一、模块设计

只需要根据P3的电路将各个子电路模块转化为Verilog，然后在mips.v中实例化各个模块，用wire将各个模块的输入输出接在一起即可。

指令	RF				ALU			MD		NPC				
	A1	A2	A3	WD	srcA	srcB	ALUop	A	WD	PC	imm26	cmpRes	PC4	NPC
add	rs	rt	rd	ALU.add	RF.RD1	RF.RD2	add			PC	instr[25:0]		PC+4	PC+4
sub	rs	rt	rd	ALU.sub	RF.RD1	RF.RD2	sub			PC	instr[25:0]		PC+4	PC+4
ori	rs		rt	ALU.or	RF.RD1	zero_ext(imm16)	or			PC	instr[25:0]		PC+4	PC+4
lui	rs		rt	ALU.lui	RF.RD1	zero_ext(imm16)	add			PC	instr[25:0]		PC+4	PC+4
lw	rs		rt	MD.RD	RF.RD1	sign_ext(imm16)	add	ALU.add		PC	instr[25:0]		PC+4	PC+4
sw	rs	rt			RF.RD1	sign_ext(imm16)	add	ALU.add	RF.RD2	PC	instr[25:0]		PC+4	PC+4
beq	rs	rt								PC	instr[25:0]	CMP.equal	PC+4	
jal			0x31	PC4						PC	instr[25:0]		PC+4	$PC_{31..28}\|\|imm26\|\|0^2$
jr	rs									PC	instr[25:0]		PC+4	RF.RD1

子电路模块设计

1. IFU

```
module NPC(
    input [31:0] PC,
    input [25:0] imm26,
    input zero,
    input [1:0] mode,
    input [31:0] regValue,
    output [31:0] PC4,
    output reg [31:0] PC_next
    );

    parameter NEXT = 2'b00, BRANCH = 2'b01, J = 2'b10, JR = 2'b11;

    always @(*) begin
        case (mode)
            NEXT:       PC_next = PC + 4;
            BRANCH: begin
                if (zero)   PC_next = PC + 4 + {{14{imm26[15]}}
,imm26[15:0], 2'b00};
                else        PC_next = PC + 4;
            end
            J:          PC_next = {PC[31:28], imm26, 2'b00};
            JR:         PC_next = regValue;
            default:    PC_next = 32'h00003000;
```

```
        endcase
    end

    assign PC4 = PC + 4;

endmodule
```

```
module PC(
    input [31:0] PC_next,
    input clk,
    input reset,
    output [31:0] PC
    );

    reg [31:0] tmp;

    always @(posedge clk) begin
        if (reset)  begin
            tmp <= 32'h00003000;
        end

        else    begin
            tmp <= PC_next;
        end
    end

    assign PC = tmp;

endmodule
```

```
module IM(
    input [13:2] addr,
    output [31:0] instr
    );

    reg [31:0] instructions [0:4095];

    initial begin
        $readmemh("code.txt", instructions);
    end
```

```
    assign instr = instructions[addr];

endmodule
```

2. ALU与CMP

```
module ALU(
    input [31:0] srcA,
    input [31:0] srcB,
    input [4:0] shamt,
    input [2:0] aluOp,
    output reg [31:0] aluRes
    );


    parameter ADD = 3'b000, SUB = 3'b001, OR = 3'b010, HIGH =
3'b011;

    always @(*) begin
        case (aluOp)
            ADD:        aluRes = srcA + srcB;
            SUB:        aluRes = srcA - srcB;
            OR:         aluRes = srcA | srcB;
            HIGH:       aluRes = {srcB[15:0], {16{1'b0}}};
            default:    aluRes = 32'h00000000;
        endcase
    end

endmodule
```

3. GRF

需要注意的是，指令中可能存在写入0，由于0的值始终为0，因此在向寄存器写数据时要判断，当写入寄存器编号regWA不为0时才可以向其中写入数据。

```
module GRF(
    input [4:0] regA1,
    input [4:0] regA2,
    input [4:0] regWA,
    input [31:0] regWD,
    input [31:0] PC,
    input regWE,
```

```
    input clk,
    input reset,
    output [31:0] regRD1,
    output [31:0] regRD2
    );

    parameter ZERO = 5'b00000;
    reg [31:0] rf [31:0];
    integer i;

    initial begin
        for (i = 0; i < 32; i = i + 1)  begin
            rf[i] <= 32'h00000000;
        end
    end

    always @(posedge clk) begin
        if(reset)   begin
            for (i = 0; i < 32; i = i + 1) begin
                rf[i] <= 32'h00000000;
            end
        end

        else        begin
            if(regWE)   begin
                $display("@%h: $%d <= %h", PC, regWA, regWD);
                if (regWA != ZERO)
                    rf[regWA] <= regWD;
            end
        end
    end

    assign regRD1 = (regA1 == ZERO) ? 32'h00000000 : rf[regA1];
    assign regRD2 = (regA2 == ZERO) ? 32'h00000000 : rf[regA2];

endmodule
```

4. DM

课下涉及对DM的指令只有`sw  lw`，考虑到课上的新增指令，因此可以为DM增加一个工作模式信号**mode**，以方便扩展。

```
module DM(
    input [31:0] memAddr,
    input [31:0] memWD,
    input [31:0] PC,
    input memWE,
    input clk,
    input reset,
    input mode,
    output reg [31:0] memRD
    );


    parameter WORD = 1'b0, BYTE = 1'b1;
    reg [31:0] memory [3071:0];
    reg [31:0] tmp;
    integer i;

    initial begin
        for (i = 0; i < 3072; i = i + 1)    begin
            memory[i] <= 32'h00000000;
        end
    end

    always @(posedge clk) begin
        if(reset)   begin
            for (i = 0; i < 3072; i = i + 1) begin
                memory[i] <= 32'h00000000;
            end
        end

        else        begin
            if(memWE)   begin
                $display("@%h: *%h <= %h", PC, memAddr, memWD);
                if (mode == WORD)
                    memory[memAddr[13:2]] <= memWD;
                else    begin
                    case (memAddr[1:0])
                        2'b00:      memory[memAddr[13:2]] <=
{24'h000000, memWD[7:0]};
                        2'b01:      memory[memAddr[13:2]] <=
{16'h0000, memWD[15:8], 8'h00};
                        2'b10:      memory[memAddr[13:2]] <=
{8'h00, memWD[23:16], 16'h0000};
```

```
                        2'b11:      memory[memAddr[13:2]] <=
{memWD[31:24], 24'h000000};
                        default:    memory[memAddr[13:2]] <=
32'h00000000;
                    endcase
                end
            end
        end
    end

    always @(*) begin
        if (mode == WORD)
            memRD = memory[memAddr[13:2]];
        else    begin
            tmp = memory[memAddr[13:2]];
            case (memAddr[1:0])
                2'b00:      memRD = {{24{tmp[7]}}, tmp[7:0]};
                2'b01:      memRD = {{24{tmp[15]}}, tmp[15:8]};
                2'b10:      memRD = {{24{tmp[23]}}, tmp[23:16]};
                2'b11:      memRD = {{24{tmp[31]}}, tmp[31:24]};
                default:    memRD = 32'h00000000;
            endcase
        end
    end

endmodule
```

5. EXT

```
module EXT(
    input [15:0] imm,
    input extMode,
    output [31:0] ext_imm
    );

    assign ext_imm = extMode ? {{16{imm[15]}}, imm} : {{16{1'b0}},
imm};

endmodule
```

6. Controller

```
module Controller(
    input [5:0] opCode,
    input [5:0] funct,
    output [2:0] aluOp,
    output regWE,
    output memWE,
    output memMode,
    output aluSrcMux,           // imm as srcB
    output extMode,             // Sign Extend
    output [1:0] regWAMux,      // 0-rt / 1-rd / 2-$ra
    output [1:0] regWDMux,      // 0-aluRes / 1-memRD / 2-PC4
    output [1:0] npcMode
    );

    parameter NPCNEXT = 2'b00, NPCBRANCH = 2'b01, NPCJ = 2'b10,
NPCJR = 2'b11;
    parameter ALUADD = 3'b000, ALUSUB = 3'b001, ALUOR = 3'b010,
ALUHIGH = 3'b011, ALUNOP = 3'b111;

    parameter NOP = 4'd0, ADD = 4'd1, SUB = 4'd2, ORI = 4'd3, LW =
4'd4, SW = 4'd5, BEQ = 4'd6, LUI = 4'd7, J = 4'd8, JAL = 4'd9, JR =
4'd10;
    reg [3:0] instrKind;

    always @(*) begin
        case (opCode)
            6'b000000:  begin
                if (funct == 6'b100000)         instrKind = ADD;
                else if (funct == 6'b100010)    instrKind = SUB;
                else if (funct == 6'b001000)    instrKind = JR;
                else                            instrKind = NOP;
            end
            6'b001101:                          instrKind = ORI;
            6'b100011:                          instrKind = LW;
            6'b101011:                          instrKind = SW;
            6'b000100:                          instrKind = BEQ;
            6'b001111:                          instrKind = LUI;
            6'b000010:                          instrKind = J;
            6'b000011:                          instrKind = JAL;
            default:                            instrKind = NOP;
        endcase
```

```
    end


    assign  regWE = instrKind == ADD || instrKind == SUB ||
instrKind == ORI || instrKind == LW || instrKind == JAL ||
instrKind == LUI;
    assign  memWE = instrKind == SW;
    assign  aluOp = (instrKind == ADD || instrKind == LW ||
instrKind == SW) ? ALUADD :
                    (instrKind == SUB) ? ALUSUB :
                    (instrKind == ORI)  ? ALUOR :
                    (instrKind == LUI) ? ALUHIGH : ALUNOP;
    assign  extMode = instrKind == LW || instrKind == SW;
    assign  aluSrcMux = instrKind == ORI || instrKind == LW ||
instrKind == SW || instrKind == LUI;
    assign  memMode = 0;
    assign  npcMode = (instrKind == BEQ) ? NPCBRANCH :
                      (instrKind == J || instrKind == JAL)   ? NPCJ
:
                      (instrKind == JR)  ? NPCJR : NPCNEXT;
    assign  regWAMux = (instrKind == ADD || instrKind == SUB) ?
2'b01 :
                       (instrKind == JAL) ? 2'b10 : 2'b00;
    assign  regWDMux = (instrKind == LW)  ? 2'b01 :
                       (instrKind == JAL) ? 2'b10 : 2'b00;

endmodule
```

顶层模块**mips**设计

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

    parameter RA = 5'b11111;

//////////////////////////////////////////////////////////////////
//////////////
    /*      IFU     */
    wire [31:0] instr;
    wire [31:0] PC;
```

```
    wire [31:0] NPC;
    wire [31:0] PC4;
    wire [1:0] npcMode;
    wire [31:0] instrAddr;
    assign instrAddr = PC - 32'h00003000;

    /*      CMP     */
    wire [1:0] cmpOp;
    wire cmpRes;
    assign cmpOp = 2'b00;

    /*      GRF     */
    wire [1:0] regWAMux;            // MUX, 0-rt / 1-rd / 2-$ra
    wire [1:0] regWDMux;            // MUX, 0-aluRes / 1-memRD / 2-
PC4
    wire [4:0] regWA;
    wire [31:0] reg1Value;
    wire [31:0] reg2Value;
    wire [31:0] regWD;
    assign regWA = regWAMux == 2'b01 ? instr[15:11] : regWAMux ==
2'b10 ? RA : instr[20:16];
    assign regWD = regWDMux == 2'b01 ? memRD : regWDMux == 2'b10 ?
PC4 : aluRes;

    /*      ALU      */
    wire aluSrcMux;                 // MUX, 0-reg2Value / 1-imm as
srcB
    wire [31:0] srcA;
    wire [31:0] srcB;
    wire [2:0] aluOp;
    wire [31:0] aluRes;
    assign srcA = reg1Value;
    assign srcB = aluSrcMux ? ext_imm : reg2Value;

    /*      DM      */
    wire [31:0] memAddr;
    wire [31:0] memWD;
    wire [31:0] memRD;
    wire memMode;
    assign memAddr = aluRes;
    assign memWD = reg2Value;

    /*      EXT     */
```

```
    wire extMode;
    wire [31:0] ext_imm;
////////////////////////////////////////////////////////////////
//////////////

    PC pcInstance (
        .PC_next(NPC),
        .clk(clk),
        .reset(reset),
        .PC(PC)
    );

    IM imInstance (
        .addr(instrAddr[13:2]),
        .instr(instr)
    );

    GRF grfInstance (
        .regA1(instr[25:21]),
        .regA2(instr[20:16]),
        .regWA(regWA),
        .clk(clk),
        .reset(reset),
        .PC(PC),
        .regWE(regWE),
        .regRD1(reg1Value),
        .regRD2(reg2Value),
        .regWD(regWD)
    );

    ALU aluInstance (
        .srcA(srcA),
        .srcB(srcB),
        .shamt(instr[10:6]),
        .aluOp(aluOp),
        .aluRes(aluRes)
    );

    DM dmInstance (
        .memAddr(memAddr),
        .memWD(memWD),
        .memWE(memWE),
        .PC(PC),
```

```
        .clk(clk),
        .reset(reset),
        .mode(memMode),
        .memRD(memRD)
    );

    Controller cuInstance (
        .opCode(instr[31:26]),
        .funct(instr[5:0]),
        .aluOp(aluOp),
        .regWE(regWE),
        .memWE(memWE),
        .memMode(memMode),
        .aluSrcMux(aluSrcMux),
        .extMode(extMode),
        .regWAMux(regWAMux),
        .regWDMux(regWDMux),
        .npcMode(npcMode)
    );

    CMP cmpInstance (
        .num1(reg1Value),
        .num2(reg2Value),
        .cmpOp(cmpOp),
        .cmpRes(cmpRes)
    );

    NPC npcInstance (
        .PC(PC),
        .imm26(instr[25:0]),
        .zero(cmpRes),
        .mode(npcMode),
        .regValue(reg1Value),
        .PC4(PC4),
        .PC_next(NPC)
    );

    EXT extInstance (
        .imm(instr[15:0]),
        .extMode(extMode),
        .ext_imm(ext_imm)
    );
```

```
endmodule
```

二、测试方案

编写MIPS汇编代码导出为16进制文件，在Verilog中通过系统任务$readmemh读入到IM中，然后开始测试，将Verilog仿真结果与Mars运行结果对比。

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

> *addr*信号来自于*ALU*计算的结果，由于*MIPS*中数据按字对齐，每个字所占地址为*4*，在*DM*中按照字存储数据，因此需要将*addr*除以*4*得到在*DM*中*memory*的下标。

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。

- 第一种方式：记录下指令对应的控制信号如何取值

```
always @(*) begin
    if (ADD)    begin
        regWE = 1;
        regWAMux = 1;
        regWDMux = 0;
    end
    // ...
end
```

- 第二种方式：记录下控制信号每种取值所对应的指令

```
assign regWE = instr==ADD || instr==SUB || instr==ORI || instr ==
LUI || instr==LW || instr==JAL;
assign regWAMux = (instr==ADD || instr==SUB) ? 2'b01 : (instr==JAL)
? 2'b10 : 2'b00;
```

> 第一种方式各条指令对应的控制信号比较明确，在拓展新指令时更方便；第二种方式更能明确各个控制信号的取值情况。

4. 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：只有在时钟上升沿到来时，若*reset*信号有效则部件复位，若*reset*信号无效则更新状态。*clk*信号的优先级比*reset*信号更高。
> 异步复位：任何时刻只要*reset*信号有效部件就复位，只有当*reset*信号无效时时钟上升沿到来才更新状态。*reset*信号的优先级比*clk*信号更高。

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。

> *add*与*addu*的区别在于，*add*在检测到溢出时会抛出异常*SignalException(IntegerOverflow)*，如果没有溢出*add*和*addu*都是将*rs*和*rt*寄存器的值相加写入*rd*寄存器。
> *addi*和*addiu*的区别与*add*和*addu*的区别类似，在不考虑溢出的前提下都是对立即数做符号扩展后和*rs*的值相加后将结果写入*rt*寄存器。