	16	8	-1

经检验，上述仿真寄存器结果正确。

实验心得

总体来说，本次设计单周期 CPU 的拓展项目带给我的收获是巨大的，我从中学到了很多东西。

还记得数字电子技术课程刚开课时，老师就在课上介绍了优秀的师兄师姐们做的课外拓展项目，鼓励我们也去尝试做类似的拓展。最初看到师兄们的单周期 CPU 设计的实验报告时，感觉自己离它非常遥远，觉得单周期 CPU 设计是一件难以完成的任务。但为了能让自己得到充分的实践锻炼，我选择开发单周期 CPU 的任务。

虽然在大一上学期《软件工程导论》课上，对计算机组成和 CPU 相关的知识有所了解，但我知道，课上传授的知识只是皮毛，是不足以为开发 CPU 提供技术基础的，在学长的推荐下，我开始阅读《深入理解计算机系统》和《计算机组成原理》。

《深入理解计算机系统》从 C 语言开始讲起，再讲到汇编，再到处理器，还引申了很多底层的知识点，整体来说，讲解的顺序是符合学生的逻辑认知习惯的。《计算机组成原理》是我们大二学期的教材，自我感觉表达比前者更晦涩，但知识更具深度。

大概花了半学期，我把上面提及到的两本书大致看完了，觉得对 CPU 的工作原理的掌握足够用来设计开发单周期 CPU。但有了知识仍不够，我们需要 Verilog 硬件描述语言，将我们的想法变成程序，甚至是硬件。在课堂上，我们对 Verilog 的了解并不是很深，因为数字电子技术实验作业大多都是通过 Block Design 完成的，教师对 Verilog 不作要求。所以在这样的背景下，我选择了自学 Verilog，争取快速地将其掌握。在 GitHub 上，我找到了关于 Verilog 的入门教程，大概花了两天的时间，我对 Verilog 有了深入的理解，在教程要求下，通过很多范例的引导，我能够比较熟练地运用 Verilog。

在完成项目的过程中，我遇到了较多的困难，印象比较深刻的就是如何规划项目架构。因为看似有了概念、知识和工具，但对一个功能完善的系统，我不知道从哪里开始切入。最后在网上几篇教程的指引下，我有了具体设计思路——面向模块的思想，即只需暴露各个子模块的接口，然后在顶层模块中将各接口连接。

在 Vivado 进行仿真时，曾出现一直无法仿真的问题，刚开始以为是代码出了问题，就从头开始排查，但始终找不到错误的原因。后来，在舍友（欧子菁）的帮助下，他告诉我是仿真文件的路径问题造成的。按照他的回答，我将我的仿真文件路径从相对路径调整为绝对路径，结果真的能如期仿真，还是很感谢这位舍友的。

由于开发外包项目的原因，这次项目的战线拉得有些长，完成代码和提笔写报告隔了将

近一个月的时间。又因为临近期末，复习压力大，所以该实验报告略显仓促，还有很多改进的空间。

最后，感谢老师为我们提供宝贵的机会和鼓励，让我体会到了系统设计的重要性：系统设计对个人知识网络的建立有极大的帮助。虽然我是软件工程的学生，未来可能不做与硬件直接相关的工作，但如果能从硬件层次理解计算机运行的过程和原理，是对软件开发有极大的帮助的。我们学到的数字电子技术知识属于硬件知识，它不能是孤立的，而应该被相互联系的。系统设计，就提供这样的过程，让我们把知识能够联系起来。在日后的学习中，我一定会挤出时间去寻找并完成类似的系统设计，努力成为优秀的软件工程人才。