# P4-Verilog单周期CPU设计文档

## 一、设计说明

使用 Logisim 开发一个简单的 MIPS 单周期处理器，设计说明如下：

1. 处理器应支持指令集为：{addu, subu, ori, lw, sw, beq, lui, jal, jr,nop}。
2. addu,subu可以不支持溢出。
3. 处理器为单周期设计。
4. 不需要考虑延迟槽。

## 二、数据通路设计

### 1. PC

#### (1) **模块接口定义**

表1 PC**模块接口定义**

|  |  |
| --- | --- |
| 文件 | 模块接口定义 |
| pc.v | module pc(  input clk,  input reset,  input [31:0] nextPC,  output reg [31:0] PC); |

#### (2) **接口说明**

表2 PC接口说明

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 信号 | 方向 | 描述 |
| 1 | clk | I | 时钟信号 |
| 2 | reset | I | 同步复位信号  1：复位  0：无效 |
| 3 | nextPC | I | PC输入值 |
| 4 | PC | O | PC输出值 |

#### (3)功能定义

表3 PC功能定义

|  |  |  |
| --- | --- | --- |
| 序号 | 功能 | 描述 |
| 1 | 复位 | reset有效时，PC被设置为0x00003000 |
| 2 | PC更新 | 时钟上升沿来临时，PC值更新为nextPC |

### 2. NPC

#### (1) **模块接口定义**

表4 PC**模块接口定义**

|  |  |
| --- | --- |
| 文件 | 模块接口定义 |
| npc.v | module npc(  input i\_Zero,  input [1:0] i\_branch,  input [25:0] i\_jal\_addr,  input [31:0] i\_jr\_addr,  input [31:0] i\_offset,  input [31:0] i\_PC,  output reg [31:0] o\_nextPC,  output [31:0] o\_PC4); |

#### (2) **接口说明**

表2 PC接口说明

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 信号 | 方向 | 描述 |
| 1 | zero | I | 判断PC是否满足beq指令跳转条件  1：满足  0：不满足 |
| 2 | branch | I | 判断当前指令是否为beq指令  1：是  0：不是 |
| 3 | jal\_addr | I | jal指令跳转地址 |
| 4 | jr\_addr | I | jr指令跳转地址 |
| 5 | offset | I | beq指令偏移量 |
| 6 | PC | I | PC输入值 |
| 7 | nextPC | O | nextPC计算结果 |
| 8 | PC4 | O | PC+4输出值 |

#### (3)功能定义

表3 PC功能定义

|  |  |  |
| --- | --- | --- |
| 序号 | 功能 | 描述 |
| 1 | 计算PC的下一个值 | zero=1且branch=01时，nextPC <= PC4 + (offset << 2);  branch=10时，  nextPC <= { PC4[31:28], jal\_addr[25:0],{2{1'b0}}};  branch=11时, nextPC <= jr\_addr;  默认情况, nextPC <= PC4; |

### 3. IM

#### (1) **模块接口定义**

表4 PC**模块接口定义**

|  |  |
| --- | --- |
| 文件 | 模块接口定义 |
| im.v | module im(  input [31:0] PC,  output [31:0] instruction); |

#### (2) **接口说明**

表2 PC接口说明

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 信号 | 方向 | 描述 |
| 1 | PC | I | 32位PC值 |
| 2 | instruction | O | 取出32位指令值 |

#### (3)功能定义

表3 PC功能定义

|  |  |  |
| --- | --- | --- |
| 序号 | 功能 | 描述 |
| 1 | 取指令 | 根据PC值取出指令 |

### 4. GRF

#### (1) **模块接口定义**

表4 PC**模块接口定义**

|  |  |
| --- | --- |
| 文件 | 模块接口定义 |
| grf.v | 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); |

#### (2) **接口说明**

表3 GRF端口说明

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 信号 | 方向 | 描述 |
| 1 | clk | I | 时钟信号 |
| 2 | reset | I | 异步复位信号，将32个寄存器中的值全部清零  1：复位  0：无效 |
| 3 | A1 | I | 5位地址输入信号，指定32个寄存器中的一个，将其中存储的数据读出到RD1 |
| 4 | A2 | I | 5位地址输入信号，指定32个寄存器中的一个，将其中存储的数据读出到RD2 |
| 5 | A3 | I | 5位地址输入信号，指定32个寄存器中的一个作为写入的目标寄存器 |
| 6 | WD | I | 32位写入数据 |
| 7 | WE | I | 写使能信号  1：可向GRF中写入数据  0：不能向GRF中写入数据 |
| 8 | PC | I | 当前PC值 |
| 9 | RD1 | O | 输出A1指定的寄存器的32位数据 |
| 10 | RD2 | O | 输出A2指定的寄存器的32位数据 |

#### (3)功能定义

表4 GRF功能定义

|  |  |  |
| --- | --- | --- |
| 序号 | 功能 | 描述 |
| 1 | 复位 | Reset有效时，将32个寄存器中的值全部清零 |
| 2 | 读数据 | 读出A1, A2地址对应寄存器中所存储数据到RD1, RD2 |
| 3 | 写数据 | 当WE有效且时钟上升沿来临时，将WD写入A3所对应的寄存器中 |

### 5. ALU

#### (1) **模块接口定义**

表4 PC**模块接口定义**

|  |  |
| --- | --- |
| 文件 | 模块接口定义 |
| alu.v | module alu(  input [31:0] A,  input [31:0] B,  input [2:0] ALUOp,  output [31:0] Result,  output Zero); |

#### (2) **接口说明**

表5 ALU端口说明

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 信号 | 方向 | 描述 |
| 1 | A | I | 参与ALU计算的第一个32位数据 |
| 2 | B | I | 参与ALU计算的第二个32位数据 |
| 3 | ALUOp | I | ALU功能的选择信号：  000：ALU进行与运算  001：ALU进行或运算  010：ALU进行加法运算  011：ALU进行减法运算 |
| 4 | Result | O | ALU的计算结果 |
| 5 | Zero | O | A,B是否相等的标志信号  1：相等  0：不相等 |

#### (3)功能定义

表6 ALU功能定义

|  |  |  |
| --- | --- | --- |
| 序号 | 功能 | 描述 |
| 1 | 与运算 |  |
| 2 | 或运算 |  |
| 3 | 加运算 |  |
| 4 | 减运算 |  |
| 5 | 判断A、B是否相等 |  |

### 6. DM

#### (1) **模块接口定义**

表4 PC**模块接口定义**

|  |  |
| --- | --- |
| 文件 | 模块接口定义 |
| dm.v | module dm(  input clk,  input reset,  input MemWrite,  input MemRead,  input [31:0] addr,  input [31:0] WriteData,  input [31:0] PC,  output [31:0] ReadData); |

#### (2) **接口说明**

表7 DM端口说明

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 信号 | 方向 | 描述 |
| 1 | clk | I | 时钟信号 |
| 2 | reset | I | 异步复位信号，将DM中的值全部清零  1：复位  0：无效 |
| 3 | MemRead | I | 写使能信号  1：可向DM中写入数据  0：无效 |
| 4 | MemWrite | I | 读使能信号  1：可读取DM中数据  0：无效 |
| 5 | addr | I | 32位地址输入信号，对DM指定地址进行读写操作 |
| 6 | WriteData | I | 32位写入数据 |
| 7 | PC | I | 当前PC值 |
| 8 | ReadData | O | 32位输出数据 |

#### (3)功能定义

表8 DM功能定义

|  |  |  |
| --- | --- | --- |
| 序号 | 功能 | 描述 |
| 1 | 复位 | Reset有效时，将DM中的值全部清零 |
| 2 | 读操作 | 读出addr地址对应存储数据到ReadData |
| 3 | 写操作 | 当时钟上升沿来临时，将WriteData写入addr地址对应位置 |

### 7. EXT

#### (1) **模块接口定义**

表4 PC**模块接口定义**

|  |  |
| --- | --- |
| 文件 | 模块接口定义 |
| ext.v | module ext(  input [15:0] in,  input EXTOp,  output [31:0] out); |

#### (2) **接口说明**

表9 EXT端口说明

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 信号 | 方向 | 描述 |
| 1 | in | I | 16位待扩展数据 |
| 2 | EXTop | I | 扩展操作信号  0：进行无符号扩展  1：进行符号扩展 |
| 3 | out | O | 32位输出数据 |

#### (3)功能定义

表10 EXT功能定义

|  |  |  |
| --- | --- | --- |
| 序号 | 功能 | 描述 |
| 1 | 无符号扩展 | 将16位立即数Imm16无符号拓展至32位输出Imm32 |
| 2 | 符号扩展 | 将16位立即数Imm16符号拓展至32位输出Imm32 |

### 8. MUX

#### (1) **模块接口定义**

表4 PC**模块接口定义**

|  |  |
| --- | --- |
| 文件 | 模块接口定义 |
| mux.v | module mux\_2\_32(  input select,  input [31:0] mi1,  input [31:0] mi2,  output [31:0] mo  );  module mux\_4\_32(  input [1:0] select,  input [31:0] mi1,  input [31:0] mi2,  input [31:0] mi3,  input [31:0] mi4,  output [31:0] mo  );  module mux\_3\_5(  input [1:0] select,  input [4:0] mi1,  input [4:0] mi2,  output [4:0] mo  ); |

#### (2) **接口说明**

表9 EXT端口说明

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 信号 | 方向 | 描述 |
| 1 | select | I | 选择信号 |
| 2 | mi（多个）x | I | 输入信号 |
| 3 | mo | O | 输出数据 |

#### (3)功能定义

表10 EXT功能定义

|  |  |  |
| --- | --- | --- |
| 序号 | 功能 | 描述 |
| 1 | 选择信号 | 根据select信号选择输入输出 |

## 三、控制器设计

在P3设计单周期CPU时我参考了牛建伟老师的作业图——

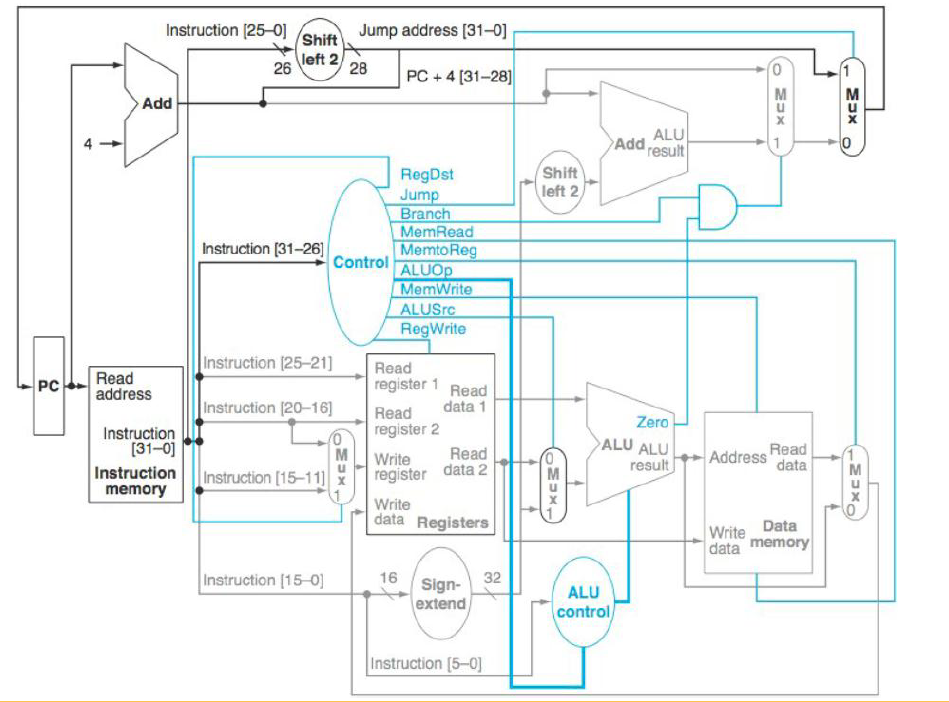


图1 参考CPU电路图

将控制器部分分为Control模块和ALU Control模块，不过在P4中我选择将Control模块和ALU Control模块合并，从而达到减少接口以简化电路的目的。

### 1. Control模块**端口与功能说明**

表11 Control模块端口说明

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 信号 | 方向 | 描述 |
| 1 | Op [5:0] | I | 6位控制信号 |
| 2 | Func [5:0] | I | 6位控制信号 |
| 3 | RegDst [1:0] | O | 00：将Rt作为GRF的A3写入地址  01：将Rd作为GRF的A3写入地址  10：将h1f作为GRF的A3写入地址 |
| 4 | ALUSrc | O | 0：将RD2作为参与ALU计算的第二个数据  1：将EXT输出作为参与ALU计算的第二个数据 |
| 5 | MemToReg [1:0] | O | 00：将ALU计算结果写入GRF  01：将DM输出值写入GRF  10：将填充到高位的 16位立即数写入GRF |
| 6 | RegWrite | O | 0：GRF写使能信号无效  1：GRF写使能信号有效 |
| 7 | MemRead | O | 0：DM读使能信号无效  1：DM读使能信号有效 |
| 8 | MemWrite | O | 0：DM写使能信号无效  1：DM写使能信号有效 |
| 9 | Branch [1:0] | O | 00：默认操作  01：当前指令为beq指令  10：当前指令为jar指令  11：当前指令为jr指令 |
| 10 | ExtOp | O | 0：进行无符号扩展  1：进行符号扩展 |
| 11 | ALUOp [2:0] | O | ALU功能的选择信号：  000：ALU进行与运算  001：ALU进行或运算  010：ALU进行加法运算  011：ALU进行减法运算 |

### 2. Control模块**真值表**

表12 Control模块**真值表**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令 | addu | subu | ori | lw | sw | beq | lui | jal | jr |
| Op字段 | 000000 | 000000 | 001101 | 100011 | 101011 | 000100 | 001111 | 000011 | 000000 |
| Func字段 | 100001 | 100011 | xxxxxx | | | | | | 001000 |
| RegDst | 01 | 01 | 00 | 00 | 00 | 00 | 00 | 10 | 01 |
| ALUSrc | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| MemToReg | 00 | 00 | 00 | 01 | 00 | 00 | 10 | 11 | 00 |
| RegWrite | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 |
| MemRead | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| MemWrite | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| Branch | 00 | 00 | 00 | 00 | 00 | 01 | 00 | 10 | 11 |
| ExtOp | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| ALUOp | 010 | 011 | 001 | 010 | 010 | 011 | 111 | 111 | 010 |

## 四、测试程序

### 1. test branch代码

module *tb*;

*// Inputs*

    reg clk;

    reg reset;

*// Instantiate the Unit Under Test (UUT)*

    mips uut (

        .clk(clk),

        .reset(reset)

    );

    initial begin

*// Initialize Inputs*

        clk = 0;

        reset = 1;

        #50;

        reset = 0;

*// Wait 100 ns for global reset to finish*

        #100;

*// Add stimulus here*

    end

    always #10 clk = ~clk;

endmodule

### 2. mars测试程序

ori $28, $0, 0x0

ori $29, $0, 0x0

ori $1, $0, 0x3456

addu $1, $1, $1

lw $1, 4($0)

sw $1, 4($0)

lui $2, 0x7878

subu $3, $2, $1

lui $5, 0x1234

ori $4, $0, 0x5

nop

sw $5, 65535($4)

lw $3, 65535($4)

beq $3, $5, 0x3

nop

beq $0, $0, 0x11

nop

ori $7, $3, 0x404

beq $7, $3, 0xe

nop

lui $8, 0x7777

ori $8, $8, 0xffff

subu $0, $0, $8

ori $0, $0, 0x1100

addu $10, $7, $6

ori $8, $0, 0x0

ori $9, $0, 0x1

ori $10, $0, 0x1

addu $8, $8, $10

beq $8, $9, 0xfffe

jal 0x3088

nop

addu $10, $10, $10

beq $0, $0, 0xffff

addu $10, $10, $10

jr $31

### 3. 测试程序结果

@00003000: $28 <= 00000000

@00003004: $29 <= 00000000

@00003008: $ 1 <= 00003456

@0000300c: $ 1 <= 000068ac

@00003010: $ 1 <= 00000000

@00003014: \*00000004 <= 00000000

@00003018: $ 2 <= 78780000

@0000301c: $ 3 <= 78780000

@00003020: $ 5 <= 12340000

@00003024: $ 4 <= 00000005

@0000302c: \*00000004 <= 12340000

@00003030: $ 3 <= 12340000

@00003044: $ 7 <= 12340404

@00003050: $ 8 <= 77770000

@00003054: $ 8 <= 7777ffff

@00003060: $10 <= 12340404

@00003064: $ 8 <= 00000000

@00003068: $ 9 <= 00000001

@0000306c: $10 <= 00000001

@00003070: $ 8 <= 00000001

@00003070: $ 8 <= 00000002

@00003078: $31 <= 0000307c

@00003088: $10 <= 00000002

@00003080: $10 <= 00000004

## 五、思考题

### 1.数据通路设计（L0.T2）

#### 1）根据你的理解，在下面给出的DM的输入示例中，地址信号addr位数为什么是[11:2]而不是[9:0]？这个addr信号又是从哪里来的？



因为我们设计的cpu的lw/sw指令以字节为单位，地址是4的倍数。而DM是以字为单位，所以对DM进行操作时，addr要右移两位来对齐字，由此位数是[11:2]。addr信号来自ALU计算结果。

#### 2）在相应的部件中，**reset的优先级**比其他控制信号（不包括clk信号）都要**高**，且相应的设计都是**同步复位**。清零信号reset是针对哪些部件进行清零复位操作？这些部件为什么需要清零？

针对以下部件进行清零复位操作：PC、GRF、DM

PC 的初始值为 0x00003000，需要清零以回到初始值重新读取指令。

GRF 中的所有寄存器初始值为 0x00000000，DM 中所有地址的初始值为 0x00000000，需要清零，否则可能会影响之后的指令执行。

### 2.控制器设计（L0.T4）

#### 1）列举出用Verilog语言设计控制器的几种编码方式（至少三种），并给出代码示例。

① 利用case语句进行编码

module *controller*(

    input [5:0] op,

    input [5:0] Fu,

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

    output [2:0] ALUOp,

    output [1:0] RegDst,

    output ALUSrc,

    output [1:0] MemtoReg,

    output RegWrite,

    output MemRead,

    output MemWrite,

    output [1:0] Branch,

    output EXTOp);

    always@(\*)

    begin

        case (op)

        6'b000000:

        begin

ALUOp[2] <=0;

            ALUOp[1] <=0;

            ALUOp[0] <=0;

            RegDst[1] <=0;

            RegDst[0] <=1;

            ALUSrc <=0;

MemtoReg[1] <=0;

            MemtoReg[0] <=0;

            RegWrite <=1;

            MemRead <=0;

            MemWrite <=0;

            ExtOp<=0;

            Branch1<=0;

            Branch2<=0;

        end

……

        endcase

    end

endmodule

② 利用assign语句进行与或识别

module *controller*(

    input [5:0] op,

    input [5:0] Fu,

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

    output [2:0] ALUOp,

    output [1:0] RegDst,

    output ALUSrc,

    output [1:0] MemtoReg,

    output RegWrite,

    output MemRead,

    output MemWrite,

    output [1:0] Branch,

    output EXTOp);

    wire R, addu, subu, ori, lw, sw, beq, lui, jal, jr;

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

    assign R        = !op[5] && !op[4] && !op[3] && !op[2] && !op[1] && !op[0];

    assign addu =  Fu[5] && !Fu[4] && !Fu[3] && !Fu[2] && !Fu[1] &&  Fu[0] && R;

    assign subu =  Fu[5] && !Fu[4] && !Fu[3] && !Fu[2] &&  Fu[1] &&  Fu[0] && R;

    assign jr   = !Fu[5] && !Fu[4] &&  Fu[3] && !Fu[2] && !Fu[1] && !Fu[0] && R;

    assign ori  = !op[5] && !op[4] &&  op[3] &&  op[2] && !op[1] &&  op[0];

    assign lw   =  op[5] && !op[4] && !op[3] && !op[2] &&  op[1] &&  op[0];

    assign sw   =  op[5] && !op[4] &&  op[3] && !op[2] &&  op[1] &&  op[0];

    assign beq  = !op[5] && !op[4] && !op[3] &&  op[2] && !op[1] && !op[0];

    assign lui  = !op[5] && !op[4] &&  op[3] &&  op[2] &&  op[1] &&  op[0];

    assign jal  = !op[5] && !op[4] && !op[3] && !op[2] &&  op[1] &&  op[0];

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

*/\*000& 001| 010+ 011-\*/*

    assign ALUOp[2]         = jal||lui;

    assign ALUOp[1]         = beq||sw||lw||lui||addu||subu||jr||jal;

    assign ALUOp[0]         = beq||lui||ori||subu||jal;

    assign RegDst[1]        = jal;

    assign RegDst[0]        = R;

    assign ALUSrc           = lw||sw||ori;

    assign MemtoReg[1]  = lui||jal;

    assign MemtoReg[0]  = lw||jal;

    assign RegWrite         = R||lui||ori||lw||jal;

    assign MemRead      = lw;

    assign MemWrite         = sw;

    assign Branch[1]        = jal||jr;

    assign Branch[0]        = beq||jr;

    assign EXTOp            = ori;

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

endmodule

③ 利用宏定义

module *controller*(

    input [5:0] op,

    input [5:0] Fu,

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

    output [2:0] ALUOp,

    output [1:0] RegDst,

    output ALUSrc,

    output [1:0] MemtoReg,

    output RegWrite,

    output MemRead,

    output MemWrite,

    output [1:0] Branch,

output EXTOp);

`define R 6'b000000

    always@(\*)

    begin

        case (op)

        `R:

        begin

ALUOp[2] <=0;

            ALUOp[1] <=0;

            ALUOp[0] <=0;

            RegDst[1] <=0;

            RegDst[0] <=1;

            ALUSrc <=0;

MemtoReg[1] <=0;

            MemtoReg[0] <=0;

            RegWrite <=1;

            MemRead <=0;

            MemWrite <=0;

            ExtOp<=0;

            Branch1<=0;

            Branch2<=0;

        end

……

        endcase

    end

endmodule

#### 2）根据你所列举的编码方式，说明他们的优缺点。

case方法较为直观，case的6位op与指令的对应不够直观，需要通过注释等实现，代码长度长。

assign方法采用与或门阵列，更接近底层实现，代码长度较短，但不能简单对应到各指令。

宏定义法相比case,将指令直接定义为对应的case ，更加直观；缺点是代码长度也很长。

### 3.在线测试相关信息（L0.T5）

#### 1）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 。

《MIPS32® Architecture For Programmers Volume II: The MIPS32® Instruction Set》中addi指令的Operation部分为：

temp  (GPR[rs]31||GPR[rs]31..0) + sign\_extend(immediate) if temp32  temp31 then

SignalException(IntegerOverflow) else

GPR[rt]  temp

endif

addiu指令的Operation部分为：

temp  GPR[rs] + sign\_extend(immediate) GPR[rt] temp

忽略溢出的前提下，addi只完成GPR[rt] temp  GPR[rs] + sign\_extend(immediate)，行为与addiu相同。

add指令的Operation部分为：

temp  (GPR[rs]31||GPR[rs]31..0) + (GPR[rt]31||GPR[rt]31..0) if temp32  temp31 then

SignalException(IntegerOverflow) else

GPR[rd]  temp

endif

addu指令的Operation部分为：

temp GPR[rs] + GPR[rt]

GPR[rd] temp

忽略溢出的前提下，add只完成GPR[rd] temp  GPR[rs] + GPR[rt]，行为与addu相同。

#### 2）根据自己的设计说明单周期处理器的优缺点。

优点：设计简单，容易实现。

缺点：不同指令需要的指令周期不同，单周期处理器的吞吐量低、速度慢。

#### 3）简要说明jal、jr和堆栈的关系。

jal和jr是配套使用的，jal调用函数，jr用于函数返回。

使用jal时，将PC+4的值存入$ra寄存器中，函数调用结束后使用jr $ra指令返回到调用函数的下一条指令。

在调用函数时，为了避免函数内部操作对$ra寄存器或其他在函数外部使用的寄存器进行修改，应该将寄存器的值存入堆栈中，调用结束后再从堆栈取出。