中国科学院大学计算机组成原理实验课

实 验 报 告

学号：2019K80009915022 姓名：栗祺菁 专业：计算机科学与技术

实验序号：prj4 实验名称：RSIC-V指令集处理器实现

一、 逻辑电路结构与仿真波形的截图及说明（比如关键RTL代码段{包含注释}及其对应的逻辑电路结构、相应信号的仿真波形和信号变化的说明等）

该实验和实验3的差别仅仅在于指令集的不同，以及状态机有微小差别。以下先简单复述实验3中三段式状态机、指令和数据寄存器的原理，不再放代码。然后再介绍一点指令集的差别。

0.RSIC-V指令集与MIPS指令集的对比：

此次实验实现的指令只有37条，而MIPS实现的指令有45条。而它们实现的是同样的功能，可以见得RSIC-V指令集更加精简。

并且注意到，尽管RSIC-V指令中立即数或偏移量有很多情况是由好几个部分拼接起来的，但是它的最高位一定是放在Instrution[31]，这样就能够在符号位扩展的时候直接利用指令的最高位进行扩展。在branch类型的指令中，offset的最低位一定为0，即分支跳转的偏移量一定为2的倍数，并且偏移量能够表达的数的范围为2^12，这比MIPS的跳转范围小（有16位的立即数）。

除此之外，RISC-V具有更加规整的指令格式。在实现R-type运算指令和I-type运算指令的时候，只有两个不同的opcode值，其它的也是一样，一个大类一个opcode值。具体的指令类型则由funct3指出（部分还要由funct7协助），而R-type和I-type的运算指令同一个操作的funct3是一样的，例如ADD和ADDI的funct3都是000。这样在译码的时候就可以将I-type和R-type合并译码。

下面简单地说一下指令的实现过程，实现思想和MIPS指令大同小异，很多内容在prj2的实验报告已经说过，不在此赘述。

0.1 第一部分是各个指令段的选取，省略

0.2 第二部分是PC的赋值：jal指令、jalr指令、branch指令各有其对应的跳转地址，除此三种之外next_PC=PC+4。

```
    assign Jump_address0=PC_s+4;//else

    assign pc_jal_offset={Instruction_r[31],Instruction_r[19:12],
        Instruction_r[20],Instruction_r[30:21],1'b0};
    assign Jump_address1=PC_s+{{11{pc_jal_offset[20]}},pc_jal_offset}; //jal

    assign pc_jalr_offset=sign_extend_imm+reg_read_data1_r;
    assign Jump_address2={pc_jalr_offset[31:1],1'b0};//jalr

    assign pc_branch_offset={Instruction_r[31],Instruction_r[7],
    Instruction_r[30:25],Instruction_r[11:8],1'b0};
    assign Jump_address3=PC_s+{{19{pc_branch_offset[12]}},pc_branch_offset};
    //blt,bltu,bge,bgeu,beq,bne
```

```
    //next PC
    assign PC_1=(opcode==7'b1101111);//jal
    assign PC_2=(opcode==7'b1100111);//jalr
    assign PC_3=
    (opcode==7'b1100011) &&
    ((funct3==3'b000 && zero)||(funct3==3'b001 && !zero)
    ||(funct3==3'b100 && !zero)||(funct3==3'b101 && zero)
    ||(funct3==3'b110 && !zero) ||(funct3==3'b111 && zero));
    //bltz,bgez,beq,bne,blez,bgtz
    //这里可以简化一下，将所有与zero相与的funct3或起来，将所有与!zero相与的funct3或起来。
funct3的各位之间应该也有逻辑关系，可以不用==表达式。当时为了看起来方便，直接这样写了。

    assign next_PC=(PC_1)?Jump_address1:(PC_2)?Jump_address2:
    (PC_3)?Jump_address3:Jump_address0;
```

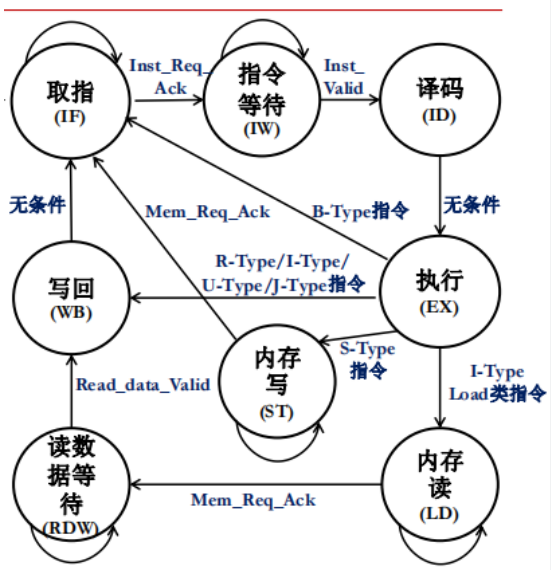
0.3 第三部分是寄存器写和读：需要写寄存器的指令有lui,auipc,jal,jalr,load,I-type计算，R-type计算。写使能有效除了要求译码出来是这些指令之外，还要求处于写回周期，即current_state[8]。写地址一直是rd。读地址一直是rs1和rs2。写寄存器的值分别为：

```
{Instruction_r[31:12]: 12'b0} //lui
PC_s+{Instruction_r[31:12]: 12'b0} //auipc
PC_s+4 //jal or jalr
alu_result //R-type计算 or I-type计算（除去shift指令）
shift_result //shift
```

0.4 第四部分是ALU:有两组ALUop的值，一组对应的是I-type计算和R-type计算，它们的funct3相同可以一起译码（除去一个减法指令，还需要考虑funct7)；另一组对应的是branch指令的寄存器读出的数相比较的操作，可能是slt、sltu、sub三种计算类型。在后面的流水实验中，我把存数和读数的地址计算操作也加到ALU中来了。ALU的操作数A一定是R[rs1]，操作数B除了是R[rs2]之外还有可能是符号位扩展的立即数（对应I-type计算类型指令）。

0.5 第五部分是Memory：逻辑和prj3相似，计算出地址，然后根据地址最低两位n的值，分别取相应字四个字节中的一个或者多个即可。这里的store和load指令的地址是不同，而MIPS中两种指令的地址相同，RSICV中没有lwl这样的指令，实现起来简单很多。不在此赘述。

1.状态机



主体状态图如上所示，省略复位状态。

思路：按照三段式状态机的步骤一步步思考。

第一段状态机：

用always时序逻辑，描述状态寄存器的同步状态跳转。

复位的时候当前状态置为RST状态，其它时间，每一拍都进入下一状态，注意这里的下一状态可能还是原状态，next_state的赋值在第二段状态机中实现。

第二段状态机：

用always组合逻辑，根据状态机当前状态和输入信号，描述下一状态的计算逻辑。

对于取值状态、指令等待状态、内存写状态、内存读状态、读数据等待状态。只有input的Ready或Valid信号有效时，next_state才是真正的下一个状态，否则就停在原状态，等待握手成功。而译码、执行和写回都是一拍内完成的，下一状态不会是原状态。执行时根据四类不同的指令，分别进到取值、写回、内存写、内存读状态。译码状态的下一状态一定是执行状态（没有NOP指令，因为没有分支延迟槽），写回状态的下一状态一直是取值状态，相当于完成了一条指令，进入下一个指令周期。

第三段状态机：

用assign组合逻辑，根据状态机当前状态，描述不同输出寄存器的同步变化。

根据当前的状态决定输出信号的值。注意Inst_Ready除了在指令等待阶段拉高之外，在复位状态也应该拉高；Read_data_Ready除了在读数据等待阶段拉高之外，在复位状态也应该拉高。这样是为了避免访存错误（现有框架无法同时复位MIPS处理器和内存控制器）。另外的三个输出信号，在相应的状态拉高即可。

Instruction寄存器和Read_data寄存器：

完成了三段式状态机的逻辑之后，还需要考虑因多周期带来的读入数据的多变性。比如对于指令Instruction，当指令等待阶段握手成功之后，输入Instruction就是这个指令周期的要用到的指令值，然后进入译码阶段。但是输入值Instruction是变化的，不能保证它在执行完这条指令之前不变化，所以应该将Inst_Ready和Inst_Valid握手成功时的指令值存下来，其它时候，这个存下来的值都保持不变。指令是由内存Memory输出的，那么同理Memory输出的数据Read_data也应该存下来。即在Read_data_Valid和Read_data_ready握手成功时，存下来读入的数据。

接下来考虑PC的值。在现在的状态转移图中，为了取下一条指令，PC是在EX阶段更新的，但是在译码阶段及写回阶段需要用到此指令周期的PC值（即更新前的PC）进行计算和写Register操作。单周期中的PC值，在当前指令的所有操作完成前是不会更改的。为了同样实现这样的逻辑，在这里再加一个寄存器PC_s，存储当前指令周期有效的PC值。PC_s只在取值阶段更新，这样在后续的组合逻辑中都只用这个存下来的PC_s作为操作数，就可以实现和单周期一样的逻辑。

最后将之前写的单周期的组合逻辑的部分，根据需要将Instruction改成Instruction_r,PC改成PC_s,Read_data改成Read_data_r。除此之外，输出的RF_wen信号应该加一个与操作，只有满足写的条件并且处于写回状态，它才可以拉高。

2.终端打印

处理器通过按地址读写端口寄存器，进行写数据操作，将要发送的8-bit字符写入TX FIFO寄存器的低8位。再通过FPGA云节点中的ARM处理器实现终端显示。

uart是UART控制器基地址指针，代表32bit的地址值。每个寄存器具有32-bit位宽，所以相邻寄存器的地址偏移值为4。UART发送数据队列入口寄存器偏移的地址数为4；UART队列状态寄存器偏移的地址数为8。uart+1指向的就是UART发送队列入口寄存器，uart+2指向的就是队列状态寄存器。要存储的是char类型的数据，将uart加1后，应再将uart强制类型转化为指向char类型的指针，相当于取该寄存器的低四

个字节，然后将其值赋值为s[i]即可。

判断发送队列是否达到满状态：用发送队列状态寄存器的值与状态标志位掩码进行按位与操作即可。因为掩码只有第三比特为1，所以与的结果为1当且仅当状态寄存器的第三比特为1，即达到满状态。这个时候应该一直处于while循环内，进行等待。不满时，再进行赋值操作。

```
int puts(const char *s){
    unsigned int i=0;
    while(s[i]!='\0'){
        while((*(uart+2)) & UART_TX_FIFO_FULL);
        *(char*)(uart+1)=s[i];
        i++;
    }
    return i;
}
```

3.性能计数器

在CPU中增加4个性能计数器：处理器运行周期、完成执行的指令数、访存指令数、跳转指令数。

运行周期只需要每个周期加1即可。完成执行的指令数可以在译码阶段加1，因为译码状态只会停留一个周期，若在取值或者指令等待阶段计数，停留周期数可能不止一个，那就会重复计数。对于访存指令计数和跳转指令计数，都是在执行阶段，若处于执行状态并且指令译码符合访存或者跳转指令的特征，则加1。下面是4个计数器的实现。

```
    //处理器运行周期
    reg [31:0] cycle_cnt;
    always @(posedge clk)
    begin
        if(rst==1'b1)
            cycle_cnt <=32'd0;
        else
            cycle_cnt<=cycle_cnt+32'd1;
    end
    assign cpu_perf_cnt_0=cycle_cnt;

    //完成的指令数
    reg [31:0] instruction_cnt;
    always @(posedge clk)
    begin
        if(rst==1'b1)
            instruction_cnt <=32'd0;
        else if(current_state[3])
            instruction_cnt<=instruction_cnt+32'd1;
        else
            instruction_cnt<=instruction_cnt;
    end
    assign cpu_perf_cnt_1=instruction_cnt;

    //访存指令数
    reg [31:0] store_or_load_cnt;
    always @(posedge clk)
    begin
        if(rst==1'b1)
            store_or_load_cnt <=32'd0;
        else if(current_state[4] && (store||load))
            store_or_load_cnt<=store_or_load_cnt+32'd1;
```

```
        else
            store_or_load_cnt<=store_or_load_cnt;
    end
    assign cpu_perf_cnt_2=store_or_load_cnt;

    //跳转指令数
    reg [31:0] pc_change_cnt;
    always @(posedge clk)
    begin
        if(rst==1'b1)
            pc_change_cnt <=32'd0;
        else if(current_state[4] && (PC_1 || PC_2 || PC_3))
            pc_change_cnt<=pc_change_cnt+32'd1;
        else
            pc_change_cnt<=pc_change_cnt;
    end
    assign cpu_perf_cnt_3=pc_change_cnt;
```

在bench.c中访问相应计数器的地址，进行计数，并输出值。总共要添加的就只有三个部分，比较简单。下面是代码：

```
typedef struct Result {
  int pass;
  unsigned long msec;
  unsigned long ins_count;
  unsigned long store_load_count;
  unsigned long bra_jump_count;

} Result;

unsigned long _uptime() {
  // TODO [COD]
  //   You can use this function to access performance counter
or cycle.
  volatile unsigned long *p;
  p=(unsigned long*)0x40020000;

  return *p;

}
unsigned long up_ins(){
  volatile unsigned long *p;
  p=(unsigned long*)0x40020008;
  return *p;
}
unsigned long up_store_load(){
  volatile unsigned long *p;
  p=(unsigned long*)0x40021000;
  return *p;
}
unsigned long up_bra_jump(){
  volatile unsigned long *p;
  p=(unsigned long*)0x40021008;
  return *p;
}
```

```
static void bench_prepare(Result *res) {
  // TODO [COD]
  //   Add preprocess code, record performance counters' initial states.
  //   You can communicate between bench_prepare() and bench_done() through
  //   static variables or add additional fields in `struct Result`
  res->msec = _uptime();
  res->ins_count = up_ins();
  res->store_load_count= up_store_load();
  res->bra_jump_count=up_bra_jump();

}

static void bench_done(Result *res) {
  // TODO [COD]
  //  Add postprocess code, record performance counters' current states.
  res->msec = _uptime() - res->msec;
  res->ins_count = up_ins() - res->ins_count;
  res->store_load_count= up_store_load() - res->store_load_count;
  res->bra_jump_count=up_bra_jump() - res->bra_jump_count;
}
```

```
unsigned long msec = ULONG_MAX;
unsigned long ins_count = ULONG_MAX;
unsigned long store_load_count = ULONG_MAX;
unsigned long bra_jump_count = ULONG_MAX;
   int succ = 1;
   for (int i = 0; i < REPEAT; i ++) {
     Result res;
     run_once(bench, &res);
     printk(res.pass ? "*" : "X");
     succ &= res.pass;
  if (res.msec < msec) msec = res.msec;
     if (res.ins_count < ins_count) ins_count = res.ins_count;
  if (res.store_load_count <store_load_count) store_load_count =
res.store_load_count;
  if (res.bra_jump_count < bra_jump_count) bra_jump_count = res.bra_jump_count;
   }

   if (succ) printk(" Passed.\n");
   else printk(" Failed.\n");

   pass &= succ;

   // TODO [COD]
   //   A benchmark is finished here, you can use printk to output some
information.
   //   `msec' is intended indicate the time (or cycle),
   //   you can ignore according to your performance counters semantics.
  printk("cycle count=%u\n",msec);
  printk("Instrutcion count=%u\n",ins_count);
  printk("store or load count=%u\n",store_load_count);
  printk("branch or jump count=%u\n",bra_jump_count);
}
```

二、 实验过程中遇到的问题、对问题的思考过程及解决方法（比如RTL代码中出现的逻辑bug，仿真、云平台调试过程中的难点等）

RISC-V的指令相比于MIPS更加精简，指令格式也更加规整。有了之前写prj2和prj3的经验，我写完代码之后，第一次push就过了，没有怎么debug。

三、 对讲义中思考题（如有）的理解和回答

对于RSIC-V和MIPS指令集性能分析：经过对FPGA中microbench的结果的比对，发现两者计数器结果的差别不大，基本都在同一个数量级上。但是综合来说RSIC-V指令集在实现相同的功能的前提下，时钟周期数、指令数，访存指令数以及跳转指令数会小于MIPS指令的值。

下面是九个测试函数中的两个，供给参考。

RSIC-V:

```
[fib] Fibonacci number: * Passed.
cycle count=179446877
Instrutcion count=2525790
store or load count=5402
branch or jump count=387765
benchmark finished
```

```
[15pz] A* 15-puzzle search: * Passed.
cycle count=534177860
Instrutcion count=5287779
store or load count=3231816
branch or jump count=631868
benchmark finished
```

MIPS：

```
[fib] Fibonacci number: * Passed.
cycle count=181123869
Instrutcion count=2549568
store or load count=5315
branch or jump count=478091
benchmark finished
```

```
[15pz] A* 15-puzzle search: * Passed.
cycle count=534177860
Instrutcion count=5287779
store or load count=3231816
branch or jump count=631868
benchmark finished
```

四、 在课后，你花费了大约5小时完成此次实验。

五、 对于此次实验的心得、感受和建议（比如实验是否过于简单或复杂，是否缺少了某些你认为重要的信息或参考资料，对实验项目的建议，对提供帮助的同学的感谢，以及其他想与任课老师交流的内容等）

这次的实验基本上就是综合了实验2和实验3，有了之前实验的基础，加上指令集又更加简单，实现起来显得得心应手。我大概查看了RSIC-V指令手册，确定指令的各类格式之后，一个晚上就写完了。主要是在开始动手写代码之前要想明白整体框架和思路，这样之后debug的次数就比较少（毕竟跑一次要这么久），希望之后的选做实验也能这么顺利。

最后感谢助教王嵩岳对我的帮助，在我有疑惑和困难的时候认真耐心地帮助我。