CS202 Computer Organization Final Project

Part1-项目需求及开发者信息

1.1）项目需求

本次项目我们实现了一个CPU，支持Minisys的基本32条指令，可以解析传入的coe文件，并通过Minisys开发板与用户完成交互，接受用户输入并将结果显示在开发板上。

1.2）开发者信息

	李文锦 12012514	**杨鹭鸣 12011412**	**王宇航 12012208**
负责任务	负责编写包括cpu顶层以及各个子模块的verilog文件，撰写部分实验报告	负责项目测试工作，编写testbench，对项目进行上板测试与debug，撰写部分实验报告	编写asm文件，搭建两个测试场景,撰写部分实验报告
贡献占比	33.3%	33.3%	33.3%

Part2-版本更迭记录

本次项目我们采用github来控制我们的版本， Vancy0/CS202-CPU-DESIGN at CPU_UPDATE (github.com)，本仓库有三个分支，main存放的是cpu的子模块文件，mips模块存放的是汇编文件，CPU_UPDATE模块存放迭代版本的信息

Part3-CPU架构设计说明

3.1）CPU特性

- **所支持的ISA**：参考了Minisys的指令集（共31条），提供了32个32位的寄存器，不支持异常处理。
- **寻址空间设计：**在此CPU中采用了哈佛结构，即指令储存与数据储存分开的方式。指令空间起始地址为0x00400000, 数据空间起始地址为0x10010000, 它们的读写宽度都为32bits, 读写深度均为16384bits，外设IO的寻址范围为 0xFFFFFC00 至 0xFFFFFFFF，寻址单位均为一个字即4个字节。
- **CPI及其他CPU特性:** 此CPU为单周期CPU，其CPI接近为1。

R-type	op	rs	rt	rd	shamt	func	使用方式	使用结果	对应解释
add	000000	rs	rt	rd	00000	100000	add 1,2,3|1=2+3	rd <- rs + rt ； 其中rs = 2，rt =3, rd=$1	
addu	000000	rs	rt	rd	00000	100001	addu 1,2,3|1=2+3	rd <- rs + rt ； 其中rs = 2，rt =3, rd=$1,无符号数	
sub	000000	rs	rt	rd	00000	100010	sub 1,2,3|1=2−3	rd <- rs - rt ； 其中rs = 2，rt =3, rd=$1	
subu	000000	rs	rt	rd	00000	100011	subu 1,2,3|1=2−3	rd <- rs - rt ； 其中rs = 2，rt =3, rd=$1,无符号数	
and	000000	rs	rt	rd	00000	100100	and 1,2,3|1=2&3	rd <- rs & rt ； 其中rs = 2，rt =3, rd=$1	
or	000000	rs	rt	rd	00000	100101	or 1,2,3|1=2||3	rd <- rs | rt ； 其中rs = 2，rt =3, rd=$1	
xor	000000	rs	rt	rd	00000	100110	xor 1,2,3|1=2^3	rd <- rs xor rt ； 其中rs = 2，rt =3, rd=$1(异或)	
nor	000000	rs	rt	rd	00000	100111	nor 1,2,3| 1=~(2||3)	rd <- not(rs | rt) ； 其中rs = 2，rt =3, rd=$1(或非)	
slt	000000	rs	rt	rd	00000	101010	slt 1,2,3|if(2<3)1=1 else $1 = 0\|if(rs < rt)rd = 1elserd = 0$；其中$rs$= 2，rt=3, rd =1		
sltu	000000	rs	rt	rd	00000	101011	sltu 1,2,3|if(2<3)1=1 else $1 = 0\|if(rs < rt)rd = 1elserd = 0$；其中$rs$= 2，rt=3, rd =1 (无符号数)		
sll	000000	00000	rt	rd	shamt	000000	sll 1,2,10	1 =2<<10	rd <- rt << shamt ； shamt存放移位的位数，也就是指令中的立即数，其中rt=2, rd =1
srl	000000	00000	rt	rd	shamt	000010	srl 1,2,10	1 =2>>10	rd <- rt >> shamt ； (logical) ，其中rt= 2, rd =1
sra	000000	00000	rt	rd	shamt	000011	sra 1,2,10	1 =2>>10	rd <- rt >> shamt ； (arithmetic) 注意符号位保留 其中rt=2, rd =1
sllv	000000	rs	rt	rd	00000	000100	sllv 1,2,3|1=2 <<3	rd <- rt << rs ； 其中rs = 3，rt =2, rd=$1	

R-type	op	rs	rt	rd	shamt	func	使用方式	使用结果	对应解释
srlv	000000	rs	rt	rd	00000	000110	srlv 1,2,3|1=2 >>3	rd <- rt >> rs；(logical)其中rs = 3，rt =2, rd=$1	
srav	000000	rs	rt	rd	00000	000111	srav 1,2,3|1=2 >>3	rd <- rt >> rs；(arithmetic) 注意符号位保留 其中rs = 3，rt =2, rd=$1	
jr	000000	rs	00000	00000	00000	001000	jr 31|$goto$31	PC <- rs	

I-type	op	rs	rt	immediate	使用方式	使用结果	对应解释
addi	001000	rs	rt	immediate	addi 1,2,100	1 =2+100	rt <- rs + (sign-extend)immediate；其中rt= 1, rs =2
addiu	001001	rs	rt	immediate	addiu 1,2,100	1 =2+100	rt <- rs + (zero-extend)immediate；其中rt= 1, rs =2
andi	001100	rs	rt	immediate	andi 1,2,10	1 =2 & 10	rt <- rs & (zero-extend)immediate；其中rt= 1, rs =2
ori	001101	rs	rt	immediate	ori 1,2,10	1 =2 | 10	rt <- rs | (zero-extend)immediate；其中rt= 1, rs =2
xori	001110	rs	rt	immediate	xori 1,2,10	1 =2 ^ 10	rt <- rs xor (zero-extend)immediate；其中 rt=1, rs =2
lui	001111	00000	rt	immediate	lui 1, 100| 1=100*65536	rt <- immediate*65536；将16位立即数放到目标寄存器高16位，目标寄存器的低16位填0	
lw	100011	rs	rt	immediate	lw 1, 10(2)	1 = $memory$[2 +10]	rt <- memory[rs + (sign-extend)immediate]；rt=1, rs =2
sw	101011	rs	rt	immediate	sw 1, 10(2)	memory[2 + 10] =1	memory[rs + (sign-extend)immediate] <- rt；rt=1, rs =2
beq	000100	rs	rt	immediate	beq 1,2,10	if(1 ==2) goto PC+4+40	if (rs == rt) PC <- PC+4 + (sign-extend)immediate<<2
bne	000101	rs	rt	immediate	bne 1,2,10	if(1! =2) goto PC+4+40	if (rs != rt) PC <- PC+4 + (sign-extend)immediate<<2
slti	001010	rs	rt	immediate	slti 1,2,10	if(2 < 10)1=1 else 1 = 0|$if(rs < (sign-extend)immediate)rt = 1 else rt = 0$；其中$rs$=2, rt=$1	
sltiu	001011	rs	rt	immediate	sltiu 1,2,10	if(2 < 10)1=1 else 1 = 0|$if(rs < (zero-extend)immediate)rt = 1 else rt = 0$；其中$rs$=2, rt=$1	

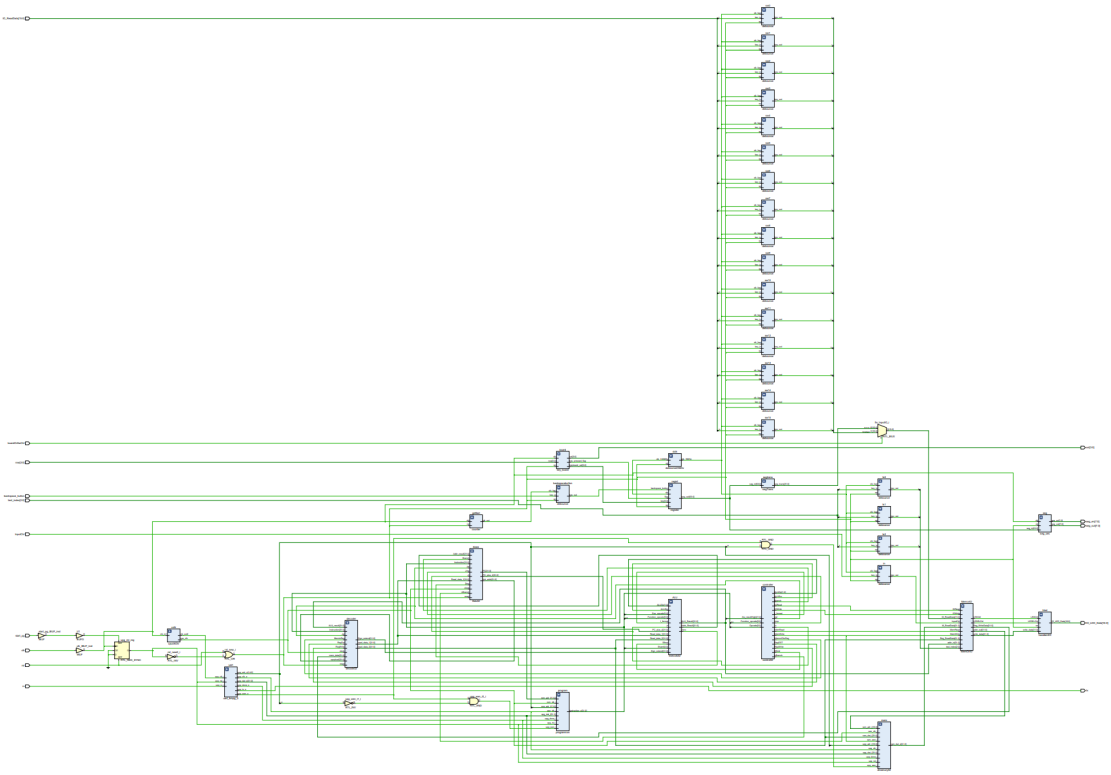
J-type	op	address	使用方式	使用结果	对应解释
j	000010	address	j 10000	goto 10000	PC <- (PC+4)[31..28],address,0,0address=10000/4
jal	000011	address	jal 10000	$31 <- PC+4; goto 10000$|31<-PC+4；PC <-(PC+4)[31..28],address,0,0；address=10000/4	

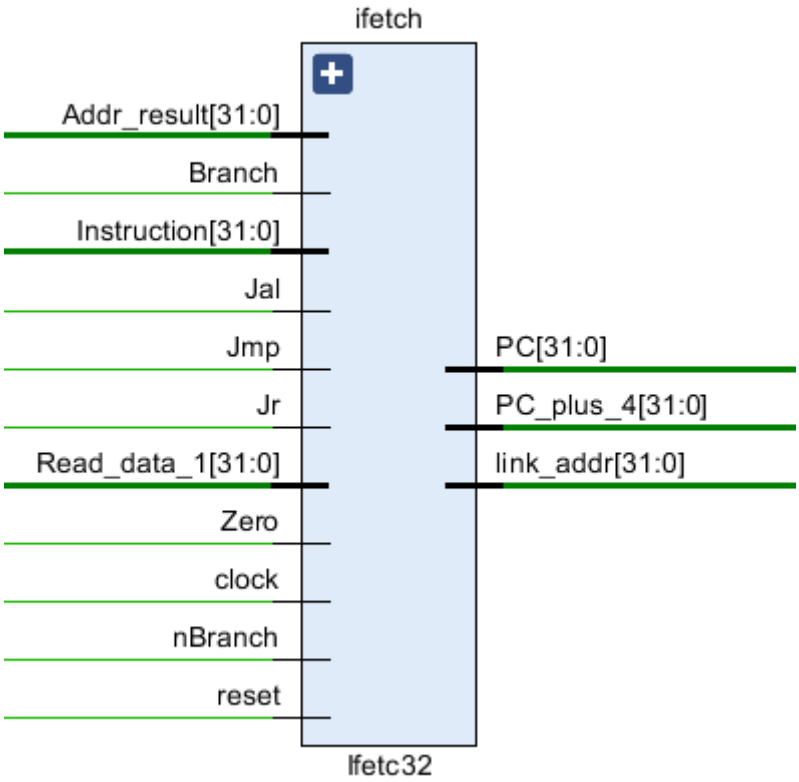
3.2）CPU接口

- **时钟**：在本CPU中使用到了开发板提供的100MHz时钟（Y18），通过ip核分别转化成为了23MHz（供单周期CPU使用）以及10MHz（供uart接口使用），占空比为50%的周期时钟信号。
- **复位**：在本CPU中，我们通过高电位信号来判断复位信号并通过开发板的按钮外设来控制复位信号的输入。在按下复位按钮（R1）之后，CPU将会重新回到初始状态，之前的所有状态将会被清空。
- **uart接口**：我们实现并提供了uart接口，该接口可以支持将生成好的coe文件发送给CPU。使用该接口需要按下CPU中设定好的用来控制通信状态的按钮（P2）来使CPU进入通信状态（此时，七段数码显示管第一位全亮来提示已经进入通信状态），在此状态下之后，使用串口调试助手将指定文件发送给CPU，当显示发送完成的字样后，再次按下控制按钮（P2）来使CPU回到普通工作模式即可。在此还用到了Y19以及V18端口来分别作为数据的接受和发送端。
- **数据输入&输出接口**
  - **控制输入方式接口**：由于我们同时实现了键盘输入以及开关输入的方式，因此另实现一个控制接口来控制输入的方式。该接口绑定在开关AA8上，当置为高电平时为键盘输入，为低电平时为开关输入。
  - **确认输入完成接口**：为了可以让CPU执行我们所希望输入的数据，在按下该按钮前，CPU不会进行下一步的读入操作。该接口绑定在按钮P5上，当确认输入完成之后，按下该按钮，CPU就会读入已经准备好的数据。
  - **小键盘接口**：实现了小键盘的输入，可以通过小键盘来输入对应数据（十六进制数）。其中，输入上限为8个十六进制数，到达上限之后不会再允许输入。(**具体的端口绑定请看project_xdc.xdc文件**)
    - **七段数码显示管**：在实现小键盘的基础上，为了输入的可视性，我们另实现了七段数码显示管接口，可以实时展示此时小键盘的输入数据。(**具体的端口绑定请看project_xdc.xdc文件**)
    - **小键盘退格接口**：在实现小键盘的基础上，为了输入的合理性以及便捷性，我们另实现了退格按钮（P1）。当按下退格按钮后，若此时键盘输入数据不为空则会删去最后一位输入的数据。
  - **开关输入接口**：除了之前已经提到过的开关之外，我们还实现了其他开关的接口。其中sw21至sw23用来控制测试样例编号的输入，支持编号0-7的输入。sw0-sw14以及sw16用来控制16位立即数的输入（由于sw15损坏，故跳过），能够支持16位二进制数的输入。(**具体的端口绑定请看project_xdc.xdc文件**)
  - **LED灯输出**：我们使用了0-15号LED灯来作为数据输入时的指示以及计算结果的显示，当输入数据或者准备进行运算并按下确认输入完成按钮后，LED灯便会显示对应数据的二进制数，若为1则亮灯，反之不亮。同时使用sw16来作为回文数判断的指示信号灯，在test1当中，若是回文数则亮灯，反之不亮。(**具体的端口绑定请看project_xdc.xdc文件**)

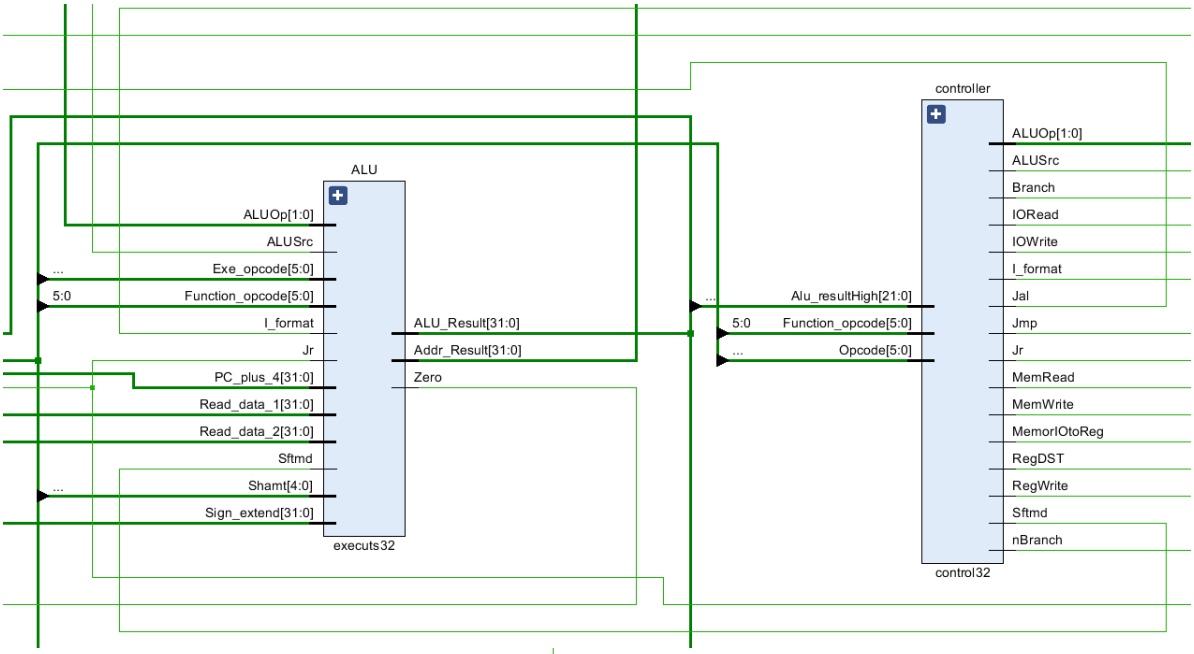
3.3）CPU内部结构

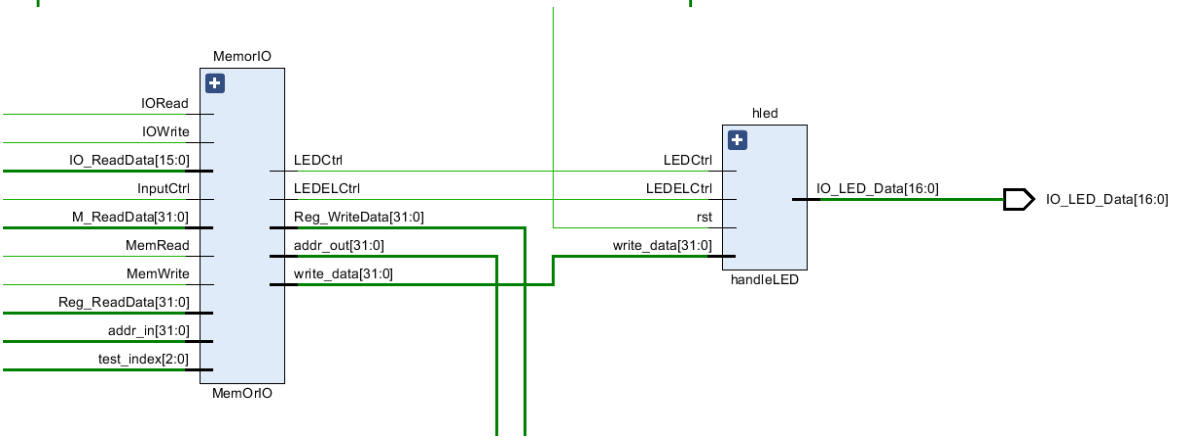
- CPU内部各子模块的接口连接关系图（仅展示部分重要模块）

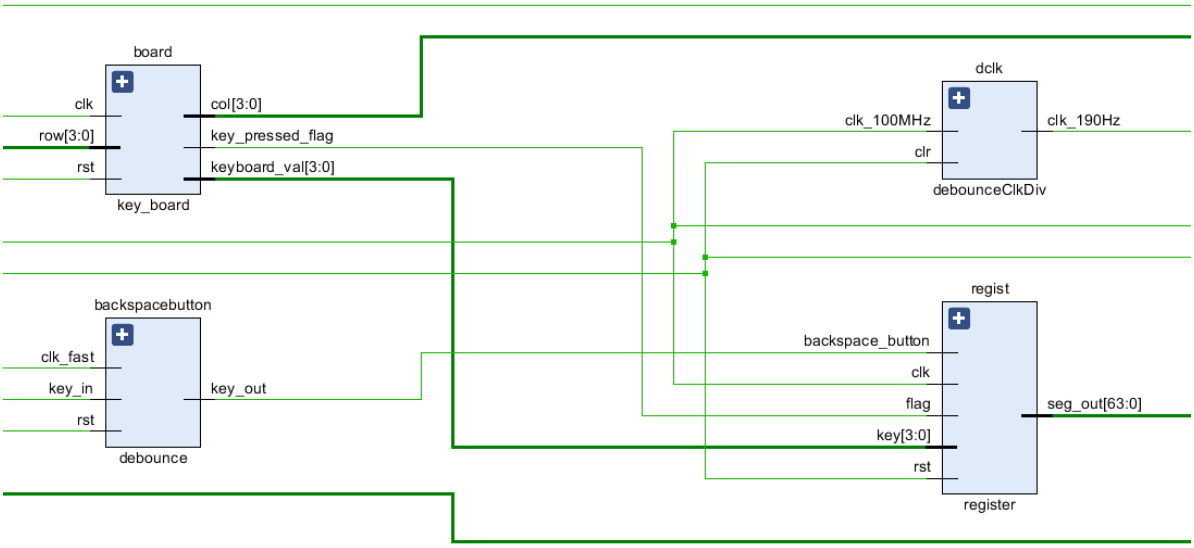





decoder
ALU_result[31:0]
Instruction[31:0]
Jal
MemtoReg
RegDst
RegWrite
clock
mem_data[31:0]
opcplus4[31:0]
reset
Sign_extend[31:0]
read_data_1[31:0]
read_data_2[31:0]
decode32


ALU
ALUOp[1:0]
ALUSrc
Exe_opcode[5:0]
5:0
Function_opcode[5:0]
I_format
Jr
PC_plus_4[31:0]
Read_data_1[31:0]
Read_data_2[31:0]
Sftmd
Shamt[4:0]
Sign_extend[31:0]
ALU_Result[31:0]
Addr_Result[31:0]
Zero
executs32
controller
Alu_resultHigh[21:0]
5:0
Function_opcode[5:0]
Opcode[5:0]
ALUOp[1:0]
ALUSrc
Branch
IORead
IOWrite
I_format
Jal
Jmp
Jr
MemRead
MemWrite
MemorIOtoReg
RegDST
RegWrite
Sftmd
nBranch
control32


MemorIO
IORead
IOWrite
IO_ReadData[15:0]
InputCtrl
M_ReadData[31:0]
MemRead
MemWrite
Reg_ReadData[31:0]
addr_in[31:0]
test_index[2:0]
LEDCtrl
LEDELCtrl
Reg_WriteData[31:0]
addr_out[31:0]
write_data[31:0]
MemOrIO
hled
LEDCtrl
LEDELCtrl
rst
write_data[31:0]
IO_LED_Data[16:0]
handleLED
IO_LED_Data[16:0]


board
clk
row[3:0]
rst
col[3:0]
key_pressed_flag
keyboard_val[3:0]
key_board
dclk
clk_100MHz
clr
clk_190Hz
debounceClkDiv
backspacebutton
clk_fast
key_in
rst
key_out
debounce
regist
backspace_button
clk
flag
key[3:0]
rst
seg_out[63:0]
register

- **CPU内部子模块的设计说明（模块名称、模块功能及功能说明）**

模块名称	模块功能	功能说明

模块名称	模块功能	功能说明
cpuclkDiv	将开发板的时钟信号调整频率后输出	输入开发板时钟信号，输出调整后的CPU clock已经供uart接口使用的时钟信号
counter	将输入的时钟信号调整频率后输出	输入时钟信号，设定参数后输出希望的时钟频率
debounce	消除外设抖动	将外设的信号时延以此来达到消抖的目的
uart_bmpg_0	uart接口模块	连接CPU和uart外设
ifetch	通过传入的指令来计算并存储PC与Next_PC值	通过分析指令的编码来判断PC与Next_PC值
programrom	从指令存储中获取指令	通过ip核根据给予的参数来获取指令
control32	通过传入的参数产生各个控制信号给到下游模块作为指示	通过解析传入的opcode和function_code来给出控制信号
decoder32	含有32个32位寄存器用来存储数据，同时根据传入的参数计算出目标寄存器以及目标数据	通过传入的控制信号来判断目标寄存器以及目标数据
executs32	计算单元，根据参数计算出地址以及数学计算结果	通过控制信号计算出相应结果
dmemory32	通过给予的参数从数据存储中获得数据	包含ip核，同时与uart接口有关，接受uart的写入
key_board	小键盘输入模块，实现了小键盘的接入	通过将小键盘绑定到指定IO数据段来将小键盘数据输入
register	用来存储小键盘输入的数据，方便对数据进行管理	用来存储小键盘输入的数据
shift	与register相关，用来判断当前是删除数据还是写入数据	通过移位来实现增加小键盘输入数据以及删除数据
segTrans	用来将小键盘输入的十六进制数转化成为二进制数	组合逻辑，通过输入判断输出
MemOrIO	外设接口模块，控制数据输入输出流向	通过控制信号判断流向，给出外设指定的IO数据地址
handleLED	储存需要输出的数据	通过寄存器类型实现储存临时输出的数据
seg_ces	七段数码显示管，用来实时显示当前小键盘输入的数据	通过快速的不断刷新显示单个数据来造成多个数据同时显现的效果

（**注：端口规格详情请见附件代码，在此不再赘述**）

Part4-测试说明

4.1）测试总览

4.1.1) 仿真测试一览

测试编号	测试方法	测试描述	测试结果	结论	修改

测试编号	测试方法	测试描述	测试结果	结论	修改
debug测试场景0	仿真测试	对于CPU功能进行最基础的测试，根据用户拨码开关输入的测试用例，直接通过LED灯执行输出	测试结果错误	对于部分寄存器、初始PC地址和NextPC没有初始化，导致仿真器认为PC地址处于Z的不定态，完全无法正常读入指令。	对于verilog所有涉及到寄存器的模块都做初始化和reset预设，对其进行初始化操作
debug测试场景0	仿真测试	同上	测试结果错误	模块传入的参数设置错误，部分参数没有使用	重新设置模块传入
debug测试场景0	仿真测试	同上	测试结果正确	测试正确	
debug测试场景1	仿真测试	对于所有指令进行针对性测试，无IO介入	测试结果正确	测试正确	
debug测试场景2	仿真测试	根据用户拨码开关输入的测试用例，直接通过LED灯执行输出参与一则运算后的结果，如LED输出测试用例+1，输出测试用例左右移结果等	测试结果正确	测试正确，但是同样内容上板测试结果完全不同，无法保证一致性，甚至出现了3+1=0和7+1=4这样的无厘头错误	找bug
debug测试场景2	仿真测试	同上	测试结果正确	不管怎么改，仿真测试就是对的，但是一上板就出问题	后找到bug，放在最后问题描述一栏

测试编号	测试方法	测试描述	测试结果	结论	修改
debug测试场景3	仿真测试	对于测试样例场景1进行仿真测试	测试结果正确	测试正确	
debug测试场景4	仿真测试	对于测试样例场景2进行仿真测试	未知	输出内容过多，无法手动对照是否正确	
debug测试场景5	仿真测试	对于添加了keyboard和回文数指示的版本进行测试	测试结果错误	不符合预期	修复bug
debug测试场景6	仿真测试	对于添加了Uart接口后的版本进行仿真测试	测试结果错误	完全无法正常读取指令，经过debug发现Instruction_o_w没有正常接上Instruction，命名需要注意	修复bug，但是还是没有正常工作
debug测试场景6	仿真测试	对于添加了Uart接口后的版本进行仿真测试	测试结果错误	可以读取指令，但是PC地址总是对比时钟慢了一个周期，对应指令执行完全不正确，读到的指令整体后移。检查后发现是IP核设置问题，prgmip32的ip核中没有取消勾选”Primitives Output Register“选项，导致输出先存进寄存器中才被读到，导致整体后移一个时钟周期	修复bug，版本正常工作
debug测试场景7	仿真测试	终版仿真测试	测试结果正确	结果正确	

4.1.2) 上板测试一览

测试编号	测试方法	测试类型	测试描述	测试结果	结论
debug测试场景0	上板测试	集成测试	根据用户拨码开关输入的测试用例，直接通过LED灯执行输出	测试结果正确	本CPU可以实现该功能
debug测试场景1	上板测试	集成测试	根据用户拨码开关输入的测试用例，直接通过LED灯执行输出参与一则运算后的结果，如LED输出测试用例+1，输出测试用例左右移结果等	测试结果错误	部分结果正确，reset按键不符合预期功能，出现不正确四则运算结果。修改部分bug后测试仍然不符合结果，出现开发板完全不亮灯、不同开发板烧写同一比特流结果不同、四则运算结果错误等问题
debug测试场景1	上板测试	集成测试	根据用户拨码开关输入的测试用例，采用轮询方法，单击按钮后将测试用例内容输出到LED灯	测试结果错误	CPU对轮询完全无法响应
测试场景1	上板测试	集成测试	根据用户输入的测试用例，进入不同的功能模块。包括对寄存器值的判断是否回文，实现与,或,亦或三种逻辑，实现逻辑左移，逻辑右移，算术右移三种移位运算	测试结果正确	本CPU可以实现该功能
测试场景2	上板测试	集成测试	根据用户输入的测试用例，进入不同的功能模块。包括对一片空间内的内容按字节（初始内容高24位均为0，低8位中，第8位在有符号数中位符号位）进行无符号数解析，有符号数解析，并进行排序，最终将结果显示在开发板上	测试结果正确	本CPU可以实现该功能
测试场景3	上板测试	集成测试	对于已经完成两个测试场景的内容，进行Uart串口的重复烧写测试	测试结果正确	一开始肯定是不对的，但是总之，经过许多的debug后，本CPU可以实现该功能

测试编号	测试方法	测试类型	测试描述	测试结果	结论
测试场景3	上板测试	集成测试	最后修复位移时第17位LED会参与输出问题	测试结果错误	仿真完全正确，但是上板总是会出奇怪的bug，比如第二位LED突然在输出时不亮了等问题。实在太玄学了，只好重写了整个方法后可以正常工作。

4.2）asm文件详细描述

4.2.1)测试场景1

1.如何实现IO模块，与用户交互

```
start: lui   $28,0xFFFF
       ori   $28,$28,0xF000

###轮询等待读入信号
begin_1:
lw $s7,0xC7C($28)
bne $s7,$zero,begin_1
begin_2:
lw $s7,0xC7C($28)
beq $s7,$zero,begin_2

###读入数据
lw $a3,0xC78($28)#a3存放的是测试用例

###输出数据
sw $v0,0xC60($28)#将v0也就是a的值显示在led灯上
```

如上，我们约定若lw，sw后面跟的地址大于0xFFFFFC60表示输出数据到开发板，大于0xFFFFFC70表示从开发板读入数据，若小于0xFFFFFC60则表示实际意义上的lw与sw。

我们通过轮询的方式等待读入信号，当我们在开发板上输入完数据后，我们需要摁下并松开开发板上指定的按钮，这样，寄存器$s7的值便会完成0->1->0的转变，便可跳出两个循环等待，这是只需要在后面lw想要的值便能实现用户输入。同时，程序会在下一次轮询时重复循环，继续等待信号完成输入，这样只要用户完成输入后，点击按钮，就可完成输入。

2.测试场景（如何判断回文）

```
###测试场景1，回文判断逻辑，$v0为a的值，最后若$s0,$t1相等则表示a为回文数
###逻辑为不停的将t0（初始值为a）的最后一位取出并逻辑右移t0，将t0的最后一位加到t1并逻辑左移t1，最后t1的值
###为逆序的a，若t1等于s0，则是回文数
addi $s0,$v0,0#将v0存入s0与t0，方便比较
addi $t0,$v0,0
addi $t1,$zero,0
Loop1:
```

```
beq $t0,$zero,over1
sll $t1,$t1,1
addi $t2,$zero,1
and $t2,$t0,$t2
or $t1,$t1,$t2
srl $t0,$t0,1
j Loop1

###测试场景均为简单指令，不再赘述
```

4.2.2)测试场景2

1.测试场景（如何排序）

```
#两层循环变量t7外层，t9内层
addi $t7,$zero,0
addi $t9,$zero,0

loop1:
    addi $t9,$zero,0    # 每次执行外层循环都将内层循环的循环变量置为0
loop2:
    # 获取a1[i] 存放如t2
    addi $t0,$t9,0
    sll $t0, $t0, 2
    lui $at,0
    addu $t0,$at,$t0
    lw $t2,40($t0)

    # 获取a1[i+1] 存放入t3
    addi $t1,$t9,1
    sll $t1, $t1, 2
    lui $at,0
    addu $t1,$at,$t1
    lw $t3,40($t1)

    #bge $t3,$t2,skip  # 如果a[i+1] > a[i],跳转到skip代码块,采用slt，bne替代
    slt $s6,$t2,$t3 #若t2小于t3，则s6为1
    bne $s6,$zero,skip
    #sw $t3,0xC60($28)
    sw $t3,40($t0)   # 否则就执行下面这两句，交换两者的值
    sw $t2,40($t1)

skip:
#for(i=0;i<n-1;i++)
#   for(j=0;j<n-i-1;j++)
   addi $t9,$t9,1   # 内层循环变量自增
   addi $t0,$t9,1   #t0为j+1
   sub $t1,$s0,$t7      # t1为n-i

   slt $s6,$t0,$t1
   bne $s6,$zero,loop2
   # 如果不满足，则将外层循环的循环变量自增，且判断是否还满足循环条件
   addi $t7,$t7,1
   addi $a2,$zero,1
```

```
    sub  $t2,$s0,$a2
    #t2为n-1，t7为i
    slt $s6,$t7,$t2
    bne $s6,$zero,loop1
    addi $t0,$zero,0
```

这次项目的排序功能我们采用了冒泡排序的方式，实现对数组的排序

2.测试场景（如何将8bit无符号数转化为8bit有符号数的补码形式）

```
addi $t0,$zero,0
changeLoop:
beq $t0, $s0,test2_end  # $t0 == $s0 的时候跳出循环
sll $t1, $t0, 2            # $t1 = $t0 << 2，即 $t1 = $t0 * 4
lui $at,0
addu $t1,$at,$t1

#取出数据集2中数据存入s1
lw $s1, 80($t1)
#s2为符号位
srl $s2,$s1,7

addi $t5,$zero,0
beq $s2,$t5,unchange

addi $a2,$zero,128
sub $s1,$s1,$a2#s1为该数值的绝对值
#s1取反加1
lui $at,0xffff
ori $at,$at,0xffff
xor $s1,$s1,$at
lui $at,0
addi $s1,$s1,1
sw $s1,80($t1)

unchange:
addi $t0, $t0, 1           # $t0 = $t0 + 1
j changeLoop

test2_end:
j begin_1
```

这里我们需要将8bit无符号数视作有符号数，其中最高位为符号位，低7位为绝对值，我们的思路是将最高位去掉（减去128），然后将该32位数据（低7bit为原无符号数的低7bit，高25bit均为0）与32位全1数据相亦或，这样就相当于逐位取反，最后加1，便得到了该数据的有符号数补码形式

3.测试场景（如何交替5秒闪烁）

```
addi $t7,$zero,1500
addi $t9,$zero,15000

loop73:
    addi $t9,$zero,15000    # 每次执行外层循环都将内层循环的循环变量置为0
loop74:
   addi $t5,$zero,1
   sub $t9,$t9,$t5   # 内层循环变量自增，且判断是否还满足循环条件
   bne $t9,$zero,loop74
   sub $t7,$t7,$t5
   bne $t7,$zero,loop73
```

这里我们采用的方式是塞一堆无用的指令让CPU运算，经过调整，发现该规模的二重循环符合要求

Part5-bonus部分

本次Project我们完成的bonus有：支持小键盘读入（小键盘带有退格功能），七段数码管显示结果，uart通信，具体的bonus演示可参考我们的视频

Part6-问题及总结

本次Project中遇到了许多问题，也学习到了非常多知识。平心而论，在做完这个project之前，我一直不明白计算机系的同学为什么要学习计算机组成原理这门课程，特别是还需要学习verilog语言以及使用它来写一个CPU。在边写代码边调试、崩溃和完善的反复循环中，我觉得这个答案慢慢明晰了起来。

verilog绝不能称得上是一门好用的语言，vivado更是和“好用的环境“完全不沾边。硬件编程和高级语言编程不能说是完全一致，只能说是差别过大。“顺序执行”这个在高级语言里已经是思维定势的过程，放在verilog中出的乱子可以堆出一座喜马拉雅。不管是组合逻辑中接线式的编程也好，时序逻辑依靠时钟触发的always语句块也罢，这都是对于计算机编程思维的一种“重构”。如果不摈弃高级语言的思维定势，multi-driven会充斥你的代码不说，之后仿真、综合、上板的debug也足以让你在十八层地狱中遨游。

除了编写代码的过程，硬件编程的debug过程和高级语言编程相比也存在着巨大的鸿沟。仿真可以说是最像高级语言编程的debug方式，通过给定输入、逻辑过程，并拉出接线来看对应的寄存器行为状况。虽然波形图凌乱无比，但我却非常喜欢这种直观的debug方式。不同于高级语言编程的单步调试，波形图用一种非常直观的方式列出了整个电路跟随时钟变化和输入变化的波动，尤其考验人的逻辑能力和分析能力。事实上，它的直观令人神清气爽。我在测试过程中，如果发现仿真图出错，我会跃跃欲试兴致勃勃地去找问题的根源，因为我知道只要仿真出错一定能问题来源。我甚至在某些瞬间觉得波形图仿真比高级语言编程的debug更方便直观。

隔壁组的同学非常不理解我怎么会认为在波形图上发现bug是一件令人愉快的事情。这就不得不说到在这次project中，我们遇到的最大阻碍——仿真结果和上板结果不一致。仿真出错尚且有得调试，但是仿真结果和上板结果不一致真的令人头脑发昏，无从下手，完全没办法做进一步操作。在代码完成初期，我们在板子上跑最简单的汇编代码：输出输入的数加一。就是这个简单的代码卡了我一整个通宵。在完全通过仿真的情况下，它烧写到板子上要么是不显示，要么是显示错误的答案，要么甚至是要一直按着reset键不放才能正常工作。这真的让我丈二和尚摸不着头脑，我绞尽脑汁怎么也想不到什么错误才会导

致这样的结果。后来我禁用了reset键，输出a+1的程序运作看似正常了。然而我发现它跑出来的代码3+1=0出现了7+1=4，而其他所有的算式都是对的。

那天晚上，我坐在一教的实验室里，觉得自己见鬼了。ALU模块里，涉及到加法运算的指令是由vivado综合出的。我想了想可能是板子的问题，于是我换了块板子，甚至换了数字逻辑的红色小板子。然而结果令人更觉恐惧，同一个比特流，烧到三块板子上，出现三种截然不同的行为模式和结果。更为恐怖的事发生在第二天，当我一大早晨打开板子又进行了一次烧录，这次它居然跑出了正确的答案。我盯着板子直掉san。下午，我们在图书馆奋战一整天解决了轮询读入的问题，好不容易上板测试成功了，保存了比特流，去吃了个晚饭，回来测试的时候，同一个比特流烧到同一个板子上，它又不工作了，甚至一个灯也不亮了。这个时候，我对唯物主义思想产生了些许动摇，我实在找不到可能去解释问题出在哪里。我和队友抓破脑袋，把所有的原因一股脑推给板子，觉得是因为板子坏了或者出了某些问题才导致我们的代码出现问题。

可是柳暗花明又一村，测试一筹莫展之际，准备先把Uart写完的队友偶然发现了我们一队三个人居然同时忽略了一点——单周期CPU的时钟应该是23MHz，而且不能够高于这个数值。我们猛然惊醒，发觉自己之前所做的事情是多么地愚蠢。这真的是一个蠢到不能再蠢的错误，但凡有点常识都不会犯这样的低级错误。所有的所谓“灵异事件”都能得到一个合理的解释，我们居然一直在拿100MHz去跑一个单周期CPU。这简直是对于板子算力的虐待。7+1=4可能是在跨时钟周期时进位出现了错误；不同的板子之前芯片算力可能有所不同；一直按着reset键清空PC可能能跑通前几个指令；时钟周期过大让各种微小因素的扰动被无限放大……

我和队友向我们的开发板郑重道歉。我真的没想到，它拼了老命为我们算来的不完善内容，我们却因此认为它是一块坏的板子。把时钟周期改为23MHz后，一切事情开始朝着正常的方向发展。轮询突然有用了，指令突然跑通了，测试场景1也莫名其妙地就过了。板子除了有些开关和按钮波动过大需要全部用190Hz的时钟消抖以外，其他并没有什么问题。project写到这里真的是我最开心的时候，所有的灵异事件都成了过去式，我正在逐渐崩塌的逻辑大厦慢慢保住了。几乎没有耗费多少时间，我们就跑通了两个场景，并开始引入Uart进行测试。

我分析了一下我们被一个如此简单的问题蒙蔽的原因。仿真无法将这个问题呈现，因为仿真提供的时钟10微秒变换一周期，和板子提供的时钟完全不同。此外，编程逻辑让我们一直在顺着问题去寻找它的来源，事实上时钟所带来的问题（或许）是把一些微小的、不可知的问题放大。这样的问题没有debug途径，板子完全不符合逻辑运行导致我们没法debug。

俗话说，事物总是呈螺旋上升的态势。发现时钟的bug以后，我突然对vivado和verilog又有了信心。我觉得所有问题我们都能找到答案，再也不会出现灵异事件了。然而问题并没有那么简单。沾上IP核以后，就连某些仿真的逻辑都和代码的逻辑逐渐对不上号。首先是23MHz的时钟并不能用于仿真，仿真输入的时钟信号被IP核换出来就变成无法在波形图中识别的信号。然后是因为IP核的一些小设置错误，导致波形图整体后移一周期，但是程序逻辑按照原来周期进行，出现各种奇妙且不可理喻的错误。

project进程越往后，我们就越小心翼翼。对于每一丝一毫的改动，我们恨不得存上”七八百个“备份版本。有些时候，我们连注释都不敢删除，生怕删了一行注释会导致程序跑的结果不对。事实上，我们的程序跑到最后还是有我实在不能理解的没有逻辑的bug出现，比如在其它指令执行完全正确的情况下，向其它和输出无关的寄存器存入内容，LED灯会随机亮起一些信号。再三再四再五再六确认了这个问题对正常程序运行没有影响，我们决定暂且置之不理。队友说，如果一直要深究的话，可能就要三天三夜睡不着觉了。我们最终决定，不是每个bug都能找到原因，”与verilog和解“，不再去思考这个问题。

再到后来，临答辩一个小时前，我们考虑算数和逻辑右移的不同，重新规划了一下输出，仅仅改了几条语句。但是第二个LED灯却在某些情况下怎么也不输出了，但最终寄存器里的值完全正确。又是仿真全对，上板出了奇怪的问题。这时候离答辩仅剩2小时，我放弃debug了，根本不想找原因，完全重写了那个模块，它又神奇地变好了。

总之，我们最后答辩的时候，板子上仍然带着不可名状的bug，虽然最后没有影响逻辑功能，板子和程序也很给力，通过了王老师非常非常严苛的逻辑测试。我们把随机乱闪的灯当成了”指示按下了轮询控制按钮“的feature，最终的答辩结果也很令人满意。曾经，我对于编程的看法是，一个程序是不允许出现bug的。写一个最短路算法，如果出了bug，它终究会犯错，是恶性的。但是最终完成这个CPU后，即使板子仍然做出了让我不解的、无法掌握逻辑的（或许是）随机闪灯的奇怪反应，但是我一点也不觉得遗憾。看RTL图、看schematic、看仿真、看代码都找不到问题的情况下，我真的无从得知vivado在综合和仿真过程中对我们的程序做了些什么。或许编程本就不应该完美主义吧。尽力做到自己的最好就行了。

回头看看我的版本命名，“突然好了.bit”、“带reset.bit“、”debug到一半.bit“、”7+1=4.bit“、”俩板子不一样.bit“这种奇葩命名，突然觉得我们所做的一切debug都是有意义的，即使它们折磨了我很多个通宵。对于我个人而言，我很感谢这次project。虽然它让我的生物钟彻底紊乱，但是它也消耗了我的剩余精力，帮助我渡过了一个本应该会是非常痛苦的夜晚，不至自己与自己内耗。

答辩结束后，我们给板子拍了多角度照片留念。我想我会记住这一块板子，它曾经那么努力，跑出了原本跑不出的算力，还被我们误解是一块坏板子，以一己之力让project进程不至于拖慢过多。大概率我以后不会再碰到要写如此巨量verilog的情况了。虽然仍有各种没能明白的问题和逻辑，但是，不管怎么样，相信事物总是会螺旋上升的吧。