计算机组成原理　P4 实验报告

一、CPU 设计方案综述

（一）总体设计概述

本 CPU 为 Verilog 实现的单周期 MIPS32 - CPU，支持的指令集包含**{addu, subu, ori, lw, sw, beq, lui, jal, jr,nop}**。为了实现这些功能，CPU 主要包含了 PC_IM、GRF、DM、CU、EXT、ALU、NPC 这些关键模块，采用模块化和层次化设计，顶层有效的驱动信号仅有内置的 clk 时钟信号和异步复位 reset 信号。

（二）关键模块定义

1. PC_IM：程序计数器+指令存储器

PC 用 Verilog 的 32 位 reg 类型变量实现。具有**异步复位**功能，复位值为起始地址 0x00000000。

IM 使用 Verilog 中匹配需求数量的 32 位寄存器的阵列实现，容量为 4KB（32bit×1024 字）。因为一条指令宽度为 32 位，故直接取 PC 提供的指令地址的 2-11 位作为寄存器堆的编号的地址。IM 输出满足 CPU 指令规范的 32 位机器码指令。

端口定义

信号名	方向	描述
clk	I	时间信号
reset	I	复位信号
npc	I	下一周期待执行指令的 32 位地址
pc	O	本周起待执行指令的 32 位地址
Instr	O	当前执行的 32 位机器码指令
shamt	O	指令 10-6 位
rs	O	指令 25-21 位
rt	O	指令 20-16 位
rd	O	指令 15-11 位
imm16	O	指令 15-0 位

Imm26	O	指令 25-0 位

2. NPC：更新程序计数器 PC 的相关逻辑

端口定义

信号名	方向	描述
pc	I	当前执行指令的 32 位地址
		3 位控制信号，根据当前指令的类型、控制取下一条指令地址的方法
		000：其余使程序顺序执行的指令
		001：beq
Br	I	其他：暂时未定义
imm16	I	I 型指令 15-0 位中存储的 16 位立即数
imm26	I	J 型指令 25-0 位中存储的 26 位立即数
RD1	I	输出 a1 指定的寄存器中的 32 位数据
		1 位控制信号，控制 beq 分支指令是否满足条件跳转
		0：条件不成立，不跳转，按照 PC+4 顺序执行指令
PCSrc	I	1：条件成立，跳转到 beq 指令立即数中立即数所指向的地址
PC_4	O	当前执行指令的下一条指令的 32 位地址
npc	O	下一周期待执行指令的 32 位地址

3. CU：根据 32 位机器码指令，判断指令内容，得到控制信号

端口定义

信号名	方向	描述
Instr	I	当前执行的 32 位机器码指令
		3 位控制信号，根据当前指令的类型、控制取下一条指令地址的方法
		000：其余使程序顺序执行的指令
		001：beq
Br	O	其他：暂时未定义
A3Sel	O	2 位控制信号，选择写入寄存器地址的信号

		00：20-16 位作为写入寄存器的编号
		01：15-11 位作为写入寄存器的编号
		其他：未定义
		2 位控制信号，选择写入寄存器的数据的信号
		00：写入 ALU 计算结果
		01：写入 DM 取数据的结果
		10：写入下一条指令的地址 PC+4
WDSel	O	11：未定义
		1 位控制信号，GRF 的写使能信号
RegWrite	O	1：可向 GRF 中写入数据　　0：不可向 GRF 中写入数据
		1 位控制信号，控制 EXT 对 16 位立即数的位扩展方式
EXTop	O	0：零扩展　　1：符号扩展
		1 位控制信号，控制 ALU 的第二个操作数的来源
		0：GRF 中 a2 指定的寄存器中的 32 位数据作为 ALU 的第二个操作数
Bsel	O	1：EXT 扩展结果的 32 位数据作为 ALU 的第二个操作数
		1 位控制信号，DM 的写使能信号
MemWrite	O	1：可向 DM 中写入数据　　0：不可向 DM 中写入数据
		4 位控制信号，控制 ALU 的计算行为
		0000：ScrA + ScrB = ALULout
		0001：ScrA - ScrB = ALULout
		0010：ScrA & ScrB = ALULout
		0011：ScrA | ScrB = ALULout
ALUControl	O	0100：ScrB << 16 = ALULout
		1 位信号，标识指令是否为分支指令
		0：不是分支指令
BRANCH	O	1：是分支指令

具体实现方式

① 取指令 31-26 位的 opcode 部分；取指令 5-0 的 func 部分。

② A. 若 opcode = 0x00 , 则指令为 R 型指令

B．若指令为 R 型指令，且 func = 0x21 , 则指令为 addu；

若指令为 R 型指令，且 func = 0x23 , 则指令为 subu。

③ 若 opcode = 0x0d , 则指令为 ori

④ 若 opcode = 0x04 , 则指令为 beq

⑤ 若 opcode = 0x2b , 则指令为 sw

⑥ 若 opcode = 0x23 , 则指令为 lw

⑦ 若 opcode = 0x0f , 则指令为 lui

⑧ 标志指令类型：若指令为 beq，则指令为分支指令，BRANCH = 1；

若指令为 sw，则指令为存储指令，STORE = 1；

若指令为 lw，则指令为加载指令，LOAD = 1；

⑨ 利用②~⑦得到的具体指令、和⑧得到的指令类型，通过 Priority Encoder + Pull Resistor 和 OR + NOR ，得到 32 位机器码指令对应的控制信号。

Br	000：其余使程序顺序执行的指令 001：beq 其他：暂时未定义
A3Sel	00：其他指令，写入 20-16 位 01：R 型指令，写入 15-11 位 其他：暂时未定义
WDSel	00：其他指令，写入 ALUout 01：LOAD 指令，写入 DMout 其他：暂时未定义
RegWrite	STORE、BRANCH 指令，不用写入 GRF
EXTop	STORE、LOAD 指令，使用符号扩展方法
Bsel	R 型、BRANCH 类指令，不使用扩展 imm 的 32 位数据作为 ALU 的第二个操作数
MemWrite	STORE 指令，需要写入 DM
ALUControl	0000：执行加法 0001：subu，执行减法 0010：and，执行且运算 0011：ori、or 等，执行或运算

0100：lui，执行移位运算

其他：暂时未定义

Tips：1. 使用了 parameter 代替各种指令的 opcode 和 func 部分，便于进行指令的比较与判定

2. 通过多个一位 wire 变量，标记指令类型

4. GRF：通用寄存器组

GRF 中包含 32 个 32 位寄存器，分别对应 0~31 号寄存器，其中 0 号寄存器读取的结果恒为 0

端口定义

信号名	方向	描述
pc	I	当前执行的机器码指令
clk	I	时钟信号
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
WE	I	写使能信号 1：可向 GRF 中写入数据　　0：不可向 GRF 中写入数据
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
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
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
WD	I	32 位数据输入信号
RD1	O	输出 a1 指定的寄存器中的 32 位数据
RD2	O	输出 a2 指定的寄存器中的 32 位数据

模块功能定义

序号	功能名称	描述
1	复位	reset 信号有效时，所有寄存器存储的数值清零，其行为与 logisim 自带部件 register 得 reset 接口完全相同
2	读数据	读出 a1，a2 地址对应寄存器中所存储得数据到 RD1,RD2
3	写数据	当 WE 有效且时钟上升沿来临时，将 WD 写入 A3 所对应得寄存器中。

5. ALU 算数逻辑单元

提供 32 位加、减、或运算及大小比较功能。

端口定义

信号名	方向	描述
ScrA	I	32 位计算数 a
ScrB	I	32 位计算数 b
		4 位控制信号，控制 ALU 的计算行为
		0000：ScrA + ScrB = ALULout
		0001：ScrA - ScrB = ALULout
		0010：ScrA & ScrB = ALULout
		0011：ScrA | ScrB = ALULout
ALUControl	I	0100：ScrB << 16 = ALULout
shamt	I	5 位移位数值，用于移位指令
ALUout	O	32 位 ALU 计算结果
		标识 ALU 的两个计算结果是否相等
ALUequal	O	0：ScrA != ScrB　1: ScrA = ScrB
		标识 ALU 的两个计算结果是否 A>B
ALUgreater	O	0：ScrA <=ScrB　1: ScrA > ScrB
		标识 ALU 的两个计算结果是否 A<B
ALUless	O	0：ScrA >= ScrB　1: ScrA < ScrB

6 .DM：数据存储器

DM 使用 Verilog 中匹配需求数量的 32 位寄存器的阵列实现，容量为 4KB（32bit×1024 字）。因为一条指令宽度为 32 位，故直接取 pc 提供的指令地址的 2-11 位作为寄存器堆的编号的地址。DM 读写的数据均是满足 CPU 指令规范的 32 位数据。

端口定义

信号名	方向	描述
pc	I	当前执行的机器码指令
WE	I	1 位，数据存储器的写使能信号
clk	I	0：不能写入　1：可以写入
reset	I	异步复位信号

A	I	10 位地址输入信号，指示数据在 DM 中存储的地址
addr	I	32 位地址输入信号
WD	I	32 位写入的数据
RD	O	32 位读出的数据

7 .EXT：数据扩展单元

端口定义

信号名	方向	描述
imm26	I	J 型指令 25-0 位中存储的 26 位立即数
		1 位控制信号，控制 EXT 对 16 位立即数的位扩展方式
EXTop	O	0：零扩展　　1：符号扩展
EXTout	O	32 位的立即数的扩展结果

（三）重要机制实现方法

1. 跳转

NPC 模块和 ALU 模块协同工作支持指令 beq 的跳转机制。

二、测试方案

1. ori 指令 & lui 指令

测试程序：

```
ori $0,$zero,0x1
ori $1,$zero,0x2
ori $2,$zero,0x3
ori $3,$zero,0x1234

lui $0,0x4566
lui $7,0x1
lui $8,0x2
lui $9,,0x3
lui $10,0x1234
```

测试结果：

$zero	0	0x00000000
$at	1	0x00000002
$v0	2	0x00000003
$v1	3	0x00001234
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00010000
$t0	8	0x00020000
$t1	9	0x00030000
$t2	10	0x12340000
$t3	11	0x00000000
$t4	12	0x00000000

```
@00003004: $ 1 <= 00000002
@00003008: $ 2 <= 00000003
@0000300c: $ 3 <= 00001234
@00003014: $ 7 <= 00010000
@00003018: $ 8 <= 00020000
@0000301c: $ 9 <= 00030000
@00003020: $10 <= 12340000
ISim>
```

0x0	00000000000000000000000000000000	00000000000000000000000000000010
0x2	00000000000000000000000000000011	00000000000000000001001000110100
0x4	00000000000000000000000000000000	00000000000000000000000000000000
0x6	00000000000000000000000000000000	00000000000000001000000000000000
0x8	00000000000000001000000000000000	00000000000000011000000000000000
0xA	00010010001101000000000000000000	00000000000000000000000000000000
0xC	00000000000000000000000000000000	00000000000000000000000000000000
0xE	00000000000000000000000000000000	00000000000000000000000000000000
0x10	00000000000000000000000000000000	00000000000000000000000000000000
0x12	00000000000000000000000000000000	00000000000000000000000000000000
0x14	00000000000000000000000000000000	00000000000000000000000000000000
0x16	00000000000000000000000000000000	00000000000000000000000000000000
0x18	00000000000000000000000000000000	00000000000000000000000000000000
0x1A	00000000000000000000000000000000	00000000000000000000000000000000
0x1C	00000000000000000000000000000000	00000000000000000000000000000000
0x1E	00000000000000000000000000000000	00000000000000000000000000000000

2. addu 指令和 subu 指令

测试程序：

```
lui $7 ,0xffff
ori $7 ,$7,0xffff
ori $8 ,$zero,0x2
addu $9,$7,$8

ori $10 ,$0,0xffff
ori $11 ,$0,0x2
addu $12,$10,$11

ori $13 ,$0,0x2
ori $14 ,$0,0xff
subu $15,$13,$14

ori $16 ,$0,0x33
ori $17 ,$0,0x22
subu $18,$16,$17
```

测试结果：

$a3	7	0xffffffff
$t0	8	0x00000002
$t1	9	0x00000001
$t2	10	0x0000ffff
$t3	11	0x00000002
$t4	12	0x00010001
$t5	13	0x00000002
$t6	14	0x000000ff
$t7	15	0xffffff03
$s0	16	0x00000033
$s1	17	0x00000022
$s2	18	0x00000011

```
Finished circuit initializ
@00003000: $ 7 <= ffff0000
@00003004: $ 7 <= ffffffff
@00003008: $ 8 <= 00000002
@0000300c: $ 9 <= 00000001
@00003010: $10 <= 0000ffff
@00003014: $11 <= 00000002
@00003018: $12 <= 00010001
@0000301c: $13 <= 00000002
@00003020: $14 <= 000000ff
@00003024: $15 <= ffffff03
@00003028: $16 <= 00000033
@0000302c: $17 <= 00000022
@00003030: $18 <= 00000011
```

0x6	00000000000000000000000000000000	11111111111111111111111111111111
0x8	00000000000000000000000000000010	00000000000000000000000000000001
0xA	00000000000000001111111111111111	00000000000000000000000000000010
0xC	00000000000000010000000000000001	00000000000000000000000000000010
0xE	00000000000000000000000011111111	11111111111111111111111100000011
0x10	00000000000000000000000000110011	00000000000000000000000000100010
0x12	00000000000000000000000000010001	00000000000000000000000000000000

3. 存储、读取指令测试（sw,sh,sb,lw,lh,lb）

测试程序：

```
ori $8, $zero, 0
lui $9,0x1234
ori $9, $9, 0x5678

sw $9, 0($8)
sh $9, 4($8)
sh $9, 10($8)
sb $9, 12($8)
sb $9, 17($8)
sb $9, 22($8)
sb $9, 27($8)

lb $11, 0($8)
lb $12, 1($8)
lb $13,2($8)
lb $14,3($8)
lh $15, 0($8)
lh $16, 2($8)
lw $17,0($8)
```

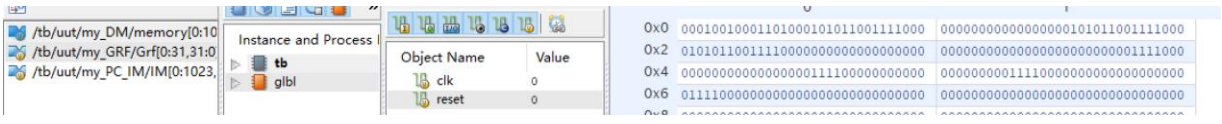
测试结果：

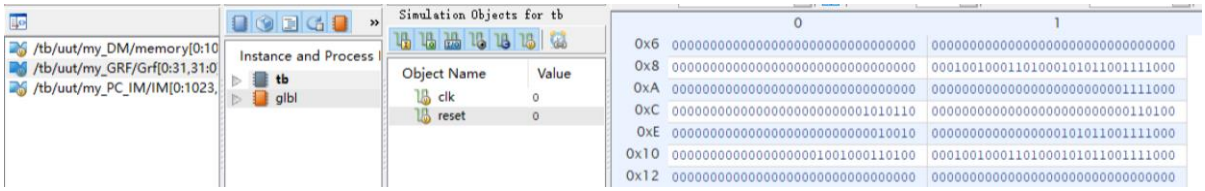
$t0	8	0x00000000
$t1	9	0x12345678
$t2	10	0x00000000
$t3	11	0x00000078
$t4	12	0x00000056
$t5	13	0x00000034
$t6	14	0x00000012
$t7	15	0x00005678
$s0	16	0x00001234
$s1	17	0x12345678
$s2	18	0x00000000

```
Finished circuit initialization pro
@00003000: $ 8 <= 00000000
@00003004: $ 9 <= 12340000
@00003008: $ 9 <= 12345678
@0000300c: *00000000 <= 12345678
@00003010: *00000004 <= 00005678
@00003014: *0000000a <= 56780000
@00003018: *0000000c <= 00000078
@0000301c: *00000011 <= 00007800
@00003020: *00000016 <= 00780000
@00003024: *0000001b <= 78000000
@00003028: $11 <= 00000078
@0000302c: $12 <= 00000056
@00003030: $13 <= 00000034
@00003034: $14 <= 00000012
@00003038: $15 <= 00005678
@0000303c: $16 <= 00001234
@00003040: $17 <= 12345678
ISim>
```

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x00000000	0x12345678	0x00005678	0x56780000	0x00000078	0x00007800	0x00780000	0x78000000	0x00000000
0x00000020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000





4. 跳转指令测试（beq 指令）

测试程序：

```
ori $1 ,$zero ,5
ori $2 ,$zero ,5
ori $3 ,$zero ,6

beq $1 ,$2 ,cut1
lui $10 ,0xffff
cut1:
lui $11 ,0xffff
beq $1 ,$3 ,cut2
lui $12 ,0xffff
cut2:
lui $13 ,0xffff
```

测试结果：

$at	1	0x00000005
$v0	2	0x00000005
$v1	3	0x00000006
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0xffff0000
$t4	12	0xffff0000
$t5	13	0xffff0000

```
Simulator is doing circuit initialization
Finished circuit initialization process.
@00003000: $ 1 <= 00000005
@00003004: $ 2 <= 00000005
@00003008: $ 3 <= 00000006
@00003014: $11 <= ffff0000
@0000301c: $12 <= ffff0000
@00003020: $13 <= ffff0000
```

0x8	00000000000000000000000000000000	00000000000000000000000000000000
0xA	00000000000000000000000000000000	11111111111111110000000000000000
0xC	11111111111111110000000000000000	11111111111111110000000000000000
0xE	00000000000000000000000000000000	00000000000000000000000000000000

5.jal，jr 指令

测试程序：

```
ori $t1, 10
LABEL:
beq $t0, $t1, END
addiu $t0, $t0, 1
j LABEL
END:
jal TEST
   ori $1,0x1234

TEST:
lui $t1, 0xffff
jr $ra
```

测试结果：（程序会进入死循环，这里的结果是单步调试得到的结果）

$zero	0	0x00000000
$at	1	0x00001234
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x0000000a
$t1	9	0xffff0000
$t2	10	0x00000000
$t3	11	0x00000000

```
@00003000: $ 9 <= 0000000a
@00003008: $ 8 <= 00000001
@00003008: $ 8 <= 00000002
@00003008: $ 8 <= 00000003
@00003008: $ 8 <= 00000004
@00003008: $ 8 <= 00000005
@00003008: $ 8 <= 00000006
@00003008: $ 8 <= 00000007
@00003008: $ 8 <= 00000008
@00003008: $ 8 <= 00000009
@00003008: $ 8 <= 0000000a
@00003010: $31 <= 00003014
@00003018: $ 9 <= ffff0000
@00003014: $ 1 <= 00001234
@00003018: $ 9 <= ffff0000
@00003014: $ 1 <= 00001234
```

三、思考题

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

答：

①在 DM 内部的存储中，数据以字为单位存储。CPU 在写入数据时只能按照字来写入（即使是 sb、sh，也同样是经过了数据的转换再按照字存入的）。如此一来，地址信号的后两位一定是 0。

我们使用寄存器堆来实现 DM 的功能，地址信号转换为寄存器堆中寄存器的编号使用；将一定为零的、没有意义的末两位 0 去掉，就可以达到地址信号转换为寄存器堆中寄存器的编号使用的目的，简单便捷。

②addr 信号：加载和存储类指令中，基地址加偏移量在 ALU 元件中的运算结果。

2. 思考 Verilog 语言设计控制器的译码方式，给出代码示例，并尝试对比各方式的优劣。

① 直接对比 6 位 opcode、6 位 func 来确定指令类型，再根据指令类型确定控制信号。

```
② `timescale 1ns / 1ps
③ module CU (
④     input [31:0] Instr,
⑤ 
⑥     output [2:0] Br,
⑦     output [2:0] A3Sel,
⑧     output [2:0] WDSel,
⑨     output RFWr,
⑩     output EXTop,
⑪     output BSel,
```

```
    output DMWr,
    output [3:0] ALUControl,
    output [2:0] DMtype,

    output isBeq,
    output isSlt
);
    // 2 cuts, meaningful code
    wire [5:0] func;
    wire [5:0] opcode;

    // instructions_number
    parameter OP_BEQ = 6'b000100;

    parameter OP_SW  = 6'b101011;
    parameter OP_SH  = 6'b101001;
    parameter OP_SB  = 6'b101000;

    parameter OP_LW  = 6'b100011;
    parameter OP_LH  = 6'b100001;
    parameter OP_LB  = 6'b100000;

    parameter OP_J  = 6'b000010;
    parameter OP_JAL  = 6'b000011;

    parameter OP_ORI  = 6'b001101;
    parameter OP_LUI  = 6'b001111;
    parameter OP_ADDIU  = 6'b001001;

    parameter FUNC_ADDU = 6'b100001;
    parameter FUNC_SUBU = 6'b100011 ;
    parameter FUNC_SLL = 6'b000000;
    parameter FUNC_JR = 6'b001000;
    parameter FUNC_AND = 6'b100100;
    parameter FUNC_SLT = 6'b101010;
    parameter FUNC_JALR= 6'b001001 ;

    assign func = Instr[5:0];
    assign opcode = Instr[31:26];

    // wires instruction
    wire  beq,  sw,sh,sb,  lw,lh,lb,  j,jal,   ori,lui,addiu   ,addu,subu,sll,jr,and_,
slt,jalr; // and : can't use
    //B
```

```
55      assign beq = (opcode == OP_BEQ)? 1'b1 : 1'b0;
56      //S
57      assign sw = (opcode == OP_SW)? 1'b1 : 1'b0;
58      assign sh = (opcode == OP_SH)? 1'b1 : 1'b0;
59      assign sb = (opcode == OP_SB)? 1'b1 : 1'b0;
60      //L
61      assign lw = (opcode == OP_LW)? 1'b1 : 1'b0;
62      assign lh = (opcode == OP_LH)? 1'b1 : 1'b0;
63      assign lb = (opcode == OP_LB)? 1'b1 : 1'b0;
64      //type J
65      assign j = (opcode == OP_J)? 1'b1 : 1'b0;
66      assign jal = (opcode == OP_JAL)? 1'b1 : 1'b0;
67      //type I
68      assign ori = (opcode == OP_ORI)? 1'b1 : 1'b0;
69      assign lui = (opcode == OP_LUI)? 1'b1 : 1'b0;
70      assign addiu = (opcode == OP_ADDIU)? 1'b1 : 1'b0;
71      //type R
72      assign addu = (opcode == 6'd0 && func == FUNC_ADDU)? 1'b1 : 1'b0 ;
73      assign subu = (opcode == 6'd0 && func == FUNC_SUBU)? 1'b1 : 1'b0 ;
74      assign sll = (opcode == 6'd0 && func == FUNC_SLL)? 1'b1 : 1'b0 ;
75      assign jr = (opcode == 6'd0 && func == FUNC_JR)? 1'b1 : 1'b0 ;
76      assign and_ = (opcode == 6'd0 && func == FUNC_AND)? 1'b1 : 1'b0 ;
77      assign slt = (opcode == 6'd0 && func == FUNC_SLT)? 1'b1 : 1'b0 ;
78      assign jalr = (opcode == 6'd0 && func == FUNC_JALR)? 1'b1 : 1'b0 ;
79
80      //
81      assign isBeq = (opcode == OP_BEQ)? 1'b1 : 1'b0;
82      assign isSlt = (opcode == 6'd0 && func == FUNC_SLT)? 1'b1 : 1'b0 ;
83
84      //TYPE
85      wire BRANCH;
86      wire STORE;
87      wire LOAD;
88      wire R;
89      assign BRANCH = (beq == 1'b1)? 1'b1 : 1'b0 ;
90      assign STORE =  ((sw == 1'b1)||(sh == 1'b1)||(sb == 1'b1))? 1'b1 : 1'b0 ;
91      assign LOAD =  ((lw == 1'b1)||(lh == 1'b1)||(lb == 1'b1))? 1'b1 : 1'b0 ;
92      assign R =  (opcode == 6'd0)? 1'b1 : 1'b0 ;
93
94
95      //ALUControl
96      assign ALUControl = (subu == 1'b1)? 4'b0001
97                        :(and_ == 1'b1)? 4'b0010
```

```
98                      :(ori == 1'b1)? 4'b0011
99                      :(lui == 1'b1)? 4'b0100
100                     :(sll == 1'b1)? 4'b0101
101                     : 4'b0000   ;
102
103     //Br   select PC
104     assign Br = (beq == 1'b1)? 3'b001
105               :(j == 1'b1)? 3'b010
106               :(jal == 1'b1)? 3'b011
107               :(jr == 1'b1)? 3'b100
108               :(jalr == 1'b1)? 3'b101
109               : 3'b000  ;
110
111     //A3Sel     select of A3 of GRF
112     assign A3Sel = (R == 1'b1)? 3'b001
113                  :(jal == 1'b1)? 3'b010
114                  : 3'b000  ;
115
116     //WDSel     select of WriteData of GRF
117     assign WDSel = (LOAD == 1'b1)? 3'b001
118                  :(jal == 1'b1 || jalr == 1'b1)? 3'b010
119                  :(slt == 1'b1)? 3'b011
120                  : 3'b000  ;
121
122     //RFWr       GRF write enable(not + or)
123     assign RFWr = ((STORE == 1'b1)||(BRANCH == 1'b1)||(jr == 1'b1)||(j == 1'b1))?
    1'b0 : 1'b1 ;
124
125    //EXTop     signed ~ yes
126    assign EXTop = ((STORE == 1'b1)||(LOAD == 1'b1)||(addiu == 1'b1))? 1'b1 : 1'b0 ;
127
128    //BSel     select ScrB of ALU / not use imm as B(not + or)
129    assign BSel = ((R == 1'b1)||(BRANCH == 1'b1)||(j == 1'b1)||(jal == 1'b1))? 1'b0 :
    1'b1 ;
130
131    //DMWr   DM enabled
132    assign  DMWr = STORE;
133
134    //DMtype
135    assign  DMtype =(lh == 1'b1 || sh == 1'b1)? 3'b001
136                   :(lb == 1'b1 || sb == 1'b1)? 3'b010
137                   : 3'b000  ;
138
139 endmodule
```

②用 assign 语句，进行位运算，不判断指令类型而直接完成操作码和控制信号的值之间的对应，是基于与或门阵列的设计。

```
module new_controller2(
input [5:0] op,
input [5:0] func,
 output [2:0] ALUCtrl,
 output [1:0] RegDst,
 output ALUSrc,
 output RegWrite,
 output MemRead,
 output MemWrite,
 output [1:0] MemtoReg,
 output ExtOp,
 output Branch1,
 output Branch2,
output Branch3
 );
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 RegDst[1] = jal;
assign RegDst[0] = r;
assign ALUSrc = lw||sw||ori;
assign RegWrite = r||lui||ori||lw||jal;
assign MemRead = lw;
assign MemWrite = sw;
assign MemtoReg[1] = lw||jal;
assign MemtoReg[0] = lui||jal;
assign ExtOp = ori;
```

```
assign Branch1 = beq;
assign Branch2 = jal;
assign Branch3 = jr;
assign ALUCtrl[2] = jal||lui;
assign ALUCtrl[1] = lw||sw||beq||lui||addu||subu||jr||jal;
assign ALUCtrl[0] = beq||lui||ori||subu||jal;
    endmodule
```

① **优点**：存在指令中间量，简单易懂；增添指令的时候可移植性强。

缺点：这种实现方式相比与或门阵列，非常浪费硬件资源；设计逻辑符合人的认知，但是不符合硬件的设计逻辑。

② **优点**：节约硬件资源；减少了不必要的指令判断，复合硬件实现逻辑；

缺点：难以读懂

3. 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 所驱动的部件具有什么共同特点？

答：PC,DM,GRF 需要 reset 复位。

具有记忆和存储功能；

不清空就可能影响下一次程序的执行。

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 。

答：add,addi 与 addu,addiu 唯一的区别就是，将加数扩展到 33 位，并且比较加法结果的 32 位和 31 位。

当溢出，temp32=A32+B32+overflow= A31+B31+overflow，temp31=A31+B31。故加法溢出当且仅当 temp32!=temp31。忽略溢出，add 与 addi 指令便不在 temp32!=temp31 条件下记录溢出、而只是记录 temp[31:0]，此时他们与 addu,addiu 行为一致。

5. 根据自己的设计说明单周期处理器的优缺点。

答：优点：程序依照“一周期一指令”顺序执行，安全性好，不会出现数据逻辑错误（比如取用了 GRF 和 DM 中还未被写入新值覆盖的旧值）

缺点：单周期 CPU 的最高频率受限于执行时间最长的指令，降低了 CPU 的效率。