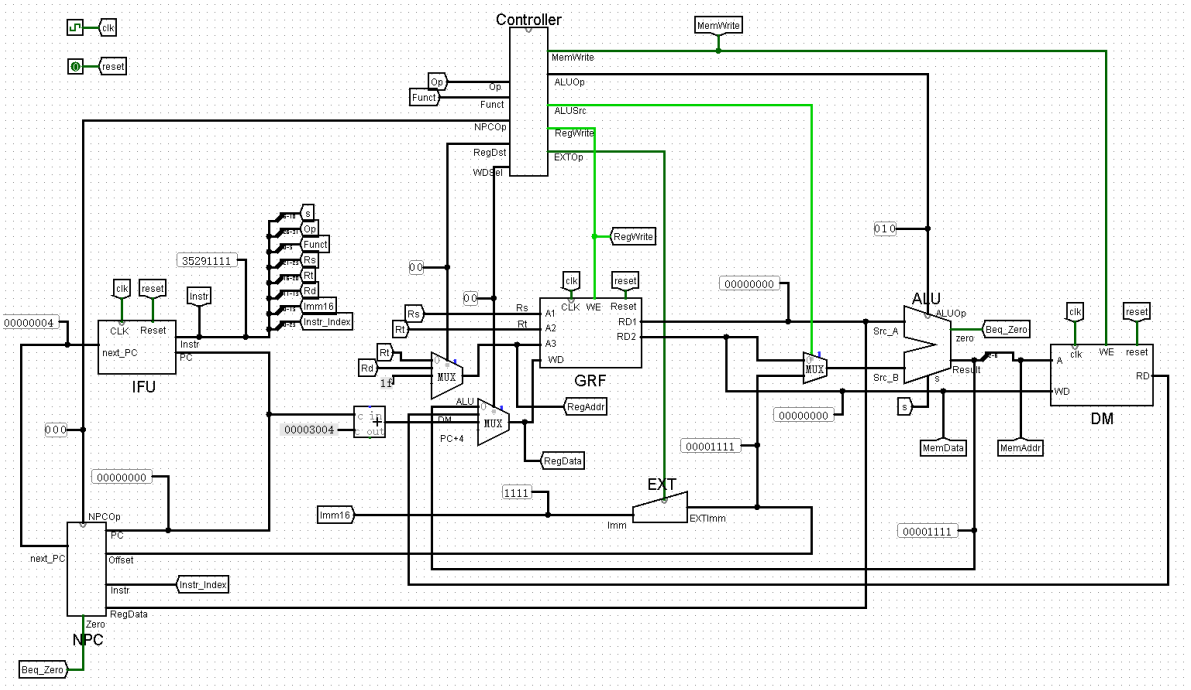
计算机组成P4实验报告

一、设计文档

在P3中，我们采用Logisim直观地搭建单周期CPU，而在P4中，我们需要采用Verilog语言对P3中设计好的CPU进行建模与抽象，以代码的形式实现单周期CPU。本单周期CPU支持的指令为{add,sub,sll,lw,sw,lui,ori,beq,jr,jal}，本部分将分为模块化设计、数据通路以及控制器设计对单周期CPU进行设计。

（1）模块化设计

在本单周期CPU中，我主要设计了以下模块：ALU、DM、EXT、GRF、IFU、NPC。由于本次实验是首次使用Verilog对单周期CPU进行建模，涉及到控制信号的部分会较为复杂，故本人采取先从偏组合逻辑部分开始设计，再逐步设计设计时序逻辑以及控制信号的模块。同时，本人还对P3中各模块的输入输出端口进行了一定优化，增加了部分端口的信号位数以满足更多情况的指令。接下来将分不同模块介绍设计过程。结合P3中已经设计好的单周期CPU数据通路，我对P4中的CPU进行如下模块化设计：



ALU：

输入输出端口设计：

信号名	方向	描述
Src_A [31:0]	I	32位输入信号，待ALU处理的数据之一
Src_B [31:0]	I	32位输入信号，待ALU处理的数据之一
Shamt[4:0]	I	5位输入信号，代表数据的偏移量
ALUOp [3:0]	I	4位输入选择信号，选择ALU需要进行的操作

信号名	方向	描述
Zero	O	输出信号；高电平代表ALU_Result为0，低电平代表其不为0
ALU_Result [31:0]	O	32位输出信号，ALU运算后得到的结果

EXT:

信号名	方向	描述
Imm [15:0]	I	16位输入信号，表示待进行拓展的立即数
EXTOp[1:0]	I	2位输入信号，01：进行符号拓展至32位；00：进行零拓展至32位
EXTImm [31:0]	O	32位输出信号，表示拓展后的立即数

DM:

信号名	方向	描述
clk	I	时钟信号
reset	I	异步复位信号；高电平时将RAM复位，低电平时无效
WE	I	写使能信号，高电平时将WD中数据写入DM中，低电平时无效
Address[4:0]	I	5位输入信号，选择RAM中一个地址单元进行读写操作
DM_WD[31:0]	I	32位输入信号，待输入至DM中的数据
PC [31:0]	I	32位输入信号，用于$display指令
ReadData[31:0]	O	32位输出信号，从DM中读出的数据

GRF:

信号名	方向	描述
clk	I	时钟信号
reset	I	异步复位信号，将32个寄存器的值全部清零
RegWrite	I	写使能信号，高电平时向A3寄存器写入数据WD3；低电平时无效
A1 [4:0]	I	5位输入信号，选择32个寄存器中的一个并将其数据输出至RD1端口
A2 [4:0]	I	5位输入信号，选择32个寄存器中的一个并将其数据输出至RD2端口
A3 [4:0]	I	5位输入信号，选择32个寄存器中的一个作为数据写入的目标寄存器

信号名	方向	描述
GRF_WD[31:0]	I	32位输入信号，当WE3有效时向GRF中写入数据
PC [31:0]	I	32位输入信号，用于$display指令
RD1 [31:0]	O	32位输出信号，输出A1中选择的寄存器中的数据
RD2 [31:0]	O	32位输出信号，输出A2中选择的寄存器中的数据

IFU:

信号名	方向	描述
clk	I	时钟信号
reset	I	异步复位信号，将PC寄存器清零。1：复位；0：无效。
next_PC[31:0]	I	32位输入信号，下一条指令的地址
Instr[31:0]	O	32位输出信号，当前指令的机器码
PC[31:0]	O	32位输出信号，当前指令的地址

NPC:

信号名	方向	描述
PC [31:0]	I	32位输入信号，当前指令的地址
Offset [31:0]	I	32位输入信号，代表地址偏移量
Instr [25:0]	I	26位输入信号，代表j/jal指令的26位地址索引
RegData [31:0]	I	32位输入信号，代表jr指令目标寄存器的值
NPCOp[2:0]	I	下一个PC的选择信号
Zero	I	表示ALU_Result是否为0，用于执行beq操作
next_PC [31:0]	O	32位输出信号，下一个PC的地址

（2）数据通路设计

数据通路的抽象设计基本同P3中logisim电路，关键在于如何从可以连线的电路抽象到用代码代替的Verilog语言之中。本人主要采取以下步骤进行数据通路设计：

1.优化端口名称：不同于Logisim中直观的电路连线，Verilog需要用wire类型的变量代替电线，这就涉及到不同模块间端口名称调用的问题。若不能够很好地命名端口名称，在编写程序时将会出现名称混用的问题。为此，本人对P3中Logisim的CPU中的端口名称进行了一定的改进，比如采用加入模块前缀的方式以区分不同模块的端口信号，将DM与GRF的写入数据端口名称分别命名为“DM_WD”与“GRF_WD”以区分不同的端口信号，方便DataPath.v中对不同模块的调用。

2.声明各模块的“输出端”：为简化各模块间连线的复杂度，需要采用一个统一的wire类型声明标准。本人采用的是声明不同模块的输出变量（如下图所示），可以理解为先将实际中的每个模块的输出端用到导线连出，然后再将这些带有连出导线的模块进行组合，以防止出现数据线混用的情况。

```
//ALU output
wire [31:0] ALU_Result;
wire Zero;

//DM output
wire [31:0] ReadData;

//EXT output
wire [31:0] EXTImm;

//GRF output
wire [31:0] RD1,RD2;

//NPC output
wire [31:0] next_PC;
```

3.在数据通路中连接Mux选择器

在基本引出好各模块的输出信号后，下一步就是将需要在数据通路中进行输出的整合。将需要采用Mux进行选择的信号用assign+三目赋值的语句进行整合为一个输出。

```
//MUX:
//RegDst
wire [4:0] Mux1_output;

assign Mux1_output=(RegDst_Sel==2'b00)?Rt:
                   (RegDst_Sel==2'b01)?Rd:
                   (RegDst_Sel==2'b10)?5'h1f:Rt;
//RegWriteData
wire [31:0] Mux2_output;
assign Mux2_output=(GRFWD_Sel==2'b00)?ALU_Result:
                   (GRFWD_Sel==2'b01)?ReadData:
                   (GRFWD_Sel==2'b10)?(PC+32'd4):ALU_Result;
//Src_B
wire [31:0] Mux3_output;
assign Mux3_output=(ALUSrc_Sel==0)?RD2:EXTImm;
```

4.实例化模块并将线连入

由于前面已经将所有模块需要输出的信号用wire定义好，故先对每个模块的输出端口进行修改，然后再将每个模块对应的输入连接即可。

```
//ALU
ALU alu (
    .Src_A(RD1),
    .Src_B(Mux3_output),
    .Shamt(Shamt),
    .ALUOp(ALUOp),
    .Zero(Zero),
    .ALU_Result(ALU_Result)
    );

//DM
DM dm (
    .PC(PC),
    .address(ALU_Result),
    .DM_WD(RD2),
    .MemWrite(MemWrite),
    .clk(clk),
    .reset(reset),
    .ReadData(ReadData)
    );
```

（3）控制器设计

1.宏处理

不同于Logisim，我们在Verilog中无需按位判断指令的类型。我采取的方案是在模块最前面分为R类型指令与非R类型指令，对于R类型指令，宏定义的6位为Funct的值；对于非R类型指令，宏定义的6位为其Op值。

```
//R类型指令 funct
`define add 6'b100000
`define sub 6'b100010
`define jr  6'b001000
`define sll 6'b000000

`define R_type 6'b000000


//I类型与J类型 Op
`define lw  6'b100011
`define sw  6'b101011
`define ori 6'b001101
`define lui 6'b001111
`define beq 6'b000100
`define j   6'b000010
`define jal 6'b000011
```

2.控制信号的生成

本次CPU设计采取与直接从Logisim中映射的方式进行设计，即“控制信号每种取值所对应的指令”。具体实现采用assign+三目运算符进行实现即可。

```
assign ALUOp=(Op==`R_type&&Funct==`add)? 4'b0000:
             (Op==`R_type&&Funct==`sub)? 4'b0001:
             (Op==`R_type&&Funct==`sll)? 4'b0100:
             (Op==`lw)?  4'b0000:
             (Op==`sw)?  4'b0000:
             (Op==`ori)? 4'b0010:
             (Op==`lui)? 4'b0011:
             (Op==`beq)? 4'b0001:4'b0000;
```

控制信号输出表格：

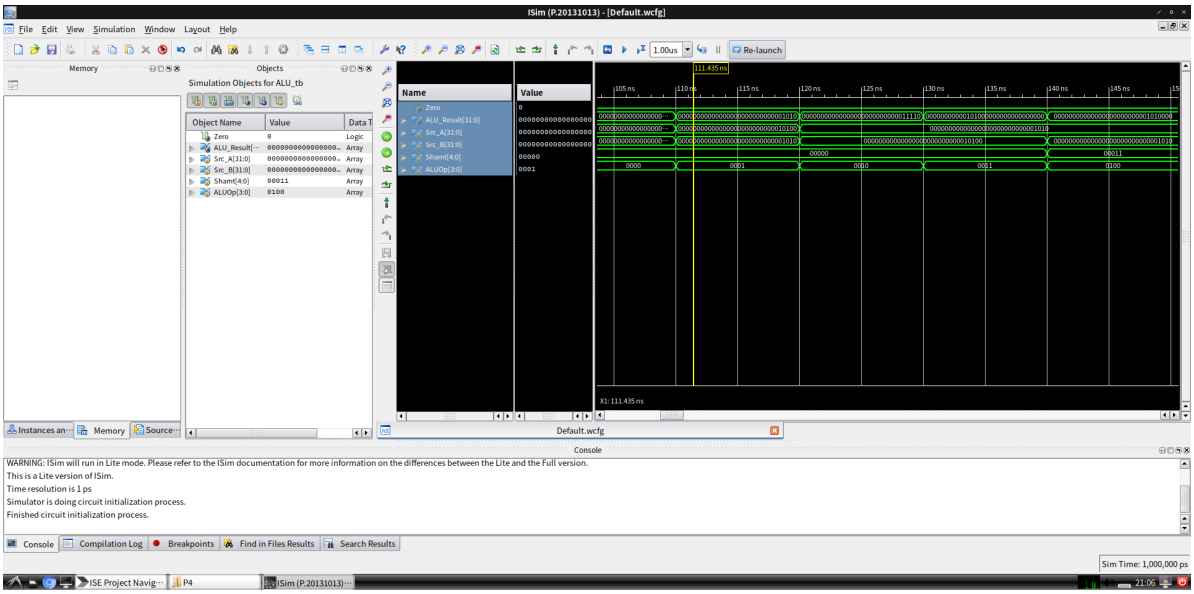
指令	add	sub	sll	lw	sw	ori	lui	beq	j	jal	jr
MemWrite	0	0	0	0	1	0	0	0	0	0	0
RegWrite	1	1	1	1	0	1	1	0	0	1	0
NPCOp1	0	0	0	0	0	0	0	0	1	1	1
NPCOp0	0	0	0	0	0	0	0	1	0	0	1
ALUOP2	0	0	1	0	0	0	0	0	x	x	x
ALUOP1	0	0	0	0	0	1	1	0	x	x	x
ALUOP0	0	1	0	0	0	0	1	1	x	x	x
EXTOp	x	x	x	1	1	0	0	x	x	x	x
ALUSrc_Sel	0	0	0	1	1	1	1	0	x	x	x
GRFWD_Sel1	0	0	0	0	x	0	0	x	x	1	x
GRFWD_Sel0	0	0	0	1	x	0	0	x	x	0	x
RegDst_Sel1	0	0	0	0	x	0	0	x	x	1	x
RegDst_Sel0	1	1	1	0	x	0	0	x	x	0	x

二、测试方案

不同于以往Verilog程序的测试，单周期CPU涉及到了数据通路与各模块、数据通路与控制器等之间的关系，情况较为复杂。我采取的思路是：从底部向顶层进行测试，先用TestBench验证局部部件的正确性,再通过构建测试数据来测试数据通路与控制器的正确性。

（1）TestBench测试

以ALU为例，本人编写了ALU_tb.v对ALU模块的准确性



（2）自主设计样例测试

在初步设计完成CPU后，主要分为计算类（ori,lui,add,sub）、存取类（sw,lw）以及跳转类（beq, j, jal, jr）指令进行测试。

计算类

```
.text

####先进行ori\lui的测试
#0附近的数字
ori $a0, $0, 0
ori $a1, $0, 1
lui $t1, 0
lui $t2, 1

#16位无符号边界
ori $t3, $0, 65534
ori $t4, $0,65535
lui $t5, 65534
lui $t6,65535

#随机数
ori $t7,$0,34322
lui $t8,5346

#目标寄存器为0
ori $0,$0,2131
lui $0,5435

####测试add\sub
add $s1,$t0,$t2
add $s2,$t4,$t6
sub $s3,$s2,$t4
sub $s4,$s3,$s3
```

```
####测试寄存器的极限值
lui $s5,32767
ori $s5,65535   #t1=0x01111111_11111111_11111111_11111111
lui $s6,65535
ori $s6,65535   #t2=0x11111111_11111111_11111111_11111111
```

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000001
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00010000
$t3	11	0x0000fffe
$t4	12	0x0000ffff
$t5	13	0x7fff0000
$t6	14	0x7ffe0000
$t7	15	0x00008612
$s0	16	0x00000000
$s1	17	0x00010000
$s2	18	0x7fefffff
$s3	19	0x7ffe0000
$s4	20	0x00000000
$s5	21	0x7fffffff
$s6	22	0xffffffff
$s7	23	0x00000000
$t8	24	0x14e20000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x00001800
$sp	29	0x00002ffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00003050
hi		0x00000000
lo		0x00000000

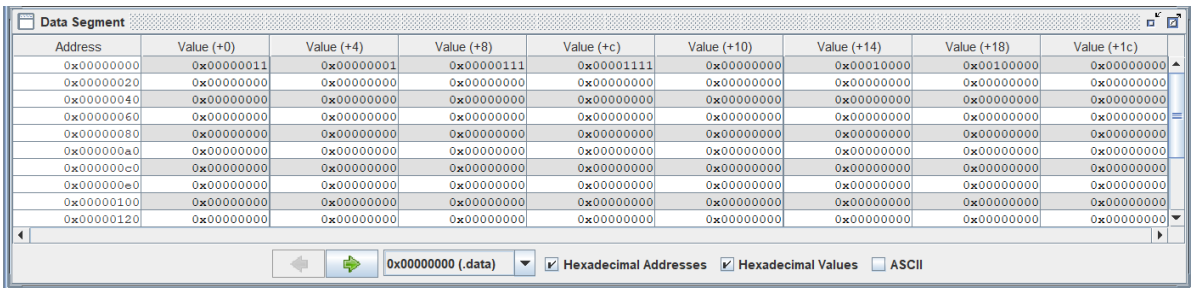
	0	1
0x1F	00000000	00000000
0x1D	00000000	00000000
0x1B	00000000	00000000
0x19	00000000	14E20000
0x17	00000000	FFFFFFFF
0x15	7FFFFFFF	00000000
0x13	7FFE0000	7FFEFFFF
0x11	00010000	00000000
0xF	00008612	7FFE0000
0xD	7FFF0000	0000FFFF
0xB	0000FFFE	00010000
0x9	00000000	00000000
0x7	00000000	00000000
0x5	00000001	00000000
0x3	00000000	00000000

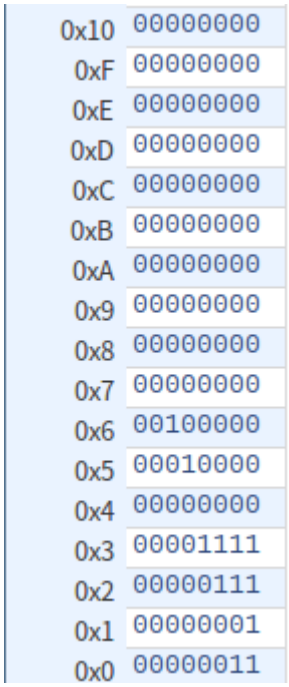
0x1	00000000	00000000

经比较，MARS中寄存器结果与Isim中Memory寄存器结果一致，故通过该测试。

存取类

```
.text`
`####测试存取指令`
`lui $t0,65535`
`ori $t0,$t0,65516  #t0 is -20`
`##base<0`
    `#offset>0`
    `ori $s1,$s1,0x00000001`
    `sw $s1,24($t0) #4 is s1`
    `lw $s2,24($t0)`
##base=0
  #offset=0
  ori $s1,$s1,0x00000010    #0 is s1
  sw $s1,0($0)
  lw $s3,0($0)
  #offset>0
  ori $s1,$s1,0x00000100
  sw $s1,8($0)  #8 is s1
  lw $s4,8($0)
##base>0
  ori $t1,$t1,20
  #offset<0
  ori $s1,$s1,0x00001000
  sw $s1,-8($t1)
  lw $s5,-8($t1)
  #offset=0
  lui $s1,0x0001
  sw $s1,0($t1)
  lw $s6,0($t1)
  #offset>0
  lui $s1,0x0010
  sw $s1,4($t1)
  lw $s7,4($t1)`
```





经比较，MARS中.data字段结果与Isim中Memory寄存器结果一致，故通过该测试。

跳转类

```
##j指令测试
ori $t1,$t1,4369
j test1

ori $t2,$t2,4369
test1:          #j:向前跳转测试

ori $t3,$t3,4369
j test2

test3:          #j：向后跳转测试
ori $t5,$t5,4369
j end1

test2:
ori $t4,$t4,4369
j test3

end1:
ori $t6,$t6,4369

##jal指令测试
jal test4
ori $s7,$0,0x1111
j beq_test
##jr指令测试
test4:
add $t7,$ra,$0  #将ra的值存到t7之中
jr $ra
```

```
beq_test:
ori $t8,$t8,0x1111
ori $t9,$t9,0x1111
j branch1
##beq指令测试

#跳转且目标在指令之前
branch1_test:
ori $s0,$s0,0x1010
j branch3

branch1:

beq $t8,$t9,branch1_test
ori $s0,$s0,0x1111
j branch3

#跳转且目标就是这条指令
#branch2:
#beq $t8,$t9,branch2
#ori $s1,0x1010
#j branch3

branch3:
#跳转且目标在指令之后
beq $t8,$t9,branch3_test
ori $s2,$s2,0x1111
j branch4
branch3_test:
ori $s2,$s2,0x1010
j branch4

#不跳转且目标在指令之前
branch4_test:
ori $s3,$s3,0x1010
j branch5

branch4:
add $t8,$0,$0
ori $t8,$t8,0x111
beq $t8,$t9,branch4_test
ori $s3,$s3,0x1111
j branch5

branch5:
#不跳转且目标就是这条指令
beq $t8,$t9,branch5
ori $s4,$s4,0x1111
j branch6
```

```
branch6:
#不跳转且目标在指令之后
beq $t8,$t9,branch6_test
ori $s5,$s5,0x1111
j end

branch6_test:
ori $s5,$s5,0x1010
j end

end:
```

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00001111
$t2	10	0x00000000
$t3	11	0x00001111
$t4	12	0x00001111
$t5	13	0x00001111
$t6	14	0x00001111
$t7	15	0x0000302c
$s0	16	0x00001010
$s1	17	0x00000000
$s2	18	0x00001010
$s3	19	0x00001111
$s4	20	0x00001111
$s5	21	0x00001111
$s6	22	0x00000000
$s7	23	0x00001111
$t8	24	0x00000111
$t9	25	0x00001111
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x00001800
$sp	29	0x00002ffc
$fp	30	0x00000000
$ra	31	0x0000302c
pc		0x000030ac
hi		0x00000000
lo		0x00000000

经比较，MARS中.寄存器结果与Isim中Memory寄存器结果一致，故通过该测试。

针对上述三个测试方案，通过验证Isim 界面中的Memory与Mars的输出结果，本单周期CPU的输出结果均与Mars结果一致，故本CPU通过自主构建数据测试。

（3）采用自动化测试

在本部分中，本人采用了由周伯阳同学在cscore网站上提供的**基于iverilog的半自动测试和自动化测试**测试方案（网址链接：http://cscore.buaa.edu.cn/#/discussion_area/690/818/posts），多次进行随机数据生成来进一步验证单周期CPU的正确性。

测试结果如下图所示：

```
C:\Users\HaojunYan\Desktop\P4_test\P4_test_2.0\P4_test>python generate.py

C:\Users\HaojunYan\Desktop\P4_test\P4_test_2.0\P4_test>python setup.py

C:\Users\HaojunYan\Desktop\P4_test\P4_test_2.0\P4_test>call getCompile.bat

C:\Users\HaojunYan\Desktop\P4_test\P4_test_2.0\P4_test>cd src

C:\Users\HaojunYan\Desktop\P4_test\P4_test_2.0\P4_test\src>iverilog -o mips_tb.v.out mips_tb.v

C:\Users\HaojunYan\Desktop\P4_test\P4_test_2.0\P4_test\src>vvp mips_tb.v.out  1>out.txt

C:\Users\HaojunYan\Desktop\P4_test\P4_test_2.0\P4_test\src>cd ..

C:\Users\HaojunYan\Desktop\P4_test\P4_test_2.0\P4_test>python comp.py
输出到：  file:///C:/Users/HaojunYan/Desktop/P4_test/P4_test_2.0/P4_test/diff_result.html

C:\Users\HaojunYan\Desktop\P4_test\P4_test_2.0\P4_test>fc out.txt std.txt
正在比较文件 out.txt 和 STD.TXT
FC: 找不到差异
```

```
 1@00003000: $29 <= 000022fc
 2@00003004: $28 <= 00001800
 3@00003008: $31 <= 00003000
 4@0000300c: $31 <= 00003010
 5@0000308c: $31 <= 00003090
 6@00003320: $31 <= 00003324
 7@000033ac: $ 6 <= 00000000
 8@000033b0: $21 <= 000001c4
 9@000033b4: *000001d0 <= 00000000
10@000033b8: $22 <= 000001fc
11@000033bc: *00000218 <= 00000000
```


```
 1@00003000: $29 <= 000022fc
 2@00003004: $28 <= 00001800
 3@00003008: $31 <= 00003000
 4@0000300c: $31 <= 00003010
 5@0000308c: $31 <= 00003090
 6@00003320: $31 <= 00003324
 7@000033ac: $ 6 <= 00000000
 8@000033b0: $21 <= 000001c4
 9@000033b4: *000001d0 <= 00000000
10@000033b8: $22 <= 000001fc
11@000033bc: *00000218 <= 00000000
```


经过多次随机数据测试，本单周期CPU均通过了数据匹配。至此，可以认定本单周期CPU具有正确性。

三、思考题

1. 阅读下面给出的 DM 的输入示例中（示例 DM 容量为 4KB，即 32bit × 1024字），根据你的理解回答，这个 addr 信号又是从哪里来的？地址信号 addr 位数为什么是 [11:2] 而不是 [9:0]？

文件	模块接口定义

dm.v	dm(clk,reset,MemWrite,addr,din,dout);
	input clk; //clock
	input reset; //reset
	input MemWrite; //memory write enable
	input [11:2] addr; //memory's address for write
	input [31:0] din; //write data
	output [31:0] dout; //read data

（1）addr信号来源：通过对单周期CPU的数据通路以及sw等存储类指令的RTL语言进行分析可知，addr信号来源应为ALU模块的运算结果，而ALU模块内部则是对从GRF中的RD1以及进行拓展后的32位立即数进行相加操作。

（2）地址位数为[11:2]而不是[9:0]的原因：在logisim中，DM采用RAM部件进行数据存储，而在Verilog中，我们采用reg类型变量来存储相关的数据。但本质上，我们都是采取了"按字选址"的方式，故需要舍弃addr信号的后两位，而采取addr的[11:2]。但是在处理lw、sw、lb、sb等指令时，我们需要采利用addr的后两位作为选择信号来进行数据的读写。

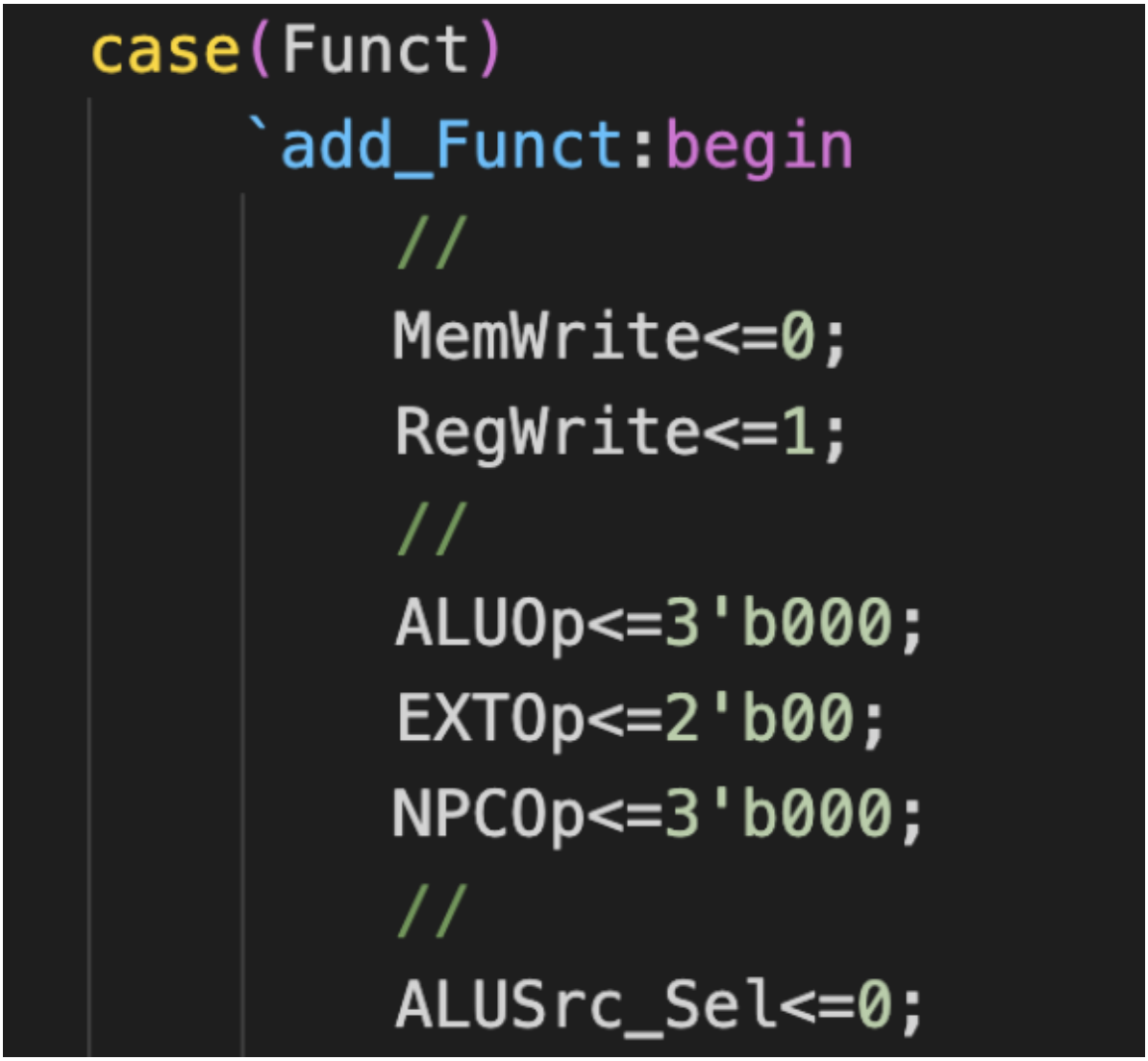
2.思考上述两种控制器设计的译码方式，给出代码示例，并尝试对比各方式的优劣。

代码示例：

1.**控制信号每种取值所对应的指令**

```
assign ALUOp=(Op==`R_type&&Funct==`add)? 4'b0000:
             (Op==`R_type&&Funct==`sub)? 4'b0001:
             (Op==`R_type&&Funct==`sll)? 4'b0100:
             (Op==`lw)?  4'b0000:
             (Op==`sw)?  4'b0000:
             (Op==`ori)? 4'b0010:
             (Op==`lui)? 4'b0011:
             (Op==`beq)? 4'b0001:4'b0000;
```

2.**指令对应的控制信号如何取值**



控制信号每种取值所对应的指令：

优势：符合Logisim以及实际连接电路时的逻辑，较容易从电路图中向Verilog代码进行映射。

劣势：由于控制信号代码相对分散，新增指令可能会遗漏考虑添加控制信号导致结果出错。

指令对应的控制信号如何取值:

优势：单条指令中的代码块中可以对各个控制信号进行充分的考虑，减少新增指令时遗漏的风险。

劣势：若需要修改控制信号，需要对每一个指令中的代码均进行修改，不利于后期控制信号的删减或修改。

3.在相应的部件中，复位信号的设计都是**同步复位**，这与 P3 中的设计要求不同。请对比**同步复位**与**异步复位**这两种方式的 reset 信号与 clk 信号优先级的关系。

同步复位:

在同步复位设计中，需要在clk信号上升沿来临的时候，才会对reset信号进行判断是否需要复位，故clk对信号优先级高于reset信号。

异步复位:

在异步复位设计中，reset信号发生变化（假设是posedge发生复位），此时将立即进行复位操作，而无需考虑clk信号的变化，故reset信号的优先级高于clk信号。

4.C 语言是一种弱类型程序设计语言。C 语言中不对计算结果溢出进行处理，这意味着 C 语言要求程序员必须很清楚计算结果是否会导致溢出。因此，如果仅仅支持 C 语言，MIPS 指令的所有计算指令均可以忽略溢出。 请说明为什么在忽略溢出的前提下，addi 与 addiu 是等价的，add 与 addu 是等价的。提示：阅读《MIPS32® Architecture For Programmers Volume II: The MIPS32® Instruction Set》中相关指令的 Operation 部分。

因为在数据通路部分，addi与addiu，add与addu是完全相同的，区别在于ALU模块中是否需要对两个源操作数进行溢出判断。在C语言中，我们不需要对计算结果溢出进行处理，意味着我们无需在ALU中判断是否有溢出发生，故二者是完全等价的。