****

**《计算机组成原理》**

**实验报告**

**学生姓名 孙淼**

**学 号 2018211958**

**专业班级 计算机科学与技术18-2班**

**指导教师 陈田老师**

**院系名称 计算机与信息学院**

**2020年 6月 1日**

**实验五、单周期CPU设计与实现——十条指令CPU**

1. **实验目的:**

通过设计并实现支持10条指令的CPU，进一步理解和掌握CPU设计的基本原理和过程。

1. **实验内容：**

设计和实现一个支持十条指令的单周期CPU。

由于我选用的是32位MIPS指令，所以在MIPS指令集中选取了10条典型指令进行实现（与实验指导书的10条指令不同）：

1. **实验原理**

依次给出指令格式定义，数据通路和控制单元的设计，并给出目标CPU的原理图。

**指令格式定义**

***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）

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

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

2.addi rt , rs ,immediate

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

| 000001 | rs(5位) | rt(5位) | immediate(16位) |
| --- | --- | --- | --- |

3.sub rd , rs , rt

功能：rd←rs – rt

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

***逻辑运算指令***

4. ori rt , rs ,immediate

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

| 010000 | rs(5位) | rt(5位) | immediate(16位) |
| --- | --- | --- | --- |

5. and rd , rs , rt

功能：rd←rs & rt；逻辑与运算。

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

6. or rd , rs , rt

功能：rd←rs | rt；逻辑或运算。

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

***传送指令***

**7.** move rd , rs

功能：rd←rs + $0 ；$0=$zero=0。

| 100000 | rs(5位) | 00000 | rd(5位) | reserved |
| --- | --- | --- | --- | --- |

***访存指令***

8. sw rt ,immediate(rs) 写存储器

功能：memory[rs+ (sign-extend)immediate]←rt；immediate符号扩展再相加。

| 100110 | rs(5位) | rt(5位) | immediate(16位) |
| --- | --- | --- | --- |

**9.** lw rt , immediate(rs) 读存储器

功能：rt ← memory[rs + (sign-extend)immediate]；immediate符号扩展再相加。

| 100111 | rs(5位) | rt(5位) | immediate(16位) |
| --- | --- | --- | --- |

***分支指令***

**10.** beq rs,rt,immediate

功能：if(rs=rt) pc←pc + 4 + (sign-extend)immediate <<2； 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”。

| 110000 | rs(5位) | rt(5位) | immediate(位移量，16位) |
| --- | --- | --- | --- |

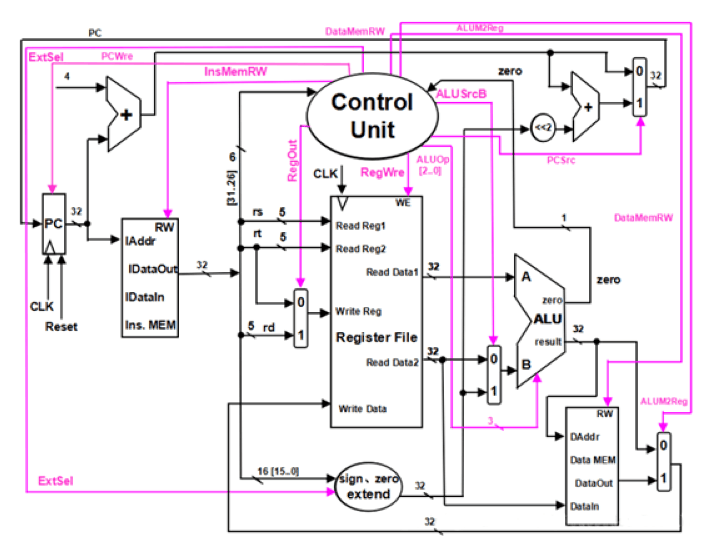
***停机指令***

11. halt

功能：停机；不改变PC的值，PC保持不变。

| 111111 | 00000000000000000000000000(26位) |
| --- | --- |

**数据通路**

上图是一个简单的基本上能够在单周期上完成所要求设计的指令功能的数据通路和必要的数据通路图。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/写信号控制（1-写，0-读。当然，也可以由时钟信号控制，但必须在图上标出）。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。

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1写，为0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

RW，数据存储器读写控制信号，为1写，为0读

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

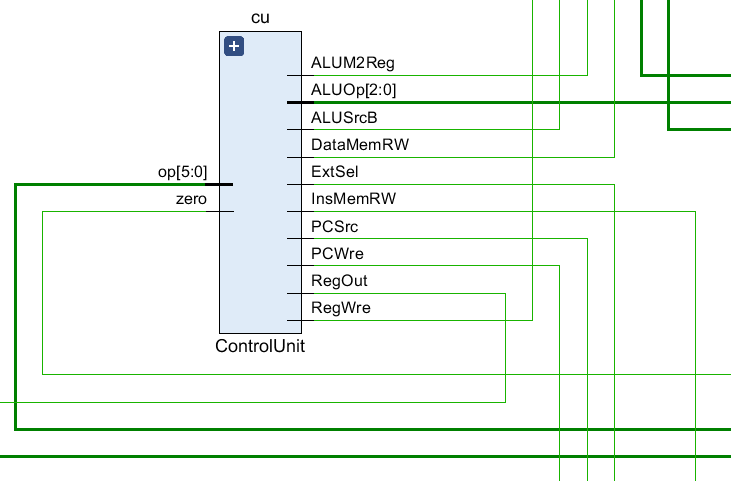
Read Data2，rt寄存器数据输出端口

WE，写使能信号，为1时，在时钟上升沿写入

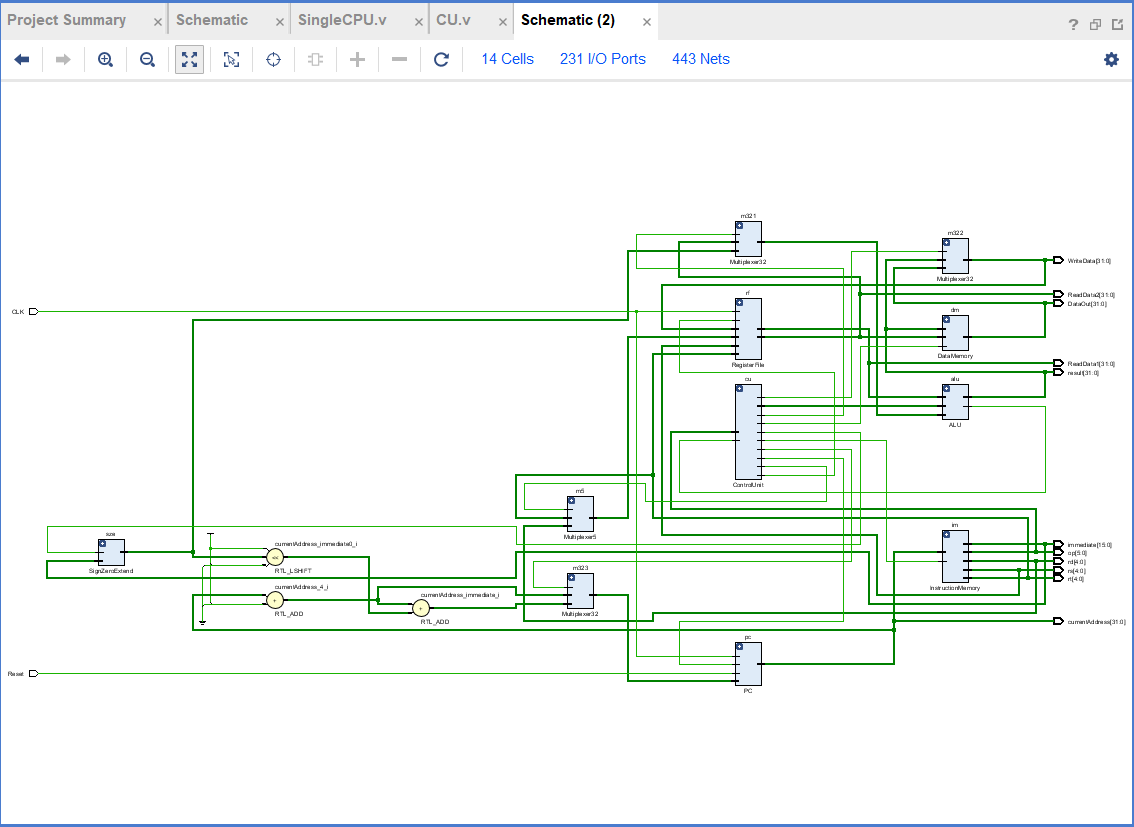
ALU：

result，ALU运算结果

zero，运算结果标志，结果为0输出1，否则输出0

控制单元

| 控制信号名 | 状态“0” | 状态“1” |
| --- | --- | --- |
| 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：add、sub、or、and、move、beq | 来自sign或zero扩展的立即数，相关指令：addi、ori、sw、lw |
| ALUM2Reg | 来自ALU运算结果的输出，相关指令：add、addi、sub、ori、or、and、move | 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：sw、halt | 寄存器组写使能，相关指令：add、addi、sub、ori、or、and、move、lw |
| InsMemRW | 读指令存储器(Ins. Data)，初始化为0 | 写指令存储器 |
| DataMemRW | 读数据存储器，相关指令：lw | 写数据存储器，相关指令：sw |
| ExtSel | 相关指令：ori，(zero-extend)immediate（0扩展） | 相关指令：addi、sw、lw、beq，(sign-extend)immediate（符号扩展） |
| PCSrc | PC←PC+4，相关指令：add、sub、ori、or、and、move、sw、lw、beq(zero=0) | PC←PC+4+(sign-extend)immediate，同时zero=1，相关指令：beq |
| RegOut | 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、move |
| ALUOp[2..0] | ALU 8种运算功能选择(000-111)，看功能表 |  |

**目标CPU的原理图**

1. **实验步骤**

每一个组件都能写成一个下层模块，实现相应的功能，并保证输入变化后输出跟着变化(不发生延迟)。   
顶层模块调用各个下层模块，根据数据通路图将模块之间连线，保证PC指令改变后，其他所有的模块都根据控制信号，发生对应的变化。   
测试模块控制CLK和Reset信号，控制PC指令的改变。

下面对实验过程进行介绍：

各个下层模块的介绍：

**PC：CLK上升沿触发，更改指令地址**

输入：CLK，Reset，PCWre，newAddress   
输出：currentAddress   
解释：由于指令地址存储在寄存器里，一开始需要赋currentAddress为0。Reset是重置信号，当为1时，指令寄存器地址重置。PCWre的作用为保留现场，如果PCWre为0，指令地址不变。

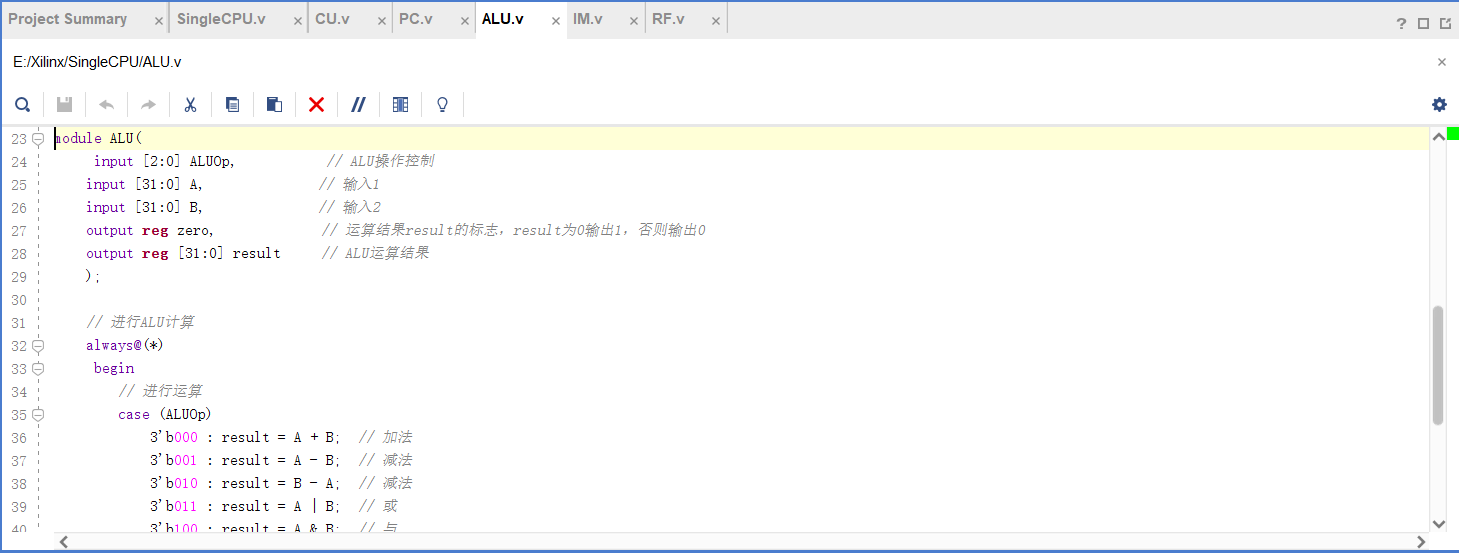
**** **InstructionMemory：储存指令，分割指令**

输入：InsMenRW，IAddr   
输出：op，rs，rt，rd，immediate   
解释：该部分为指令寄存器，通过一个64大小的8位寄存器数组来保存从文件输入的全部指令。然后通过输入的地址，找到相应的指令，并分割成op，rs，rt，rd，immediate输出。（由于寄存器地址+4，所以不用右移变换成真正的地址

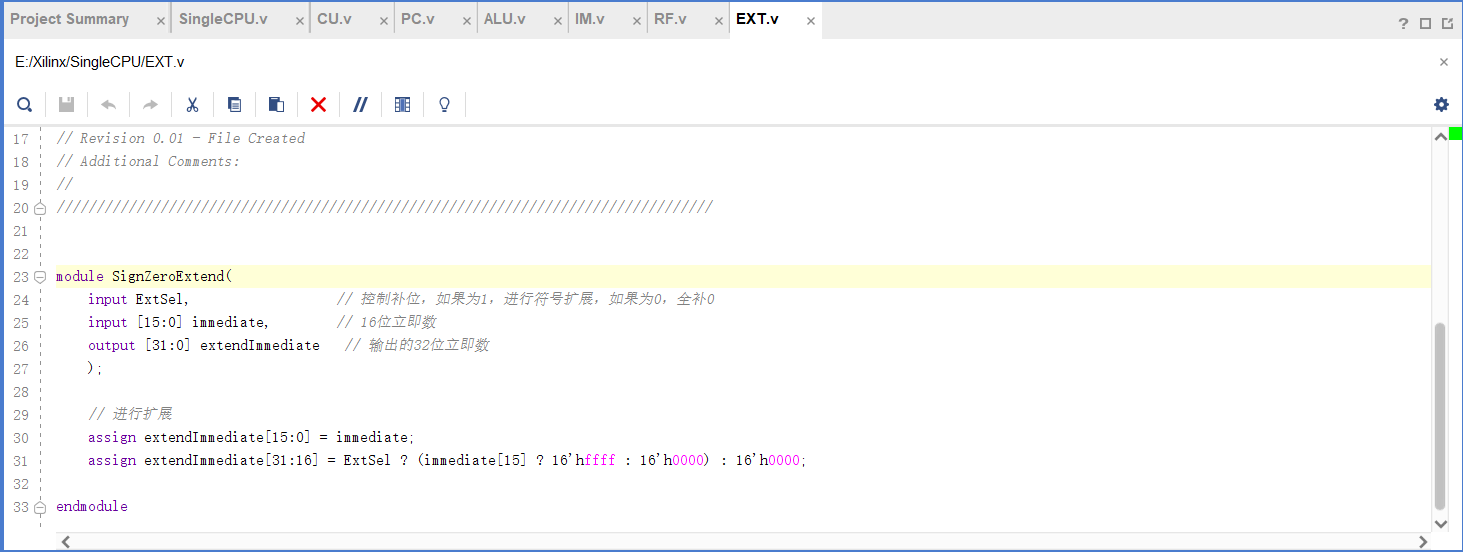
**RegisterFile：储存寄存器组，并根据地址对寄存器组进行读写**

输入：CLK，RegWre，rs，rt，WriteReg，WriteData   
输出：ReadData1，ReadData2   
解释：该部分为寄存器读写单元，RegWre的作用是控制寄存器是否写入。同上，通过一个16大小的32位寄存器数组来模拟寄存器，开始时全部置0。通过访问寄存器的地址，来获取寄存器里面的值，并进行操作。（PS：由于$0恒为0，所以写入寄存器的地址不能为0）

**ALU（算术逻辑单元）：用于逻辑指令计算和跳转指令比较**

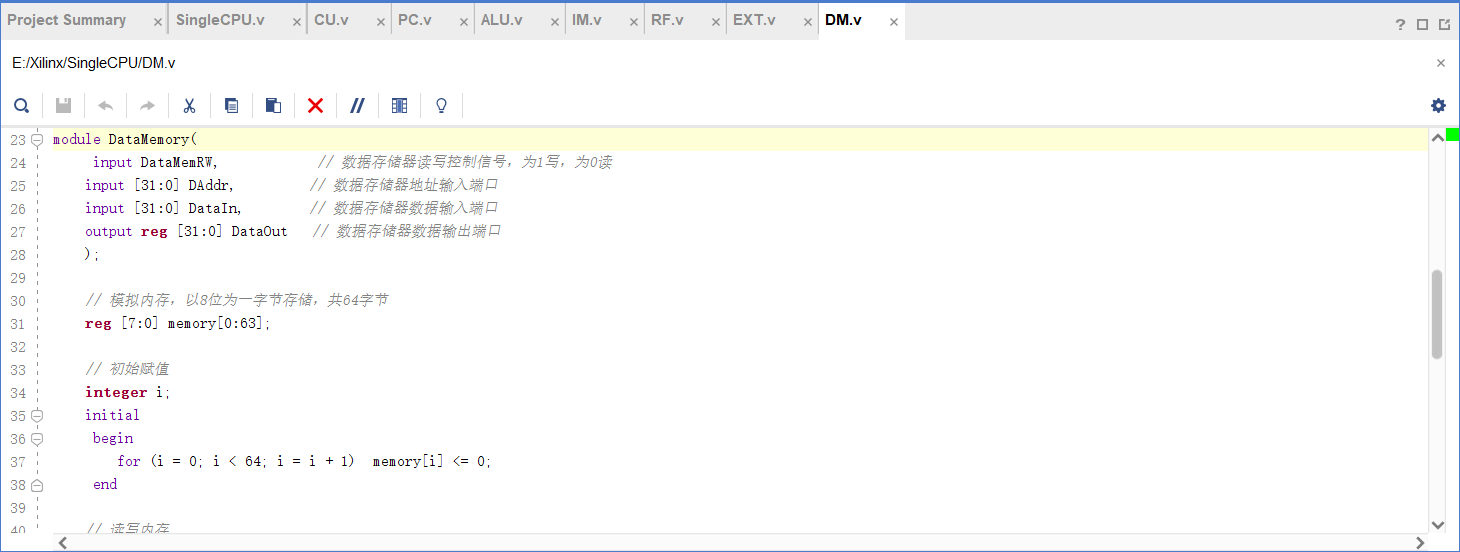
输入：ALUOp，A，B   
输出：zero，result   
解释：ALUOp用于控制算数的类型，AB为输入数，result为运算结果，zero主要用于beq和bne指令的判断。

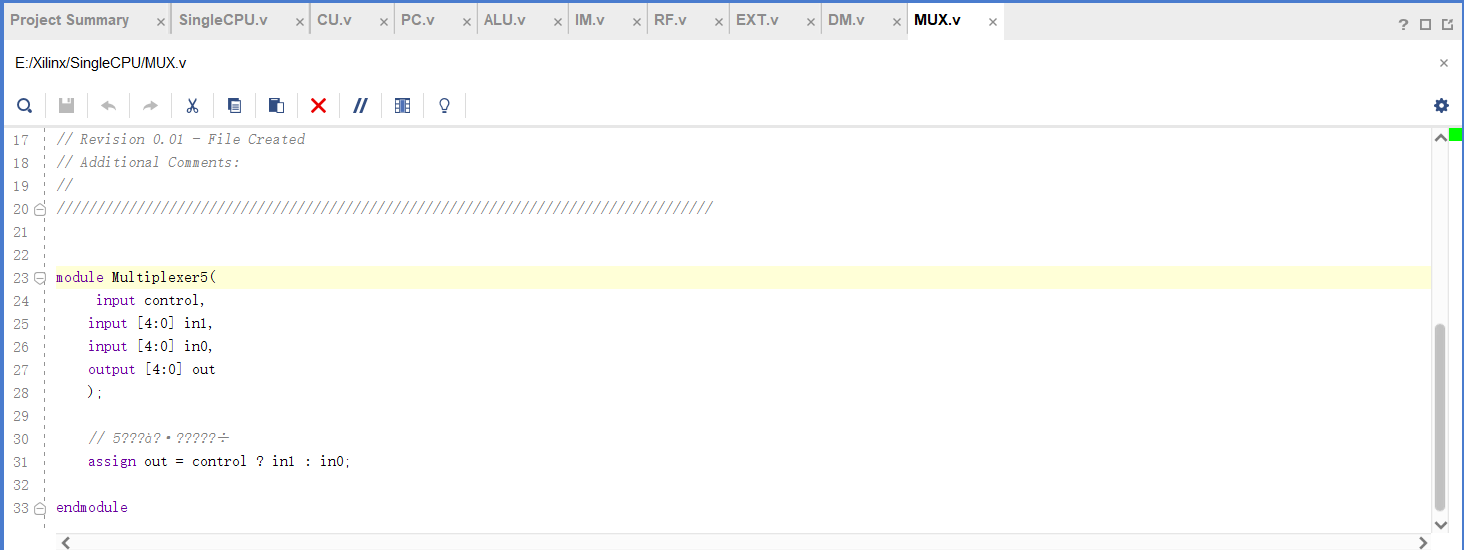
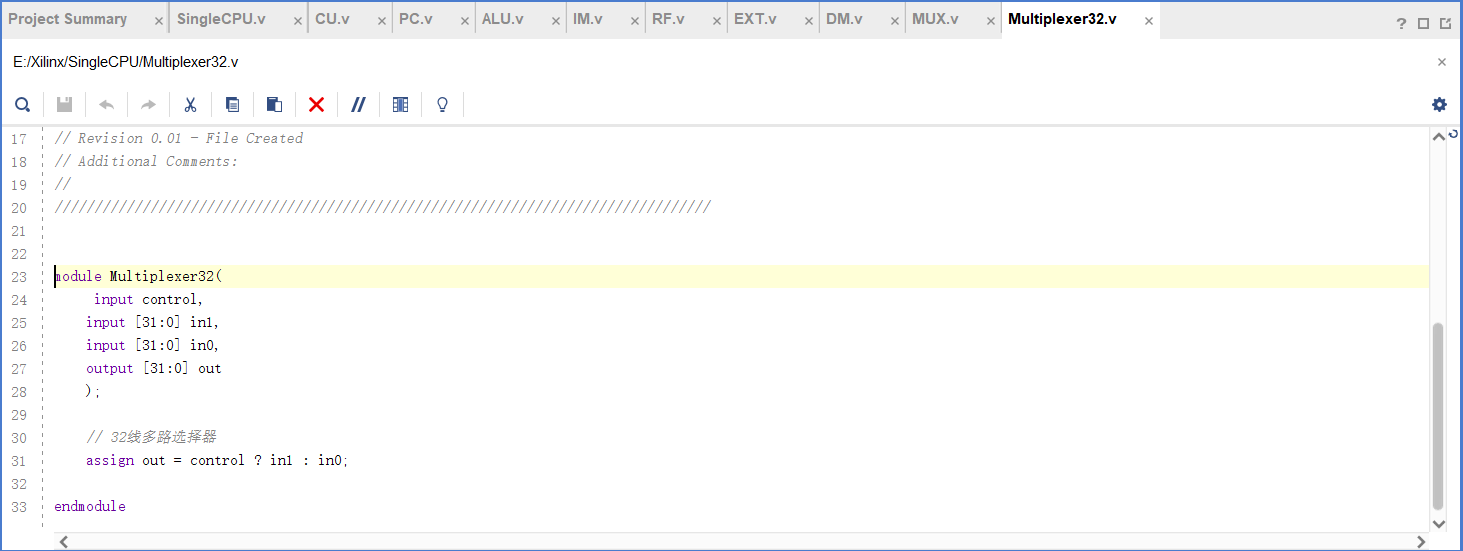
**SignZeroExtend：用于immediate的扩展**

输入：ExtSel，immediate   
输出：extendImmediate   
解释：比较简单的一个模块。ExtSel为控制补位信号。判断后，将extendImmediate的前16位全补1或0即可。

**DataMemory：用于内存存储，内存读写**

输入：DataMenRW，DAddr，DataIn   
输出：DataOut   
解释：该部分控制内存存储。同上，用64大小的8位寄存器数组模拟内存（内存小主要是因为编译快），内存部分采用小端模式。DataMenRW控制内存读写。由于指令为真实地址，所以不需要\*4。

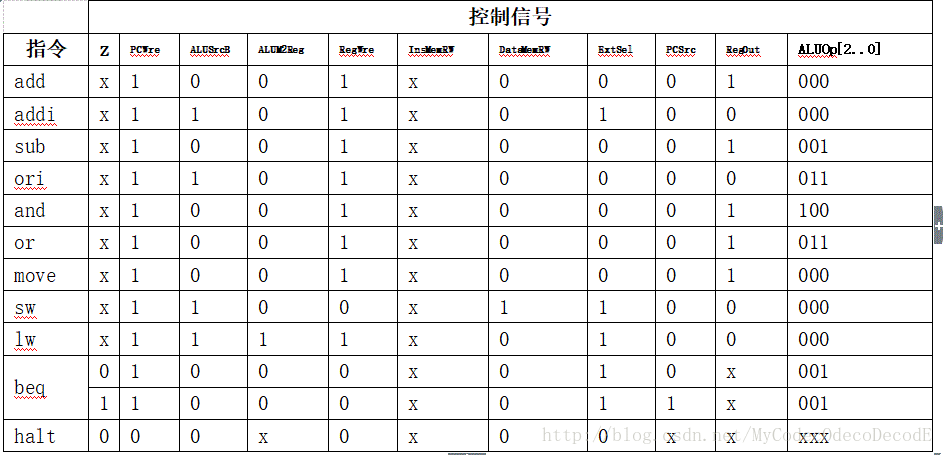
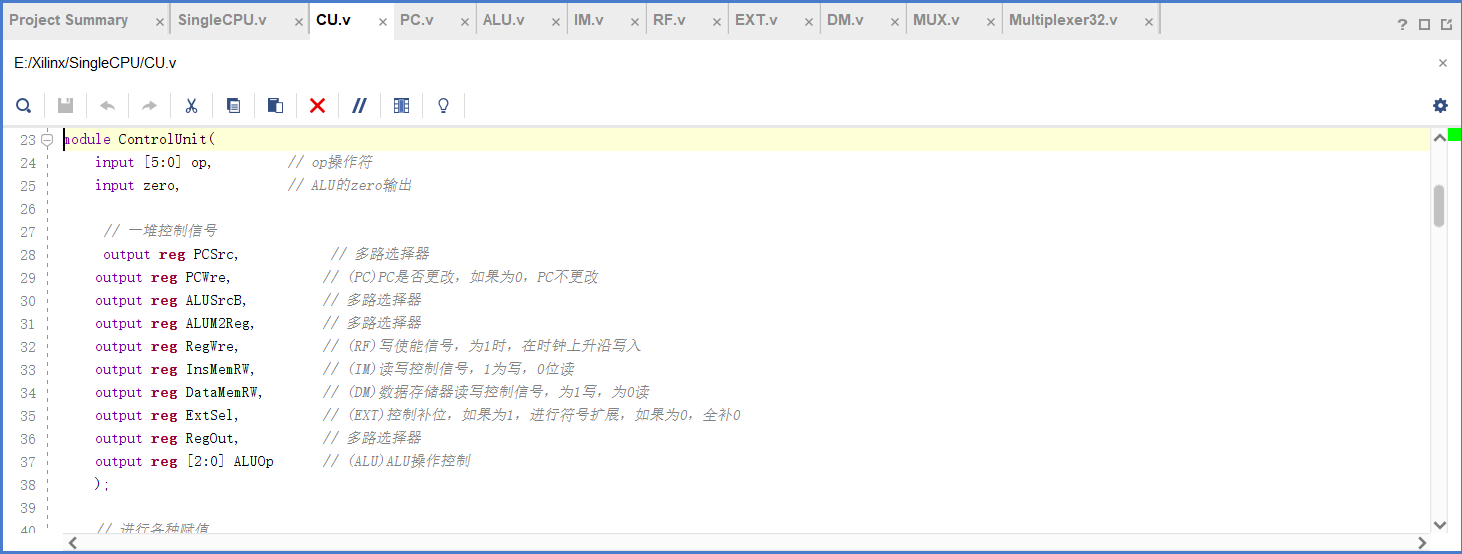
**Multiplexer：5线和32线二路选择器**

输入：control，in1，in0   
输出：out   
解释：多路选择器

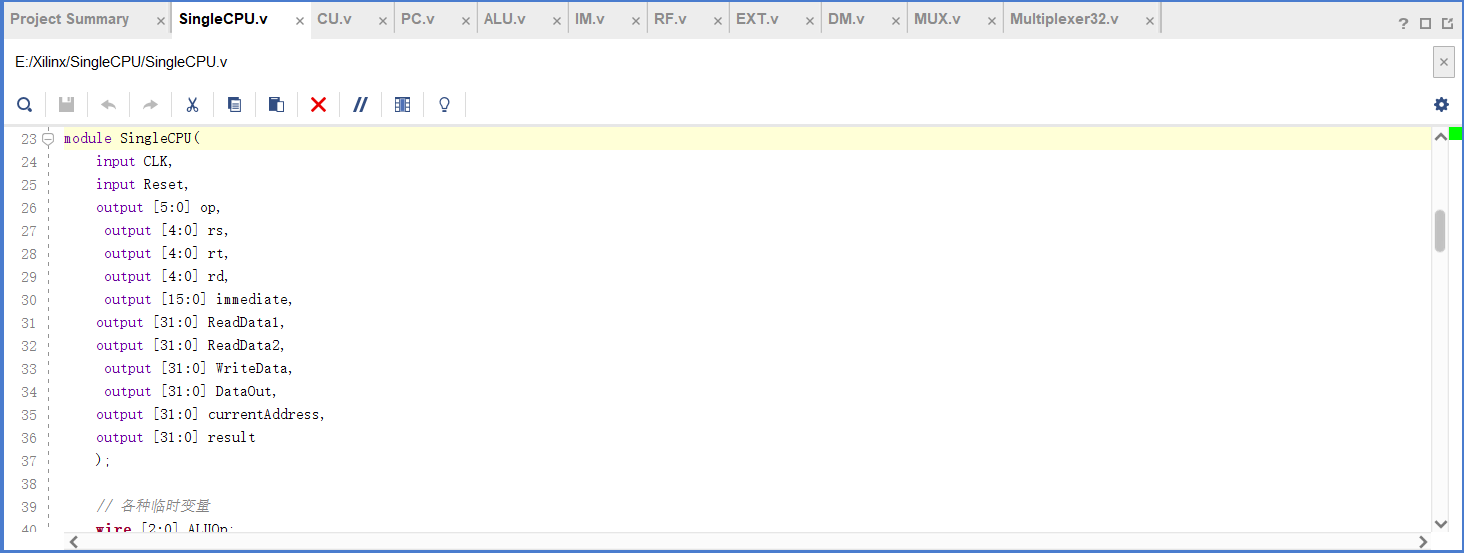
**ControlUnit：控制信号模块，通过解析op得到该指令的各种控制信号**

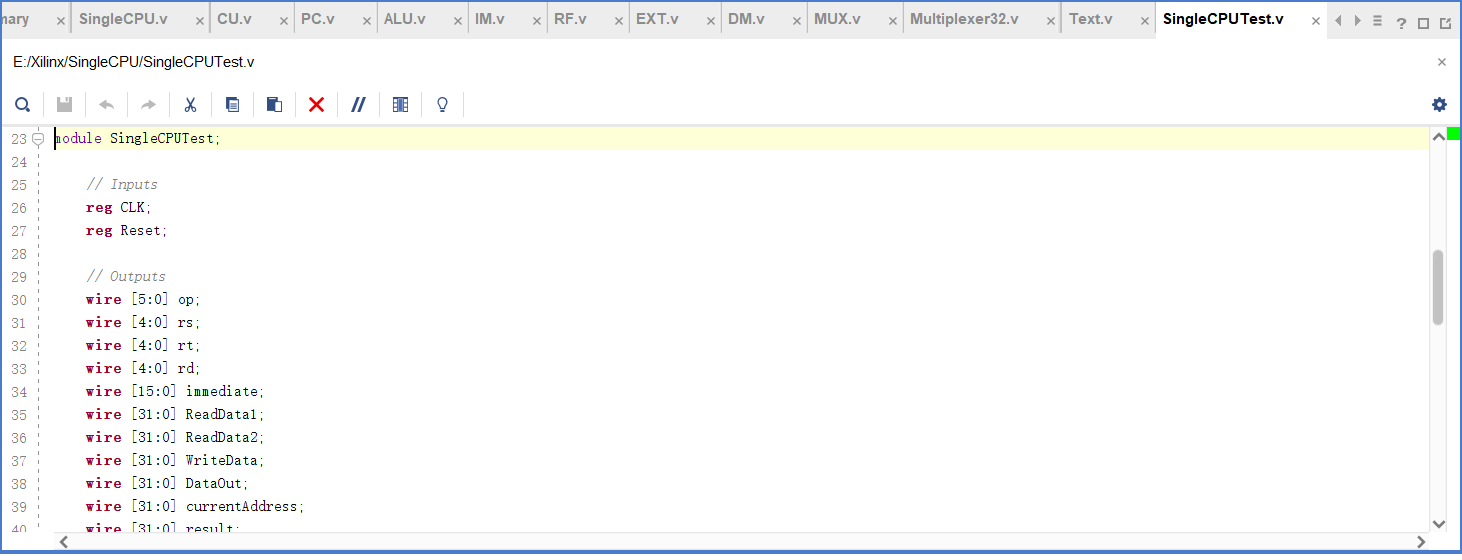
输入：op，zero   
输出：各类控制信号   
解释：通过上表，可以将每个指令case到相应的控制信号上。

首先，需要得到控制信号表： 通过信号控制表，可以很轻松的写控制模块。

然后，通过顶层模块，调用下层模块并将它们输入输出连在一起：

**SingleCPU：顶层连接模块**

该模块只需要输入输出正确

最后，加入如下测试单元，即可进行仿真模拟：

使用的MIPS测试指令如下：

//addi $1, $0, 4

000001 00000 00001 0000000000000100

//addi $2, $0, 8

000001 00000 00010 0000000000001000

//sw $2, 0($2)

100110 00010 00010 0000000000000000

//add $3, $2, $1

000000 00010 00001 00011 00000000000

//sub $3, $3, $1

000010 00011 00001 00011 00000000000

//beq $2, $3, -2

110000 00010 00011 1111111111111110

//ori $1, $1, 1

010000 00001 00001 0000000000000001

//or $3, $2, $1

010010 00010 00001 00011 00000000000

//move $3, $2

100000 00010 00000 00011 00000000000

//and $1, $3, $2

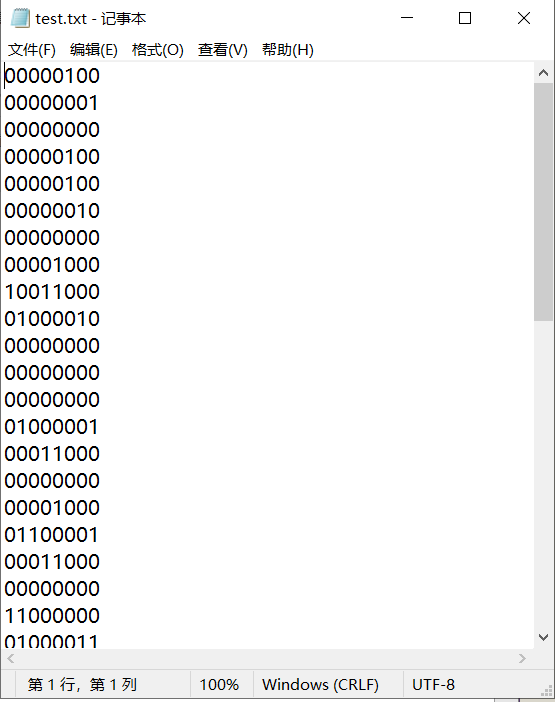
010001 00011 00010 00001 00000000000

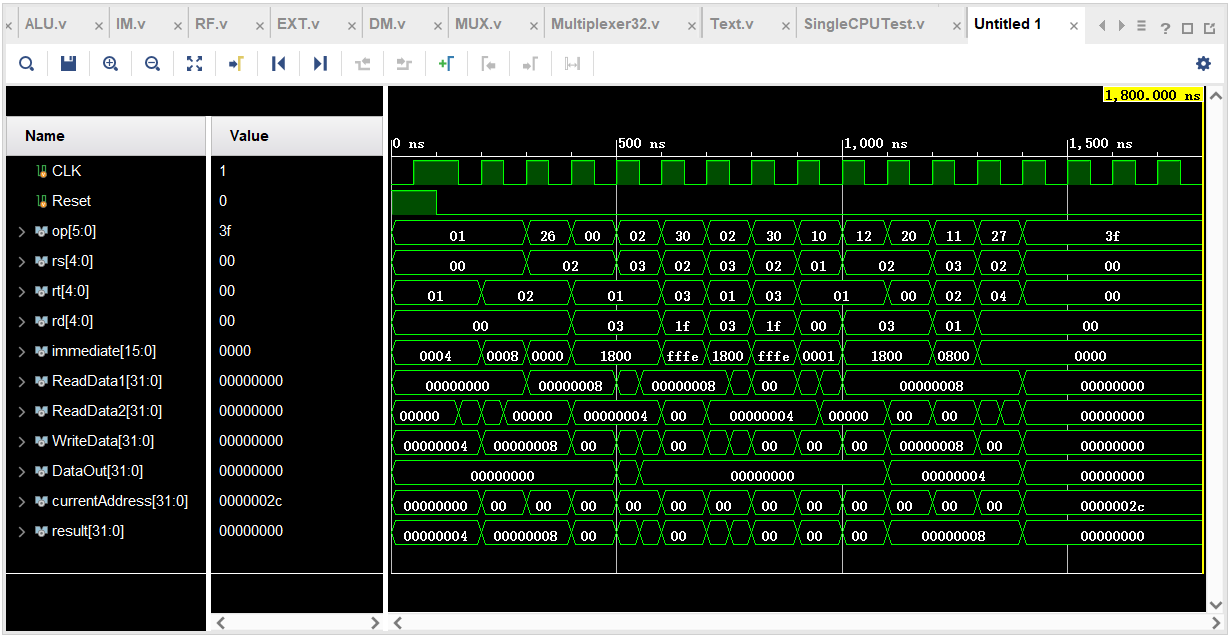
//lw $4, 0($2)

100111 00010 00100 0000000000000000

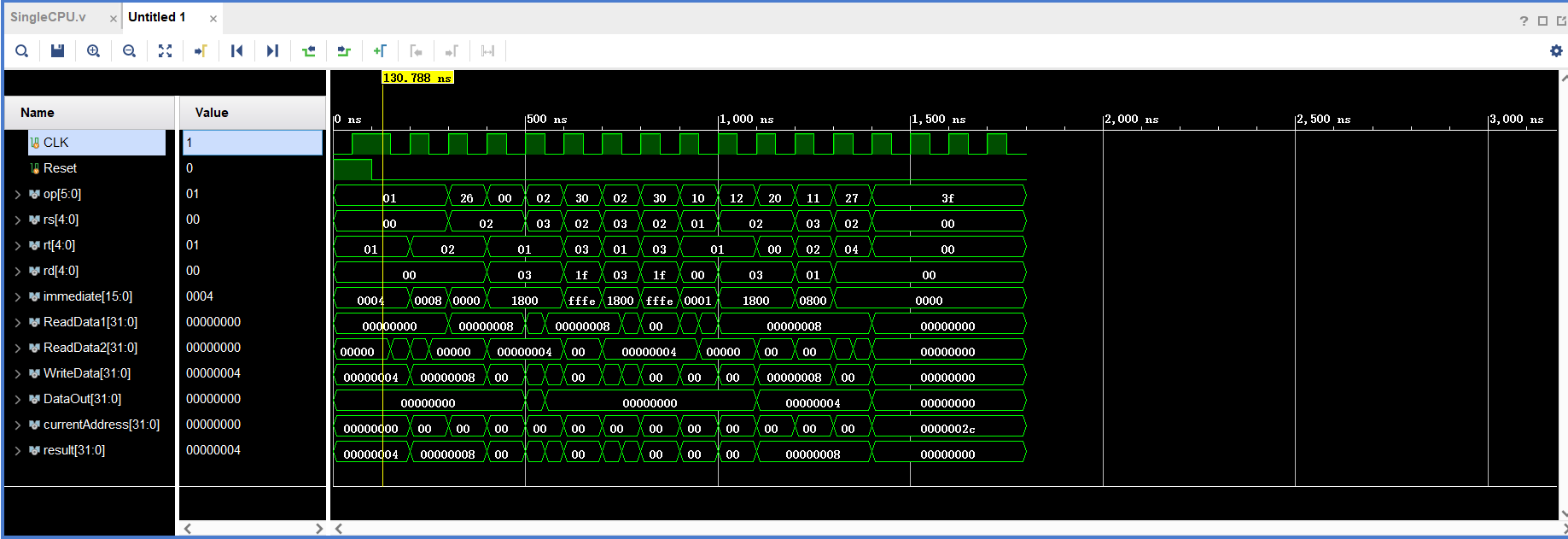
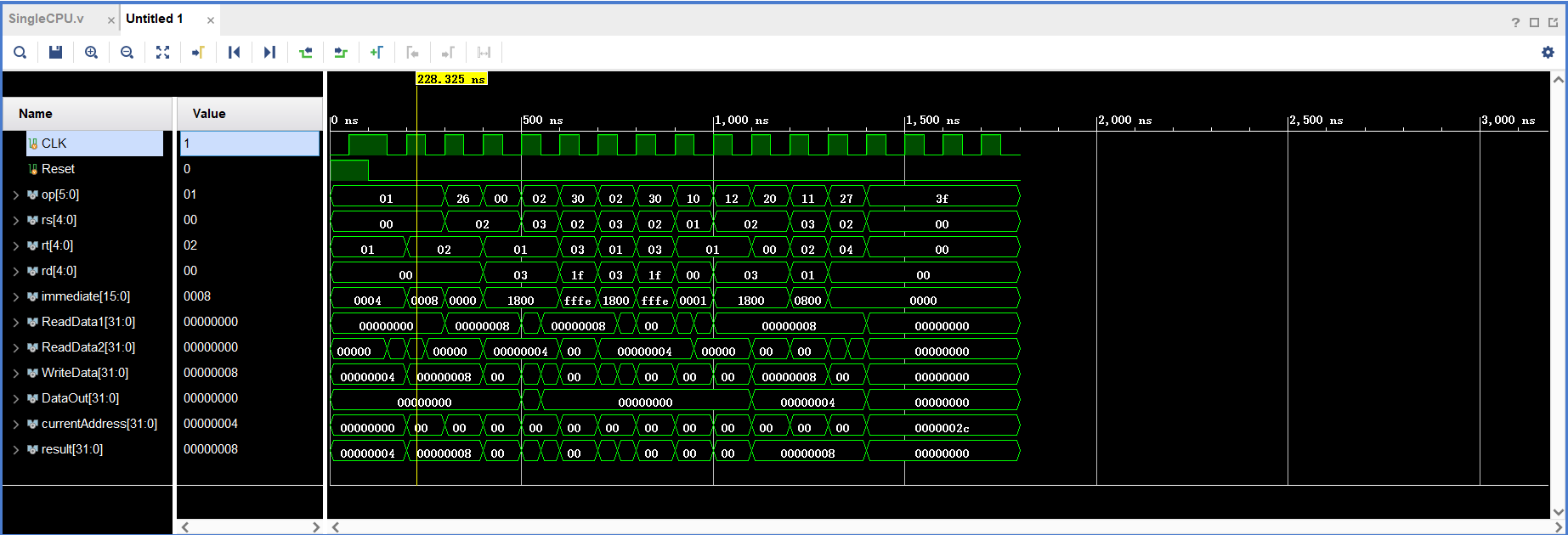
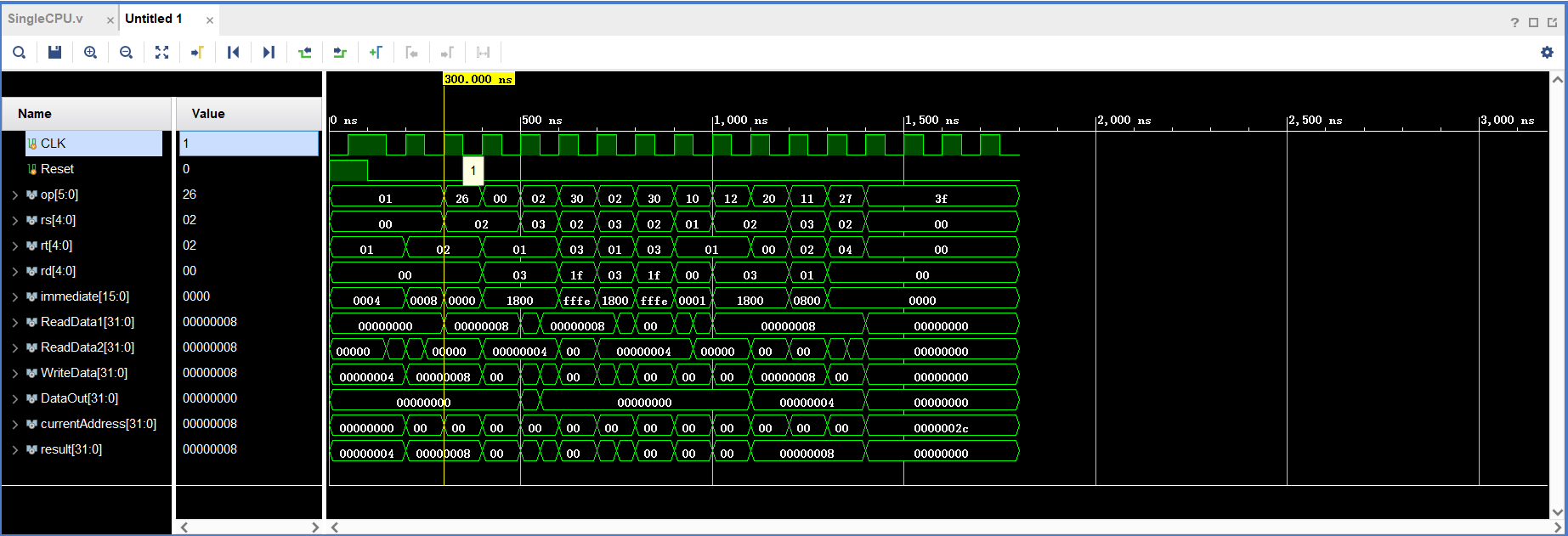
//halt

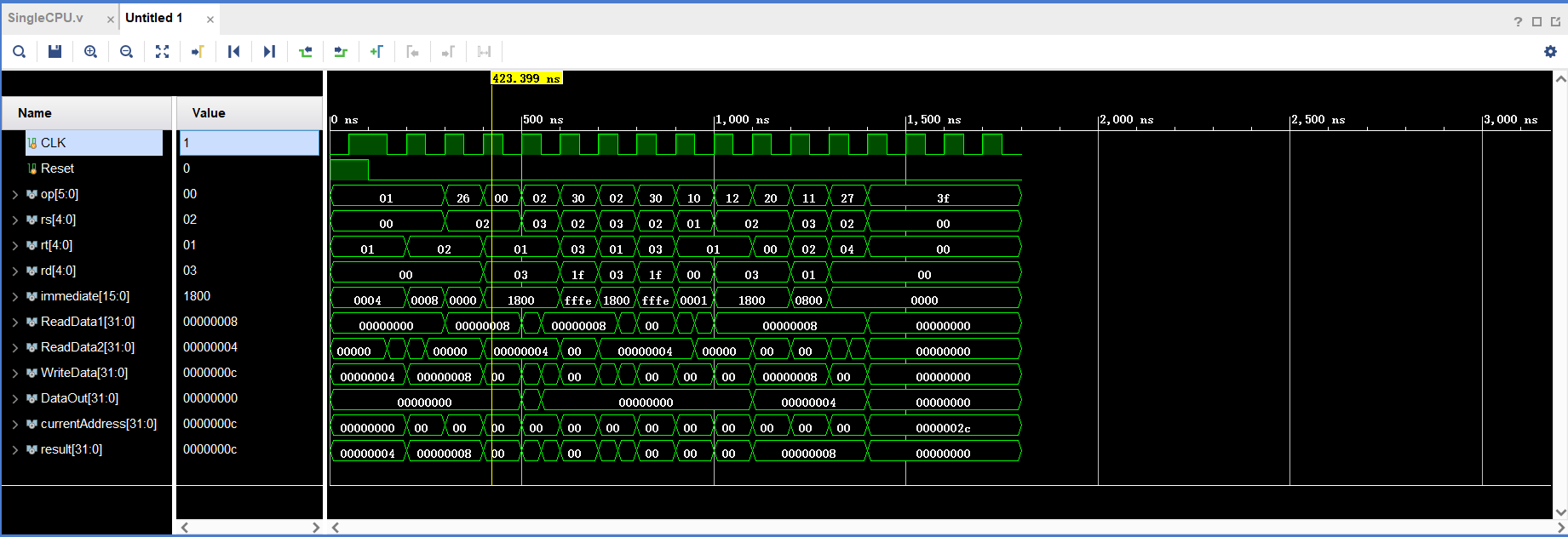
111111 00000000000000000000000000

储存路径为"E:/Xilinx/SingleCPU/test/test.txt"

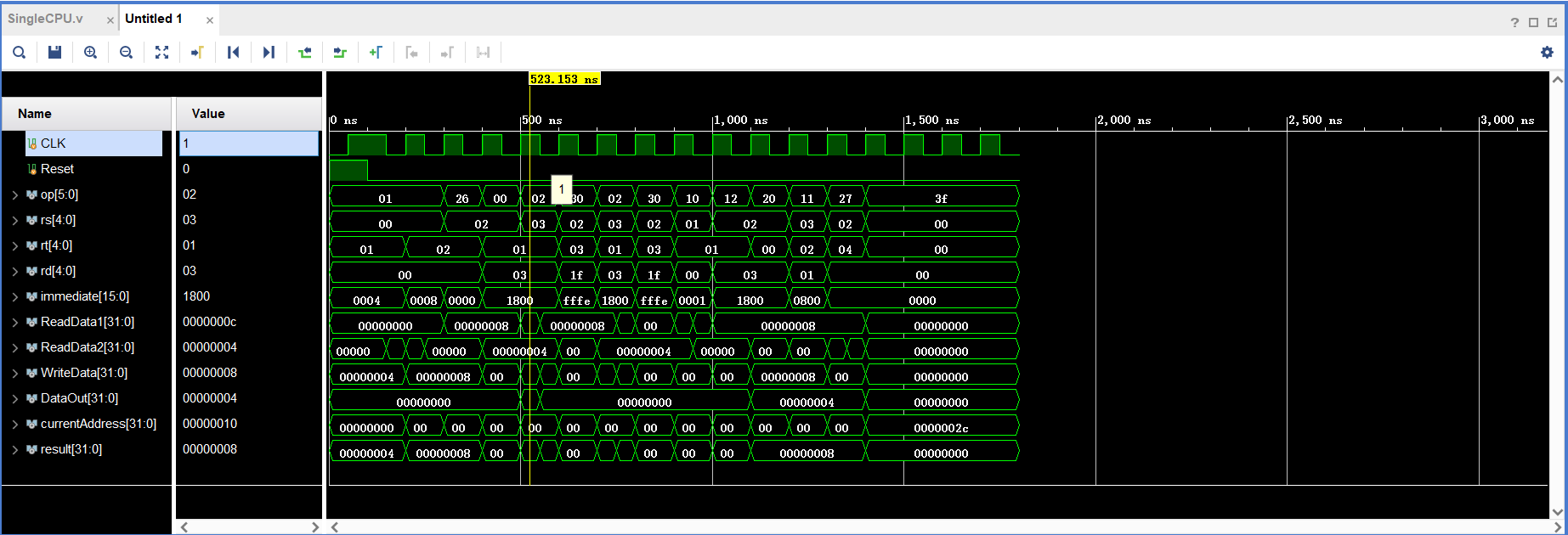
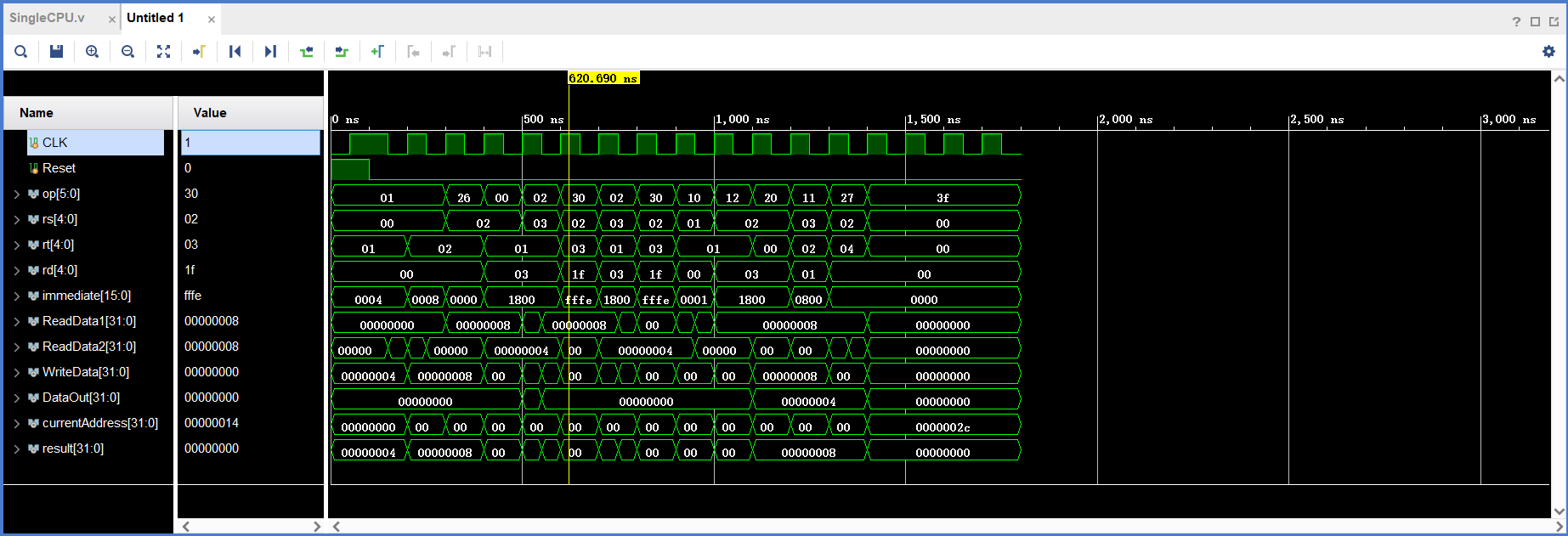
模拟结果和波形如下：

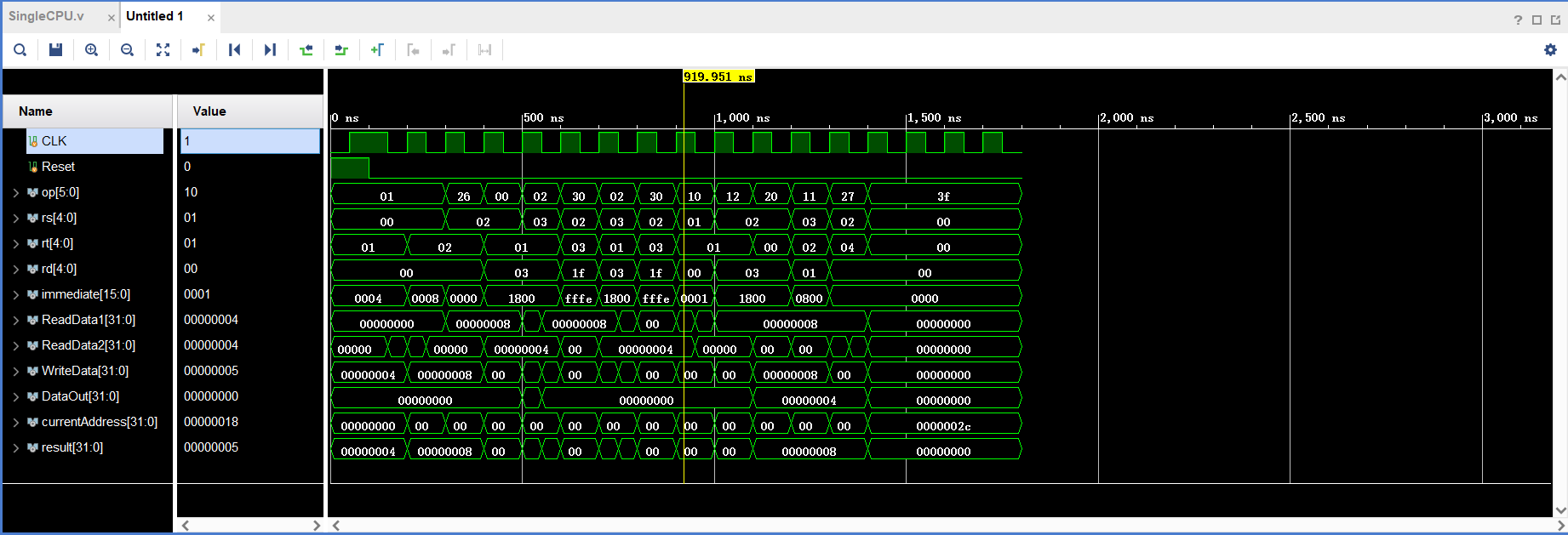
下面是对波形图的分析：

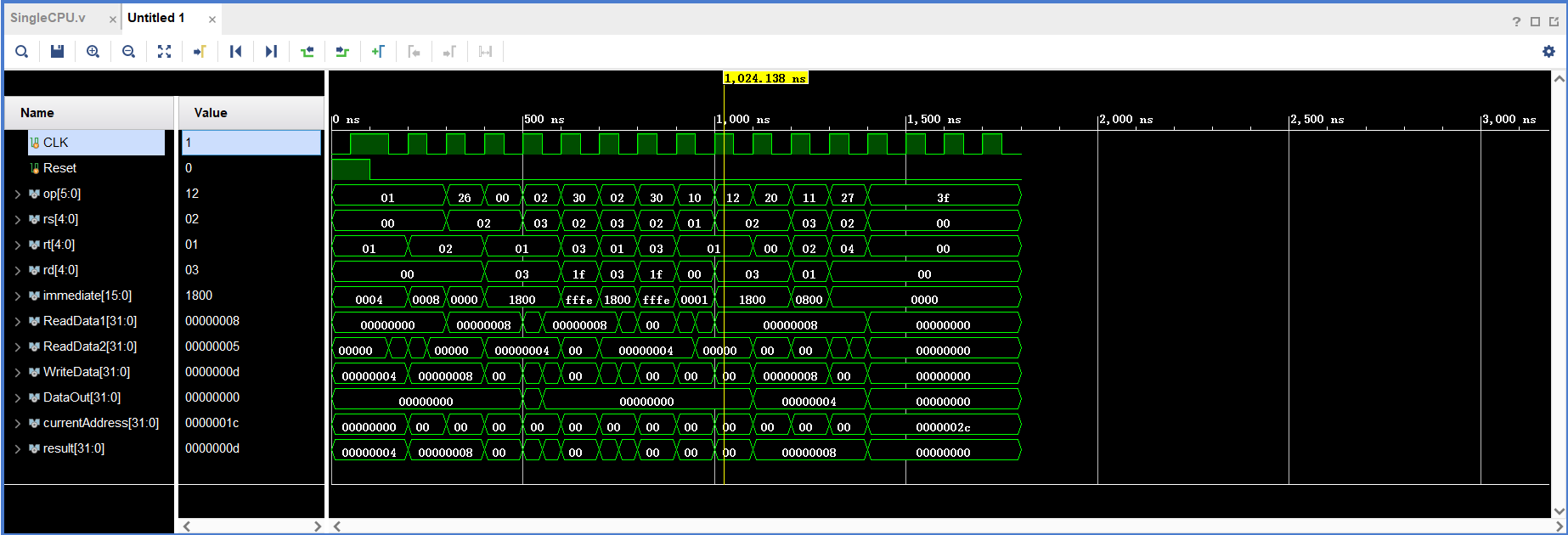
上图是addi $1, $0, 4, rt <- rs + (sign-extend)immediate ；其中rt=$1,rs=$0上图是addi $2, $0, 8, rt <- rs + (sign-extend)immediate ；

上图是sw $2, 0($2),  memory[rs + (sign-extend)immediate] <- rt ；rt=$1,rs=$2

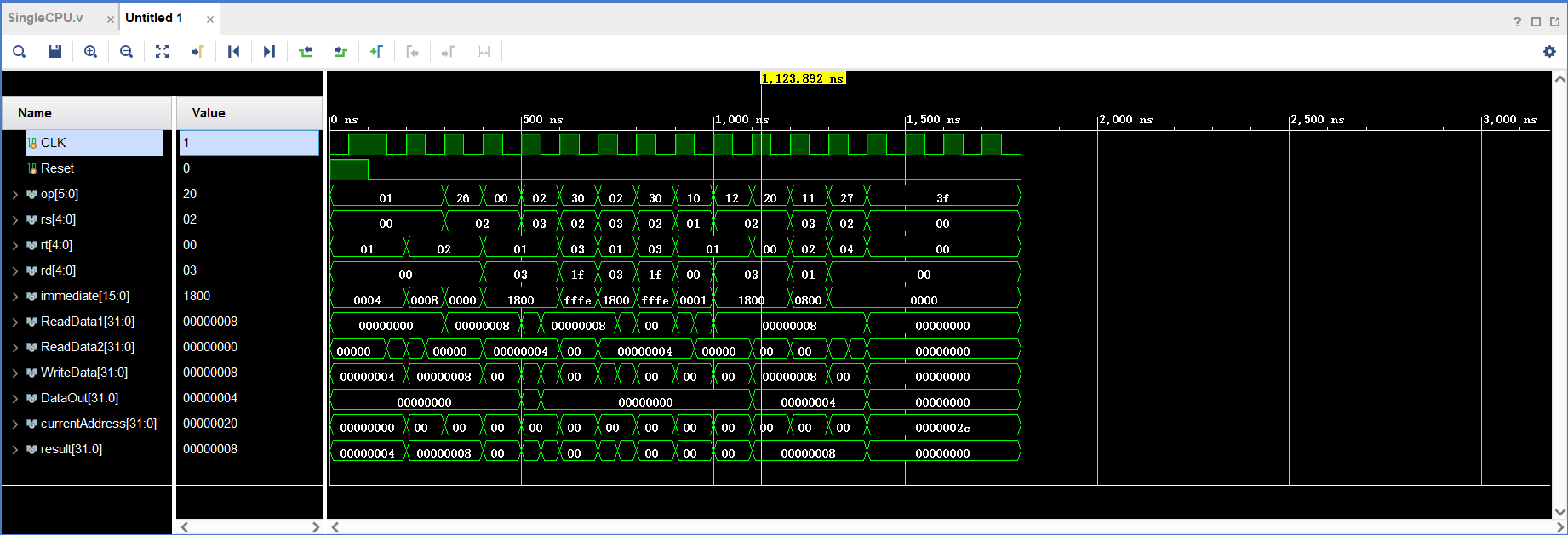
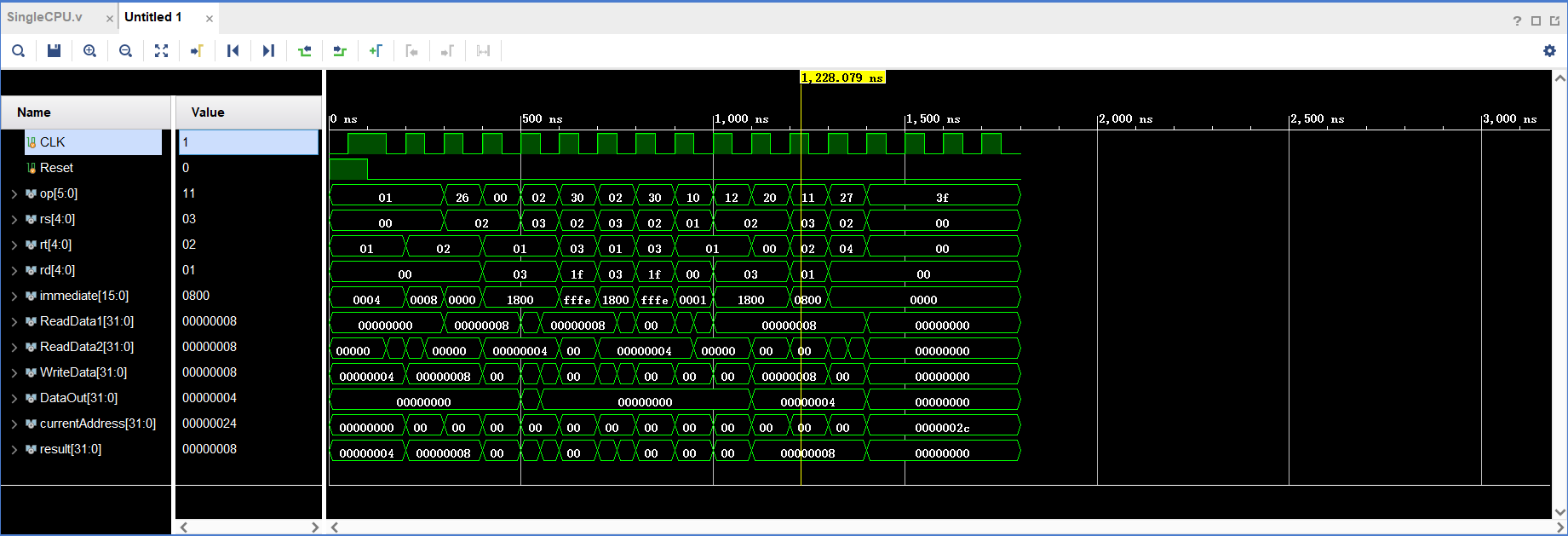
上图是add $3, $2, $1, rd <- rs & rt   ；其中rs＝$2，rt=$1, rd=$3；

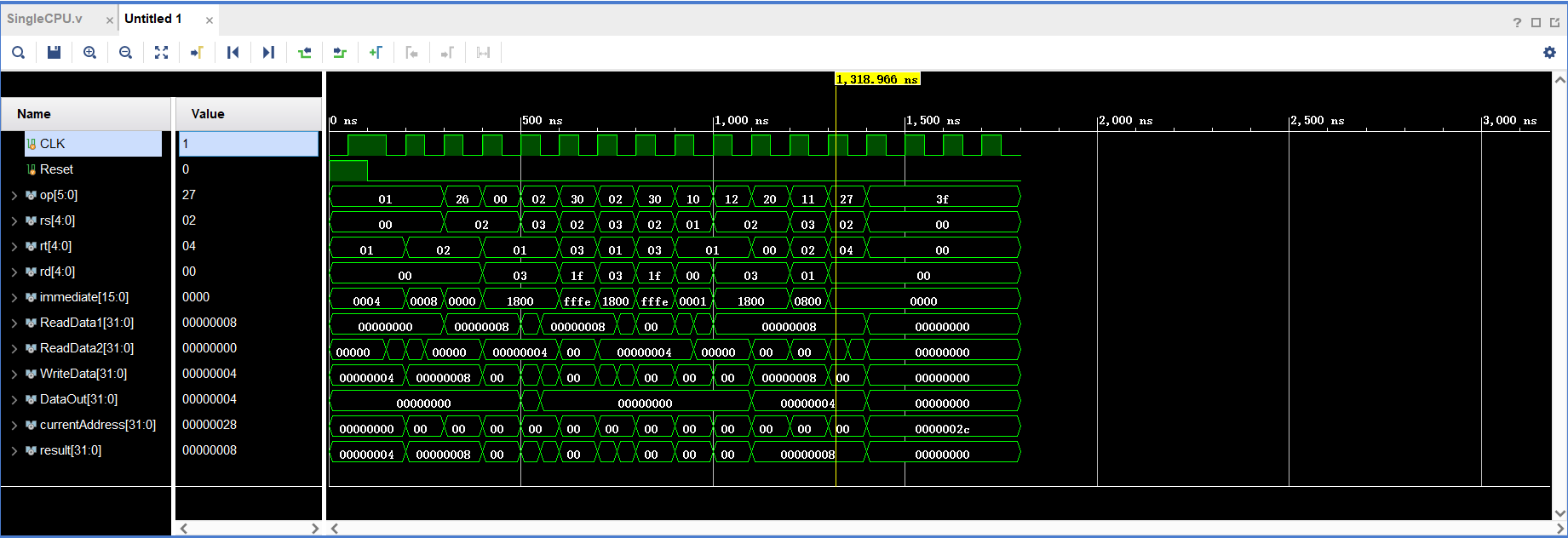
上图是sub $3, $3, $1,  rd <- rs - rt   ；其中rs＝$3，rt=$1, rd=$3

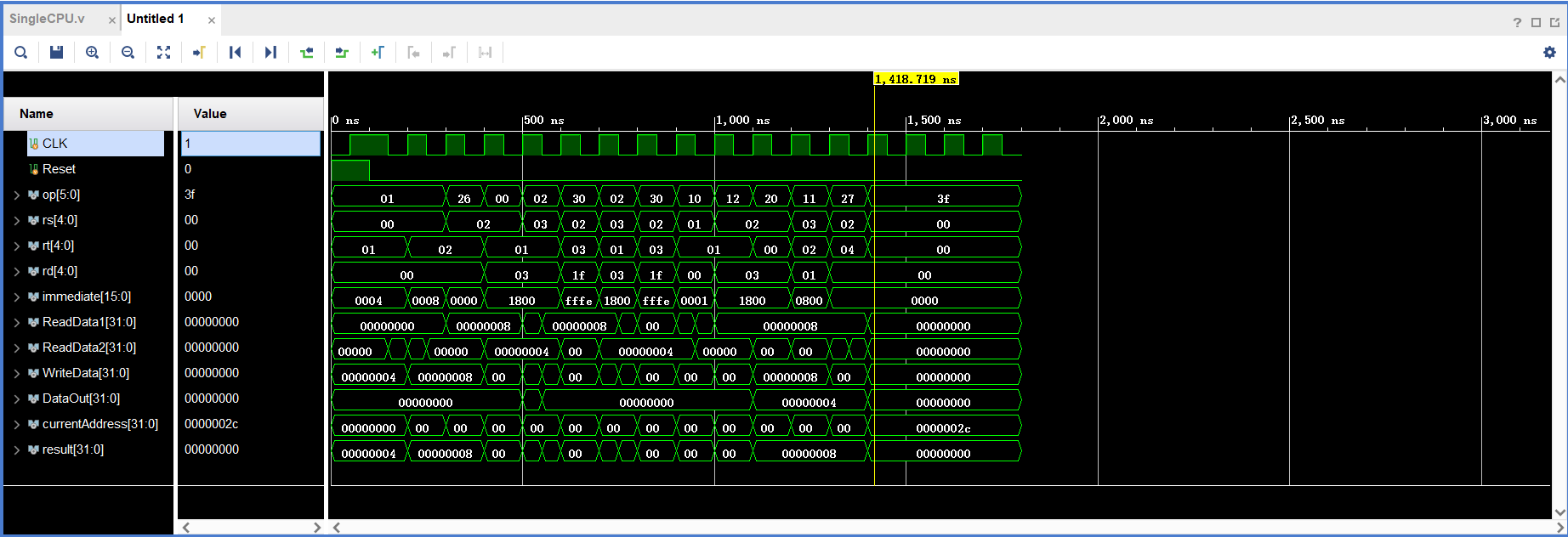
上图是beq $2, $3, -2,  if (rs == rt) PC <- PC+4 + (sign-extend)immediate<<2

上图是ori $1, $1, 1, rd<-rs|im

上图是or $3, $2, $1,  rd <- rs | rt   ；其中rs＝$2，rt=$1, rd=$3

上图是move $3, $2, rt <-rs;

上图是and $1, $3, $2,  rd <- rs & rt   ；其中rs＝$3，rt=$2, rd=$1

上图是lw $4, 0($2),  rt <- memory[rs + (sign-extend)immediate] ；

上图是halt,停机！

1. **总结**

整个实验涉及大量的新知识，下面对学到的新知识进行一次总结归纳：

1. **wire与reg**

在设计中，输入信号一般来说你是不知道上一级是寄存器输出还是组合逻辑输出，那么对于本级来说就是一根导线，也就是wire型。而输出信号则由你自己来决定是寄存器输出还是组合逻辑输出，wire型、reg型都可以。但一般的，整个设计的外部输出（即最顶层模块的输出），要求是寄存器输出，较稳定、扇出能力也较好。、

有几种情况变量需要定义成wire。  
第一：assign 语句  
例如：  
reg a,b;  
wire and\_result;  
...  
assign and\_result =a&&b;  
可以试试把wire定义成reg。综合器会报错。  
第二：元件例化时候的输出必须用wire  
例如：  
wire dout;  
ram u\_ram  
(  
...  
.out(dout)  
...  
);

wire表示直通，即只要输入有变化，输出马上无条件地反映；reg表示一定要有触发，输出才会反映输入。在verilog里面，变量有wire和reg两种类型，wire类型意为线，它不可储存中间结果，通过输入得出输出，类似纯函数，只要输入变化，输出立即变化，如果没有输入，自然就没有输出。reg类型意为寄存器，它可以赋初值，可以储存中间结果，只有当满足某种条件时(比如时钟上升沿)，它才会变化，其他时间会一直保持最后一次变化的值

**2.initial（多用于仿真赋值）**

initial块内的语句从上到下只执行一次不符合电路的运行模式，所以是不可综合的 它一般用来仿真时赋值；你这个代码里面的initial就是用来给输入赋值测试你这个全加器的。。

**3.always**

always块内的语句是反复执行的，一般写成类似于 always @ (posedge clock or negedge reset) 或 always @ (\*)的形式，写成时序或组合逻辑，

一般always@(\*)是指里面的语句是组合逻辑的。\*代替了敏感变量。意思是说敏感变量由综合器根据always里面的输入变量自动添加，不用自己考虑。而一般时序逻辑要写成always@(posedge clk or negedge rst)//[时钟信号](https://www.baidu.com/s?wd=%E6%97%B6%E9%92%9F%E4%BF%A1%E5%8F%B7&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)clk上升沿或者[复位信号](https://www.baidu.com/s?wd=%E5%A4%8D%E4%BD%8D%E4%BF%A1%E5%8F%B7&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao" \t "_blank)rst下降沿的时候执行always块内的代码。always @( ) 是一种固定用法，括号内填写行为触发条件。

always @(a, b, s)中，括号里面的输入信号a，b，s表示敏感信号；这句表示，敏感信号列表中的任何一个信号发生变化，将会引发 begin …… end 之间的行为。 Verilog用begin和end包围代码段，相当于c语言中大括号{ }的功能。

例如：

input clk;//时钟信号

reg a,b,c;

//触发动作：时钟上升沿

always@(posedge clk)

begin

b = a;

c = b;

end

这里的“=”，叫做**“阻塞赋值”**，和C语言里的赋值语句意思差不多，使用也差不多。 比如,有这么一段代码：那么时钟上升沿出现后，c的值就等于a，b的值也是a，这个行为在描述的时候，语句先后顺序，决定了赋值的先后，值是立即更新的。

always@(posedge clk)

begin

b <= a;

c <= b;

end

**“非阻塞赋值”**，表示方法是 <=，这种赋值称为“并行赋值”，在always@ 代码块内书写时，没有先后顺序。一般用于设计时序逻辑电路，举个例子：含义：时钟上升沿出现前a,b的值，在上升沿结束后分别赋给b,c.

非阻塞赋值“<=”符号，  
右侧所有变量，代表的是这些变量上升沿前的值，  
左侧变量表示，上升沿后该变量将要更新为右侧表达式的值。

所以，这两句书写没有先后顺序，调换次序后表达的含义是一样的，新寄存器的值是在always块结束后同时更新的。

@(posedge clk)

指clk上升沿到来时执行一次。posedge表示上升沿（positive edge），下降沿（negative dege）则使用negedge表示。

如果给这个计数器加一个异步复位信号，那么就在敏感信号列表里面加上它：

reg [4:0] count;

always @(posedge Clk or posedge Reset)

begin

if (Reset == 1) count <=0 ;

else count <= count + 1'b1;

end

这是Verilog描述异步时序逻辑电路的基本格式。  
在always块内判断异步信号时，如果异步信号高电平有效，Reset必须使用上升沿posedge修饰；反之，低电平有效用negedge，同时将 if (Reset == 1)改为 if (Reset == 0)。要注意的是，异步信号必须在进入always语句块后，立即进行判断。  
在敏感信号列表中将（posedge Reset）去掉后，电路仍然有复位功能，只不过变成了同步复位。

最后提一句：

阻塞和非阻塞赋值，都只能写在always@()代码块内。  
assign后面的=是和assign一起连用的，并不是阻塞赋值。

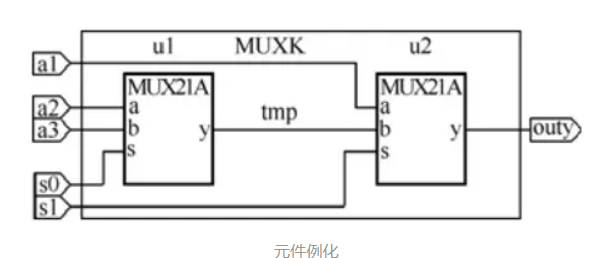
**4. verilog中，像2`B01 、 4`H0 、 16`H4012 之类的是什么意思？**

2.4.16表示位宽。计算位宽时，都要转成二进制来计算，因为时序元件只能存储二值电平。  
对于4'h0，由于是这里的0是十六进制的，它表示二进制的4'b0000，所以是四位的。

对于16'h4012，每个数字表示4位，一共4个数字，所以一共是4＊4＝16位。

我的理解是前面一个数字代表总共多少bit。然后b，h代表用二进制，十六进制来表示，后面是实际的用多少进制来表示的数据。

**5.元件例化（用于CPU设计中顶层模块的实现）**



元件例化是指在一个module中调用了其他的module。  
例如，上图中模块MUXK调用了两个MUX21A模块u1和u2。实现方式如下：

module MUXK(input a1,a2,a3,s0,s1,output outy)

wire tmp;

MUX21A u1(.a(a2), .b(a3) , .s(s0), .y(tmp));

MUX21A u2(.a(a1), .b(tmp) , .s(s1), .y(outy));

endmodule

解释如下：  
1、要调用某个子模块，直接写上子模块名MUX21A，后面加上实例名u1，再后面括号内是子模块引脚信号与当前顶层module内信号的连接关系。  
2、实例名u1，u2可以省略。  
3、子模块引脚名(.a .b .s .y)可以省略，条件是括号内被连接的信号排列顺序，需要与子模块定义时的引脚名保持一致。  
4、如果连接线tmp是单根信号线，可以省略不定义。  
即，简略版的描述可以这样：

module MUXK(input a1,a2,a3,s0,s1,output outy)

// wire tmp;

MUX21A (a2, a3 , s0, tmp);

MUX21A (a1, tmp, s1, outy);

endmodule

需要注意的事项：虽然上面说了一些可以省略的内容，但为了保证便于调试，以及防止因省略产生歧义，强烈建议不要省略。

**6.指令怎么来？**

在IM组件和RW组件分别开两个寄存器数组，用来模拟指令内存和数据内存，通过文件读取，从test.txt(test文件夹中)读指令到IM的指令内存(从0开始)，IM组件通过输入的IAddr(数组地址下标)，得到相应的指令。

**7.指令怎么变化？**

在PC端，有两个外部输入：CLK和Reset。其中PC内部有指令寄存器，每次CLK上升沿触发后，会改成新的指令，同时，当Reset=1时，指令寄存器也会置0。

**8.模块和模块间怎么连接？**

此时，需要一个顶层模块，相当于main函数，它会根据数据通路图，将一些下层模块的输出，连接到另一些下层模块的输入中。

**9.写好的cpu怎么运行？**

需要在顶层模块再添加一个测试文件，测试文件提供外部输入(CLK和Reset)，然后模块就会自动运行得到相应的仿真结果。