数字逻辑与计算机组成实验报告

实验十一 RV32I 单周期 CPU

姓名：崔家才

学号：201220014

班级：计科 1 班

邮箱：201220014@smail.nju.edu.cn

实验时间：2021.11.27

目录

1 实验目的

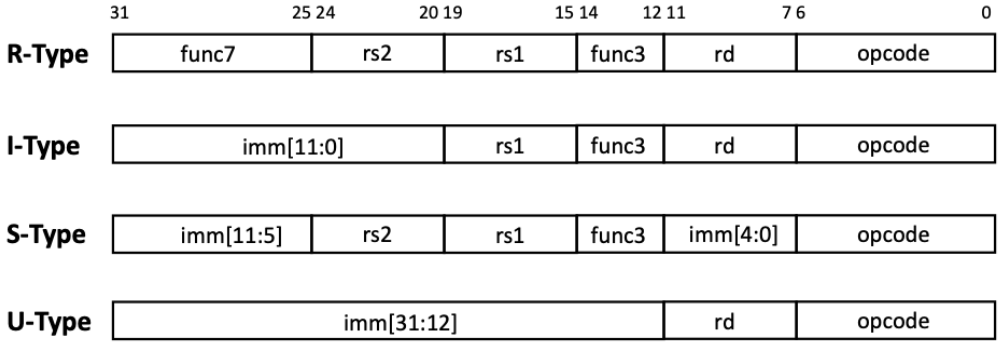
本实验的目标是利用 FPGA 实现 RV32I 指令集中除系统控制指令之外的其余指令。

利用单周期方式实现 RV32I 的控制通路及数据通路，并能够顺利通过 功能仿真。

2 实验原理

2.1 RV32I 指令编码

2.1.1 RV32I 指令编码四种基本格式



2.1.2 各指令格式的简要解析

R-Type:为寄存器操作数指令，含 2 个源寄存器 rs1,rs2 和一个目的寄存器 rd。

I-Type:为立即数操作指令，含一个源寄存器和一个目的寄存器和一个 12bit 立即数操作数

S-Type:为存储器写指令，含两个源寄存器和一个 12bit 立即数。

B-Type:为跳转指令，实际是 S-Type 的变种。与 S-Type 主要的区别是立即数编码。S-Type 中的 imm[11:5]变为{immm[12], imm[10:5]}，imm[4:0]变为{imm[4:1], imm[11]}。

U-Type:为长立即数指令，含一个目的寄存器和 20bit 立即数操作数。

J-Type:为长跳转指令，实际是 U-Type 的变种。与 U-Type 主要的区别是立即数编码。U-Type 中的 imm[31:12]变为{imm[20], imm[10:1], imm[11], imm[19:12]}。

2.2 RV32I 中的通用寄存器

2.2.1 RV32I 中通用寄存器的定义与用法

Register	Name	Use	Saver
x0	zero	Constant 0	–
x1	ra	Return Address	Caller
x2	sp	Stack Pointer	Callee
x3	gp	Global Pointer	–
x4	tp	Thread Pointer	–
x5~x7	t0~t2	Temp	Caller
x8	s0/fp	Saved/Frame pointer	Callee
x9	s1	Saved	Callee
x10~x11	a0~a1	Arguments/Return Value	Caller
x12~x17	a2~a7	Arguments	Caller
x18~x27	s2~s11	Saved	Callee
x28~x31	t3~t6	Temp	Caller

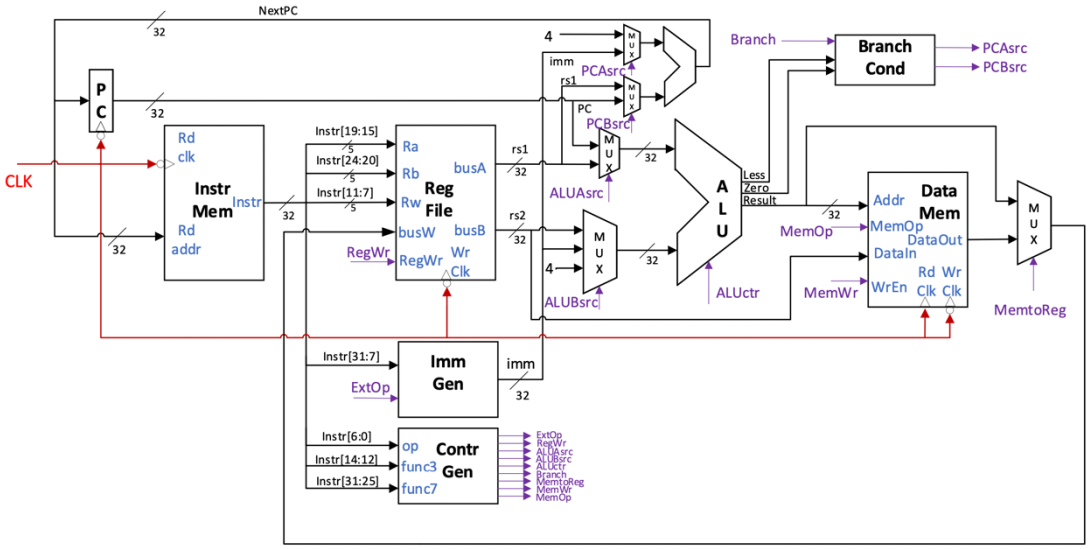
注：具体实现详见实验 10 报告。

2.3 RV32I 电路实现

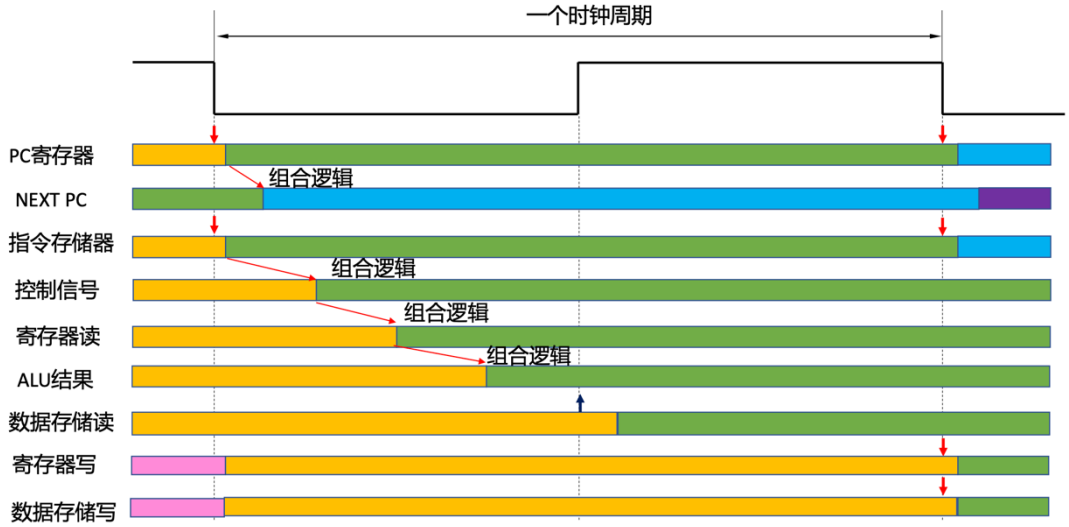
2.3.1 指令执行的基本流程

1. **取指令**:使用本周期新的 PC 从指令存储器中取出指令，并将其放入指令寄存器(IR)中
2. **指令译码**:对取出的指令进行分析，生成本周期执行指令所需的控制信 号，并计算下一条指令的地址
3. **读取操作数**:从寄存器堆中读取寄存器操作数，并完成立即数的生成
4. **运算**:利用 ALU 对操作数进行必要的运算
5. **访问内存**:包括读取或写入内存对应地址的内容
6. **寄存器写回**:将最终结果写回到目的寄存器中

2.3.2 单周期 CPU 的电路设计



2.3.3 单周期 CPU 的时序设计



2.4 CPU 接口设计

```
module rv32is(
    input    clock, //CPU时钟，下降沿有效
    input    reset, //reset信号，高电平有效
    input  [31:0] imemdataout, //指令存储器提供的32位指令数据
    input  [31:0] dmemdataout, //数据存储器提供的32位读取数据

    output [31:0] imemaddr, //32位指令地址
    output   imemclk, //指令读取时钟，上升沿读取，如果要在时钟下降沿读取指令，请将时钟取反后提供给imemclk

    output [31:0] dmemaddr, //32位数据地址
    output [31:0] dmemdatain, //32位写入数据内容

    output   dmemrdclk, //数据存储器的读取时钟，上升沿有效
    output   dmemwrclk, //数据存储器的写入时钟，上升沿有效，如果要在CPU时钟的下降沿写入，请将CPU时钟取反后输出到本信号上

    output [2:0] dmemop, //数据存储器的读写方式，按讲义设置
    output   dmemwe, //数据存储器写使能，高电平有效
    output [31:0] dbgdata //32位测试数据，本实验中请将PC连接到此信号上
    );
```

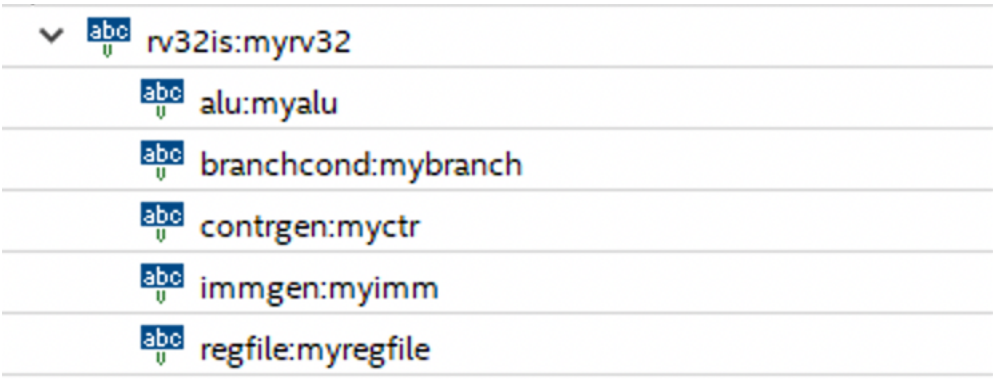
接口描述详见上图注释。

3 实验环境/器材

Quarters17.1 集成开发环境 Lite 版，头哥在线测试平台。

4 程序主要代码及设计思路

4.1 模块划分



CPU 模块:主要对外接口包括时钟、Reset、指令存储器的地址/数据线、数据存储器的地址及数据线和自行设计的调试信号。

ALU 模块: 主要对外接口是 ALU 的输入操作数、ALU 控制字、ALU 结果输出和标志位输出等。

加法器模块

桶型移位器模块

寄存器堆模块:主要对外接口是读写寄存器号输入、写入数据输入、寄存器控制信号、写入时钟、输出数据。

控制信号生成模块:主要对外接口是指令输入及各种控制信号输出。

立即数生成器模块:主要对外接口是指令输入，立即数类型及立即数输出。

跳转控制模块:主要对外接口是 ALU 标志位输入、跳转控制信号输入及 PC 选择信号输出。

PC 生成模块:主要对外接口是 PC 输入、立即数输入，rs1 输入，PC 选择信号及 NEXTPC 输出。

4.2 指令译码

```
28      // instruction decode
29      wire [31:0] instr;
30      assign instr = imemdataout;
31
32      wire [4:0] ra, rb, rw;
33      wire [6:0] op;
34      wire [2:0] func3;
35      wire [6:0] func7;
36
37      assign op = instr[6:0];
38      assign ra = instr[19:15];
39      assign rb = instr[24:20];
40      assign rw = instr[11:7];
41      assign func3 = instr[14:12];
42      assign func7 = instr[31:25];
```

指令译码主要采用组合逻辑，译码结果留待其他模块调用。

4.3 控制器

```
// control signal genetation
wire [2:0] extOp, memOp, branch;
wire regWr, memToReg, memWr, aluAsrc;
wire [1:0] aluBsrc;
wire [3:0] aluCtr;

contrgen myctr(op, func3, func7, extOp, regWr, aluAsrc, aluBsrc, aluCtr, branch, memToReg, memWr, memOp);
```

控制器根据之前指令译码的结果生成所需的各种控制信号，其中控制器内部是一个**纯组合逻辑**，具体内容详见 contrgen.v 文件。

因为其他各个部分的实现都需要基于指令解码的结果以及控制信号，故而先实现解码器和控制器。

4.4 寄存器堆

```
// regfile
wire [31:0] busW, busA, busB;
regfile myregfile(ra, rb, rw, busW, regWr, regwrclk, busA, busB);
```

从根据指令解码结果和控制信号从寄存器堆中取出所需的数据，放入 busA 和 busB 中。这里 regwrclk 为寄存器堆的写时钟，寄存器堆的具体设计详见 **regfile.v** 以及实验 10 报告。

4.5 立即数生成器

```
// immediate number generation
wire [31:0] imm;
immgen myimm(extOp, instr, imm);
```

根据指令以及控制信号生成立即数，具体代码详见 **immgen.v**。

4.6 PC 定义、ALU 源操作数生成模块及 ALU

```
//program counter
reg [31:0] pc = 0;
assign dbgdata = pc;

// alu
reg [31:0] dataA, dataB;
wire [31:0] result;
wire less, zero;
always @ (*) begin
   case (aluAsrc)
      1'b0: dataA = busA;
      1'b1: dataA = pc;
   endcase
   case (aluBsrc)
      2'b00: dataB = busB;
      2'b01: dataB = imm;
      2'b10: dataB = 32'd4;
   endcase
end
alu myalu(dataA, dataB, aluCtr, less, zero, result);
```

因为 ALU 需要用到 pc 的值，所以先将 pc 定义出来，初始为 0，关于 pc 的更新模块见 4.8，因为 verilog 的不同语句块之间是并行的，所以代码上的先后并不影响。

根据控制器生成的 aluAsrc 和 aluBsrc 信号，利用纯组合逻辑生成 ALU 所需的源操作数，传入 ALU 模块。

ALU 模块的具体实现详见 **alu.v** 以及实验 10 报告。

4.7 分支控制模块

```
// branch
wire pcAsrc, pcBsrc;
branchcond mybranch(branch, zero, less, pcAsrc, pcBsrc);
```

根据控制信号以及 ALU 生成的标志位信息生成 pc 更新模块源操作数控制信号，本模块也是一个纯组合逻辑。

分支控制模块具体实现详见 **branchcond.v**。

4.8 pc 更新模块

```
// pc
wire [31:0] pcA, pcB;
wire [31:0] nextpc;
assign pcA = pcAsrc ? imm : 32'd4;
assign pcB = pcBsrc ? busA : pc;
assign nextpc = reset ? 32'd0 : pcA + pcB;
assign imemaddr = nextpc;

always @ (negedge clock) begin
   if (reset) begin
      pc <= 32'd0;
   end
   else begin
      pc <= nextpc;
   end
end
```

首先根据分支模块提供的利用纯组合逻辑生成下一次的值 nextpc，在 CPU 时钟下降沿时将 pc 更新为 nextpc。

此处需要注意的是赋给指令存储器的是 nextpc，而不是 pc；这样当下降沿到来的时候，指令存储器会根据 nextpc 取指令，pc 也会更新为 nextpc，这样保证了 pc 和指令是相对应的。

还有一点需要注意的是，reset 信号下，pc 和 nextpc 都需要清零，否则会影响初始的 pc 时序。

4.9 数据存储器读写模块

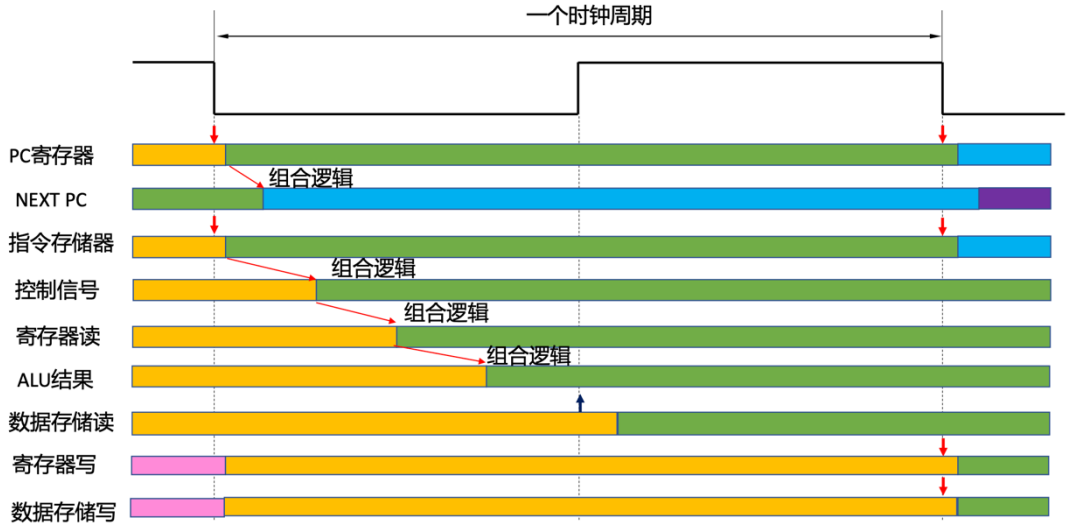
```
// mem
assign dmemop = memOp;
assign dmemwe = memWr;
assign dmemaddr = result;
assign dmemdatain = busB;
assign busW = memToReg ? dmemdataout : result;
```

根据控制信号，以 alu 计算结果为地址读写数据存储器，读出数据根据控制信号决定是否要送往寄存器堆；写入数据为寄存器堆 busB 输出。

4.10 CPU 各模块的时序

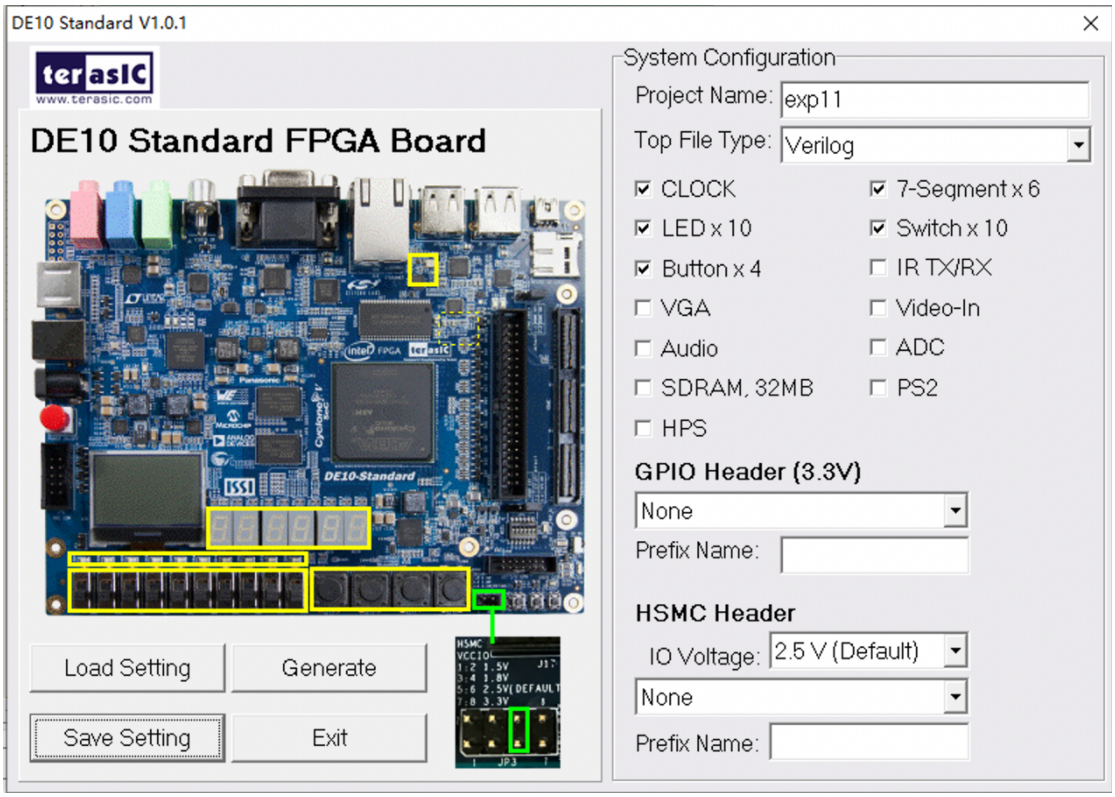
```
// clocks
assign imemclk = ~clock;
assign dmemrdclk = clock;
assign dmemwrclk = ~clock;
wire regwrclk;
assign regwrclk = ~clock;
```

各模块时钟和 CPU 时钟的关系如上图，时序设计示意图如下：

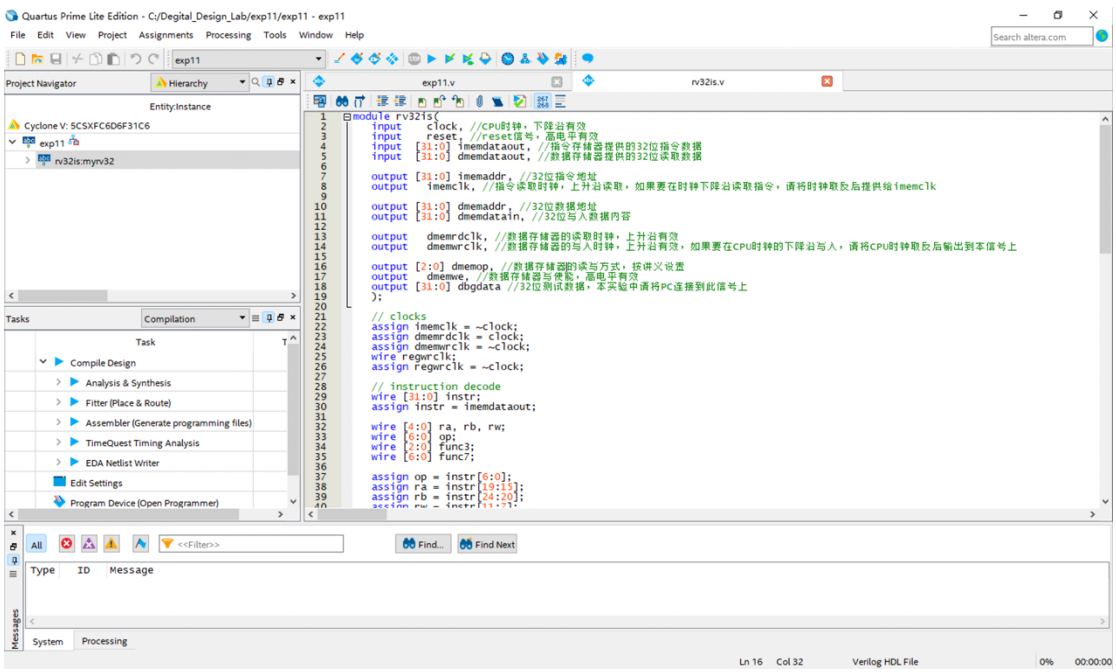


5 实验过程

1、利用 SystemBuilder 建立 Quartus 工程项目；



2、完成主要模块的代码编写；



3、编译、调试、测试、验证结果。

4、在线测试与验收。

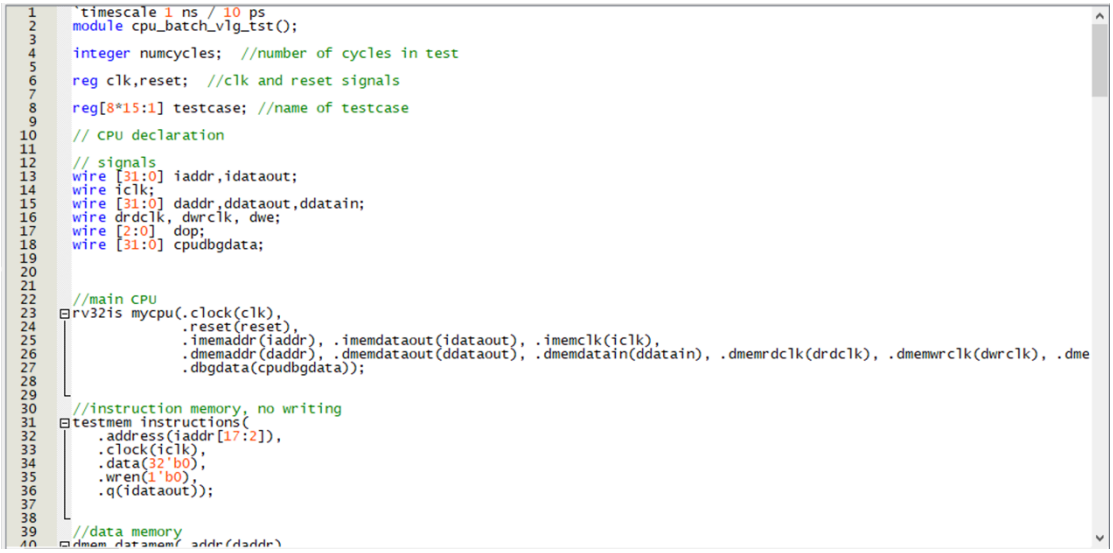
6 测试方法

6.1 单周期 CPU 的 ModelSim 单步测试

```
`timescale 1 ns / 10 ps
module cpu_single_vlg_tst();

integer numcycles;  //number of cycles in test

reg clk,reset;  //clk and reset signals

reg[8*8:1] testcase; //name of testcase

// 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 [31:0] cpudbgdata;


//main CPU
rv32is mycpu(.clock(clk),
             .reset(reset),
             .imemaddr(iaddr), .imemdataout(idataout), .imemclk(iclk),
             .dmemaddr(daddr), .dmemdataout(ddataout), .dmemdatain(ddatain), .dmemrdclk(drdclk), .dmemwrclk(dwrclk), .dme
             .dbgdata(cpudbgdata));

//instruction memory, no writing
testmem instructions(
    .address(iaddr[17:2]),
    .clock(iclk),
    .data(32'b0),
    .wren(1'b0),
    .q(idataout));

//data memory
dmem datamem( addr(daddr)
```

具体内容详见 cpu_step.vt。

6.2 单周期 CPU 的 ModelSim 官方测试集



```
`timescale 1 ns / 10 ps
module cpu_batch_vlg_tst();

integer numcycles;  //number of cycles in test

reg clk,reset;  //clk and reset signals

reg[8*15:1] testcase; //name of testcase

// 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 [31:0] cpudbgdata;


//main CPU
rv32is mycpu(.clock(clk),
             .reset(reset),
             .imemaddr(iaddr), .imemdataout(idataout), .imemclk(iclk),
             .dmemaddr(daddr), .dmemdataout(ddataout), .dmemdatain(ddatain), .dmemrdclk(drdclk), .dmemwrclk(dwrclk), .dme
             .dbgdata(cpudbgdata));

//instruction memory, no writing
testmem instructions(
    .address(iaddr[17:2]),
    .clock(iclk),
    .data(32'b0),
    .wren(1'b0),
    .q(idataout));

//data memory
dmem datamem( addr(daddr)
```

具体内容详见 cpu_batch.vt。

7 实验结果

7.1 ModelSim 仿真测试

7.1.1 单周期 CPU 单步测试结果

```
# ~~~ Begin test case      add ~~~
# ~~~ OK: end of cycle   1 reg 06 need to be 00000064, get 00000064
# ~~~ OK: end of cycle   1 PC/dbgdata need to be 00000004, get 00000004
# ~~~ OK: end of cycle   2 reg 07 need to be 00000014, get 00000014
# ~~~ OK: end of cycle   2 PC/dbgdata need to be 00000008, get 00000008
# ~~~ OK: end of cycle   3 reg 1c need to be 00000078, get 00000078
# ~~~ Begin test case      alu ~~~
# ~~~ OK: end of cycle   1 reg 06 need to be 0000004f, get 0000004f
# ~~~ OK: end of cycle   2 reg 07 need to be 00000003, get 00000003
# ~~~ OK: end of cycle   3 reg 1c need to be 0000004c, get 0000004c
# ~~~ OK: end of cycle   4 reg 1c need to be 00000003, get 00000003
# ~~~ OK: end of cycle   5 reg 1c need to be 00000278, get 00000278
# ~~~ OK: end of cycle   6 reg 1c need to be 00000000, get 00000000
# ~~~ OK: end of cycle   7 reg 1c need to be 00000001, get 00000001
# ~~~ OK: end of cycle   8 reg 1c need to be 0000004c, get 0000004c
# ~~~ OK: end of cycle   9 reg 1c need to be 00000009, get 00000009
# ~~~ OK: end of cycle  10 reg 1c need to be 0000004f, get 0000004f
# ~~~ OK: end of cycle  11 reg 06 need to be fffffffb1, get fffffffb1
# ~~~ OK: end of cycle  12 reg 1c need to be fffffffb4, get fffffffb4
# ~~~ OK: end of cycle  13 reg 1c need to be fffffff6, get fffffff6
# ~~~ OK: end of cycle  14 reg 1c need to be 1fffffff6, get 1fffffff6
# ~~~ OK: end of cycle  15 reg 1c need to be 00000001, get 00000001
# ~~~ OK: end of cycle  16 reg 1c need to be 00000000, get 00000000
# ~~~ Begin test case      mem ~~~
# ~~~ OK: end of cycle   1 reg 0a need to be 00008000, get 00008000
# ~~~ OK: end of cycle   2 reg 0a need to be 00008010, get 00008010
# ~~~ OK: end of cycle   3 reg 06 need to be 000004d2, get 000004d2
# ~~~ OK: end of cycle   4 mem addr= 00008014 need to be 000004d2, get 000004d2
# ~~~ OK: end of cycle   5 reg 07 need to be 000004d2, get 000004d2
# ~~~ OK: end of cycle   6 mem addr= 00008018 need to be 00000000, get 00000000
# ~~~ OK: end of cycle   7 reg 06 need to be 000000ff, get 000000ff
# ~~~ OK: end of cycle   8 mem addr= 00008018 need to be 000000ff, get 000000ff
# ~~~ OK: end of cycle   9 reg 07 need to be ffffffff, get ffffffff
# ~~~ OK: end of cycle  10 reg 07 need to be 000000ff, get 000000ff
# ~~~ OK: end of cycle  11 mem addr= 00008018 need to be 0000ffff, get 0000ffff
# ~~~ OK: end of cycle  12 reg 07 need to be ffffffff, get ffffffff
# ~~~ OK: end of cycle  13 reg 07 need to be 0000ffff, get 0000ffff
# ~~~ OK: end of cycle  14 reg 07 need to be ffffffff, get ffffffff
# ~~~ OK: end of cycle  15 reg 07 need to be 00000000, get 00000000
# ~~~ OK: end of cycle  16 mem addr= 0000801c need to be 00000000, get 00000000
```

```
# ~~~ OK: end of cycle  17 reg 06 need to be 00000078, get 00000078
# ~~~ OK: end of cycle  18 mem addr= 0000801c need to be 00000078, get 00000078
# ~~~ OK: end of cycle  19 reg 06 need to be 00000056, get 00000056
# ~~~ OK: end of cycle  20 mem addr= 0000801c need to be 00005678, get 00005678
# ~~~ OK: end of cycle  21 reg 06 need to be 00000034, get 00000034
# ~~~ OK: end of cycle  22 mem addr= 0000801c need to be 00345678, get 00345678
# ~~~ OK: end of cycle  23 reg 06 need to be 00000012, get 00000012
# ~~~ OK: end of cycle  24 mem addr= 0000801c need to be 12345678, get 12345678
# ~~~ OK: end of cycle  25 reg 07 need to be 12345678, get 12345678
# ~~~ Begin test case   branch ~~~
# ~~~ OK: end of cycle   1 reg 05 need to be 00000064, get 00000064
# ~~~ OK: end of cycle   2 reg 06 need to be fffffffe, get fffffffe
# ~~~ OK: end of cycle   3 PC/dbgdata need to be 0000000c, get 0000000c
# ~~~ OK: end of cycle   4 PC/dbgdata need to be 00000014, get 00000014
# ~~~ OK: end of cycle   5 PC/dbgdata need to be 00000018, get 00000018
# ~~~ OK: end of cycle   6 PC/dbgdata need to be 0000001c, get 0000001c
# ~~~ OK: end of cycle   7 PC/dbgdata need to be 00000024, get 00000024
# ~~~ OK: end of cycle   8 PC/dbgdata need to be 00000028, get 00000028
# ~~~ OK: end of cycle   9 PC/dbgdata need to be 00000044, get 00000044
# ~~~ OK: end of cycle   9 reg 01 need to be 0000002c, get 0000002c
# ~~~ OK: end of cycle  10 PC/dbgdata need to be 00000048, get 00000048
# ~~~ OK: end of cycle  10 reg 05 need to be 0000007b, get 0000007b
# ~~~ OK: end of cycle  11 PC/dbgdata need to be 00000038, get 00000038
# ~~~ OK: end of cycle  12 PC/dbgdata need to be 0000003c, get 0000003c
# ~~~ OK: end of cycle  13 reg 0a need to be 00c0ffee, get 00c0ffee
# ~~~ OK: end of cycle  14 PC/dbgdata need to be 0000004c, get 0000004c
```

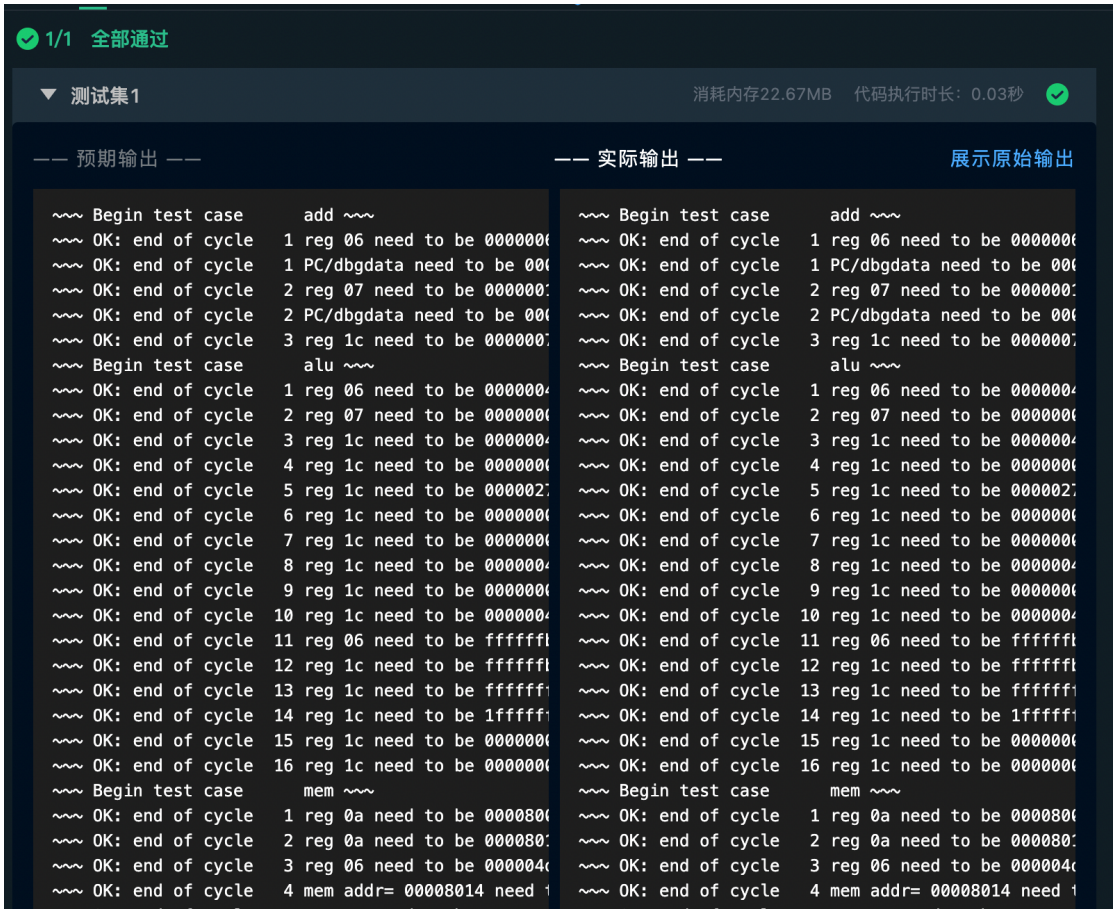
7.1.2 单周期 CPU 官方测试集测试结果

```
# ~~~ Begin test case rv32ui-p-simple ~~~
# ~~~ OK:test case rv32ui-p-simple finshed OK at cycle  32.
# ~~~ Begin test case    rv32ui-p-add ~~~
# ~~~ 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.
```

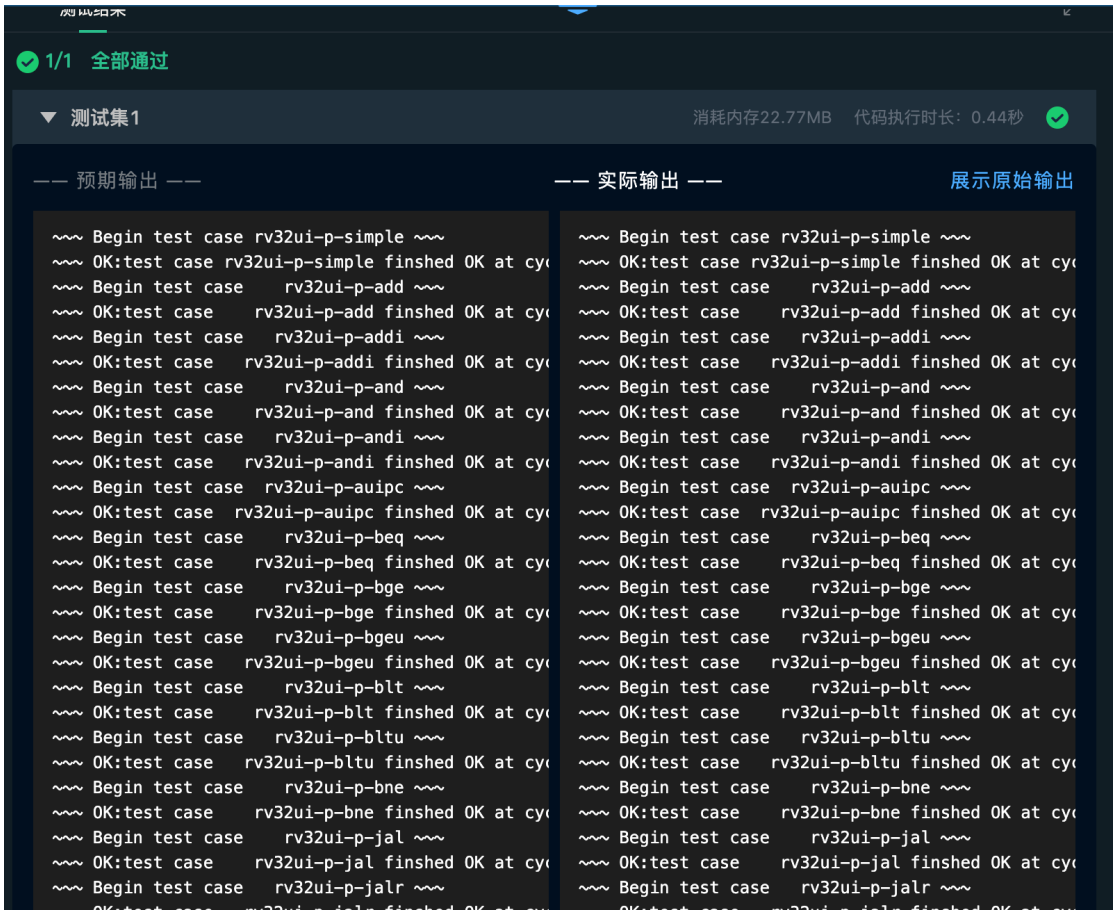
```
# ~~~ Begin test case     rv32ui-p-or ~~~
# ~~~ OK:test case     rv32ui-p-or finshed OK at cycle 479.
# ~~~ Begin test case    rv32ui-p-ori ~~~
# ~~~ OK:test case    rv32ui-p-ori finshed OK at cycle 196.
# ~~~ Begin test case     rv32ui-p-sb ~~~
# ~~~ OK:test case     rv32ui-p-sb finshed OK at cycle 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.
```

7.2 头歌在线测试

7.2.1 单周期 CPU 功能测试的头歌在线测试结果



7.2.2 单周期 CPU 官方测试的头歌在线测试结果



8 实验中遇到的问题和解决办法

8.1 单周期 CPU 的时序设计问题

起初 reset 初始化的时候没有将 nextpc 连同 pc 一起初始化，导致后序出现了诸多时序问题。

9 实验得到的启示

有时候时序错误可能并不是本身设计有问题，而是在初始化的时候没有给定一个合适的初态。

10 意见和建议

本次暂无。