《计算机组成原理与接口技术实验》

实验报告

学 院 名 称 ：	数据科学与计算机学院
学 生 姓 名 ：	罗剑杰
学　　号 ：	15331229
专业（班级）：	15 软件工程三（6）班
时　　间 ：	2017　年　4　月　10　日

成 绩 ：

实验二 ：单周期CPU设计

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。需设计的指令与格式如下：

==> 算术运算指令

（1）**add rd , rs, rt** （**说明：以助记符表示，是汇编指令；以代码表示，是机器指令**）

000000	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd←rs + rt。**reserved 为预留部分，即未用，一般填“0”。**

（2）addi rt , rs ,**immediate**

000001	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：rt←rs + (sign-extend)**immediate；immediate** 符号扩展再参加“加”运算。

（3）sub rd , rs , rt

000010	rs(5 位)	rt(5 位)	rd(5 位)	reserved

完成功能：rd←rs - rt

==> 逻辑运算指令

（4）ori rt , rs ,**immediate**

010000	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：rt←rs | (zero-extend)**immediate；immediate** 做“0”扩展再参加“或”运算。

（5）and rd , rs , rt

010001	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd←rs & rt；逻辑与运算。

（6）or rd , rs , rt

010010	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd←rs | rt；逻辑或运算。

==>移位指令

（7）sll rd, rt,sa

011000	未用	rt(5 位)	rd(5 位)	sa	reserved

功能：rd<－rt<<(zero-extend)sa，左移 sa 位 ，(zero-extend)sa

==> 存储器读/写指令

（8）sw　rt ,**immediate**(rs) 写存储器

100110	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：memory[rs+ (sign-extend)**immediate**<<2]←rt；**immediate** 符号扩展乘以 4 再相加。

（9) lw　rt , **immediate**(rs) 读存储器

100111	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：rt ← memory[rs + (sign-extend)**immediate**<<2]；**immediate** 符号扩展乘以 4 再相加。

==> 分支指令

（10）beq　rs,rt,**immediate**

110000	rs(5 位)	rt(5 位)	**immediate**(位移量，16 位)>>2

功能：if(rs=rt) pc←pc + 4 + (sign-extend)**immediate** <<2　else pc ←pc + 4

特别说明：**immediate 是从 PC+4 地址开始和转移到的指令之间指令条数。**immediate 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 immediate 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”。

注：原文档 sw 和 lw 有错误，拿到扩展后的立即数后应该要先向左移两位（乘以 4）之后再送入 ALU 里面，上面已经补充改正

==> 停机指令

（11）halt

111111	00000000000000000000000000(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。

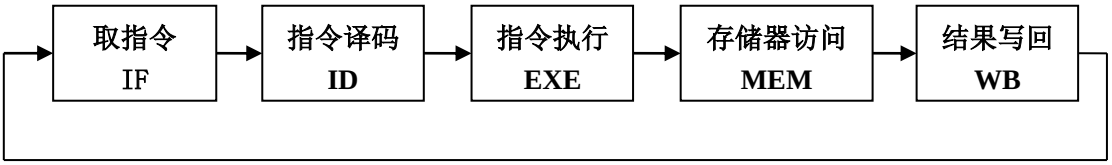
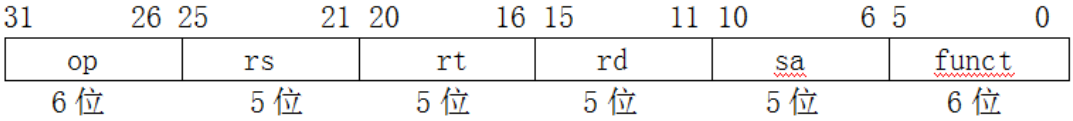


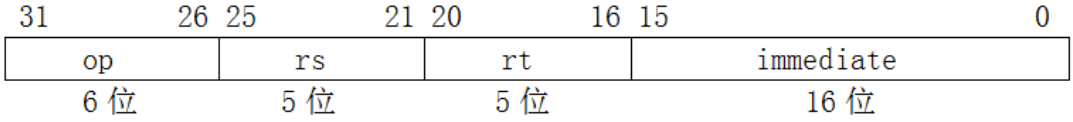
图 1　单周期 CPU 指令处理过程

MIPS 指令的三种格式:

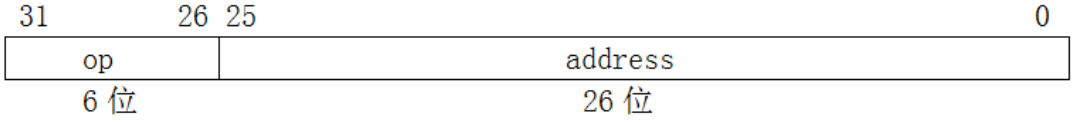
R 类型:



I 类型:



J 类型:



其中，

op：为操作码；

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

rt：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：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；

func：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；

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（Laod）/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。

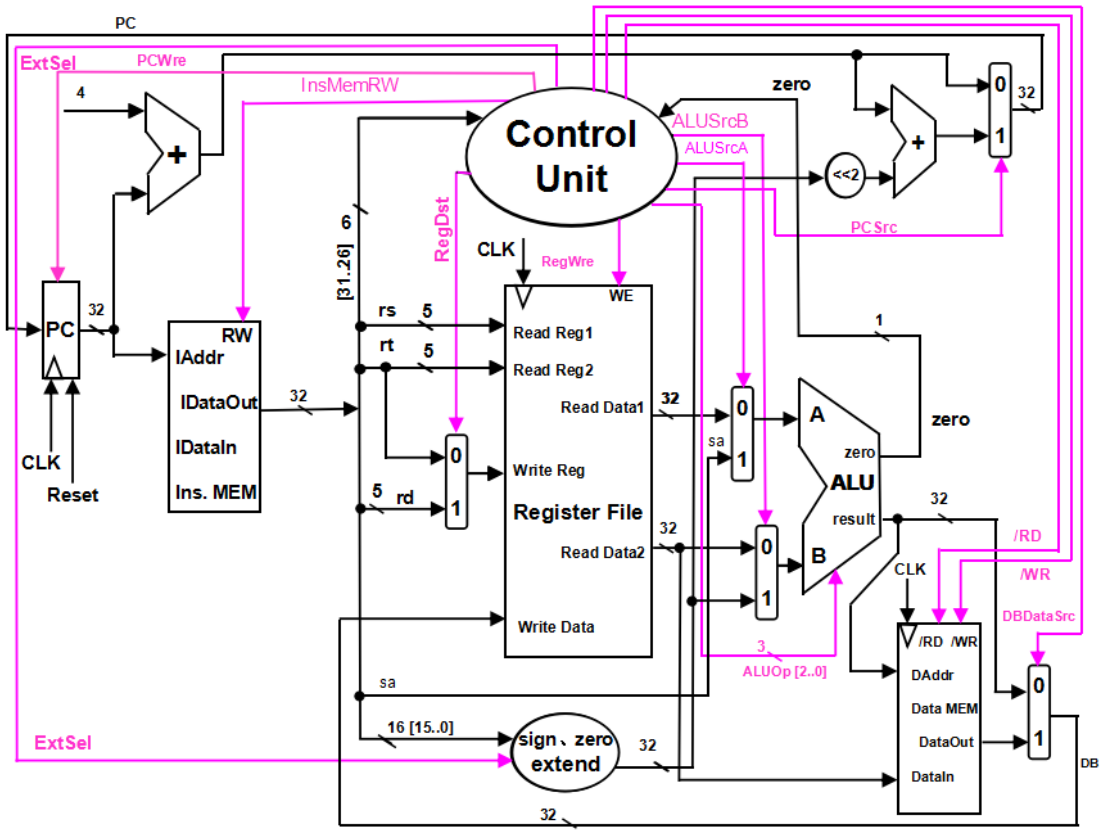


图 2　单周期 CPU 数据通路和控制线路图

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，先给出地址，然后由读或写信号控制操作。对于寄存器组，读操作时，先给出地址，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，在时钟边沿触发写入。图中有一个移位器没有体现出来，应该在extend出来到ALUB输入口前的二选一器前插入，当指令为sw或lw的时候需要把立即数左移两位，使能信号恰好可以使用DBDataSrc。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
ALUSrcB	来自寄存器堆 data2 输出，相关指令：addu、subu、or、and、bne	来自 sign 或 zero 扩展的立即数，相关指令：addiu、ori、sw、lw
DBDataSrc	来自 ALU 运算结果的输出，相关指令：add、addi、sub、ori、or、and、sll	来自数据存储器（Data MEM）的输出，相关指令：lw
RegWre	无写寄存器组寄存器，相关指令：beq、sw、halt	寄存器组写使能，相关指令：add、addi、sub、ori、or、and、sll、lw

InsMemRW	写指令存储器	读指令存储器(Ins. Data)
/RD	读数据存储器，相关指令：lw	输出高阻态
/WR	写数据存储器，相关指令：sw	无操作
ExtSel	(zero-extend)**immediate**（0 扩展）相关指令：ori，	(sign-extend)**immediate**（符号扩展）相关指令：addi、sw、lw、beq，
RegDst	写寄存器组寄存器的地址，来自 rt 字段，相关指令：addi、ori、lw	写寄存器组寄存器的地址，来自 rd 字段，相关指令：add、sub、and、or、sll
PCSrc	PC←PC+4，相关指令：add、addi、sub、ori、or、and、sw、sll、lw、beq(zero=0)	PC←PC+4+(sign-extend)**immediate**，相关指令：beq（zero=1）
ALUOp[2..0]	ALU 8 种运算功能选择(000-111)，看功能表	

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= (A<B) ?1: 0	比较 A 与 B 不带符号
011	Y = B<<A	B 左移 A 位
100	Y = A ∨ B	或
101	Y = A ∧ B	与
110	Y = A ⊕ B	异或
111	Y = A ⊙ B	同或

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，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开始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。另外，值得注意的问题，设计时，用模块化的思想方法设计，关于 ALU 设计、存储器设计、寄存器组设计等等，也是必须认真考虑的问题。

四. 实验器材

电脑一台、Xilinx Vivado 软件一套。

五. 实验分析与设计

1. 实验总体过程：

A 学习verilog相关语法

B 运行跑马灯学习Vivado的操作使用

C 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，再根据关系表可以写出各控制信号的逻辑表达式

D 参考老师提供的模块的代码，按照C步骤得出的相互关系修改各个模块

E 完成顶层模块，设计测试模块进行测试调试

F 调试完成，书写实验报告

2.实验过程C中得出的各种表格。

表三 控制信号与指令之间的关系表

	ExtSel	PCWre	InsMemRW	RegDst	RegWre	ALUSrcA	ALUSrcB
add	0	1	0	1	1	0	0
addi	1	1	0	0	1	0	1
sub	0	1	0	1	1	0	0
ori	0	1	0	0	1	0	1
and	0	1	0	1	1	0	0
or	0	1	0	1	1	0	0
sll	0	1	0	1	1	1	0
sw	1	1	0	0	0	0	1
lw	1	1	0	0	1	0	1
beq(z=1)	1	1	0	0	0	0	1
beq(z=0)	1	1	0	0	0	0	1
halt	0	0	0	0	0	0	0

表四（续表三）控制信号与指令之间的关系表

	PCSrc	ALUOp[2:0]	RD	WR	DBDataSrc	zero
add	0	000	0	0	0	0
addi	0	000	0	0	0	0
sub	0	001	0	0	0	0
ori	0	100	0	0	0	0
and	0	101	0	0	0	0

or	0	100	0	0	0	0
sll	0	011	0	0	0	0
sw	0	000	1	0	1	0
lw	0	000	0	1	1	0
beq(z=1)	1	001	0	0	x	1
beq(z=0)	0	001	0	0	x	0
halt	0	xxx	0	0	x	0

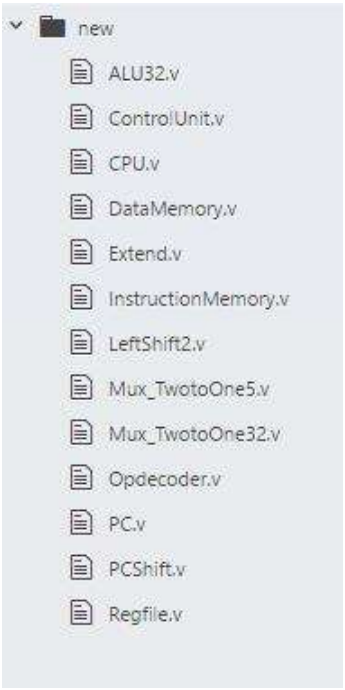
表五 测试程序设计

地址	汇编程序	指令代码				
		op(6)	rs(5)	rt(5)	rd(5)(+sa(5))/immediate(16)	寄存器应有的结果
0x00000000	addi $1, $0, 6	000001	00000	00001	0000 0000 0000 0110	$1=0x00000006
0x00000004	ori $2,$0,15	010000	00000	00010	0000 0000 0000 1111	$2=0x0000000F
0x00000008	sw $2, 2($0)	100110	00000	00010	0000 0000 0000 0010	
0x0000000C	lw $3, 2($0)	100111	00000	00011	0000 0000 0000 0010	$3=0x0000000F
0x00000010	sll $4, $1, 1	011000	XXXXX	00001	00100(rd) 00001(sa)xxxxxx	$4=0x0000000C
0x00000014	sub $5, $2, $1	000010	00010	00001	00101 xxxxxxxxxxx	$5=0x00000009
0x00000018	and $6,$1, $2	010001	00001	00010	00110 xxxxxxxxxxx	$6=0x00000006
0x0000001C	or $7, $1, $2	010010	00001	00010	00111 xxxxxxxxxxx	$7=0x0000000F
0x00000020	beq $1, $8, 2	110000	00001	01000	0000 0000 0000 0010	
0x00000024	add $8, $0, $1	000000	00000	00001	01000 xxxxxxxxxxx	$8=0x0000006
0x00000028	beq $1, $8, -3	110000	00001	01000	1111 1111 1111 1101	
0x0000002C	halt	111111	00000	00000	0000 0000 0000 0000	

（注：以上代码中的x表示不确定值，在本条指令中不影响值得计算结果，但是在实际的指令内存文件中为方便统一，所有的x都设置为0）

3.相关模块的设计

本设计的所有模块如图3所示




```
23  module Opdecoder(
24      input [31:0] inst,
25      output wire [5:0] opcode,
26      output wire [4:0] rs,
27      output wire [4:0] rt,
28      output wire [4:0] rd,
29      output wire [4:0] sa,
30      output wire [15:0] immed,
31      output wire [25:0] addr
32      );
33
34      assign opcode = inst[31:26];
35      assign rs = inst[25:21];
36      assign rt = inst[20:16];
37      assign rd = inst[15:11];
38      assign sa = inst[10:6];
39      assign immed = inst[15:0];
40      assign addr = inst[25:0];
41  endmodule
```

图3 所有模块　　**图4 Opdecoder模块实现**

其中，从模块化的思想来设计，自己分离出了两个个人的模块：

1. **Opdecoder**(输入32位指令，分割出rd，rs，opcode等等)如图4所示

2. **LeftShift2**（判断如果是lw或者是sw指令的话，扩展的立即数先左移两位处理）本来应该再加多一个信号，用来判断是不是sw或者是lw指令，连进这个模块，但是到后来实践的时候发现**DBDataSrc恰好可以完成这个信号所用到的功能**，所以在这里就把DBDataSrc再连出来，如图5，图6所示

```
23  module LeftShift2(
24      input DoubleEn,
25      input [31:0] in_32,
26      output [31:0] out_32
27      );
28
29      assign out_32 = (DoubleEn) ? {in_32[29:0], 2'b00} : in_32;
30  endmodule
31
```

```
// double imm_32 for the shift
LeftShift2 doubleHandler(
    .DoubleEn(cu.DBDataSrc),
    .in_32(extend.imm_32),

    .out_32()
    );
```

图5 LeftShift2模块实现　　**图6 顶层模块CPU的实例化**

其他模块的实现思路：

1. **ALU32**：拿到两个操作数rega，regb，通过case判断ALUopcode来执行相关逻辑，得出result，result又会影响zero的输出反馈回ControlUnit

2. **DataMemory**：里面有一个单位长度为8的reg的数组，数组长度为60（足够使用就好），表示数据内存的大小。通过RD的值来对输出DataOut持续赋值，在时钟下降沿的时候通过WR来判断是否需要将DataIn的数据存入到相应的内存位置中。

3. **Extend**：通过控制信号ExtSel，来进行相应的符号位扩展。

ExtSel	imm_16最高位	扩展内容
0	0	0
0	1	0
1	0	0
1	1	1

从真值表中不难看出，当 **ExtSel && imm_16[15] == 1** 为真的时候，扩展1；否则扩展0

4. **InstructionMemory**：初始化的时候调用系统调用$readmemb读入已经写好的指令文件（"C:/Users/longj/Desktop/vivado/VivadoProject/single_cycle_cpu/memFile.txt"）（若要在其他机器上运行请调整好正确的文件路径），因为指令存储器和数据存储器存储单元宽度一律使用8位，所以通过RW控制信号来判断是否从内存中连续读取4条记录构建出32位的指令，指令输出到Opdecoder里面进行解码，Opdecoder再把解码的内容送入到ControlUnit中进行相关的信号生成。

5. **Mux_TwotoOne5**：5位的二选一多路选择器，通过信号sel来判断应该输出哪一个数据。

6. **Mux_TwotoOne32**：同Mux_TwotoOne5，只是输入输出数据为32位

7. **PC**：当clk上升的时候，输出curPC进行改变，若信号reset为0，则初始化把起始指令地址设为0，若reset=1，则将来自PCShift模块的nextPC赋给curPC，实现pc的更新

8. **PCShift**：PC移位器，通过来自ControlUnit的信号PCSrc来判断如何得到下一个pc的地址，先将输入的curPC加4，若PCSrc为1，则还要加上beq带来的偏移量，然后把输出nextPC送到PC模块中。

9. **Regfile**：寄存器存储器，32 位作为存储器存储单元宽度，总共给 32 个寄存器，其中 0 号寄存器不可写，所有寄存器的在初始化的时候都初始化为 0。通过 RdReg1addr 和 RdReg2addr 给 reg1dataOut 和 reg2dataOut 进行持续的赋值，当 clk 下降沿的时候通过信号 WrRegaddr 进行判断要不要写寄存器

10. **ControlUnit**：输入来自Opdecoder送来的opcode，来产生所有的信号。为每一个指令，都通过opcode声明对应的一个唯一的fingerprint，然后通过表三和表四，生成相应的信号。

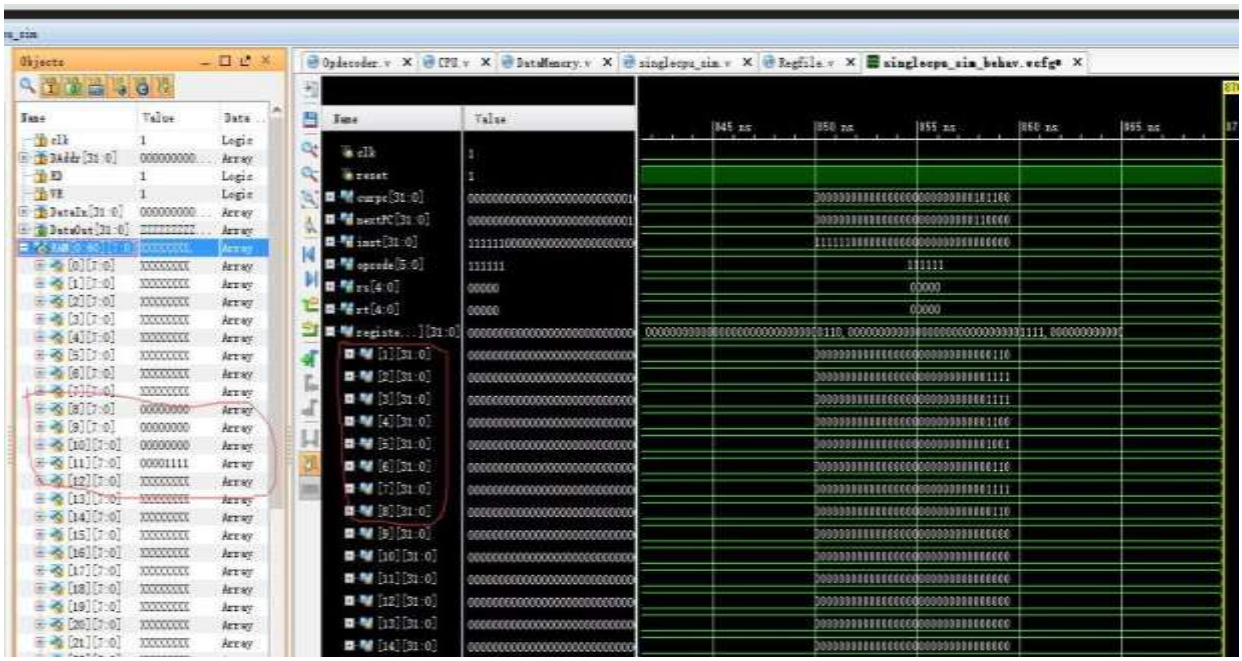
11. **CPU**：顶层模块，把相关以上的所有的子模块实例化并且连接起来，其中有较多

output并没有进行赋值实现，原因是在仿真模拟的时候可以通过手动添加观察值来看到所有模块想监控的数据内容，所以没有必要output到顶层模块中了

12. **仿真模块**：实例化一个CPU，把clk和reset传进去，整个单周期CPU就可以开始工作

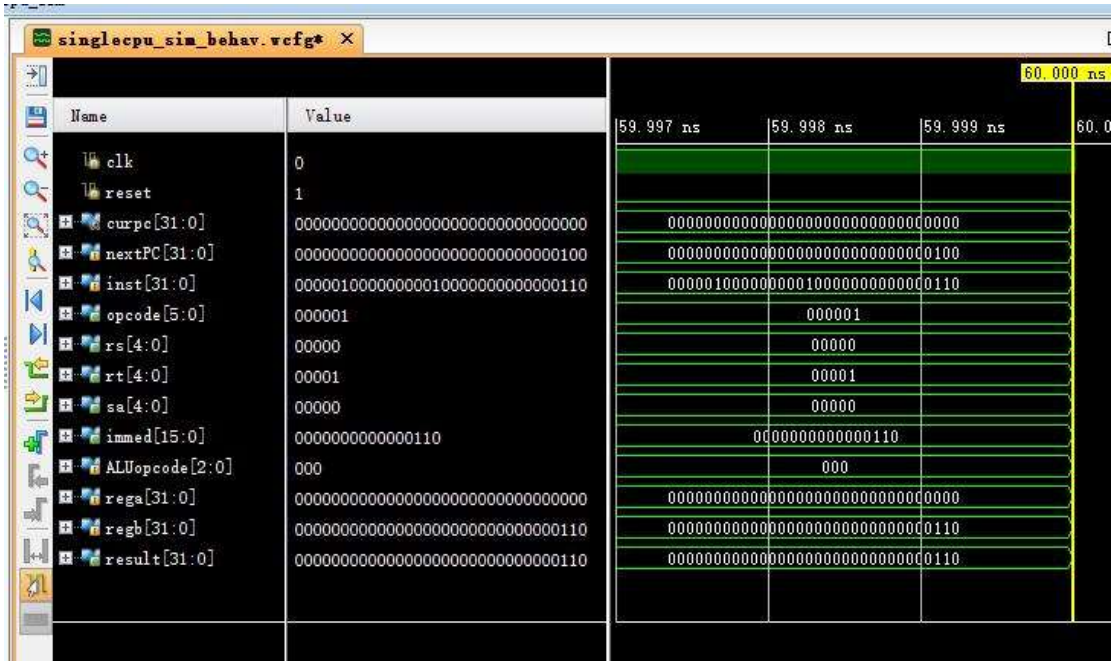
4. 测试数据指令的实验结果

1）整个程序运行完成后的所有用到的寄存器的值和表五的预期值相比较，发现可以完全符合，datamem的存储情况也和预期的一致。从而我们可以知道整个运行的结果是正确的。

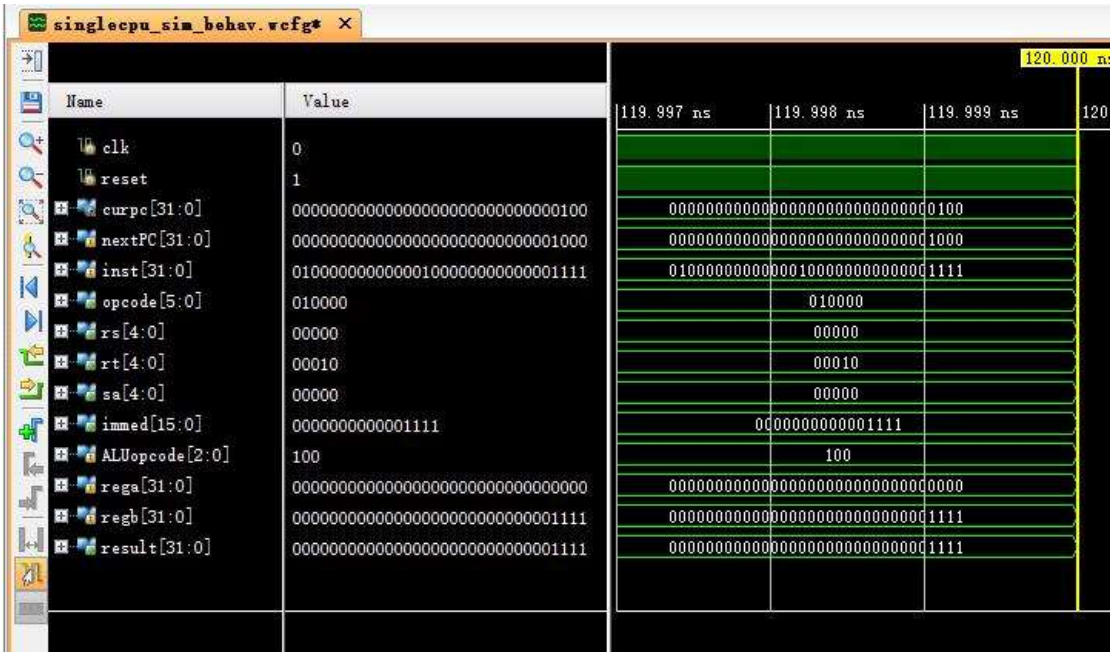


2）我们把register的结果数组从观测列表中移出,然后观察每一条指令的opdecoder的输出，看每一步的指令解析是否正确：

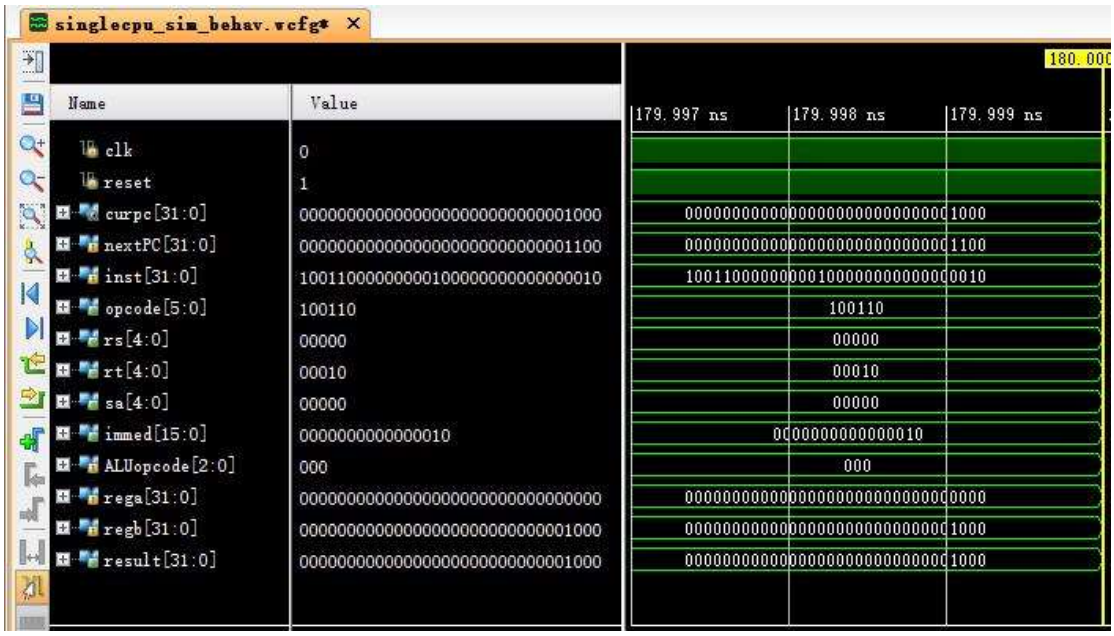
addi


singlecpu_sim_behav.wcfg*
Name
Value
clk 0
reset 1
curpc[31:0] 00000000000000000000000000000000
nextPC[31:0] 00000000000000000000000000000100
inst[31:0] 00000100000000010000000000000110
opcode[5:0] 000001
rs[4:0] 00000
rt[4:0] 00001
sa[4:0] 00000
immed[15:0] 0000000000000110
ALUopcode[2:0] 000
rega[31:0] 00000000000000000000000000000000
regb[31:0] 00000000000000000000000000000110
result[31:0] 00000000000000000000000000000110
60.000 ns
59.997 ns
59.998 ns
59.999 ns

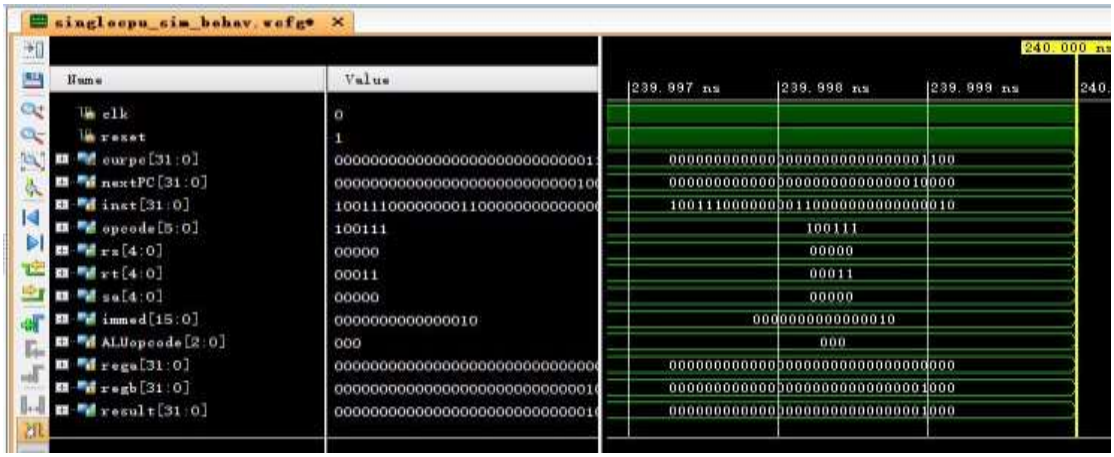
ori


singlecpu_sim_behav.wcfg*
Name
Value
clk 0
reset 1
curpc[31:0] 00000000000000000000000000000100
nextPC[31:0] 00000000000000000000000000001000
inst[31:0] 01000000000001000000000000001111
opcode[5:0] 010000
rs[4:0] 00000
rt[4:0] 00010
sa[4:0] 00000
immed[15:0] 0000000000001111
ALUopcode[2:0] 100
rega[31:0] 00000000000000000000000000000000
regb[31:0] 00000000000000000000000000001111
result[31:0] 00000000000000000000000000001111
120.000 ns
119.997 ns
119.998 ns
119.999 ns

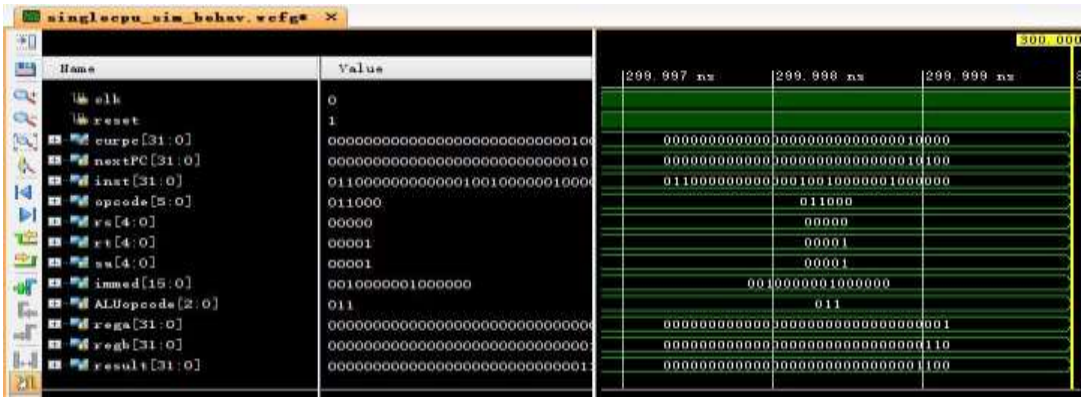
sw


singlecpu_sim_behav.wcfg*
Name
Value
clk
reset
curpc[31:0]
nextPC[31:0]
inst[31:0]
opcode[5:0]
rs[4:0]
rt[4:0]
sa[4:0]
immed[15:0]
ALUopcode[2:0]
rega[31:0]
regb[31:0]
result[31:0]
179.997 ns
179.998 ns
179.999 ns

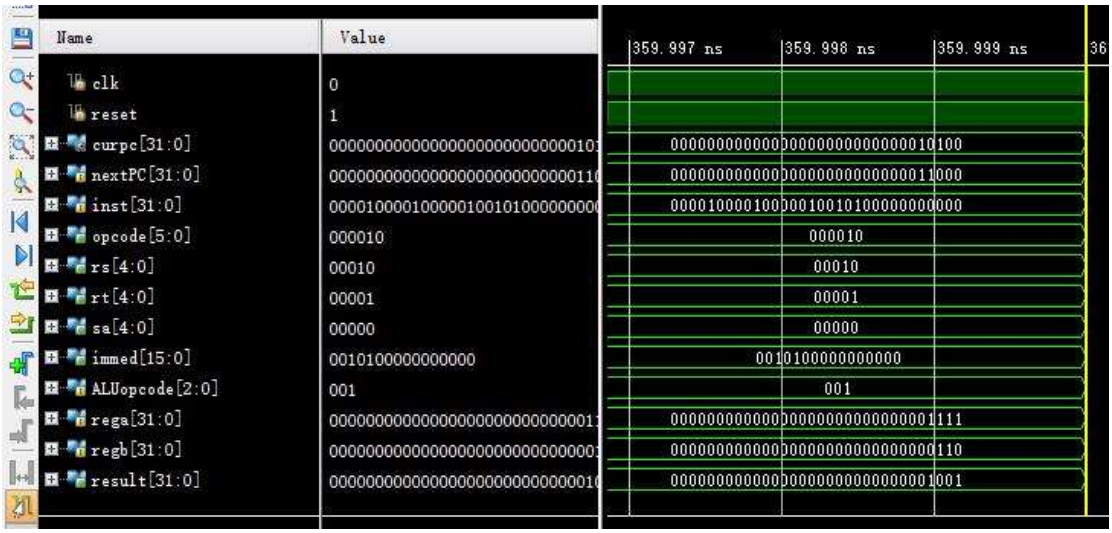
lw


singlecpu_sim_behav.wcfg*
Name
Value
clk
reset
curpc[31:0]
nextPC[31:0]
inst[31:0]
opcode[5:0]
rs[4:0]
rt[4:0]
sa[4:0]
immed[15:0]
ALUopcode[2:0]
rega[31:0]
regb[31:0]
result[31:0]
239.997 ns
239.998 ns
239.999 ns

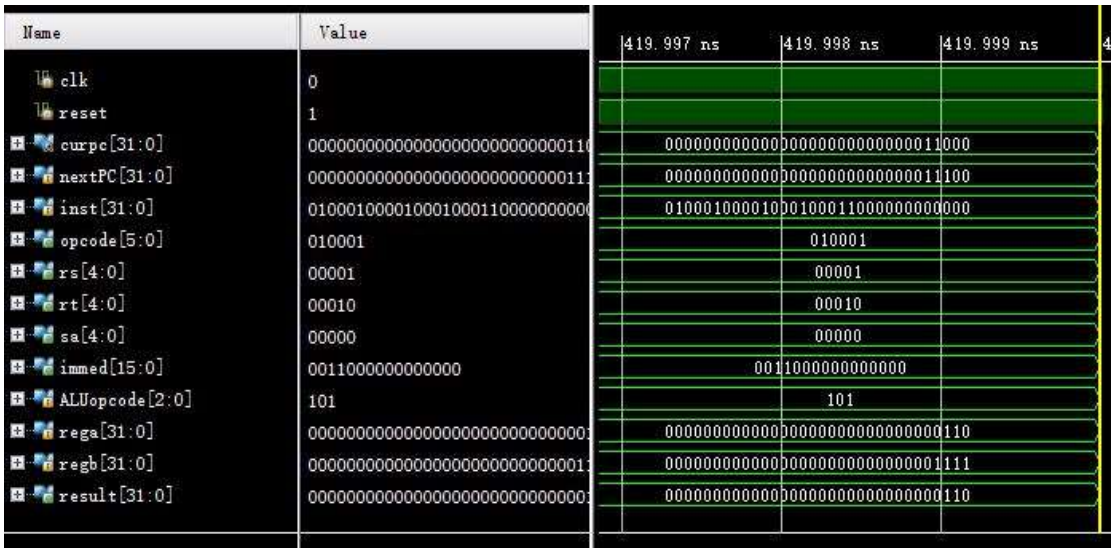
sll


singlecpu_sim_behav.wcfg*
Name
Value
clk
reset
curpc[31:0]
nextPC[31:0]
inst[31:0]
opcode[5:0]
rs[4:0]
rt[4:0]
sa[4:0]
immed[15:0]
ALUopcode[2:0]
rega[31:0]
regb[31:0]
result[31:0]
299.997 ns
299.998 ns
299.999 ns

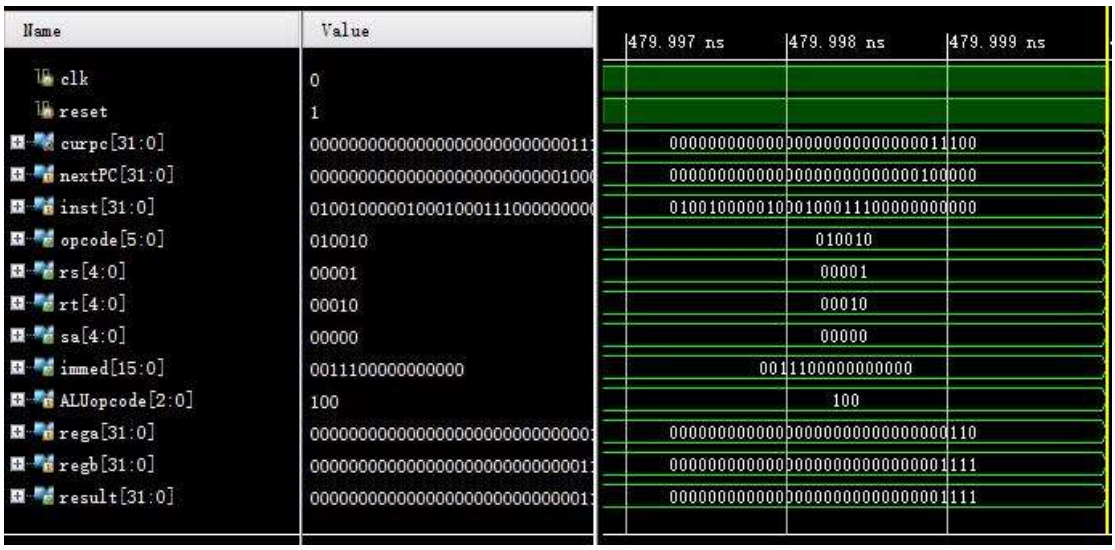
sub


Name
Value
359.997 ns
359.998 ns
359.999 ns
clk
0
reset
1
curpc[31:0]
nextPC[31:0]
inst[31:0]
opcode[5:0]
000010
rs[4:0]
00010
rt[4:0]
00001
sa[4:0]
00000
immed[15:0]
0010100000000000
ALUopcode[2:0]
001
rega[31:0]
regb[31:0]
result[31:0]

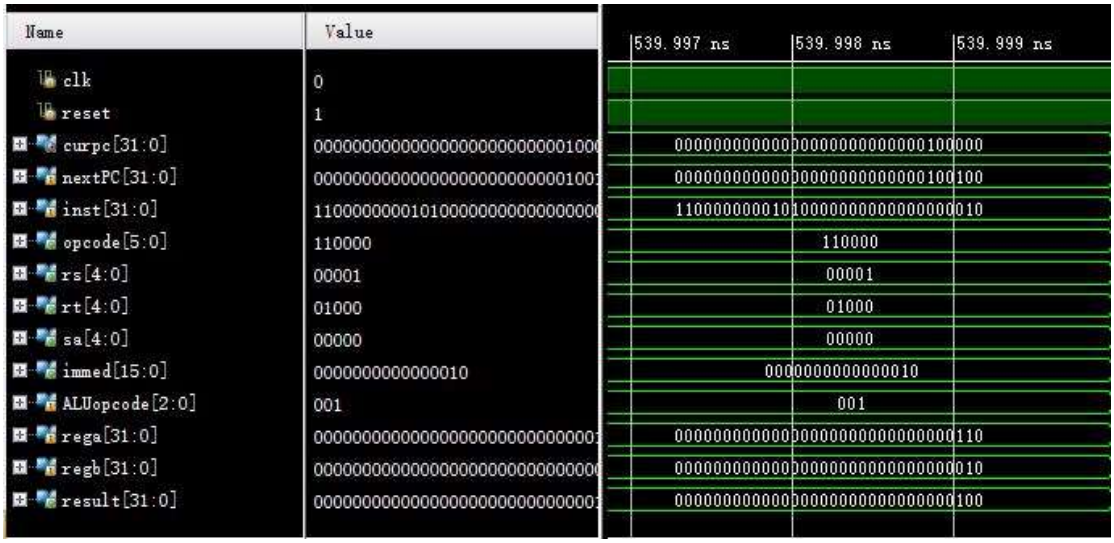
and


Name
Value
419.997 ns
419.998 ns
419.999 ns
clk
0
reset
1
curpc[31:0]
nextPC[31:0]
inst[31:0]
opcode[5:0]
010001
rs[4:0]
00001
rt[4:0]
00010
sa[4:0]
00000
immed[15:0]
0011000000000000
ALUopcode[2:0]
101
rega[31:0]
regb[31:0]
result[31:0]

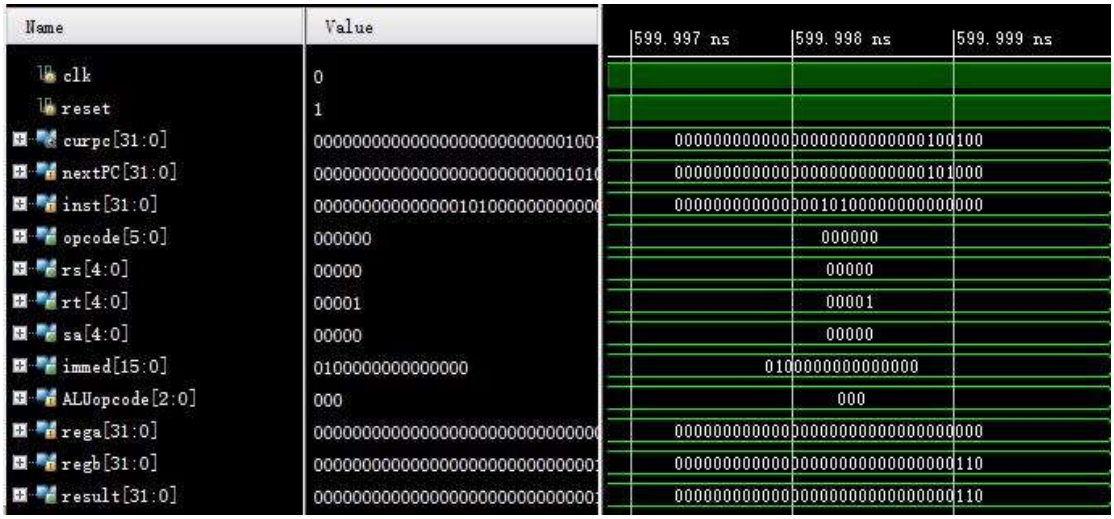
or


Name
Value
479.997 ns
479.998 ns
479.999 ns
clk
0
reset
1
curpc[31:0]
nextPC[31:0]
inst[31:0]
opcode[5:0]
010010
rs[4:0]
00001
rt[4:0]
00010
sa[4:0]
00000
immed[15:0]
0011100000000000
ALUopcode[2:0]
100
rega[31:0]
regb[31:0]
result[31:0]

beq(false)第一个



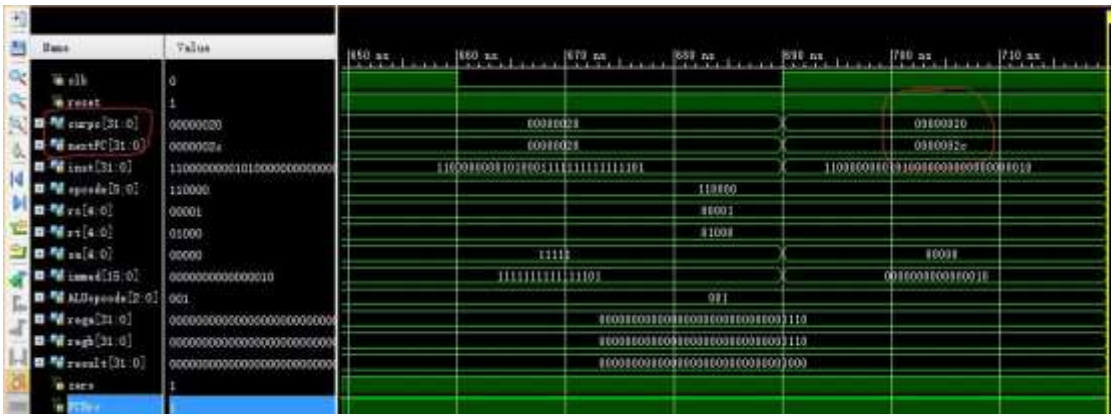
add



beq(true)第二个

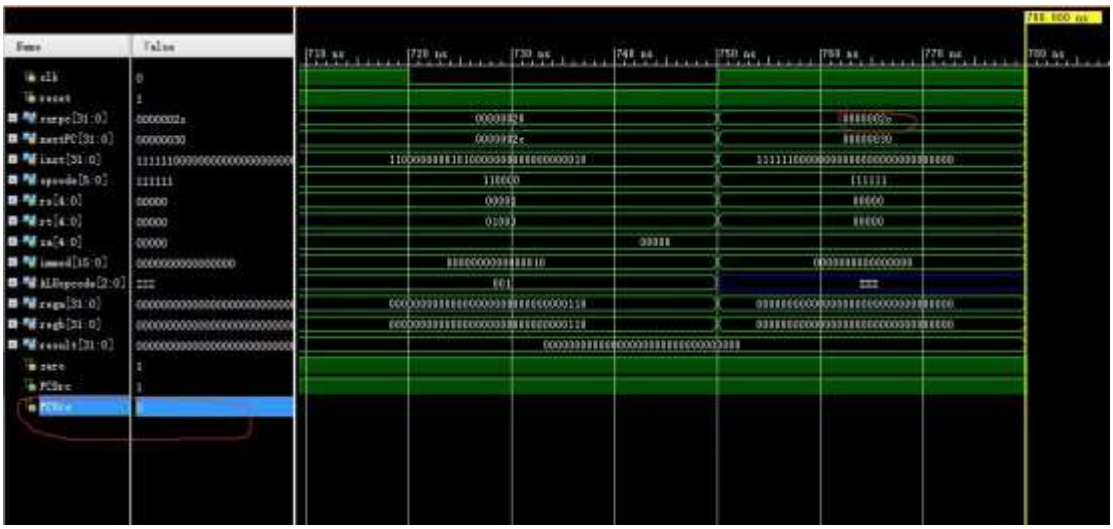
注意到这里的curpc比nextpc要大，而且对比表五说明的确返回到了正确的位置，也就是第一个beq语句

beq(true)第一个



（注意到这里的nextpc将会跳转到halt的指令中去）

halt



停机成功，PCWre已经为0，pc不会改变，然后对于对于111111这个opcode，我在CU里面把它归为default分支，统一输出高阻态zzz。将表五和每一个截图中的处理后输出的opcode，rs，rt，rd，immediate对比发现都是匹配的，于是整个仿真模拟结束。

六. 实验心得

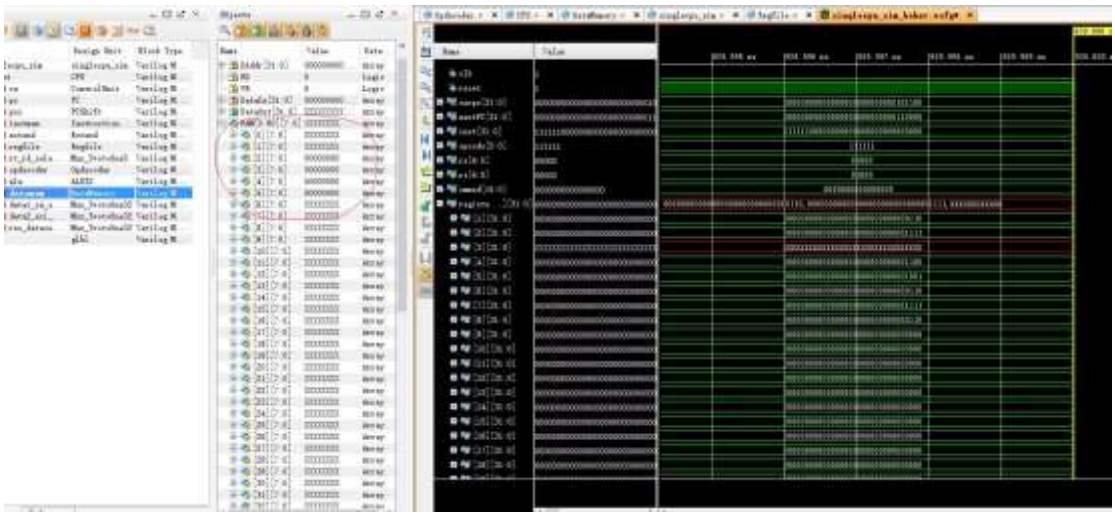
1.总体了解了单周期cpu的整个工作过程。

2.遇到的问题1，指令读不进去，解决方法，原来是路径设置得不对，改正即可。

3.老师所给的参考代码中，RAM中的信号的意思的定义和word中的定义恰好相反！这个坑了我很久，导致sw和lw指令总是不能正常运行，整个调试的过程中，单步执行到RAM，再分析相关指令和信息的表格，更换一下赋值的条件，即可解决。

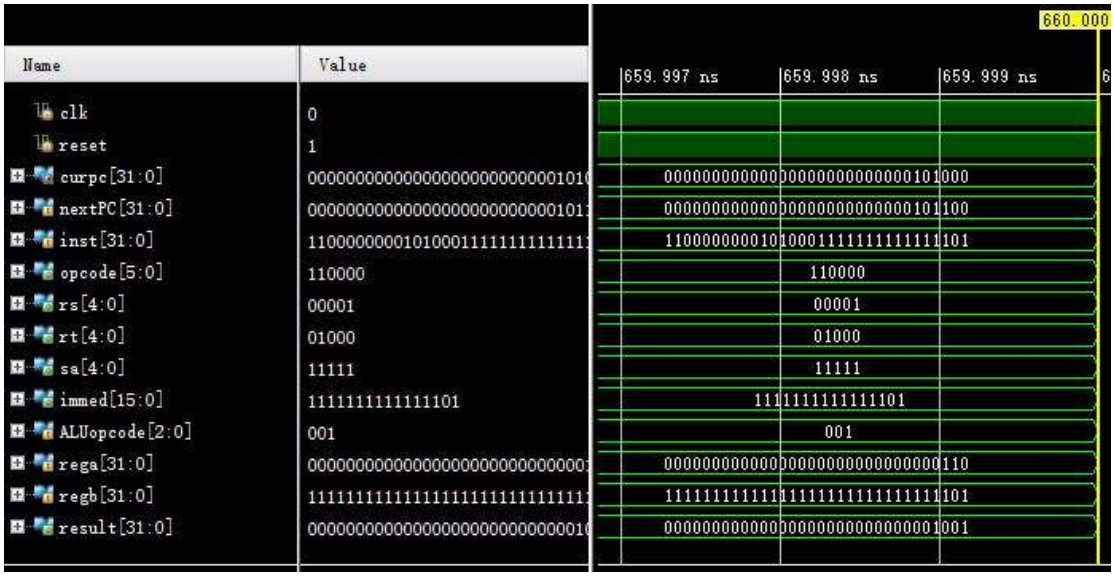
4. 初次执行的原来的代码的时候，会发现第三个寄存器存的是不确定的值，说明lw有问题，那么sw也有可能有问题，查看datamem的相关内容，发现存的位置出现了问题，实际

上偏移量的立即数还需要左移两位（乘以4）才能够到达正确的位置，这个在老师一开始给的文档里面出现了疏漏！思考，在不大改变原来逻辑的前提上，增加了一个新的模块LeftShift2，然后用来判断是否需要对过来的32位扩展数进行移位处理，控制信号用DBDataSrc恰好可以满足，故不需要在cu里面增加额外的信号，然后在顶层模块里面插入这个LeftShift2模块，就可以解决问题了。



5. beq不跳转问题

修复了sw和lw的bug之后，整个程序运行似乎很正常，然后相关的寄存器的值都已经匹配预期值，本来以为实验已经成功，但是在单步运行一步步比较指令行为的时候才发现beq指令只执行了两次，这个与预期的不符，因为beq的执行并没有改变寄存器的值，所以只看寄存器的值就没能看出来。实际上期望的是第一个beq为false不跳转，然后第二个beq为true跳转回第一个beq，然后第一个beq为true跳转到halt指令，但是现在明显与预期的不同。查看出现问题的指令的附近的值发现nextpc的选择没有达到预期（如图7），所以需要开始调试。



调试过程：

首先，跑到第一个beq那里，然后在ALU32的case上面设置一个断点，通过单步执行不断检查里面的对应的操作数是否正确，**特别是ALUopcode为001的时候**（这个时候是和beq有关的），在第二次beq里面，就发现了regb出现了问题，regb出现的是指令里面的立即数（-3）！而期望的应该是$8里面的值，那么就是regb的输入端出现了问题，进一步返回去追踪，发现ALUSrcB为1，导致了二选一多路选择器选择了立即数输入，返回去看ALUSrcB的生成代码，发现了问题。

```
assign RegWre = add | addi | sub | ori | and_ | or_ | s
assign ALUSrcA = sll;
assign ALUSrcB = addi | ori | sw | lw | beq;
assign PCSrc = zero;
```

此处，beq的立即数参与了ALU的运算，原来是混淆了beq的立即数使用的场合，beq的立即数应该是传到pcshift里面进行偏移量处理的，但是却误送到了ALU中，导致result出错，进一步导致zero出错，nextpc没有进行立即数偏移处理，解决方法是把上述截图中的beq给去掉，再次仿真，即可得到正确的仿真行为。

6.再次体会到了模块化设计思想的重要性，分离了逻辑，为我调试第5点中所提到的问题带来了极大的方便，如果没有把二选一选择器分离出来，那么我就要看包含了选择器的扩展器模块，无疑会加大调试的难度。

7.其实一开始是会摸不着头脑的，但是到了后来明白了顶层模块只是实例化了相应的子模块，然后传入需要的input参数作为输入，输出参数是可选的时候，就可以很方便地把整个cpu给串联起来了。然后进行模拟，一步一步从pc开始debug，就可以逐渐完成整个实验内容。整个实验再一次锻炼了我独立思考的能力，对计算机体系结构有了进一步的了解，收货颇多。