

**硬件课程设计（I）**

**实验报告**

**班 级 9191062301**

**学生姓名 孙傲歆**

**学 号 919106840333**

**题 目 多周期CPU的设计与实现**

**指导教师 杜姗姗**

**2022年9月**

# **目 录**

[**目 录** 2](#_Toc112229314)

[一、实验目的 3](#_Toc112229315)

[二、实验设备 3](#_Toc112229316)

[2.1 硬件设施 3](#_Toc112229317)

[2.2 开发环境 3](#_Toc112229318)

[三、实验任务 3](#_Toc112229319)

[四、多周期CPU框架设计 5](#_Toc112229320)

[4.1 总体框架设计 5](#_Toc112229321)

[4.2 顶层框架设计 6](#_Toc112229322)

[五、多周期CPU指令归纳表 7](#_Toc112229323)

[六、Verilog程序代码 9](#_Toc112229324)

[6.1 CPU顶层代码（multi\_cycle\_cpu.v） 9](#_Toc112229325)

[6.2 取指模块代码（fetch.v） 15](#_Toc112229326)

[6.3 译码模块代码（decode.v） 17](#_Toc112229327)

[6.4 执行模块代码（exe.v） 22](#_Toc112229328)

[6.5 访存模块代码（mem.v） 24](#_Toc112229329)

[6.6 写回模块代码（wb.v） 27](#_Toc112229330)

[七、结果展示 28](#_Toc112229331)

[7.1 测试所用汇编程序详述 28](#_Toc112229332)

[7.2 仿真波形图 29](#_Toc112229333)

[7.3 上板验证图 32](#_Toc112229334)

[八、心得总结 33](#_Toc112229335)

一、实验目的

1.在单周期CPU实验完成的提前下，理解多周期的概念。

2.熟悉并掌握多周期CPU的原理和设计。

3.进一步提升运用verilog语言进行电路设计的能力。

二、实验设备

2.1 硬件设施

1.装有Xilinx Vivado的计算机一台。

2.LS-CPU-EXB-002教学系统实验箱一套。

2.2 开发环境

语言：verilog

系统：win11

集成开发环境：Vivado 2019.2；Visual Studio Code

三、实验任务

1.本次实验是对单周期CPU实验的拔高，也是为流水线CPU打下基础。前期的实验准备同单周期CPU的实验，在单周期CPU中只要求实现了十几条指令，但此处要求扩展到30多条指令。

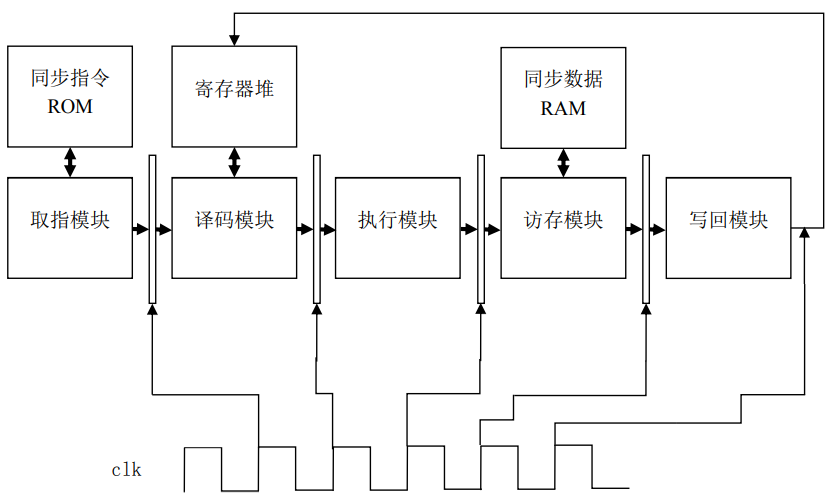
多周期CPU是指，一条指令需要花费多个周期才能完成所有操作，在每个周期内只做一部分操作，比如：取指、译码、执行、访存、写回，此时，一条指令执行完，共需5个周期，每个周期只做一部分操作。

将CPU划分为多周期的优势在于，每个时钟周期内CPU需要做的工作就变少，因此频率可以更高，且每个部件做的事情单一了，比如取指部件只负责从指令存储器中取出指令，因此CPU可以进行流水工作，也相当于一个时钟周期完成一条指令。频率更高，依然相当于是一个周期完成一条指令，因此CPU可以运行的更快。

本次实验就是将组成原理实验中实现的单周期CPU升级为多周期的，并扩展指令到30条以上。

2.依据单周期实验的设计框图，将其划分为多个功能块，每个功能块占用一个周期完成，即每个功能块从上一个功能块获取信息，作相关动作，完成后将结果锁存到寄存器中，作为下一个功能块的输入。建议划分为教课书上的5个功能块：取指、译码、执行、访存、写回，将理论与实践相结合。

3.画出划分后的多周期CPU的框图，大致框图如图3.1。从图中可以看出指令每个周期走完一个功能块，进入下一个功能块。标注的clk箭头是去往相邻模块的中间锁存器，是因为每个模块的输出需要锁存到寄存器中，下一个模块会从该寄存器中读出数据作为自己的输入，寄存器的锁存是需要时钟控制的。值得注意的是，写回模块所做的就是从访存模块获取要写入寄存器堆的数据和目的寄存器，送往寄存器堆，其所需的clk信号是最终发生在寄存器堆的写操作上的。



**图3.1：多周期CPU的大致框图**

4.本次课程设计是需要用到之前组成原理实验的成果的，比如ALU模块、寄存器堆模块、指令ROM模块和数据RAM模块，其中ROM和RAM建议使用调用库IP实例化的同步存储器，因为存储器在实际应用中基本都是同步读写的，为了更贴近真实情况，此处建议使用同步RAM和ROM。

5.在存储器实验中生成的同步RAM和ROM，都是在发送地址后的下一拍才能获得对应数据的。故在在使用同步存储器时，从指令和数据存储器中读取数据就需要等待一拍时钟了，即取指令需要两拍时间，load操作也需要两拍时间。在真实的处理器系统中，取指令和访存其实都是需要多拍时钟的。

6.本次实验，需要完成表3.1和表3.2的填写。

**表3.1：mips基础指令特性归纳表**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **指令**  **类型** | **汇编指令** | **指令码** | **源操作**  **数 1** | **源操作**  **数 2** | **目的寄**  **存器** | **功能描述** |
| R 型指令 | addu rd , rs , rt | 000000 rs | rt | rd |00000|100001 | [rs] | [rt] | rd | GPR[rd]=GPR[rs]+GPR[rt] |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
| I 型指令 | addiu rt,rs,imm | 001001 rs | rt | imm | [rs] | sign\_ext(imm) | rt | GPR[rt]=GPR[rs]+sign\_ext(imm) |
|  |  |  |  |  |  |
| J 型  指令 | j target | 000010|target | PC | Target |  | 跳转, PC={PC[31:28],target,2’b00} |

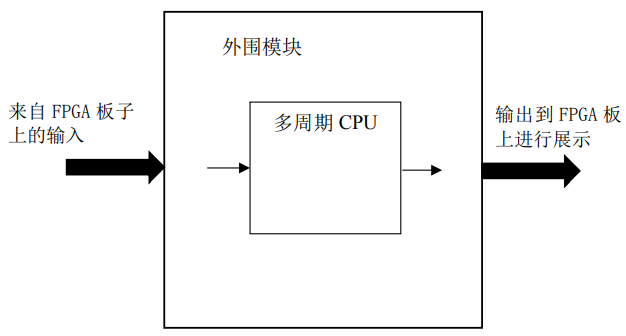
**表3.2：测试所用汇编程序详述**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **指令地址** | **汇编指令** | **结 果描 述** | **机器指令的机器码** | |
| **16 进制** | **二进制** |
| 00H | addiu ,$1, $0,#1 | [$1] = 0000\_0001H | 24010001 | 0010\_0100\_0000\_0001 0000\_0000\_0000\_0001 |
|  |  |  |  |  |
|  |  |  |  |  |

7.根据设计的实验方案，使用verilog编写相应代码。

8.对编写的代码进行仿真，得到正确的波形图。

9.将以上设计作为一个单独的模块，设计一个外围模块去调用该模块，见图3.2。外围模块中需调用封装好的LCD触摸屏模块，观察多周期CPU的内部状态，比如32个寄存器的值，各模块PC的值等。并且需要利用触摸功能输入特定数据 RAM地址，从该RAM的调试端口读出数据显示在屏上，以达到实时观察数据存储器内部数据变化的效果。通过这些手段，可以在板上充分验证CPU的正确性。



**图3.2：多周期CPU设计实验的顶层模块大致框图**

10. 将编写的代码进行综合布局布线，并下载到实验箱中的FPGA板子上进行演示。

四、多周期CPU框架设计

4.1 总体框架设计

参考经典的MIPS五段流水模式，这里也同样将多周期CPU分成五个模块，分别是取指（IF）、译码（ID）、执行（EXE）、访存（MEM）、写回（WB）。具体的CPU整体设计框架图如图4.1所示。下面将对这五个模块的设计理念进行概述：

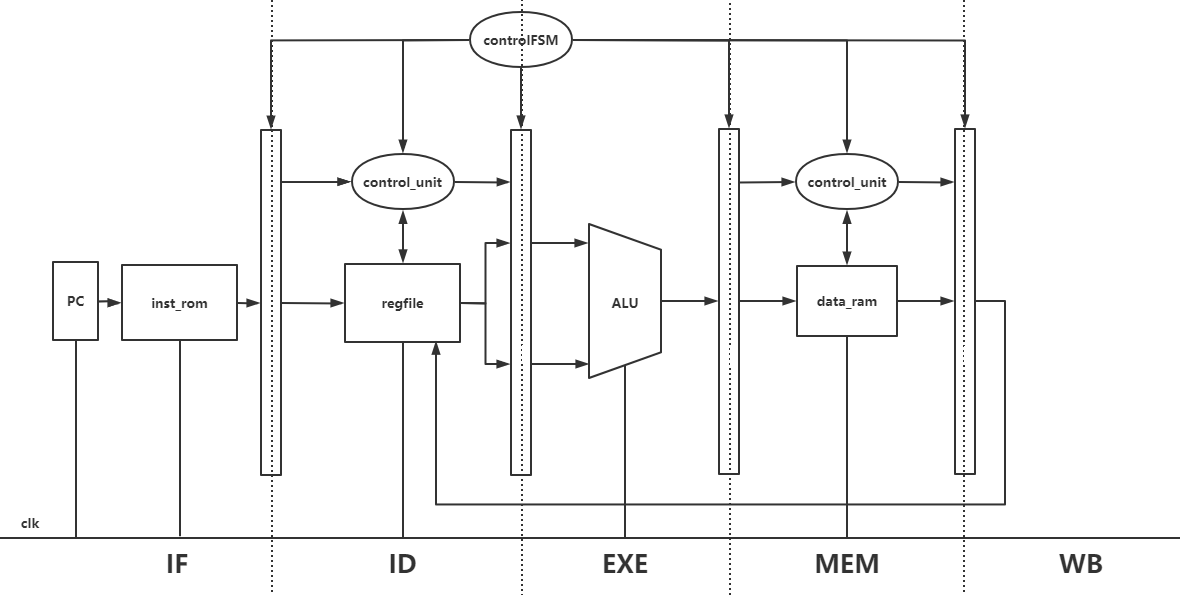
**1.取指（IF）：**该模块包括指令计数器PC和用于存储指令序列的inst\_rom。进入取指状态的同时，PC 更新为下一 PC 值。故取指状态下，将 PC 值作为指令存储器的地址去取指令。由于同步指令存储器在下一时钟周期返回指令，因此取指需要两个时钟周期的时间。当取指结束后锁存取指阶段产生的结果——当前 PC 值和指令。

**2.译码（ID）：**该模块包括寄存器堆Regfile和一个控制单元。其类似于单周期 CPU 的译码阶段，主要完成指令译码、读寄存器、判断跳转等。控制单元区分各条指令并产生用于译码、执行、访存、写回的控制信号。当译码结束后锁存译码阶段产生的结果用于下一状态执行：分别用于执行、访存、写回的控制信号、用于执行阶段的两个源操作数、用于访存阶段的写入内存数据、用于写回阶段的写寄存器地址。

**3.执行（EXE）：**该模块包括算术逻辑单元ALU，而一般ALU内还有最基本的加法器等器件。其主要是ALU模块完成相关操作。当执行结束后锁存执行阶段产生的结果及前级传递的结果：用于访存、写回的控制信号、ALU 结果、内存写入数据、寄存器写地址。

**4.访存（MEM）：**该模块包括内存单元data\_ram和一个控制单元。该模块主要完成完成对数据存储器的读或写，并选择出将要写回寄存器的值。当访存结束后锁存访存阶段产生的结果及前级传递的结果：用于写回的控制信号、写回数据、写回地址。

**5.写回（WB）：**最简单的模块，指令结束前完成寄存器的写入。

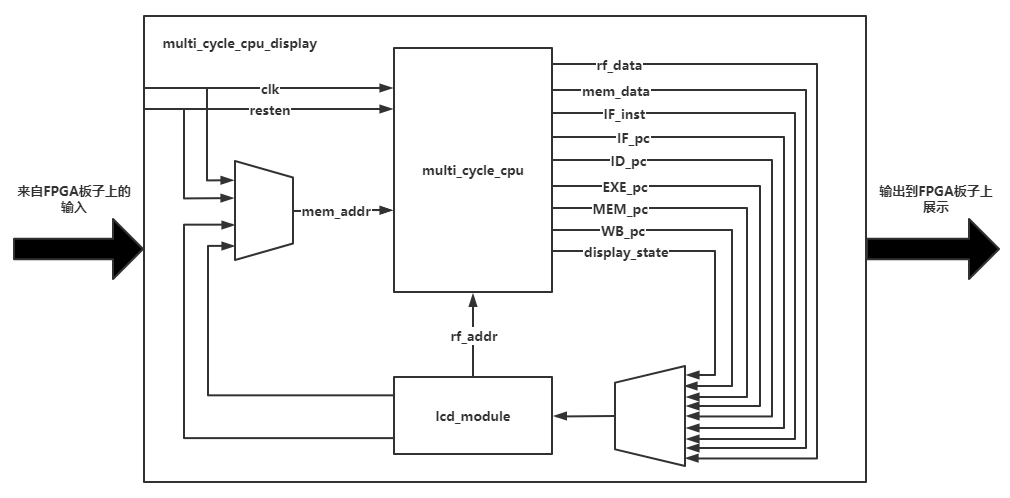


**图4.1：**多周期CPU的整体设计框图

4.2 顶层框架设计

上述的设计作为一个单独的模块，设计一个外围模块去调用该模块，外围模块中需调用封装好的LCD触摸屏模块，观察多周期CPU的内部状态，从而验证所做CPU的正确性和汇编指令序列的正确性。

这里将直接调用组原实验中单周期CPU实验所给的“lcd\_module.dcp”，且调用lcd触摸板的相关接口没有进行任何修改。具体的顶层框架设计框图如图4.2所示。此外表4.1还给出了顶层模块接口的说明，即multi\_cycle\_cpu模块的接口说明。



**图4.2：**多周期CPU顶层模块设计框图

**表4.1：**顶层模块接口说明

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **序号** | **接口名称信息** | **位宽** | **方向(I/O)** | **说明** |
| 1 | clk | 1 | I | 时钟信号 |
| 2 | resten | 1 | I | 复位信号 |
| 3 | rf\_addr | [4:0] | I | 访问寄存器地址 |
| 4 | mem\_addr | [31:0] | I | 访问存储器地址 |
| 5 | rf\_data | [31:0] | O | 寄存器数据 |
| 6 | mem\_data | [31:0] | O | 存储器数据 |
| 7 | IF\_inst | [31:0] | O | 当前所取指令 |
| 8 | IF\_pc | [31:0] | O | IF段PC值 |
| 9 | ID\_pc | [31:0] | O | ID段PC值 |
| 10 | EXE\_pc | [31:0] | O | EXE段PC值 |
| 11 | MEM\_pc | [31:0] | O | MEM段PC值 |
| 12 | WB\_pc | [31:0] | O | WB段PC值 |
| 13 | display\_state | [31:0] | O | 多周期CPU的状态 |

五、多周期CPU指令归纳表

该多周期CPU以组成原理、系统结构课程中的单周期CPU为基础，在单周期CPU指令集的基础上，增加了约20条算术逻辑运算指令、数据传送指令和控制指令，共计实现了 36 条指令。具体实现的指令集合如表5.1所示，表中给出了每一套指令的指令形式、指令码、两个源操作数、目的操作数以及功能描述。

**表5.1**:多周期CPU指令特性归纳表

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **指令**  **类型** | **汇编指令** | **指令码** | **源操作**  **数 1** | **源操作**  **数 2** | **目的寄存器** | **功能描述** |
| R型指令 | addu rd,rs,rt | 000000|rs|rt|rd|00000|100001 | [rs] | [rt] | [rd] | 无符号加 |
| subu rd,rs,rt | 000000|rs|rt|rd|00000|100011 | [rs] | [rt] | [rd] | 无符号减 |
| slt rd,rs,rt | 000000|rs|rt|rd|00000|101010 | [rs] | [rt] | [rd] | 小于置位 |
| sltu rd,rs,rt | 000000|rs|rt|rd|0000 0|101011 | [rs] | [rt] | [rd] | 无符号小于置位 |
| jalr rs | 000000|rs|00000|11111|00000|001001 | [rs] |  | 31 | 使用寄存器的链接跳转 |
| jr rs | 000000|rs|0000000000|00000|001000 | [rs] |  |  | 使用寄存器的跳转 |
| and rd,rs,rt | 000000|rs|rt|rd|00000|100100 | [rs] | [rt] | [rd] | 与运算 |
| nor rd,rs,rt | 000000|rs|rt|rd|00000|100111 | [rs] | [rt] | [rd] | 或非运算 |
| or rd,rs,rt | 000000|rs|rt|rd|00000|100101 | [rs] | [rt] | [rd] | 或运算 |
| xor rd,rs,rt | 000000|rs|rt|rd|00000|100110 | [rs] | [rt] | [rd] | 异或运算 |
| sll rd,rt,shf | 000000|00000|rt|rd|shf|000000 |  | [rt] | [rd] | 逻辑左移 |
| sllv rd,rt,rs | 000000|rs|rt|rd|0000 0|000100 | [rs] | [rt] | [rd] | 逻辑可变左移 |
| sra rd,rt,shf | 000000|00000|rt|rd|s hf|000011 |  | [rt] | [rd] | 算术右移 |
| srav rd,rt,rs | 000000|rs|rt|rd|0000 0|000111 | [rs] | [rt] | [rd] | 算术可变右移 |
| srl rd,rt,shf | 000000|00000|rt|rd|shf|000010 |  | [rt] | [rd] | 逻辑右移 |
| srlv rd,rt,rs | 000000|rs|rt|rd|0000 0|000110 | [rs] | [rt] | [rd] | 逻辑可变右移 |
| I 型指令 | addiu rt,rs,imm | 001001|rs|rt|imm | [rs] | sign\_ext(imm) | [rt] | 无符号立即数加 |
| slti rt,rs,imm | 001010|rs|rt|imm | [rs] | sign\_ext (imm) | [rt] | 立即数小于置位 |
| sltiu rt,rs,imm | 001011|rs|rt|imm | [rs] | sign\_ext (imm) | [rt] | 立即数无符号小于置位 |
| beq rs,rt,offset | 000100|rs|rt|offset | [rs] | [rt] |  | 相等跳转 |
| bgez rs,offset | 000001|rs|00001|off set | [rs] |  |  | 大于等于0跳转 |
| bgtz rs,offset | 000111|rs|00000|off set | [rs] |  |  | 大于0跳转 |
| blez rs,offset | 000110|rs|00000|off set | [rs] |  |  | 小于等于0跳转 |
| bltz rs,offset | 000001|rs|00000|off set | [rs] |  |  | 小于0转移 |
| bne rs,rt,offset | 000101|rs|rt|offset | [rs] | [rt] |  | 不相等跳转 |
| lw rt,offset(b) | 100011|b|rt|offset | [b] | sign\_ext(offset) | [rt] | 存储器取数（word） |
| sw rt,offset(b) | 101011|b|rt|offset | [b] | sign\_ext(offset) | [rt] | 存储器存数（word） |
| lb rt,offset(b) | 100000|b|rt|offset | [b] | sign\_ext (offset) | [rt] | 存储器取数（byte） |
| lbu rt,offset(b) | 100100|b|rt|offset | [b] | sign\_ext (offset) | [rt] | 存储器取数（byte） |
| sb rt,offset(b) | 101000|b|rt|offset | [b] | sign\_ext (offset) | [rt] | 存储器存数（byte） |
| andi rt,rs,imm | 001100|rs|rt|imm | [rs] | zero\_ext (imm) | [rt] | 立即数与运算 |
| lui rt,imm | 001111|00000|rt|imm |  | {imm,16'd0} | [rt] | 立即数载入 |
| ori rt,rs,imm | 001101|rs|rt|imm | [rs] | zero\_ext (imm) | [rt] | 立即数或运算 |
| xori rt,rs,imm | 001110|rs|rt|imm | [rs] | zero\_ext (imm) | [rt] | 立即数异或运算 |
| J 型  指令 | j target | 000010|target |  |  |  | 无条件跳转 |
| jal target | 000011|target |  |  |  | 无条件跳转并链接 |

六、Verilog程序代码

6.1 CPU顶层代码（multi\_cycle\_cpu.v）

多周期CPU模块，共实现36条指令。其中指令rom和数据ram均实例化xilinx IP得到，为同步读写。其余模块均为verilog代码。该模块主要用于：（1）控制多周期的状态机的运行（2）控制5级间的总线（3）实例化各个模块。具体的代码如下所示**（注：这里我将源码复制到VsCode中再复制到该文档中，可以保留VsCode黑色背景，以更好地展示代码）**：

`timescale 1ns / 1ps

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

module multi\_cycle\_cpu( // 多周期 cpu

    input clk, // 时钟

    input resetn, // 复位信号，低电平有效

    input [ 4:0] rf\_addr,

    input [31:0] mem\_addr,

    output [31:0] rf\_data,

    output [31:0] mem\_data,

    output [31:0] IF\_pc,

    output [31:0] IF\_inst,

    output [31:0] ID\_pc,

    output [31:0] EXE\_pc,

    output [31:0] MEM\_pc,

    output [31:0] WB\_pc,

    output [31:0] display\_state

    );

//-------------------{控制多周期的状态机}begin--------------------

//

    reg [2:0] state; // 当前状态

    reg [2:0] next\_state; // 下一状态

//展示当前处理器正在执行哪个模块

    assign display\_state = {29'd0,state};

// 状态机状态

    parameter IDLE = 3'd0; // 开始

    parameter FETCH = 3'd1; // 取指

    parameter DECODE = 3'd2; // 译码

    parameter EXE = 3'd3; // 执行

    parameter MEM = 3'd4; // 访存

    parameter WB = 3'd5; // 写回

    always @ (posedge clk) // 当前状态

    begin

        if (!resetn) begin // 如果复位信号有效

            state <= IDLE; // 当前状态为 开始

        end

        else begin // 否则

            state <= next\_state; // 为下一状态

        end

    end

    wire IF\_over; // IF 模块已执行完

    wire ID\_over; // ID 模块已执行完

    wire EXE\_over; // EXE 模块已执行完

    wire MEM\_over; // MEM 模块已执行完

    wire WB\_over; // WB 模块已执行完

    wire jbr\_not\_link;//分支指令(非 link 类)，只走 IF 和 ID 级

    always @ (\*) // 下一状态

    begin

    case (state)

IDLE:

    begin

        next\_state = FETCH; // 开始->取指

    end

FETCH:

    begin

        if (IF\_over)

        begin

            next\_state = DECODE; // 取指->译码

        end

        else

        begin

            next\_state = FETCH; // 取指->译码

        end

    end

DECODE:

    begin

        if (ID\_over)

        begin // 译码->执行或写回

            next\_state = jbr\_not\_link ? FETCH : EXE;

        end

        else

        begin

        next\_state = DECODE; // 取指->译码

        end

    end

EXE:

    begin

        if (EXE\_over)

        begin

            next\_state = MEM; // 执行->访存

        end

        else

        begin

            next\_state = EXE; // 取指->译码

        end

    end

MEM:

    begin

        if (MEM\_over)

        begin

            next\_state = WB; // 访存->写回

        end

        else

        begin

            next\_state = MEM; // 取指->译码

        end

    end

WB:

    begin

        if (WB\_over)

        begin

            next\_state = FETCH; // 写回->取指

        end

        else

        begin

            next\_state = WB; // 取指->译码

        end

    end

default : next\_state = IDLE;

    endcase

    end

//5 模块的 valid 信号

    wire IF\_valid;

    wire ID\_valid;

    wire EXE\_valid;

    wire MEM\_valid;

    wire WB\_valid;

    assign IF\_valid = (state == FETCH ); //当前状态为取指时，IF 级有效

    assign ID\_valid = (state == DECODE); //当前状态为译码时，ID 级有效

    assign EXE\_valid = (state == EXE ); //当前状态为执行时，EXE 级有效

    assign MEM\_valid = (state == MEM ); //当前状态为访存时，MEM 级有效

    assign WB\_valid = (state == WB ); //当前状态为写回时，WB 级有效

//--------------------{控制多周期的状态机}end--------------

//----------------------{5 级间的总线}begin----------------------

//

    wire [ 63:0] IF\_ID\_bus; // IF->ID 级总线

    wire [149:0] ID\_EXE\_bus; // ID->EXE 级总线

    wire [105:0] EXE\_MEM\_bus; // EXE->MEM 级总线

    wire [ 69:0] MEM\_WB\_bus; // MEM->WB 级总线

//锁存以上总线信号

    reg [ 63:0] IF\_ID\_bus\_r;

    reg [149:0] ID\_EXE\_bus\_r;

    reg [105:0] EXE\_MEM\_bus\_r;

    reg [ 69:0] MEM\_WB\_bus\_r;

//IF 到 ID 的锁存信号

    always @(posedge clk)

    begin

        if(IF\_over)

        begin

            IF\_ID\_bus\_r <= IF\_ID\_bus;

        end

    end

//ID 到 EXE 的锁存信号

    always @(posedge clk)

    begin

        if(ID\_over)

        begin

            ID\_EXE\_bus\_r <= ID\_EXE\_bus;

        end

    end

//EXE 到 MEM 的锁存信号

    always @(posedge clk)

    begin

        if(EXE\_over)

        begin

            EXE\_MEM\_bus\_r <= EXE\_MEM\_bus;

        end

    end

//MEM 到 WB 的锁存信号

    always @(posedge clk)

    begin

        if(MEM\_over)

        begin

            MEM\_WB\_bus\_r <= MEM\_WB\_bus;

        end

    end

//-----------------------{5 级间的总线}end-----------------------

//----------------------{其他交互信号}begin----------------------

//跳转总线

    wire [ 32:0] jbr\_bus;

//IF 与 inst\_rom 交互

    wire [31:0] inst\_addr;

    wire [31:0] inst;

//MEM 与 data\_ram 交互

    wire [ 3:0] dm\_wen;

    wire [31:0] dm\_addr;

    wire [31:0] dm\_wdata;

    wire [31:0] dm\_rdata;

//ID 与 regfile 交互

    wire [ 4:0] rs;

    wire [ 4:0] rt;

    wire [31:0] rs\_value;

    wire [31:0] rt\_value;

//WB 与 regfile 交互

    wire rf\_wen;

    wire [ 4:0] rf\_wdest;

    wire [31:0] rf\_wdata;

//-----------------------{其他交互信号}end-----------------------

//

//---------------------{各模块实例化}begin-----------------------

//

    wire next\_fetch; //即将运行取指模块，需要先锁存 PC 值

//当前状态为 decode，且指令为跳转分支指令(非 link 类)，且 decode 执行完成

//或者，当前状态为 wb，且 wb 执行完成，则即将进入 fetch 状态

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

    fetch IF\_module( // 取指级

    .clk (clk ), // I, 1

    .resetn (resetn ), // I, 1

    .IF\_valid (IF\_valid ), // I, 1

    .next\_fetch(next\_fetch), // I, 1

    .inst (inst ), // I, 32

    .jbr\_bus (jbr\_bus ), // I, 33

    .inst\_addr (inst\_addr ), // O, 32

    .IF\_over (IF\_over ), // O, 1

    .IF\_ID\_bus (IF\_ID\_bus ), // O, 64

    .IF\_pc (IF\_pc ),

    .IF\_inst (IF\_inst )

    );

    decode ID\_module( // 译码级

    .ID\_valid (ID\_valid ), // I, 1

    .IF\_ID\_bus\_r (IF\_ID\_bus\_r ), // I, 64

    .rs\_value (rs\_value ), // I, 32

    .rt\_value (rt\_value ), // I, 32

    .rs (rs ), // O, 5

    .rt (rt ), // O, 5

    .jbr\_bus (jbr\_bus ), // O, 33

    .jbr\_not\_link(jbr\_not\_link), // O, 1

    .ID\_over (ID\_over ), // O, 1

    .ID\_EXE\_bus (ID\_EXE\_bus ), // O, 150

    .ID\_pc (ID\_pc )

    );

    exe EXE\_module( // 执行级

    .EXE\_valid (EXE\_valid ), // I, 1

    .ID\_EXE\_bus\_r(ID\_EXE\_bus\_r), // I, 150

    .EXE\_over (EXE\_over ), // O, 1

    .EXE\_MEM\_bus (EXE\_MEM\_bus ), // O, 106

    .EXE\_pc (EXE\_pc )

    );

    mem MEM\_module( // 访存级

    .clk (clk ), // I, 1

    .MEM\_valid (MEM\_valid ), // I, 1

    .EXE\_MEM\_bus\_r(EXE\_MEM\_bus\_r), // I, 106

    .dm\_rdata (dm\_rdata ), // I, 32

    .dm\_addr (dm\_addr ), // O, 32

    .dm\_wen (dm\_wen ), // O, 4

    .dm\_wdata (dm\_wdata ), // O, 32

    .MEM\_over (MEM\_over ), // O, 1

    .MEM\_WB\_bus (MEM\_WB\_bus ), // O, 70

    .MEM\_pc (MEM\_pc )

    );

    wb WB\_module( // 写回级

    .WB\_valid (WB\_valid ), // I, 1

    .MEM\_WB\_bus\_r(MEM\_WB\_bus\_r), // I, 70

    .rf\_wen (rf\_wen ), // O, 1

    .rf\_wdest (rf\_wdest ), // O, 5

    .rf\_wdata (rf\_wdata ), // O, 32

    .WB\_over (WB\_over ), // O, 1

    .WB\_pc (WB\_pc )

    );

    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

    );

    regfile rf\_module( // 寄存器堆模块

    .clk (clk ), // I, 1

    .wen (rf\_wen ), // I, 1

    .raddr1 (rs ), // I, 5

    .raddr2 (rt ), // I, 5

    .waddr (rf\_wdest ), // I, 5

    .wdata (rf\_wdata ), // I, 32

    .rdata1 (rs\_value ), // O, 32

    .rdata2 (rt\_value ), // O, 32

    .test\_addr(rf\_addr),

    .test\_data(rf\_data)

    );

    data\_ram data\_ram\_module( // 数据存储模块

    .clka (clk ), // I, 1, 时钟

    .wea (dm\_wen ), // I, 1, 写使能

    .addra (dm\_addr[9:2]), // I, 8, 读地址

    .dina (dm\_wdata ), // I, 32, 写数据

    .douta (dm\_rdata ), // O, 32, 读数据

    //display mem

    .clkb (clk ),

    .web (4'd0 ),

    .addrb (mem\_addr[9:2]),

    .doutb (mem\_data ),

    .dinb (32'd0 )

    );

//----------------------{各模块实例化}end------------------------

endmodule

6.2 取指模块代码（fetch.v）

多周期CPU取指模块，该模块主要从inst\_rom**（inst\_rom为实例化xilinx IP，在此不再赘）**中取出预先准备好的测试程序，述并控制PC值的变化。数据流向为IF->ID。具体的代码如下所示：

`timescale 1ns / 1ps

`define STARTADDR 32'd0 // 程序起始地址为 0

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

    );

//-----{程序计数器 PC}begin

    wire [31:0] next\_pc;

    wire [31:0] seq\_pc;

    reg [31:0] pc;

//跳转 pc

    wire jbr\_taken;

    wire [31:0] jbr\_target;

    assign {jbr\_taken, jbr\_target} = jbr\_bus; //跳转总线

    assign seq\_pc[31:2] = pc[31:2] + 1'b1; //下一指令地址：PC=PC+4

    assign seq\_pc[1 :0] = pc[1:0];

// 新指令：若指令跳转，为跳转地址；否则为下一指令

    assign next\_pc = jbr\_taken ? jbr\_target : seq\_pc;

    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

//-----{程序计数器 PC}end

//-----{发往 inst\_rom 的取指地址}begin

    assign inst\_addr = pc;

//-----{发往 inst\_rom 的取指地址}end

//-----{IF 执行完成}begin

    always @(posedge clk)

    begin

        IF\_over <= IF\_valid;

    end

//如果指令 rom 为异步读的，则 IF\_valid 即是 IF\_over 信号，

//即取指一拍完成

//-----{IF 执行完成}end

//-----{IF->ID 总线}begin

    assign IF\_ID\_bus = {pc, inst};

//-----{IF->ID 总线}end

//-----{展示 IF 模块的 PC 值和指令}begin

    assign IF\_pc = pc;

    assign IF\_inst = inst;

//-----{展示 IF 模块的 PC 值和指令}end

endmodule

6.3 译码模块代码（decode.v）

多周期CPU译码模块，该模块主要将取出的十六进制代码进行译码分析，确定指令的类型及功能。数据流向为IF->ID和ID->EXE。具体的代码如下所示：

`timescale 1ns / 1ps

module decode( // 译码级

    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 ID\_over, // ID 模块执行完成

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

    //展示 PC

    output [ 31:0] ID\_pc

    );

//-----{IF->ID 总线}begin

    wire [31:0] pc;

    wire [31:0] inst;

    assign {pc, inst} = IF\_ID\_bus\_r; // IF->ID 总线传

//-----{IF->ID 总线}end

//-----{指令译码}begin

    wire [5:0] op;

    wire [4:0] rd;

    wire [4:0] sa;

    wire [5:0] funct;

    wire [15:0] imm;

    wire [15:0] offset;

    wire [25:0] target;

    assign op = inst[31:26]; // 操作码

    assign rs = inst[25:21]; // 源操作数 1

    assign rt = inst[20:16]; // 源操作数 2

    assign rd = inst[15:11]; // 目标操作数

    assign sa = inst[10:6]; // 特殊域，可能存放偏移量

    assign funct = inst[5:0]; // 功能码

    assign imm = inst[15:0]; // 立即数

    assign offset = inst[15:0]; // 地址偏移量

    assign target = inst[25:0]; // 目标地址

// 实现指令列表

    wire inst\_ADDU, inst\_SUBU , inst\_SLT , inst\_AND;

    wire inst\_NOR , inst\_OR , inst\_XOR , inst\_SLL;

    wire inst\_SRL , inst\_ADDIU, inst\_BEQ , inst\_BNE;

    wire inst\_LW , inst\_SW , inst\_LUI , inst\_J;

    wire inst\_SLTU, inst\_JALR , inst\_JR , inst\_SLLV;

    wire inst\_SRA , inst\_SRAV , inst\_SRLV, inst\_SLTIU;

    wire inst\_SLTI, inst\_BGEZ , inst\_BGTZ, inst\_BLEZ;

    wire inst\_BLTZ, inst\_LB , inst\_LBU , inst\_SB;

    wire inst\_ANDI, inst\_ORI , inst\_XORI, inst\_JAL;

    wire op\_zero; // 操作码全 0

    wire sa\_zero; // sa 域全 0

    assign op\_zero = ~(|op);

    assign sa\_zero = ~(|sa);

    assign inst\_ADDU = op\_zero & sa\_zero & (funct == 6'b100001);//无符号加法

    assign inst\_SUBU = op\_zero & sa\_zero & (funct == 6'b100011);//无符号减法

    assign inst\_SLT = op\_zero & sa\_zero & (funct == 6'b101010);//小于则置位

    assign inst\_SLTU = op\_zero & sa\_zero & (funct == 6'b101011);//无符号小则置

    assign inst\_JALR = op\_zero & (rt==5'd0) & (rd==5'd31)& sa\_zero & (funct == 6'b001001); //跳转寄存器并链接

    assign inst\_JR = op\_zero & (rt==5'd0) & (rd==5'd0 )& sa\_zero & (funct == 6'b001000); //跳转寄存器

    assign inst\_AND = op\_zero & sa\_zero & (funct == 6'b100100);//与运算

    assign inst\_NOR = op\_zero & sa\_zero & (funct == 6'b100111);//或非运算

    assign inst\_OR = op\_zero & sa\_zero & (funct == 6'b100101);//或运算

    assign inst\_XOR = op\_zero & sa\_zero & (funct == 6'b100110);//异或运算

    assign inst\_SLL = op\_zero & (rs==5'd0) & (funct == 6'b000000);//逻辑左移

    assign inst\_SLLV = op\_zero & sa\_zero & (funct == 6'b000100);//变量逻辑左移

    assign inst\_SRA = op\_zero & (rs==5'd0) & (funct == 6'b000011);//算术右移

    assign inst\_SRAV = op\_zero & sa\_zero & (funct == 6'b000111);//变量算术右移

    assign inst\_SRL = op\_zero & (rs==5'd0) & (funct == 6'b000010);//逻辑右移

    assign inst\_SRLV = op\_zero & sa\_zero & (funct == 6'b000110);//变量逻辑右移

    assign inst\_ADDIU = (op == 6'b001001); //立即数无符号加法

    assign inst\_SLTI = (op == 6'b001010); //小于立即数则置位

    assign inst\_SLTIU = (op == 6'b001011); //无符号小于立即数则置位

    assign inst\_BEQ = (op == 6'b000100); //判断相等跳转

    assign inst\_BGEZ = (op == 6'b000001) & (rt==5'd1); //大于等于 0 跳转

    assign inst\_BGTZ = (op == 6'b000111) & (rt==5'd0); //大于 0 跳转

    assign inst\_BLEZ = (op == 6'b000110) & (rt==5'd0); //小于等于 0 跳转

    assign inst\_BLTZ = (op == 6'b000001) & (rt==5'd0); //小于 0 跳转

    assign inst\_BNE = (op == 6'b000101); //判断不等跳转

    assign inst\_LW = (op == 6'b100011); //从内存装载字

    assign inst\_SW = (op == 6'b101011); //向内存存储字

    assign inst\_LB = (op == 6'b100000); //load 字节（符号扩展）

    assign inst\_LBU = (op == 6'b100100); //load 字节（无符号扩展）

    assign inst\_SB = (op == 6'b101000); //向内存存储字节

    assign inst\_ANDI = (op == 6'b001100); //立即数与

    assign inst\_LUI = (op == 6'b001111) & (rs==5'd0); //立即数装载高半字节

    assign inst\_ORI = (op == 6'b001101); //立即数或

    assign inst\_XORI = (op == 6'b001110); //立即数异或

    assign inst\_J = (op == 6'b000010); //跳转

    assign inst\_JAL = (op == 6'b000011); //跳转和链接

    //跳转分支指令

    wire inst\_jr; //寄存器跳转指令

    wire inst\_j\_link;//链接跳转指令

    assign inst\_jr = inst\_JALR | inst\_JR;

    assign inst\_j\_link = inst\_JAL | inst\_JALR;

    assign jbr\_not\_link= inst\_J | inst\_JR | inst\_BEQ | inst\_BNE | inst\_BGEZ | inst\_BGTZ | inst\_BLEZ | inst\_BLTZ;//全部非 link 类跳转指令

    //load store

    wire inst\_load;

    wire inst\_store;

    assign inst\_load = inst\_LW | inst\_LB | inst\_LBU; // load 指令

    assign inst\_store = inst\_SW | inst\_SB; // store 指令

    //alu 操作分类

    wire inst\_add, inst\_sub, inst\_slt,inst\_sltu;

    wire inst\_and, inst\_nor, inst\_or, inst\_xor;

    wire inst\_sll, inst\_srl, inst\_sra,inst\_lui;

    assign inst\_add = inst\_ADDU | inst\_ADDIU | inst\_load | inst\_store | inst\_j\_link; // 做加法

    assign inst\_sub = inst\_SUBU; // 减法

    assign inst\_slt = inst\_SLT | inst\_SLTI; // 有符号小于置位

    assign inst\_sltu= inst\_SLTIU | inst\_SLTU; // 无符号小于置位

    assign inst\_and = inst\_AND | inst\_ANDI; // 逻辑与

    assign inst\_nor = inst\_NOR; // 逻辑或非

    assign inst\_or = inst\_OR | inst\_ORI; // 逻辑或

    assign inst\_xor = inst\_XOR | inst\_XORI; // 逻辑或非

    assign inst\_sll = inst\_SLL | inst\_SLLV; // 逻辑左移

    assign inst\_srl = inst\_SRL | inst\_SRLV; // 逻辑右移

    assign inst\_sra = inst\_SRA | inst\_SRAV; // 算术右移

    assign inst\_lui = inst\_LUI; // 立即数装载高位

//使用 sa 域作为偏移量的移位指令

    wire inst\_shf\_sa;

    assign inst\_shf\_sa = inst\_SLL | inst\_SRL | inst\_SRA;

//依据立即数扩展方式分类

    wire inst\_imm\_zero; //立即数 0 扩展

    wire inst\_imm\_sign; //立即数符号扩展

    assign inst\_imm\_zero = inst\_ANDI | inst\_LUI | inst\_ORI | inst\_XORI;

    assign inst\_imm\_sign = inst\_ADDIU | inst\_SLTI | inst\_SLTIU | inst\_load | inst\_store;

//依据目的寄存器号分类

    wire inst\_wdest\_rt; // 寄存器堆写入地址为 rt 的指令

    wire inst\_wdest\_31; // 寄存器堆写入地址为 31 的指令

    wire inst\_wdest\_rd; // 寄存器堆写入地址为 rd 的指令

    assign inst\_wdest\_rt = inst\_imm\_zero | inst\_ADDIU | inst\_SLTI | inst\_SLTIU | inst\_load;

    assign inst\_wdest\_31 = inst\_JAL;

    assign inst\_wdest\_rd = inst\_ADDU | inst\_SUBU | inst\_SLT | inst\_SLTU | inst\_JALR | inst\_AND | inst\_NOR | inst\_OR

    | inst\_XOR | inst\_SLL | inst\_SLLV | inst\_SRA | inst\_SRAV | inst\_SRL | inst\_SRLV;

//-----{指令译码}end

//-----{分支指令执行}begin

//无条件跳转

    wire j\_taken;

    wire [31:0] j\_target;

    assign j\_taken = inst\_J | inst\_JAL | inst\_jr;

//寄存器跳转地址为 rs\_value,其他跳转为{pc[31:28],target,2'b00}

    assign j\_target = inst\_jr ? rs\_value :

    {pc[31:28],target,2'b00};

//branch 指令

    wire rs\_equql\_rt;

    wire rs\_ez;

    wire rs\_ltz;

    assign rs\_equql\_rt = (rs\_value == rt\_value); // GPR[rs]==GPR[rt]

    assign rs\_ez = ~(|rs\_value); // rs 寄存器值为 0

    assign rs\_ltz = rs\_value[31]; // rs 寄存器值小于 0

    wire br\_taken;

    wire [31:0] br\_target;

    assign br\_taken = inst\_BEQ & rs\_equql\_rt // 相等跳转

    | inst\_BNE & ~rs\_equql\_rt // 不等跳转

    | inst\_BGEZ & ~rs\_ltz // 大于等于 0 跳转

    | inst\_BGTZ & ~rs\_ltz & ~rs\_ez // 大于 0 跳转

    | inst\_BLEZ & (rs\_ltz | rs\_ez) // 小于等于 0 跳转

    | inst\_BLTZ & rs\_ltz; // 小于 0 跳转

// 分支跳转目标地址：PC=PC+offset<<2

    assign br\_target[31:2] = pc[31:2] + {{14{offset[15]}}, offset};

    assign br\_target[1:0] = pc[1:0];

//jump and branch 指令

    wire jbr\_taken;

    wire [31:0] jbr\_target;

    assign jbr\_taken = j\_taken | br\_taken;

    assign jbr\_target = j\_taken ? j\_target : br\_target;

//ID 到 IF 的跳转总线

    assign jbr\_bus = {jbr\_taken, jbr\_target};

//-----{分支指令执行}end

//-----{ID 执行完成}begin

//由于是多周期的，不存在数据相关

//故 ID 模块一拍就能完成所有操作

//故 ID\_valid 即是 ID\_over 信号

    assign ID\_over = ID\_valid;

//-----{ID 执行完成}end

//-----{ID->EXE 总线}begin

//EXE 需要用到的信息

//ALU 两个源操作数和控制信号

    wire [11:0] alu\_control;

    wire [31:0] alu\_operand1;

    wire [31:0] alu\_operand2;

//所谓链接跳转是将跳转返回的 PC 值存放到 31 号寄存器里

//在多周期 CPU 里，不考虑延迟槽，故链接跳转需要计算 PC+4，存放到 31 号寄存器里

    assign alu\_operand1 = inst\_j\_link ? pc :

    inst\_shf\_sa ? {27'd0,sa} : rs\_value;

    assign alu\_operand2 = inst\_j\_link ? 32'd4 :

    inst\_imm\_zero ? {16'd0, imm} :

    inst\_imm\_sign ? {{16{imm[15]}}, imm} : rt\_value;

    assign alu\_control = {inst\_add, // ALU 操作码，独热编码

    inst\_sub,

    inst\_slt,

    inst\_sltu,

    inst\_and,

    inst\_nor,

    inst\_or,

    inst\_xor,

    inst\_sll,

    inst\_srl,

    inst\_sra,

    inst\_lui};

//访存需要用到的 load/store 信息

    wire lb\_sign; //load 一字节为有符号 load

    wire ls\_word; //load/store 为字节还是字,0:byte;1:word

    wire [3:0] mem\_control; //MEM 需要使用的控制信号

    wire [31:0] store\_data; //store 操作的存的数据

    assign lb\_sign = inst\_LB;

    assign ls\_word = inst\_LW | inst\_SW;

    assign mem\_control = {inst\_load,

    inst\_store,

    ls\_word,

    lb\_sign };

//写回需要用到的信息

    wire rf\_wen; //写回的寄存器写使能

    wire [4:0] rf\_wdest; //写回的目的寄存器

    assign rf\_wen = inst\_wdest\_rt | inst\_wdest\_31 |

    inst\_wdest\_rd;

    assign rf\_wdest = inst\_wdest\_rt ? rt ://在不写寄存器堆时，设置为0

    inst\_wdest\_31 ? 5'd31 :

    inst\_wdest\_rd ? rd : 5'd0;

    assign store\_data = rt\_value;

    assign ID\_EXE\_bus = {alu\_control,alu\_operand1,alu\_operand2,//EXE 需要使用的信息

    mem\_control,store\_data, //MEM 需要使用的信号

    rf\_wen, rf\_wdest, //WB 需要使用的信号

    pc}; //PC 值

//-----{ID->EXE 总线}end

//-----{展示 ID 模块的 PC 值}begin

    assign ID\_pc = pc;

//-----{展示 ID 模块的 PC 值}end

endmodule

6.4 执行模块代码（exe.v）

多周期CPU执行模块，该模块主要调用ALU模块进行相关的运算**（该ALU模块为组原实验中所做，在此不再赘述）**。数据流向为ID->EXE和EXE->MEM。具体的代码如下所示：

`timescale 1ns / 1ps

module exe( // 执行级

    input EXE\_valid, // 执行级有效信

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

    output EXE\_over, // EXE 模块执行完成

    output [105:0] EXE\_MEM\_bus, // EXE->MEM 总线

    //展示 PC

    output [ 31:0] EXE\_pc

    );

//-----{ID->EXE 总线}begin

//EXE 需要用到的信息

//ALU 两个源操作数和控制信号

    wire [11:0] alu\_control;

    wire [31:0] alu\_operand1;

    wire [31:0] alu\_operand2;

//访存需要用到的 load/store 信息

    wire [3:0] mem\_control; //MEM 需要使用的控制信号

    wire [31:0] store\_data; //store 操作的存的数据

//写回需要用到的信息

    wire rf\_wen; //写回的寄存器写使能

    wire [4:0] rf\_wdest; //写回的目的寄存器

//pc

    wire [31:0] pc;

    assign {alu\_control,

    alu\_operand1,

    alu\_operand2,

    mem\_control,

    store\_data,

    rf\_wen,

    rf\_wdest,

    pc } = ID\_EXE\_bus\_r;

//-----{ID->EXE 总线}end

//-----{ALU}begin

    wire [31:0] alu\_result;

    alu alu\_module(

    .alu\_control (alu\_control ), // I, 12, ALU 控制信号

    .alu\_src1 (alu\_operand1), // I, 32, ALU 操作数 1

    .alu\_src2 (alu\_operand2), // I, 32, ALU 操作数 2

    .alu\_result (alu\_result ) // O, 32, ALU 结果

    );

//-----{ALU}end

//-----{EXE 执行完成}begin

//由于是多周期的，不存在数据相关

//且所有 ALU 运算都可在一拍内完成

//故 EXE 模块一拍就能完成所有操作

//故 EXE\_valid 即是 EXE\_over 信号

    assign EXE\_over = EXE\_valid;

//-----{EXE 执行完成}end

//-----{EXE->MEM 总线}begin

    assign EXE\_MEM\_bus = {mem\_control,store\_data, //load/store 信息和 store 数据

    alu\_result, //alu 运算结果

    rf\_wen,rf\_wdest, // WB 需要使用的信号

    pc}; // PC

//-----{EXE->MEM 总线}end

//-----{展示 EXE 模块的 PC 值}begin

    assign EXE\_pc = pc;

//-----{展示 EXE 模块的 PC 值}end

endmodule

6.5 访存模块代码（mem.v）

多周期CPU访存模块，该模块主要是控制load/store指令和data\_ram**（data\_ram为实例化xilinx IP，在此不再赘）**进行数据交互。数据流向为EXE->MEM和MEM->WB。具体的代码如下所示：

`timescale 1ns / 1ps

module mem( // 访存级

    input clk, // 时钟

    input MEM\_valid, // 访存级有效信号

    input [105:0] EXE\_MEM\_bus\_r,// EXE->MEM 总线

    input [ 31:0] dm\_rdata, // 访存读数据

    output [ 31:0] dm\_addr, // 访存读写地址

    output reg [ 3:0] dm\_wen, // 访存写使能

    output reg [ 31:0] dm\_wdata, // 访存写数据

    output MEM\_over, // MEM 模块执行完成

    output [ 69:0] MEM\_WB\_bus, // MEM->WB 总线

    //展示 PC

    output [ 31:0] MEM\_pc

    );

//-----{EXE->MEM 总线}begin

//访存需要用到的 load/store 信息

    wire [3 :0] mem\_control; //MEM 需要使用的控制信号

    wire [31:0] store\_data; //store 操作的存的数据

//alu 运算结果

    wire [31:0] alu\_result;

//写回需要用到的信息

    wire rf\_wen; //写回的寄存器写使能

    wire [4:0] rf\_wdest; //写回的目的寄存器

//pc

    wire [31:0] pc;

    assign {mem\_control,

    store\_data,

    alu\_result,

    rf\_wen,

    rf\_wdest,

    pc } = EXE\_MEM\_bus\_r;

//-----{EXE->MEM 总线}end

//-----{load/store 访存}begin

    wire inst\_load; //load 操作

    wire inst\_store; //store 操作

    wire ls\_word; //load/store 为字节还是字,0:byte;1:word

    wire lb\_sign; //load 一字节为有符号 load

    assign {inst\_load,inst\_store,ls\_word,lb\_sign} = mem\_control;

//访存读写地址

    assign dm\_addr = alu\_result;

//store 操作的写使能

    always @ (\*) // 内存写使能信号

    begin

        if (MEM\_valid && inst\_store) // 访存级有效时,且为 store 操作

        begin

            if (ls\_word)

            begin

                dm\_wen <= 4'b1111; // 存储字指令，写使能全 1

            end

            else

            begin // SB 指令，需要依据地址底两位，确定对应的写使能

                case (dm\_addr[1:0])

                2'b00 : dm\_wen <= 4'b0001;

                2'b01 : dm\_wen <= 4'b0010;

                2'b10 : dm\_wen <= 4'b0100;

                2'b11 : dm\_wen <= 4'b1000;

                default : dm\_wen <= 4'b0000;

                endcase

            end

        end

        else

        begin

            dm\_wen <= 4'b0000;

        end

    end

//store 操作的写数据

    always @ (\*) // 对于 SB 指令，需要依据地址低两位，移动 store 的字节至对应位置

    begin

        case (dm\_addr[1:0])

        2'b00 : dm\_wdata <= store\_data;

        2'b01 : dm\_wdata <= {16'd0, store\_data[7:0],8'd0};

        2'b10 : dm\_wdata <= {8'd0, store\_data[7:0],16'd0};

        2'b11 : dm\_wdata <= {store\_data[7:0], 24'd0};

        default : dm\_wdata <= store\_data;

        endcase

    end

//load 读出的数据

    wire load\_sign;

    wire [31:0] load\_result;

    assign load\_sign = (dm\_addr[1:0]==2'd0) ? dm\_rdata[ 7] :

    (dm\_addr[1:0]==2'd1) ? dm\_rdata[15] :

    (dm\_addr[1:0]==2'd2) ? dm\_rdata[23] : dm\_rdata[31] ;

    assign load\_result[7:0] = (dm\_addr[1:0]==2'd0) ? dm\_rdata[ 7:0 ] :

    (dm\_addr[1:0]==2'd1) ? dm\_rdata[15:8 ] :

    (dm\_addr[1:0]==2'd2) ? dm\_rdata[23:16] :

    dm\_rdata[31:24] ;

    assign load\_result[31:8]= ls\_word ? dm\_rdata[31:8] :

    {24{lb\_sign & load\_sign}};

//-----{load/store 访存}end

//-----{MEM 执行完成}begin

//由于数据 RAM 为同步读写的,

//故对 load 指令，取数据时，有一拍延时

//即发地址的下一拍时钟才能得到 load 的数据

//故 mem 在进行 load 操作时有需要两拍时间才能取到数据

//而对其他操作，则只需要一拍时间

    reg MEM\_valid\_r;

    always @(posedge clk)

    begin

        MEM\_valid\_r <= MEM\_valid;

    end

    assign MEM\_over = inst\_load ? MEM\_valid\_r :MEM\_valid;

//如果数据 ram 为异步读的，则 MEM\_valid 即是 MEM\_over 信号，

//即 load 一拍完成

//-----{MEM 执行完成}end

//-----{MEM->WB 总线}begin

    wire [31:0] mem\_result; //MEM 传到 WB 的 result 为 load 结果或 ALU 结果

    assign mem\_result = inst\_load ? load\_result : alu\_result;

    assign MEM\_WB\_bus = {rf\_wen,rf\_wdest, // WB 需要使用的信号

    mem\_result, // 最终要写回寄存器的数据

    pc}; // PC 值

//-----{MEM->WB 总线}begin

//-----{展示 MEM 模块的 PC 值}begin

    assign MEM\_pc = pc;

//-----{展示 MEM 模块的 PC 值}end

endmodule

6.6 写回模块代码（wb.v）

多周期CPU写回模块，该模块主要用于传递寄存器堆的写使能/写地址和写数据等信息。数据流向为EXE->WB。具体的代码如下所示：

`timescale 1ns / 1ps

module wb( // 写回级

    input WB\_valid, // 写回级有效

    input [69:0] MEM\_WB\_bus\_r, // MEM->WB 总线

    output rf\_wen, // 寄存器写使能

    output [ 4:0] rf\_wdest, // 寄存器写地址

    output [31:0] rf\_wdata, // 寄存器写数据

    output WB\_over, // WB 模块执行完成

    //展示 PC

    output [ 31:0] WB\_pc

    );

//-----{MEM->WB 总线}begin

//寄存器堆写使能和写地址

    wire wen;

    wire [4:0] wdest;

//MEM 传来的 result

    wire [31:0] mem\_result;

//pc

    wire [31:0] pc;

    assign {wen,wdest,mem\_result,pc} = MEM\_WB\_bus\_r;

//-----{MEM->WB 总线}end

//-----{WB 执行完成}begin

//WB 模块只是传递寄存器堆的写使能/写地址和写数据

//可在一拍内完成

//故 WB\_valid 即是 WB\_over 信号

    assign WB\_over = WB\_valid;

//-----{WB 执行完成}end

//-----{WB->regfile 信号}begin

    assign rf\_wen = wen & WB\_valid;

    assign rf\_wdest = wdest;

    assign rf\_wdata = mem\_result;

//-----{WB->regfile 信号}end

//-----{展示 WB 模块的 PC 值}begin

    assign WB\_pc = pc;

//-----{展示 WB 模块的 PC 值}end

endmodule

七、结果展示

7.1 测试所用汇编程序详述

为了验证多周期CPU的正确性，我设计了以下汇编程序代码段，该程序代码段共有39条指令，含括了所设计的三种类型的指令。具体的测试汇编程序如表7.1所示。

**表7.1：**测试所用汇编程序详述

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **指令地址** | **汇编指令** | **结 果描 述** | **机器指令的机器码** | |
| **16 进制** | **二进制** |
| 00H | addiu $1, $0,#1 | [$1] = 0000\_0001H | 24010001 | 0010\_0100\_0000\_0001\_0000\_0000\_0000\_0001 |
| 04H | sll $2, $1,#4 | [$2] = 0000\_0010H | 00011100 | 0000\_0000\_0000\_0001\_0001\_0001\_0000\_0000 |
| 08H | addu $3, $2,$1 | [$3] = 0000\_0011H | 00411821 | 0000\_0000\_0100\_0001\_0001\_1000\_0010\_0001 |
| 0CH | srl $4, $2,#2 | [$4] = 0000\_0004H | 00022082 | 0000\_0000\_0000\_0010\_0010\_0000\_1000\_0010 |
| 10H | slti $25,$4,#5 | [$25] = 0000\_0001H | 28990005 | 0010\_1000\_1001\_1001\_0000\_0000\_0000\_0101 |
| 14H | subu $5, $3,$4 | [$5] = 0000\_000DH | 00642823 | 0000\_0000\_0110\_0100\_0010\_1000\_0010\_0011 |
| 18H | sw $5, #20($0) | Mem[0000\_0014H] = 0000\_000DH | AC050014 | 1010\_1100\_0000\_0101\_0000\_0000\_0001\_0100 |
| 1CH | nor $6, $5,$2 | [$6] = FFFF\_FFE2H | 00A23027 | 0000\_0000\_1010\_0010\_0011\_0000\_0010\_0111 |
| 20H | or $7, $6,$3 | [$7] = FFFF\_FFF3H | 00C33825 | 0000\_0000\_1100\_0011\_0011\_1000\_0010\_0101 |
| 24H | xor $8, $7,$6 | [$8] = 0000\_0011H | 00E64026 | 0000\_0000\_1110\_0110\_0100\_0000\_0010\_0110 |
| 28H | sw $8, #28($0) | Mem[0000\_001CH]= 0000\_0011H | AC08001C | 1010\_1100\_0000\_1000\_0000\_0000\_0001\_1100 |
| 2CH | slt $9, $6,$7 | [$9]=0000\_0001H | 00C7482A | 0000\_0000\_1100\_0111\_0100\_1000\_0010\_1010 |
| 30H | addiu $1, $0,#8 | [$1] = 0000\_0008H | 24010008 | 0010\_0100\_0000\_0001\_0000\_0000\_0000\_1000 |
| 34H | lw $10,#20($1) | [$10]=0000\_0000H | 8C2A0014 | 1000\_1100\_0010\_1010\_0000\_0000\_0001\_0100 |
| 38H | and $11,$2,$1 | [$11]=0000\_0000H | 00415824 | 0000\_0000\_0100\_0001\_0101\_1000\_0010\_0100 |
| 3CH | sw $11,#28($1) | Mem[0000\_0024H]= 0000\_0000H | AC2B001C | 1010\_1100\_0010\_1011\_0000\_0000\_0001\_1100 |
| 40H | sw $4, #16($1) | Mem[0000\_0018H]= 0000\_0004H | AC240010 | 1010\_1100\_0010\_0100\_0000\_0000\_0001\_0000 |
| 44H | jal #25 | 跳转到 64H,[$31] = 0000\_0048 H | 0C000019 | 0000\_1100\_0000\_0000\_0000\_0000\_0001\_1001 |
| 48H | lui $12,#12 | [$12] = 000C\_0000H | 3C0C000C | 0011\_1100\_0000\_1100\_0000\_0000\_0000\_1100 |
| 4CH | srav $26,$12,$2 | [$26] = 0000\_000CH | 004CD007 | 0000\_0000\_0100\_1100\_1101\_0000\_0000\_0111 |
| 50H | sllv $27,$26,$1 | [$27] = 0000\_0018H | 003AD804 | 0000\_0000\_0011\_1010\_1101\_1000\_0000\_0100 |
| 54H | jalr $27 | 跳转到 18H ,[$31] = 0000\_0064H | 0360F809 | 0000\_0011\_0110\_0000\_1111\_1000\_0000\_1001 |
| 58H | sb $26,#5($3) | MEM[0000\_0016H]= 000C\_000DH | A07A0005 | 1010\_0000\_0111\_1010\_0000\_0000\_0000\_0101 |
| 5CH | sltu $13,$3,$3 | [$13] = 0000\_0000H | 0063682B | 0000\_0000\_0110\_0011\_0110\_1000\_0010\_1011 |
| 60H | bgtz $13,#3 | 不跳转 | 1DA00003 | 0001\_1101\_1010\_0000\_0000\_0000\_0000\_0011 |
| 64H | sllv $14,$6,$4 | [$14] =FFFF\_FE20H | 00867004 | 0000\_0000\_1000\_0110\_0111\_0000\_0000\_0100 |
| 68H | sra $15,$14,#2 | [$15] =FFFF\_FF88H | 000E7883 | 0000\_0000\_0000\_1110\_0111\_1000\_1000\_0011 |
| 6CH | srlv $16,$15,$1 | [$16] =00FF\_FFFFH | 002F8006 | 0000\_0000\_0010\_1111\_1000\_0000\_0000\_0110 |
| 70H | srav $16,$15,$1 | [$16] =FFFF\_FFFFH | 002F8007 | 0000\_0000\_0010\_1111\_1000\_0000\_0000\_0111 |
| 74H | addiu $11,$0,#140 | [$11] = 0000\_008CH | 240B008C | 0010\_0100\_0000\_1011\_0000\_0000\_1000\_1100 |
| 78H | lw $28,#3($10) | [$28] = 0000\_0000H | 8D5C0003 | 1000\_1101\_0101\_1100\_0000\_0000\_0000\_0011 |
| 7CH | sb $15,#8($5) | Mem[0000\_0015H] = 0000\_0088H | A0AF0008 | 1010\_0000\_1010\_1111\_0000\_0000\_0000\_1000 |
| 80H | lb $18,#8($5) | [$18] =FFFF\_FF88H | 80B20008 | 1000\_0000\_1011\_0010\_0000\_0000\_0000\_1000 |
| 84H | lbu $19,#8($5) | [$19] = 0000\_0088H | 90B30008 | 1001\_0000\_1011\_0011\_0000\_0000\_0000\_1000 |
| 88H | sltiu $24,$15,#0xFFFF | [$24] = 0000\_0001H | 2DF8FFFF | 0010\_1101\_1111\_1000\_1111\_1111\_1111\_1111 |
| 8CH | andi $20,$15,#0xFFFF | [$20] = 0000\_FF88H | 31F4FFFF | 0011\_0001\_1111\_0100\_1111\_1111\_1111\_1111 |
| 90H | ori $21,$15,#0xFFFF | [$21] =FFFF\_FFFFH | 35F5FFFF | 0011\_0101\_1111\_0101\_1111\_1111\_1111\_1111 |
| 94H | xori $22,$15,#0xFFFF | [$22] = FFFF\_0077H | 39F6FFFF | 0011\_1001\_1111\_0110\_1111\_1111\_1111\_1111 |
| 98H | j #00H | 跳转指令 00H | 08000000 | 0000\_1000\_0000\_0000\_0000\_0000\_0000\_0000 |

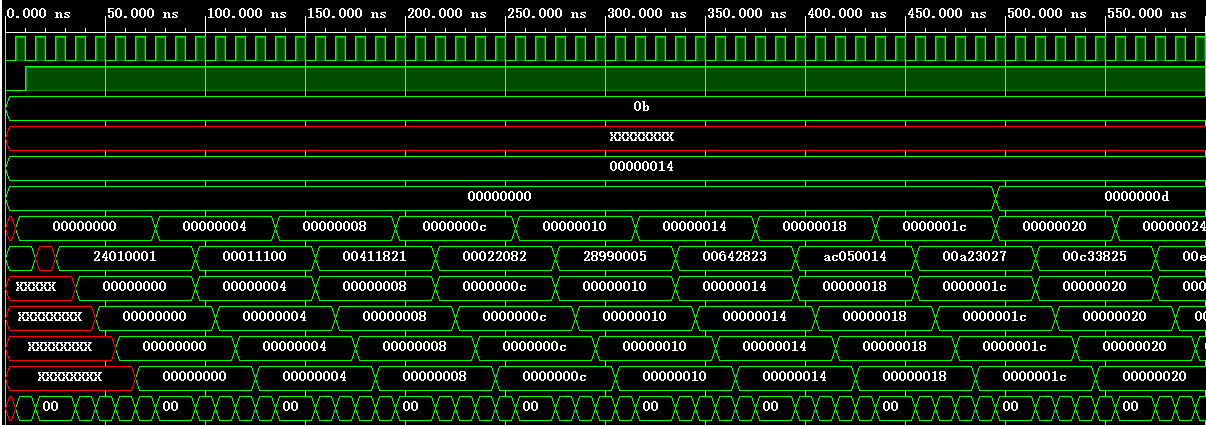
7.2 仿真波形图

为了进行功能仿真，这里我编写了一个testbench（仿真平台），由于代码数量过多，不便于展示寄存器堆和存储器的数据变化，这里仅选取11号寄存器和地址为0000\_0014存储单元进行数据的勘测。数据变化在上板验证能够更好地体现，这里着重验证汇编程序指令的流动。

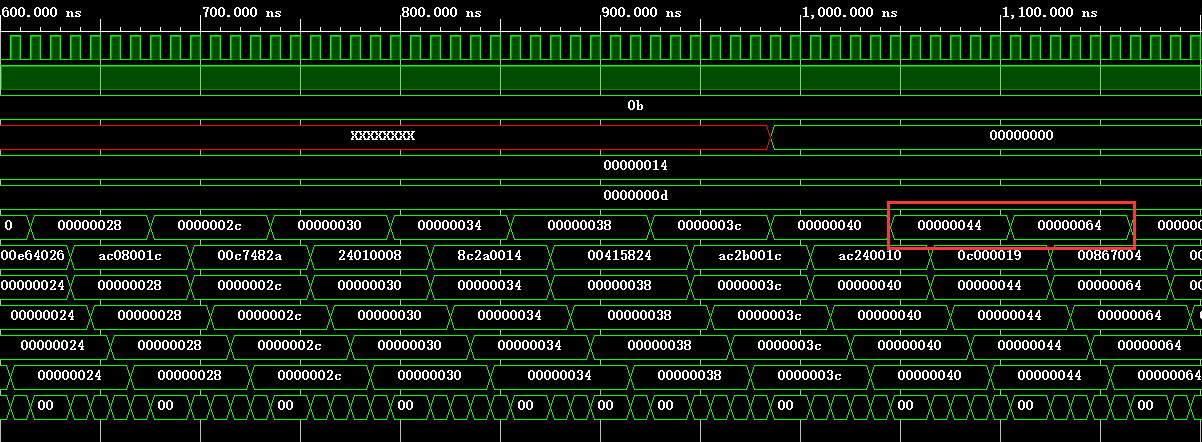
下图7.1-7.4为执行以上40条指令的波形图，每幅图运行600ns。从图7.1-7.4，我们不难看出指令确实一条条的从inst\_rom中读取了出来，以上波形图的第八行即为所取指令的六进制编码，代码的读取无误。其次指令每个功能段的PC值也会正常地自增，数据流动无误。

而从图7.2和7.4我们也可以看出程序的转移类指令也是准确无误的。图7.2当指令执行到jal #25（0C000019）时，PC值直接跳转到了0000\_0064H。同样地，图7.4当指令执行到最后一条指令j #00H（08000000）时，PC值直接跳转到了0000\_0000H，即程序开头。之后程序将以此循环运行。

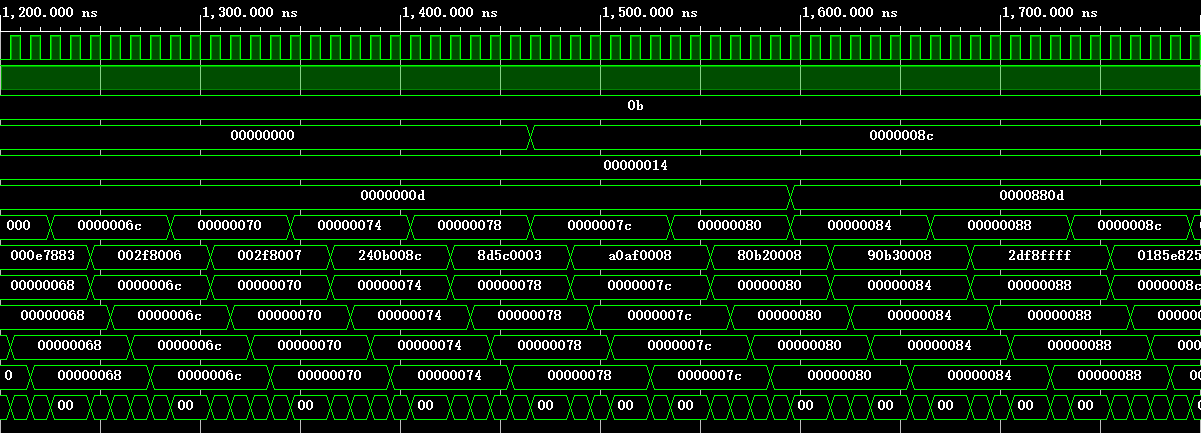
**图7.1：**仿真波形图1



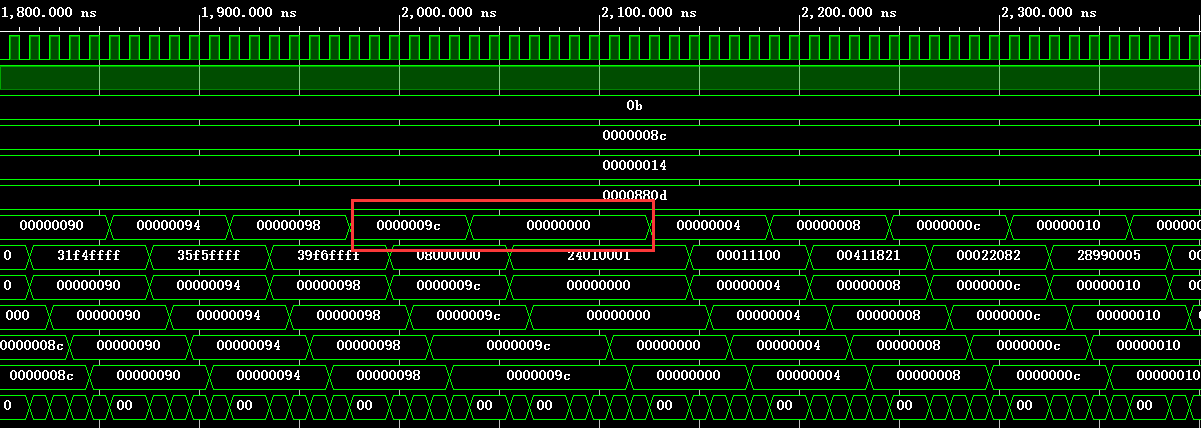
**图7.2：**仿真波形图2



**图7.3：**仿真波形图3

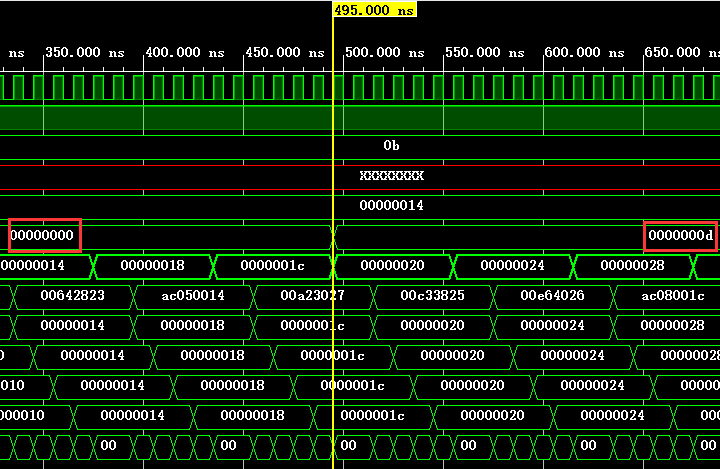


**图7.4：**仿真波形图4

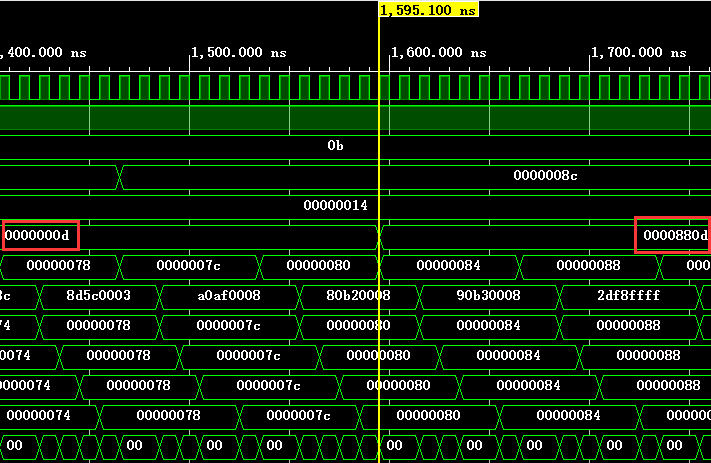


从下图7.5和7.6，我们可以看出指令执行到sw $5, #20($0)时，指令写内存单元0000\_0014H，内存单元的值从原来的0000\_0000H变成了0000\_000DH。而当指令执行到sb $15,#7($5)时，指令再次写内存单元0000\_0014H,内存单元从原来的0000\_000DH又变成了0000\_8800H。可以看出数据内存单元的读写是没有问题的。

**图7.5：**仿真波形图5

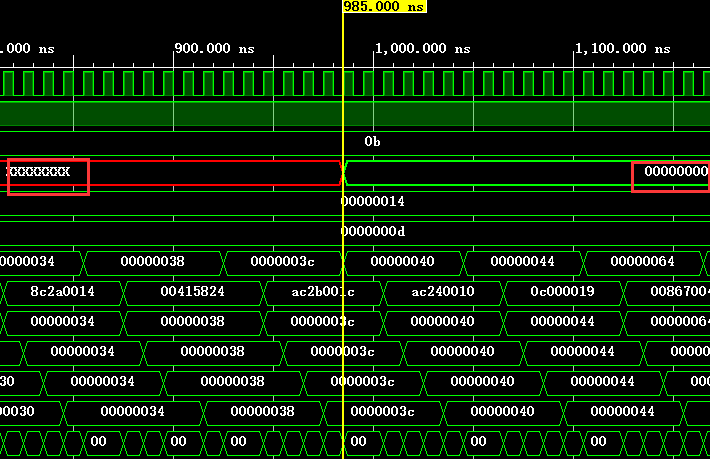


**图7.6：**仿真波形图6

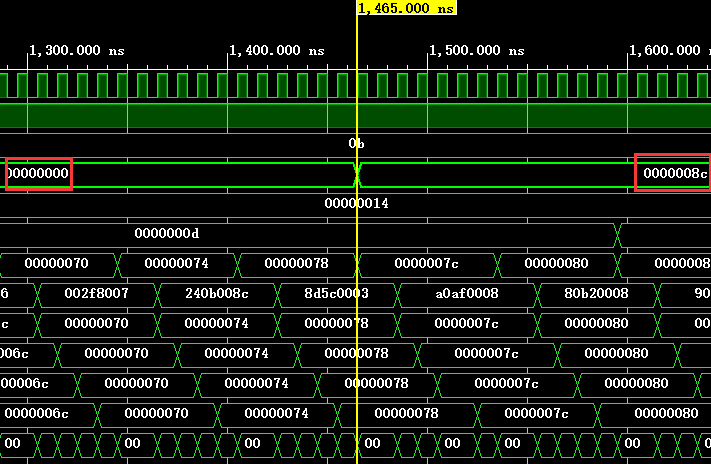


从下图7.7和7.8，我们可以看出指令执行到and $11,$2,$1时，指令写内存单元11号寄存器单元，寄存器单元的值从原来的XXXX\_XXXX变成了0000\_0000H。而当指令执行到addiu $11,$0,#140时，指令再次写11号寄存器单元,寄存器单元从原来的0000\_0000H又变成了0000\_008CH。可以看出寄存器堆的读写也是没有问题的。

**图7.7：**仿真波形图7



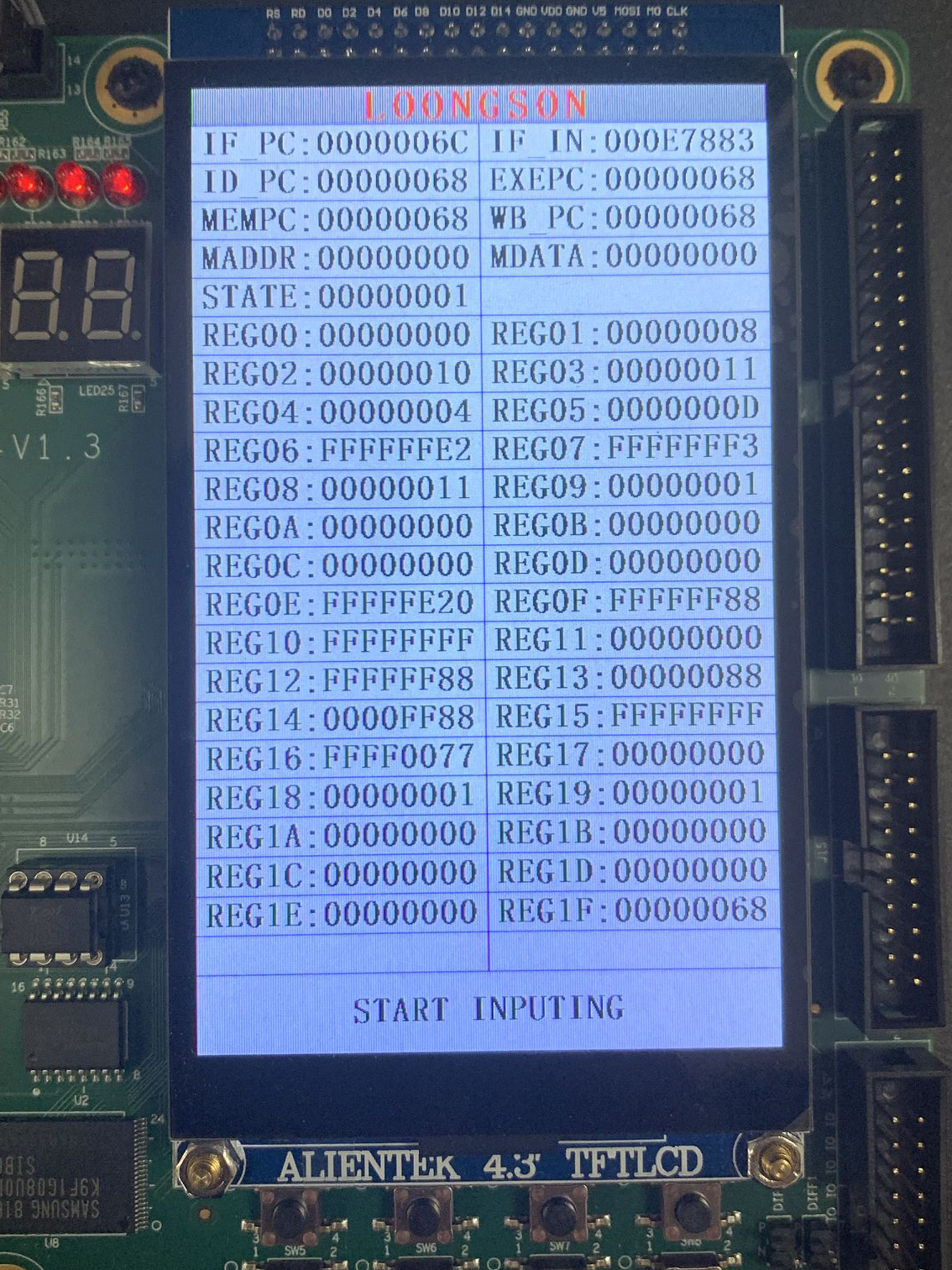
**图7.8：**仿真波形图8



7.3 上板验证图

这里使用图7.9所示的上板验证图来验证程序的正确性。

**图7.9：**上板验证图



从上图我们可以看出，目前刚刚取出的指令是000E7883,也就是 指令sra $15,$14,#2这条，当前IF\_PC为6CH，而其他段的PC均为68H，说明指令刚进行到取指，还没有取出下一条指令。程序的流动逻辑正确。

其次寄存器的值也符合程序的运行存储逻辑，由于涉及到的寄存器过多，这里便不再一一赘述。需要说明的是上图展示的值是程序第二次循环时的情况，即程序其实已经运行过一次，从最后一条j #00H跳到第一条指令，所以说被访问的寄存器的值都已被写入。

八、心得总结

本次硬件课程设计个人感觉还是有相当难度的。一个是对Verilog语言的不熟悉，另一个则是编码前设计思路的整理与确定。和上学期制作编译器的软件课设一样，前期整理思路花了很多时间，思路确定后代码的编写反而显得要简单一些。

本次课设个人觉得有两大难点：一是跳转指令的设计，跳转指令作为所有指令中最为特殊的存在，其会直接改变程序数据流动的方向，改变程序PC值，如何正确的改变PC值，在什么时候改变PC值都是需要考虑的。二是内存单元的访存，本来打算像单周期CPU那样直接用代码模拟内存RAM，结果难以实现，最后通过查找相关资料了解了可以直接用创建IP的方式创建RAM和ROM，这样就显得简单了很多。

总之，这次硬件课程设计是组原和系统结构实验的再一次加强。可以看到从计算机逻辑实验、组原实验再到系统结构实验，是一个层层递进的过程。在这个过程中，我对计算机底层逻辑实现的理解不断深入，对于Verilog语言的掌握也一步步加深。该课设很有趣，但由于本人水平不足，没有制作更加高级的流水线CPU，而且制作过程中借鉴了不少网络上的代码和思路。

本次课设应该是大学期间最后一次接触Verilog了，但这几次与Verilog编程的简单接触让我感受到了计算机底层编程的魅力以及计算机实现的神奇。在摸索中学习确实是否有意思，每次将报错的代码改正确总有一种说不上来的成就感。

最后，感谢老师在本次课程设计中的指导和付出的努力，祝老师身体健康，科研顺利！