**计算机体系结构实验课程第一次实验报告**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 实验名称 | 多周期CPU | | | 班级 | 李雨森班 |
| 学生姓名 | 姚知言 | 学号 | 2211290 | 指导老师 | 董前琨 |
| 实验地点 | 实验楼A306 | | 实验时间 | 2024年9月9日 | |

1. **实验目的**

在单周期 CPU 实验完成的提前下，理解多周期的概念。

熟悉并掌握多周期 CPU 的原理和设计。

进一步提升运用 verilog 语言进行电路设计的能力。

为后续实现流水线 cpu 的课程设计打下基础

1. **实验内容说明**

复习单周期 CPU 的实验内容，归纳常用的 MIPS 指令，确定自己准备实现的 MIPS 指令，对其进行分析；

依据自己设计中实现的指令，编写一段不少于 40 行的汇编程序，要求包含所有实现的指令；

认真学习多周期的概念，了解流水线的概念，明白划分为多周期的意义；

认真学习 CPU 各模块的功能，确认模块的划分。设计本次实验的方案，画出实验方案的设计框图；

确定设计中与 FPGA 板上交互的接口，画出包含外围模块的整体设计框图；

确认多周期 CPU 的设计框图的正确性；

编写 verilog 代码，将表 8.2 中自己编写的汇编程序翻译为二进制，以coe 文件的方式初始化到指令 ROM 中；

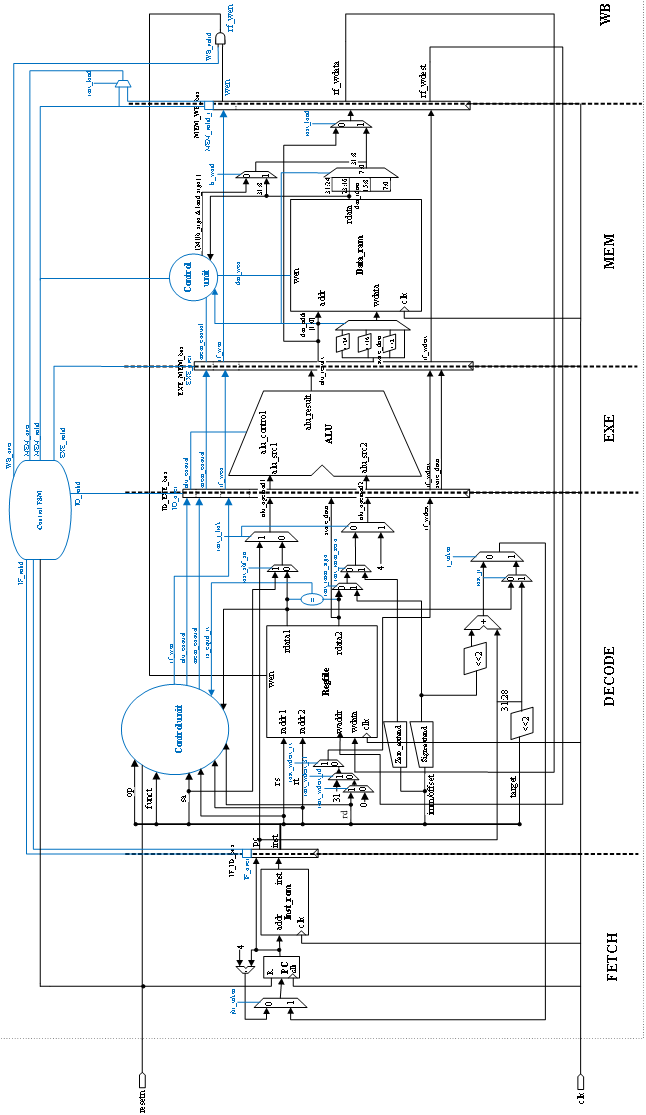
对该模块进行仿真，得出正确的波形，截图作为实验报告结果一项的材料；

完成调用多周期 CPU 的外围模块的设计，并编写代码；

对代码进行综合布局布线下载到实验箱里 FPGA 板上，进行上板验证。

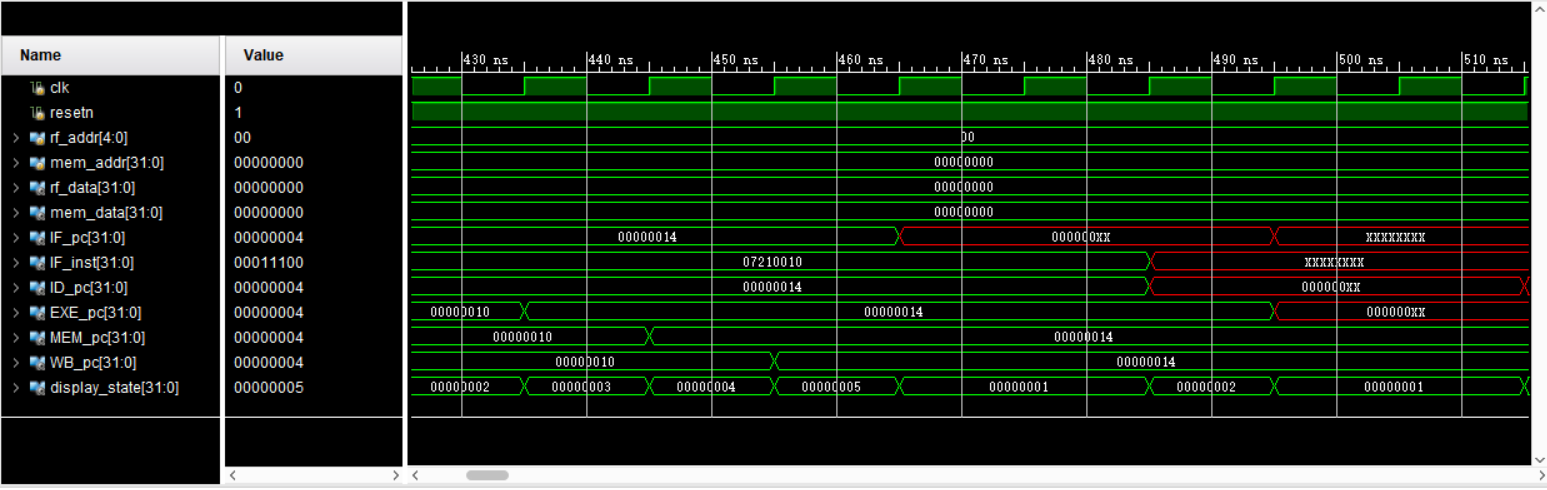
实验结束后，需按照规定的格式完成实验报告的撰写。

1. **实验原理图**



1. **实验步骤**

（1）代码修改



通过会仿真结果的观察，很容易发现在执行第0x14的语句后，取指发生了一些问题，也就是在执行bgez $25,#16跳转后，CPU应当从译码阶段返回取指阶段。

观察源代码，可以发现其他四条总线都通过触发器保存在了寄存器中，想到可能是总线的保存存在问题，于是想到设置一个jbr\_bus\_r，通过jbr\_bus\_r <= jbr\_bus维护。

重新仿真，此时全部仿真都变绿了，但是发现在0x14之后就变成了0x18，并没有进行跳转，意识到触发器的保存会慢一个周期，因此需要一个办法等待指令到达后再在fetch中读取。

此时，注意到原本的图中fetch阶段是经过两个时钟周期的，想到可以借鉴这个设计，将decode也改为两个周期，并将触发器的存储改在ID\_valid阶段（ID\_over比ID\_valid慢一个时钟周期），这样就可以完成触发器的更新了。

//decode.v

always @(posedge clk)

begin

ID\_over <= ID\_valid;

end

(此时还需要为decode.v增加一个输入参数clk，并在multi\_cycle\_cpu中同步进行模块调用的修改，否则将会缺少时钟周期的参数报错）

//multi\_cycle\_cpu.v

always @(posedge clk)

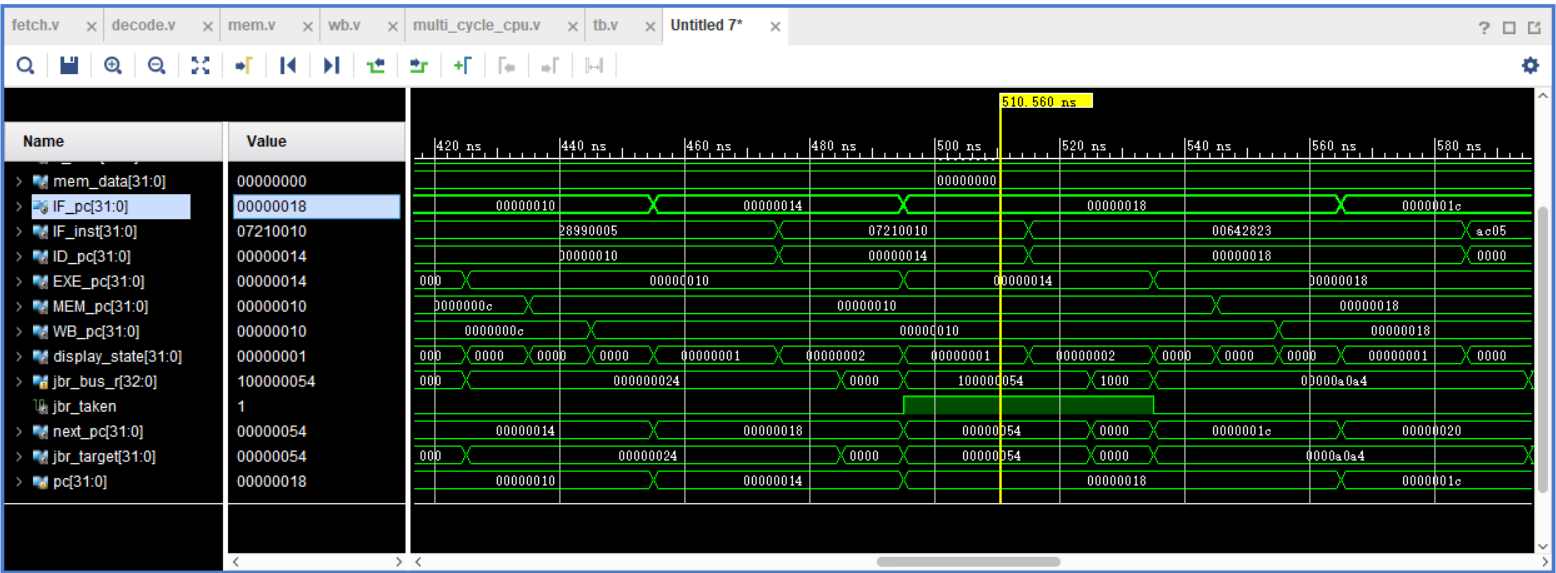
if(ID\_valid)

begin

jbr\_bus\_r <= jbr\_bus;

end

此时仍然不能实现预期结果，观察下图pc，jbr\_target结果可以发现，jbr\_target看起来没什么问题，但是pc却没有得到更新，意识到更新pc的时间可能存在问题。



//fetch.v

always @(posedge clk) // PC程序计数器

begin

if (!resetn)

begin

pc <= `STARTADDR; // 复位，取程序起始地址

end

else if (next\_fetch)

begin

pc <= next\_pc; // 不复位，取新指令

end

end

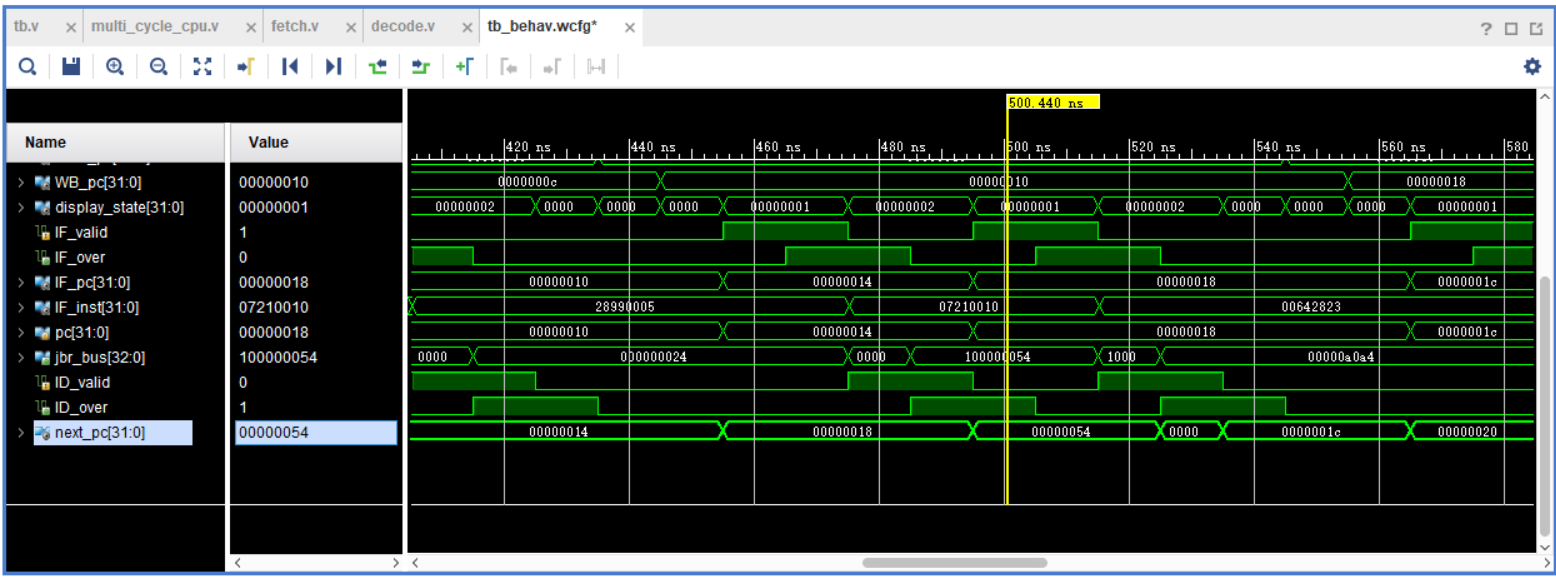
从中可以发现，设置合理的next\_fetch是解决问题的关键。

起初设想过，将next\_fetch去掉，直接在IF阶段开始的时候（第一次失败分析原因后，还为jbr总线增加了resetn初始值）读取指令，但是引发了更严重的问题（主要是对于经过5个阶段后跳转的指令出现了问题）。

后放弃该设想，转而调整next的判定条件，然而无论是将ID\_over调整为ID\_valid还是保持不变，都无法在规定时间上完成更新。

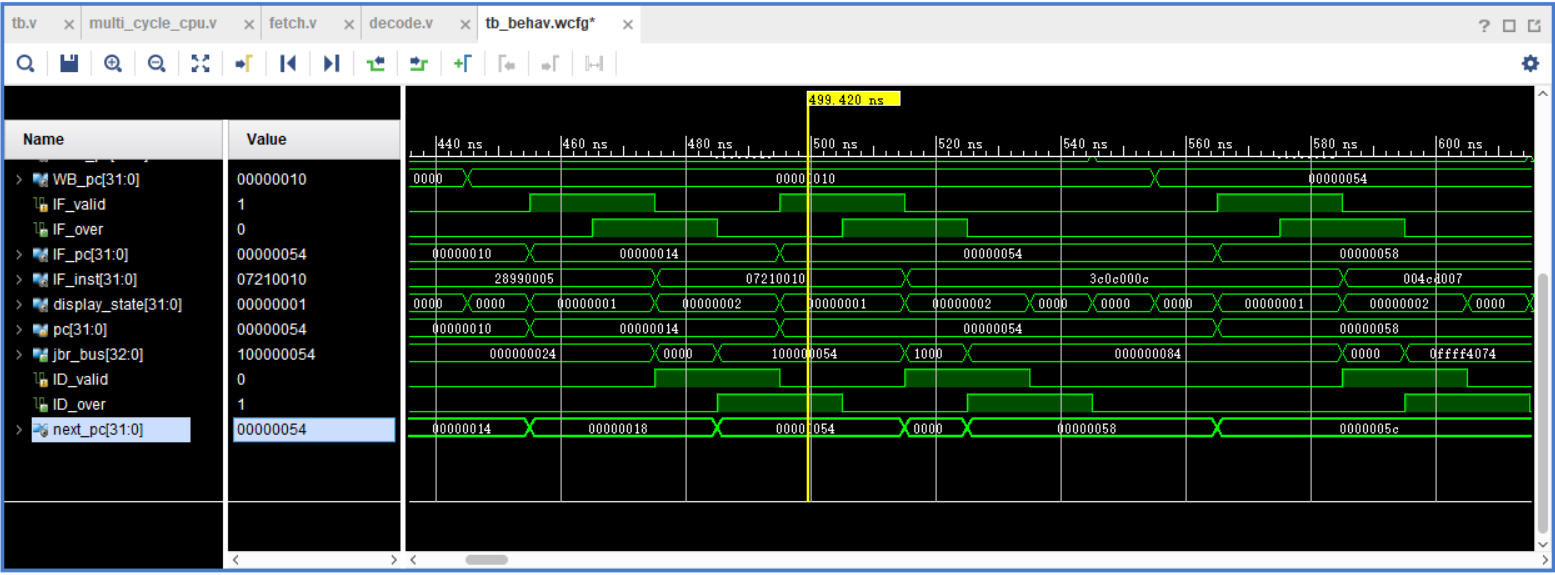
assign next\_fetch = (state==DECODE & ID\_over & jbr\_not\_link )

| (state==WB & WB\_over);



从图中可以看到，虽然next\_pc能及时完成更新，但始终慢上一个始终周期，这是0x18已经进入取指阶段了，并不合理。

最终想到，如果不使用寄存器而使用wire完成更新，就不存在这个问题了，于是删除jbr\_bus\_r，回调传入fetch的参数。（可能另一种解决方案是把decode改为3周期保障更新，但没有必要）



由此，next\_pc可以和ID\_over在同一时间周期完成更新，保证了PC在取指之前的顺利更新，完成修改。

修改的全部代码如下：

//decode.v

`timescale 1ns / 1ps

module decode( // 译码级

input clk,

input ID\_valid, // 译码级有效信号

input [ 63:0] IF\_ID\_bus\_r, // IF->ID总线

input [ 31:0] rs\_value, // 第一源操作数值

input [ 31:0] rt\_value, // 第二源操作数值

output [ 4:0] rs, // 第一源操作数地址

output [ 4:0] rt, // 第二源操作数地址

output [ 32:0] jbr\_bus, // 跳转总线

output jbr\_not\_link,// 指令为跳转分支指令,且非link类指令

output reg ID\_over, // ID模块执行完成

//增加一拍时延，保证数据传输

output [149:0] ID\_EXE\_bus, // ID->EXE总线

//展示PC

output [ 31:0] ID\_pc

);

......

//-----{ID执行完成}begin

//为保证数据传输，需要ID是两周期的

always @(posedge clk)

begin

ID\_over <= ID\_valid;

end

//-----{ID执行完成}end

......

endmodule

（2）指令ROM中指令执行过程分析

为区别原有注释，个人分析标蓝处理。

//multi\_cycle\_cpu.v

inst\_rom inst\_rom\_module( // 指令存储器

.clka (clk ), // I, 1 ,时钟

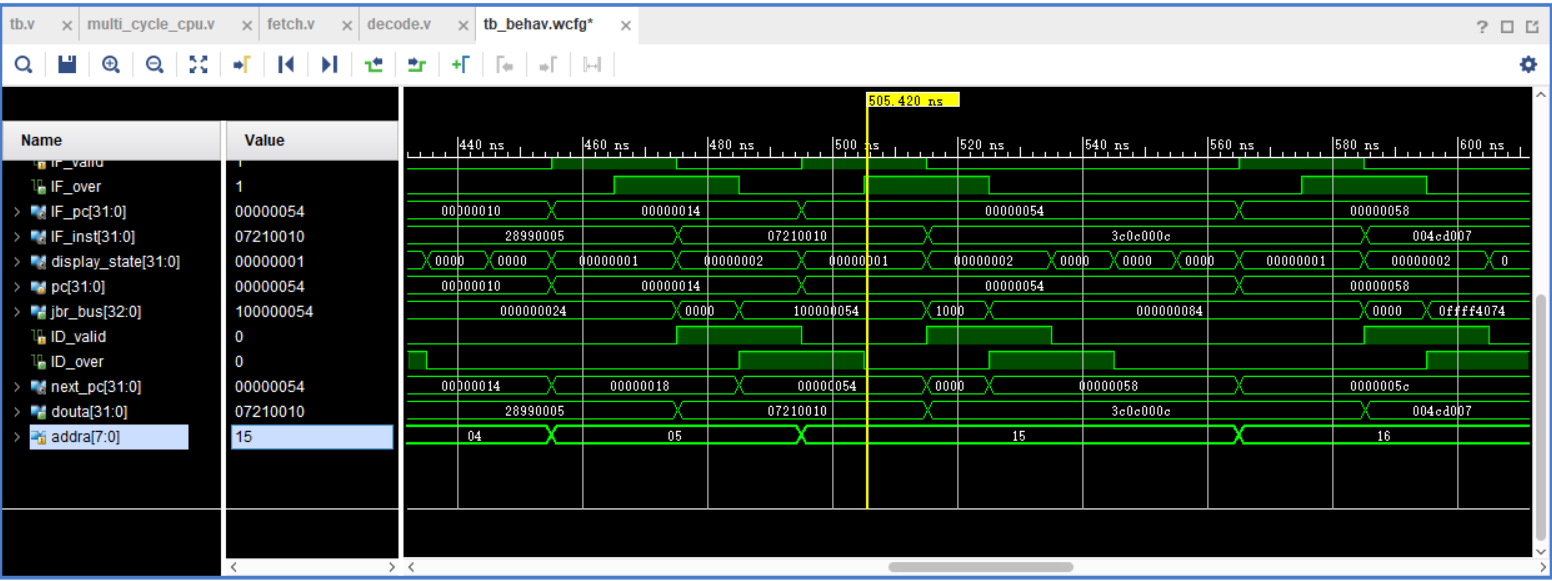
.addra (inst\_addr[9:2]), // I, 8 ,指令地址

.douta (inst ) // O, 32,指令

);

这里是指令ROM在上层模块multi\_cycle\_cpu中的调用，可以看出指令ROM读入的共有两个参数，clk时钟不必多说，addra 读取的值inst\_addr[9:2]指令地址，输出douta为inst指令。

到这里，其实已经较为明确了，由于一条指令的长度是4个字节，读取inst\_addr[9:2]即去掉后两位，对应的位置刚好是设计的.coe文件的行数，对应行数的指令返回输出就可以了。



对应display\_state来说，我们可以发现每一次addra的更新时间都是IF阶段开始的时刻，和PC的更新时间是同步的，具体PC的更新时间也就是上一部分中提到了next\_fetch的位置，在IF的前一个时钟周期开始，刚好在IF开始的时候完成更新，而对应的douta的更新时间是每次IF结束ID开始的时候。

也正是因为rom取指需要有一拍的时钟周期时延，取指阶段才需要2拍的时间。

//fetch.v

module fetch( // 取指级

input clk, // 时钟

input resetn, // 复位信号，低电平有效

input IF\_valid, // 取指级有效信号

input next\_fetch,// 取下一条指令，用来锁存PC值

input [31:0] inst, // inst\_rom取出的指令

input [32:0] jbr\_bus, // 跳转总线

output [31:0] inst\_addr, // 发往inst\_rom的取指地址

output reg IF\_over, // IF模块执行完成

output [63:0] IF\_ID\_bus, // IF->ID总线

//展示PC和取出的指令

output [31:0] IF\_pc,

output [31:0] IF\_inst

);

简单观察以下fetch的输入输出，fetch的输出inst\_addr对应的就是发往rom的取指地址，而最后输出的IF\_PC和它也没有任何差别，从赋值语句中就可以看出来。

//fetch.v

assign inst\_addr = pc;

......

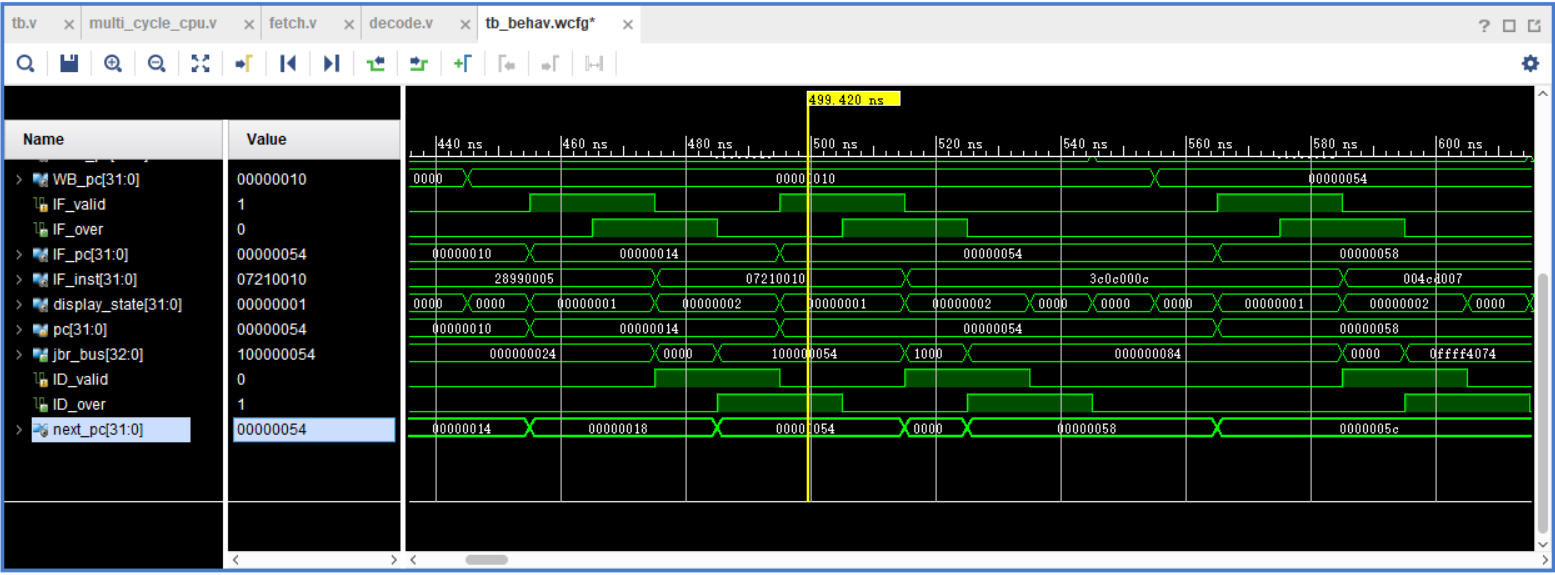
assign IF\_pc =pc;

assign IF\_inst = inst;

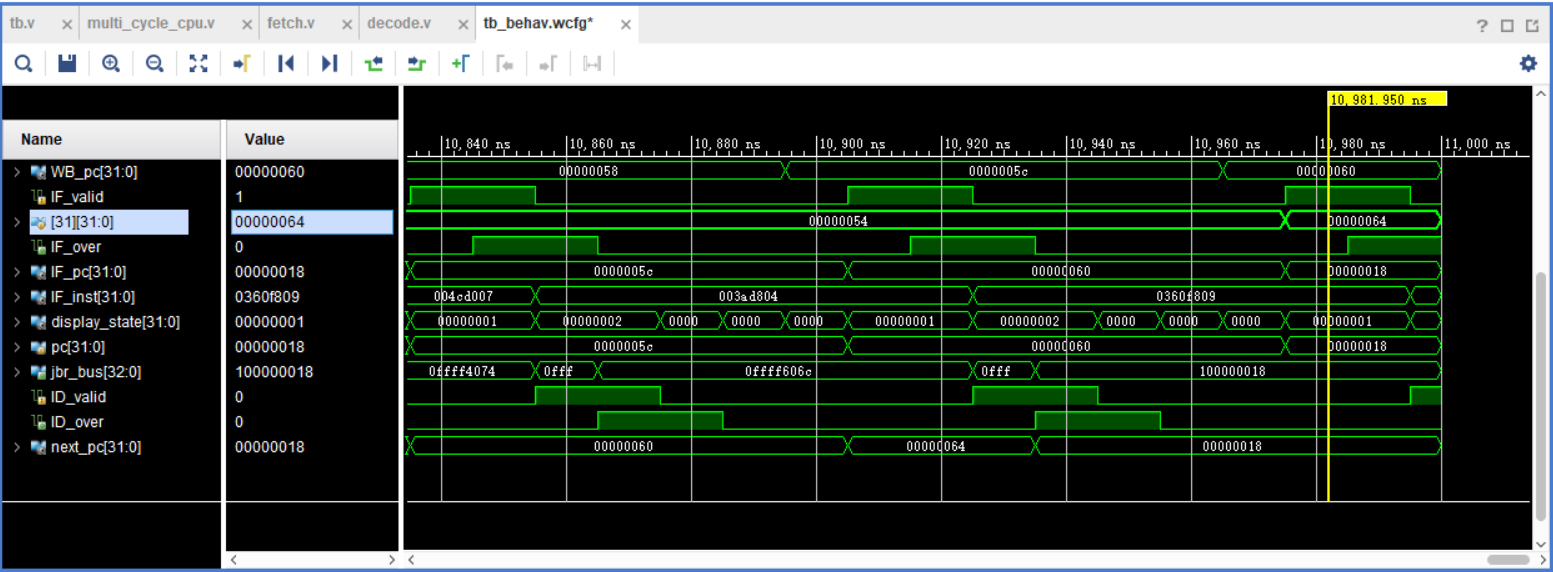
而fetch阶段最终返回的IF\_inst，也就是指令ROM的返回结果。

总的来说，fetch把pc参数带给指令ROM，ROM去掉后两位后存储文件.coe中找到对应的指令，最终再给到取指阶段打包。

1. **实验结果分析**



在0x14执行2周期后顺利跳转到0x54。 0x14的指令是bgez $25,#16，此时$25的值是1，大于等于0，向后跳转16个，刚好是0x54，完成了修改。



程序在11000ns中已经运行多轮，并没有发生崩溃。而且图中0x60的指令应该为jalr $27，实现效果是跳转到寄存器$27的数值（18H），完成了该操作（说明之前的修改并没有影响5周期跳转的实现）。同时查看图中高亮部分，可以发现$31确实被修改为0x64。

1. **总结感想**

在这次实验中，我完成了多周期CPU的理解，并进行了Bug的修复以及rom指令的介绍等。虽然在bug的修复上走了一些弯路，然后发现最终实际需要修改的代码量如此之少，但是在此过程中，我也更加对多周期CPU加深了一些理解。完成实验后我也曾设想过是否可以仅对这一种特殊指令增加时延，但考虑到后期将要实现流水线CPU，无论如何都是要迁就最长时间的阶段的，似乎纠结于这个问题的意义并不是很大。

无论如何，这次实验让我深刻熟悉了多周期CPU的架构，也为后续实验打下良好的基础。

注意： 1、班级用任课老师姓名表示。

2、实验报告提交的文件名为“学号\_姓名\_组成原理第一次实验.pdf”，注意要导出成pdf文件。