计算机组成原理实验报告

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MIPS - CPU，支持的指令集包含{addu、subu、ori、lw、sw、beq、lui、sll、srl、sra、slt、j、jal、jr、jalr、nop}。为了实现这些功能，CPU 主要包含了 PC、NPC、IM、GRF、ALU、DM、EXT、ShamtEXT……，这些模块按照自顶向下的顶层设计逐级展开。

（二）关键模块定义

1. PC

当前 PC 值的存储由一个 32 位 reg 变量保存。PC 模块端口定义如下：

信号名	方向	位宽	描述
clk	I	1	时钟信号
reset	I	1	同步复位信号
NPC	I	32	由 NPC 模块计算的下一个 PC 值
PC	O	32	当前的 PC

端口 Verilog 声明如下：

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

2. NPC

NPC 模块端口定义如下：

信号名	方向	位宽	描述
_26IMM	I	26	指令 instr 的低 26 位
ALUZero	I	1	由 ALU 传入的判零信号
PC	I	32	当前的 PC
NPCOp	I	3	下一个 PC 的计算选择信号
GPR_rs	I	32	从寄存器 RD1 端口传入的信号，用于跳转寄存器相关的指令
NPC	O	32	计算出的下一个 PC 值
PC_PLUS_4	O	32	当前 PC + 4 的值

NPCOp 控制信号真值表如下：

NPCOp[2:0]	功能	NPCOp[2:0]	功能
000	NPC = PC + 4	100	\
001	BEQ 指令 NPC 计算	101	\
010	跳转到 26 位立即数	110	\
011	跳转到 GPR_rs 寄存器	111	\

端口 Verilog 声明如下：

```
module NPC(
    input [25:0] _26IMM,
    input ALUZero,
    input [31:0] PC,
    input [2:0] NPCOp,
    input [31:0] GPR_rs,
    output reg [31:0] NPC,
    output [31:0] PC_PLUS_4
    );
```

3. IM

IM 端口声明如下：

信号名	方向	位宽	描述
PC	I	32	当前 PC 值
Opcode	O	6	指令高 6 位
Funct	O	6	指令低 6 位
rs	O	1	写数据使能信号
rt	O	1	同步复位信号
rd	O	32	DM 中 A 地址读出的数据
Shamt	O	5	指令 shamt 片段
_16IMM	O	16	指令低 16 位
_26IMM	O	26	指令低 26 位

端口 Verilog 声明如下：

```
module IM(
    input [31:0] PC,
    output [5:0] opcode,
    output [5:0] funct,
    output [4:0] rs,
    output [4:0] rt,
    output [4:0] rd,
    output [4:0] shamt,
    output [15:0] _16IMM,
    output [25:0] _26IMM
    );
```

IM 内部 32 bit * 1024 字的存储器的具体实现如下：

```
reg [31:0] im[0:1023];
```

4. EXT

EXT 模块端口定义如下：

信号名	方向	位宽	描述
ZeroEXT	I	1	扩展信号选择

_16IMM	I	16	待扩展的 16 位立即数
EXTResult	O	32	扩展后的 32 位信号

端口 Verilog 声明如下：

```
module EXT(
    input ZeroEXT,
    input [15:0] _16IMM,
    output reg [31:0] EXTResult
    );
```

5. ShamtEXT

ShamtEXT 模块端口定义如下：

信号名	方向	位宽	描述
Shamt	I	5	指令 shamt 字段
Shamt32	O	32	Shamt 零扩展成 32 位数

端口 Verilog 声明如下：

```
module ShamtEXT(
    input [4:0] shamt,
    output [31:0] shamt32
    );
```

6. GRF

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。具体模块端口定义如下：

信号名	方向	位宽	描述
clk	I	1	时钟信号
reset	I	1	同步复位信号

WE3	I	1	写使能信号
A1	I	5	读地址
A2	I	5	读地址
A3	I	5	写地址
WD	I	32	32 位写入数据
RD1	O	32	32 位读出数据
RD2	O	32	32 位读出数据

端口 Verilog 声明如下：

```
module GRF(
    input clk,
    input reset,
    input WE3,
    input [31:0] PC,
    input [4:0] A1,
    input [4:0] A2,
    input [4:0] A3,
    input [31:0] WD,
    output [31:0] RD1,
    output [31:0] RD2
    );
```

GRF 内部 32 bit * 32 的寄存器的具体实现如下：

```
reg [31:0] register[0:31];
```

7. DM

DM 端口声明如下：

信号名	方向	位宽	描述
clk	I	1	时钟信号
A	I	1	读地址
WD	I	32	待写入 DM 的 32 位数据信号

WE	I	1	写数据使能信号
reset	I	1	同步复位信号
RD	O	32	DM 中 A 地址读出的数据

端口 Verilog 声明如下：

```
module DM(
    input clk,
    input reset,
    input [31:0] PC,
    input [31:0] A,
    input [31:0] WD,
    input WE,
    output [31:0] RD
    );
```

DM 内部 32 bit * 1024 字的存储器的具体实现如下：

```
reg [31:0] dm[0:1023];
```

8. ALU

ALU 支持 AND、OR、ADD、SUB、SLL、SRL、SRA、SLT（有符号）等运算，其中 ADD、SUB 运算不支持溢出检测。

ALU 端口声明如下：

信号名	方向	位宽	描述
ALUSrcA	I	32	ALU 操作数 1
ALUSrcB	I	32	ALU 操作数 2
ALUOp	I	4	ALU 功能选择信号
ALUZero	O	1	ALU 判零信号
ALUResult	O	32	ALU 运算结果

端口 Verilog 声明如下：

```
module ALU(
    input [31:0] ALUSrcA,
    input [31:0] ALUSrcB,
    output reg [31:0] ALUResult,
    output ALUZero,
    input [3:0] ALUOp
    );
```

ALU 控制信号真值表如下：

ALUOp[3:0]	功能	ALUOp[3:0]	功能
0000	A AND B	1000	\
0001	A OR B	1001	\
0010	A + B	1010	\
0011	A - B	1011	\
0100	A 逻辑左移 B	1100	\
0101	A 逻辑右移 B	1101	\
0110	A 算数右移 B	1110	\
0111	SLT	1111	\

9. Controller

控制器 Controller 端口声明如下：

控制器			
端口声明	方向	位宽	描述
Opcode	I	6	指令 Instr[31:26]
Funct	I	6	指令 Instrc[5:0]
REGorMEM	O	1	grf 回写控制信号之一： 0：ALUResult 回写； 1：DM 读出的值回写
MemWrite	O	1	DM 写使能信号： 0：不能写入；

			1：可以写入
NPCOp	O	3	NPC 计算方式选择信号： 000：PC = PC + 4; 001：PC = PC + 4 + ALUZero && Sign_ext(16imm || 00) (beq) 010：PC = PC[31:28] || 26imm || 00 (j、jal) 011：PC = GPR[rs] (jr $ra，注：$ra 不一定得是$31) 100：暂未定义 101：暂未定义 110：暂未定义 111：暂未定义
ZeroEXT	O	1	EXT 对 16 位立即数的扩展方式： 0：符号扩展 1：零扩展
A1op	O	2	GRF 中 A1 端口选择信号： 00：读取 rs（Instr[25:21]） 01：读取 rt（Instr[20:16]） 10：暂未定义 11：暂未定义
A2op	O	1	GRF 中 A2 端口选择信号： 0：读取 rt（Instr[20:16]） 1：暂未定义
A3op	O	2	GRF 中 A3 端口选择信号： 00：读取 rd（Instr[15:11]） 01：读取 rt（Instr[20:16]） 10：读取常数 31（即$r31 址） 11：暂未定义
REGop	O	2	GRF 写入数据端口选择信号： 00：回写 REGorMEM 选择后的信号

			01：回写 16 位立即数加载至高位后的 32 位数据 10：回写当前 PC 加 4 的值 11：暂未定义
RegWrite	O	1	GRF 写使能信号： 0：不可写 1：可写
ALU_Aop	O	1	ALU SrcA 端口选择信号： 0：读取 RD1 1：暂未定义
ALU_Bop	O	2	ALU SrcB 端口选择信号： 00：读取 RD2 01：读取 16 位立即数（根据 ALUZero 选择）扩展后的 32 位数据 10：读取指令 shamt 片段 0 扩展后的结果 11：暂未定义
ALUOp	O	4	ALU 功能选择信号： 0000：and 0001：or 0010：add（不考虑溢出） 0011：sub（不考虑溢出） 0100：sll 0101：srl 0110：sra 0111：slt（注：有符号!） 1000—1111：暂未定义

端口 verilog 声明如下：

```
module CONTROLLER(
    input [5:0] opcode,
    input [5:0] funct,
    output reg REGorMEM,
    output reg MemWrite,
    output reg [2:0] NPCOp,
    output reg ZeroEXT,
    output reg [1:0] A1op,
    output reg A2op,
    output reg [1:0] A3op,
    output reg [1:0] REGop,
    output reg RegWrite,
    output reg ALU_Aop,
    output reg [1:0] ALU_Bop,
    output reg [3:0] ALUOp
    );
```

控制器真值表如下：（本电路实现中 X 全置 0）

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	指令	opcode	funct	REGorMEM	MemWrite	NPCop	ZeroEXT	A1op	A2op	A3op	REGop	RegWrite	ALU_Aop	ALU_Bop	ALUOp
2	addu	000000	100001	0	0	000	0	00	0	00	00	1	0	00	0010
3	subu	000000	100011	0	0	000	0	00	0	00	00	1	0	00	0011
4	ori	001101		0	0	000	1	00	0	01	00	1	0	01	0001
5	lw	100011		1	0	000	0	00	0	01	00	1	0	01	0010
6	sw	101011		0	1	000	0	00	0	00	00	0	0	01	0010
7	beq	000100		0	0	001	0	00	0	00	00	0	0	00	0000
8	lui	001111		0	0	000	0	00	0	01	01	1	0	00	0000
9	sll	000000	000000	0	0	000	0	01	0	00	00	1	0	10	0100
10	srl	000000	000010	0	0	000	0	01	0	00	00	1	0	10	0101
11	sra	000000	000011	0	0	000	0	01	0	00	00	1	0	10	0110
12	slt	000000	101010	0	0	000	0	00	0	00	00	1	0	00	0111
13	j	000010		0	0	010	0	00	0	00	00	0	0	00	0000
14	jal	000011		0	0	010	0	00	0	10	10	1	0	00	0000
15	jalr	000000	001001	0	0	011	0	00	0	00	10	1	0	00	0000
16	jr	000000	001000	0	0	011	0	00	0	00	00	0	0	00	0000
17	and、andi、or、addi不加入，避免opcode、funct冲突太多！！！！														

（三）重要机制实现方法

1. 跳转

该 CPU 涉及的跳转指令有：beq、j、jal、jr、jalr。

BEQ：相等时转移

编码	31 26	25 21	20 16	15 0
	beq 000100	rs	rt	offset
	6	5	5	16
格式	beq rs, rt, offset			
描述	if (GPR[rs] == GPR[rt]) then 转移			

操作	if (GPR[rs] == GPR[rt])
	PC ← PC + 4 + sign_extend(offset‖0^2)
	else
	PC ← PC + 4			

示例	beq $s1, $s2, -2			
其他				

对于 beq 指令，控制器解析 beq 指令，输出的 NPCOp = 001，同时 ALU 判断两个操作数 GPR[rs]、GPR[rt]是否相等，若相等将 ALUZero 置一。在 NPC 内，当 ALUZero = 1 并且 NPCOp = 001 时，才执行跳转，即 PC = PC + 4 + ALUZero && Sign_ext(16imm || 00)，若不满足该条件，则只执行 PC = PC + 4。

J：跳转

编码	31 26	25 0
	j 000010	instr_index
	6	26
格式	j target	
描述	j 指令是 PC 相关的转移指令。当把 4GB 划分为 16 个 256MB 区域，j 指令可以在当前 PC 所在的 256MB 区域内任意跳转。	
操作	PC ← PC31..28‖instr_index‖0^2	
示例	j Loop_End	
其他	如果需要跳转范围超出了当前 PC 所在的 256MB 区域内时，可以使用 JR 指令。	

对于 j 指令，控制器解析 j 指令，将 NPCOp 置为 010，同时 NPC 读入 26 位立即数，PC = {PC[31:28]，26imm，2{0}}（verilog 位拼接运算符），从而实现跳转。

JAL：跳转并链接

编码	31 26 25 0	
	jal 000011	instr_index
	6	26
格式	`jal target`	
描述	jal 指令是函数指令，PC 转向被调用函数，同时将当前 PC+4 保存在 GPR[31]中。当把 4GB 划分为 16 个 256MB 区域，jal 指令可以在当前 PC 所在的 256MB 区域内任意跳转。	
操作	`PC ← PC31..28‖instr_index‖0`^{2} `GPR[31] ← PC + 4`	
示例	`jal my_function_name`	
其他	jal 与 jr 配套使用。jal 用于调用函数，jr 用于函数返回。当所调用的函数地址超出了当前 PC 所在的 256MB 区域内时，可以使用 jalr 指令。	

对于 jal 指令，其 NPC 相关控制信号与上述 j 指令相同。不同的是，对于 jal 指令，还需要通过 REGop 信号选择 10，将 PC + 4 的值存入$31 寄存器，为此，在 grf 写数据端口 WD3 放置了一个 4 选 1 多路选择器，其第三个选择端口为当前 PC 加 4 的值。

JR：跳转至寄存器

编码	31 26	25 21	20 11	10 6	5 0
	special 000000	rs	0 00 0000 0000	0 00000	jr 001000
	6	5	10	5	6
格式	`jr rs`				
描述	`PC ← GPR[rs]`				
操作	`PC ← GPR[rs]`				
示例	`jr $31`				
其他	jr 与 jal/jalr 配套使用。jal/jalr 用于调用函数，jr 用于函数返回。				

对于 jr 指令，其将 GPR[rs]寄存器存的 32 位值存入 PC，在数据通路上只需要在 NPC 的输入端口增加一个读入寄存器（32 位数据）的端口，同时设置 NPCOp = 011 时，将 pc 设置为输入的 32 位数据的值。

JALR：跳转并链接

编码	31 26	25 21	20 16	15 11	10 6	5 0
	special 000000	rs	0 00000	rd	0 00000	jalr 001001
	6	5	5	5	5	6
格式	`jalr rd, rs`					
描述	jalr 指令是函数指令，PC 转向被调用函数(函数入口地址保存在 GPR[rs]中)，同时将当前 PC+4 保存在 GPR[rd]中。					
操作	`PC ← GPR[rs]` `GPR[rd] ← PC + 4`					
示例	`jalr $s1, $31`					

对于 jalr 指令，其涉及的 PC 操作与 jr 相同，都是将 GPR[rs]存入 PC。同时，它与 jal 指令相似，都需要将当前 pc 加 4 的值存入寄存器。不同的是，jal 将 pc + 4 存入$31 寄存器，故 A3op 选择 10；而 jalr 将 pc + 4 存入$rd 寄存器，故 A3op 选择 00。

二、测试方案

（一）典型测试样例

下面为对 cpu 中涉及的所有指令进行的一次相对全覆盖的测试：

（注意：在下面第 41 到 43 行对跳转指令 jalr 的测试中，我将$30 寄存器赋值为 0x3004 作为跳转的地址而非 0x0004，否则会导致其在 logisim 中能正常执行跳转并在第三行通过 jr $28 返回，而在 Mars 中会出现溢出错误。）

```
    addu $30, $30, $0
    beq $0, $30, start
    jr $28
start:
    lui $0, 0xffff
    ori $0, $0, 0xffff

    lui $1, 0xffff
    lui $2, 0xfff
    lui $3, 0xff
    ori $1, $1, 0xffff  # $1 = -1
    ori $2, $2, 0xfff
    ori $3, $3, 0xff
    ori $4, $0, 1
    ori $5, $0, 2
    ori $28, $0, 0x10

    nop  # see PC

    addu $6, $1, $4  # $6 = 0
    addu $7, $1, $5  # $7 = 1
    sll $7, $7, 2  # $7 = 4

    subu $8, $4, $1  # $8 = 2
    subu $9, $5, $4  # $9 = 1

    beq $6, $7, start

loop:  # $6 = 0
    beq $6, $28, end_loop
    sw $1, 0($6)
    sw $2, 4($6)
    addu $6, $6, $7  # $6 += 4
    addu $1, $1, $6  # $1 += 4
    addu $2, $2, $6  # $2 += 4
    j loop
end_loop:
    jal load_words
```

```
39
40      ori $30, $0, 0
41  test_jalr:
42      ori $30, $0, 0x3004
43      jalr $28, $30  # $28 = pc + 4; pc = $30
44
45      beq $0, $0, final_test
46
47
48  load_words:
49      lw $1, 0($0)
50      lw $2, 4($0)
51      lw $3, 8($0)
52      lw $4, 12($0)
53      lw $5, 16($0)
54      lw $6, 20($0)
55      lw $7, 24($0)
56      lw $8, 28($0)
57      jr $ra
58
59
60  final_test:
61      lui $1, 0xffff
62      ori $1, 0xf
63      lui $2, 0xffff
64      ori $2, $2, 213
65      lui $3, 0x6
66      ori $3, $3, 0xff32
67      lui $28, 0x7
68
69      sll $4, $1, 8
70      srl $5, $4, 4
71      sra $6, $4, 4
72      slt $7, $1, $3
73      slt $8, $3, $28
74      lui $28, 0x5
75      slt $7, $3, $28
76
```

三、思考题

（一）题目描述：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

由于 MIPS 中数据存储器按照字节寻址，一个字占 4 字节；而 verilog 中 dm 按字寻址（reg [31:0] dm[0:1023]），dm 以字为单位存放数据，所以输入进来的 addr 地址的数值大小是 dm 所在地址数值的 4 倍，故需要对输入的 addr 除以 4，即从 addr 第三位开始取值。同时实验要求地址大小为 1024，2 的 10 次方，即地址二进制宽度为 10，故取 addr[11:2]。

（二）题目描述：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）根据 opcode 和 funct 解读指令，从而根据指令确定控制信号的值（本 CPU 采用的方法）。代码示例：

```
// controller
`define R_OP          6'b000000
`define ORI_OP        6'b001101
`define LW_OP         6'b100011
`define SW_OP         6'b101011
`define BEQ_OP        6'b000100
`define LUI_OP        6'b001111
`define J_OP          6'b000010
`define JAL_OP        6'b000011

`define ADDU_FUNCT    6'b100001
`define SUBU_FUNCT    6'b100011
`define SLL_FUNCT     6'b000000
`define SRL_FUNCT     6'b000010
`define SRA_FUNCT     6'b000011
`define SLT_FUNCT     6'b101010
`define JALR_FUNCT    6'b001001
`define JR_FUNCT      6'b001000
```

```
always@(*) begin
    case(opcode)
        `R_OP: begin
            case(funct)
                `ADDU_FUNCT: begin
                    REGorMEM = 1'b0;
                    MemWrite = 1'b0;
                    NPCOp = 3'b000;
                    ZeroEXT = 1'b0;
                    A1op = 2'b00;
                    A2op = 1'b0;
                    A3op = 2'b00;
                    REGop = 2'b00;
                    RegWrite = 1'b1;
                    ALU_Aop = 1'b0;
                    ALU_Bop = 2'b00;
                    ALUOp = `ADD;
                end
```

优点：直观，debug 方便；

缺点：代码量相对较大；

（2）通过设置一个 reg 类型临时变量 temp，将所有控制信号按照位拼接合在一起，直接通过连续赋值语句 assign 将其赋值给位拼接后的输出控制信号。代码示例如下：**（非本 cpu 的方法，代码直接用别人的，这不算抄袭吧）**

```
always @(cmd) begin
    if(cmd==0) temp=0;//nop
    else case(cmd[31:26])
        0:case(cmd[5:0])
            0:temp=16'b00_1_01_10_00_00_0_1010;//sll
            2:temp=16'b00_1_01_10_00_00_0_1000;//srl
            3:temp=16'b00_1_01_10_00_00_0_1001;//sra
            4:temp=16'b00_1_01_00_00_00_0_1010;//sllv
            6:temp=16'b00_1_01_00_00_00_0_1000;//srlv
            7:temp=16'b00_1_01_00_00_00_0_1001;//srav
            8:temp=16'b00_0_00_00_00_00_1_0000;//jr
            9:temp=16'b00_1_01_00_00_10_1_0000;//jalr
```

```
assign {ExtOp,RegWrite,RegDst,ALUSrc,Branch,MemWrite,RegSrc,Jump,ALUCtrl}=temp;
```

优点：写起来方便，代码量小

缺点：temp 信号本身不直观，不易维护

（3）与或门阵列对控制信号的每一位进行赋值**（非本 cpu 的方法，代码直接用别人的，这不算抄袭吧）**

```
wire addu,subu,ori,lw,sw,beq,lui,jal,jr,nop;
assign addu = (opcode==6'b000000&&func==6'b100001) ? 1: 0;
assign subu = (opcode==6'b000000&&func==6'b100011) ? 1 : 0;
assign ori = (opcode==6'b001101) ? 1 : 0;
assign lw = (opcode==6'b100011) ? 1 : 0;
assign sw = (opcode==6'b101011) ? 1 : 0;
assign beq = (opcode==6'b000100) ? 1 : 0;
assign lui = (opcode==6'b001111) ? 1 : 0;
assign jal = (opcode==6'b000011) ? 1 : 0;
assign jr = (opcode==6'b000000&&func==6'b001000) ? 1 : 0;
```

```
assign NPCOp[0] = beq;
assign NPCOp[1] = jal||jr;
assign GRFWE = addu||subu||ori||lw||lui||
assign ALUOp[0] = addu||ori||lw||sw;
assign ALUOp[1] = subu||ori||beq;
assign ALUOp[2] = 0;
assign EXTOp[0] = ori||beq;
assign EXTOp[1] = lw||sw||beq;
assign EXTOp[2] = lui;
assign DMWE = sw;
assign GRFA3_MUXOp[0] = addu||subu;
assign GRFA3_MUXOp[1] = jal;
assign GRFWD_MUXOp[0] = lw||jal;
assign GRFWD_MUXOp[1] = lui||jal;
assign ALUB_MUXOp = addu||subu||beq;
assign NPCIMM_MUXOp[0] = jal;
assign NPCIMM_MUXOp[1] = jr;
```

优点：综合成电路的成本最低（即电路元器件用的最少）

缺点：debug 时不方便，不易找到控制器发生的错误。

（三）题目描述：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？

Reset 驱动的都是能存储值（有记忆功能）的部件。即 PC 中的寄存器，GRF 中的寄存器，DM 中的寄存器。

（四）题目描述：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 。

从 ADD 和 ADDU 的 RTL 可以发现，ADD 仅仅只是多了一个在发生溢出时抛出异常的操作，在忽略溢出的前提下 ADD 与 ADDU 自然等价。ADDI 与 ADDIU 同理。

ADD：符号加

编码	31 26	25 21	20 16	15 11	10 6	5 0
	special 000000	rs	rt	rd	0 00000	add 100000
	6	5	5	5	5	6
格式	add rd, rs, rt					
描述	GPR[rd] ← GPR[rs]+GPR[rt]					

操作	temp ← (GPR[rs]$_{31}$‖GPR[rs]) + (GPR[rt]$_{31}$‖GPR[rt])
	if temp$_{32}$ ≠ temp$_{31}$ then
	SignalException(IntegerOverflow)
	else
	GPR[rd] ← temp$_{31..0}$
	endif					

示例	add $s1, $s2, $s3					
其他	temp$_{32}$≠ temp$_{31}$ 代表计算结果溢出。 如果不考虑溢出，则 add 与 addu 等价。					

（五）题目描述：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，模块内高内聚，模块间低耦合，CPI = 1，执行一条指令仅需一周期

缺点：由于一个周期只执行一条指令，故需根据时间延迟最长的那条指令来确定时钟信号变化，即时钟频率低，性能低。