P4 Verilog 实现单周期 CPU 实验报告

18373599 崔建彬

一．实现内容及要求

1.处理器为 32 位处理器。

2.处理器应支持的指令集为： 处理器应支持指令集为：{addu, subu, ori, lw, sw, beq, lui, jal, jr,nop}。

3.nop 机器码为 0x00000000， 即空指令，不进行任何有效行为（修改寄存器等）

4.addu,subu 可以不支持溢出。

5.处理器为单周期设计。

6.不需要考虑延迟槽。

7.需要采用模块化和层次化设计。

8.顶层文件为 mips.v，接口定义如下：

文件	模块接口定义

mips.v	module mips(clk,reset);
	input clk; //clock
	input reset; //reset

二．模块说明

1.PC

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

模块接口

信号名	方向	功能描述
Clk	I	时钟信号
Reset	I	复位信号: 0 无效 1 复位
next_pc	I	更新的 PC 值（时钟上升沿更新）
pc	O	P 当前 pc 的值

```
//////////////////////////////////////////////////////////////////////////////
//////////////
```

```
module PC(
    input [31:0] next_pc,
    input clk,
    input reset,
    output reg [31:0] PC
    );
    always @(posedge clk)begin
        if(reset==1) begin
            PC <= 32'h00003000;
        end
        else begin
```

```
            PC <= next_pc;
        end
    end


endmodule
```

2. IM

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

模块接口

信号名	方向	功能描述
PC[31:0]	I	32 位 PC 地址
Instruction[31:0]	O	32 位当前指令

```
module IM(
    input [31:0] PC,
    output [31:0] Instruction
    );

    reg [31:0]im [0:1024];
    initial begin
        $readmemh("code.txt",im);
    end
    assign Instruction = im[PC[11:2]];
```

```
endmodule
```

3. ALU.v

文件	模块接口定义
ALU.v	module ALU(input [31:0] A, input [31:0] B, input [2:0] ALUCtrl, output reg[31:0] Result, output zero);

模块接口

信号名	方向	功能描述
A[31:0]	I	32 位输入数据 A
B[31:0]	I	32 位输入数据 B
ALUCtrl[2:0]	I	控制信号 000：加法 001：减法 010：与运算 011:或运算
Result[31:0]	O	32 位数据输出
zero	O	A,B 是否相等的标志信号 1：相等 0：不相等

```
module ALU(

    input [31:0] A,

    input [31:0] B,

    input [2:0] ALUCtrl,

    output reg [31:0] Result,

    output zero

    );
```

```
    always @(*)begin
        case(ALUCtrl)
            3'b000:begin
                Result <= A + B;
            end
            3'b001:begin
                Result <= A - B;
            end
            3'b010:begin
                Result <= A | B;
            end
            3'b011:begin
                Result <= A & B;
            end
            default begin
                Result <= 0;
            end
        endcase
    end
    assign zero = (A==B)? 1'b1:1'b0;
endmodule
```

4.GRF

文件	模块接口定义
GRF.v	module GRF(input clk, input reset, input RegWrite, input [4:0] RA1, input [4:0] RA2, input [4:0] WA, input [31:0] WD, input [31:0] PC, output [31:0] RD1, output [31:0] RD2);

模块接口

信号名	方向	功能描述
WD[31:0]	I	写入数据的输入
RA1[4:0]	I	读入寄存器地址 1
RA2[4:0]	I	读入寄存器地址 2
WA[4:0]	I	写入寄存器地址
clk	I	时钟信号
reset	I	复位信号 0：无效 1：复位
RegWrite	I	是否可以写入的信号 0：不可写入 1：可写入
PC[31:0]	I	当前 PC 地址
RD1[31:0]	O	32 位数据输出 1
RD2[31:0]	O	32 位数据输出 2

```
module GRF(

    input clk,

    input reset,

    input [4:0] RA1,

    input [4:0] RA2,

    input [4:0] WA,

    input [31:0] WD,
```

```
output [31:0] RD1,
output [31:0] RD2,
input RegWrite,
input [31:0] PC
);
reg [31:0]Register[0:31];
integer i;

assign RD1 = Register[RA1];
assign RD2 = Register[RA2];
initial begin
    for(i=0;i<32;i=i+1)begin
        Register[i]<=0;
    end
end

always @(posedge clk)begin
    if(reset == 1)begin
        for(i=0;i<32;i=i+1)begin
            Register[i]<=0;
        end
    end
    else if(RegWrite==1&&WA!=5'b00000)begin
        Register[WA] <= WD;
```

```
            $display("@%h: $%d <= %h", PC, WA,WD);
        end
    end

endmodule
```

5.DM

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

模块接口

信号名	方向	功能描述
Clk	I	时钟信号
Reset	I	复位信号 0:不复位 1:复位
MemWrite	I	读写控制信号 0:无操作 1：写操作
MemRead	I	读写控制信号 0:无操作 1：读操作
MemAddr[31:0]	I	操作寄存器地址
WriteData[31:0]	I	要被写入的 32 位数据
PC[31:0]	I	当前 PC
ReadData[31:0]	O	32 位数据输出

```
module DM(
    input clk,
```

```
input reset,
input MemWrite,
input MemRead,
input [31:0] MemAddr,
input [31:0] MemData,
input [31:0] PC,
output [31:0] ReadData
);
reg [31:0] dm [0:1023];
integer i;
initial begin
   for(i=0;i<1024;i=i+1)begin
      dm[i] <= 0;
   end
end
assign ReadData = MemRead?dm[MemAddr[11:2]]:0;
always @(posedge clk)begin
   if(reset==1)begin
      for(i=0;i<1024;i=i+1)begin
         dm[i] <= 0;
      end
   end
   else begin
        if(MemWrite == 1) begin
```

```
            dm[MemAddr[11:2]] <= MemData;
            $display("@%h: *%h <= %h",PC, MemAddr,MemData);
              end
        end
    end


endmodule
```

6. EXT.v

文件	模块接口定义
EXT.v	module EXT(input [15:0] in, input ExtOp, output[31:0] out);

模块接口

信号名	方向	功能描述
In[15:0]	I	16 位数据输入
Out[31:0]	O	32 位转化后输出数据
ExtOp	I	控制信号 0：符号扩展 1：无符号扩展

```
    module EXT(
     input [15:0] in,
     output [31:0] out,
     input ExtOp
     );
    assign out =(ExtOp==0) ? {{16{in[15]}},in}: {{16{1'b0}},in};//
与时钟上升沿无关
endmodule
```

7. Controller.v

文件	模块接口定义
controller.v	module Controller(input [5:0] op, input [5:0] func, output reg [2:0] ALUCtrl, output reg [1:0] RegDst, output reg [1:0] MemtoReg, output reg MemRead, output reg MemWrite, output reg ALUSrc, output reg RegWrite, output reg ExtOp, output reg [1:0]branch);

模块接口

信号名	方向	功能描述
op[5:0]	I	6 位 opcode 段
func[5:0]	I	6 位 func 段
ALUCtrl[2:0]	O	ALU 控制信号
RegDst[1:0]	O	GRF 写入数据的选择信号
MemToReg[1:0]	O	GRF 写入数据控制信号
MemRead	O	DM 读信号
MemWrite	O	DM 写信号
ALUSrc	O	ALU 的 B 输入选择信号
RegWrite	O	寄存器堆写入信号
ExtOp	O	Ext 选择信号
Branch[1:0]	O	next _ pc 选择信号

三. 控制器设计思路

1.next_pc 有 4 种选择，一个是 pc4,一个是 beq 跳转的 pc 值，一个是 jal 跳转的立即数 pc 值，一个是 jr 跳转的 32 位数的 pc 值，所以应用 branch[1:0]来选择，同时配合 zero 信号来判断是否满足 beq 跳转要求。

2. ALUSrc: ALU 的 B 输入由两个选择，一个是 RD2,一个是 imm，所以需要一个 ALUSrc 选择控制信号

3.RegWrite:控制寄存器堆写入信号

4.MemRead/MemWrite， 控制内存写入读出信号

5.ExtOp: 扩展信号，10 条指令中，除了 ori 外，均无需无符号扩展，只有 ori 进行无符号扩展。

6.MemToReg, 写入 GRF 的数据，可能位 alu 的输出 Result,可能为 dm 的读入输出 ReadData,可能为 lui 的输出，可能为 PC+4 的值 PC4，所以需要两位控制信号

7.ALUCtrl，控制 ALU 进行的运算，与这十条指令有关的运算有加法/减法/或运算。

信号名	方向	功能描述
Op[5:0]	I	6 位 opcode 段
Func[5:0]	I	6 位 func 段
RegDst[1:0]	0	写入地址控制信号 2’b00: 选择 Rt 2’b01
ALUSrc	0	ALU 第二操作数选择控制 0:选择 ReadData2 1：选择立即数
RegWrite	0	GRF 写入控制
MemRead	0	DM 读信号
MemWrite	0	DM 写信号
MemToReg[1:0]	0	GRF 写入数据的选择信号 2’b00: alu 的输出 2’b01: dm 的输出 2’b10: lui 的输出 2’b11:pc4 的输出
ExtOp	0	高位扩展方式选择信号 0：符号扩展 1:非符号扩展
Branch[1:0]	0	写入 NPC 信号： 2’b00: pc4 2’b01: jal 2’b10: beq

		2’b11: jr
ALUCtrl[2:0]	0	ALU 的控制信号: 3’b000: add 3’b001: sub 3’b010:or

	lw	sw	beq	lui	ori	nop	jal	addu	subu	jr
Op	100011	101011	000100	001111	001101	0000000	000011	000000	000000	000000
Func5								100001	100011	001000
RegDst[1:0]	00	0x	0x	00	00	xx	10	01	01	01
ALUSrc	1	1	0	x	1	x	x	0	0	0
RegWrite	1	0	0	1	1	x	1	1	1	1
MemRead	1	0	0	0	0	x	0	0	0	0
MemWrite	0	1	0	0	0	x	0	0	0	0
MemToReg[1:0]	01	00	00	10	00	xx	11	00	00	xx
EXTOp	0	0	0	0	1	x	x	0	0	0
Branch[1:0]	00	00	01/00	00	00	00	10	00	00	11
ALUCtrl[2:0]	000	000	xxx	xxx	001	xxx	xxx	000	001	xxx

四. 数据通路设计及 TestBench

数据通路如下

指令	PC	IM.A	GRF				ALU		DM		EXT	Shift
			RA1	RA2	WA	WD	ALU	B	A	WD		
R 型	PC4	PC	Rs	Rt	Rd	ALU	RF.RD1	RF.RD2				
lw	PC4	PC	Rs		Rt	DM.RD	RF.RD1	Signed.imm16	ALU		imm16	
sw	PC4	PC	Rs	Rt			RF.RD1	Signed.imm16	ALU	RF.RD2	imm16	
beq	PC4/PCbeq	PC	Rs	Rt			RF.RD1	RF.RD2			imm16	Signed_ext
ori	PC4	PC	Rs		Rt	ALU	RF.RD1	Zero.imm16			imm16	
lui	PC4	PC			Rt	imm+16{0}						
nop	PC4	PC										
jal	PCjal	PC			0x1f							
jr	ALU	PC	RS	Rt	Rd		RF.RD1	RF.RD2				

1.mux.v

模块接口

文件	模块接口定义
mux.v	module Mux(input [4:0] Rt, input [4:0] Rd, input [31:0] RD1, input [31:0] RD2, input [31:0] imm32, input [31:0] Result, input [15:0] imm16, input [31:0] ReadData, input [1:0] RegDst, input ALUSrc, input [1:0] MemToReg, input [31:0] PC4, input [31:0] PCbeq, input [31:0] PCjal, input zero, input [1:0] branch,

	output reg[4:0] WA, output reg[31:0] B, output reg[31:0] WD, output reg[31:0] next_pc);

2.mips.v

文件	模块接口定义
mips.v	module mips(input clk, input reset);

3.TestBench

```
module test_cpu;

  // Inputs

  reg clk;

  reg reset;



  // Instantiate the Unit Under Test (UUT)

  mips uut (

      .clk(clk),

      .reset(reset)

  );

  always #10 clk = ~clk;

  initial begin

      // Initialize Inputs
```

```
        clk = 0;
        reset = 1;

        // Wait 100 ns for global reset to finish
        #98;
        reset = 0;
        // Add stimulus here
    end
endmodule
```

五．测试程序

```
  ori $1,11
  ori $2,22
  ori $3,33
  lui $4,12
  lui $5,23
  lui $6,24
  lui $7,25
  lui $8,34
  lui $9,12
  addu $10,$9,$9
  addu $11,$2,$3
  addu $12,$5,$6
  subu $13,$3,$5
```

```
subu $14,$5,$4

subu $15,$2,$6

nop

lui $16,12

beq $9,$16, next #应跳转

nop

lui $1,1

lui $2,1

lui $3,1

lui $4,1

haha:

lui $5,1

lui $6,1

lui $7,1

lui $8,1

lui $9,1

lui $10,1

next:

beq $1,$2,haha #应不跳转，否则死循环

sw $1,0($0)

sw $2,4($0)

sw $3,8($0)

sw $4,12($0)

sw $5,16($0)
```

```
sw $6,20($0)

sw $7,24($0)

lw $17,0($0)

lw $18,4($0)

jal ok

lw $19, 8($0)

jal end

ok:

lw $0,0($0)

jr $31

end:

subu $3,$3,$0

subu $31,$0, $31
```

机器码

```
3421000b

34420016

34630021

3c04000c

3c050017

3c060018

3c070019

3c080022

3c09000c
```

01295021

00435821

00a66021

00656823

00a47023

00467823

00000000

3c10000c

1130000b

00000000

3c010001

3c020001

3c030001

3c040001

3c050001

3c060001

3c070001

3c080001

3c090001

3c0a0001

1022fff9

ac010000

```
ac020004

ac030008

ac04000c

ac050010

ac060014

ac070018

8c110000

8c120004

0c000c2a

8c130008

0c000c2c

8c000000

03e00008

00601823

001ff823
```

MARS 结果

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0
$at	1	11
$v0	2	22
$v1	3	33
$a0	4	786432
$a1	5	1507328
$a2	6	1572864
$a3	7	1638400
$t0	8	2228224
$t1	9	786432
$t2	10	1572864
$t3	11	55
$t4	12	3080192
$t5	13	-1507295
$t6	14	720896
$t7	15	-1572842
$s0	16	786432
$s1	17	11
$s2	18	22
$s3	19	33
$s4	20	0
$s5	21	0
$s6	22	0
$s7	23	0
$t8	24	0
$t9	25	0
$k0	26	0
$k1	27	0
$gp	28	6144
$sp	29	12284
$fp	30	0
$ra	31	-12456
pc		12472
hi		0
lo		0

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x00000000	11	22	33	786432	1507328	1572864	1638400	0
0x00000020	0	0	0	0	0	0	0	0
0x00000040	0	0	0	0	0	0	0	0
0x00000060	0	0	0	0	0	0	0	0
0x00000080	0	0	0	0	0	0	0	0
0x000000a0	0	0	0	0	0	0	0	0
0x000000c0	0	0	0	0	0	0	0	0
0x000000e0	0	0	0	0	0	0	0	0
0x00000100	0	0	0	0	0	0	0	0
0x00000120	0	0	0	0	0	0	0	0
0x00000140	0	0	0	0	0	0	0	0
0x00000160	0	0	0	0	0	0	0	0
0x00000180	0	0	0	0	0	0	0	0
0x000001a0	0	0	0	0	0	0	0	0

魔改后输出

@00003000: $ 1 <= 0000000b

@00003004: $ 2 <= 00000016

@00003008: $ 3 <= 00000021

@0000300c: $ 4 <= 000c0000

@00003010: $ 5 <= 00170000

@00003014: $ 6 <= 00180000

@00003018: $ 7 <= 00190000

@0000301c: $ 8 <= 00220000

@00003020: $ 9 <= 000c0000

@00003024: $10 <= 00180000

@00003028: $11 <= 00000037

@0000302c: $12 <= 002f0000

@00003030: $13 <= ffe90021

@00003034: $14 <= 000b0000

@00003038: $15 <= ffe80016

@00003040: $16 <= 000c0000

@00003078: *00000000 <= 0000000b

@0000307c: *00000004 <= 00000016

@00003080: *00000008 <= 00000021

@00003084: *0000000c <= 000c0000

@00003088: *00000010 <= 00170000

@0000308c: *00000014 <= 00180000

@00003090: *00000018 <= 00190000

@00003094: $17 <= 0000000b

@00003098: $18 <= 00000016

@0000309c: $31 <= 000030a0

@000030a8: $ 0 <= 0000000b

@000030a0: $19 <= 00000021

@000030a4: $31 <= 000030a8

@000030b0: $ 3 <= 00000021

@000030b4: $31 <= ffffcf58

ISE 输出

```
@00003000: $ 1 <= 0000000b
@00003004: $ 2 <= 00000016
@00003008: $ 3 <= 00000021
@0000300c: $ 4 <= 000c0000
@00003010: $ 5 <= 00170000
@00003014: $ 6 <= 00180000
@00003018: $ 7 <= 00190000
@0000301c: $ 8 <= 00220000
@00003020: $ 9 <= 000c0000
@00003024: $10 <= 00180000
@00003028: $11 <= 00000037
@0000302c: $12 <= 002f0000
@00003030: $13 <= ffe90021
@00003034: $14 <= 000b0000
@00003038: $15 <= ffe80016
@00003040: $16 <= 000c0000
@00003078: *00000000 <= 0000000b
@0000307c: *00000004 <= 00000016
@00003080: *00000008 <= 00000021
@00003084: *0000000c <= 000c0000
@00003088: *00000010 <= 00170000
@0000308c: *00000014 <= 00180000
```

@00003090: *00000018 <= 00190000
@00003094: $17 <= 0000000b
@00003098: $18 <= 00000016
@0000309c: $31 <= 000030a0
@000030a0: $19 <= 00000021
@000030a4: $31 <= 000030a8
@000030b0: $ 3 <= 00000021
@000030b4: $31 <= ffffcf58
内存区：

	0	1	2	3
0x0	0000000B	00000016	00000021	000C0000
0x4	00170000	00180000	00190000	00000000
0x8	00000000	00000000	00000000	00000000
0xC	00000000	00000000	00000000	00000000
0x10	00000000	00000000	00000000	00000000
0x14	00000000	00000000	00000000	00000000
0x18	00000000	00000000	00000000	00000000
0x1C	00000000	00000000	00000000	00000000
0x20	00000000	00000000	00000000	00000000
0x24	00000000	00000000	00000000	00000000
0x28	00000000	00000000	00000000	00000000
0x2C	00000000	00000000	00000000	00000000

寄存器堆：

	0	1	2	3
0x0	00000000	0000000B	00000016	00000021
0x4	000C0000	00170000	00180000	00190000
0x8	00220000	000C0000	00180000	00000037
0xC	002F0000	FFE90021	000B0000	FFE80016
0x10	000C0000	0000000B	00000016	00000021
0x14	00000000	00000000	00000000	00000000
0x18	00000000	00000000	00000000	00000000
0x1C	00000000	00000000	00000000	FFFFCF58

六．思考题

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？

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

Lw,sw 存储以字节为单位，而设计的 DM 以字为单位，我们的 MemAddr 也是以字节为单位，4 个字节一个字，所以应右移两位，才是真正的 MemAddr。Addr 信号来自于 ALU 输出 Result，取 Result 中[11:2]位

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？

PC:复位到 0x00003000 处

DM：复位清空内存

GRF:复位清空内存

清零可以进行下一个数据集测试，而且可以将 PC 值复位到 0x00003000 而不是 0x00000000.

3.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. 用 case 语句实现操作码和控制信号的值之间的对应

```
always @(*)begin
      case(op)
      6'b100011: //lw
      begin
          RegDst <= `Rt;
          RegWrite <= 1;
          ALUSrc <= `imm;
          MemtoReg <= `dm;
          ALUCtrl <= `add;
          MemRead <= 1;
          MemWrite <= 0;
          branch <= `pc;
          ExtOp <= 0;
      end
      default:
      begin
          RegDst <= `Rt;
            RegWrite <= 0;
            ALUSrc <= `imm;
            MemtoReg <= `alu;
```

```
            ALUCtrl <= `add;
            MemRead <= 0;
            MemWrite <= 0;
            branch <= `pc;
            ExtOp <= 0;
      end
      endcase
    end
```

2. 用 assign 语句实现控制器；

通过与或逻辑实现控制器

下为部分示意代码

```
module controller (
    input [5:0] op,
    input [5:0] func,
    output MemWrite,
    output MemRead,
    output RegWrite
   );


wire r,lw,sw,beq,lui,ori,jal,jr,addu,subu;
assign r = !op[0]&&!op[1]&&!op[2]&&!op[3]&&!op[4]&&!op[5];
assign lw = op[0]&&op[1]&&!op[2]&&!op[3]&&!op[4]&&op[5];
```

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

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

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

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

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

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

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

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

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

     assign MemRead = lw;

     assign MemWrite = sw;
```

3.利用宏或 define 定义

下为部分示意代码

```
`include "cjbdefine.v"

module Controller(

   input [5:0] op,

   input [5:0] func,

   output reg [2:0] ALUCtrl,

   output reg [1:0] RegDst,

   output reg [1:0] MemtoReg,
```

```
    output reg MemRead,
    output reg MemWrite,
    output reg ALUSrc,
    output reg RegWrite,
    output reg ExtOp,
    output reg [1:0]branch
    );
always @(*)begin
       case(op)
       6'b000000: begin

           case(func)
               6'b100001: //addu
               begin
                   RegDst <= `Rd;
                   RegWrite <= 1;
                   ALUSrc <= `RD2;
                   MemtoReg <= `alu;
                   ALUCtrl <= `add;
                   MemRead <= 0;
                   MemWrite <= 0;
                   branch <= `pc;
                   ExtOp <= 0;
```

```
end
6'b100011://subu
begin
    RegDst <= `Rd;
    RegWrite <= 1;
    ALUSrc <= `RD2;
    MemtoReg <= `alu;
    ALUCtrl <= `sub;
    MemRead <= 0;
    MemWrite <= 0;
    branch <= `pc;
    ExtOp <= 0;
end
6'b001000: //jr
begin
    RegDst <= `Rd;
    RegWrite <= 0;
    ALUSrc <= `RD2;
    MemtoReg <= `alu;
    ALUCtrl <= `add;
    MemRead <= 0;
    MemWrite <= 0;
    branch <= `jr;
    ExtOp <= 0;
```

```
end
   6'b000000: //nop
begin
    RegDst <= `Rt;
        RegWrite <= 0;
        ALUSrc <= `imm;
       MemtoReg <= `alu;
        ALUCtrl <= `add;
        MemRead <= 0;
        MemWrite <= 0;
        branch <= `pc;
        ExtOp <= 0;
end
   default:
   begin
    RegDst <= `Rt;
        RegWrite <= 0;
        ALUSrc <= `imm;
       MemtoReg <= `alu;
        ALUCtrl <= `add;
        MemRead <= 0;
        MemWrite <= 0;
        branch <= `pc;
        ExtOp <= 0;
```

```
            end
          endcase
          end
     endcase
end
```

其中 define 内容定义如下：

```
//RegDst
`define Rt 2'b00
`define Rd 2'b01
`define jal 2'b10
//ALUSrc
`define RD2 0
`define imm 1
//MemtoReg
`define alu 2'b00
`define dm 2'b01
`define lui 2'b10
`define pc4 2'b11

//ALUCtrl
`define add 3'b000
`define sub 3'b001
`define or 3'b0010
//branch
```

```
`define pc 2'b00

`define beq 2'b01

//`define jal 2'b10

`define jr 2'b11
```

4.根据你所列举的编码方式，说明他们的优缺点。

第一种及第三种均采取 always @(*)配合 case 情况进行判断，第三种比较于第一种更容易直接看出是哪种信号，缺点在于无法直接看出所写的控制信号的值，容易在 define.v 模块出错。这种 case 判断情况更符合大多数人思维，较为清晰，缺点为代码较为冗长，容易出现错误，不易发现

第二种采取 assign 方式，先定义各条指令为 wire，采用先和后与的方式，写出控制信号的值，代码相对简短。缺点为无法直接看出 0/1 状况，连线过程中容易出错，不易被发现。

5.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 。

溢出说明如下：

```
temp = (GPR[rs]31||GPR[rs]) + (GPR[rt]31||GPR[rt])

if temp32 ≠ temp31 then

SignalException(IntegerOverflow)
```

```
else
GPR[rd] ← temp 31..0
Endif
```

addu 指令操作如下

```
GPR[rd] ← GPR[rs] + GPR[rt]
```

Add 指令将设置 33 位临时变量 temp, temp = GRF[rs]+GRF[rt]。若相加后,temp 第 32 位与第 31 位不等，则抛出溢出，但是 temp 的后 32 位，仍为 GRF[rs]+GRF[rt] 的结果。若不考虑溢出，则其结果仍为 addu 操作后的结果。

Addiu 与 addi 同理

addi 为

```
temp ←= (GPR[rs]31||GPR[rs]) + sign_extend(immediate)
if temp 32 ≠ temp 31 then
SignalException(IntegerOverflow)
else
GPR[rt] ← temp 31..0
endif
```

与上面相同，若不考虑溢出，则 temp 后 32 位计算结果 addiu 与 addi 相同。

5.根据自己的设计说明单周期处理器的优缺点。

优点：结构简单，数据通路清晰，易于添加修改指令。

缺点: 1.所有指令都在一个周期内完成，导致周期时间增加，使得处理器处理速度慢。2.将 IM 和 DM 分开，与实际中应用的处理器不一致。

6.简要说明 jal、jr 和堆栈的关系。

Jal 和 jr 一般一起使用，jal 用于调用函数，并把返回地址传给$31,而 jr 用于返回上一次调用地址，从$31 中读取地址。而堆栈用于存储部分局部变量，在递归中，为了防止返回地址被修改，可将$31 存入堆栈中，待返回时，再将$31 取出。