



中山大學
SUN YAT-SEN UNIVERSITY

《计算机组成原理实验》 实验报告

(项目一)

学 院 名 称 : 计算机学院

专业 (班级) : 计算机科学与技术

学 生 姓 名 : 熊彦钧

学 号 : 23336266

时 间 : 2024 年 11 月 23 日

项目一：单周期CPU设计与实现

一、实验目的

- (1) 掌握单周期 CPU 数据通路图的构成、原理及其设计方法；
- (2) 掌握单周期 CPU 的实现方法，代码实现方法；
- (3) 认识和掌握指令与 CPU 的关系；
- (4) 掌握测试单周期 CPU 的方法。

二、实验内容

设计一个单周期 CPU，该 CPU 至少能实现以下指令功能操作。指令与格式如下：

(红色的为在实验要求以外额外添加的指令)

==> 算术运算指令

(1) add rd, rs, rt

000000	rs(5 位)	rt(5 位)	rd(5 位)	00000 100000
--------	---------	---------	---------	--------------

功能：GPR[rd] \leftarrow GPR[rs] + GPR[rt]。

(2) sub rd, rs, rt

000000	rs(5 位)	rt(5 位)	rd(5 位)	00000 100010
--------	---------	---------	---------	--------------

功能：GPR[rd] \leftarrow GPR[rs] - GPR[rt]。

(3) addiu rt, rs, immediate

001001	rs(5 位)	rt(5 位)	immediate(16 位)	
--------	---------	---------	-----------------	--

功能：GPR[rt] \leftarrow GPR[rs] + sign_extend(immediate); immediate 做符号扩展再参加“与”运算。

(4) addu rd, rs, rt

000000	rs(5 位)	rt(5 位)	rd(5 位)	00000 100001
--------	---------	---------	---------	--------------

功能：GPR[rd] \leftarrow GPR[rs] + GPR[rt]。

(5) subu rd, rs, rt

000000	rs(5 位)	rt(5 位)	rd(5 位)	00000 100011
--------	---------	---------	---------	--------------

功能：GPR[rd] \leftarrow GPR[rs] - GPR[rt]。

(6) addi rt, rs, immediate

001000	rs(5 位)	rt(5 位)	immediate(16 位)	
--------	---------	---------	-----------------	--

功能：GPR[rt] \leftarrow GPR[rs] + sign_extend(immediate); immediate 做符号扩展再参加“与”运算。

==> 逻辑运算指令

(7) andi rt, rs, immediate

001100	rs(5 位)	rt(5 位)	immediate(16 位)	
--------	---------	---------	-----------------	--

功能：GPR[rt] \leftarrow GPR[rs] and zero_extend(immediate); immediate 做 0 扩展再参加“与”运算。

(8) and rd, rs, rt

000000	rs(5 位)	rt(5 位)	rd(5 位)	00000 100100
--------	---------	---------	---------	--------------

功能: $\text{GPR}[\text{rd}] \leftarrow \text{GPR}[\text{rs}] \text{ and } \text{GPR}[\text{rt}]$ 。

(9) `ori rt, rs, immediate`

001101	rs(5 位)	rt(5 位)	immediate(16 位)
--------	---------	---------	-----------------

功能: $\text{GPR}[\text{rt}] \leftarrow \text{GPR}[\text{rs}] \text{ or } \text{zero_extend}(\text{immediate})$ 。

(10) `or rd, rs, rt`

000000	rs(5 位)	rt(5 位)	rd(5 位)	00000 100101
--------	---------	---------	---------	--------------

功能: $\text{GPR}[\text{rd}] \leftarrow \text{GPR}[\text{rs}] \text{ or } \text{GPR}[\text{rt}]$ 。

(11) `xori rt, rs, immediate`

001110	rs(5 位)	rt(5 位)	immediate(16 位)
--------	---------	---------	-----------------

功能: $\text{GPR}[\text{rt}] \leftarrow \text{GPR}[\text{rs}] \text{ xor } \text{zero_extend}(\text{immediate})$ 。

(12) `xor rd, rs, rt`

000000	rs(5 位)	rt(5 位)	rd(5 位)	00000 100110
--------	---------	---------	---------	--------------

功能: $\text{GPR}[\text{rd}] \leftarrow \text{GPR}[\text{rs}] \text{ xor } \text{GPR}[\text{rt}]$ 。

==> 移位指令

(13) `sll rd, rt, sa`

000000	00000	rt(5 位)	rd(5 位)	sa(5 位)	000000
--------	-------	---------	---------	---------	--------

功能: $\text{GPR}[\text{rd}] \leftarrow \text{GPR}[\text{rt}] \ll \text{sa}$ 。

==> 比较指令

(14) `slti rt, rs, immediate` 带符号数

001010	rs(5 位)	rt(5 位)	immediate(16 位)
--------	---------	---------	-----------------

功能: if $\text{GPR}[\text{rs}] < \text{sign_extend}(\text{immediate})$ $\text{GPR}[\text{rt}] = 1$ else $\text{GPR}[\text{rt}] = 0$ 。

(15) `slt rd, rs, rt`

000000	rs(5 位)	rt(5 位)	rd (5 位)	00000 101010
--------	---------	---------	----------	--------------

功能: $\text{GPR}[\text{rd}] \leftarrow (\text{GPR}[\text{rs}] < \text{GPR}[\text{rt}])$ 。

(16) `sltu rd, rs, rt`

000000	rs(5 位)	rt(5 位)	rd (5 位)	00000 101011
--------	---------	---------	----------	--------------

功能: if $(0 \mid \mid \text{GPR}[\text{rs}]) < (0 \mid \mid \text{GPR}[\text{rt}])$ $\text{GPR}[\text{rd}] \leftarrow 0 \text{GPRLEN} - 1 \mid \mid 1$ else $\text{GPR}[\text{rd}] \leftarrow 0 \text{GPRLEN}$ 。

==> 存储器读/写指令

(17) `sw rt, offset (rs)` 写存储器

101011	rs(5 位)	rt(5 位)	offset(16 位)
--------	---------	---------	--------------

功能: $\text{memory}[\text{GPR}[\text{base}] + \text{sign_extend}(\text{offset})] \leftarrow \text{GPR}[\text{rt}]$ 。

(18) `lw rt, offset (rs)` 读存储器

100011	rs(5 位)	rt(5 位)	offset (16 位)
--------	---------	---------	---------------

功能: $\text{GPR}[\text{rt}] \leftarrow \text{memory}[\text{GPR}[\text{base}] + \text{sign_extend}(\text{offset})]$ 。

==> 分支指令

(19) `beq rs, rt, offset`

000100	rs(5 位)	rt(5 位)	offset (16 位)
--------	---------	---------	---------------

功能: if $(\text{GPR}[\text{rs}] = \text{GPR}[\text{rt}])$ $\text{pc} \leftarrow \text{pc} + 4 + \text{sign_extend}(\text{offset}) \ll 2$

else $\text{pc} \leftarrow \text{pc} + 4$

特别说明: **offset** 是从 **PC+4** 地址开始和转移到的指令之间指令条数。**offset** 符号扩展之后左移 2 位再相加。为什么要左移 2 位? 由于跳转到的指令地址肯定是 4 的倍数 (每条指令占 4 个字节), 最低两位是 “00”, 因此将 **offset** 放进指令码中的时候, 是右移了 2

位的，也就是以上说的“指令之间指令条数”。

(20) bne rs,rt, offset

000101	rs(5 位)	rt(5 位)	offset (16 位)
--------	---------	---------	---------------

功能: if(GPR[rs] != GPR[rt]) $pc \leftarrow pc + 4 + \text{sign_extend}(\text{offset}) << 2$

else $pc \leftarrow pc + 4$

(21) blez rs, offset

000110	rs(5 位)	00000	offset (16 位)
--------	---------	-------	---------------

功能: if(GPR[rs] ≤ 0) $pc \leftarrow pc + 4 + \text{sign_extend}(\text{offset}) << 2$

else $pc \leftarrow pc + 4$ 。

==>跳转指令

(22) j addr

000010	addr(26 位)
--------	------------

功能: $PC \leftarrow \{PC[31:28], \text{addr}, 2'b0\}$, 无条件跳转。

说明: 由于 MIPS32 的指令代码长度占 4 个字节, 所以指令地址二进制数最低 2 位均为 0, 将指令地址放进指令代码中时, 可省掉! 这样, 除了最高 6 位操作码外, 还有 26 位可用于存放地址, 事实上, 可存放 28 位地址, 剩下最高 4 位由 $pc+4$ 最高 4 位拼接上。

==> 停机指令

(23) halt

111111	00000000000000000000000000000000(26 位)
--------	--

功能: 停机; 不改变 PC 的值, PC 保持不变。

三、实验原理

单周期 CPU 指的是一条指令的执行在一个时钟周期内完成, 然后开始下一条指令的执行, 即一条指令用一个时钟周期完成。电平从低到高变化的瞬间称为时钟上升沿, 两个相邻时钟上升沿之间的时间间隔称为一个时钟周期。时钟周期一般也称振荡周期 (如果晶振的输出没有经过分频就直接作为 CPU 的工作时钟, 则时钟周期就等于振荡周期。若振荡周期经二分频后形成时钟脉冲信号作为 CPU 的工作时钟, 这样, 时钟周期就是振荡周期的两倍。)

CPU 在处理指令时, 一般需要经过以下几个步骤:

(1) 取指令(IF): 根据程序计数器 PC 中的指令地址, 从存储器中取出一条指令, 同时, PC 根据指令字长度自动递增产生下一条指令所需要的指令地址, 但遇到“地址转移”指令时, 则控制器把“转移地址”送入 PC, 当然得到的“地址”需要做些变换才送入 PC。

(2) 指令译码(ID): 对取指令操作中得到的指令进行分析并译码, 确定这条指令需要完成的操作, 从而产生相应的操作控制信号, 用于驱动执行状态中的各种操作。

(3) 指令执行(EXE): 根据指令译码得到的操作控制信号, 具体地执行指令动作, 然后转移到结果写回状态。

(4) 存储器访问(MEM): 所有需要访问存储器的操作都将在这个步骤中执行, 该步骤给出存储器的数据地址, 把数据写入到存储器中数据地址所指定的存储单元或者从存储器中得到数据地址单元中的数据。

(5) 结果写回(WB): 指令执行的结果或者访问存储器中得到的数据写回相应的目的寄存器中。

单周期 CPU, 是在一个时钟周期内完成这五个阶段的处理。



图 1 单周期 CPU 指令处理过程

MIPS 指令的三种格式：

R 类型：

31	2625	2120	1615	1110	65	0
op	rs	rt	rd	sa	funct	
6 位	5 位	5 位	5 位	5 位	6 位	

I 类型：

31	2625	2120	1615	0
op	rs	rt	immediate	
6 位	5 位	5 位	16 位	

J 类型：

31	2625	0
op	address	
6 位	26 位	

其中，

op：为操作码；

rs：只读。为第 1 个源操作数寄存器，寄存器地址（编号）是 00000~11111，00~1F；

rt：可读可写。为第 2 个源操作数寄存器，或目的操作数寄存器，寄存器地址（同上）；

rd：只写。为目的操作数寄存器，寄存器地址（同上）；

sa：为位移量（shift amt），移位指令用于指定移多少位；

funct：为功能码，在寄存器类型指令中（R 类型）用来指定指令的功能与操作码配合使用；

immediate：为 16 位立即数，用作无符号的逻辑操作数、有符号的算术操作数、数据加载（Load）/数据保存（Store）指令的数据地址字节偏移量和分支指令中相对程序计数器（PC）的有符号偏移量；

address：为地址。

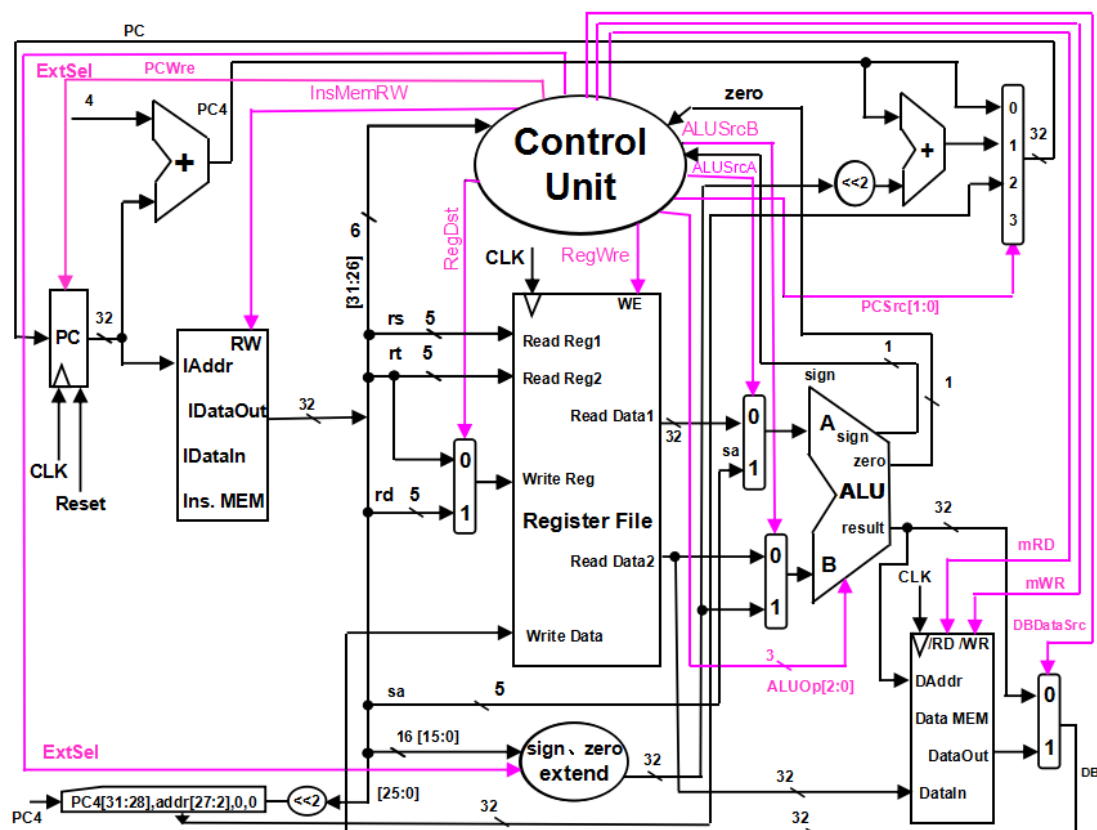


图 2 单周期 CPU 数据通路和控制线路图

图 2 是一个简单的基本上能够在单周期 CPU 上完成所要求设计的指令功能的数据通路和必要的控制线路图。其中指令和数据各存储在不同存储器中，即有指令存储器和数据存储器。访问存储器时，先给出内存地址，然后由读或写信号控制操作。对于寄存器组，先给出寄存器地址，读操作时不需要时钟信号，输出端就直接输出相应数据；而在写操作时，在 WE 使能信号为 1 时，在时钟边沿触发将数据写入寄存器。图中控制信号作用如表 1 所示，表 2 是 ALU 运算功能表。

表 1 控制信号的作用

控制信号名	状态 “0”	状态 “1”
Reset	初始化 PC 为 0	PC 接收新地址
PCWre	PC 不更改，相关指令：halt	PC 更改，相关指令：除指令 halt 外
ALUSrcA	来自寄存器堆 data1 输出，相关指令：add、sub、addiu、or、and、andi、ori、slti、beq、bne、blez、sw、lw	来自移位数 sa，同时，进行 (zero-extend)sa，即 $\{27\{1'b0\}, sa\}$ ，相关指令：sll
ALUSrcB	来自寄存器堆 data2 输出，相关指令：add、addu、sub、subu、or、and、xor、slt、sltu、beq、bne、blez	来自 sign 或 zero 扩展的立即数，相关指令：addi、addiu、andi、ori、xori、slti、sw、lw
DBDataSrc	来自 ALU 运算结果的输出	来自数据存储器 (Data MEM) 的输出

		出，相关指令：lw
RegWre	无写寄存器组寄存器，相关指令：beq、bne、blez、sw、halt、j	寄存器组写使能，相关指令：add、addu、addi、addiu、sub、subu、ori、or、and、andi、xor、xori、slt、sltu、slti、sll、lw
InsMemRW	写指令存储器	读指令存储器（取指令）(Ins. Data)
mRD	输出高阻态	读数据存储器，相关指令：lw
mWR	无操作	写数据存储器，相关指令：sw
RegDst	写寄存器组寄存器的地址，来自 rt 字段（用于 I 型指令）	写寄存器组寄存器的地址，来自 rd 字段（用于 R 型指令）
ExtSel	(zero-extend)immediate (0 扩展)	(sign-extend)immediate (符号扩展)
PCSrc[1..0]	00: $pc \leftarrow pc+4$ ，相关指令：add、addiu、sub、or、ori、and、andi、slti、sll、sw、lw、beq(zero=0)、bne(zero=1)、blez(sign=0&&zero=0); 01: $pc \leftarrow pc+4+(sign-extend)immediate$ ，相关指令：beq(zero=1)、bne(zero=0)、blez(sign=1 zero=1); 10: $pc \leftarrow \{(pc+4)[31:28], addr[27:2], 2'b00\}$ ，相关指令：j; 11: 未用	
ALUOp[2..0]	ALU 8 种运算功能选择(000-111)，看功能表	

相关部件及引脚说明：**Instruction Memory: 指令存储器，**

Iaddr, 指令存储器地址输入端口

IDataIn, 指令存储器数据输入端口（指令代码输入端口）

IDataOut, 指令存储器数据输出端口（指令代码输出端口）

RW, 指令存储器读写控制信号，为 0 写，为 1 读

Data Memory: 数据存储器，

Daddr, 数据存储器地址输入端口

DataIn, 数据存储器数据输入端口

DataOut, 数据存储器数据输出端口

/RD, 数据存储器读控制信号，为 0 读

/WR, 数据存储器写控制信号，为 0 写

Register File: 寄存器组

Read Reg1, rs 寄存器地址输入端口

Read Reg2, rt 寄存器地址输入端口

Write Reg, 将数据写入的寄存器端口，其地址来源 rt 或 rd 字段

Write Data, 写入寄存器的数据输入端口

Read Data1, rs 寄存器数据输出端口

Read Data2, rt 寄存器数据输出端口

WE, 写使能信号，为 1 时，在时钟边沿触发写入

ALU: 算术逻辑单元

result, ALU 运算结果

zero, 运算结果标志，结果为 0，则 zero=1；否则 zero=0

sign, 运算结果标志，结果最高位为 0，则 sign=0，正数；否则，sign=1，负数

表 2 ALU 运算功能表

ALUOp[2..0]	功能	描述
000	$Y = A + B$	加
001	$Y = A - B$	减
010	$Y = B \ll A$	B 左移 A 位
011	$Y = A \vee B$	或
100	$Y = A \wedge B$	与
101	$Y = (A < B) ? 1 : 0$	比较 A 与 B 不带符号
110	$Y = (((rega < regb) \&\& (rega[31] == regb[31])) \parallel ((rega[31] == 1 \&\& regb[31] == 0))) ? 1 : 0$	比较 A 与 B 带符号
111	$Y = A \oplus B$	异或

需要说明的是以上数据通路图是根据要实现的指令功能的要求画出来的，同时，还必须确定 ALU 的运算功能。从数据通路图上可以看出控制单元部分需要产生各种控制信号，当然，也有些信号必须要传送给控制单元。从指令功能要求和数据通路图的关系得出以上表 1，这样，从表 1 可以看出各控制信号与相应指令之间的相互关系，根据这种关系就可以得出控制信号与指令之间的关系表（留给学生完成），再根据关系表可以写出各控制信号的逻辑表达式，这样控制单元部分就可实现了。

指令执行的结果总是在时钟下降沿保存到寄存器和存储器中，PC 的改变是在时钟上升沿进行的，这样稳定性较好。

Basys3板的使用说明

怎么认识CPU执行指令后，指令的正确与否？

说明：指令存储器中的指令地址范围：0~255；数据存储器中的数据地址范围：0~255。也就是只使用低8位。

开关说明：（以下数据都来自CPU）（SW15、SW14、SW0为Basys3板上开关名，BTN0为按键名）

开关SW_in (SW15、SW14)状态情况如下。显示格式： 左边两位数码管BB： 右边两位数码管BB。以下是数码管的显示内容。

SW_in = 00：显示 当前 PC值:下条指令PC值

SW_in = 01：显示 RS寄存器地址:RS寄存器数据

SW_in = 10：显示 RT寄存器地址:RT寄存器数据

SW_in = 11：显示 ALU结果输出 :DB总线数据。

复位信号（reset）接开关SW0，按键（单脉冲）接按键BTN0。

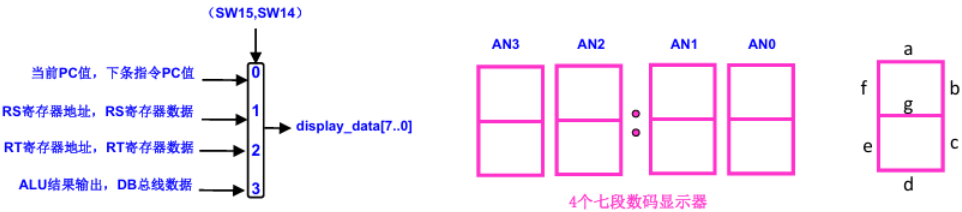
另外，

1、7段数码管的位控信号AN3-AN0，每组编码中只有一位为0（亮），其余都是1（灭）。

2、七段数码显示器编码与引脚对应关系为（左到右，高到低）：七段共阳极数码管->1gfedcba;七段共阴极数码管->0gfedcba。

3、必须有足够的刷新频率，频率太高或太低都不成，系统时钟必须适当分频，否则效果达不到。

指令执行采用单步（按键控制）执行方式，由开关（SW15、SW14）控制选择查看数码管上的相关信息，地址和数据。地址或数据的输出经以下模块代码转换后接到数码管上。



设计思路：

1、实现CPU在板上运行需要两个时钟信号，CPU工作时钟和Basys3板系统时钟。CPU工作时钟即为按键，是CPU正常工作时钟信号，按键必须进行消抖处理；Basys3板系统时钟即为板提供的正常工作时钟信号，即为100MHZ。Basys3板系统时钟信号引脚对应管脚W5。

2、每个按键周期，4个数码管都必须刷新一次。数码管位控信号 AN3-AN0是1110、1101、1011、0111，为0时点亮该数码管，当然，还应该为数码管各位“1gfedcba”引脚输出信号，最高位为“1”。比如，“当前PC值”低8位中的高4位和低4位，必须经下页转换后送给数码管各引脚。

显示模块设计大概分为4个部分：

(1) 对Basys3板系统时钟信号进行分频，分频的目的用于计数器；

(2) 生成计数器，计数器用于产生4个数。这4数用于控制4个数码管；

(3) 根据计数器产生的数生成数码管相应的位控信号（输出）和接收CPU来的相应数据；

(4) 将从CPU 接收到的相应数据转换为数码管显示信号，再送往数码管显示（输出），即下页内容。

还必须清楚，数码管显示的内容是受开关控制的，不同情况显示内容是不同的。看上页说明。

LED	PIN	CLOCK	PIN	SWITCH	PIN	BUTTON	PIN	Seven-segment digital tube	PIN
LD0	U16	MRCC	W5	SW0	V17	BTNU	T18	AN0	U2
LD1	E19			SW1	V16	BTNR	T17	AN1	U4
LD2	U19			SW2	W16	BTND	U17	AN2	V4
LD3	V19			SW3	W17	BTNL	W19	AN3	W4
LD4	W18			SW4	W15	BTNC	U18	CA	W7
LD5	U15			SW5	V15			CB	W6
LD6	U14			SW6	W14			CC	U8
LD7	V14			SW7	W13			CD	V8
LD8	V13	USB (J2)	P1H	SW8	V2			CE	U5
LD9	V3	PS2_CLK	C17	SW9	T3			CF	V5
LD10	W3	PS2_DAT	B17	SW10	T2			CG	U7
LD11	U3			SW11	R3			DP	V7
LD12	P3			SW12	W2				
LD13	N3			SW13	U1				
LD14	P1			SW14	T1				
LD15	L1			SW15	R2				

四、实验设备

PC机一台，BASYS 3 实验板一块，Xilinx Vivado 开发软件一套。

一.实验过程与结果

第一部分：相关数据

1.所要实现的指令的指令格式表

指令	指令格式						指令示例
R 型指令	op[31: 26]	rs[25: 21]	rt[20: 16]	rd[16: 11]	shamt[10: 6]	func[5: 0]	
add	000000	rs	rt	rd	0	100000	add \$d, \$s, \$t
addu	000000	rs	rt	rd	0	100001	addu \$d, \$s, \$t

sub	000000	rs	rt	rd	0	100010	sub \$d, \$s, \$t
subu	000000	rs	rt	rd	0	100011	subu \$d, \$s, \$t
and	000000	rs	rt	rd	0	100100	and \$d, \$s, \$t
or	000000	rs	rt	rd	0	100101	or \$d, \$s, \$t
xor	000000	rs	rt	rd	0	100110	xor \$d, \$s, \$t
slt	000000	rs	rt	rd	0	101010	slt \$d, \$s, \$t
sltu	000000	rs	rt	rd	0	101011	sltu \$d, \$s, \$t
sll	000000	0	rt	rd	shamt	000000	sll \$d, \$t, shamt
I 型指令	op[31: 26]	rs[25: 21]	rt[20: 16]	immediate[15: 0]			
addi	001000	rs	rt	immediate(-,+)			addi \$t, \$s, imm
addiu	001001	rs	rt	immediate(-,+)			addiu \$t, \$s, imm
andi	001100	rs	rt	immediate(0,+)			andi \$t, \$s, imm
ori	001101	rs	rt	immediate(0,+)			ori \$t, \$s, imm
xori	001110	rs	rt	immediate(0,+)			xori \$t, \$s, imm
lw	100011	rs	rt	immediate(-,+)			lw \$t, offset(\$s)
sw	101011	rs	rt	immediate(-,+)			sw \$t, offset(\$s)
beq	000100	rs	rt	immediate(-,+)			beq \$s, \$t, offset
bne	000101	rs	rt	immediate(-,+)			bne \$s, \$t, offset
slti	001010	rs	rt	immediate(-,+)			slti \$t, \$s, imm
blez	000110	rs	00000	immediate(-,+)			bltz \$s, offset
J 型指令	op[31: 26]	index[25: 0]					
j	000010	address					j target
停机指令							
halt	111111	0000000000000000000000000000 (26 位)					

该表的创建目的为把所有指令的指令格式以及MIPS中该指令的示例整合在一起，方便在生成测试程序段时进行查表，避免因指令之间的差异导致生成测试程序段出现错误。

2. 各个指令的控制信号真值表

指令名称	OP码	func码	PCWre	ALUSrcA	ALUSrcB	DBDataSrc	RegWre	InsMemRW	mRD	mWR	RegDst	ExtSel	PCSrc	ALUOp
add	000000	100000	1	0	0	0	1	1	x	0	1	x	00	000
addu	000000	100001	1	0	0	0	1	1	x	0	1	x	00	000
sub	000000	100010	1	0	0	0	1	1	x	0	1	x	00	001
subu	000000	100011	1	0	0	0	1	1	x	0	1	x	00	001
and	000000	100100	1	0	0	0	1	1	x	0	1	x	00	100
or	000000	100101	1	0	0	0	1	1	x	0	1	x	00	011
xor	000000	100110	1	0	0	0	1	1	x	0	1	x	00	111
slt	000000	101010	1	0	0	0	1	1	x	0	1	x	00	110
sltu	000000	101011	1	0	0	0	1	1	x	0	1	x	00	101
sll	000000	000000	1	1	0	0	1	1	x	0	1	x	00	010
addi	001000		1	0	1	0	1	1	x	0	0	1	00	000
addiu	001001		1	0	1	0	1	1	x	0	0	1	00	000
andi	001100		1	0	1	0	1	1	x	0	0	0	00	100
ori	001101		1	0	1	0	1	1	x	0	0	0	00	011
xori	001110		1	0	1	0	1	1	x	0	0	0	00	111
lw	100011		1	0	1	1	1	1	1	0	0	1	00	000
sw	101011		1	0	1	x	0	1	x	1	x	1	00	000
beq	000100		1	0	0	x	0	1	x	0	x	1 zero 01 or 00	001	001
bne	000101		1	0	0	x	0	1	x	0	x	1 /zero 01 or 00	001	001
slti	001010		1	0	1	0	1	1	x	0	0	1	00	110
blez	000110		1	0	0	x	0	1	x	0	x	1 sign 01 or 00	001	001
j	000010		1	x	x	x	0	1	x	0	x	1	10	x
halt	111111		0	x	x	x	x	x	x	x	x	x	x	x

该表整合了所要实现程序的控制信号真值，以及每一个信号都op码和func码，方便在书写control Unit模块的时候进行查表对照。

3.测试程序段

地址	汇编程序	指令代码					16 进制数代码	
		op(6)	rs(5)	rt(5)	rd(5)/immediate (16)			
0x00000000	addiu \$1,\$0,8	001001	00000	00001	0000000000001000	=	24010008	
0x00000004	ori \$2,\$0,2	001101	00000	00010	0000000000000010	=	34020002	
0x00000008	add \$3,\$2,\$1	000000	00010	00001	0001100000100000	=	00411820	
0x0000000C	sub \$5,\$3,\$2	000000	00011	00010	0010100000100010	=	00622822	
0x00000010	and \$4,\$5,\$2	000000	00101	00010	0010000000100100	=	00A22024	
0x00000014	or \$8,\$4,\$2	000000	00100	00010	0100000000100101	=	00824025	
0x00000018	sll \$8,\$8,1	000000	00000	01000	0100000001000000	=	00084040	
0x0000001C	bne \$8,\$1,-2 (≠,转 18)	000101	01000	00001	1111111111111110	=	1501FFFE	
0x00000020	slti \$6,\$2,4	001010	00010	00110	0000000000000100	=	28460004	
0x00000024	slti \$7,\$6,0	001010	00110	00111	0000000000000000	=	28470000	
0x00000028	addiu \$7,\$7,8	001001	00111	00111	0000000000001000	=	24D70008	
0x0000002C	beq \$7,\$1,-2 (=,转 28)	000100	00111	00001	1111111111111110	=	10D1FFFE	
0x00000030	sw \$2,4(\$1)	101011	00001	00010	0000000000000100	=	AC220008	
0x00000034	lw \$9,4(\$1)	100011	00001	01001	0000000000000100	=	8C290004	
0x00000038	addiu \$10,\$0,-2	001001	00000	01010	1111111111111110	=	240AFFFE	
0x0000003C	addiu \$10,\$10,1	001001	01010	01010	0000000000000001	=	254A0001	
0x00000040	blez \$10,-2(≤0,转 3C)	000110	01010	00000	1111111111111110	=	1940FFFE	
0x00000044	andi \$11,\$2,2	001100	00010	01011	0000000000000010	=	304B0002	
0x00000048	j 0x0000004C	000010	00000	00000	0000000000010011	=	08000013	
0x0000004C	or \$8,\$4,\$2	000000	00100	00010	0100000000100101	=	00824025	

C							
0x00000050	halt	111111	00000	00000	0000000000000000	=	FC000000

指令存储器读取指令的文本（对应代码中Instruction Memory的
Instruction.txt）

Instructions.txt

文件 编辑 查看

00100100 00000001 00000000 00001000
00110100 00000010 00000000 00000010
00000000 01000001 00011000 00100000
00000000 01100010 00101000 00100010
00000000 10100010 00100000 00100100
00000000 10000010 01000000 00100101
00000000 00001000 01000000 01000000
00010101 00000001 11111111 11111110
00101000 01000110 00000000 00000100
00101000 11000111 00000000 00000000
00100100 11100111 00000000 00001000
00010000 11100001 11111111 11111110
10101100 00100010 00000000 00000100
10001100 00101001 00000000 00000100
00100100 00001010 11111111 11111110
00100101 01001010 00000000 00000001
00011001 01000000 11111111 11111110
00110000 01001011 00000000 00000010
00001000 00000000 00000000 00010011
00000000 10000010 01000000 00100101
11111100 00000000 00000000 00000000

行 21, 列 37 | 775 个字符 | 100% | Windows (CRLF) | UTF-8

该文本采用每八个二进制数字为一个数字串的形式存储，对应CPU中每一条指令的地址都是四的倍数这一规律，并且符合对齐原则，便于CPU进行指令的读取和译码。在Instruction memory和Data Memory模块的代码均体现了该存储方式的特点。

第二部分：代码

(一) 单周期CPU部分

(1) PC

模块实现代码：

```
module PC(
```

```

    input CLK,reset,PCWre,
    input[1:0] PCSrc,
    input [31:0] immExt,
    input [31:0] JumpPC,
    output reg [31:0] address,
    output [31:0] nextaddress,
    output [31:0] PC4    //用于储存当前地址+4后的地址
);
    assign PC4 = address+4;
    assign                                     nextaddress=
(PCSrc==2'b01)?address+4+(immExt<<2):((PCSrc==2'b10)?JumpPC:address+4);
    //根据PCSrc的值来确定下一个指令的地址
    always @(negedge CLK) begin//当clock下降沿到来或Reset下降沿到来时，对
地址进行改变或者置零
        if(reset==0) begin //reset=0时，初始化PC为0
            address=0;
        end
        else if(PCWre) begin
            address=nextaddress;
        end
    end
endmodule

```

PC模块仿真:

```

module PC_sim();
    //输入
    reg CLK;
    reg reset;
    reg PCWre;
    reg [1:0] PCSrc;
    reg [31:0] immExt;
    reg [31:0] JumpPC;
    //输出
    wire [31:0] address;
    wire [31:0] nextaddress;
    PC uut(
        .CLK(CLK),
        .reset(reset),
        .PCWre(PCWre),
        .PCSrc(PCSrc),
        .immExt(immExt),
        .nextaddress(nextaddress),
        .address(address),
        .JumpPC(JumpPC)
    );
endmodule

```

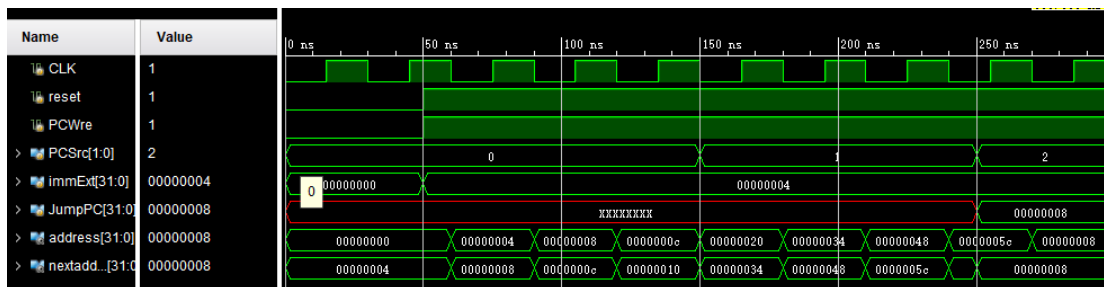
```
always #15 CLK = !CLK; //时钟每15ns变化一次
initial begin
    //记录数据变化
    $dumpfile("PC.vcd");
    $dumpvars(0, PC_sim);
    //初始化
    CLK = 0;
    reset = 0;
    PCWre = 0;
    PCSrc = 2'b00;
    immExt = 0;
    //不跳转，顺序执行下一条地址
    #50;
    reset = 1;
    PCWre = 1;
    PCSrc = 2'b00;
    immExt = 4;
    //不跳转，顺序执行下一条地址
    #50;
    reset = 1;
    PCWre = 1;
    PCSrc = 2'b00;
    immExt = 4;
    //偏移值跳转，执行跳转之后的指令
    #50;
    reset = 1;
    PCWre = 1;
    PCSrc = 2'b01;
    immExt = 4;
    //偏移值跳转，执行跳转之后的指令
    #50;
    reset = 1;
    PCWre = 1;
    PCSrc = 2'b01;
    immExt = 4;
    //无条件跳转，执行跳转之后的命令
    #50;
    reset = 1;
    PCWre = 1;
    PCSrc = 2'b10;
    JumpPC=8;
    //结束
    #50;
    $stop;
```

```

end
endmodule

```

仿真测试截图：



根据仿真测试的波形图，可以看到，PC成功实现了顺序执行、分支跳转以及无条件跳转的操作，对于任何时刻，只要是同一帧，nextaddress的值永远为当前PC的下一个PC。仿真测试很好地证明了PC这一模块设计成功。

(2) Instruction Memory

模块实现代码：

```

module InstructionMemory(
    input [31:0] PC4,
    input [31:0] IAddr,
    input InsMemRW,
    output [5:0] op,
    output [4:0] rs,rt,rd,
    output signed [15:0] immediate,
    output [5:0] func,
    output[4:0] shamt,
    output [31:0] JumpPC,
    output reg [31:0] IDataOut //输出：32位指令，用于计算JumpPC
);
    reg [7:0] Mem[0:127]; //存储器
    assign op=IDataOut[31:26];
    assign rs=IDataOut[25:21];
    assign rt=IDataOut[20:16];
    assign rd=IDataOut[15:11];
    assign immediate=IDataOut[15:0];
    assign func=IDataOut[5:0];
    assign shamt=IDataOut[10:6];
    assign JumpPC={{PC4[31:28]},{IDataOut[25:0]},{2'b00}}; //投机：提前计算
    //好可能存在的跳转指令
    initial begin
        $readmemb("C:/Users/jhinx/Desktop/SingleCycleCPU2/Instructions.txt",Mem);
        //从文件中读取指令
        IDataOut=0;
    end
    always @(IAddr or InsMemRW) begin //InsMemRW信号为1时读取存储器

```

```

        if(InsMemRW==1) begin
            IDataOut[7:0] = Mem[IAddr + 3];
            IDataOut[15:8] = Mem[IAddr + 2];
            IDataOut[23:16] = Mem[IAddr + 1];
            IDataOut[31:24] = Mem[IAddr];
        end
    end
endmodule

```

模块仿真代码

```

module IM_sim;
//输入
reg [31:0] IAddr;
reg InsMemRW;
//输出
wire [5:0] op;
wire [5:0] rs, rt, rd;
wire [15:0] immediate;
wire [5:0] shamt;
wire [5:0] func;
wire [31:0] IDataOut;
InstructionMemory uut(
    .IAddr(IAddr),
    .InsMemRW(InsMemRW),
    .op(op),
    .rs(rs),
    .rt(rt),
    .rd(rd),
    .immediate(immediate),
    .shamt(shamt),
    .func(func),
    .IDataOut(IDataOut)
);

```

```

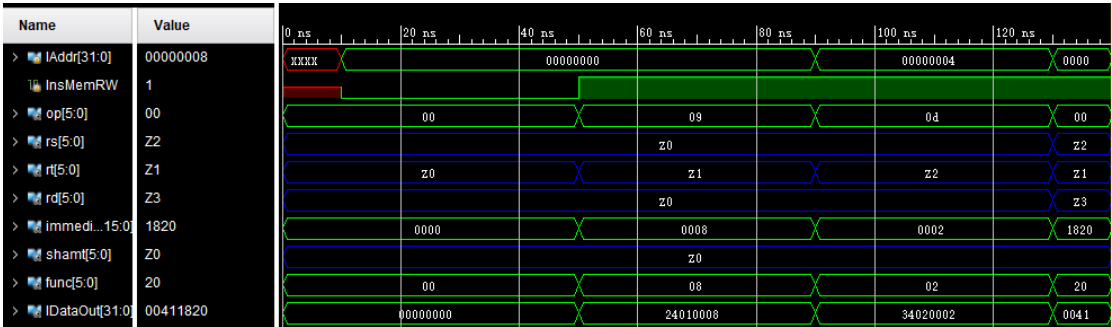
initial begin
    //记录数据的文件名
    $dumpfile("InstructionMemory.vcd");
    $dumpvars(0, IM_sim);
    //初始化
    #10;
    InsMemRW = 0;
    IAddr[31:0] = 32'd0;
    //第一次读指令
    #40;
    InsMemRW = 1;

```



```
IAddr[31:0] = 32'd0;
//第二次读指令
#40;
InsMemRW = 1;
IAddr[31:0] = 32'd4;
//第三次读指令
#40;
InsMemRW = 1;
IAddr[31:0] = 32'd8;
//结束
#10;
$stop;
end
endmodule
```

仿真测试截图



由于InstructionMemory固定在选定的文本文档中读取指令，因此在下方列出仿真测试中涉及的三条指令：

地址	汇编程序	指令代码					16 进制数代码
		op(6)	rs(5)	rt(5)	rd(5)/immediate (16)		
0x00000000	addiu \$1,\$0,8	001001	00000	00001	0000000000001000	=	24010008
0x00000004	ori \$2,\$0,2	001101	00000	00010	0000000000000010	=	34020002
0x00000008	add \$3,\$2,\$1	000000	00010	00001	000110000100000	=	00411820

根据仿真测试的波形图，可以看到，Instruction Memory成功读取了每一条32位指令的相关信息（如op、rs、rt、rd等），并且成功地执行了该指令，IDataOut输出了正确的答案。并且，在InsMemRW为0时，模块无法读取指令，满足该控制信号的设计目的。仿真测试很好地证明了Instruction Memory这一模块设计成功。

(3) Register File

模块实现代码

```
module RegisterFile(
    input CLK,RegDst,RegWre,DBDataSrc,
    input [4:0] rs,rt,rd,
    input [31:0] datafromALU,datafromRW, //来自ALU计算结果的值，来自内存
    output [31:0] Data1,Data2, //从寄存器中读取的值
    output [31:0] writeData //写回寄存器的值
```

```

    );
    wire [4:0] writeReg;    //要写入的寄存器的端口
    assign writeReg=RegDst?rd:rt;    //控制目标地址来源，0为rt(I型指令)，1为
rd(R型指令)
    assign writeData=DBDataSrc?datafromRW:datafromALU;    //控制写回寄
存器的数据来源
    reg [31:0] register[0:31];    //定义32个32位寄存器
    integer i;
    initial begin
        for(i=0;i<32;i=i+1)
            register[i]<=0;
        end    //给每个寄存器赋初值0
    assign Data1=register[rs];
    assign Data2=register[rt];
    always @ (negedge CLK) begin
        if(RegWre)    //RegWre为1时，允许写入寄存器，并且防止寄存器0被修改(第0
个寄存器恒为0)
            register[writeReg] <= writeData;
        end
    endmodule
仿真测试代码
module RF_sim();
    //输入
    reg CLK, RegDst, RegWre, DBDataSrc;
    reg [4:0] rs, rt, rd;
    reg [31:0] datafromALU, datafromRW;
    //输出
    wire [31:0] Data1;
    wire [31:0] Data2;
    RegisterFile uut(
        .CLK(CLK),
        .RegDst(RegDst),
        .RegWre(RegWre),
        .DBDataSrc(DBDataSrc),
        .rs(rs),
        .rt(rt),
        .rd(rd),
        .datafromALU(datafromALU),
        .datafromRW(datafromRW),

        .Data1(Data1),
        .Data2(Data2)
    );

```

```
always #15 CLK = !CLK;
```

```
initial begin
```

```
    //record
```

```
    $dumpfile("RF.vcd");
```

```
    $dumpvars(0, RF_sim);
```

```
    //初始化
```

```
    CLK = 0;
```

```
    //测试样例1
```

```
    #10;
```

```
    CLK = 0;
```

```
    RegDst = 1;//处理R型指令
```

```
    RegWre = 1;//允许写寄存器
```

```
    DBDataSrc = 0;//使用来自存储器的输出
```

```
    rs = 5'b00000;
```

```
    rt = 5'b00001;
```

```
    rd = 5'b00010;
```

```
    datafromALU = 32'd1;//来自ALU的输出
```

```
    datafromRW = 32'd2;//来自RW的输出
```

```
    //测试样例2
```

```
    #10;
```

```
    RegDst = 0;//处理I型指令
```

```
    RegWre = 0;//不允许写寄存器
```

```
    DBDataSrc = 1;//使用来自ALU的输出
```

```
    rs = 5'b00011;
```

```
    rt = 5'b00100;
```

```
    rd = 5'b00101;
```

```
    datafromALU = 32'd3;//来自ALU的输出
```

```
    datafromRW = 32'd4;//来自RW的输出
```

```
    //测试样例3
```

```
    #10;
```

```
    RegDst = 0;//处理I型指令
```

```
    RegWre = 1;//允许写寄存器
```

```
    DBDataSrc = 1;//使用来自ALU的输出
```

```
    rs = 5'b00011;
```

```
    rt = 5'b00100;
```

```
    rd = 5'b00101;
```

```
    datafromALU = 32'd3;//来自ALU的输出
```

```
    datafromRW = 32'd4;//来自RW的输出
```

```
    #10;
```

```
    RegDst = 0;//处理I型指令
```

```
    RegWre = 1;//允许写寄存器
```

```
    DBDataSrc = 1;//使用来自ALU的输出
```

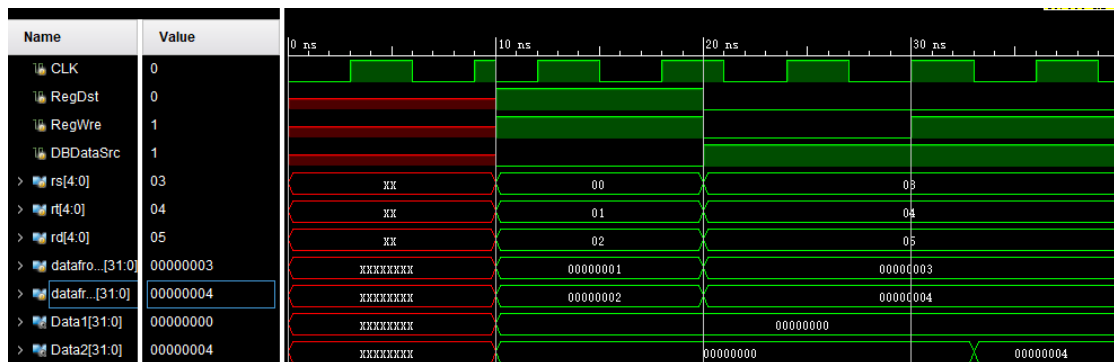
```
    rs = 5'b00011;
```

```

        rt = 5'b00100;
        rd = 5'b00101;
        datafromALU = 32'd3; // 来自ALU的输出
        datafromRW = 32'd4; // 来自RW的输出
        $stop;
    end
endmodule

```

仿真测试结果



根据仿真测试的波形图,可以看到,Register File成功实现了寄存器写入的操作,当RegWre为0时,寄存器堆不允许写入,当为1时才允许写入数据。同时DBDataSrc和RegDst也很好执行了相应的功能。仿真测试很好地证明了Register File这一模块设计成功。

(4) ALU

模块实现代码

```

module ALU(
    input [31:0] ReadData1,
    input [31:0] ReadData2,
    input [31:0] immExt,
    input [4:0] shamt,
    input [2:0] ALUop,
    input ALUSrcA,ALUSrcB,
    output zero,sign,
    output reg [31:0] result
);
    wire [31:0] InA;
    wire [31:0] InB;

    assign InA=ALUSrcA?{27{1'b0}},shamt:ReadData1;    //控制第一个操作数来源,
    0: rs寄存器; 1: 立即数
    assign InB=ALUSrcB?immExt:ReadData2;    //控制第二个操作数来源, 0: rt寄存
    器; 1: 立即数
    assign zero=(result==0)?1:0;
    assign sign=result[31];
    always@(*) begin
        case(ALUop)
            3'b000:
                result=InA+InB;

```

```

        3'b001:
            result=InA-InB;
        3'b010:
            result=InB<<InA;
        3'b011:
            result=InA|InB;    // 或运算
        3'b100:
            result=InA&InB;
        3'b101:
            result=(InA<InB)?1:0;    // 无符号数比较
        3'b110:
            // 符号数比较
result=(((InA<InB)&&(InA[31]==InB[31]))|((InA[31]==1)&&(InB[31]==0)))?1:0;    //
符号数比较
        3'b111:
            result=InA^InB;    // 异或运算
        default:
            result = 32'h0000;
    endcase
end
endmodule

```

仿真测试代码

```

module ALU_sim();

    //input
    reg [31:0] ReadData1;
    reg [31:0] ReadData2;
    reg [31:0] immExt;
    reg [31:0] shamt;
    reg [2:0] ALUOp;
    reg ALUSrcA, ALUSrcB;

    //output
    wire zero;
    wire [31:0] result;

    ALU uut(
        .ReadData1(ReadData1),
        .ReadData2(ReadData2),
        .immExt(immExt),
        .shamt(shamt),
        .ALUOp(ALUOp),
        .ALUSrcA(ALUSrcA),
        .ALUSrcB(ALUSrcB),

```

```
.zero(zero),  
.result(result)  
);  
  
initial begin  
    //record  
    $dumpfile("ALU32.vcd");  
    $dumpvars(0, ALU_sim);  
    //add1  
    ReadData1 = 0;  
    ReadData2 = 0;  
    immExt = 1;  
    shamt = 1;  
    ALUop = 3'b000;  
    ALUSrcA = 0;  
    ALUSrcB = 0;  
    //add2  
    #50;  
    ReadData1 = 0;  
    ReadData2 = 0;  
    immExt = 1;  
    shamt = 1;  
    ALUop = 3'b000;  
    ALUSrcA = 1;  
    ALUSrcB = 0;  
    //sub1  
    #50;  
    ReadData1 = 1;  
    ReadData2 = 2;  
    immExt = 3;  
    shamt = 4;  
    ALUop = 3'b001;  
    ALUSrcA = 0;  
    ALUSrcB = 0;  
    //sub2  
    #50;  
    ReadData1 = 1;  
    ReadData2 = 2;  
    immExt = 3;  
    shamt = 4;  
    ALUop = 3'b001;  
    ALUSrcA = 1;  
    ALUSrcB = 0;  
    //左移1
```

```
#50;
ReadData1 = 1;
ReadData2 = 2;
immExt = 2;
shamt = 4;
ALUOp = 3'b010;
ALUSrcA = 0;
ALUSrcB = 0;
//左移2
#50;
ReadData1 = 1;
ReadData2 = 2;
immExt = 2;
shamt = 4;
ALUOp = 3'b010;
ALUSrcA = 1;
ALUSrcB = 0;
//or1
#50;
ReadData1 = 1;
ReadData2 = 2;
immExt = 2;
shamt = 4;
ALUOp = 3'b011;
ALUSrcA = 0;
ALUSrcB = 0;
//or2
#50;
ReadData1 = 1;
ReadData2 = 2;
immExt = 2;
shamt = 4;
ALUOp = 3'b011;
ALUSrcA = 1;
ALUSrcB = 0;
//and1
#50;
ReadData1 = 1;
ReadData2 = 2;
immExt = 2;
shamt = 4;
ALUOp = 3'b100;
ALUSrcA = 0;
ALUSrcB = 0;
```

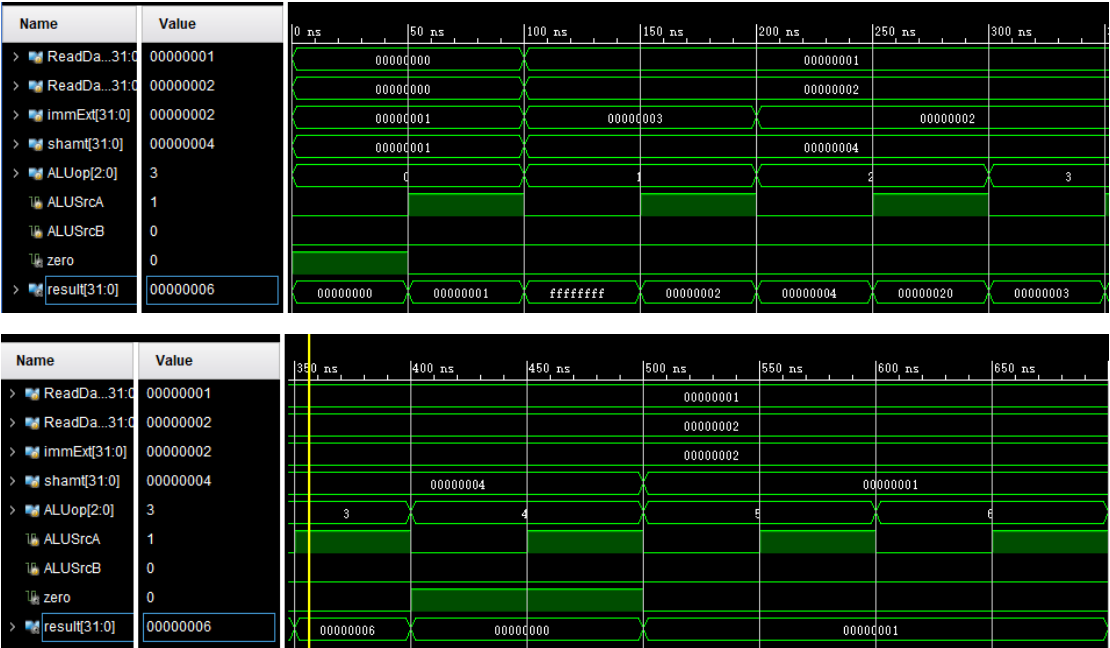
```
//and2
#50;
ReadData1 = 1;
ReadData2 = 2;
immExt = 2;
shamt = 4;
ALUOp = 3'b100;
ALUSrcA = 1;
ALUSrcB = 0;
//不带符号比较1
#50;
ReadData1 = 1;
ReadData2 = 2;
immExt = 2;
shamt = 1;
ALUOp = 3'b101;
ALUSrcA = 0;
ALUSrcB = 0;
//不带符号比较2
#50;
ReadData1 = 1;
ReadData2 = 2;
immExt = 2;
shamt = 1;
ALUOp = 3'b101;
ALUSrcA = 1;
ALUSrcB = 0;
//带符号比较1
#50;
ReadData1 = 1;
ReadData2 = 2;
immExt = 2;
shamt = 1;
ALUOp = 3'b110;
ALUSrcA = 0;
ALUSrcB = 0;
//带符号比较2
#50;
ReadData1 = 1;
ReadData2 = 2;
immExt = 2;
shamt = 1;
ALUOp = 3'b110;
ALUSrcA = 1;
```

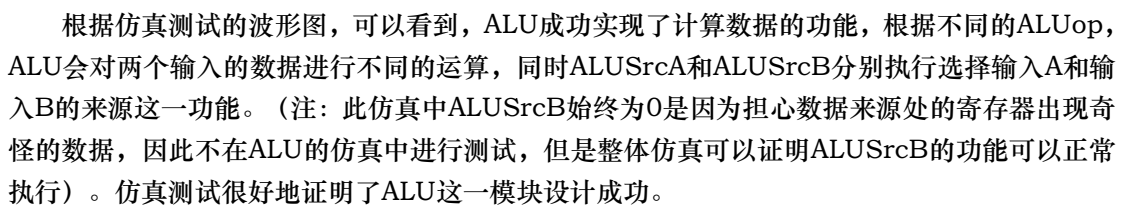


```
ALUSrcB = 0;
//nor1
#50;
ReadData1 = 1;
ReadData2 = 2;
immExt = 2;
shamt = 1;
ALUOp = 3'b111;
ALUSrcA = 0;
ALUSrcB = 0;
//nor2
#50;
ReadData1 = 1;
ReadData2 = 2;
immExt = 2;
shamt = 1;
ALUOp = 3'b111;
ALUSrcA = 1;
ALUSrcB = 0;
//stop
#50;
$stop;

end
endmodule
```

仿真测试结果





模块实现代码

仿真测试代码

```
reg CLK;
```

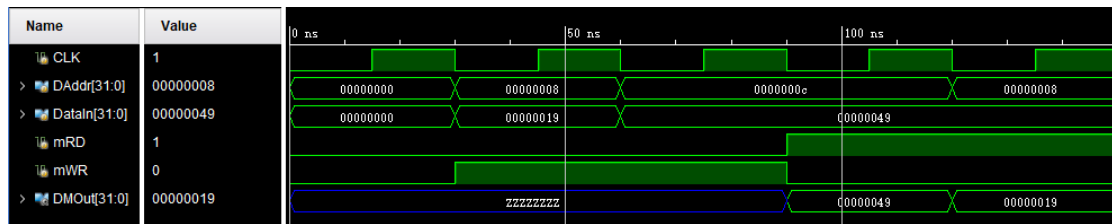
```
reg [31:0] DAddr;
reg [31:0] DataIn;
reg mRD;
reg mWR;
//输出
wire [31:0] DMOut;
DataMemory uut(
    .CLK(CLK),
    .DAddr(DAddr),
    .DataIn(DataIn),
    .mRD(mRD),
    .mWR(mWR),
    .DMOut(DMOut)
);
always #15 CLK = !CLK;
initial begin
    //
    $dumpfile("DM.vcd");
    $dumpvars(0, DM_sim);
    //初始化
    CLK = 0;
    DAddr = 0;
    DataIn = 0;
    mRD = 0;    //为1，正常读；为0，输出高阻态（相当于开路）
    mWR = 0;    //为1，写；为0，无操作
    #30;//30ns后，CLK下降沿写
    DAddr = 8;
    DataIn = 25;
    mRD = 0;
    mWR = 1;
    #30;//60ns后，CLK下降沿写
    DAddr = 12;
    DataIn = 73;
    mRD = 0;
    mWR = 1;
    #30;//90ns后开始读
    DAddr = 12;
    mRD = 1;
    mWR = 0;
    #30;//120ns后开始读
    DAddr = 8;
    mRD = 1;
    mWR = 0;
    #30;
```

```

        $stop;//150ns后停
    end
endmodule

```

仿真测试结果



根据仿真测试的波形图，可以看到，Data Memory成功实现了数据存储器的读写操作，当mRD为0时，不允许进行数据读取操作，因此DMOut一直为代码所定义的高阻态(ZZZZZZZZ)；于此同时，mWR为1，DataMemory将来自DataIn的数据写入存储器中。当mWR为1时，数据存储器允许读取数据，DMOut输出相应数据存储器中的数据。仿真测试很好地证明了DataMemory这一模块设计成功。

(6) SignZeroExtend

模块实现代码

```

module SignZeroExtend(
    input wire [15:0] immediate,
    input ExtSel,
    output wire [31:0] immExt
);
    assign immExt[15:0] = immediate[15:0];
    assign immExt[31:16] = (ExtSel == 1) ? {16{immediate[15]}} : 16'b0;
    //如果ExtSel信号为1，则进行符号扩展，否则进行零扩展
endmodule

```

仿真测试代码

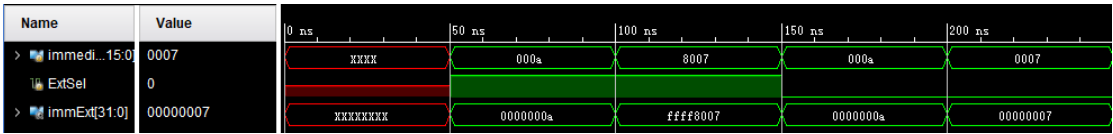
```

module SZE_sim();
    //输入
    reg signed [15:0] immediate;
    reg ExtSel;
    //输出
    wire [31:0] immExt;
    SignZeroExtend uut(
        .immediate(immediate),
        .ExtSel(ExtSel),
        .immExt(immExt)
    );
    initial begin
        //记录数据
        $dumpfile("SZE.vcd");
        $dumpvars(0, SZE_sim);
        //Test1
    end
endmodule

```

```
#50;
ExtSel = 1;
immediate[15:0] = 15'd10;
//Test2
#50;
ExtSel = 1;
immediate[15:0] = 15'd7;
immediate[15]=1;
//Test3
#50;
ExtSel = 0;
immediate[15:0] = 15'd10;
//Test4
#50;
ExtSel = 0;
immediate[15:0] = 15'd7;
#50;
$stop;
end
endmodule
```

仿真测试结果



根据仿真测试的波形图，可以看到，SignZeroExtend成功实现了立即数拓展的操作。当ExtSel=0时，对立即数进行零扩展，当ExtSel=1时，对立即数进行符号扩展。注意：在这一模块的仿真程序中，由于我们的立即数都是按照十进制数来进行赋值的，所以对于立即数的正负，我们需要手动定义十六位立即数的最高位（具体参照仿真代码），经测试，只有手动赋值了才可以正确地进行符号拓展。仿真测试很好地证明了SignZeroExtend这一模块设计成功。

(7) controlUnit

模块实现代码（该方法为使用case语句进行赋值）

```
module ControlUnit(
    input [5:0] op,
    input zero,sign,
    input [5:0] func,
    output reg PCWre,
    output reg ALUSrcA,
    output reg ALUSrcB,
    output reg DBDataSrc,
    output reg RegWre,
    output reg InsMemRW,
    output reg mRD,
```

```

    output reg mWR,
    output reg RegDst,
    output reg ExtSel,
    output reg [1:0]PCSrc,
    output reg [2:0]ALUOp
);
    always@(*) begin
        case(op)
            6'b000000: begin
                case(func)
                    6'b100000: begin //add
                        PCWre=1;ALUSrcA=0;ALUSrcB=0;DBDataSrc=0;RegWre=1;InsMemRW=1;mRD=0;m
                        WR=0;RegDst=1;ExtSel=1;PCSrc=2'b00;ALUOp=3'b000;
                    end
                    6'b100001: begin //addu
                        PCWre=1;ALUSrcA=0;ALUSrcB=0;DBDataSrc=0;RegWre=1;InsMemRW=1;mRD=0;m
                        WR=0;RegDst=1;ExtSel=1;PCSrc=2'b00;ALUOp=3'b000;
                    end
                    6'b100010: begin //sub
                        PCWre=1;ALUSrcA=0;ALUSrcB=0;DBDataSrc=0;RegWre=1;InsMemRW=1;mRD=0;m
                        WR=0;RegDst=1;ExtSel=1;PCSrc=2'b00;ALUOp=3'b001;
                    end
                    6'b100011: begin //subu
                        PCWre=1;ALUSrcA=0;ALUSrcB=0;DBDataSrc=0;RegWre=1;InsMemRW=1;mRD=0;m
                        WR=0;RegDst=1;ExtSel=1;PCSrc=2'b00;ALUOp=3'b001;
                    end
                    6'b100100: begin //and
                        PCWre=1;ALUSrcA=0;ALUSrcB=0;DBDataSrc=0;RegWre=1;InsMemRW=1;mRD=0;m
                        WR=0;RegDst=1;ExtSel=1;PCSrc=2'b00;ALUOp=3'b100;
                    end
                    6'b100101: begin //or
                        PCWre=1;ALUSrcA=0;ALUSrcB=0;DBDataSrc=0;RegWre=1;InsMemRW=1;mRD=0;m
                        WR=0;RegDst=1;ExtSel=1;PCSrc=2'b00;ALUOp=3'b011;
                    end
                    6'b100110: begin //xor
                        PCWre=1;ALUSrcA=0;ALUSrcB=0;DBDataSrc=0;RegWre=1;InsMemRW=1;mRD=0;m
                        WR=0;RegDst=1;ExtSel=1;PCSrc=2'b00;ALUOp=3'b111;

```

```

        end
        6'b101010: begin    //slt
        PCWre=1;ALUSrcA=0;ALUSrcB=0;DBDataSrc=0;RegWre=1;InsMemRW=1;mRD=0;m
WR=0;RegDst=1;ExtSel=1;PCSrc=2'b00;ALUOp=4'b110;
        end
        6'b101011: begin    //sltu
        PCWre=1;ALUSrcA=0;ALUSrcB=0;DBDataSrc=0;RegWre=1;InsMemRW=1;mRD=0;m
WR=0;RegDst=1;ExtSel=1;PCSrc=2'b00;ALUOp=4'b101;
        end
        6'b000000: begin    //sll
        PCWre=1;ALUSrcA=1;ALUSrcB=0;DBDataSrc=0;RegWre=1;InsMemRW=1;mRD=0;m
WR=0;RegDst=1;ExtSel=1;PCSrc=2'b00;ALUOp=3'b010;
        end
        endcase
        end
        6'b001000: begin    //addi
        PCWre=1;ALUSrcA=0;ALUSrcB=1;DBDataSrc=0;RegWre=1;InsMemRW=1;mRD=0;m
WR=0;RegDst=0;ExtSel=1;PCSrc=2'b00;ALUOp=3'b000;
        end
        6'b001001: begin    //addiu
        PCWre=1;ALUSrcA=0;ALUSrcB=1;DBDataSrc=0;RegWre=1;InsMemRW=1;mRD=0;m
WR=0;RegDst=0;ExtSel=1;PCSrc=2'b00;ALUOp=3'b000;
        end
        6'b001100: begin    //andi
        PCWre=1;ALUSrcA=0;ALUSrcB=1;DBDataSrc=0;RegWre=1;InsMemRW=1;mRD=0;m
WR=0;RegDst=0;ExtSel=0;PCSrc=2'b00;ALUOp=3'b100;
        end
        6'b001101: begin    //ori
        PCWre=1;ALUSrcA=0;ALUSrcB=1;DBDataSrc=0;RegWre=1;InsMemRW=1;mRD=0;m
WR=0;RegDst=0;ExtSel=0;PCSrc=2'b00;ALUOp=3'b011;
        end
        6'b001110: begin    //xori
        PCWre=1;ALUSrcA=0;ALUSrcB=1;DBDataSrc=0;RegWre=1;InsMemRW=1;mRD=0;m
WR=0;RegDst=0;ExtSel=0;PCSrc=2'b00;ALUOp=3'b111;
        end

```

```

        6'b100011: begin    //lw
PCWre=1;ALUSrcA=0;ALUSrcB=1;DBDataSrc=1;RegWre=1;InsMemRW=1;mRD=1;m
WR=0;RegDst=0;ExtSel=1;PCSrc=2'b00;ALUOp=3'b000;
        end
        6'b101011: begin    //sw
PCWre=1;ALUSrcA=0;ALUSrcB=1;DBDataSrc=0;RegWre=0;InsMemRW=1;mRD=0;m
WR=1;RegDst=0;ExtSel=1;PCSrc=2'b00;ALUOp=3'b000;
        end
        6'b000100: begin    //beq
PCWre=1;ALUSrcA=0;ALUSrcB=0;DBDataSrc=0;RegWre=0;InsMemRW=1;mRD=0;m
WR=0;RegDst=0;ExtSel=1;ALUOp=3'b001;
        PCSrc=(zero==1)?(2'b01):(2'b00);
        end
        6'b000101: begin    //bne
PCWre=1;ALUSrcA=0;ALUSrcB=0;DBDataSrc=0;RegWre=0;InsMemRW=1;mRD=0;m
WR=0;RegDst=0;ExtSel=1;ALUOp=3'b001;
        PCSrc=(zero==0)?(2'b01):(2'b00);
        end
        6'b001010: begin    //slti
PCWre=1;ALUSrcA=0;ALUSrcB=1;DBDataSrc=0;RegWre=1;InsMemRW=1;mRD=0;m
WR=0;RegDst=0;ExtSel=1;PCSrc=2'b00;ALUOp=3'b110;
        end
        6'b000110: begin    //blez
PCWre=1;ALUSrcA=0;ALUSrcB=0;DBDataSrc=0;RegWre=0;InsMemRW=1;mRD=0;m
WR=0;RegDst=0;ExtSel=1;ALUOp=3'b001;
        PCSrc=(zero==0)?2'b01:((sign==1)?2'b01:2'b00);
        end
        6'b000010: begin    //j
PCWre=1;ALUSrcA=0;ALUSrcB=1;DBDataSrc=0;RegWre=0;InsMemRW=1;mRD=0;m
WR=0;RegDst=0;ExtSel=1;PCSrc=2'b10;ALUOp=3'b000;
        end
        6'b111111: begin    //halt
PCWre=0;ALUSrcA=0;ALUSrcB=1;DBDataSrc=0;RegWre=0;InsMemRW=1;mRD=0;m
WR=0;RegDst=0;ExtSel=1;PCSrc=2'b00;ALUOp=3'b000;
        end
    endcase

```



```

        end
    endmodule

```

该模块代码的另一种实现方式（使用assign进行赋值）

```

    input [5:0] op,
    input zero,sign,
    input [5:0] func,
    output PCWre,
    output ALUSrcA,
    output ALUSrcB,
    output DBDataSrc,
    output RegWre,
    output InsMemRW,
    output mRD,
    output mWR,
    output RegDst,
    output ExtSel,
    output [1:0] PCSrc,
    output [2:0] ALUOp

    parameter R=6'b000000;
    assign PCWre=(op==6'b111111)?0:1;
    assign ALUSrcA=(op==R&&func==6'b000000)?1:0;
    assign
ALUSrcB=(op==R | op==6'b000100 | op==6'b000101 | op==6'b000110)?0:1;
    assign DBDataSrc=(op==6'b100011)?1:0;
    assign
RegWre=(op==6'b101011 | op==6'b000100 | op==6'b000101 | op==6'b000110 | op=
=6'b000010)?0:1;
    assign InsMemRW=1;
    assign mRD=(op==6'b100011)?1:0;
    assign mWR=(op==6'b101011)?1:0;
    assign RegDst=(op==R)?1:0;
    assign
ExtSel=(op==R | op==6'b001100 | op==6'b001101 | op==6'b001110)?0:1;
    assign PCSrc=(op==6'b000010)?2'b10
                :(op==6'b000100)?(zero==1)?2'b01:2'b00
                :(op==6'b000101)?(zero==1)?2'b00:2'b01
                :(op==6'b000110)?(sign==1)?2'b01:2'b00
                :2'b00;
    assign
ALUOp=((op==R&&func==6'b100010) | (op==R&&func==6'b100011) | op==6'b00010
0 | op==6'b000101 | op==6'b000110)?3'b001
                :(op==R&&func==6'b000000)?3'b010
                :((op==R&&func==6'b100101) | op==6'b001101)?3'b011

```

```

:((op==R&&func==6'b100100) || op==6'b001100)?3'b100
:((op==R&&func==6'b101011)?3'b101
:((op==R&&func==6'b101010) || op==6'b001010)?3'b110
:((op==R&&func==6'b100110) || op==6'b001110)?3'b111
:3'b000;

```

(8) SingleCycleCPU (顶层综合模块)

模块实现代码

```

module SingleCycleCPU(
    input CLK,reset,
    output wire [31:0]
curPC,nextPC,IDataOut,DataOut1,DataOut2,result,writeData
);
    wire [2:0] ALUop;
    wire [31:0] immExt;
    wire [5:0] op,func;
    wire [15:0] immediate;
    wire [4:0] rs,rt,rd;
    wire [4:0] shamt;
    wire [31:0] JumpPC;
    wire [31:0] DMOOut;
    wire [31:0] PC4;
    wire [1:0] PCSrc;
    wire
zero,sign,PCWre,ALUSrcA,ALUSrcB,RegWre,InsMemRW,mRD,mWR,RegDst,DBData
Src,ExtSel;
    ALU
alu(DataOut1,DataOut2,immExt,shamt,ALUop,ALUSrcA,ALUSrcB,zero,sign,result);
    PC pc(CLK,reset,PCWre,PCSrc,immExt,JumpPC,curPC,nextPC,PC4);
    ControlUnit
CU(op,zero,sign,func,PCWre,ALUSrcA,ALUSrcB,DBDataSrc,RegWre,InsMemRW,mR
D,mWR,RegDst,ExtSel,PCSrc,ALUop);
    DataMemory DM(CLK,result,DataOut2,mRD,mWR,DMOOut);
    InstructionMemory
IM(PC4,curPC,InsMemRW,op,rs,rt,rd,immediate,func,shamt,JumpPC,IDataOut);
    RegisterFile
RF(CLK,RegDst,RegWre,DBDataSrc,rs,rt,rd,result,DMOOut,DataOut1,DataOut2,write
Data);
    SignZeroExtend SZE(immediate,ExtSel,immExt);
Endmodule

```

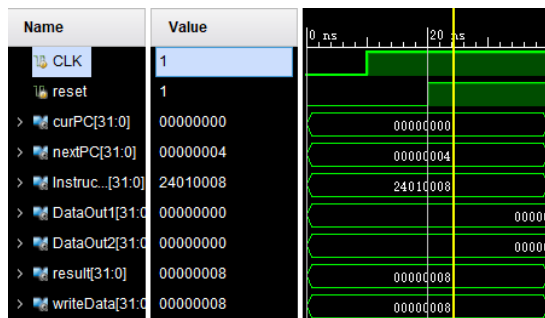
(顶层模块output定义的顺序决定了仿真的波形出现顺序，以及烧板时数据的出现顺序，这一点非常重要!!! 需要注意)

(子模块的wire定义也要和子模块代码中各个input和output的定义顺序相一致，否则就会出现错误)

仿真测试代码

```
module SingleCycleCPU_sim();
    reg CLK;
    reg reset;
    //输出
    wire [31:0] curPC;
    wire [31:0] nextPC;
    wire [31:0] Instruction;
    wire [31:0] DataOut1;
    wire [31:0] DataOut2;
    wire [31:0] result;
    wire [31:0] writeData;
    //
    SingleCycleCPU uut(
        .CLK(CLK),
        .reset(reset),
        .curPC(curPC),
        .nextPC(nextPC),
        .IDataOut(Instruction),
        .DataOut1(DataOut1),
        .DataOut2(DataOut2),
        .result(result),
        .writeData(writeData)
    );
    initial begin
        //记录数据
        $dumpfile("SCCPU.vcd");
        $dumpvars(0, SingleCycleCPU_sim);
        //初始化输入
        CLK = 0;
        reset = 0;//刚开始设置PC为0
        #10;
        CLK = 1;
        #10;
        reset = 1;
        //产生时钟信号
        forever #20 begin
            CLK = !CLK;
        end
    end
endmodule
```

仿真测试结果



指令: addiu \$1,\$0,8

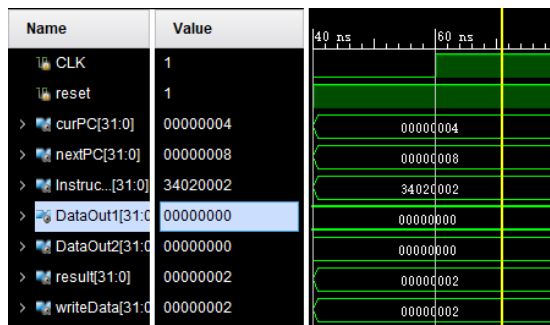
00100100000000010000000000001000=24010008;

此时的 PC=0, 下一地址为顺序执行, 等于 4;

此时\$1 和\$0 的值都是初始值 0, 因此 DataOut1 和 DataOut2 都等于 0;

计算得到的结果为 $0+8=8$, 因此 result=8;

结果写回\$1 中, 数值为 8, 因此 WriteData=8.



指令: ori \$2,\$0,2

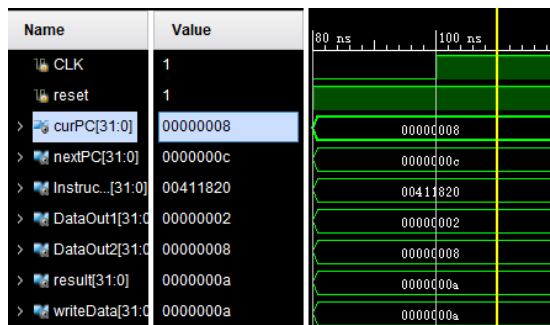
0011010000000010000000000000010=34020002;

此时的 PC=4, 下一地址为顺序执行, 等于 8;

此时\$2 和\$0 的值都是初始值 0, 因此 DataOut1 和 DataOut2 都等于 0;

计算得到的结果为 $0|2=2$, 因此 result=2;

结果写回\$2 中, 数值为 2, 因此 WriteData=2.



指令: add \$3,\$2,\$1

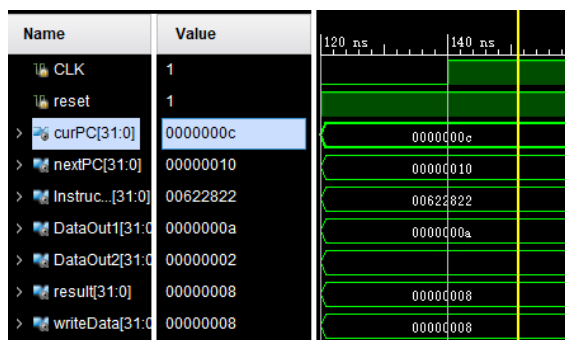
00000000010000010001100000100000=00411820;

此时的 PC=8, 下一地址为顺序执行, 等于 12(c);

此时\$2=2, \$1=8, 因此 DataOut1=2, DataOut2=8;

计算得到的结果为 $2+8=10$, 因此 result=10(a);

结果写回\$3 中, 数值为 10(a), 因此 WriteData=10(a).



指令: sub \$5,\$3,\$2

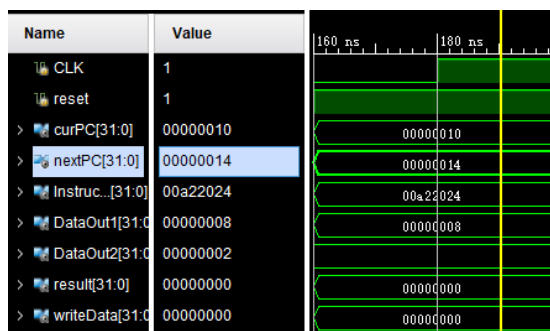
00000000011000100010100000100010=00622822;

此时的 PC=12, 下一地址为顺序执行, 等于 16(10);

此时 \$3=10, \$2=2, 因此 DataOut1=10(a), DataOut2=2;

计算得到的结果为 $10-2=8$, 因此 result=8;

结果写回\$5 中, 数值为 8, 因此 WriteData=8.



指令: and \$4,\$5,\$2

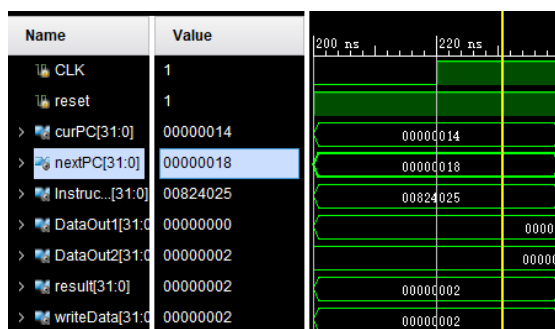
00000000101000100010000000100100=00a22024;

此时的 PC=16(10), 下一地址为顺序执行, 等于 20(14);

此时\$5=8, \$2=2, 因此 DataOut1=8, DataOut2=2;

计算得到的结果为 $8\&2=0$, 因此 result=0;

结果写回\$4 中, 数值为 0, 因此 WriteData=0.



指令: or \$8,\$4,\$2

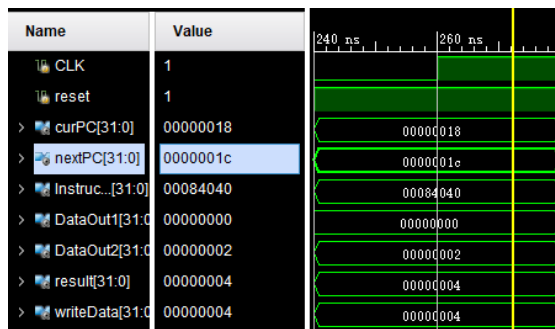
00000000100000100100000000100101=00824025;

此时的 PC=20(14), 下一地址为顺序执行, 等于 24(18);

此时\$4=0, \$2=2, 因此 DataOut1=0, DataOut2=2;

计算得到的结果为 0|2=2, 因此 result=2;

结果写回\$8中, 数值为 2, 因此 WriteData=2.



指令: sll \$8,\$8,1

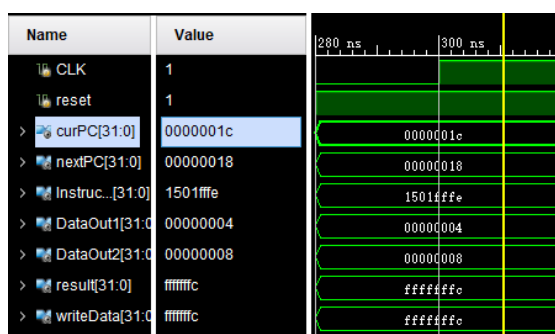
00000000000010000100000000100000=00084040;

此时的 PC=24(18), 下一地址为顺序执行, 等于 28(1c);

因为 sll 指令的 rs 部分为 00000, \$8=2, 因此 DataOut1=0, DataOut2=2;

计算得到的结果为 $2 \ll 1 = 4$, 因此 result=4;

结果写回\$8中, 数值为 4, 因此 WriteData=4.



指令: bne \$8,\$1,-2

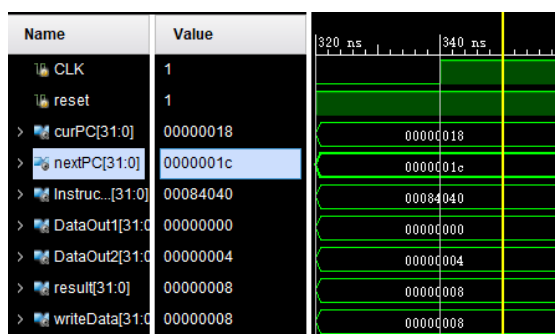
00010101000000001111111111111110=1501FFFE;

此时的 PC=28(1c), 如果发生跳转的话, 下一指令的地址为 $PC+4+(-2)*4=PC-4$, 即 24(18);

此时\$8=4, \$1=8, 因此 DataOut1=4, DataOut2=8;

因为 $4 \neq 8$, 因此执行条件跳转;

因为 $4-8$ 的结果为 -4, 因此 result 和 writeData 均等于 -4(fffffffc), 但该数据不会被写回。



指令: sll \$8,\$8,1

00000000000010000100000000100000=00084040;

此时的 PC=24(18), 下一地址为顺序执行, 等于 28(1c);

因为 sll 指令的 rs 部分为 00000, \$8=4, 因此 DataOut1=0, DataOut2=4;

计算得到的结果为 $4 \ll 1 = 8$, 因此 result=8;

结果写回\$8中, 数值为 8, 因此 WriteData=8.

指令: bne \$8,\$1,-2

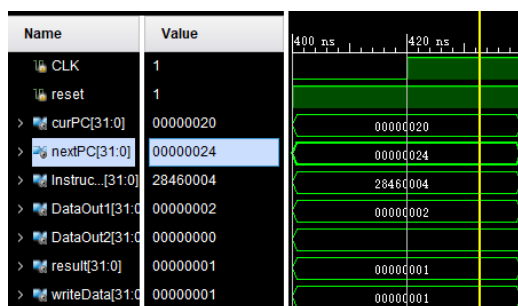
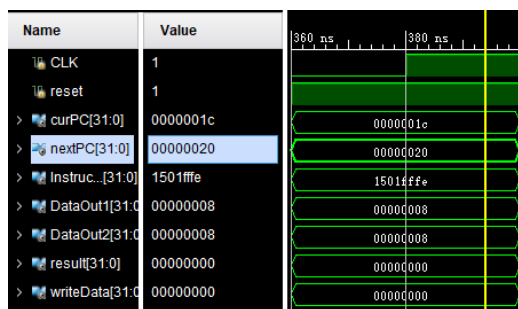
00010101000000001111111111111110=1501FFFE;

此时的 PC=28(1c), 如果发生跳转的话, 下一指令的地址为 $PC+4+(-2)*4=PC-4$, 即 24(18);

此时\$8=8, \$1=8, 因此 DataOut1=8, DataOut2=8;

因为 $8=8$, 因此不执行条件跳转; 下一条指令的地址为顺序跳转, 即 32(20);

因为 $8-8$ 的结果为 0, 因此 result 和 writeData 均等于 0, 但该数据不会被写回。

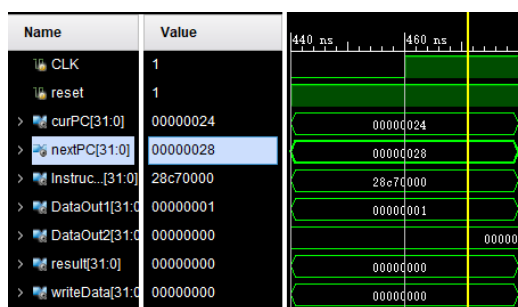


指令: slti \$6,\$2,4

0010100001100011100000000000000100=28460004;

此时的 PC=32(20), 下一地址为顺序执行, 等于 36(24);

此时\$2=2, \$6=0, 因此 DataOut1=2, DataOut2=0; 因为 2 小于立即数 4, 因此\$6 会被赋值为 1, 因此 result= WriteData=1.

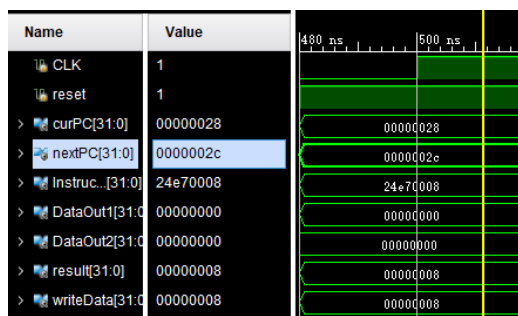


指令: slti \$7,\$6,0

00101000111000111000000000000000=28470000;

此时的 PC=36(24), 下一地址为顺序执行, 等于 40(28);

此时\$6=1, \$7=0, 因此 DataOut1=1, DataOut2=0; 因为 1 大于立即数 0, 因此\$7 会被赋值为 0, 因此 result= WriteData=0.

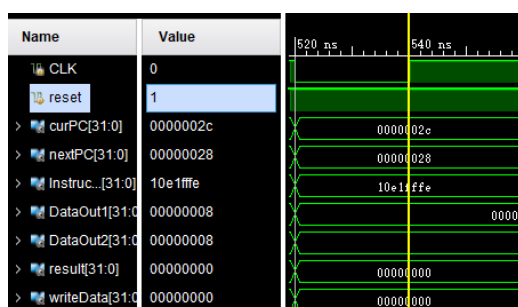


指令: addiu \$7,\$7,8

001001001110011110000000000001000=24D70008;

此时的 PC=40(28), 下一地址为顺序执行, 等于 44(2c);

此时\$7 的值为 0, 且 rs 和 rt 所指向的寄存器都是\$7, 因此 DataOut1 和 DataOut2 都等于 0; 计算得到的结果为 0+8=8, 因此 result=8; 结果写回\$7 中, 数值为 8, 因此 WriteData=8.



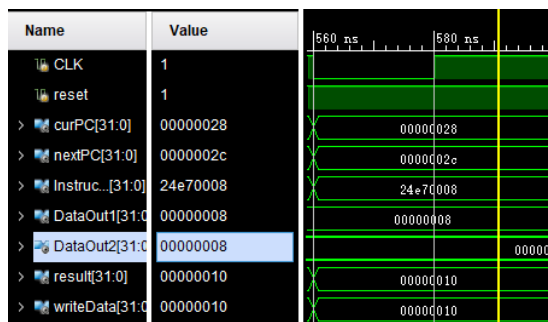
指令: beq \$7,\$1,-2

0001000011100001111111111111110=10D1FFFE;

此时的 PC=44(2c), 如果发生跳转的话, 下一指令的地址为 PC+4+(-2)*4=PC-4, 即 40(28);

此时\$7=8, \$1=8, 因此 DataOut1=8, DataOut2=8; 因为 8=8, 因此执行条件跳转;

因为 8-8 的结果为 0, 因此 result 和 writeData 均等于 0, 但该数据不会被写回。



指令: addiu \$7,\$7,8

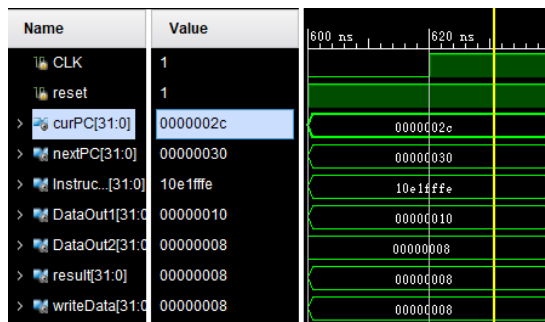
00100100111001110000000000001000=24D70008;

此时的 PC=40(28), 下一地址为顺序执行, 等于 44(2c);

此时\$7 的值为 8, 且 rs 和 rt 所指向的寄存器都是\$7, 因此 DataOut1 和 DataOut2 都等于 8;

计算得到的结果为 $8+8=16$, 因此 result=16(10);

结果写回\$7 中, 数值为 16, 因此 WriteData=16(10).



指令: beq \$7,\$1,-2

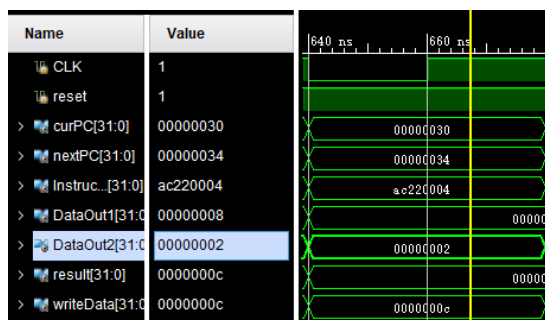
000100001110000111111111111110=10D1FFFFE;

此时的 PC=44(2c), 如果发生跳转的话, 下一指令的地址为 $PC+4+(-2)*4=PC-4$, 即 40(28);

此时 \$7=16, \$1=8, 因此 DataOut1=16(10),

DataOut2=8; 因为 $16 \neq 8$, 因此不执行条件跳转; 下一条指令的地址为顺序跳转, 即 48(30);

因为 $16-8$ 的结果为 8, 因此 result 和 writeData 均等于 0, 但该数据不会被写回。

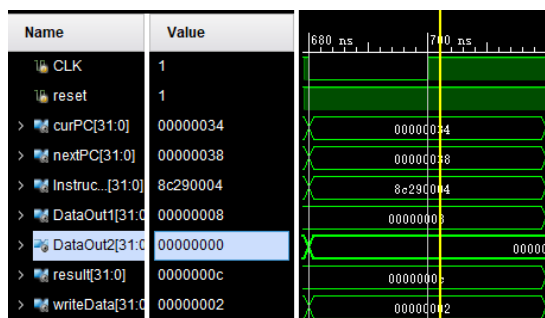


指令: sw \$2,4(\$1)

1010110000100010000000000000100=AC220008;

此时的 PC=48(30), 下一地址为顺序执行, 等于 52(34);

此时\$1 的值为 8, \$2 的值为 2, 因此 DataOut1=8, DataOut2=2; 该指令执行后, 会把\$2 的值存到地址为 $\$1 + \text{偏移值}$ 处(该指令为 +4)的位置, 因此 result=writeData= $8+4=12$ (c);

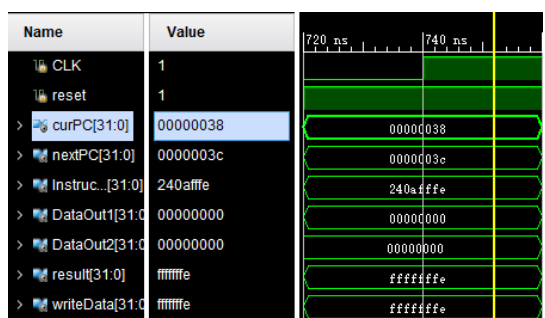


指令: lw \$9,4(\$1)

1000110000101001000000000000100=8C290004;

此时的 PC=52(34), 下一地址为顺序执行, 等于 56(38);

此时\$1 的值为 8, \$9 的值为 0, 因此 DataOut1=8, DataOut2=0; 该指令执行后, 会把地址为 $\$1 + \text{偏移值}$ 处(该指令为 +4)的值存到\$9 中, 因此 result= $8+4=12$ (c); 而 writeData=2, 即地址为 $\$1+4$ 处所存储的值。

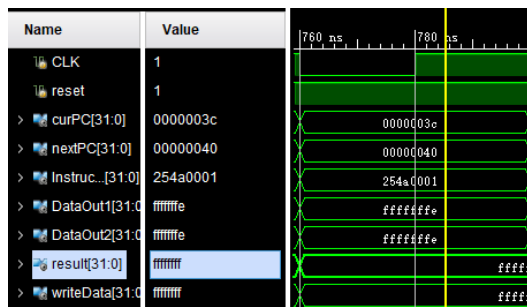


指令: addiu \$10,\$0,-2

001001000000101011111111111110=240AFFFE;

此时的 PC=56(38), 下一地址为顺序执行, 等于 60(3c);

此时\$10 和\$0 的值都是初始值 0, 因此 DataOut1 和 DataOut2 都等于 0; 计算得到的结果为 $0-2=-2$, 因此 result=-2(fffffffe); 结果写回\$10 中, 数值为-2, 因此 WriteData=-2(fffffffe).

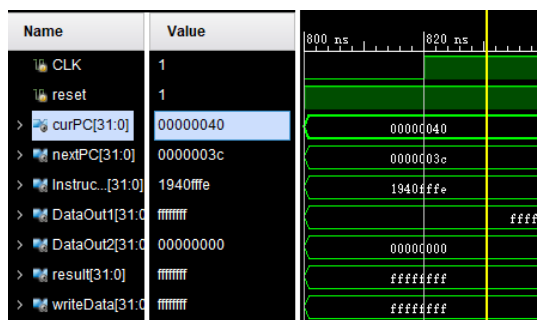


指令: addiu \$10,\$10,1

00100101010010100000000000000001=254A0001;

此时的 PC=60(3c), 下一地址为顺序执行, 等于 64(40);

此时\$10 的值为-2, 因此 DataOut1 和 DataOut2 都等于 -2(fffffffe); 计算得到的结果为 -2+1=-1, 因此 result=-1(ffffff); 结果写回\$10 中, 数值为-1, 因此 WriteData=-1(ffffff).



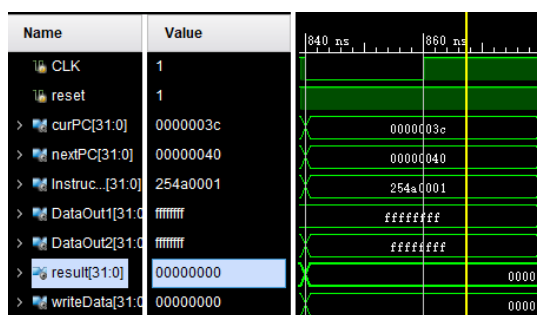
指令: blez \$10,-2

0001100101000000111111111111110=1940FFFE;

此时的 PC=64(40), 如果发生跳转的话, 下一指令的地址为 $PC+4+(-2)*4=PC-4$, 即 60(3c);

此时\$10=-1, 而 blez 指令默认 rt 位置的值为 0, 因此 DataOut1=-1(ffffff), DataOut2=0; 因为 $-1 \leq 0$, 因此执行条件跳转;

因为 $-1+0$ (rt 位置为 00000) 的结果为 -1, 因此 result 和 writeData 均等于 -1(ffffff), 但该数据不会被写回。

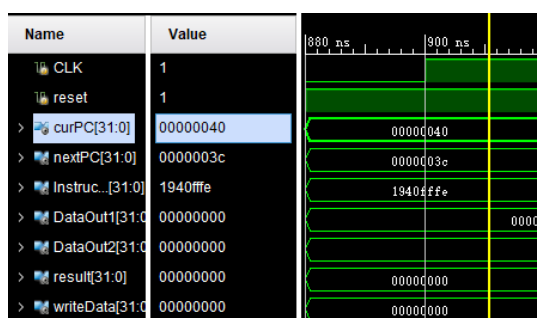


指令: addiu \$10,\$10,1

00100101010010100000000000000001=254A0001;

此时的 PC=60(3c), 下一地址为顺序执行, 等于 64(40);

此时\$10 的值为-1, 因此 DataOut1 和 DataOut2 都等于 -1(ffffff); 计算得到的结果为 $-1+1=0$, 因此 result=0; 结果写回\$10 中, 数值为 0, 因此 WriteData=0.

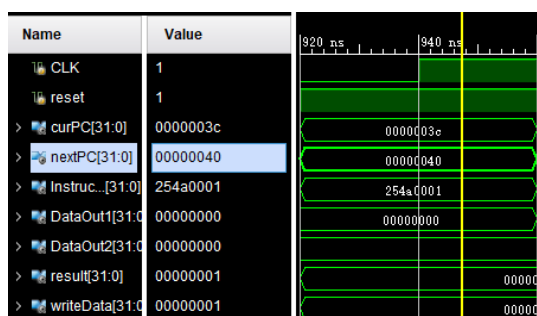


指令: blez \$10,-2

0001100101000000111111111111110=1940FFFE;

此时的 PC=64(40), 如果发生跳转的话, 下一指令的地址为 $PC+4+(-2)*4=PC-4$, 即 60(3c);

此时\$10=0, 而 blez 指令默认 rt 位置的值为 0, 因此 DataOut1=0, DataOut2=0; 因为 $0 \leq 0$, 因此执行条件跳转; 因为 $0+0$ (rt 位置为 00000) 的结果为 0, 因此 result 和 writeData 均等于 0, 但该数据不会被写回。



指令: addiu \$10,\$10,1

00100101010010100000000000000001=254A0001;

此时的 PC=60(3c), 下一地址为顺序执行, 等于 64(40);

此时\$10 的值为 0, 因此 DataOut1 和 DataOut2 都等于 0; 计算得到的结果为 $0+1=1$, 因此 result=1; 结果写回\$10 中, 数值为 1, 因此 WriteData=1.

Name	Value
CLK	1
reset	1
> curPC[31:0]	00000040
> nextPC[31:0]	00000044
> Instruc_[31:0]	1940ffe
> DataOut1[31:0]	00000001
> DataOut2[31:0]	00000000
> result[31:0]	00000001
> writeData[31:0]	00000001

Timing diagram showing signals over time. The diagram has two time markers: 960 ns and 980 ns. The signals shown are CLK, reset, curPC[31:0], nextPC[31:0], Instruc_[31:0], DataOut1[31:0], DataOut2[31:0], result[31:0], and writeData[31:0]. The signals are represented by horizontal lines with values indicated at specific time points.

指令: blez \$10,-2

000110010100000011111111111110=1940FFFE;

此时的 PC=64(40)，如果发生跳转的话，下一指令的地址为 $PC+4+(-2)*4=PC-4$ ，即 60(3c)；此时 \$10=1，因此 DataOut1=1，DataOut2=0；因为 $1 \leq 0$ 不成立，因此不执行条件跳转，转而实行顺序跳转，下一指令地址为 $PC+4=68(44)$ ；因为 $1+0$ (rt 位置为 00000)的结果为 1，因此 result 和 writeData 均等于 1，但该数据不会被写回。

Name	Value
CLK	1
reset	1
> curPC[31:0]	00000044
> nextPC[31:0]	00000048
> Instruc_[31:0]	304b0002
> DataOut1[31:0]	00000002
> DataOut2[31:0]	00000000
> result[31:0]	00000002
> writeData[31:0]	00000002

指令: `andi $11,$2,2`

```
00110000010010110000000000000010=304B0002;
```

此时的 PC=68(44)，下一地址为顺序执行，等于 72(48)；此时 \$2 的值为 2，\$11 的值为 0，因此 DataOut1=2，DataOut2=0；

计算得到的结果为 $2 \& 2 = 2$ ，因此 `result=2`;

结果写回\$11 中，数值为 2，因此 WriteData=2.

Name	Value	
CLK	1	
reset	1	
> curPC[31:0]	00000048	
nextPC[31:0]	0000004c	
> Instruc_[31:0]	08000013	
DataOut1[31:0]	00000000	
DataOut2[31:0]	00000000	
result[31:0]	00000013	
writeData[31:0]	00000013	

指令: j 0x0000004C

$0000100000000000000000000000000010011=08000013;$

此时的 PC=72(48)，下一地址为无条件跳转，等于二进制的(10011)*4=18*4=76(4c)；此时 rs 和 rt 所处的指令字段数值均为 0，因此 DataOut1=DataOut2=0；因为指令的二进制数(10011)等于十六进制数(13)，因此 result=writeData=十六进制数 (13)；

Name	Value
CLK	1
reset	1
curPC[31:0]	0000004c
nextPC[31:0]	00000050
Instruc_[31:0]	00824025
DataOut1[31:0]	00000000
DataOut2[31:0]	00000002
result[31:0]	00000002
writeData[31:0]	00000002

指令: or \$8,\$4,\$2

```
00000000100000100100000000100101=00824025;
```

此时的 PC=76(4c)，下一地址为顺序执行，等于 80(50)；

此时\$4=0，\$2=2，因此 DataOut1=0，DataOut2=2；

计算得到的结果为 $0|2=2$ ，因此 `result=2`;

结果写回\$8 中，数值为 2，因此 WriteData=2.

Name	Value
CLK	0
reset	1
curPC[31:0]	00000050
nextPC[31:0]	00000054
Instruc...[31:0]	fc000000
DataOut1[31:0]	00000000
DataOut2[31:0]	00000000
result[31:0]	00000000
writeData[31:0]	00000000

指令: halt(伪指令)

1111100000000000000000

```
00000000=FC000000;
```

此时的 PC=80(50)，下一地址为顺序执行，等于 84(54)；

但是因为是停机指令，PC 不会更新，所有且的数据均被赋值为 0，因此波形会一直停留在这一个状态，且 DataOut1=DataOut2=result=writeData=0。

(二) basys3板子部分

(1) Display_7seg (七段数码管)

模块实现代码

```

module Display_7seg(
    input [3:0] display_data,
    output reg [7:0] dispcode
);
    always @(display_data) begin
        case (display_data)
            4'b0000:dispcode=8'b1100_0000;//0
            4'b0001:dispcode=8'b1111_1001;//1
            4'b0010:dispcode=8'b1010_0100;//2
            4'b0011:dispcode=8'b1011_0000;//3
            4'b0100:dispcode=8'b1001_1001;//4
            4'b0101:dispcode=8'b1001_0010;//5
            4'b0110:dispcode=8'b1000_0010;//6
            4'b0111:dispcode=8'b1101_1000;//7
            4'b1000:dispcode=8'b1000_0000;//8
            4'b1001:dispcode=8'b1001_0000;//9
            4'b1010:dispcode=8'b1000_1000;//A
            4'b1011:dispcode=8'b1000_0011;//b
            4'b1100:dispcode=8'b1100_0110;//C
            4'b1101:dispcode=8'b1010_0001;//d
            4'b1110:dispcode=8'b1000_0110;//E
            4'b1111:dispcode=8'b1000_1110;//F
            default:dispcode=8'b0000_0000;
        endcase
    end
endmodule

```

(2) clk_div (时钟分频模块)

模块实现代码

```

module clk_div(
    input CLK,
    output reg[3:0]AN
);
    parameter T1MS=100000; //表示一毫秒的时钟周期
    reg [21:0] counter;
    initial begin
        counter<=0;
        AN<=4'b0111;//默认选通第一个数码管
    end
    always@(posedge CLK) begin
        counter<=counter+1;
        if (counter==T1MS) begin

```

```

        counter<=0;
        case (AN)
            4'b0111:AN<=4'b1110;
            4'b1110:AN<=4'b1101;
            4'b1101:AN<=4'b1011;
            4'b1011:AN<=4'b0111;
            4'b0000:AN<=4'b0111;
        endcase
    end
end
endmodule

```

(3) Debounce (消抖模块)

模块实现代码

```

module Debounce(
    input clk,
    input key_in,
    output key_out
);
    parameter SAMPLE_TIME=5000;
    reg [15:0] count_low;    // 计数器，记录处于低电平的时间
    reg [15:0] count_high;
    reg key_out_reg;        // 存储消抖后的按键状态

    always@(posedge clk) begin
        count_low<= key_in?0:count_low+1;
        count_high<= key_in?count_high+1:0;
        if(count_low==SAMPLE_TIME)
            key_out_reg<=0;    // 按键稳定于低电平已经一段时间了
        else if(count_high==SAMPLE_TIME)
            key_out_reg<=1;
    end
    assign key_out=!key_out_reg;    // 通过反转来输出稳定的按键信号
endmodule

```

(4) Basys3顶层综合模块

模块实现代码

```

module Basys3(
    input CLK,
    input [1:0]SW,    // 开关信号，用于选择显示不同的内容
    input Reset,
    input button,
    output [3:0]AN,
    output [7:0]display
);
    wire [31:0]PC;

```

```

    wire [31:0]newaddress;
    wire [31:0]Instruction;
    wire [31:0]Reg_S;
    wire [31:0]Reg_T;
    wire [31:0]ALUOUT;
    wire [31:0]W_data;
    wire myCLK;
    reg [3:0]data;
    Debounce debounce(CLK,button,myCLK);
    SingleCycleCPU
SingleCycleCPU(myCLK,Reset,PC,newaddress,Instruction,Reg_S,Reg_T,ALUOUT,W_
data);
    Display_7seg Display_7seg(data,display);
    clk_div clk_div(CLK,AN);
    always@(myCLK)begin
        case(AN)
            4'b1110:    begin
                case(SW)
                    2'b00:data<=newaddress[3:0];
                    2'b01:data<=Reg_S[3:0];
                    2'b10:data<=Reg_T[3:0];
                    2'b11:data<=W_data[3:0];
                endcase
            end
            4'b1101:    begin
                case(SW)
                    2'b00:data<=newaddress[7:4];
                    2'b01:data<=Reg_S[7:4];
                    2'b10:data<=Reg_T[7:4];
                    2'b11:data<=W_data[7:4];
                endcase
            end
            4'b1011:    begin
                case(SW)
                    2'b00:data<=PC[3:0];
                    2'b01:data<=Instruction[24:21];
                    2'b10:data<=Instruction[19:16];
                    2'b11:data<=ALUOUT[3:0];
                endcase
            end
            4'b0111 : begin
                case(SW)
                    2'b00:data<=PC[7:4];
                    2'b01:data<={3'b000,Instruction[25]};

```

```

                2'b10:data<={3'b000,Instruction[20]};
                2'b11:data<=ALUOUT[7:4];
            endcase
        end
    endcase
end
endmodule

```

（三）引脚分配文件

自动生成的代码如下：

```

set_property IOSTANDARD LVCMOS33 [get_ports {AN[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {AN[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {AN[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {AN[0]}]
set_property PACKAGE_PIN U2 [get_ports {AN[0]}]
set_property PACKAGE_PIN U4 [get_ports {AN[1]}]
set_property PACKAGE_PIN V4 [get_ports {AN[2]}]
set_property PACKAGE_PIN W4 [get_ports {AN[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SW[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SW[0]}]
set_property PACKAGE_PIN R2 [get_ports {SW[1]}]
set_property PACKAGE_PIN T1 [get_ports {SW[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports button]
set_property PACKAGE_PIN T17 [get_ports button]
set_property IOSTANDARD LVCMOS33 [get_ports CLK]
set_property PACKAGE_PIN W5 [get_ports CLK]
set_property IOSTANDARD LVCMOS33 [get_ports Reset]
set_property PACKAGE_PIN V17 [get_ports Reset]

set_property IOSTANDARD LVCMOS33 [get_ports {display[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {display[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {display[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {display[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {display[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {display[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {display[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {display[0]}]
set_property PACKAGE_PIN V7 [get_ports {display[7]}]
set_property PACKAGE_PIN U7 [get_ports {display[6]}]
set_property PACKAGE_PIN V5 [get_ports {display[5]}]
set_property PACKAGE_PIN U5 [get_ports {display[4]}]
set_property PACKAGE_PIN V8 [get_ports {display[3]}]
set_property PACKAGE_PIN U8 [get_ports {display[2]}]
set_property PACKAGE_PIN W6 [get_ports {display[1]}]
set_property PACKAGE_PIN W7 [get_ports {display[0]}]

```

第三部分：烧板结果

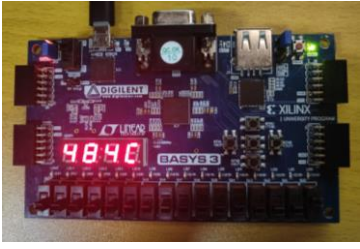
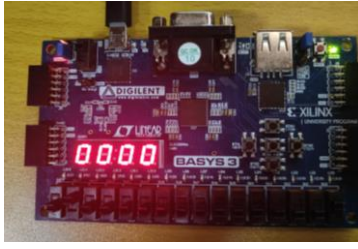

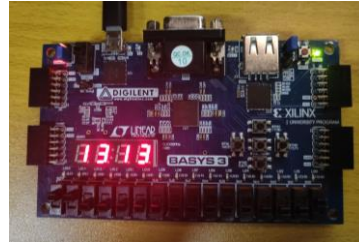
由于篇幅问题，而且在仿真波形中已详细列出了每一条指令的情况，因此仅展示部分指令的烧板情况。

下面以第一条指令作为烧板结果的展示样例：

1. addiu \$1,\$0,8			
当前PC：下条PC	rs地址：rs数据	rt地址：rt数据	ALU结果：DB数据
			
2.add \$3,\$2,\$1			
当前PC：下条PC	rs地址：rs数据	rt地址：rt数据	ALU结果：DB数据
			
3.bne \$8,\$1,-2			
当前PC：下条PC	rs地址：rs数据	rt地址：rt数据	ALU结果：DB数据
			

4.slti \$6,\$2,4			
当前PC: 下条PC	rs地址: rs数据	rt地址: rt数据	ALU结果: DB数据
			
5.sw \$2,4(\$1)			
当前PC: 下条PC	rs地址: rs数据	rt地址: rt数据	ALU结果: DB数据
			
6.lw \$9,4(\$1)			
当前PC: 下条PC	rs地址: rs数据	rt地址: rt数据	ALU结果: DB数据
			
7.blez \$10,-2			
当前PC: 下条PC	rs地址: rs数据	rt地址: rt数据	ALU结果: DB数据
			

8.j 0x0000004C

当前PC：下条PC	rs地址：rs数据	rt地址：rt数据	ALU结果：DB数据
			

经过检验和验收，包括上述展示的烧板结果在内，所有指令的烧板结果均正确，单周期CPU设计成功。

二.实验心得

1.本实验借鉴资料如下：

<https://github.com/Hide-on-bush2/CPU/blob/master/%E5%8D%95%E5%91%A8%E6%9C%9FCPU%E5%AE%9E%E9%AA%8C%E6%8A%A5%E5%91%8A.md>

<https://starashzero.github.io/Co-homework/singleCPU.html>

<https://wu-kan.cn/2018/11/23/%E5%8D%95%E5%91%A8%E6%9C%9FCPU%E8%AE%BE%E8%AE%A1/>

首先感谢老师所提供的三位学长的实验报告，他们作为“前人”，在代码方面提供了许多帮助。但是同时应指出：三位学长的代码均出现了不同程度的错误和问题。例如对于JumpPC的32位二进制数构成，以及clk_div（时钟分频）模块中相关变量的赋值出现错误等。

2.本人实验的创新点如下：

- （1）本人在老师所要求的12条指令的基础下，额外增添了10条指令的实现，特别是几位学长所没有实现的异或运算。但由于时间关系，本人并未对这十条指令进行验证，只在ALU模块的仿真测试中增加了xor指令的测试。
- （2）本人在书写控制单元的时候，除了使用几位学长和课堂PPT所给出的使用assign语法对控制信号进行赋值这一方法，还提供了使用case语句对控制信号进行赋值的方法，并且在仿真测试和烧板验证时，本人使用的是利用case语句的代码，经验证，代码可以正常且正确地运行。

(3) 在学长的基础上,本人完善了大部分模块的仿真测试,使得仿真测试更严谨完善,例如PC模块的仿真测试,本人添加了关于无条件跳转的测试样例,等等。

3.本人在实验过程中遇到的困难如下:

(1) 对于vivado代码语法的不熟悉,如果没有几位学长的实验报告作为参考,仅凭自己、不借鉴任何资料,独立完成单周期CPU的设计是十分困难的。

(2) vivado自身环境的问题,在设计过程中,难免会在一开始设计的不必要或者错误的模块,但是仅仅依靠vivado自身自带的删除功能,似乎不能将错误的模块删除干净,最终会导致整个模块始终不能进行仿真或者综合(会一直报错)。目前已知只有两个解决方法:新建一个project,只把正确的代码复制到新的project内;或者去翻project所在的文件夹,存储代码相关的文件,把日志清空,重新进行仿真或综合。

本人一开始并不知道vivado的环境如此“垃圾”,一直以为是自己代码的问题,在debug上白白浪费了五个小时的时间,最后新建project后发现原封不动的代码跑通了,过程实在是容易令人红温。

4.实验心得:

(1) 实验前期准备非常重要,如果一开始就立刻书写代码,不仅对代码一头雾水,而且可能根本不知道自己在做什么。建议在书写代码之前,先列出代码的指令格式表和控制信号真值表,并且先借助数据通路图确定每条指令的控制信号真值,先理解了每条指令是怎么运行的。这样不仅可以提高代码的书写速度,并且理解每一条代码是怎么运行之后,更容易发现自己仿真的错误。

(2) 如果写完模块实现代码后,直接进行烧板,大概率会出现各种各样的错误,建议推广“对每一个模块进行仿真”这一操作,不仅可以细化问题,还方便进行检验和代码优化。

(3) 由于本校并未,可能也没有时间介绍Verilog语言的相关语法知识,因此在必要的时候,可以借助gpt,让gpt为自己讲解代码的作用,不仅可以更好地理解代码,还有助于发现代码的逻辑错误。