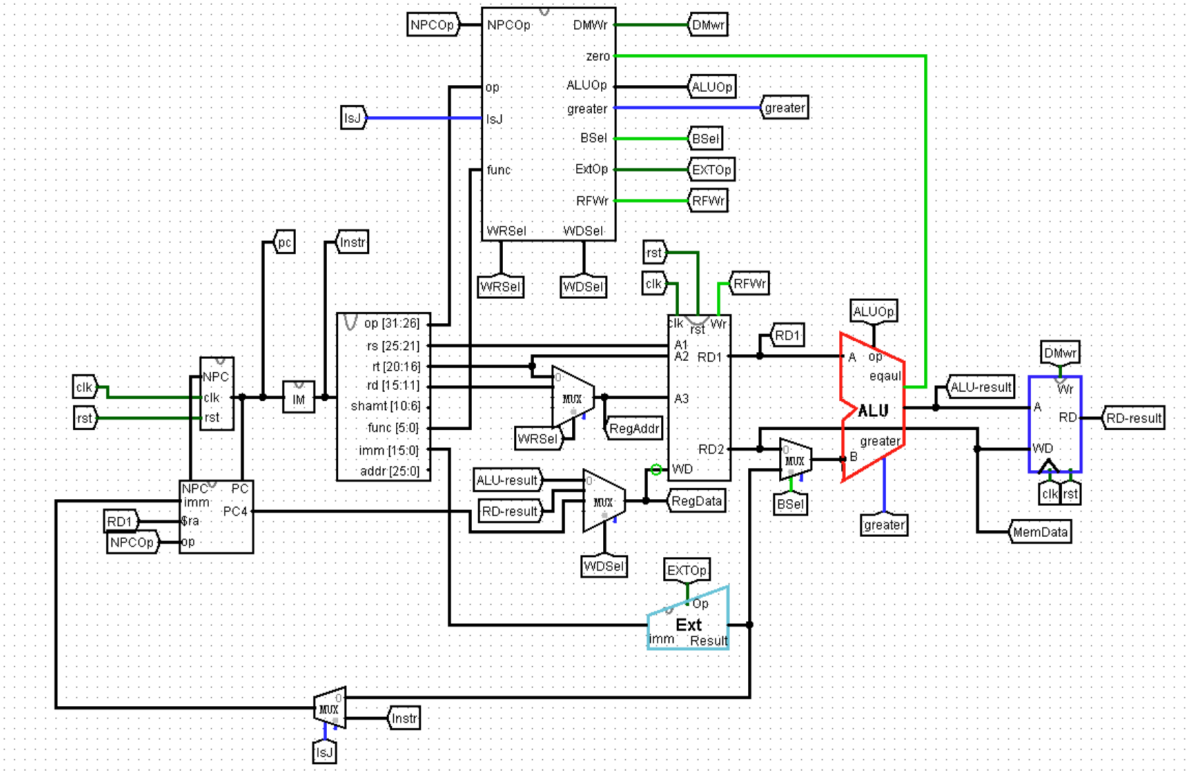
一、设计草稿

CPU顶层架构模型



> 整个单周期CPU的工作流程如下
>
> - 根据程序计数器PC从ROM中取出指令
> - 译码，确定指令类型
> - 根据指令类型执行操作
>   - 读写寄存器
>   - 读写内存
>   - 更新程序计数器PC

依据上述流程，我们设计的 CPU 应包含以下几个功能模块

- IFU（取指令单元，从存储指令的 ROM 模块中取出下一条指令的32位二进制码）
- $Split$（将每条指令的 32 位二进制码分成分别表示寄存器号、`OpCode` 码等的二进制码段）
- $Controller$（控制器，根据 splitter 得到的 6 位 `OpCode` 码和 6 位 `func` 码确定指令的类型并输出对应的控制信号）
- GRF（寄存器堆）
- ALU（算术逻辑单元，实现指令需要的数学运算）
- EXT（位扩展器，根据需要进行相应的位扩展）等基本部件
- DM（数据存储器，内存）

对本次实验所实现的指令，依据其执行的操作进行分类

功能	指令
读、写寄存器	add，sub，lui
读内存、写寄存器	lw
读寄存器、写内存	sw
读寄存器	beq

下面我们将根据各个指令的功能，完成各个功能模块的设计

（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

端口名称	I/O	位宽	功能
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
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 [31:0] imm,
    input [31:0] ra,
```

```
    output [31:0] PC4,
    output [31:0] NextPC
    );

assign PC4 = PC + 4;
wire [31:0] bnext = PC + 4 +(imm<<2);
wire [31:0] jnext = {PC[31:28],imm[25:0],{2'b00}};

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

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

端口名称	I/O	位宽	功能
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
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 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

module _CTRL(
    input [5:0] op,
    input [5:0] func,
    input equal,
    input greater,
    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
    );

wire DMWr;

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);
```

根据真值表完成**或逻辑**

指令	WRSel	WDSel	RFWr	ExtOp	BSel	ALUOp	DMWr	NPCOp	isj
add	01	00	1	X	0	00	0	00	0
sub	01	00	1	X	0	01	0	00	0
ori	00	00	1	0	1	10	0	00	0
lw	00	01	1	1	1	00	0	00	0
sw	10	XX	0	1	1	00	1	00	0

指令	WRSel	WDSel	RFWr	ExtOp	BSel	ALUOp	DMWr	NPCOp	isj
beq	10	XX	0	1	0	XX	0	equal=0 : 00 equal=1 : 01	0
lui	00	00	1	0	1	11	0	00	0
jal	10	10	1	0	0	00	0	11	1
jr	0	0	0	0	0	0	0	10	0

```
assign RFWr=add|sub|ori|lw|jal|lui;
assign WRSel[0]=add|sub;
assign WRSel[1]=jal;
assign WDSel[0]=lw;
assign WDSel[1]=jal;
assign ExtOp=lw|sw|beq;
assign BSel=ori|lw|sw|lui;
assign ALUOp[0]=sub|lui;
assign ALUOp[1]=ori|lui;
assign ALUOp[2]=0;
assign DMWr=sw;
assign NPCOp[0]=(beq&equal)|jal;
assign NPCOp[1]=jal|jr;
assign isj=jal;

endmodule
```

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：

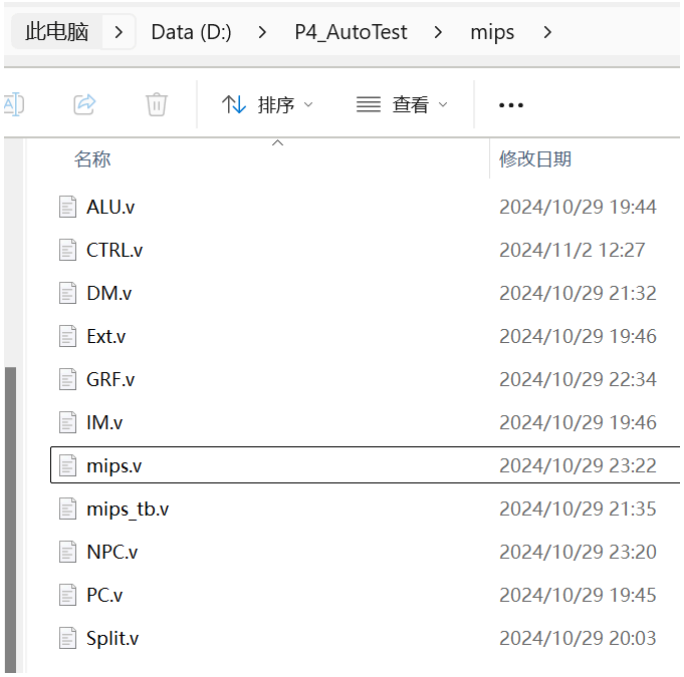
- 配置环境变量。

  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

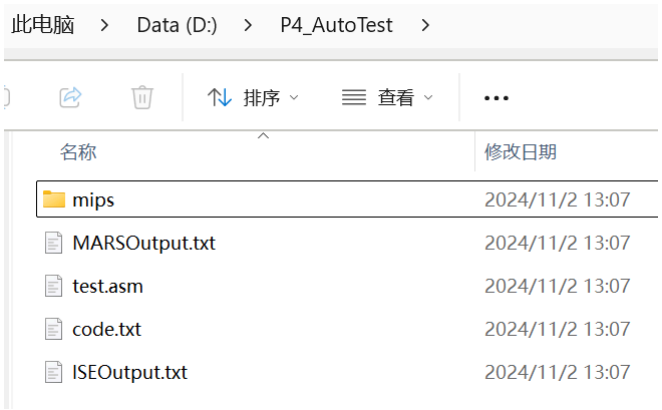
- 在D盘下建立一个新目录，命名为P4_AutoTest
- 在P4_AutoTest建立一个新目录，命名为mips，其需要包含你工程目录下的所有模块的.v文件和顶层模块的测试文件。

  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:\P4_AutoTest"中，并命名为test.asm

点击运行P4_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:\P4_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.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]？

> addr的单位是字节，而DM地址的单位是字，位宽为10，故需要取addr的2-11位

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。

> - **指令对应的控制信号如何取值**
>
>   有9个控制信号，位数一共15位，需要用一个15位reg变量存储，最后用Splitter分为各个控制指令。下面以add为例
>
>   ```
>   reg [15:0] ctrl;
>   assign RFWr = ctrl[0];
>   assign WRSel = ctrl[2:1];
>   assign WDSel = ctrl[4:3];
>   assign ExtOp = ctrl[5];
>   assign BSel = ctrl[6];
>   assign ALUOp = ctrl[9:7];
>   assign DMWr = ctrl[10];
>   assign NPCOp = ctrl[12:11];
>   assign DMSel = ctrl[14:13];
>
>   always @(*) begin
>       if(add == 1) begin
>           reg = 15'b000000000000001;
>       end
>   end
>   ```
>
> - **控制信号不同取值对应的指令**
>
>   参见上文Controller模块

> 前者可便于添加指令时的调整，但是不利于观察控制信号和指令的关系。后者可以直观地反映每个控制信号与指令的关系，但是在添加啊指令时需要遍历修改每个控制信号，比较繁琐。

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。**

> 同步复位中，reset在clk上升沿才有效，故clk优先级更高
>
> 异步复位中，reset在任何时候都有效，故reset优先级更高

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。

> **add:**
>
> ```
> temp ← (GPR[rs]31||GPR[rs]31..0) + sign_extend(immediate)
>
> if temp32 ≠ temp31 then
> SignalException(IntegerOverflow)
>
> else
> GPR[rt] ← temp
> endif
> ```
>
> **addu:**
>
> ```
> temp ← GPR[rs] + GPR[rt]
>
> GPR[rd] ← temp
> ```
>
> 二者的区别仅在于add中的temp为33位，比addu的多了一位，用于判断是否溢出。而rd寄存器中储存的同样都是运算结果的低32位，故在忽略溢出的前提下，二者等价。同理，addi和addiu亦如此