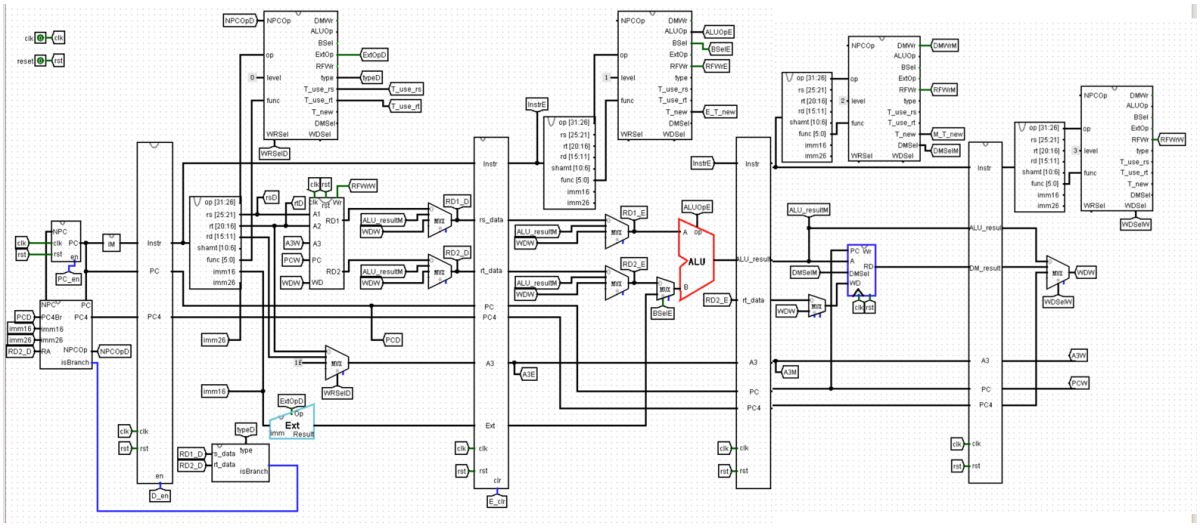
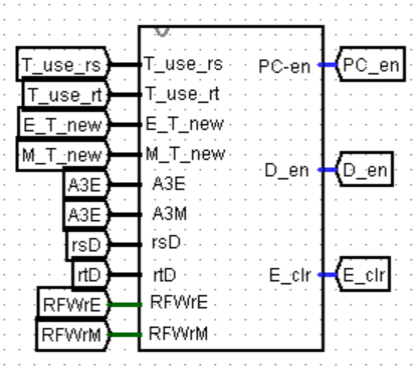
一、设计草稿

CPU顶层架构模型





(一) IFU

根据PC从ROM中取出指令。由于每执行一条指令后，PC的值会随之更新，所以需要设计一个FSM实现PC的状态转移。由此，我们将IFU细分为PC（状态存储）、NPC（次态逻辑）、IM（取指令）三个模块

PC:

端口名称	I/O	位宽	功能
NPC	I	32	输入寄存器的次态PC值
clk	I	1	时钟信号
rst	I	1	同步复位信号
PC	O	32	当前的PC值

```
module _PC(
    input [31:0] NextPC,
    input clk,
    input rst,
    output reg [31:0] nowPC
    );
```

```
always @(posedge clk )begin
    if(rst)begin
        nowPC<=32'h3000;
    end
    else begin
        nowPC<=NextPC;
    end
end
endmodule
```

- **IM:**

  需要注意PC和ROM中指令地址的映射关系

  $addr = (PC - 0x00003000)/4$

端口名称	I/O	位宽	功能
PC	I	32	当前的PC值
instruction	O	32	取出的指令

```
module _IM(
    input [31:0] PC,
    output [31:0] instruction
    );
reg [31:0] Rom [0:4095];

initial begin
        $readmemh("p4_testcode.txt", Rom);
end

wire [11:0] truPC = (PC-32'h3000)/4;
assign instruction = Rom[truPC];

endmodule
```

- **NPC:**

  对于PC的次态逻辑，根据指令的类型，有以下四种情况：

  - 通常情况下，如add、lw、sw等指令，$PC = PC + 4$
  - 分支指令，如beq、bne等指令，$PC = PC + 4 + signExtend(offset||0^2)$，即需要和一个符号扩展后的32位立即数运算
  - 跳转指令，如j、jal指令，$PC = (PC[31:28]||instrIndex||0^2)$，也是需要和一个符号扩展后的32位立即数运算，同时jal指令执行时，会将当前$PC + 4$的值储存在寄存器`$ra`当中
  - 跳转并链接指令，如jr、jalr指令，$PC = GPR[rs]$

端口名称	I/O	位宽	功能
PC	I	32	当前PC

端口名称	I/O	位宽	功能
Op	I	2	选择信号 00 : 计算顺序地址 01 : 计算beq地址 10 : 计算jr地址 11 : 计算jal地址
imm	I	32	参与计算的立即数（已符号扩展为32位）
$ra	I	32	jr所跳转的寄存器
PC4	O	32	PC + 4
nextPC	O	32	次态PC

```
module _NPC(
    input [31:0] PC,
    input [1:0] op,
    input [15:0] imm16,
     input [25:0] imm26,
     input [31:0] PC4Br,
    input [31:0] ra,
     input isBranch,
    output [31:0] PC4,
    output [31:0] NextPC
    );

assign PC4 = PC + 8;
wire [31:0]sign_imm = {{16{imm16[15]}},imm16};
wire [31:0] bnext = PC4Br + 4 +(sign_imm<<2);
wire [31:0] jnext = {PC[31:28],imm26,{2'b00}};

assign NextPC = (op==0) ? (PC+4) :
                    (op==1 && isBranch==1) ? (bnext):
                    (op==2) ? (ra):
                    (op==3) ? jnext:
                    PC+4;

endmodule
```

（二）Split

利用分位器分出指令的各部分编码，封装在模块中，使电路更整洁

```
module _Split(
    input [31:0] Instr,
    output [5:0] op,
    output [4:0] rs,
    output [4:0] rt,
    output [4:0] rd,
```

```
    output [4:0] shamt,
    output [5:0] func,
    output [15:0] imm
    );

assign op = Instr[31:26];
assign rs = Instr[25:21];
assign rt = Instr[20:16];
assign rd = Instr[15:11];
assign shamt = Instr[10:6];
assign func = Instr[5:0];
assign imm = Instr[15:0];


endmodule
```

(三) GRF

已在P0课下时实现

端口名称	I/O	位宽	功能
clk	I	1	时钟信号
rst	I	1	同步复位信号
Wr	I	1	写使能信号
A1	I	5	5位地址读取信号
A2	I	5	5位地址读取信号
A3	I	5	5位地址写入信号，指定一个寄存器作为写入数据的目标
WD	I	32	32位数据写入信号
WPC	I	32	当前的PC值
RD1	O	32	输出A1指定的寄存器中的32位数据
RD2	O	32	输出A1指定的寄存器中的32位数据

```
module _GRF(
    input clk,
    input rst,
    input Wr,
    input [4:0] A1,
    input [4:0] A2,
    input [4:0] A3,
    input [31:0] WD,
     input [31:0] WPC,
    output  [31:0] RD1,
    output  [31:0] RD2
    );
```

```
reg [31:0] register[0:31];
assign RD1 = register[A1];
assign RD2 = register[A2];

integer i;
always @(posedge clk)begin
    if(rst)begin
        for(i=0;i<32;i=i+1)begin
            register[i]<=0;
        end
    end
    else begin
        if(Wr==1&&A3!=0)begin
            register[A3]<=WD;
            $display("@%h: $%d <= %h", WPC, A3,WD);
        end
    end
end

endmodule
```

(四) ALU

此模块主要是实现寄存器和立即数之间的运算。下表是本次实验的指令所涉及的运算

指令	运算
add	加（不考虑溢出）
lw，sw	加（寄存器加立即数得到内存地址）
sub	减（不考虑溢出）
ori	或
lui	左移（立即数加载至高位）
beq	判断是否相等

beq比较特殊，其不需要写寄存器或内存，且运算结果为0或1，我们可将其运算结果单独作为模块的一个输出；而剩下的运算结果用多路选择器输出。

端口名称	I/O	位宽	功能
A	I	32	输入运算数A
B	I	32	输入运算数B

端口名称	I/O	位宽	功能
Op	I	2（后续实验可能会扩展）	选择信号 00 : A + B 01 : A - B 10 : A | B 11 : B<<16
Result	O	32	运算结果
equal	O	1	0： A != B 1 : A == B

```
module _ALU(
    input [31:0] A,
    input [31:0] B,
    input [2:0] Op,
    output [31:0] Result,
    output equal,
    output greater
    );

assign equal = (A==B);
assign greater = (A>B);          //无符号

assign Result =     (Op==0) ? (A+B):
                        (Op==1) ? (A-B):
                        (Op==2) ? (A|B):
                        (Op==3) ?(B<<16):
                        0;
endmodule
```

(五) Ext

端口名称	I/O	位宽	功能
imm	I	16	16位立即数
Op	I	1	0： 0扩展 1： 符号扩展
Result	O	32	扩展结果

```
module _Ext(
    input [15:0] imm,
    input [0:0] Op,
    output [31:0] Result
    );

assign Result[15:0]=imm;
assign Result[31:16]= (Op==0) ? {16{1'b0}} : {16{imm[15]}};

endmodule
```

(六) DM

端口名称	I/O	位宽	功能
Addr	I	32	待操作的内存地址
WD	I	32	写入内存的数据
DMWr	I	1	写使能信号
DMSel	I	2	读写方式 00 : 字 01 : 半字 10 : 字节
clk	I	1	时钟信号
rst	I	1	同步复位信号
pc	I	32	当前pc值
RD	O	32	Addr中储存的数据

需要注意Addr和RAM内存中地址的映射关系

```
module _DM(
    input [31:0] Addr,
    input [31:0] WD,
    input DMWr,
     input [1:0] DMSel,
    input clk,
    input rst,
     input [31:0]pc,
    output [31:0] RD
    );
reg [31:0] dm[0:3071];
wire [11:0]truAdr = Addr/4;
assign RD =  dm[truAdr];
```

```
integer i;
always @(posedge clk)begin
    if(rst)begin
        for (i = 0; i < 3072; i = i + 1) begin
            dm[i] <= 32'h0;
        end
    end
    else begin
        if (DMWr == 1) begin
            dm[truAdr] <= WD;
            $display("@%h: *%h <= %h", pc, Addr,WD);
        end
    end
end

endmodule
```

（七）数据通路

至此已经完成了 $Controller$ 外各个模块的搭建，下面我们要结合控制信号，完成顶层电路的连接，最后再实现 $Controller$ 模块

- $Split$ 和 GRF 的连接

  首先设置**RFWr**信号选择是否执行写寄存器操作

  读写寄存器，需要仔细研究相关指令的RTL语言

  rs [25 : 21] 和rt [20 : 16] 总是被读取的寄存器，默认依次接在AD1和AD2端口；

  被写入的寄存器，根据指令类型，或是rt [20 : 16] ，或是rd [15 : 11]，于是设置**WRSel**信号进行选择；

  被写入的数据，根据指令类型，或是ALU运算结果，或是RAM内存中的数据，或是PC4，于是设置**WDSel**进行选择。

- $Split$ 和 Ext 的连接

  将 imm [15 : 0] 接入Ext 输入端口

  设置**EXTOp**信号选择扩展方式

- GRF 和 ALU 的连接

  首先设置**ALUOp**信号选择是否执行运算操作

  运算的对象，依据指令的不同，或是rs [25 : 21] 和rt [20 : 16] 运算，或是rs [25 : 21] 和 imm [15 : 0] 扩展后运算，或是 imm [15 : 0] 左移运算，默认将rs [25 : 21] 接到A端口，于是设置**BSel**信号对接入B端口的数据进行选择

- GRF、ALU 和 DM 的连接

  加载和存储操作

  首先设置**DMWr**信号选择是否执行存储操作

存储到内存的总是 rt [20 : 16] 寄存器中的数据，因此将 $GRF.RD2$ 端口与 $DM.WD$ 端口相连；

待操作的内存地址由 ALU 计算得到，因此将 ALU输出接到 $DM.A$ 端口

整个顶层电路如下：

```
module mips(
    input clk,
    input reset
    );

//控制信号
wire [1:0]NPCOp;
wire [1:0]WRSel;
wire [1:0]WDSel;
wire [2:0]ALUOp;
wire [1:0]DMSel;
wire DMWr;
wire EXTOp;
wire BSel;
wire RFWr;
wire equal;
wire greater;

//IFU
wire [31:0]NextPC;
wire [31:0]pc;
wire [31:0]PC4;
wire [31:0]Instr;
wire [5:0]op;
wire [5:0]func;
wire [4:0]rs;
wire [4:0]rt;
wire [4:0]rd;
wire [4:0]shamt;
wire [15:0]imm_16;
wire [4:0]A1=rs;
wire [4:0]A2=rt;
wire [4:0]A3;
wire [31:0]RD1;
wire [31:0]RD2;
wire [31:0]GRFWD;
wire IsJ;
wire [31:0]B;
wire [31:0]imm_ext;
wire [31:0]ALU_result;
wire [31:0]RD_result;

wire [31:0]imm_32= (IsJ==1) ? Instr : imm_ext;

assign A3 = (WRSel==0) ?  rt:
      (WRSel==1) ?  rd:
      (WRSel==2) ?  31:
```

```
      0;

assign GRFWD = (WDSel==0) ? ALU_result:
           (WDSel==1) ? RD_result:
           (WDSel==2) ? PC4:
           0;

assign B = (BSel==0) ? RD2 : imm_ext;

_CTRL
CTRL(op,func,equal,greater,RFWr,WRSel,WDSel,EXTOp,BSel,ALUOp,DMWr,DMSel,NPCOp);
_PC PC(NextPC,clk,reset,pc);
_NPC NPC(pc,NPCOp,imm_ext,RD1,PC4,NextPC);
_IM IM(pc,Instr);
_Split Split(Instr,op,rs,rt,rd,shamt,func,imm_16);
_GRF GRF(clk,reset,RFWr,A1,A2,A3,GRFWD,pc,RD1,RD2);
_Ext Ext(imm_16,EXTOp,imm_ext);
_ALU ALU(RD1,B,ALUOp,ALU_result,equal,greater);
_DM DM(ALU_result,RD2,DMWr,DMSel,clk,reset,pc,RD_result);


endmodule
```

(八) Controller

输入端口名称	I/O	位宽
OP	I	6
func	I	6
equal	I	1

首先通过**与逻辑**确定指令类型

```
`define DLEVEL 0
`define ELEVEL 1
`define MLEVEL 2
`define WLEVEL 3

`define SPECIAL 6'b000000
`define ADD     6'b100000
`define SUB     6'b100010
`define ORI     6'b001101
`define LW      6'b100011
`define SW      6'b101011
`define BEQ     6'b000100
`define JAL     6'b000011
`define JR      6'b001000
`define LUI     6'b001111
```

```
`define LH      6'b100001
`define LB      6'b100000
`define SH      6'b101001
`define SB      6'b101000

module _CTRL(
    input [5:0] op,
    input [5:0] func,
    input [1:0] level,
    output RFWr,
    output [1:0] WRSel,
    output [1:0] WDSel,
    output ExtOp,
    output BSel,
    output [2:0] ALUOp,
    output DMWr,
    output [1:0] DMSel,
    output [1:0] NPCOp,
    output [2:0] type,
    output [1:0] T_use_rs,
    output [1:0] T_use_rt,
    output [1:0] T_new
    );

wire add = (op==`SPECIAL&&func==`ADD);
wire sub = (op==`SPECIAL&&func==`SUB);
wire ori = (op==`ORI);
wire lw = (op==`LW);
wire sw = (op==`SW);
wire beq =(op==`BEQ);
wire jal = (op==`JAL);
wire jr  = (op==`SPECIAL&&func==`JR);
wire lui = (op==`LUI);
wire lh = (op==`LH);
wire lb = (op==`LB);
wire sh = (op==`SH);
wire sb = (op==`SB);


assign T_use_rs =   (beq|jr) ? 0 :
                    (add|sub|ori|lw|sw) ? 1 :
                    3;

assign T_use_rt =     (beq) ? 0 :
                      (add|sub) ? 1:
                      (sw) ? 2 :
                      3;

assign T_new = ((level==`ELEVEL)&&(lw|jal)) ? 2 :
                ((level==`ELEVEL)&&(add|sub|ori|lui)) ? 1 :
                ((level==`MLEVEL)&&(lw|jal)) ? 1 :
                0;


endmodule
```

根据真值表完成**或逻辑**

指令	WRSel	WDSel	RFWr	ExtOp	BSel	ALUOp	DMWr	NPCOp
含义	00：写入rt 01：写入rd 10：写入$ra	00：写ALU计算结果 01：写DM取出的数据 10：写PC+8						00：+4 01：b类 10：GPR[$ra] 11：j类
add	01	00	1	0	0	00	0	00
sub	01	00	1	0	0	01	0	00
ori	00	00	1	0	1	10	0	00
lw	00	01	1	1	1	00	0	00
sw	00	00	0	1	1	00	1	00
beq	00	00	0	1	0	00	0	01
lui	00	00	1	0	1	11	0	00
jal	10	10	1	0	0	00	0	11
jr	0	0	0	0	0	0	0	10

```
assign RFWr=add|sub|ori|lw|jal|lui|lh|lb;
assign WRSel[0]=add|sub;
assign WRSel[1]=jal;
assign WDSel[0]=lw|lh|lb;
assign WDSel[1]=jal;
assign ExtOp=lw|sw|beq|lh|lb|sh|sb;
assign BSel=ori|lw|sw|lui|lh|lb|sh|sb;
assign ALUOp[0]=sub|lui;
assign ALUOp[1]=ori|lui;
assign ALUOp[2]=0;
assign DMWr=sw|sh|sb;
assign DMSel[0] = lh|sh;
assign DMSel[1] = lb|sb;
assign NPCOp[0]= beq|jal;
assign NPCOp[1]= jal|jr;
assign type =  (beq)? 3'b000:
                    0;
endmodule
```

课上指令

计算类

P5中一般只需要增加ALU的功能，但一定要看清楚新指令的计算行为，最好在MARS里先模拟一下。

- 一般来说新指令的计算行为会稍微复杂一点，用 `always @(*)` 写会比较简单，用 `assign` 的话可以定义一个 `function`。
- 一般情况下，`Tnew` 和 `Tuse` 与**calc_R型指令**保持一致即可。
- 循环移位可以采用以下写法——

```verilog
//以循环左移为例
if(B[4:0] == 5'd0) out = A;
else out = A << B[4:0] | A >> (5'd31 - B[4:0] + 5'd1);
```

条件跳转类

一般跳转类指令有以下几种要求——

1. **条件跳转+无条件链接**
2. **条件跳转+条件链接**
3. **条件跳转+条件（无条件）链接+不跳转时清空延迟槽**

- **条件跳转**比较好做，一般只需增加**CMP模块**中的判断功能即可。
- 如果是**无条件链接**的话也比较简单，可以直接在D级将 `RFWrite`（GRF写入使能）置**1**并让它流水，并更改一下 `A3`(GRF写入地址，一般是要链接到31号寄存器)，最后在W级将GRF写入数据选择成**PC+8**即可。
- 如果是**条件链接**，则需要在D级根据CMP模块的输出结果判断 `RFWrite` 是否有效，写法如下——

```verilog
//为了确定当前指令是新指令，我们设置一个check信号随新指令一起流水，check有效则表示当前指令是新
//D_RFWrite是从D级主控制器输出的信号
wire D_RFWrite_new = check_D ? (D_CMP_out ? 1'b1 : 1'b0) : D_RFWrite;
//这时我们流水到下一级的就是D_RFWrite_new，而不是D_RFWrite
E_Reg  u_E_Reg (//input
                //.....................................
                .RFWrite_D              ( RFWrite_D_new      ),

                //output
                //.....................................
                .RFWrite_E              ( RFWrite_E          ),
                );
```

- 如果题目要求**不跳转时清空延迟槽**，则需要根据当前**CMP模块输出结果**判断是否清空**D级流水寄存器**。**需要注意的是**，如果当前正在处于 `stall` 状态时，不能清空延迟槽（`stall` 说明前面指令的 `Tnew` 大于新指令的 `Tuse`，即需要传入CMP模块的两个值的**最新值**还没有计算出来，因此还无法转发到CMP中）。写法如下——

```verilog
wire D_Reg_clr = check_D & ~D_CMP_out & ~stall;
```

条件访存类

在这里，第一步依然是合并指令到load类或者store类中，进行一些常规的控制信号处理操作。不过这时我们不能再对阻塞模块和转发置之不理。因为根据题目的特定要求，它有的时候在M级需要rs值（只是举个例子），这个时候我们就需要在M级加上对rs的转发；
在阻塞模块里，我们也要做出修改，因为我们在M级才能得到条件真值，这个时候如果D级有需要rs、rt的动作（有客户来取衣服了），E级的情形就是不确定的，因此我们需要在E级进行一个保守的**条件约束**：**如果E级也是该条指令，并且此时D级要用该条指令要写入的寄存器，我们就stall**。形如：

```
//Then, write back to D_grf
wire stall_rs_e = (TuseRS < TnewE) && D_rs_addr && (E_[InsName] ? D_rs_addr == 5'd31 : D_rs_addr == E_RFDst);
```

当然根据不同指令的要求不同，这里给E级强加的暂停条件也不同，请大家根据指令的RTL描述自行变通。但我认为条件暂停的根本是只约束E级，因为事实上只有这里可以产生RegDst等控制信号的不定值（也有人认为M级也要约束，但对于课上测试结果应该是一样的）。

二、测试方案

本人用python写了自动化测评程序，自动生成测试数据，将自己ISE运行结果与Mars运行结果对拍

使用说明

评测机具备两种模式：

1. 自动生成10组测试数据：逐组数据进行评测，当WA时将会中断，并保留本次数据及输出
2. 手动输入测试数据：自己编写MIPS代码进行评测

在运行评测机前，你需要进行如下操作：

- 配置环境变量。

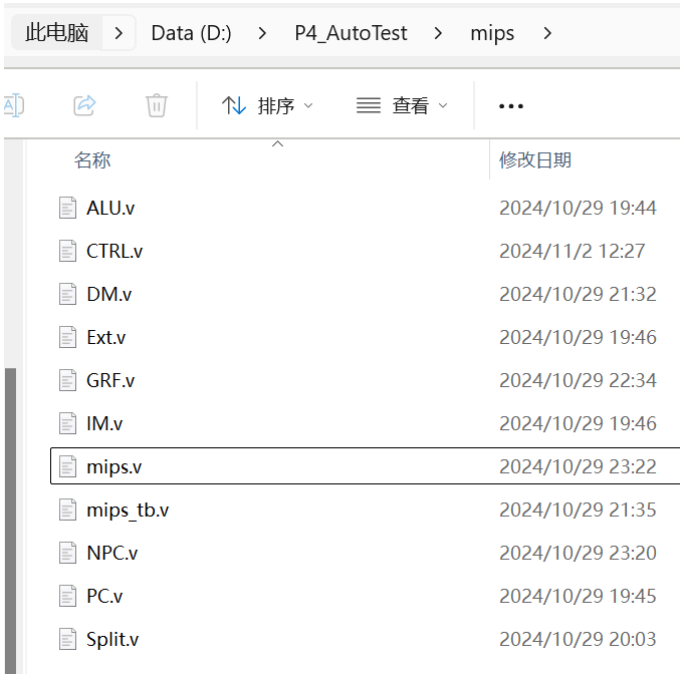
  由于ISE的安装包过大，不再将其打包进评测机内，评测时将使用你本机的ISE

  设置环境变量，key 为 `XILINX` ，value 为 ISE 的安装路径，通常以 `14.7/ISE_DS/ISE/` 结尾，下图仅为示例

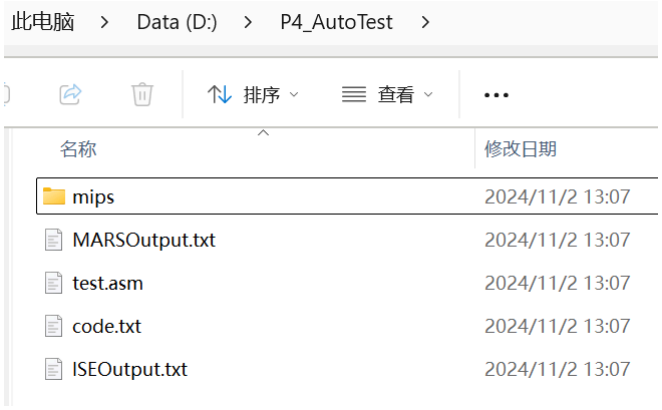
- 在D盘下建立一个新目录，命名为P5_AutoTest
- 在P5_AutoTest建立一个新目录，命名为mips，其需要包含你工程目录下的所有模块的.v文件和顶层模块的测试文件。

  其中，顶层模块命名为mips.v，其测试文件命名为mips_tb.v



- 如果你希望使用模式2进行评测，请将编写好的.asm文件复制到"D:\P5_AutoTest"中，并命名为test.asm

点击运行P5_AutoTest.exe，选择评测模式后按下Enter。评测结果将会显示在控制台上，同时在"D:\P5_AutoTest"中生成此次测试的数据点、你的输出（ISEOutput.txt）和正确输出（MARSOutput.txt）



如果你想保存本次测试数据，请将这几个文件复制到其他文件夹中。本目录中的数据，在下次评测时将会被新的数据覆盖

工具原理

可划分为三个部分

- 生成测试数据

  利用随机数生成指令，规模可在源码中调整

  细节不再详述，保证了生成的MIPS代码可以在Mars上正常运行，不会产生数据溢出、越界等错误，同时达到了对所有指令所有情形的全面覆盖
- 运行ISE和Mars_CO_v0.5.0，得到并格式化二者输出
- 比较输出，返回测评结果

备注

运行后，test.asm中将会有addu和subu，而不是add和sub。这是因为生成测试数据时未考虑溢出问题，而为了保证Mars的正常运行，将生成的add和sub全部替换成了addu和subu，但是机器码仍然是替换前根据add和sub编译的，所以无需对此产生疑虑。

另外，当你选择模式2进行评测时，你的MIPS代码使用add和sub即可，运行时会自动替换

三、思考题

1.我们使用提前分支判断的方法尽早产生结果来减少因不确定而带来的开销，但实际上这种方法并非总能提高效率，请从流水线冒险的角度思考其原因并给出一个指令序列的例子。

> 如果比较操作的结果不能及时反馈到流水线，那么后续依赖于比较结果的指令将无法执行，导致阻塞增多
>
> ```
> ori $1,$0,4
> lw $2,4($1)
> beq $1,$2,loop
> ```

2.因为延迟槽的存在，对于 jal 等需要将指令地址写入寄存器的指令，要写回 PC + 8，请思考为什么这样设计?

> jal的后一条指令在延迟槽中将被无条件执行，为了避免其在`jr $ra`后再次执行，需要写回PC+8

3.我们要求所有转发数据都来源于流水寄存器而不能是功能部件（如 DM、ALU），请思考为什么?

> 如果转发源选择各个功能部件，那么将使关键路径变长，增加接收级的工作时间，即增大了时钟周期，和设计流水线的初衷——“切割关键路径来提升时钟频率” 背道而驰了

4.我们为什么要使用 GPR 内部转发？该如何实现?

> GPR 采用内部转发机制相当于 将M_to_W 流水线寄存器的值直接实时反馈到 GPR 的输出端，从而当前处于 D 级的指令可以直接用到对应寄存器的值，即 W 级到 D 级的转发。 如果不采用内部转发机制，需要额外建立从 M_to_W 流水线寄存器转发到 D 级的数据通路。
>
> 实现方式:
>
> ```
> assign RD1 = (A1==A3&&RFWr==1&&A3!=0) ? WD : register[A1];
> assign RD2 = (A2==A3&&RFWr==1&&A3!=0) ? WD : register[A2];
> ```

5.我们转发时数据的需求者和供给者可能来源于哪些位置？共有哪些转发数据通路?

> 详见顶层架构图

6.在课上测试时，我们需要你现场实现新的指令，对于这些新的指令，你可能需要在原有的数据通路上做哪些扩展或修改？提示：你可以对指令进行分类，思考每一类指令可能修改或扩展哪些位置。

> - 计算类:
>
>   无需修改数据通路，仅修改ALU即可
>
> - 跳转类

> 有条件、无条件的跳转、链接，或许还有清空延迟槽的操作。需要修改CMP，并考虑寄存器是否冲突
>
> - 访存类
>
>   可能需要增添转发和阻塞逻辑

7.简要描述你的译码器架构，并思考该架构的优势以及不足。

> 采用了分布式译码
>
> 优势：无需在各流水线寄存器中添加各个信号，减少了实现时的复杂度。
>
> 不足：每一级都需要译码

8.请详细描述你的测试方案及测试数据构造策略

> 在P4评测机的基础上做了修改，减少了指令涉及的寄存器数量，使得冲突更易发生