22373321 张瀚文

整体思路

- 单周期处理器包括顶层模块、控制器和数据通路，此为顶层目录，同时有code.txt以存放机器码
- 顶层目录中的每个module都由一个独立的Verilog HDL文件组成，以减小对其他模块的影响和修改，条理更加清晰。
- 首先完成各个模块代码的编写，然后将这些模块统一交给一个顶层的Verilog文件管理：在顶层Verilog文件中定义一些内部的wire型变量，利用模块间的逻辑关系，对各个模块进行**实例化**，串接在一起

相比较P3的更新优化

考虑到添加新指令后PC的计算更加复杂，新增一个NPC模块用于计算PC的更新值。

数据通路的设计

本次需要实现的指令： {add,sub,ori,lw,sw,beq,lui,j,jal,jr}

IM

在ISE中生成匹配需求数量的32位寄存器阵列,注意IM和DM中要从小到大编号 使用$readmemh将十六进制码导入至IM中。

```
reg [31:0] IM[0:4095]; //from low to high
initial begin
    $readmemh("code.txt",IM);
end
```

宏文件

在P3的logisim中，我对指令的判断与匹配主要是通过compartor进行的。在P4的Verilog中，为了防止大量的重复传入传出导致代码的冗余，新增macro.v文件，在其中定义CODE_IDENTIFY及OP_DEFINE，其中

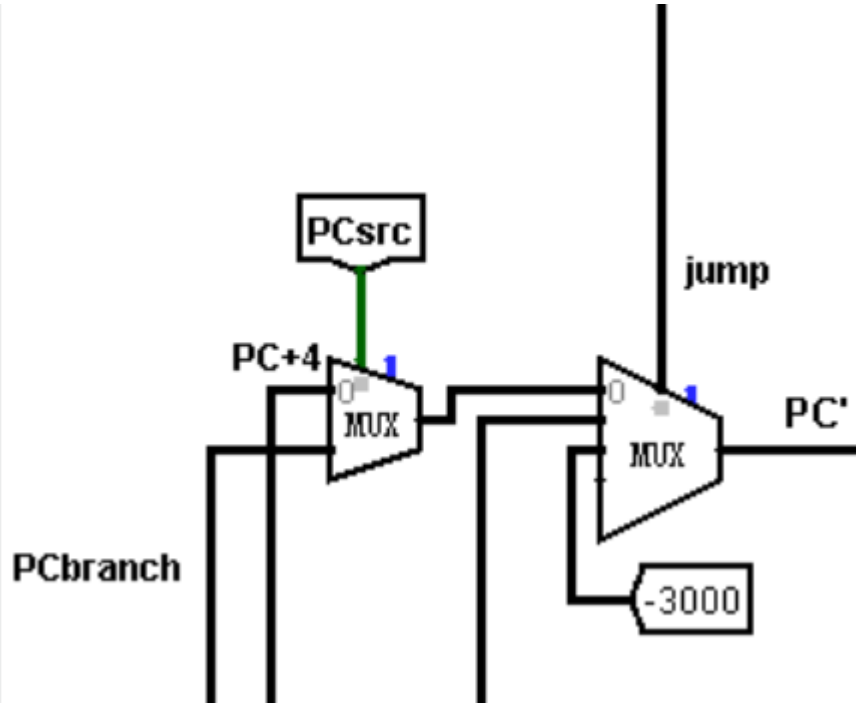
CODE_IDENTIFY主要用于指令的识别，而OP_DEFINE主要用于ALU_op和com_op的定义。

```
`define CODE_IDENTIFY   \
   wire [5:0] op = Instr[31:26]; \
   wire [5:0] funct = Instr[5:0]; \
   /* R_type*/ \
   wire R_type = (op == 6'b000000); \
   wire add = R_type & (funct == 6'b100000); \
   wire sub = R_type & (funct == 6'b100010); \
   wire sll = R_type & (funct == 6'b000000); \
   wire shamt = sll; \
   /* I_type*/ \
   wire ori = (op == 6'b001101); \
   wire addi = (op == 6'b001000); \
   wire lui = (op == 6'b001111); \
   wire addiu = (op == 6'b001001);\
   wire I_type = ori | lui | addiu | addi; \
   /* Write */ \
   wire sw = (op == 6'b101011); \
   wire write = sw; \
   /* Load */ \
   wire lw = (op == 6'b100011); \
   wire load = lw; \
   /* Branch */ \
   wire beq = (op ==6'b000100); \
   wire branch = beq; \
   /* Jump*/ \
   wire jal = (op == 6'b000011); \
   wire jr = R_type & (funct == 6'b001000); \
   wire jump1 = jal; \
   wire jump2 = jr; \
   wire J_write = jal; \
   wire ra = jal;

`define OP_DEFINE   \
   /* ALU_op */ \
   localparam add_op = 5'b00000; \
   localparam addi_op = 5'b00000; \
   localparam addiu_op = 5'b00000; \
   localparam sub_op = 5'b00001; \
   localparam sw_op = 5'b00000; \
   localparam lw_op = 5'b00000; \
   localparam sll_op = 5'b00100; \
   localparam or_op = 5'b00010; \
   localparam lui_op = 5'b00011; \
   /* compare_op */ \
```

Controller

- 新增jump信号，P4相较于P3多出了jal、jr、j指令，于是在原有表格的基础上增添jump信号，考虑到jr和jal不同的PC跳转方式，结合之前的branch跳转，令jump信号为多位信号，结合mux对不同的next_PC进行选择。其logisim的实现方式为：




```
assign jump = {jump2,jump1};
```

对应Verilog语言为：其中jump1表示j和jal信号，jump2表示jr信号。

- 扩展MtR改为D2R信号。新增jal指令后，需要向寄存器中写入除了alu的计算结果和DM中的值的其他值——PC+4，因此将D2R扩展为两位信号。

```
assign D2R = {ra,load};
```

其中ra代表要向第31号寄存器，即$ra写入值，load表示要向寄存器中写入DM中的值。

- 不同控制信号及其所代表的含义如下表所示：

信号名	信号类型	信号含义
ALUcontrol	选择信号	选择ALU进行运算的种类
Imm_ext	选择信号	选择立即数的扩展方式
D2R	选择信号	为00：WD来自ALU；为01：WD来自DM；为10：WD来自PC+4
MW	使能信号	表示是否向内存中写入数据
Bra	选择信号	表示是否是跳转指令
ALUsrcB	选择信号	选择参与运算的srcB来自寄存器还是立即数
ALUsrcA	选择信号	选择参与运算的srcA来自寄存器还是shamt（移位）
RD	选择信号	选择最终要写入的寄存器
RW	使能信号	是否向寄存器堆中写入数据
jump	选择信号	00：PCsrc；01：j/jal类跳转；10：跳转到寄存器中储存的地址

信号名	信号类型	信号含义
PCsrc	选择信号	0：PC+4；1：PCbranch

NPC

对于新增模块NPC，主要用于计算不同情况下PC的更新值

- 情况1：正常后移，PC+4
- 情况2：branch类跳转，立即数×4+PC+4
- 情况3：j/jal类跳转，Instr的低26位和PC的高四位拼接后补两位0
- 情况4：从寄存器中获取
  如图所示：

```
module NPC(
    input [1:0] jump,
    input [31:0] this_pc,
    input [31:0] Imm, //signExtend
    input [31:0] ra, //需要改一下，如果不是从ra寄存器中获取呢
    input [25:0] partInstr,
    input PCsrc, //in main it is input
    output [31:0] next_pc
    );

    /* PC+4 */
    wire [31:0] pcRegular;
    assign pcRegular = this_pc + 4;

    /* PCbranch */
    wire [31:0] pcBranch;
    assign pcBranch = (Imm << 2) + pcRegular;

    /* pcJump */
    wire [31:0] pcJump;
    assign pcJump = {pcRegular[31:28],partInstr,2'b00};

    /* pcBranch or pc+4 */
    wire [31:0] pc1;
    assign pc1 = (PCsrc == 1) ? pcBranch : pcRegular;

    /*final*/
    assign next_pc = (jump == 2'b01) ? pcJump :
                     (jump == 2'b10) ? ra : pc1;

endmodule
```

测试方案

1. 借鉴了往届学长的优秀做法，自行编写了print_infor文件，对输入的指令进行反汇编，将每一步操作进行打印，便于查错。 如图：

```
            if(Instr == 0) begin
               $display("\nI%h: nop",PC);
            end
         end
         if(add) begin
            $display("\nI%h: add $%0d,$%0d,$%0d",PC,rd,rt,rs);
         end
         if(sub) begin
            $display("\nI%h: sub $%0d,$%0d,$%0d",PC,rd,rt,rs);
         end
         if(lw) begin
            $display("\nI%h: lw $%0d,%0d($%0d)",PC,rt,imm,rs);
         end
         if(sw) begin
            $display("\nI%h: sw $%0d,%0d($%0d)",PC,rt,imm,rs);
         end
         if(addi) begin
            $display("\nI%h: addi $%0d,$%0d,%0d",PC,rt,rs,imm);
         end
         if(addiu) begin
            $display("\nI%h: addiu $%0d,$%0d,%0d",PC,rt,rs,imm);
         end
         if(ori) begin
            $display("\nI%h: ori $%0d,$%0d,%0d",PC,rt,rs,imm);
         end
         if(lui) begin
            $display("\nI%h: lui $%0d,%0d",PC,rt,imm);
         end
         if(beq) begin
            $display("\nI%h: beq $%0d,$%0d,%0d",PC,rs,rt,imm);
         end
         if(jal) begin
            $display("\nI%h: jal %h",PC,addr);
         end
         if(jr) begin
            $display("\nI%h: jr $%0d",PC,rs);
         end
         if(branch) begin
            if(compare_condition) begin
               $display("PC <= %h",next_pc);
            end
            else begin
               $display("PC <x %h",next_pc);
            end
```

2. 在testbench中，首先调用print模块，并在最后对DM和GRF中的内容进行打印，以便核对。
3. 第一组测试数据：使用题目中所给的code.txt进行基本性能的测试。
4. 第二组测试数据，对于add，sub等基础运算指令，选用边界数和随机的大数进行测试。

```
.text
ori $a0,$zero,0xFFFD
ori $a1,$zero,0xFFFE #ori测试

lui $a2,0xFFFD
lui $a3,0xFFFE #lui测试

li $t0,-2147483648
li $t1,-2147483647
```

```
li $t2,2147483646
li $t3,2147483645
li $t4,1919156834
li $t5,3233
li $t6,-1000786109
li $t7,-45678  #注意li的实现依托于ori和lui以及可能的addiu

add $s0,$t0,$t2 #+-
add $s1,$t4,$t5 #++
add $s2,$t6,$t7 #--
sub $s2,$t2,$t3 #++
sub $s3,$t5,$t7 #+-
sub $s4,$t7,$t6 #--

li $v0,10
syscall
```

3. 第三组测试数据，对于lw，sw指令，检测当offset和$base为0和负数时能否正确处理。

```
.text
li $s0,10

li $s1,-400 #base -
li $s2,0
li $s3,400

sw $s0,400($s1) # +-
sw $s0,4($s2) #+0
sw $s0,-4($s3) #-+
sw $s0,0($s3) #0+

lw $t0,400($s1) # +-
lw $t0,4($s2) #+0
lw $t0,-4($s3) #-+
lw $t0,0($s3) #0+

li $v0,10
syscall
```

4. 第四组测试数据，针对jal、jr、j指令，检测其能否正确跳转并写入对应的寄存器。

```
.text

li $s0,1
li $s1,2

move $a0,$s0
move $a1,$s1
j add1
```

```
sw $s0,4($zero)
add1:
jal sum
li $v0,10
syscall

sum:
move $t0,$a0
move $t1,$a1
add $t2,$t0,$t1
jr $ra
```

思考题

1. DM的addr信号应由ALU模块计算得到，由于DM按字保存数据，而addr信号以字节为单位，因此截断最后两位。然而此做法不利于后续指令的扩展，因为lb,lh,sh,sb等指令需要addr信号的后两位（倒数第二位）做选择位。
2.
    - 方法一：指令对应的控制信号如何取值，即：

```
always@(*) begin
   case(op)
      `add: begin
         RW = 1;
         ALU_op = 5'b00000;
         D2R = 2'b00;
         ......
      end
      `sub: begin
         RW = 1;
         ALU_op = 5'b00001;
         D2R = 2'b00;
         ......
      end
      ......
end
```

    - 方法二：控制信号每种取值所对应的指令，即：

```
assign ALU_op = (add|addu|addi|addiu|lw|sw) ? 5'b00000 :
sub ? 5'b00001 : ......
```

对比来看，方法一的设计似乎更便于添加新的扩展信号，且代码逻辑更加简洁。 3. + 同步复位优先看clk信号，其次看reset信号，当且仅当clk为上升沿且reset信号为1时才复位.

- 异步复位优先看reset信号，与clk信号并无关系。一旦reset信号有效，则对各部分清零（或者复位）

4.
    - add与addu所执行的运算是一致的，均为将两个寄存器中的值读出相加后写入目标寄存器，唯一不同的就是，add考虑溢出，当二者相加结果溢出时，会触发溢出异常，此时目标寄存器中的值并不会改变，而addu不进行溢出处理，目标寄存器中的值仍会改变。
    - addi与addiu同理，只是其中一个运算数不再从寄存器中读出，而是改为由指令中的立即数参与运算。addi会触发溢出异常，而addiu不会。 因此，在忽略溢出的前提下，addi与addiu，add与addu是等价的。

一些提醒自己的点

- 关于模块实例化，括号里面的是跟这个端口相连的线的名称。
- 务必注意拼接时点名位数，这次搭建主要的bug在于：
    - PCsrc是放在大模块里面的，但是第一次编代码的时候遗漏了
    - {}拼接，不可遗漏位数，比如lui操作，应当写为：`{Imm,{16{1'b0}}}`而不是`{Imm,{16{0}}}`
- 不要再出现类似`wire addr = Address[13:2]`这种情况了！！！
- Markdown写表格的时候跟其他段落相隔开就能识别出来是表格了