**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）**

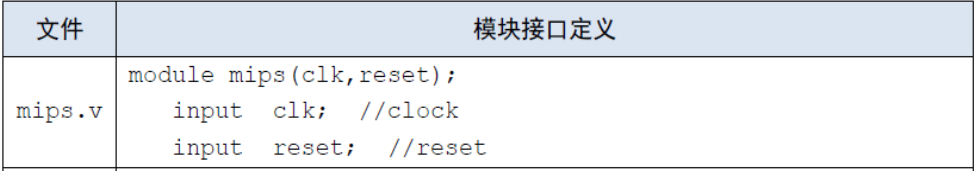
4.addu,subu可以不支持溢出。

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：



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] | O | 写入地址控制信号  2’b00: 选择Rt  2’b01 |
| ALUSrc | O | ALU第二操作数选择控制  0:选择ReadData2  1：选择立即数 |
| RegWrite | O | GRF 写入控制 |
| MemRead | O | DM读信号 |
| MemWrite | O | DM写信号 |
| MemToReg[1:0] | O | 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 | O | 高位扩展方式选择信号  0：符号扩展  1:非符号扩展 |
| Branch[1:0] | O | 写入NPC信号：  2’b00: pc4  2’b01: jal  2’b10: beq  2’b11: jr |
| ALUCtrl[2:0] | O | 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

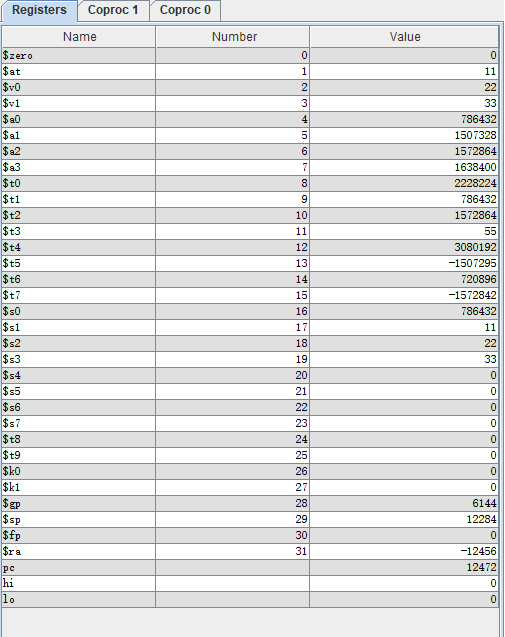
8c000000

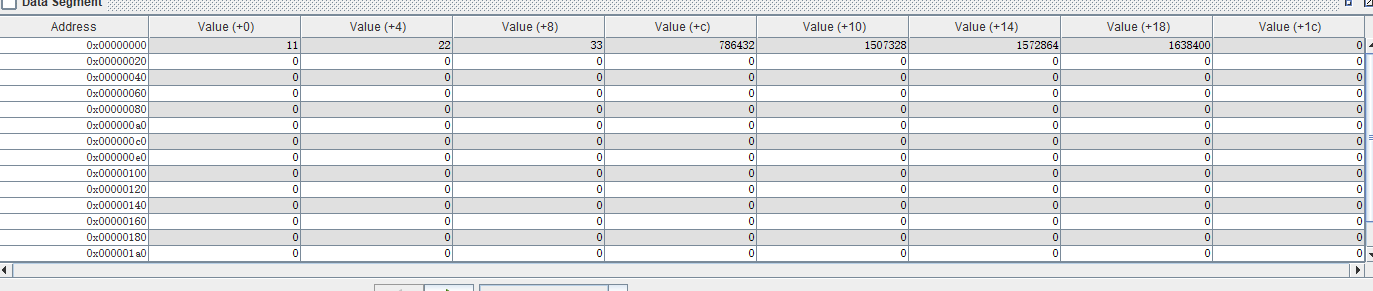
03e00008

00601823

001ff823

MARS结果





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

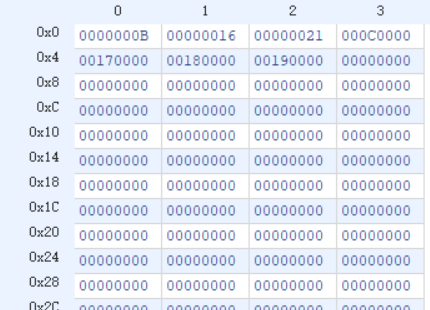
@000030a0: $19 <= 00000021

@000030a4: $31 <= 000030a8

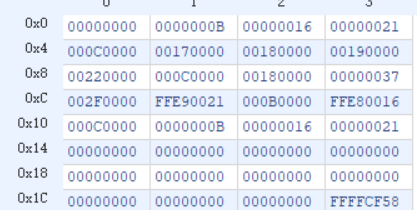
@000030b0: $ 3 <= 00000021

@000030b4: $31 <= ffffcf58

内存区：



寄存器堆：



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？



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。