P4课下-Verilog搭建单周期CPU

处理器为 32 位单周期处理器，不考虑延迟槽。支持指令集：`add, sub, ori, lw, sw, beq, lui, jal, jr, nop`。其中 `nop` 为空指令，机器码 `0x00000000`； `add, sub` 按无符号处理，不考虑溢出；顶层有效驱动信号包括且仅包括**同步复位信号 reset 和时钟信号 clk** 。

IM（指令存储器）

在 ISE 中，我们直接将指令的十六进制代码从 code.txt 中读到 IM ，使用 `$readmemh` 指令完成相应功能即可。
该模块接收由 NPC 传入的指令，输出 PC 指令作为 NPC 的输入，在 NPC 模块中进一步进行跳转或累加。

信号名	方向	描述
clk	I	时钟信号
reset	I	同步复位信号
NPC	I	32 位 PC 指令输入信号
Instr	O	32 位指令信号
PC	O	32 位 PC 指令输出信号

```
module IM(
    input clk,
    input reset,
    input [31:0] NPC,
    output [31:0] Instr,
    output reg [31:0] PC
    );

        reg [31:0] im[0:4095];
        wire [31:0] addr;

        initial begin
                $readmemh("code.txt", im);
                PC = 32'h0000_3000;
        end

        always @(posedge clk) begin
                if (reset) begin
                        PC <= 32'h0000_3000;
                end
                else begin
                        PC <= NPC;
                end
        end
```

```
        assign addr = PC - 32'h0000_3000;
        assign Instr = im[(addr >> 2)];

endmodule
```

PC（程序计数器）

不同于上一个项目的 PC，考虑到实际使用，这里的 PC 将用于实现指令地址的累加、跳转和复位。复位后，指令地址将指向 0x00003000。

信号名	方向	描述
PC	I	32 位 PC 输入
imm16	I	16 位立即数
imm26	I	26 位立即数
BranchOp	I	3 位跳转方式
jrreg	I	32 位寄存器值，用于 jr 跳转
NPC	O	32 位新指令地址

```
module PC(
    input [31:0] PC,
    input [15:0] imm16,
    input [25:0] imm26,
    input [2:0] BranchOp,
    input [31:0] jrreg,
    output [31:0] NPC
    );

        wire [31:0] imm;
        assign imm = (BranchOp == 3'b001) ? {{16{imm16[15]}}, imm16} :
                                            (BranchOp == 3'b010) ? {{6{imm26[25]}},
imm26} : 0;

        assign NPC = (BranchOp == 3'b000) ? (PC + 3'd4) :
                                            (BranchOp == 3'b001) ? (PC + 3'd4 + (imm <<
2)) :
                                            (BranchOp == 3'b010) ? ({PC[31:28],
imm[25:0], 2'b00}) :
                                            (BranchOp == 3'b011) ? (3'd4 + jrreg) : 0;

endmodule
```

GRF（通用寄存器组）

使用 `reg [31:0] regs[0:31];` 构造 32 个寄存器，并全部初始化为 0 。**时钟上升沿**且**写使能信号高电平**时，使用 `$display` 进行输出。
注意 0 号寄存器的值永远为 0 。

信号名	方向	描述
clk	I	时钟信号
reset	I	同步复位信号
WE	I	写使能信号
A1	I	5 位地址输入信号，读取输出到 RD1
A2	I	5 位地址输入信号，读取输出到 RD2
A3	I	5 位地址输入信号，作为目标写入寄存器
WD	I	32 位数据输入
PC	I	32 位指令地址信号，用作 display 输出
RD1	O	32 位 A1 数据输出
RD2	O	32 位 A2 数据输出

```
module GRF(
    input clk,
    input reset,
    input WE,
    input [4:0] A1,
    input [4:0] A2,
    input [4:0] A3,
    input [31:0] WD,
    input [31:0] PC,
    output [31:0] RD1,
    output [31:0] RD2
    );

        integer i;
        reg [31:0] regs[0:31];

        always @(posedge clk) begin
                if (reset) begin
                        for (i = 0; i < 32; i = i + 1) begin
                                regs[i] <= 32'h0000_0000;
                        end
                end
                else begin
                        if (WE) begin
                                $display("@%h: $%d <= %h", PC, A3, WD);
                                if (A3 != 5'b0) begin
                                        regs[A3] <= WD;
                                end
                        end
                end
        end

        assign RD1 = regs[A1];
        assign RD2 = regs[A2];
```

```
endmodule
```

DM（数据存储器）

使用 `reg [31:0] dm[0:3071];` 构造内存，初始化为 0 。

信号名	方向	描述
clk	I	时钟信号
reset	I	同步复位信号
WE	I	写使能信号
addr	I	16 位写入地址
WD	I	32 位写入数据
PC	I	32 位指令地址，用于 display 输出
RD	O	32 位数据输出

```
module DM(
    input clk,
    input reset,
    input WE,
    input [15:0] addr,
    input [31:0] WD,
    input [31:0] PC,
    output [31:0] RD
    );

        reg [31:0] dm[0:3071];
        wire [11:0] MemAddr;
        wire [31:0] Addr;
        integer i;

        always @(posedge clk) begin
                if (reset) begin
                        for (i = 0; i < 3072; i = i + 1) begin
                                dm[i] <= 32'h0000_0000;
                        end
                end
                else begin
                        if (WE) begin
                                dm[MemAddr] <= WD;
                                $display("@%h: *%h <= %h", PC, Addr, WD);
                        end
                end
        end

        assign MemAddr = addr[13:2];
        assign RD = dm[MemAddr];
```

```
        assign Addr = {{16{1'b0}}, addr};

endmodule
```

EXT（扩展单元）

信号名	方向	描述
Imm	I	16 位立即数
ExtOp	I	扩展方式，高电平为符号扩展
Imm32	O	32 位输出

ALU（算术逻辑单元）

信号名	方向	描述
ALUctr	I	3 位选择信号 000：加法 001：减法 010：或 011：逻辑左移 16 位
Op1	I	32 位操作数
Op2	I	32 位操作数
Zero	O	判断运算结果是否为 0
ALUResult	O	32 位运算结果

```
module ALU(
    input [2:0] ALUctr,
    input [31:0] Op1,
    input [31:0] Op2,
    output Zero,
    output reg [31:0] ALUResult
    );

        always @(*) begin
                case (ALUctr)
                        3'b000: ALUResult = Op1 + Op2;
                        3'b001: ALUResult = Op1 - Op2;
                        3'b010: ALUResult = Op1 | Op2;
                        3'b011: ALUResult = Op2 << 16;
                endcase
        end

        assign Zero = (ALUResult == 0) ? 1'b1 : 1'b0;

endmodule
```

Controller（控制器）

信号名	方向	描述
Instr	I	32 位指令
RegDst	O	控制信号，确定目标寄存器的指定方式 0：将结果写入 `rt` 寄存器（I 型指令） 1：将结果写入 `rd` 寄存器（R 型指令）
ALUSrc	O	控制信号，选择 ALU 的第二个源操作数 0：来自寄存器文件的一个寄存器 1：来自指令中的立即数字段
MemtoReg	O	控制信号，控制数据从内存到寄存器的传输 0：执行指令后，不将内存中的数据写入寄存器文件 1：执行指令后，将内存中的数据写入寄存器文件
RegWrite	O	控制信号，控制数据是否从内部数据路径写入寄存器文件 0：不应该将执行结果写入寄存器文件 1：应该将执行结果写入寄存器文件中的一个指定寄存器
MemWrite	O	控制信号，指示是否应该将数据从数据缓存写入DM
BranchOp	O	3 位控制信号，表示跳转的种类 001：beq 跳转 010：jal 跳转，需要将 PC + 4 写入 31 号寄存器 （$ra） 011：jr 跳转
ExtOp	O	控制信号，操作数扩展信号 0：0扩展 1：符号扩展
ALUctr	O	3 位控制信号，选择 ALU 运算符
imm16	O	16 位立即数
imm26	O	26 位立即数
rs	O	5 位 rs
rt	O	5 位 rt
rd	O	5 位 rd

指令	add	sub	ori	lw	sw	beq	lui	jal	jr
RegDst	1	1							
ALUSrc			1	1	1		1		
MemtoReg				1					
RegWrite	1	1	1	1			1	1	
MemWrite					1				
BranchOp						001		010	011
ExtOp				1	1	1			
ALUctr	000	001	010	000	000	001	011	000	000

指令	op5	op4	op3	op2	op1	op0
add	0	0	0	0	0	0
sub	0	0	0	0	0	0
ori	0	0	1	1	0	1
lw	1	0	0	0	1	1
sw	1	0	1	0	1	1
beq	0	0	0	1	0	0
lui	0	0	1	1	1	1
nop	0	0	0	0	0	0
jal	0	0	0	0	1	1
jr	0	0	0	0	0	0

指令	func5	func4	func3	func2	func1	func0
add	1	0	0	0	0	0
sub	1	0	0	0	1	0
nop	0	0	0	0	0	0
jr	0	0	1	0	0	0

```
module Controller(
    input [31:0] Instr,
    output RegDst,
    output ALUSrc,
        // more
    );

        wire [5:0] op;
        wire [5:0] func;

        assign op = Instr[31:26];
        assign func = Instr[5:0];
        assign imm16 = Instr[15:0];
        assign imm26 = Instr[25:0];
        assign rs = Instr[25:21];
        assign rt = Instr[20:16];
        assign rd = Instr[15:11];

        wire add = (op == 6'b0 && func == 6'b100000);
        wire sub = (op == 6'b0 && func == 6'b100010);
        // more

        assign RegDst = add || sub;
        assign ALUSrc = ori || lw || sw || lui;
        // more
```

```
        assign BranchOp[0] = beq || jr;
        assign BranchOp[1] = jal || jr;
        assign BranchOp[2] = 0;
        // more

endmodule
```

mips（顶层模块组装）

机器码指令

J 类型指令

本次项目中，`jal, jr` 是 J 类型指令。
`jal`：

op	target
6bit	26bit

注意：将 PC 的最高 4 位与 target 拼接，最低两位补 0 作为新的 PC 值；原 PC 值 +4 存在 31 号寄存器中。
`jr`：

op	rs	padding
6bit	5bit	21bit

注意：取出 rs 寄存器的值，作为新的 PC 。

寄存器编码

reg	name	usage
$0	$zero	常量 0
$1	$at	保留给汇编器使用的临时变量
$2-$3	$v0-$v1	函数调用返回值
$4-$7	$a0-$a3	函数调用参数
$8-$15	$t0-$t7	临时变量
$16-$23	$s0-$s7	需要保存的变量
$24-$25	$t8-$t9	临时变量
$26-$27	$k0-$k1	留给操作系统使用
$28	$gp	全局指针
$29	$sp	堆栈指针

reg	name	usage
$30	$fp	帧指针
$31	$ra	返回地址

要点简析

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

        wire [31:0] NPC, Instr, PC;
        IM u_IM (
                .clk(clk),
                .reset(reset),
                .NPC(NPC),
                .Instr(Instr),
                .PC(PC)
        );

        wire [15:0] imm16;
        wire [25:0] imm26;
        wire [2:0] BranchOp, BRCHOp;
        assign BRCHOp = (BranchOp != 3'b001) ? BranchOp :
                                                (Zero == 1'b1) ? BranchOp : 3'b000;
        wire [31:0] jrreg;
        PC u_PC (
                .PC(PC),
                .imm16(imm16),
                .imm26(imm26),
                .BranchOp(BRCHOp),
                .jrreg(jrreg),
                .NPC(NPC)
        );

        // Controller

        wire [31:0] immALU;
        EXT u_EXT_ALU (
                .Imm(imm16),
                .ExtOp(ExtOp),
                .Imm32(immALU)
        );

        wire [31:0] RD1, RD2, ALUResult;
        wire Zero;
        ALU u_ALU (
                .ALUctr(ALUctr),
                .Op1(RD1),
                .Op2(ALUSrc ? immALU : RD2),
```

```
                .Zero(Zero),
                .ALUResult(ALUResult)
        );

        wire [31:0] RD;
        GRF u_GRF (
                .clk(clk),
                .reset(reset),
                .WE(RegWrite),
                .A1(rs),
                .A2(rt),
                .A3(RegDst ? rd :
                   (BranchOp == 3'b010) ? 5'd31 : rt),
                .WD(MemtoReg ? RD :
                   (BranchOp == 3'b010) ? (PC + 3'd4) : ALUResult),
                .PC(PC),
                .RD1(RD1),
                .RD2(RD2)
        );
        assign jrreg = RD1;

        DM u_DM (
                .clk(clk),
                .reset(reset),
                .WE(MemWrite),
                .addr(ALUResult[15:0]),
                .WD(RD2),
                .PC(PC),
                .RD(RD)
        );

endmodule
```

照着 P3 翻译即可，没啥需要具体说的。和 P3 的不同之处应该是使用 3bit 存储跳转指令了吧，其实也方便课上添加跳转。**祝 P4 上机顺利！**

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 直接与 ALUResult 相连，因为 DM 的写入地址一般是一个寄存器值与立即数偏转相加的结果；因为在 DM 中是按字存储的，需要舍弃最低两位。

> 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：即为笔者使用的方法，这种方式非常直观，且符合逻辑，从机器码到操作数再到控制信号，一目了然。
控制信号每种取值所对应的指令：似乎指令与控制信号之间联系更加紧密了（？），没有太多体会。

> 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，clk 优先级高；异步复位，reset 优先级高。

> 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。

addi 和 addiu 都是将 rs 与立即数（符号扩展）相加，结果存储在 rd 中；add 和 addu 都是将 rs 和 rt 相加，结果存储在 rd 中。