Lab8 单周期 CPU 设计

在冯诺依曼体系架构中，CPU 可以分成数据通路和控制器两大基本组成部分。指令执行过程中数据所经过的路径，包括路径上的部件称为数据通路。控制器则根据指令的操作码和功能码生成数据通路中所需的控制信号。

CPU 的基本职能是周而复始地执行指令，指令按顺序存放在内存连续单元中，将要执行的指令的地址由 PC 给出。通常，一条指令执行过程的大致如下。

（1）取指令。从 PC 指出的内存单元中取出指令送到指令寄存器（IR）。

（2）对 IR 中的指令操作码译码并计算下条指令地址，并根据指令操作码和功能码生成操作控制信号。

（3）计算源操作数地址并获取源操作数。根据寻址方式计算源操作数地址，根据若源操作数是存储器数据还是寄存器数据，分别进行不同的读取操作。

（4）对操作数进行相应的运算。在 ALU 或加法器等运算部件中对取出的操作数进行运算。

（5）计算目的操作数地址并存储结果。根据寻址方式计算目的操作数的地址，将运算结果存入存储单元中，或存入通用寄存器中。

指令执行过程中的每个操作步骤都有先后顺序，为了使计算机能正确执行指令，CPU 必须按正确的时序产生操作控制信号。

单周期 CPU 中每条指令的执行时间都是 1 个时钟周期，按照执行时间最长的指令来设置时钟周期。

一、实验目的

1. 掌握RV32I 单周期CPU的设计方法。
2. 掌握单周期CPU的仿真测试程序的设计方法。
3. 掌握单周期CPU官方测试集的测试方法。

二、实验环境

1. Vivado 开发环境
2. Xilinx A7-100T 实验板
3. RARS 汇编和 RISC-V 实时模拟器
4. Ripes 汇编编辑器和 RISC-V 模拟器

三、实验原理

CPU 设计涉及数据通路和控制部件的设计，设计过程如下。

第 1 步：分析每条指令的功能。

第 2 步：根据指令的功能给出所需的基本功能部件，并将它们互连，以实现每条指令的数据通路。

第 3 步：确定每个基本功能部件所需控制信号的取值。

第 4 步：汇总所有指令涉及的控制信号，生成反映指令操作码与控制信号之间关系的真值表。

本次实验以 RISC-V 指令系统中的基础指令集 RV32I 为例来说明具体的设计过程。

RISC-V 非压缩方式下的六种指令格式如如 8.1 所示。

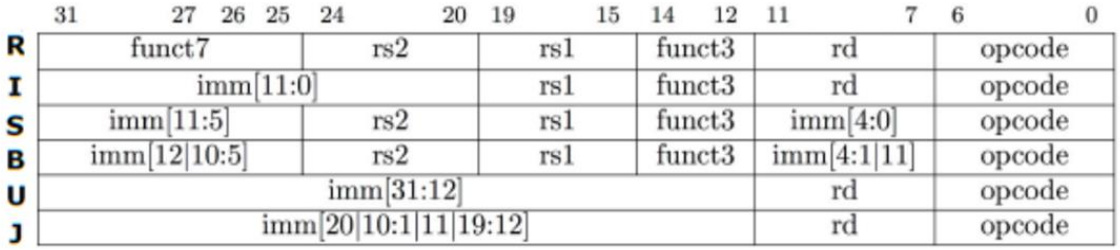

图 8.1　RISC-V 指令格式

实验选择了除了系统控制指令以外的 37 条 RV32I 指令作为实现目标，涵盖了整数运算、存储器访问、分支转移等几个大类，包括运算指令（19 条）、分支控制指令（8 条）、访存指令（8 条）和其他指令（2 条）。

设计处理器的第一步是确认每条指令的功能，表 8.1 给出了 37 条目标指令功能描述。因为每条指令的第一步都是取指令并 PC 加 4，使 PC 指向下条指令，都省略了对这一步的描述。

表 8.1　37 条目标指令列表

分类	指令	名称	功能	类型
运算指令	add　rd, rs1, rs2	加法	R[rd]←R[rs1] + R[rs2]	R
	sub　rd, rs1, rs2	减法	R[rd]←R[rs1] - R[rs2]	R
	and　rd,rs1, rs2	与	R[rd]←R[rs1] & R[rs2]	R
	or　rd,rs1,rs2	或	R[rd]←R[rs1] | R[rs2]	R
	xor　rd,rs1,rs2	异或	R[rd]←R[rs1] ^ R[rs2]	R
	sll　rd,rs1,rs2	逻辑左移	R[rd]←R[rs1] << R[rs2]	R
	srl　rd,rs1,rs2	逻辑右移	R[rd]←R[rs1] >> R[rs2](Zero-extend)	R
	sra　rd,rs1, rs2	算术右移	R[rd]←R[rs1] >> R[rs2] (Sign-extend)	R
	slt　rd, rs1, rs2	小于则置位	R[rd]←(R[rs1] < R[rs2])？1:0	R
	sltu　rd,rs1 rs2	无符号小于则置位		R
	addi　rd,rs, imm12	立即数加法	R[rd]←R[rs1] + SEXT(imm12)	I
	andi　rd,rs1 imm12	立即数取与	R[rd]←R[rs1] & SEXT(imm12)	I

	ori rd, rs1, imm12	立即数取或	R[rd]←R[rs1] | SEXT(imm12)	I
	xori rd,rs1, imm12	立即数取异或	R[rd]←R[rs1] ^ SEXT(imm12)	I
	slli rd,rs1, imm12	立即数逻辑左移	R[rd]←R[rs1] << SEXT(imm12)	I*
	srli rd,rs1, imm12	立即数逻辑右移	R[rd]←R[rs1] >> SEXT(imm12) (Zero-extend)	I*
	srai rd,rs1,imm12	立即数算术右移	R[rd]←R[rs1] >> SEXT(imm12) (Sign-extend)	I*
	slti rd,rs1, imm12	小于立即数则置位	R[rd]←(R[rs1] < SEXT(imm12)) ？ 1:0	I
	sltiu rd,rs1, imm12	无符号小于立即数则置位		I
访存指令	lb rd, imm12(rs1)	字节加载	R[rd]←SEXT(M[R[rs1] + SEXT(imm12)][7:0])	I
	lbu rd, imm12(rs1)	无符号字节加载	R[rd]←M[R[rs1] + SEXT(imm12)][7:0]	I
	lh rd, imm12(rs1)	半字加载	R[rd]←SEXT(M[R[rs1] + SEXT(imm12)][15:0])	I
	lhu rd, imm12(rs1)	无符号半字加载	R[rd]←M[R[rs1] + SEXT(imm12)][15:0]	I
	lw rd, imm12(rs1)	字加载	R[rd]←M[R[rs1] + SEXT(imm12)]	I
	sb rs2, imm12(rs1)	存字节	M[R[rs1] + SEXT(imm12)]←R[rs2][7:0]	S
	sh rs2, imm12(rs1)	存半字	M[R[rs1] + SEXT(imm12)]←R[rs2][15:0]	S
	sw rs2, imm12(rs1)	存字	M[R[rs1] + SEXT(imm12)]←R[rs2]	S
分支控制指令	beq rs1,rs2, offset	相等时分支跳转	if (R[rs1] == R[rs2]) PC ← PC + SEXT (offset)	B
	bge rs1,rs2,offset	大于等于时分支	if (R[rs1] ≥ R[rs2]) PC ← PC + SEXT (offset)	B
	bgeu rs1,rs2,offset	无符号大于等于时分支		B
	blt rs1,rs2 ,offset	小于时分支	if (R[rs1] < R[rs2]) PC ← PC + SEXT (offset)	B
	bltu rs1,rs2 ,offset	无符号小于时分支		B
	bne rs1,rs2,offset	不等时分支跳转	if (R[rs1] ≠ R[rs2]) PC ← PC + SEXT (offset)	B
	jal rd, offset	跳转并链接	R[rd]←PC + 4 PC←PC + SEXT(offset)	J
	jalr rd,rs1, offset	跳转并寄存器链接	R[rd]←PC + 4 PC←R[rs1] + SEXT(offset)	I
其他指令	auipc rd, imm20	PC加立即数	R[rd]← PC+(imm <<12)	U
	lui rd, imm20	高位立即数加载	R[rd]←imm20 << 12	U

不同类型指令的功能及数据传输通道分析如下：

R-型指令是对两个寄存器 Rs1 和 Rs2 内容的运算并将结果写入 Rd 寄存器。数据传输通道涉及寄存器堆和 ALU。

I-型指令带立即数的运算类指令都涉及对 12 位立即数进行符号位扩展，然后和 Rs1 的内容进行运算，最终把 ALU 的运算结果送目的寄存器 Rd。数据传输通道增加了立即数扩展器和操作数 B 的选择器。要注意的是 3 条立即数移位指令（I*），需要特别处理移位位数的范围问题。读取存储器指令也属于 I-型指令，寻址方式采用寄存器间接寻址，通过立即数和和 Rs1 的内容相加得到。

S-型指令数据寻址方式采用寄存器间接寻址，通过立即数和寄存器 Rs1 的内容相加，得到访存地址。存取数据的长度可以是 1 个字节、半字或 1 个字。

B-型指令根据不同的大小关系比较结果进行控制转移，通过在 ALU 中执行减法操作 sub 来判别两个操作数的大小关系。在数据通路中主要增加了目标转移地址的计算，操作数 B 的来源可选择常量 4 或立即数。

J-型指令是无条件转移指令，jal 指令中给出了 20 位立即数表示的偏移量，其功能是将 PC+4 的值写入到寄存器 Rd 中，然后通过 PC 加上符号位扩展的偏移量来计算跳转目标地址，并写入到 PC 中。Jalr 指令跳转目标地址通过读取 Rs1 寄存器的数值加上符号位扩展的偏移量来计算，并将原 PC+4 的值写入寄存器 Rd 中。在数据通道中涉及操作数 A 的来源选择器和操作数 B 选择器中数据输入端增加了常量 4 的输入。在下地址逻辑中实现计算目标地址时，操作数 A 的来源可选择为 PC 和寄存器 Rs1。

U-型指令中指令 lui 是对 20 位立即数进行低位补零，结果送目的寄存器 Rd。指令 auipc 是对 20 位立即数进行低位补零，并与当前指令地址寄存器 PC 相加，运算结果送目的寄存器 Rd。涉及到获取当前指令地址寄存器的输入。

在上述指令功能实现时，有几个需要特别注意的地方，如移位指令中，移位长度限定。目标转移地址的末位地址等。

根据指令功能和数据通路的分析，可以将 CPU 内部结构划分成不同模块。这些不同模块构成单周期 CPU 的数据通路，在控制信号的操纵下按时序完成指令功能，其原理图如 8.2 所示的。图中所有加下划线的都是控制信号名，控制信号线用虚线表示。指令执行结果总是在下个时钟到来时开始保存在寄存器、数据存储器或 PC 中。

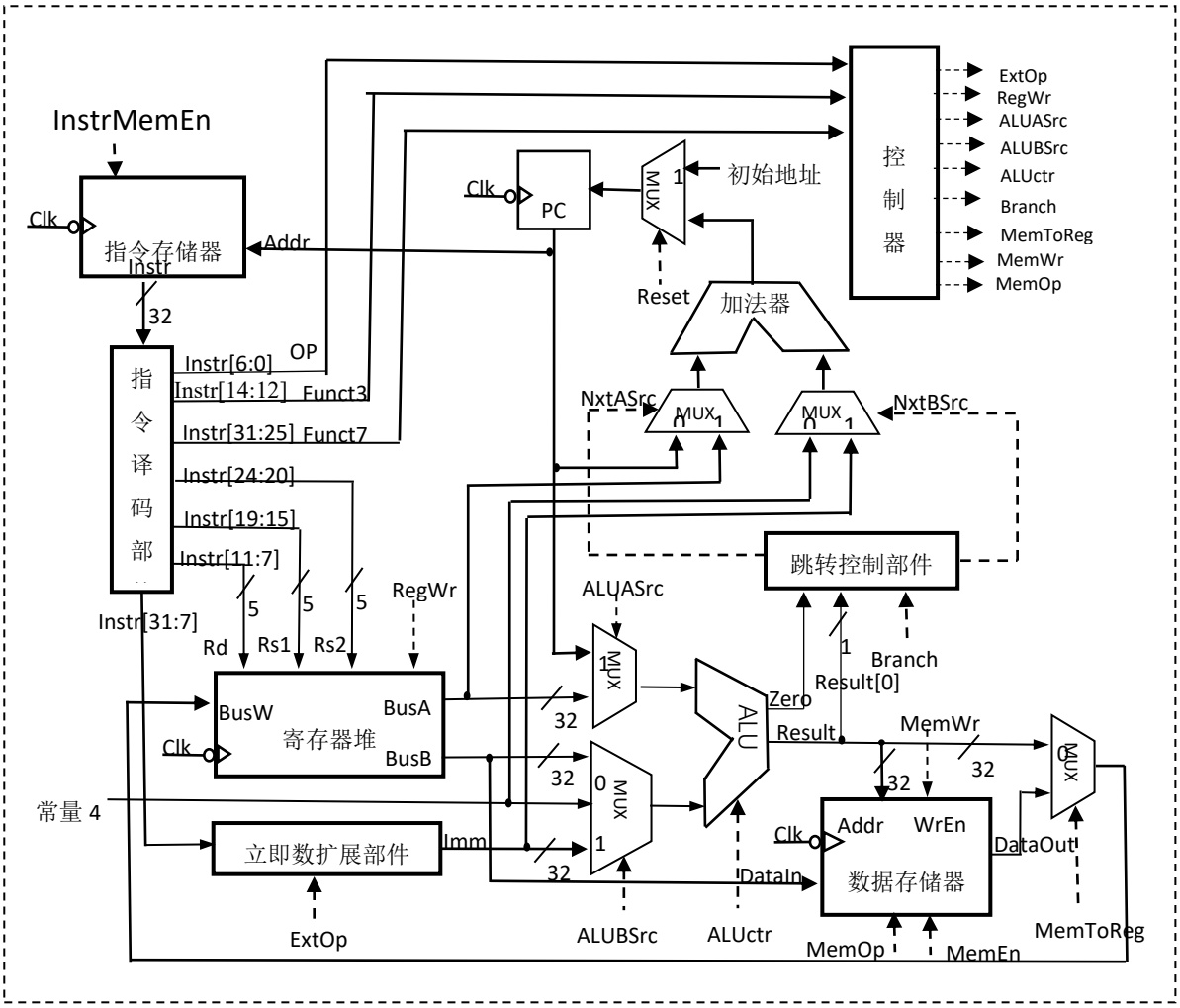

图 8.2 单周期 CPU 原理图

1、数据通路设计

数据通路主要由取指令部件、指令存储器、取操作数、算术逻辑运算单元 ALU、跳转控制部件、数据存储器等组成。

1）、取指令部件

程序运行的第一步就是取指令，取指令部件就是处理器将指令从指令存储器由程序计数器 PC 值指定地址中读取出来的过程。初始时（系统复位或刚启动）32 位的 PC 寄存器中保存着当前程序在指令存储器中起始指令的物理地址。开始执行程序后，一方面把地址送到指令存储器的地址端输出指令内容，另一方面通过下地址逻辑来计算机下条指令的地址，然后送回 PC 寄存器。在单周期处理器中，每个时钟周期执行一条指令，所以每来一个时钟信号 Clk，PC 寄存器的值都会被更新一次，因而，PC 寄存器无须“写使

能”信号控制。注意 PC 寄存器是下降沿触发，取指令部件示意图如图 8.3 所示，包括 PC 寄存器、下地址逻辑和指令存储器等 3 个部分。

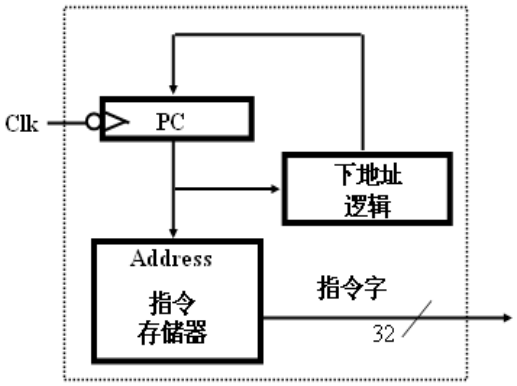

图 8.3. 取指令部件示意图

PC 寄存器的主要功能室输出当前指令地址。当复位信号有效时输出程序代码的初始地址，当复位信号无效时，在单周期 CPU 设计中 PC 寄存器的写使能信号始终有效，当时钟边沿信号有效时写入下条地址的 PC 值。实际设计使用 32 位寄存器来实现。

指令存储器专门用来存放指令。在单周期 CPU 设计中，由于需要在一个时钟周期内完成所有的指令操作，因此指令存储器需要采用**异步读**的方式来实现，实验中的指令存储器采用 FPGA 中的**分布式** RAM 来实现。

指令存储器主要接口包括当前指令机器代码，以及时钟信号、当前指令地址线 PC、片选信号等输入信号。在 RV32I 中，每条指令的长度都是 4 个字节，因此所有代码地址都是 4 字节对齐，即 PC 最低两位总是 0。

假设指令存储器的空间时为 64kB（16K 条指令），指令存储器的存储单元设置为 32 位，时钟上升沿时输出读取指令内容，则指令存储器模块 InstrMem 的参考代码如下：

```
module InstrMem(
    output reg [31:0] instr,     //输出 32 位指令
    input [31:0] addr,           //地址字长 32 位，实际有效字长根据指令存储器容量来确定
    input InstrMemEn,            //指令存储器片选信号
    input clk                    //时钟信号，下降沿有效
);
    (* ram_style="distributed" *) reg [31:0] ram[16384:0];//64KB 的存储器空间，可存储 16k 条指令，地址有效长度 16 位
    always @ (posedge clk) begin
        if (InstrMemEn) instr = ram[addr[15:2]];
    end
endmodule
```

在程序执行过程中，下一条指令地址的计算有多种情形：

1、顺序执行指令，则 PC=PC+4。

2、无条件跳转指令，jal 指令，PC = PC + 立即数 imm；jalr 指令，PC = R[rs1] + 立即数 imm。

3、分支转移指令，根据比较运算的结果和 Zero 标志位来判断，条件成立则 PC = PC +立即数 imm，否则 PC=PC+4。

下地址模块主要接口是 PC 输入、立即数输入，寄存器 A 口输入、跳转控制信号，输出下一条指令地址。根据图 8.2 设计的下地址模块 nextPC 的参考代码如下：

```
module nextPC(
    output [31:0] nxtPC,        //下一个取指令地址，32 位，取低 16 位
    input [31:0] BusA,          //BusA
    input [31:0] curPC,Imm,     //PC 值、立即数
    input NxtASrc, NxtBSrc      //选择信号，由分支控制部件产生
    );
    wire [31:0] NxtA, NxtB;
    assign NxtA = NxtASrc ? BusA&32'hfffffffe:curPC;
    assign NxtB = NxtBSrc ? Imm&32'hfffffffe:32'd4;
    assign nxtPC=NxtA+NxtB;endmodule
```

2）、取操作数部件

取操作数部件包括指令解析、立即数扩展和寄存器堆等几个部分。

从指令存储器中读取指令后，需要根据图 8.1 所示的 RISC-V 指令格式进行解析，分解出指令字段。opcode 为操作码字段，funct3 和 funct7 为功能码字段，imm 为立即数字段，rs1 和 rs2 为源操作数寄存器编号，rd 为目的寄存器编号。则指令解析模块 InstrParse 的参考代码如下：

```
module InstrParse(
            output [6:0] opcode,      //指令编码 7 位
            output [4:0] rd,          //目的寄存器编号 5 位
            output [2:0] funct3,      //3 位功能码
            output [4:0] rs1,         //源寄存器 1 编号 5 位
            output [4:0] rs2,         //源寄存器 2 编号 5 位
            output [7:0] funct7,      //7 位功能码
            input [31:0] instr        //指令
 );
    assign opcode = instr[6:0];
    assign rd = instr[11:7];
    assign funct3 = instr[14:12];
    assign rs1 = instr[19:15];
    assign rs2 = instr[24:20];
    assign funct7 = instr[31:25];
endmodule
```

立即数扩展器将根据图 8.1 中给出的不同指令格式对指令中的立即数编码进行拼接和扩展，生成 32 位操作数。有 5 种类型指令中包含立即数，因此对应的立即数扩展控制码 ExtOp 需要 3 位。使用一个多路选择器对应的 5 种立即数扩展操作：immI、immU、immS、immB、immJ。

根据图 8.2 设计的立即数扩展器模块 InstrToImm 的参考代码设计如下：

```
module InstrToImm (
    input [31:0] instr,     //32 位指令
    input [2:0] ExtOp,      //扩展控制码
    output reg [31:0] imm
 );
    wire [31:0] immI, immU, immS, immB, immJ;
    assign immI = {{20{instr[31]}}, instr[31:20]};
    assign immU = {instr[31:12], 12'b0};
    assign immS = {{20{instr[31]}}, instr[31:25], instr[11:7]};
    assign immB = {{20{instr[31]}}, instr[7], instr[30:25], instr[11:8], 1'b0};
    assign immJ = {{12{instr[31]}}, instr[19:12], instr[20], instr[30:21], 1'b0};

    always@(*)
    begin
        case(ExtOp)
            3'b000:
                imm = immI;
            3'b001:
                imm = immU;
            3'b010:
                imm = immS;
            3'b011:
                imm = immB;
            3'b100:
                imm = immJ;
            default:
                imm = immI;
        endcase
    end
endmodule
```

寄存器堆的设计已经在前面的实验中完成，需要注意的是，在 RISC-V 体系架构中，寄存器 0 的值始终为 0，可能需要修改原先的设计程序。

数据通路中算术逻辑单元和数据存储器在前面的实验中已经设计完成。在本实验中需要根据表 8.2 所示的控制信号实际赋值进行修改。

2、控制器设计

控制器根据指令代码中的操作码 opcode、功能码 funct3 和功能码 funct7 来生成对应的控制信号。控制信号引导具体的指令在数据通路上完成相应的操作，实现指令功能。需要数据通路中的分析，需要生成的控制信号主要有包括：

ExtOp：控制立即数扩展器的输出数据，宽度为 3 位。

RegWr：寄存器堆写使能信号，高电平有效。

ALUASsrc：选择 ALU 输入端 A 的来源。

ALUBSrc：选择 ALU 输入端 B 的来源，宽度为 2 位。

ALUctr：选择 ALU 执行的操作，宽度为 4 位。

Branch：用于生成分支控制信号，宽度为 3 位。

MemtoReg：选择寄存器 rd 写回数据来源，ALU 输出还是数据存储器输出。

MemWr：数据存储器写使能信号。

MemOp：用于数据存储器读写数据字节长度（字节、半字、字）以及在读取时是否进行符号位扩展，宽度为 3 位。

本次实验涉及指令控制信号具体定义如表 8.2 所示，需要注意的是，这只是一种编码方案，可以自行设计其他的编码方案。

表 8.2 RV32I 指令控制信号列表

指令	类型	op[6:0]	funct3	funct7[5]	ExtOp	RegWr	ALUASrc	ALUBSrc	ALUctr
lui	U	0110111	×	×	001	1	×	10	1111
auipc	U	0010111	×	×	001	1	1	10	0000
addi	I	0010011	000	×	000	1	0	10	0000
slti	I	0010011	010	×	000	1	0	10	0010
sltiu	I	0010011	011	×	000	1	0	10	0011
xori	I	0010011	100	×	000	1	0	10	0100
ori	I	0010011	110	×	000	1	0	10	0110
andi	I	0010011	111	×	000	1	0	10	0111
slli	I	0010011	001	0	000	1	0	10	0001
srli	I	0010011	101	0	000	1	0	10	0101
srai	I	0010011	101	1	000	1	0	10	1101
add	R	0110011	000	0	×	1	0	00	0000
sub	R	0110011	000	1	×	1	0	00	1000
sll	R	0110011	001	0	×	1	0	00	0001
slt	R	0110011	010	0	×	1	0	00	0010
sltu	R	0110011	011	0	×	1	0	00	0011
xor	R	0110011	100	0	×	1	0	00	0100

srl	R	0110011	101	0	×	1	0	00	0101
sra	R	0110011	101	1	×	1	0	00	1101
or	R	0110011	110	0	×	1	0	00	0110
and	R	0110011	111	0	×	1	0	00	0111
jal	J	1101111	×	×	100	1	1	01	0000
jalr	I	1100111	000	×	000	1	1	01	0000
beq	B	1100011	000	×	011	0	0	00	0010
bne	B	1100011	001	×	011	0	0	00	0010
blt	B	1100011	100	×	011	0	0	00	0010
bge	B	1100011	101	×	011	0	0	00	0010
bltu	B	1100011	110	×	011	0	0	00	0011
bgeu	B	1100011	111	×	011	0	0	00	0011
lb	I	0000011	000	×	000	1	0	10	0000
lh	I	0000011	001	×	000	1	0	10	0000
lw	I	0000011	010	×	000	1	0	10	0000
lbu	I	0000011	100	×	000	1	0	10	0000
lhu	I	0000011	101	×	000	1	0	10	0000
sb	S	0100011	000	×	010	0	0	10	0000
sh	S	0100011	001	×	010	0	0	10	0000
sw	S	0100011	010	×	010	0	0	10	0000

表 8.2 RV32I 指令控制信号列表（续）

指令	类型	op[6:0]	funct3	funct7[5]	Branch	MemtoReg	MemWr	MemOp
lui	U	0110111	×	×	000	0	0	×
auipc	U	0010111	×	×	000	0	0	×
addi	I	0010011	000	×	000	0	0	×
slti	I	0010011	010	×	000	0	0	×
sltiu	I	0010011	011	×	000	0	0	×
xori	I	0010011	100	×	000	0	0	×
ori	I	0010011	110	×	000	0	0	×
andi	I	0010011	111	×	000	0	0	×
slli	I	0010011	001	0	000	0	0	×
srli	I	0010011	101	0	000	0	0	×
srai	I	0010011	101	1	000	0	0	×
add	R	0110011	000	0	000	0	0	×
sub	R	0110011	000	1	000	0	0	×
sll	R	0110011	001	0	000	0	0	×
slt	R	0110011	010	0	000	0	0	×
sltu	R	0110011	011	0	000	0	0	×

xor	R	0110011	100	0	000	0	0	×
srl	R	0110011	101	0	000	0	0	×
sra	R	0110011	101	1	000	0	0	×
or	R	0110011	110	0	000	0	0	×
and	R	0110011	111	0	000	0	0	×
jal	J	1101111	×	×	001	0	0	×
jalr	I	1100111	000	×	010	0	0	×
beq	B	1100011	000	×	100	×	0	×
bne	B	1100011	001	×	101	×	0	×
blt	B	1100011	100	×	110	×	0	×
bge	B	1100011	101	×	111	×	0	×
bltu	B	1100011	110	×	110	×	0	×
bgeu	B	1100011	111	×	111	×	0	×
lb	I	0000011	000	×	000	1	0	000
lh	I	0000011	001	×	000	1	0	001
lw	I	0000011	010	×	000	1	0	010
lbu	I	0000011	100	×	000	1	0	100
lhu	I	0000011	101	×	000	1	0	101
sb	S	0100011	000	×	000	×	1	000
sh	S	0100011	001	×	000	×	1	001
sw	S	0100011	010	×	000	×	1	010

根据指令的操作码、功能码来设计每一个控制信号的逻辑表达式，生成 CPU 控制器。通过分析 RV32I 指令编码可以发现，相同类型指令的操作码基本相同，通过功能码来区分，因此可以把相同操作码用某个标志位来表示。在 7 位功能码字段中，只有少量的第 5 位 func7[5]为 1，与其他 7 位功能码不同。大部分控制信号可以根据指令类型直接赋值，部分控制信号需要根据 3 位功能码来赋值，只有极少数控制信号需要加上 7 位功能码的第 5 位来表示。

根据指令的操作码、功能码来设计控制信号的逻辑表达式，生成单周期 CPU 的控制器。 控制器的设计电路原理图如图 8.4 所示。

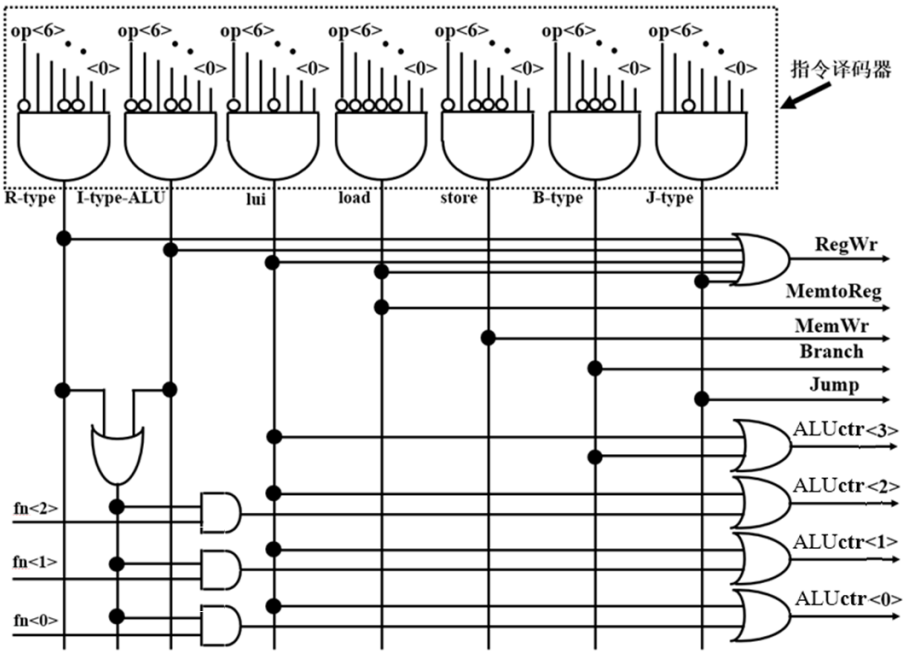

图 8.4 控制器电路原题图

根据表 8.2 的定义，控制器模块 Control 的端口定义如下。

```
module Control(
    output reg [2:0] ExtOp,
    output reg RegWr,
    output reg ALUASrc,
    output reg [1:0] ALUBSrc,
    output reg [3:0] ALUctr,
    output reg [2:0] Branch,
    output reg MemtoReg,
    output reg MemWr,
    output reg [2:0] MemOp,
    input [6:0] opcode,
    input [2:0] funct3,
    input [6:0] funct7
  );
 //add your code here
endmodule
```

根据表 8.2 和图 8.4 所示的元流通，完成控制器模块代码的设计。

跳转控制模块生成下地址逻辑中加法器两个操作数的选择信号 NxtASrc 和 NxtBSrc，根据控制信号 Branch 和 ALU 零标志位 Zero 及小于标志位 Result[0]信号来决定，具体定义如表 8.3 所示。

表 8.3 Branch 控制信号含义

Branch	NxtASrc	NxtBSrc	指令跳转类型
000	0	0	非跳转指令

001	0	1	jal：无条件跳转 PC 目标
010	1	1	jalr：无条件跳转寄存器目标
100	0	Zero	beq：条件分支，等于
101	0	! Zero	bne：条件分支，不等于
110	0	Result[0]	blt,bltu：条件分支，小于
111	0	! Result[0]	bge,bgeu：条件分支，大于等于

根据表 8.3 所示，跳转控制模块 BranchControl 的参考代码设计如下：

```
module BranchControl(
    output reg NxtASrc, NxtBSrc,
    input zero, result0,
    input [2:0] Branch
    );
    always @ (*) begin
        case (Branch)
            3'b000: begin NxtASrc = 1'b0; NxtBSrc = 1'b0; end    //非跳转指令
            3'b001: begin NxtASrc = 1'b0; NxtBSrc = 1'b1; end    //jal
            3'b010: begin NxtASrc = 1'b1; NxtBSrc = 1'b1; end    //jalr
            3'b100: begin NxtASrc = 1'b0; NxtBSrc = (zero===1'bx)?1'b1:zero; end    //beq
            3'b101: begin NxtASrc = 1'b0; NxtBSrc = (zero===1'bx)?1'b1:~zero; end    //bne
            3'b110: begin NxtASrc = 1'b0; NxtBSrc = (result0===1'bx)?1'b1:result0; end    //blt, bltu
            3'b111: begin NxtASrc = 1'b0; NxtBSrc = (result0===1'bx)?1'b1:~result0; end    //bge, bgeu
            default: begin NxtASrc = 1'b0; NxtBSrc = 1'b0; end
        endcase
    end
 endmodule
```

至此已经完成单周期 CPU 设计的所有模块，需要将单独开发各个模块并进行单元测试，最后整合成系统。当然模块划分不是唯一的，可以根据自行进行调整。

四、实验内容

1、单周期 CPU 设计

为了保证 CPU 每次重启时，都能从相同的初始状态开始执行程序，设计一个复位信号 Reset，用来初始化 CPU 中各个部件的控制信号和设置 PC 寄存器的起始地址。

为了在后续的指令测试集验证和计算机系统实验中简化外设对存储器的访问，本次实验中将指令存储器和数据存储器作为独立模块，与 CPU 设计分离，放在顶层模块中引用。

由于指令存储器和数据存储器没有包含在 CPU 模块中，因此在 CPU 模块中需要提供指令存储器和数据存储器的输入输出端口信号以及时钟信号 clk 和复位信号 Reset。

假设单周期 CPU 模块 SingleCycleCPU 的端口定义如下

```
module SingleCycleCPU(
    input               clock,
    input               reset,
    output [31:0] InstrMemaddr,         //指令存储器地址
    input   [31:0] InstrMemdataout,     //指令内容
    output          InstrMemclk,         // 指令存储器读取时钟，为了实现异步读取，设置读取时钟和写入时钟反相
    output [31:0] DataMemaddr,          //数据存储器地址
    input   [31:0] DataMemdataout,      //数据存储器输出数据
    output [31:0] DataMemdatain,        //数据存储器写入数据
    output          DataMemrdclk,       //数据存储器读取时钟，为了实现异步读取，设置读取时钟和写入时钟反相
    output          DataMemwrclk,       //数据存储器写入时钟
    output [2:0]    DataMemop,          //数据读写字节数控制信号
    output          DataMemwe,          //数据存储器写入使能信号
    output [15:0] dbgdata               //debug 调试信号，输出 16 位指令存储器地址有效地址
););

//add your code here
endmodule
```

在前期的实验中已经完成模块添加中代码中，完成 CPU 模块代码的设计。

2、单步功能仿真

RISC-V CPU 是一个较为复杂的数字系统，在实验过程中需要对每一个环节进行详细的测试才能够保证系统整体的可靠性。

首先确保每一个子模块都正常工作，因此在完成各个模块的代码编写后需要进行对应的测试。具体可以包括：

代码复查：检查代码编写过程中是否有问题，尤其是变量名称、数据线宽度等易出错的地方。检查编译中的警告，判断是否警告会带来错误。

RTL 复查：利用 RTL Viewer 检查系统编译输出的 RTL 是否符合设计构想，有没有悬空或未连接的引脚。

仿真测试：通过针对独立模块的 testbench 进行功能仿真，尤其需要注意 ALU、寄存器堆、及内容的功能正确性。对于存储器件需要分析时序正确性，即数据是否在正确的时间读取，写入时是否按预期写入等。

在完成基本单元测试后，可以进行 CPU 整体的联调。整体联调的主要目的是验证各个指令基本功能的正确性。本次实验提供了仿真测试模块 SingleCycleCPU_tb 来进行单步指令的执行和验证。

在模块中，首先定义了一部分测试中需要用到的变量：

```
module SingleCycleCPU_tb ();
integer numcycles;        //number of cycles in test
reg clk,reset;            //clk and reset signals
reg[8*30:1] testcase;     //name of testcase，测试用例名，为字符串格式，用来载入不同的测试用例。
```

然后实例化 CPU 中的部件，包括 CPU 模块、指令存储器模块和数据存储器模块：

```
// CPU declaration
// signals
wire [31:0] iaddr,idataout;
wire iclk;
wire [31:0] daddr,ddataout,ddatain;
wire drdclk, dwrclk, dwe;
wire [2:0]    dop;
wire [23:0] cpudbgdata;
//main CPU
SingleCycleCPU    mycpu(.clock(clk),
                        .reset(reset),
                        .InstrMemaddr(iaddr), .InstrMemdataout(idataout), .InstrMemclk(iclk),
                        .DataMemaddr(daddr), .DataMemdataout(ddataout), .DataMemdatain(ddatain),
                        .DataMemrdclk(drdclk), DataMemwrclk(dwrclk), .DataMemop(dop),
                         .DataMemwe(dwe),    .dbgdata(cpudbgdata));

//instruction memory, no writing
InstrMem myinstrmem(.instr(idataout),.addr(iaddr),.InstrMemEn(1'b1),.clk(iclk));

//data memory
 DataMem mydatamem(
                          .addr(daddr), .dataout(ddataout),    .datain(ddatain),
                          .rdclk(drdclk), .wrclk(dwrclk), .memop(dop), .we(dwe));
```

在实验过程中可以根据设计的模块以及接口定义自行进行更改。

在测试过程中，如果设计的存储器模块有 BUG，可以使用仿真测试模块中定义了一系列的辅助任务 task，帮助完成各类测试操作：

```
//useful tasks
task step;       //step for one cycle ends 1ns AFTER the negative edge of the next cycle
    begin
        #9    clk=1'b1;
        #10    clk=1'b0;
```

```
            numcycles = numcycles + 1;
            #1 ;
        end
    endtask

    task stepn;             //step n cycles
        input integer n;
        integer i;
        begin
            for (i =0; i<n ; i=i+1)    step();
        end
    endtask

    task resetcpu;      //reset the CPU and the test
        begin
            reset = 1'b1;
            step();
            #5 reset = 1'b0;
            numcycles = 0;
        end
    endtask
```

step 任务是将 CPU 时钟前进一个周期，在单周期 CPU 中等价于单步执行一条指令。注意这里的周期是以下降沿开始的，在实际测试中可以将时间步进到下一个周期的下降沿后一个时间单位，这主要是由于单周期 CPU 是在下一上升沿进行写入，对数据的验证要在上升沿略后一些的时间进行。stepn 任务用于执行 n 条指令，resetcpu 用于将 cpu 重置，从预定开始执行的地址重新执行。

Testbench 中定义了测试程序载入任务 loadtestcase：

```
    task loadtestcase;      //load intstructions to instruction mem
        begin
            $readmemh({testcase, ".hex"},instructions.ram);
            $display("---Begin test case %s-----", testcase);
        end
    endtask
```

该任务用于载入指令文件，指令文件为文本格式，文件后缀改为.hex 以便区分。建议放在 XXX.sim/sim_1/behav/xsim 子目录下，用相对目录名来定位文件。使用$readmemh 读入到指定的指令存储中去，由于指令存储空间的声明不在顶层模块 instructions 中，需要使用 instructions.ram 来访问模块内部声明的变量 ram。在仿真测试文件需要按照实际定位来设置应该访问的变量位置。

测试文件还定义一系列的断言任务，辅助检查寄存器或者内存中的内容，并在出错时提供提示信息。

```
 task checkreg;         //check registers
 input [4:0]    regid;
```

```
input [31:0]   results;
reg [31:0]     debugdata;
begin
            debugdata=mycpu.myregfile.regs[regid]; //get register content
if(debugdata==results)
        begin
                $display("OK: end of cycle %d reg %h need to be %h,get %h",
                        numcycles-1, regid, results, debugdata);
        end
else    begin
                $display("!!!Error: end of cycle %d reg %h need to be %h,
                        get %h", numcycles-1, regid, results, debugdata);
         end
end
endtask
```

在这个任务中访问了 CPU 内部定义的寄存器堆 myregfile 中的 regs 变量，并根据所需要的访问寄存器编号 regid 来读取数据，并和预期数据进行比较。任务会提示比较结果，方便进行调试。同样的，也可以编写类似的内存内容比较模块，对内存中的内容进行检查。

假定需要测试 CPU 中加法指令的正确性，就可以编写一小段汇编，例如：

```
addi t1,zero,100
addi t2,zero,20
add t3,t1,t2
```

在这段汇编执行过程中，检查各个寄存器结果，观察代码执行的正确性。使用 RISC-V 汇编器工具如 RARS，Ripes 等将这段汇编转换为二进制，并写入 addtest.hex 文件中。然后将 addtest.hex 文件拷贝到指定项目目录的 XXX.sim/sim_1/behav/xsim 目录下，仿真程序就可以读取到测试文件 addtest.hex。示例文件的具体内容如下：

```
06400313
01400393
00730E33
```

仿真模块的具体执行代码如下：

```
initial begin:TestBench
        #80
        // output the state of every instruction
        $monitor("cycle=%d, pc=%h, instruct= %h op=%h,rs1=%h,rs2=%h, rd=%h, imm=%h",
                        numcycles, mycpu.pc, mycpu.instr, mycpu.op,
                        mycpu.rs1,mycpu.rs2,mycpu.rd,mycpu.imm);
        testcase = "addtest";
        loadtestcase();
        resetcpu();
```

```
        step();
        checkreg(6,100); //t1==100
        step();
        checkreg(7,20); //t2=20
        step();
        checkreg(28,120); //t3=120
        $stop
    end
```

执行过程中，首先使用$monitor 来定义需要观察的变量，只要这些变量发生变化仿真程序会自动地打印出变量的内容。这样，可以在每条指令执行时看到对应的 PC 及指令解码的关键信息。可以自定义需要观察的信号。在载入 add 测试用例后，testbench 单步执行了 3 次，每次执行完就按照预期的执行结果检查了 t1、t2 和 t3 寄存器。点击 Run Simulation，执行功能仿真。仿真的结果在 Tcl Console 里输出，实际输出如下：

```
---Begin test case                    addtest-----
cycle=          x, pc=xxxxxxxx, instruct= xxxxxxxx op=xx, rs1=xx,rs2=xx, rd=xx, imm=xxxxxxxx,test=xxxxxxxx,test2=x
cycle=          x, pc=00000000, instruct= 06400313 op=13, rs1=00,rs2=04, rd=06, imm=00000064,test=00000000,test2=0
cycle=          0, pc=00000000, instruct= 06400313 op=13, rs1=00,rs2=04, rd=06, imm=00000064,test=00000000,test2=0
cycle=          0, pc=00000004, instruct= 01400393 op=13, rs1=00,rs2=14, rd=07, imm=00000014,test=00000000,test2=0
cycle=          1, pc=00000004, instruct= 01400393 op=13, rs1=00,rs2=14, rd=07, imm=00000014,test=00000000,test2=0
OK: end of cycle          0 reg 06 need to be 00000064, get 00000064
cycle=          1, pc=00000008, instruct= 00730e33 op=33, rs1=06,rs2=07, rd=1c, imm=00000007,test=00000064,test2=0
cycle=          2, pc=00000008, instruct= 00730e33 op=33, rs1=06,rs2=07, rd=1c, imm=00000007,test=00000064,test2=0
OK: end of cycle          1 reg 07 need to be 00000014, get 00000014
cycle=          2, pc=0000000c, instruct= xxxxxxxx op=xx, rs1=xx,rs2=xx, rd=xx, imm=xxxxxxxx,test=xxxxxxxx,test2=x
cycle=          3, pc=0000000c, instruct= xxxxxxxx op=xx, rs1=xx,rs2=xx, rd=xx, imm=xxxxxxxx,test=xxxxxxxx,test2=x
OK: end of cycle          2 reg 1c need to be 00000078, get 00000078
$stop called at time : 165 ns : File "D:/My_design/lab8/lab8.srcs/sim_1/new/SingleCycleCPU_tb.v" Line 192
```

从输出中可以看到初始化结束后，代码从全零地址开始执行，每个周期结束后会对寄存器进行检查。注意这里检查点在上升沿到来后，所以在第 n 周期结束时，PC 和指令已经是下一条指令的内容了。

按照自己的设计修改单步仿真 testbench，并自行设计测试用例来对 CPU 进行初步联调。确保 CPU 可以完成基础功能。

3、测试集验证

单步功能仿真用于简单验证 CPU 中各条指令的基本情况，确保 CPU 可以完成基础功能。为了排除 CPU 中潜在的 bug，需要对 CPU 实现进行详细的测试，避免后面在搭建整个计算机系统时由于 CPU 的问题出现难以定位的 bug。需要使用 RISC-V 的官方测试集来对 CPU 进行全面的系统测试。

RISC-V 官方测试集针对不同的 RISC-V 指令变种都提供了测试。在本实验中，主要使用 rv32ui 也就是 RV32 的基本指令集，u 表示是用户态，i 表示是整数基本指令集。实验中采用的环境是无虚拟地址的环境，即只使用物理地址访问内存。所以，主要关注 rv32ui-p 开头的测试即可。

由于官方测试集需要使用 risc-v gcc 工具链，测试集的代码中使用了系统调用指令，需要对官方测试的代码进行一定的修改才可以用于本次实验的测试。感兴趣的同学可以自行下载 risc-v gcc 工具链进行修改。

本次实验提供了已经修改好的测试用例 38 个测试文件 rv32ui-p-***.hex，将这些文件拷贝到 XXX.sim/sim_1/behav/xsim 目录下，就可以进行测试验证。

需要修改测试文件以便支持对官方测试集的仿真。主要增加了以下辅助任务：

测试集 TestBench

```
integer maxcycles =10000;
task run;
      integer i;
      begin
         i = 0;
         while ( (mycpu.instr!=32'hdead10cc) && (i<maxcycles))
         begin
               step();
                i = i+1;
          end
       end
endtask
```

代码运行任务 run 会一直用单步运行代码，直到遇到定义的代码终止信号“dead10cc”为止。如果代码一直不终止也会在给定最大运行周期（10000）后停止仿真。

```
task checkmagnum;
        begin
              if (numcycles>maxcycles)
               begin
                 $display ("!!!Error:test case %s does not terminate!", testcase);
               end
               else if (mycpu.myregfile.regs[10]==32'hc0ffee)
                        begin
```

```
                $display ("OK:test case %s finshed OK at cycle %d.",
                        testcase, numcycles-1);
            end
            else if (mycpu.myregfile.regs[10]==32'hdeaddead)
                begin
                    $display ("!!!ERROR:test case %s finshed with error
                            in cycle %d.",testcase, numcycles-1);
                end
                else
                begin
                    $display ("!!!ERROR:test case %s unknown error in
                            cycle %d.",testcase, numcycles-1);
                end
        end
    endtask
```

对仿真结果测试是通过对仿真结束后 a0 寄存器中数据是否符合预期来进行判断的。当然如果程序不终止，或者 a0 数据不正常也会报错。

数据存储可以用实验生成的 hex 文件进行初始化，一般只有在访存指令的测试时才需要初始化数据存储。

```
task loaddatamem;
    begin
        $readmemh ({testcase, "_d.hex"},datamem.ram);
    end
endtask
```

也提供了一个简单的可以执行单个测试用例的任务。

```
task run_riscv_test;
    begin
        loadtestcase();
        loaddatamem();
        resetcpu();
        run();
        checkmagnum();
    end
endtask
```

在仿真过程中只需要按顺序执行所有需要的仿真即可：

```
testcase = "rv32ui-p-simple";
run_riscv_test();
testcase = "rv32ui-p-add";
run_riscv_test();
```

```
testcase = "rv32ui-p-addi";
run_riscv_test();
testcase = "rv32ui-p-and";
run_riscv_test();
…
```

在测试集仿真过程中可以暂时注释$monitor 任务，只有在出错时再检查具体测试用例为何出错。官方测试集仿真通过后的输出结果示例如下。

```
run all
OK:test case                    rv32ui-p-add finshed OK at cycle          456.
---Begin test case                rv32ui-p-addi-----
OK:test case                   rv32ui-p-addi finshed OK at cycle          233.
---Begin test case                 rv32ui-p-and-----
OK:test case                    rv32ui-p-and finshed OK at cycle          476.
---Begin test case                rv32ui-p-andi-----
OK:test case                   rv32ui-p-andi finshed OK at cycle          189.
---Begin test case               rv32ui-p-auipc-----
OK:test case                  rv32ui-p-auipc finshed OK at cycle           50.
---Begin test case                 rv32ui-p-beq-----
OK:test case                    rv32ui-p-beq finshed OK at cycle          282.
---Begin test case                 rv32ui-p-bge-----
OK:test case                    rv32ui-p-bge finshed OK at cycle          300.
---Begin test case                rv32ui-p-bgeu-----
OK:test case                   rv32ui-p-bgeu finshed OK at cycle          325.
---Begin test case                 rv32ui-p-blt-----
OK:test case                    rv32ui-p-blt finshed OK at cycle          282.
---Begin test case                rv32ui-p-bltu-----
OK:test case                   rv32ui-p-bltu finshed OK at cycle          307.
---Begin test case                 rv32ui-p-bne-----
OK:test case                    rv32ui-p-bne finshed OK at cycle          282.
---Begin test case                 rv32ui-p-jal-----
OK:test case                    rv32ui-p-jal finshed OK at cycle           46.
---Begin test case                rv32ui-p-jalr-----
OK:test case                   rv32ui-p-jalr finshed OK at cycle          106.
```

```
---Begin test case                rv32ui-p-lb-----
OK:test case                      rv32ui-p-lb finshed OK at cycle          236.
---Begin test case                rv32ui-p-lbu-----
OK:test case                      rv32ui-p-lbu finshed OK at cycle         236.
---Begin test case                rv32ui-p-lh-----
OK:test case                      rv32ui-p-lh finshed OK at cycle          248.
---Begin test case                rv32ui-p-lhu-----
OK:test case                      rv32ui-p-lhu finshed OK at cycle         255.
---Begin test case                rv32ui-p-lui-----
OK:test case                      rv32ui-p-lui finshed OK at cycle          56.
---Begin test case                rv32ui-p-lw-----
OK:test case                      rv32ui-p-lw finshed OK at cycle          258.
---Begin test case                rv32ui-p-or-----
OK:test case                      rv32ui-p-or finshed OK at cycle          479.
---Begin test case                rv32ui-p-ori-----
OK:test case                      rv32ui-p-ori finshed OK at cycle         196.
---Begin test case                rv32ui-p-sb-----
OK:test case                      rv32ui-p-sb finshed OK at cycle          421.
---Begin test case                rv32ui-p-sh-----
OK:test case                      rv32ui-p-sh finshed OK at cycle          474.
---Begin test case                rv32ui-p-sll-----
OK:test case                      rv32ui-p-sll finshed OK at cycle         484.
---Begin test case                rv32ui-p-slli-----
OK:test case                      rv32ui-p-slli finshed OK at cycle        232.
---Begin test case                rv32ui-p-slt-----
OK:test case                      rv32ui-p-slt finshed OK at cycle         450.
---Begin test case                rv32ui-p-slti-----
OK:test case                      rv32ui-p-slti finshed OK at cycle        228.
---Begin test case                rv32ui-p-sltiu-----
OK:test case                      rv32ui-p-sltiu finshed OK at cycle       228.
---Begin test case                rv32ui-p-sltu-----
OK:test case                      rv32ui-p-sltu finshed OK at cycle        450.
```

```
---Begin test case                    rv32ui-p-sra-----
OK:test case                    rv32ui-p-sra finshed OK at cycle          503.
---Begin test case                    rv32ui-p-srai-----
OK:test case                    rv32ui-p-srai finshed OK at cycle          247.
---Begin test case                    rv32ui-p-srl-----
OK:test case                    rv32ui-p-srl finshed OK at cycle          497.
---Begin test case                    rv32ui-p-srli-----
OK:test case                    rv32ui-p-srli finshed OK at cycle          241.
---Begin test case                    rv32ui-p-sub-----
OK:test case                    rv32ui-p-sub finshed OK at cycle          448.
---Begin test case                    rv32ui-p-sw-----
OK:test case                    rv32ui-p-sw finshed OK at cycle          481.
---Begin test case                    rv32ui-p-xor-----
OK:test case                    rv32ui-p-xor finshed OK at cycle          478.
---Begin test case                    rv32ui-p-xori-----
OK:test case                    rv32ui-p-xori finshed OK at cycle          198.
$stop called at time : 229590 ns : File "D:/My_design/lab8/lab8.srcs/sim_1/new/SingleCycleCPU_tb.v" Line 273
```

对于单周期 CPU，由于需要在一个周期内完成指令执行的所有步骤，很可能不能以 100MHz 运行。请观察你的 CPU 综合后 Timing Analysis 结果是否存在时序不满足，即某些 model 下 Setup Slack 为负数。此时，可以考虑调整设计减少关键路径时延，或者降低主频。

测试集测试通过后，可考虑加载实际程序上实验开发板进行验证，在顶层模块中，通过拨档开关来输入参数，通过七段数码管来显示输出。例如：验证累加和程序或生成第 n 个斐波那契数列程序，并上开发板验证，实验板 8 个开关输入初始值 n，7 段数码管或 led 灯显示结果。（可选）

请根据上述描述，按照下列步骤完成实验。

1、 使用 Vivado 创建一个新工程 lab8。

2、 点击添加设计源码文件，加入 lab8.zip 里的 InstrMem.v、InstrParse.v、InstrToImm.v、 nextPC.v、BranchControl.v、Control.v、SingleCycleCPU.v 等文件以及前面实验已经实现的 DataMem.v、barrelsft32.v、ALU32.v 等文件。

3、 点击添加测试文件，加入 lab8.zip 里的 SingleCycleCPU_tb .v 文件。将 testcase 目录下的文件拷贝到 XXX.sim/sim_1/behav/xsim 目录下。

4、 点击添加约束文件，加入 lab8.zip 里的 SingleCycleCPU.xdc 文件。

5、 根据实验要求，完成源码文件的设计。

6、 对工程进行仿真测试。

7、 仿真通过后，进行综合、实现并生成比特流文件。

8、 生成位流文件后，加载到实验开发板，进行调试验证，并记录验证过程。

五、思考题

1、分析比较运算实用独立比较器件和利用减法运算来实现的区别。

2、分析为什么单周期 CPU 中指令存储器的读操作必须是异步读取？寄存器堆如何实现写后读操作？

3、思考如何实现支持 RV32M 指令集的单周期中央处理器？