硬件课程设计(I)实验报告

922106840127 刘宇翔

目录

一、实验目的

1. 在单周期 CPU 实验完成的提前下，理解多周期和五级流水的概念。

2. 熟悉并掌握多周期及流水 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

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。

三、实验任务

1.本次设计是对单周期 CPU 实验的拔高，前期的实验准备同单周期 CPU 的实验，在单周期 CPU 中只要求实现了五条指令，但此处要求扩展到 25 条以上指令，且必须包括四大类型：**传送、运算、访存、控制转移。**

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。

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，效率更高，依然相当于是一个周期完成一条指令，因此 CPU 可以运行得更快。

本次实验就是将前面所实现的单周期 CPU 转化为多周期或带五级流水的 CPU，并扩展指令到 25 条。

2.基于单周期 CPU 设计框图，将整体逻辑划分为取指（IF）、译码（ID）、执行（EX）、访存（MEM）、写回（WB）五个功能单元。各单元分别在一个时钟周期内完成相应操作，并将结果通过时钟控制的中间寄存器锁存，作为下一单元的输入。

3.绘制多周期 CPU 框图，展示指令按周期依次流经 IF→ID→EX→MEM→WB。图中所有 “clk” 标注的箭头均指向中间锁存器，用于将各功能单元输出在时钟沿时存入寄存器；WB 单元的时钟信号对应寄存器堆的写操作。

4.本设计沿用先前实验中搭建的 ALU、寄存器堆、指令 ROM、数据 RAM 模块。建议采用 IP 实例化的同步 ROM/RAM，以贴近实际同步存取特性。

5.由于同步存储器在地址到达后需延迟一拍才能输出数据，IF 与 MEM 访问均需两拍时钟完成。该延迟应在设计与仿真中充分考虑。

6.使用 Verilog 实现上述多周期 CPU 及其各子模块，并编写测试平台。

7.对设计进行仿真验证，确保各功能单元及指令流水的波形与预期一致。

8.将 CPU 模块封装并集成到顶层工程（如图 3.2），调用 LCD 触摸屏 IP，实时监控内部状态（包括 32 × 寄存器文件、各模块 PC 值等），并通过触摸输入指定 RAM 地址，从调试端口读取并显示数据，实现板上在线调试与验证。

四、实验整体设计与设计框图

1.CPU 架构设计概述

本次设计的核心是一款遵循多周期执行模型的 32 位 MIPS 处理器。尽管在逻辑上，我们将处理器的执行流程分解为取指（IF）、译码（ID）、执行（EXE）、访存（MEM）、写回（WB）五个阶段，这与经典的 MIPS 五段式流水线在功能划分上具有相似性，但其底层的运行范式存在本质区别。

流水线架构通过空间复用，让多条指令并行处于不同阶段；而我们的多周期架构则通过时间复用，让单条指令在连续的多个时钟周期内，依次独占整个数据通路的不同部分，其执行脉络由一个中央有限状态机（FSM）统一调度。

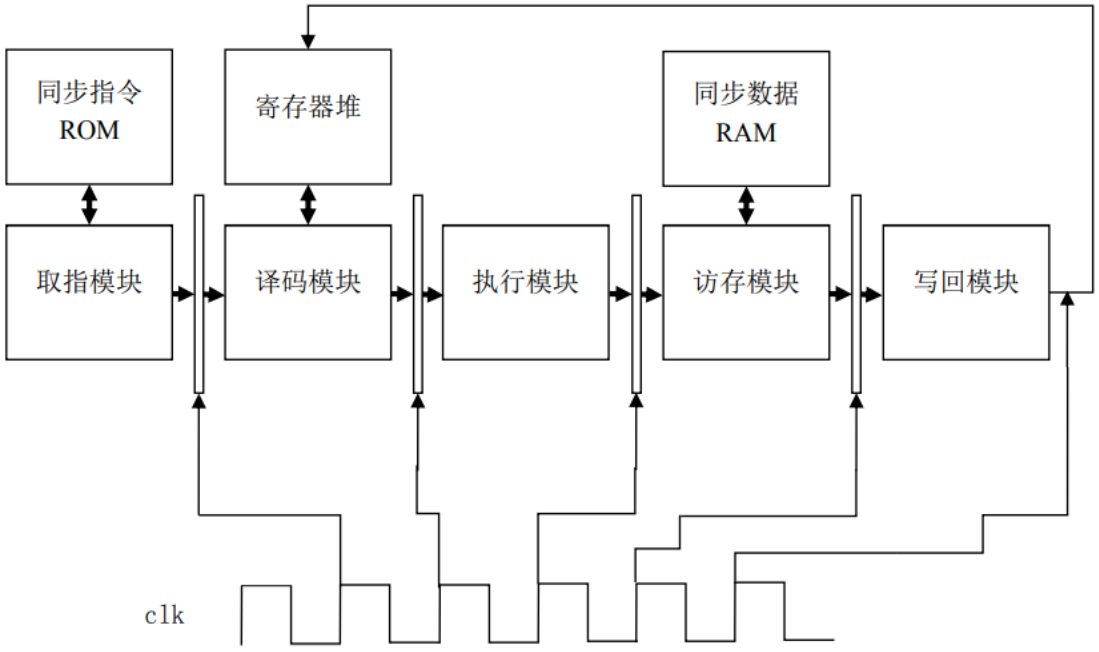


图 1 多周期 CPU 的大致框图

2.各阶段具体设计与构想

（1）取指（IF）阶段

本阶段的核心任务是从指令存储器（Instruction ROM）中获取指令。其关键部

件为程序计数器（PC），PC 的更新并非简单的时钟驱动自增，而是受控于中央 FSM 的状态转换逻辑。当一条指令完成或发生跳转时，FSM 才会发出 next_fetch 信号，授权 PC 更新至下一地址（顺序执行的 PC+4 或分支跳转的目标地址）。

由于本设计采用 FPGA 片上同步 Block RAM 作为指令存储器，其固有的单周期读延迟特性，决定了取指操作必须跨越两个时钟周期：第一周期发出 PC 地址，第二周期才能接收到有效的指令码。为保证数据同步，在 IF 阶段的末尾，当前 PC 值与获取的指令码会被锁存到 IF_ID 级间寄存器中。

（2）译码（ID）阶段

此阶段是处理器的“控制中枢”与“指令解析中心”。当 IF_ID 寄存器锁存稳定后，ID 阶段开始工作。它承担双重关键职责：其一，解析指令中的 rs 和 rt 字段，作为地址访问寄存器堆（Register File）以获取源操作数；其二，也是多周期设计的精髓所在，是根据指令的 opcode 和 funct 字段，生成一组贯穿后续所有阶段的、完整的“控制字（Control Word）”。

该控制字包含了 ALU 的操作类型、操作数来源、内存的读写模式、写回寄存器的目标以及最终的写使能信号。这使得后续的 EXE、MEM、WB 阶段更像是“被动”的执行单元，严格遵循 ID 阶段预设的指令剧本进行操作，体现了典型的“集中译码，分布式执行”的设计哲学。

（3）执行（EXE）阶段

作为 CPU 的“算术与逻辑核心”，本阶段主要由算术逻辑单元（ALU）构成。ALU 的通用性在此得到充分体现，它的输入源于 ID 阶段复杂的多路选择器逻辑，能够灵活地选择寄存器值或经过符号扩展的立即数作为操作数。

其输出 alu_result 的用途也同样多样化：对于算术/逻辑指令，它就是最终的计算结果；对于 Load/Store 指令，它则作为有效地址（EA）被送往访存阶段；对于分支指令，它则用于计算分支目标。可以说，EXE 阶段是数据处理和地址计算的汇集点。

（4）访存（MEM）阶段

本阶段是 CPU 与数据存储器（Data RAM）交互的唯一接口，其功能具有显著的二元性。对于 lw、sw 等访存指令，它会根据 ID 阶段传来的控制信号，向 RAM 发出读/写命令，并处理相应的数据传输。

对于其他非访存指令，它则扮演一个“透明通道”的角色，仅将来自 EXE 阶段的 alu_result 原样传递下去。一个至关重要的硬件结构位于此阶段的出口：一个二路选择器，它根据当前指令是否为 load，来决定最终送往 WB 阶段的数据源——是来自 Data RAM 的 load_result，还是来自 ALU 的 alu_result。

同时，为应对同步数据 RAM 的读延迟，本阶段也内嵌了暂停逻辑，确保在执行 load 指令时能额外停留一个周期，以保障数据读取的正确性。

（5）写回（WB）阶段

作为指令执行流的终点，WB 阶段的功能纯粹而关键——“提交结果”。它接收来自 MEM 阶段的最终数据以及目标寄存器地址，在中央 FSM 进入 WB 状态的有效周期内，将数据精确地写回寄存器堆。

此操作由写使能信号 rf_wen 严格控制，确保了只有在指令执行的最后时刻，处理器的体系结构状态（Architectural State）才会发生改变，从而保证了指令执行的原子性。

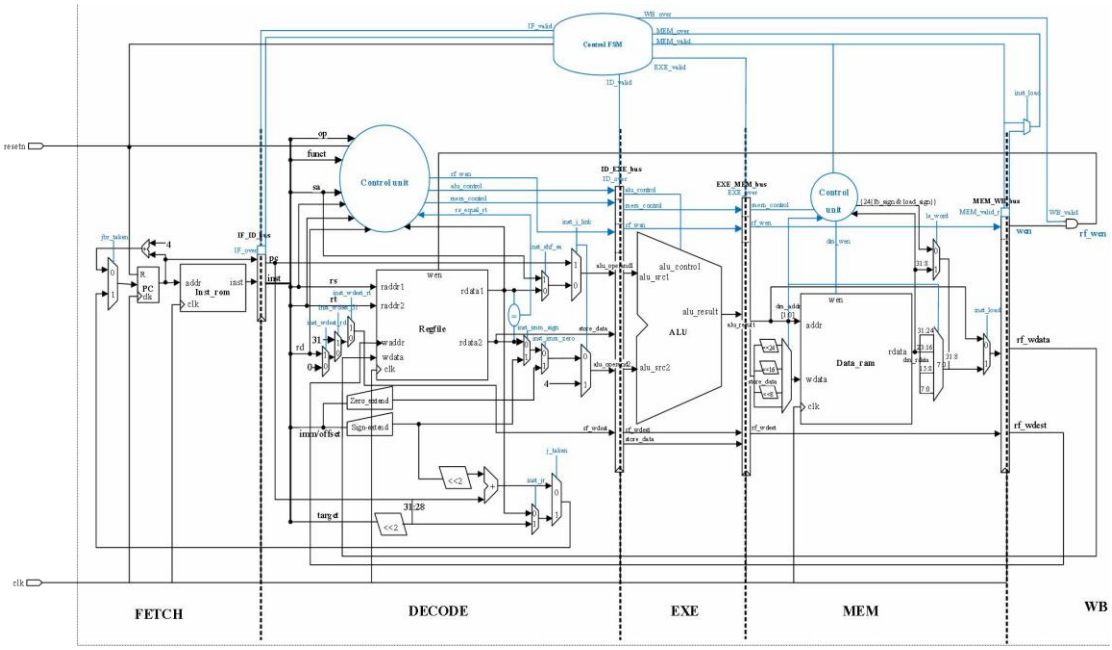


图 2 多周期 CPU 的整体设计框图

五、多周期 CPU 指令归纳表

表 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	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|00000|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|00000|000100	[rs]	[rt]	[rd]	逻辑可变左移
	sra rd,rt,shf	000000|00000|rt|rd|shf|000011		[rt]	[rd]	算术右移
	srav rd,rt,rs	000000|rs|rt|rd|00000|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|00000|000110	[rs]	[rt]	[rd]	逻辑可变右移
	move rd,rs	000000|rs|00000|rd|00000|10010	[rs]		[rd]	传送指令
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|offset	[rs]			大于等于0跳转
	bgtz rs,offset	000111|rs|00000|offset	[rs]			大于0跳转
	blez rs,offset	000110|rs|00000|offset	[rs]			小于等于0跳转
	bltz rs,offset	000001|rs|00000|offset	[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

本次 CPU 设计遵循经典的五级流水线思想（取指、译码、执行、访存、写回），但通过一个中央状态机控制，使得每条指令在多个时钟周期内分步完成。报告将首先分析作为控制核心的 CPU 顶层模块，随后依次深入取指（IF）、译码（ID）、执行（EXE）、访存（MEM）和写回（WB）五个阶段的硬件模块。由于实际编码中，为扩展 CPU 指令种类与条数导致的重复性工作较多，因而我计划通过对每个模块的功能进行系统性阐述，并结合关键 Verilog 代码的讲解，本报告将清晰地展示该多周期 CPU 的架构、工作流程以及核心设计细节。

1．CPU 顶层模块 (multi_cycle_cpu.v)

1.1 功能阐述

CPU 顶层模块（multi_cycle_cpu）是整个处理器的“神经中枢”和“骨架”。它的核心功能有三点：

1. **中央状态机控制**：它内部实现了一个核心的状态机，用以精确控制指令执行的五个阶段（FETCH, DECODE, EXE, MEM, WB）。CPU 的“多周期”特性正是由这个状态机驱动的，状态机根据当前阶段任务是否完成（_over 信号）来决定是停留在当前状态还是跳转到下一个状态。
2. **构建数据通路**：它定义了连接五级流水线阶段的“总线”（Buses），并通过寄存器（IF_ID_bus_r, ID_EXE_bus_r 等）在每个时钟周期锁存中间结果。这确保了在指令从一个阶段流向下一个阶段时，数据的稳定性和同步性。
3. **模块化集成**：它像一块主板，将所有独立的硬件单元（如取指模块、译码模块、ALU、寄存器堆、数据/指令内存等）实例化，并将它们通过总线和控制信号正确地连接起来，形成一个完整、协同工作的 CPU 系统。

1.2 关键代码讲解

1. 核心状态机

1.1 CPU 顶层模块：状态机

```
// 状态机状态定义

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
//  状态机时序逻辑：在时钟上升沿更新当前状态
always @ (posedge clk)
begin
    if (!resetn) begin
        state <= IDLE; //  复位时回到 IDLE
    end
    else begin
        state <= next_state; //  正常工作时，进入下一状态
    end
end
//  状态机组合逻辑：根据当前状态和完成信号，决定下一状态
always @ (*)
begin
    case (state)
        // ... (其他状态)
        DECODE:
        begin
            if (ID_over)
            begin //  如果译码完成
                //  如果是“非链接”的跳转指令，直接跳回取指
                //  否则，按顺序进入执行
                next_state = jbr_not_link ? FETCH : EXE;
            end
```

```
                else
                begin
                    next_state = DECODE; //  否则停留在译码
                end
            end
            // ... (其他状态)
            WB:
            begin
                if (WB_over)
                begin
                    next_state = FETCH; //  写回完成，开始取下一条指令
                end
                else
                begin
                    next_state = WB; //  否则停留在写回
                end
            end
        endcase
    end
```

state 寄存器保存 CPU 的当前工作阶段。next_state 决定了下一个时钟周期 state 将变成什么。状态机的流转是线性的（FETCH -> DECODE -> EXE -> MEM -> WB），但有两个关键的特殊路径：

1.WB -> FETCH：一条指令执行完毕后，返回取指阶段，开始下一条指令。

2.DECODE -> FETCH：这是一个优化。对于像 beq, bne, j, jr 这样不涉及计算、访存和写回的跳转指令（jbr_not_link 为真），在译码阶段确定了跳转地址后，就可以直接返回取指阶段去取新的指令，从而跳过 EXE, MEM, WB 三个阶段，提高了执行效率。

而_over 信号是每个阶段模块的输出，它告诉状态机“我的工作做完了”，是状态跳转的关键前提。

2. 级间总线与锁存寄存器

这些总线和寄存器构成了级间数据流动的管道。

1.2 CPU 顶层模块：级间总线与锁存寄存器

```
// 定义 IF 级到 ID 级的总线和其锁存寄存器
wire [ 63:0] IF_ID_bus;
reg [ 63:0] IF_ID_bus_r;
// IF 到 ID 的锁存逻辑
always @(posedge clk)
begin
    // 当 IF 级完成其工作时 (IF_over 为高)
    // 就将 IF 级输出的总线数据锁存到寄存器 IF_ID_bus_r 中
    if(IF_over)
    begin
    IF_ID_bus_r <= IF_ID_bus;
    end
end

// (ID->EXE, EXE->MEM, MEM->WB 的锁存逻辑与此类似
```

IF_ID_bus 是一个线网（wire），它实时反映取指模块的输出。

IF_ID_bus_r 是一个寄存器（reg），它起到了“防火墙”的作用。当状态机从 FETCH 进入 DECODE 时，IF_over 信号会触发一次锁存，将当时的 PC 值和指令码存入 IF_ID_bus_r。

这样做的好处是：即使取指模块已经开始为下一条指令工作，译码模块仍然可以安稳地使用 IF_ID_bus_r 中保存好的、属于当前指令的数据，确保了数据处理的独立性和正确性。

2.取指模块 (fetch.v)

2.1 功能阐述

取指（IF）是指令执行的第一步。该模块的核心功能是管理程序计数器（PC），并根据 PC 的值向指令存储器（inst_rom）请求指令。具体来说，它负责：

1. 维护 PC 值，在每个时钟周期计算出下一条指令的地址。
2. 处理跳转（Branch/Jump）逻辑。当译码阶段传来跳转信号时，PC 需要更新为跳转目标地址，而不是简单地+4。
3. 将当前 PC 值和从内存中取出的指令码打包，通过 IF_ID_bus 发送给下一级。

2.2 关键代码讲解

1. PC 更新逻辑

这是取指模块最核心的部分，决定了程序的执行流。

2.1 取值模块：PC 更新逻辑

```
// PC 的下一状态：顺序执行或跳转
wire [31:0] next_pc;
wire [31:0] seq_pc;
assign seq_pc[31:2] = pc[31:2] + 1'b1; // 顺序执行地址 (PC+4)
assign next_pc = jbr_taken ? jbr_target : seq_pc; // 核心选择逻辑
// PC 寄存器更新
always @(posedge clk)
begin
    if (!resetn)
    begin
        pc <= `STARTADDR; // 复位时，PC 指向程序起始地址
    end
    else if (next_fetch) // 收到“取下一条指令”的信号
```

```
    begin
        pc <= next_pc; // 更新 PC
    end
end
```

seq_pc 代表顺序执行流，即 PC+4。

jbr_taken 和 jbr_target 来自译码模块的 jbr_bus，分别代表“是否跳转”和“跳转到哪里”。

assign next_pc = jbr_taken ? jbr_target : seq_pc; 这行代码是一个关键的二路选择器。如果 jbr_taken 为真，next_pc 就被设为跳转目标地址；否则，它就是 PC+4。

PC 寄存器并不是每个时钟周期都更新。next_fetch 信号 ((state==DECODE & ID_over & jbr_not_link)|(state==WB & WB_over)) 作为更新的使能信号，确保 PC 只在一条指令执行完毕或一个非链接跳转发生后才更新，这符合多周期 CPU 的工作模式。

2. 向下一级传递数据

2.2 取值模块：向下一级传递数据

```
// IF->ID 总线
assign IF_ID_bus = {pc, inst};
```

此行代码非常直观，它将 32 位的 PC 值和 32 位的指令码 inst 拼接成一个 64 位的总线信号，传递给下一级。这样做可以让译码模块不仅知道指令是什么，还知道这条指令在内存中的位置（PC 值），这对于计算某些跳转地址和实现 JAL/JALR 等链接指令至关重要。

3.译码模块 (decode.v)

3.1 功能阐述

译码（ID）模块是 CPU 的“控制单元”。它接收来自取指模块的指令码，并执行以下复杂任务：

1. **指令解析**：将 32 位的指令码分解成操作码（op）、功能码（funct）、寄存器地址（rs, rt, rd）和立即数（imm）等字段。
2. **指令识别**：根据操作码和功能码，判断出具体是 36 条指令中的哪一条（如 ADDU, LW, BEQ 等）。

3. **生成控制信号**：基于识别出的指令，生成一系列控制信号，打包到 ID_EXE_bus 中，用于指导后续的执行、访存和写回阶段的工作。例如，它会告诉 ALU 要做加法还是逻辑运算，告诉访存阶段是读内存还是写内存。

4. **读取寄存器**：将指令中的 rs 和 rt 字段作为地址，从寄存器堆（RegFile）中读取操作数。

5. **处理分支**：计算分支指令的目标地址，并判断分支条件是否满足，将结果通过 jbr_bus 反馈给取指模块。

3.2 关键代码讲解

1. 指令识别与归类

这是译码模块的基础，通过大量的逻辑与和比较，将二进制码映射为具体的指令。

3.1 译码模块：指令识别与归类

```
// 解析指令字段
assign op = inst[31:26];
assign rs = inst[25:21];
// ...
// 识别具体指令（以 ADDU 为例）
// ADDU: op 码为 0, sa 域为 0, funct 码为'100001'
assign inst_ADDU = op_zero & sa_zero & (funct == 6'b100001);
// 将具体指令按功能归类（以加法为例）
assign inst_add = inst_ADDU | inst_ADDIU | inst_load | inst_store | inst_j_link;
```

首先，代码会将 inst 的各个位域提取出来。然后，通过组合逻辑 assign inst_ADDU = ... 来识别特定指令。例如，inst_ADDU 信号只有在 op, sa, funct 字段都满足 MIPS 指令集对 addu 指令的规定时才为高电平。

最后，为了简化后续阶段的控制，代码将功能相似的指令归为一类。例如，inst_add 信号代表所有需要 ALU 执行加法运算的指令。LW/SW 指令计算地址需要加法，JAL/JALR 指令计算返回地址 PC+4（在此设计中）也需要加法。这种归类使得为 ALU 生成的控制信号更为简洁。

2. ALU 操作数选择

根据指令类型，为 ALU 的两个输入端口选择正确的数据源。

3.2 译码模块：ALU 操作数选择

```
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;
```

这是一个典型的三级/四级嵌套的条件操作符，功能上等价于一个多路选择器（Mux）。

alu_operand1 的选择逻辑：

如果是链接跳转指令 (inst_j_link)，源操作数 1 是当前的 PC 值（用于计算 PC+4）。如果是立即数移位指令 (inst_shf_sa)，源操作数 1 是指令中的 sa 域。否则，源操作数 1 是通用情况，即从寄存器堆读出的 rs_value。

alu_operand2 的选择逻辑：

如果是链接跳转，源操作数 2 是立即数 4。如果是零扩展的立即数指令 (inst_imm_zero)，源操作数 2 是 16 位立即数进行零扩展。如果是符号扩展的立即数指令 (inst_imm_sign)，源操作数 2 是 16 位立即数进行符号位扩展。否则，源操作数 2 是通用情况，即从寄存器堆读出的 rt_value。

这段代码完美体现了 CPU 如何根据指令动态配置数据通路。

3. 向后续阶段打包控制信息

将所有译码结果打包，传递给下一级。

3.3 译码模块：向后续阶段打包控制信息

```
assign ID_EXE_bus = {alu_control, alu_operand1, alu_operand2,
// EXE 级信息
mem_control, store_data,                    // MEM 级信息
rf_wen, rf_wdest,                           // WB 级信息
pc};                                        // PC 值
```

ID_EXE_bus 是一条非常宽的总线，它像一个“指令信息包”，包含了这条指令从执行到写回所需的所有信息。

alu_control: 告诉 ALU 做什么运算。

mem_control: 告诉访存单元是读、是写、是读/写字节还是字。

rf_wen, rf_wdest: 告诉写回单元是否要写寄存器、以及写哪个寄存器。

这种设计使得后续的 EXE, MEM, WB 阶段的逻辑变得非常简单，它们只需“听从”译码阶段传来的指令即可，无需再次解析指令。

4.执行模块 (exe.v)

4.1 功能阐述

执行（EXE）模块是 CPU 的“计算核心”。它的结构相对简单，主要职责是：

1. 接收来自译码模块的 ALU 控制信号和操作数。
2. 调用 ALU（算术逻辑单元）完成指定的数学运算（如加、减）或逻辑运算（如与、或、移位）。
3. 将 ALU 的计算结果，连同从译码阶段传来的访存和写回控制信号，一起打包传递给访存（MEM）阶段。

4.2 关键代码讲解

1. ALU 的调用

这是执行模块的核心操作。

4.1 执行模块：ALU 的调用

```
// 解包从 ID->EXE 总线传来的数据
assign {alu_control, alu_operand1, alu_operand2, ...} = ID_EXE_bus_r;
// 实例化并连接 ALU 模块
alu alu_module(
    .alu_control (alu_control ), // 输入: ALU 控制信号
    .alu_src1    (alu_operand1), // 输入: ALU 操作数 1
    .alu_src2    (alu_operand2), // 输入: ALU 操作数 2
```

```
    .alu_result  (alu_result )   // 输出: ALU 计算结果
);
```

模块首先从上一级锁存的总线 ID_EXE_bus_r 中解包出自己需要的信息：alu_control，alu_operand1 和 alu_operand2。

然后，它将这些信号直接连接到已有的 alu 模块的对应端口。alu 模块本身是一个组合逻辑电路，它会立即根据输入计算出 alu_result。执行阶段的核心工作就是这一次 ALU 的调用。

2. 将信息传递给 MEM 级

4.2 执行模块：将信息传递给 MEM 级

```
// 打包 EXE->MEM 总线
assign  EXE_MEM_bus  =  {mem_control,  store_data,  //  MEM  级需要的 load/store 信息
alu_result,              // 本级产生的 ALU 运算结果
rf_wen, rf_wdest,        // WB 级需要的写回信息
pc};
```

EXE_MEM_bus 总线的内容与 ID_EXE_bus 非常相似，但有一个关键区别：它传递的是 alu_result（ALU 的计算结果），而不是 ALU 的操作数。

对于 Load/Store 指令，alu_result 将作为内存地址。对于计算类指令，alu_result 将是最终要写回寄存器的值。mem_control 和 rf_wen/rf_wdest 等控制信号在此阶段未被使用，只是被原样“接力”传递下去。

5.访存模块 (mem.v)

5.1 功能阐述

访存（MEM）模块是 CPU 与数据存储器（data_ram）交互的唯一接口。它的功能根据指令类型分为三类：

1. **Load 指令**：将来自执行阶段的 alu_result 作为地址，向 data_ram 发出读请求，并将读出的数据进行必要的处理（如字节读取时的符号扩展），然后传递给写回阶段。
2. **Store 指令**：将来自执行阶段的 alu_result 作为地址，将来自译码阶段的 store_data（即 rt_value）作为数据，向 data_ram 发出写请求。

3. **其他指令**：对于非访存指令，该模块不起作用，仅将 alu_result 和控制信号直接传递给写回阶段。

5.2 关键代码讲解

1. Store 指令写操作

5.1 访存模块：Store 指令写操作

```
// 内存写使能信号
always @ (*)
begin
    if (MEM_valid && inst_store) // 访存级有效,且为 store 操作
    begin
        if (ls_word) // 如果是 SW 指令(存字)
        begin
            dm_wen <= 4'b1111; // 4 个字节的写使能全开
        end
        else // 如果是 SB 指令(存字节)
        begin
            case (dm_addr[1:0]) // 根据地址的低 2 位决定写哪个字节
                2'b00 : dm_wen <= 4'b0001; // 写最低字节
                2'b01 : dm_wen <= 4'b0010;
                2'b10 : dm_wen <= 4'b0100;
                2'b11 : dm_wen <= 4'b1000; // 写最高字节
                default : dm_wen <= 4'b0000;
            endcase
        end
    end
    else
    begin
```

```
            dm_wen <= 4'b0000; // 其他情况不写
        end
    end
```

dm_wen 是一个 4 位的写使能信号，分别对应一个 32 位字中的 4 个字节。

sw 指令需要写入整个 32 位字，因此 dm_wen 为 4'b1111。

sb 指令只写一个字节。MIPS 架构是字节寻址的，dm_addr 的最低两位 [1:0] 决定了要操作的是 32 位对齐地址中的第几个字节。case 语句根据这 2 位的值，只使能对应字节的写信号，实现了精确的字节写入。

2. Load 指令的同步读延迟处理

这是体现多周期 CPU 特性和同步 RAM 交互的关键点。

5.2 访存模块：Load 指令的同步读延迟处理

```
// 由于数据 RAM 为同步读，发出地址后，下一拍才能获得数据
// 因此 load 指令在 MEM 阶段需要两拍
reg MEM_valid_r;
always @(posedge clk)
begin
    MEM_valid_r <= MEM_valid;
end
assign MEM_over = inst_load ? MEM_valid_r : MEM_valid;
```

设计使用的 RAM 是“同步读”的，意味着当你在时钟周期 N 发出一个读地址 dm_addr，数据 dm_rdata 要在时钟周期 N+1 的上升沿之后才有效。

MEM_valid 信号在 CPU 进入 MEM 状态的第一个周期变为高电平。

MEM_valid_r 是 MEM_valid 延迟一拍的结果。

assign MEM_over = inst_load ? MEM_valid_r : MEM_valid; 这行代码的含义是：

如果当前指令**不是** Load 指令，那么 MEM 阶段的任务一拍就能完成，所以当 MEM_valid 有效时，MEM_over 也立即有效，状态机可以进入 WB。

如果当前指令**是** Load 指令，MEM 阶段需要等待一拍来获取数据。因此 MEM_over 信号要等到 MEM_valid_r 变为高电平（即进入 MEM 状态的第二个周

期）才有效。这会使 CPU 状态机在 MEM 状态停留一个额外的时钟周期，以等待数据准备好。

3. 结果选择

5.3 访存模块：结果选择

```
// MEM 传到 WB 的 result 为 load 结果或 ALU 结果

assign mem_result = inst_load ? load_result : alu_result;
```

这是一个关键的选择器。它决定了最终要写回寄存器的数据来源。

如果是 Load 指令，那么应将从 data_ram 读出的 load_result 传递给 WB 级。

对于所有其他指令（计算类、跳转类等），应将 EXE 级计算出的 alu_result 传递给 WB 级。

6.写回模块 (wb.v)

6.1 功能阐述

写回（WB）是指令执行的最后一步，也是结构最简单的一级。它的唯一任务是：

1. 接收来自访存阶段的数据（mem_result）和写回控制信号（rf_wen, rf_wdest）。
2. 将这些信号传递给寄存器堆（RegFile），完成最终的数据写入操作。当 rf_wen 有效时，mem_result 的数据将被写入到地址为 rf_wdest 的寄存器中。

6.2 关键代码讲解

6.1 写回模块：执行流程

```
//-----{MEM->WB 总线}begin

assign {wen, wdest, mem_result, pc} = MEM_WB_bus_r;

//-----{MEM->WB 总线}end


//-----{WB->regfile 信号}begin

// 只有当 WB 级有效时，才真正使能寄存器堆的写操作
```

```
assign rf_wen = wen & WB_valid;

assign rf_wdest = wdest;

assign rf_wdata = mem_result;

//-----{WB->regfile 信号}end


//-----{WB 执行完成}begin

assign WB_over = WB_valid;

//-----{WB 执行完成}end
```

写回模块本身不进行任何计算或复杂的逻辑判断，它几乎是一个纯粹的“管道”。

assign rf_wen = wen & WB_valid; 是一个重要的保护措施。wen 信号是在译码阶段就已确定的是否需要写回的意图。而 WB_valid 保证了写操作只在状态机精确地处于 WB 状态的那个周期发生，防止了在错误的时间点污染寄存器堆。

assign WB_over = WB_valid; 表明 WB 阶段的操作非常简单，可以在一个时钟周期内完成。一旦状态机进入 WB 状态 (WB_valid 为高)，WB_over 也立即为高，通知状态机可以在下一个周期开始取新指令了。

七、程序测试与上板验证

7.1 课程设计测试所用汇编程序详述

为了全面、严谨地测试本次课程设计所实现的多周期 CPU，我设计了一套包含 37 条指令的测试程序。该程序不仅在执行逻辑上环环相扣，更在指令的选择上力求广泛和深入，旨在系统性地验证 CPU 的算术逻辑单元（ALU）、数据通路、控制器、存储器接口以及跳转逻辑等各项功能模块的正确性。

本测试程序共包含 38 条 MIPS 指令。指令的选取覆盖了 MIPS 指令集中的三大核心类型：**R 型指令**、**I 型指令**和 **J 型指令**，确保了 CPU 设计的绝大多数功能都得到了有效的验证。

1.指令的功能类型与分类统计

（1）R 型指令（寄存器-寄存器操作指令）

这类指令主要用于测试 ALU 在处理寄存器数据时的核心能力，包括算术、逻辑、比较和移位功能。程序中用到了：

- **算术指令**：addu, subu
- **逻辑指令**：and, or, nor, xor
- **比较指令**：slt, sltu
- **移位指令**：sll, srl

（2）I 型指令（立即数相关指令）

这类指令是测试 CPU 数据通路灵活性、立即数扩展单元和存储器接口的关键。程序中用到了：

- **立即数算术与比较指令**：addiu, slti, sltiu
- **Load/Store 指令**：lw, sw, lb, sb。这组指令是验证 CPU 与数据存储器（Data RAM）交互的核心，特别是对字节操作（sb, lb）的测试。
- **分支指令**：beq, bne。用于测试 CPU 的条件跳转控制逻辑。
- **常数加载指令**：lui。用于测试将 16 位立即数加载到寄存器高位的功能。

（3）J 型指令（无条件跳转指令）

这类指令用于测试 CPU 的无条件跳转控制逻辑，是验证程序流控制的重要部分。程序中用到了：

- **跳转指令**: j, jal

（4）数量统计

整个测试程序共由 38 条指令构成，各类指令的详细分布如下：

- **R 型指令：共 20 条**：这是测试程序中数量最多的指令类型，充分检验了 ALU 在处理寄存器数据时的各种能力。
- **I 型指令：共 14 条**：该类型指令覆盖了立即数运算、内存访问和条件分支，全面考核了数据通路和控制逻辑。
- **J 型指令：共 3 条**：用于测试最核心的无条件跳转功能。

2.本测试所用汇编程序的优点

（1） **ALU 功能全面验证**：程序中的 R 型指令和 I 型算术指令组合，完整地测试了 ALU 的加、减、与、或、非、异或、小于则置位等全部核心运算功能，确保了计算核心的正确性。

（2） **数据通路与立即数处理的深度考核**：addiu, slti 等指令验证了数据通路中立即数符号扩展单元的正确性，而 lui 指令则验证了常数加载通路。这确保了 CPU 可以正确处理不同类型的立即数操作数。

（3） **内存访问功能深度测试**：本程序不仅测试了基本的字读写（lw, sw），更关键的是，通过 sb 和 lb 指令，深入测试了非对齐字节的内存访问。这严谨地验证了访存模块的字节写使能逻辑（dm_wen）和字节读出时的符号扩展逻辑，这些都是设计中极易出错的细节。

（4） **控制流逻辑严谨考核**：程序包含了条件分支（beq, bne）和无条件跳转（j, jal），覆盖了程序控制流的各种情况。特别是 jal 指令的加入，有效地测试了 CPU 在执行跳转时保存返回地址到$ra 寄存器（$31）的链接功能，这是很多复杂程序（如函数调用）所必需的。

3.小结

综上所述，这套测试程序的设计是成功且高效的。它通过丰富多样的指令组合，全面地考核了 CPU 所实现的 38 条指令，系统性地遍历了 CPU 内部几乎所有的关键数据通路和控制逻辑，有力地证明了我们所设计的多周期 CPU 在功能上的完整性、鲁棒性和正确性。

其中，占比较高的 R 型指令和 I 型指令对 CPU 的计算核心、内存接口和分支逻辑进行了反复、深入的测试，而 J 型指令则确保了最基本的程序流控制的正确性。

表 2 测试所用汇编程序详述

No.	PC	指令	2 进制	16 进制
1	00	addiu $1, $0, 1	001001 00000 00001 0000000000000001	24010001
2	04	addu $2, $1, $1	000000 00001 00001 00010 00000 100001	00211021
3	08	subu $3, $2, $1	000000 00010 00001 00011 00000 100011	00411823
4	0C	slt $4, $3, $2	000000 00011 00010 00100 00000 101010	0062202A
5	10	slti $5, $4, 2	001010 00100 00101 0000000000000010	28850002
6	14	sltu $6, $5, $4	000000 00101 00100 00110 00000 101011	00A4302B
7	18	addiu $7, $6, 5	001001 00110 00111 0000000000000101	24C70005
8	1C	subu $8, $7, $3	000000 00111 00011 01000 00000 100011	00E34023
9	20	sll $9, $8, 1	000000 00000 01000 01001 00001 000000	00084840
10	24	srl $10, $9, 1	000000 00000 01001 01010 00001 000010	00095042
11	28	beq $1, $3, 1	000100 00001 00011 0000000000000001	10230001

No.	PC	指令	2 进制	16 进制
12	2C	bne $1,$3,4	000101 00001 00011 0000000000000100	14230004
13	30	subu $12, $9, $7	000000 01001 00111 01100 00000 100011	01276023
14	34	sll $13, $5, 2	000000 00000 00101 01101 00010 000000	00056880
15	38	srl $14, $13, 1	000000 00000 01101 01110 00001 000010	000D7042
16	3C	srl $15, $14, 1	000000 00000 01110 01111 00001 000010	000E7842
17	40	addu $16, $15, $13	000000 01111 01101 10000 00000 100001	01ED8021
18	44	subu $17, $16, $14	000000 10000 01110 10001 00000 100011	020E8823
19	48	addiu $18, $17, 5	001001 10001 10010 0000000000000101	26320005
20	4C	sltiu $19, $18, 10	001011 10010 10011 0000000000001010	2E53000A
21	50	lui $20, 0x1234	001111 00000 10100 0001001000110100	3C141234
22	54	bne $3, $4, 2	000101 00011 00100 0000000000000010	14640002
23	58	sw $2,0($18)	101011 10010 00010 0000000000000000	AE420000

No.	PC	指令	2 进制	16 进制
24	5C	lw $21,0($18)	100011 10010 10101 0000000000000000	8E550000
25	60	sb $3,4($18)	101000 10010 00011 0000000000000100	A2430004
26	64	lb $23,4($18)	100000 10010 10111 0000000000000100	82570004
27	68	lb $25,4($18)	100000 10010 11001 0000000000000100	82590004
28	6C	j 0x1D	000010 00000000000000000000 11101	0800001D
29	70	jal 0x1C	000011 00000000000000000000 11100	0C00001C
30	74	addu $26, $25, $23	000000 11001 10111 11010 00000 100001	0337D021
31	78	subu $27, $26, $23	000000 11010 10111 11011 00000 100011	0357D823
32	7C	addu $28, $27, $26	000000 11011 11010 11100 00000 100001	033AE021
33	80	and $29,$1,$2	000000 00001 00010 11101 00000 100100	0022E824
34	84	nor $30,$3,$4	000000 00011 00100 11110 00000 100111	0064F027

No.	PC	指令	2 进制	16 进制
35	88	or $23,$5,$6	000000 00101 00110 10111 00000 100101	00A6B825
36	8c	xor $26,$7,$8	000000 00111 01000 11010 00000 100110	00E8D026
37	90	move $24,$27	000000 11011 00000 11000 00000 100010	0360C022
38	94	j 0x00	000010 00000000000000000000 00000	08000000

7.2 指令执行含义与寄存器赋值变化逐条解释

表 3 指令执行流程解释

No.	PC	指令	功能描述	寄存器变化
1	0x00	addiu $1, $0, 1	$1 ← 0 + 1	$1 = 1
2	0x04	addu $2, $1, $1	$2 ← $1 + $1	$2 = 2
3	0x08	subu $3, $2, $1	$3 ← $2 - $1	$3 = 1
4	0x0C	slt $4, $3, $2	$4 ← ($3 < $2) ? 1 : 0	$4 = 1
5	0x10	slti $5, $4, 2	$5 ← ($4 < 2) ? 1 : 0	$5 = 1
6	0x14	sltu $6, $5, $4	$6 ← ($5 < $4) unsigned ? 1 : 0	$6 = 0

No.	PC	指令	功能描述	寄存器变化
7	0x18	addiu $7, $6, 5	$7 ← $6 + 5	$7 = 5
8	0x1C	subu $8, $7, $3	$8 ← $7 - $3	$8 = 4
9	0x20	sll $9, $8, 1	$9 ← $8 << 1	$9 = 8
10	0x24	srl $10, $9, 1	$10 ← $9 >> 1	$10 = 4
11	0x28	beq $1, $3, 1	若 $1==$3 则 PC ← PC+4×1 → Taken（跳至 0x30）	—
12	0x2C	bne $1, $3, 4	Skipped（被上一条跳过）	—
13	0x30	subu $12, $9, $7	$12 ← $9 - $7	$12 = 8 - 5 = 3
14	0x34	sll $13, $5, 2	$13 ← $5 << 2	$13 = 1 << 2 = 4
15	0x38	srl $14, $13, 1	$14 ← $13 >> 1	$14 = 4 >> 1 = 2
16	0x3C	srl $15, $14, 1	$15 ← $14 >> 1	$15 = 2 >> 1 = 1
17	0x40	addu $16, $15, $13	$16 ← $15 + $13	$16 = 1 + 4 = 5
18	0x44	subu $17, $16, $14	$17 ← $16 - $14	$17 = 5 - 2 = 3
19	0x48	addiu $18,$17,5	$18 ← $17+5	$18 = 3 + 5 =8

No.	PC	指令	功能描述	寄存器变化
20	0x4C	sltiu $19, $18, 10	$19 ←($18 <10)unsigned?1:0	$19 = 1
21	0x50	lui $20,0x1234	$20 ← 0x1234 << 16	$20 = 0x1234_0000
22	0x54	bne $3, $4, 2	若 $3≠$4 则 Not Taken → 顺序执行	—
23	0x58	sw $2,0($18)	MEM[$18+0] ← $2	—
24	0x5C	lw $21,0($18)	$21 ← MEM[$18+0]	$21 = 2
25	0x60	sb $3,4($18)	MEM[$18+4] (byte) ← low-8($3)	—
26	0x64	lb $23,4($18)	$23 ← sign-extend MEM[$18+4]	$23 = sign-extend(1) = 1
27	0x68	lb $25,4($18)	$25 ← sign-extend MEM[$18+4] ← 同样是 0x01 → 0x00000001	$25 = 1
28	0x6C	j 0x1D	PC ← 0x1D << 2 = 0x74	—
29	0x70	jal 0x1C	Skipped（被上一条跳转过）	—

No.	PC	指令	功能描述	寄存器变化
30	0x74	addu $26,$25,$23	$26 ← $25 + $23	$26 = 1 + 1 = 2
31	0x78	subu $27,$26,$23	$27 ← $26 - $23	$27 = 2 - 1 = 1
32	0x7C	addu $28,$27,$26	$28 ← $27 + $26	$28 = 1 + 2 = 3
33	0x80	and $29,$1,$2	$29 ← $1 & $2	$29 = 1 & 2 = 0
34	0x84	nor $30,$3,$4	$30 ← ~($3 | $4) (1|1=1 → ~1 = 0xFFFFFFFF)	$30 = 0xFFFFFFFE
35	0x88	or $23,$5,$6	$23 ← $5 | $6	$23 = 1 | 0 = 1
36	0x8C	xor $26,$7,$8	$26 ← $7 ⊕ $8	$26 = 5 ⊕ 4 = 1
37	0x90	move $24,$27	$24 ←$27	$24 = $27 =1
38	0x94	j 0x00	PC ← 0x00 << 2 = 0x00 → 程序结束 / 返回	—

该指令集测试表格全面展现了 MIPS 指令功能的高度多样性。

它不仅覆盖了 addu/subu 等基础算术、and/or/sll 等位逻辑运算，还包含了 slt 系列的有符号/无符号比较，构成了完整的计算核心测试。更关键的是，通过 lw/sw 和 lb/sb 指令，程序深入验证了 CPU 处理字与字节两种不同粒度的内存交互能力。最后，beq/bne 的条件分支与 j/jal 的无条件跳转，系统性地考核了 CPU 最核心的程序流控制机制。整个测试流程设计严谨，充分验证了 CPU 的综合处理能力。

7.3 仿真波形图验证

为了能够较为清晰的进行功能仿真并对指令的功能进行验证，这里我使用了一个 testbench，用于对寄存器或者内存地址中的数据进行监测。

由于每条指令基本都是对不同的寄存器进行操作，一个指令执行完成后只有一个寄存器的数值发生了变化，因而我采用监测写回总线的 PC 值的变化，来判断对应的第几条指令已经执行完毕，从而将 **rf_addr 的寄存器编号值或是 mem_addr 的访存地址内容**改为对应的修改后的相应的寄存器值以及访存地址的内容变化。经过这样的操作，可以做到每一条指令完成之后看到对应寄存器或内存地址的数据变化，从而有效地验证了程序和指令功能。

下图 3-图 7 是本次课程设计使用的 38 条指令的仿真波形图，每幅图运行 600ns。由图 3-图 7，我们可以看出指令是按照输入顺序依次从 inst_rom 中读取了出来，并进入 CPU 进行处理的时候也是严格按照五段功能段进行流入与流出。同时，也可以看到每个功能段的 PC 值也会正常地自增，且从 coe 文件读取的指令十六进制机器码也正确的读出，且执行时的数据流动无误。

经过和自己手写模拟的寄存器变化的值进行比对可以看出寄存器值的变化均符合预期理想内容，且都能正常运行，不存在读不出值，rf_data 为 XXXXXXXX 的情况，因而可以证明，我的课程设计的指令内容的执行达到了预期要求。

接下来，我会对我的 MIPS 测试程序的特殊指令（例如条件转移、访存指令、J 型跳转）进行对应位置的讲解，较为简单的加减移位运算以及按位逻辑运算在此忽略。

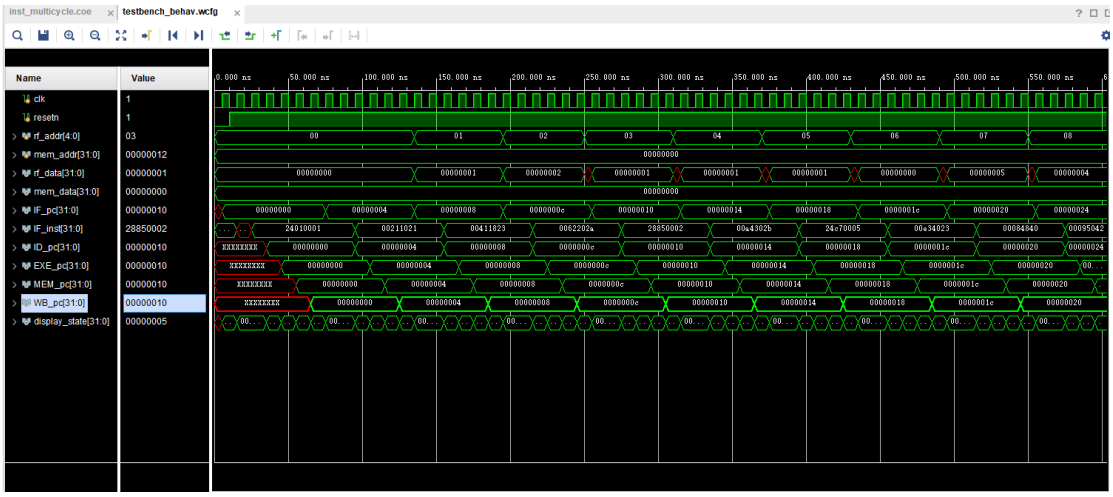


图 3 0-600ns 时的指令执行情况(WB_pc 在 00H 到 20H 之间)

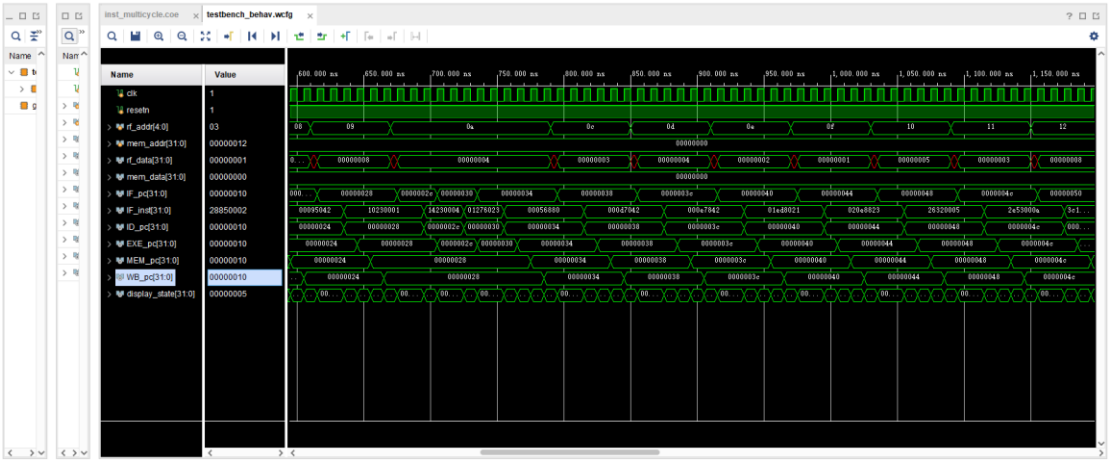


图 4 600-1200ns 时的指令执行情况(WB_pc 在 24H 到 4CH 之间)

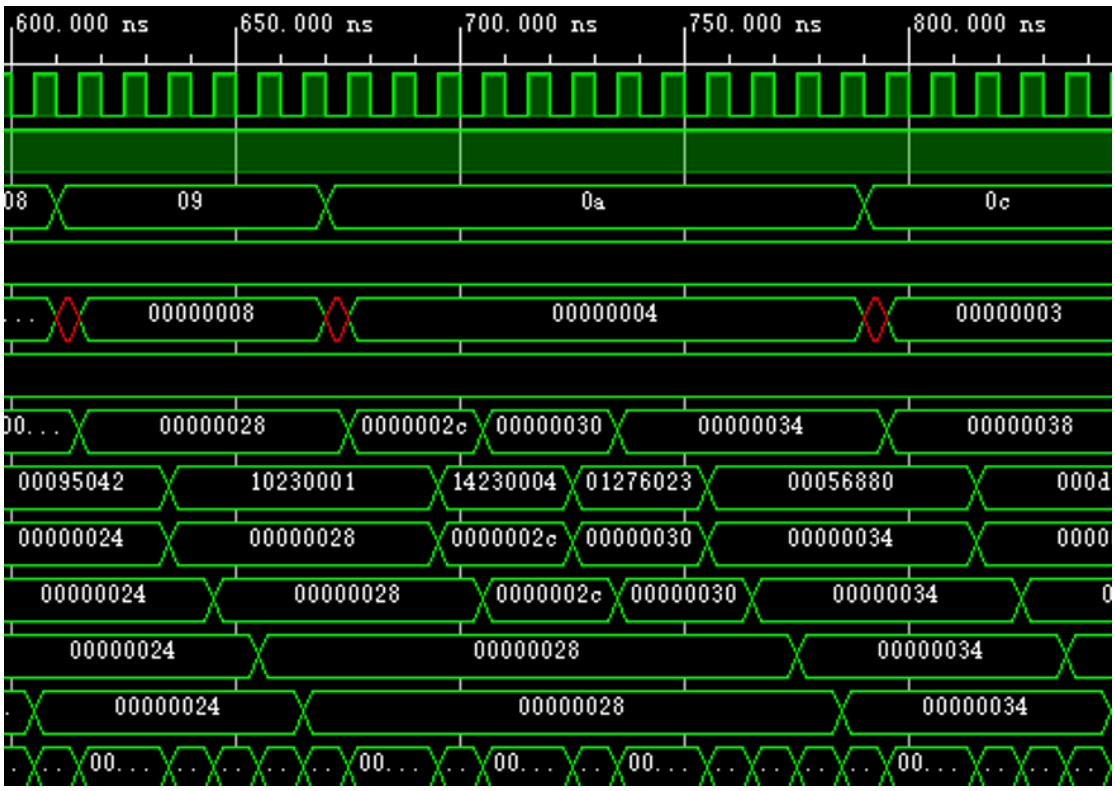


图 5 PC 值为 28H 时对应的 beq 跳转的实际仿真波形图

在此，我对 PC 值为 28H 时对应的 beq $1, $3, 1 指令进行说明。经过第一条指令和第三条指令的执行，可以计算得出$1 和$3 寄存器的值都是 1，beq 的分支跳转条件成立。

而根据 beq 指令的指令跳转规则我们可以得知，当对应的 offset 偏移量为 1 时，即偏移 4H 的指令时，译码器会先让 pc 自增+4，再对输入指令的 offset 这个值左移两位，即 pc 再加一个 4。其原因主要是本次硬件课程设计的 CPU 是 32 位，而 CPU 中的 pc 地址和内存地址都是按字节编码，因此地址占 4 字节空

间。而 offset 是 4 的倍数的设计是为了能够节省两位二进制比特，从而获得更大的寻址空间。

根据图 5 的指令实际执行过程可以看出，这条指令流入 CPU 后，IF_pc 先是自增+4，而在这条指令执行到 EXE 阶段时，即指令执行阶段的时候，beq 条件成立的时候会驱动 IF_pc 加上偏移量 4，从而使得 pc 值跳到 30H 的位置，从而执行 subu $12,$9,$7 的指令，跳过了理应在 0x2C 时输入的 bne $1,$3,4 的指令，避免了进一步的跳转，而在后续我们可以看到 12 号寄存器成功计算出了理想值，故而可以得证分支指令的正确执行。

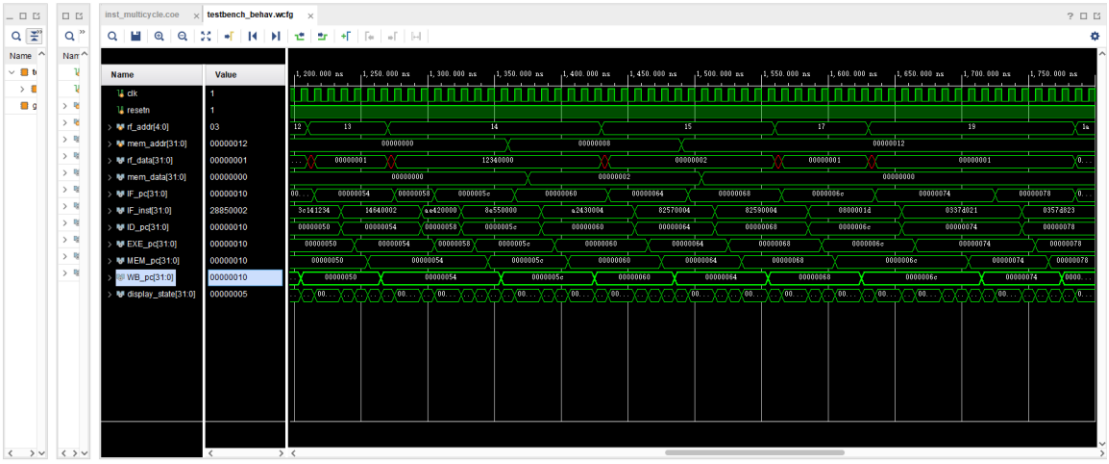


图 6 1200-1800ns 时的指令执行情况(WB_pc 在 50H 到 74H 之间)

我将对 PC 值自 58H 开始的一系列访存与跳转指令的仿真波形进行连续说明。在执行此段程序前，通过前序指令的运算，我已经得知寄存器$18 的值为 8，寄存器$2 的值为 2，寄存器$3 的值为 1。接下来对访存指令原理进行叙述。

首先是位于 PC 值 58H 的 sw $2, 0($18)指令，这是一条字存储指令。当该指令执行到 MEM 访存阶段时，我们可以从仿真波形图中清晰地观察到，数据通路正确地将基址$18 与偏移量 0 相加，计算出的有效地址 8 被送至 mem_addr 端口。与此同时，待写入的数据$2（值为 2）出现在 dm_wdata 总线上，且内存写使能信号 dm_wen 高亮有效（值为 4'b1111），表示一个完整的 32 位字被写入。紧随其后的 lw $21, 0($18)指令在执行到 MEM 阶段时，mem_addr 再次显示为地址 8，由于是同步读操作，在下一周期，我们可以看到 mem_data 端口成功返回了我们刚刚存入的数据 2。最终，在 WB 写回阶段，寄存器$21 的值被正确更新为 2，这有力地证明了我们所设计的 CPU 字级读写通路是准确无误的。

紧接着，程序通过 sb 与 lb 指令，对 CPU 处理非对齐字节的能力进行了更为严苛的测试。在 PC 为 60H 的 sb $3, 4($18)指令执行时，访存地址计算为 $18+4=12（即 0CH）。波形图中最关键的一幕是，由于地址 0CH 的低两位为 2'b00，我们设计的访存控制逻辑精确地将 dm_wen 设置为 4'b0001，仅对 32 位

内存单元的最低字节进行写入，将$3 的低 8 位（值为 1）存入。随后的两条 lb 指令再次访问地址 12，波形显示 mem_data 读出了包含该字节的整个字，但我们设计的访存模块硬件逻辑成功地从中提取出字节 1，并进行了正确的符号位扩展（正数扩展为 0x01）。最终，$23 和$25 寄存器均被正确写入值 1，这充分验证了 CPU 字节寻址、部分写使能以及加载时符号扩展等高级功能的正确性。

最后，在 PC 为 6CH 的位置，j 0x1D 指令展示了 CPU 最核心的无条件控制流转移功能。根据 MIPS 指令规定，该指令的目标地址由立即数 0x1D 左移两位得到，即 0x74。在仿真波形中可以清晰地看到，当该指令在 ID 级被译码后，跳转控制总线 jbr_bus 立即向 IF 级发出跳转请求。因此，下一个周期的 IF_pc 值没有按顺序更新为 0x70，而是被强制覆写为目标地址 0x74。这一精确的跳转直接导致了位于地址 0x70 的 jal 指令被完全跳过，从未进入取指阶段，从而完美验证了我们设计的处理器无条件跳转逻辑的正确性与即时性。

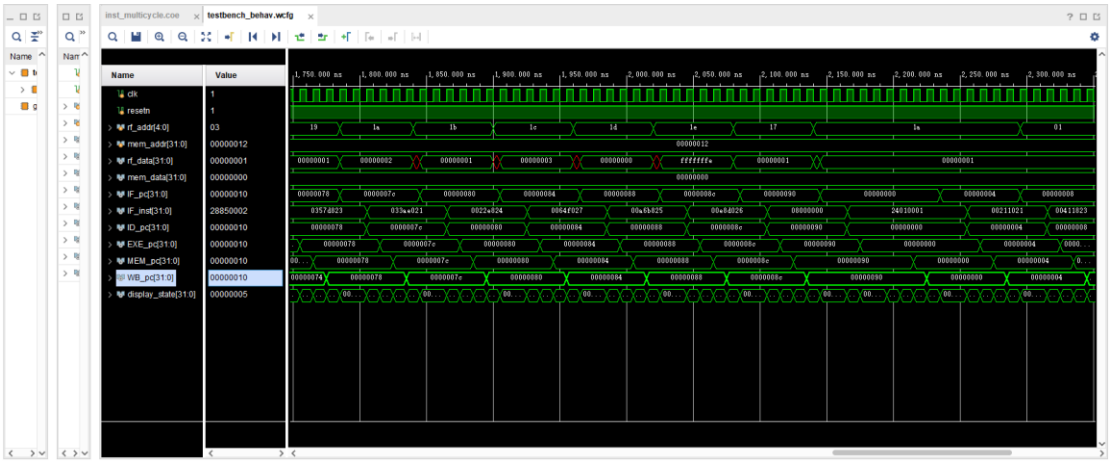


图 7 1750-2350ns 时的指令执行情况(WB_pc 在 74H 到 04H 之间)

在指令的结尾，我设置了 j 00H 以使得程序可以循环运行，能正确执行。

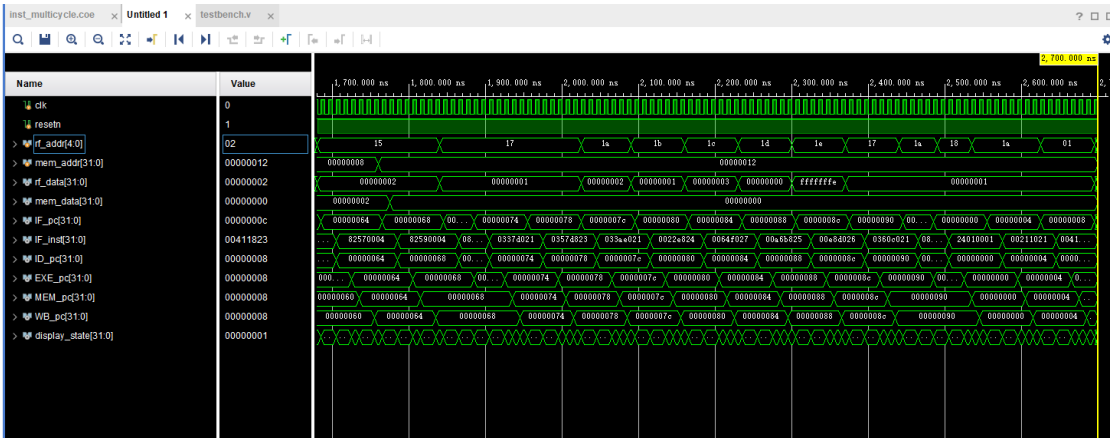
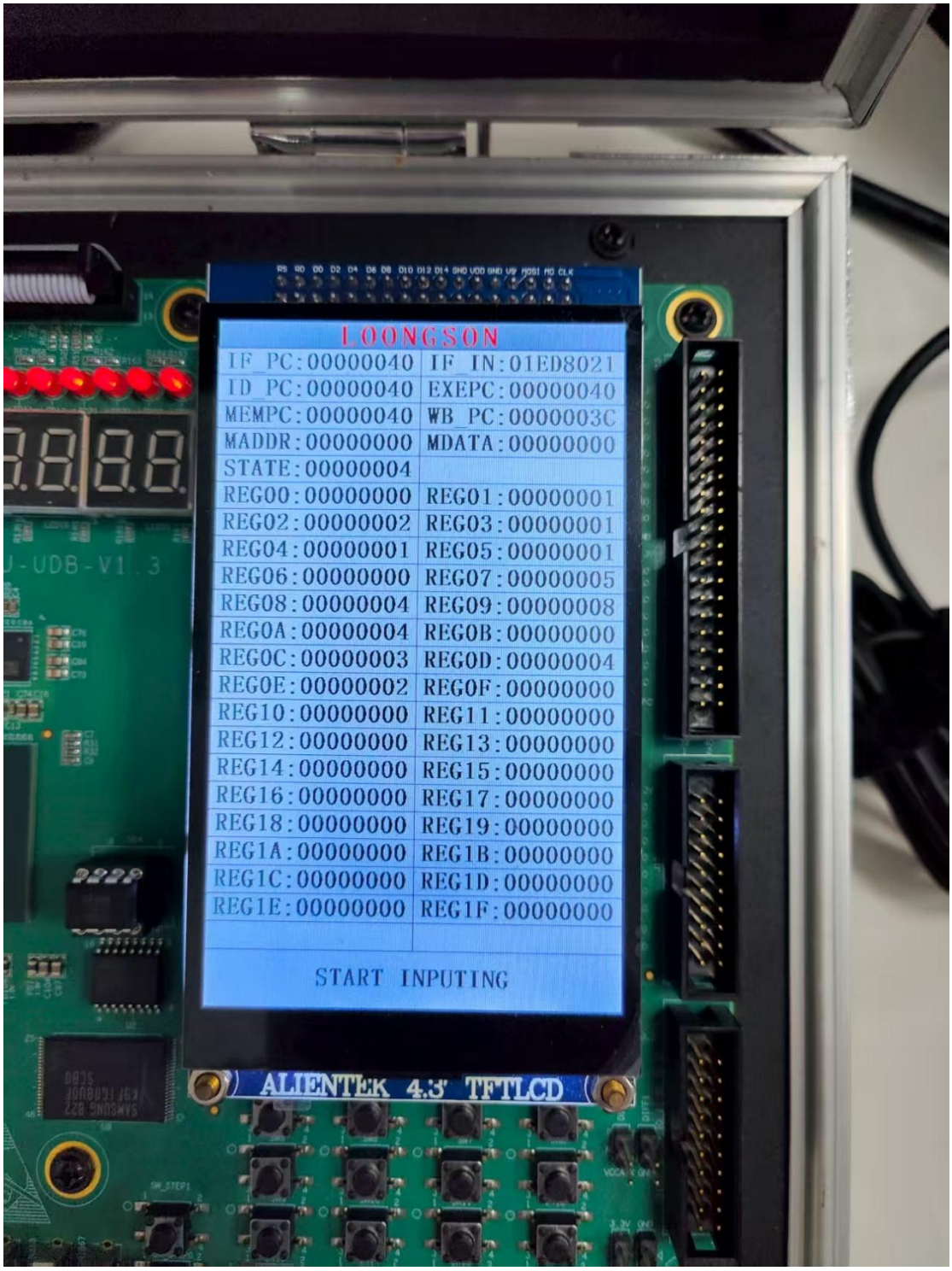
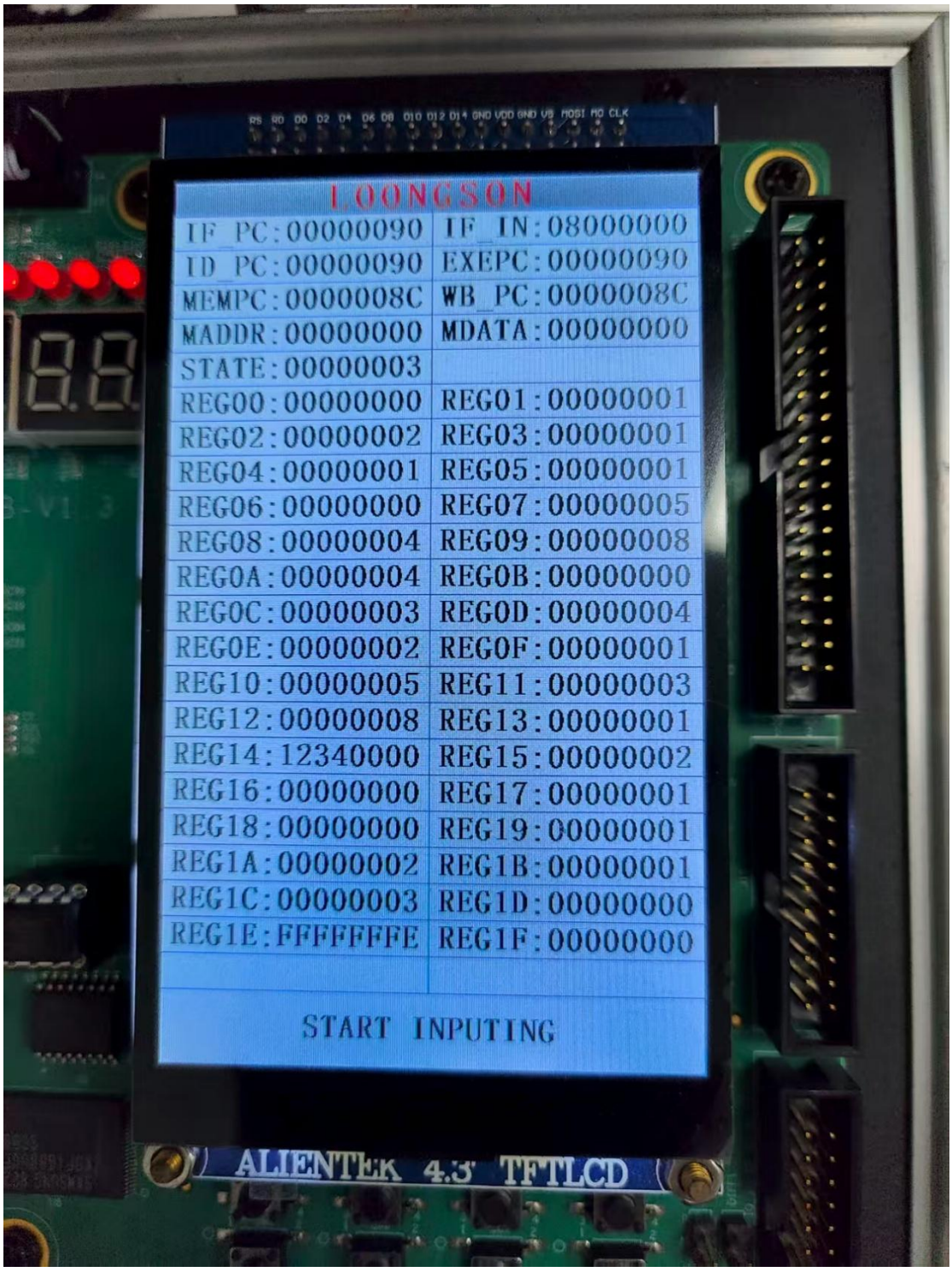


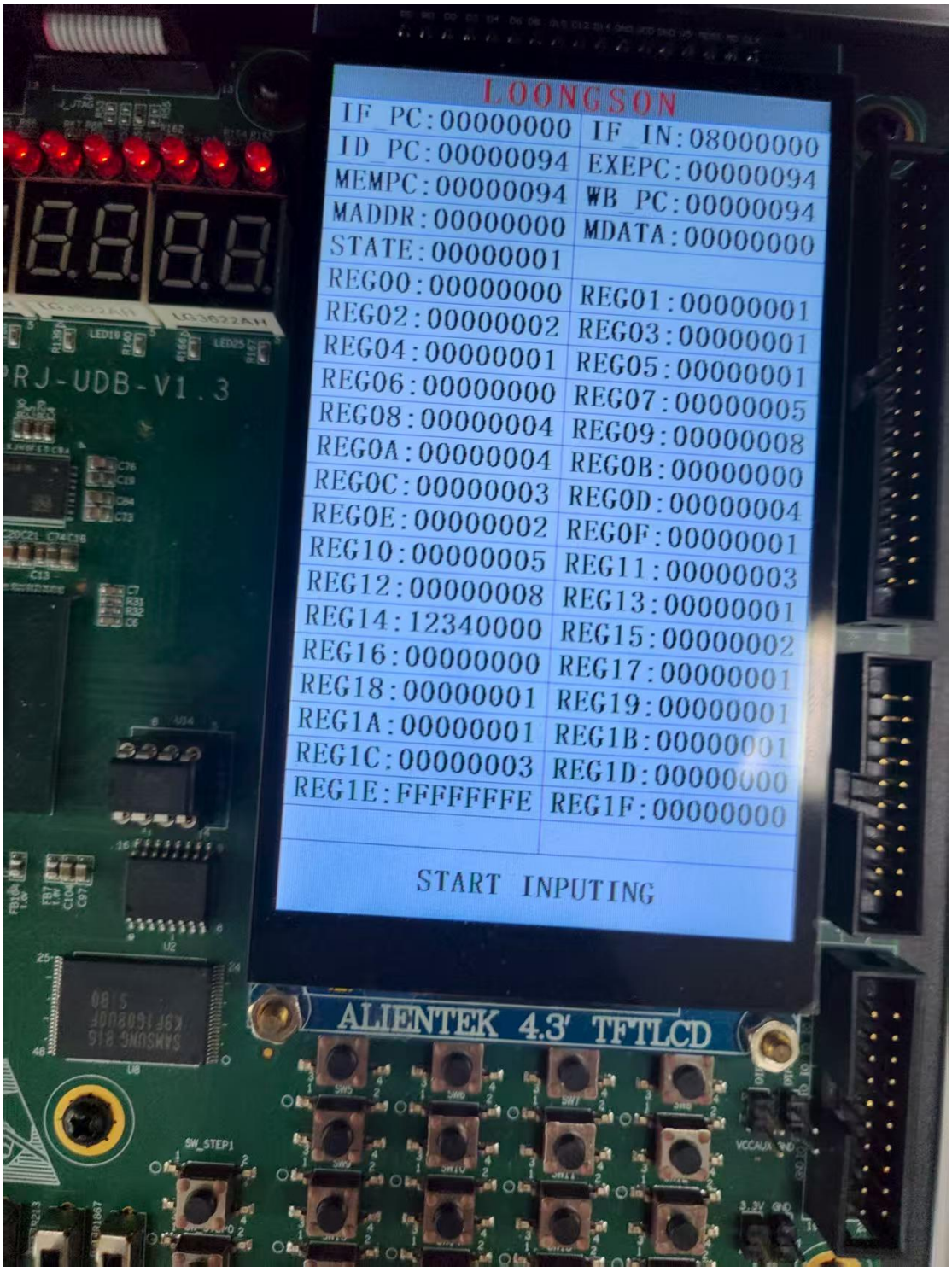
图 8 1950-2500ns 时的指令执行情况(WB_pc 在 80H 到 10H 之间)

7.4 上板验证图

由于我设计的指令验证程序对于大部分寄存器都是只操作一次，修改一次值，因而这里只截取几个阶段的上板实际验证图来进行展示，不再赘述。




LOONGSON
IF_PC:00000090
IF_IN:08000000
ID_PC:00000090
EXEPC:00000090
MEMPC:0000008C
WB_PC:0000008C
MADDR:00000000
MDATA:00000000
STATE:00000003
REG00:00000000
REG01:00000001
REG02:00000002
REG03:00000001
REG04:00000001
REG05:00000001
REG06:00000000
REG07:00000005
REG08:00000004
REG09:00000008
REG0A:00000004
REG0B:00000000
REG0C:00000003
REG0D:00000004
REG0E:00000002
REG0F:00000001
REG10:00000005
REG11:00000003
REG12:00000008
REG13:00000001
REG14:12340000
REG15:00000002
REG16:00000000
REG17:00000001
REG18:00000000
REG19:00000001
REG1A:00000002
REG1B:00000001
REG1C:00000003
REG1D:00000000
REG1E:FFFFFFFE
REG1F:00000000
START INPUTING
ALIENTEK 4.3' TFTLCD


LOONGSON
IF_PC:00000000
IF_IN:08000000
ID_PC:00000094
EXEPC:00000094
MEMPC:00000094
WB_PC:00000094
MADDR:00000000
MDATA:00000000
STATE:00000001
REG00:00000000
REG01:00000001
REG02:00000002
REG03:00000001
REG04:00000001
REG05:00000001
REG06:00000000
REG07:00000005
REG08:00000004
REG09:00000008
REG0A:00000004
REG0B:00000000
REG0C:00000003
REG0D:00000004
REG0E:00000002
REG0F:00000001
REG10:00000005
REG11:00000003
REG12:00000008
REG13:00000001
REG14:12340000
REG15:00000002
REG16:00000000
REG17:00000001
REG18:00000001
REG19:00000001
REG1A:00000001
REG1B:00000001
REG1C:00000003
REG1D:00000000
REG1E:FFFFFFFE
REG1F:00000000
START INPUTING
ALIENTEK 4.3' TFTLCD
RJ-UDB-V1.3
SW_STEP1

八、心得体会

本次硬件课程设计是一场理论与实践的深度融合。从最初面对复杂需求的茫然，到最终成功构建出能执行指令集的 CPU，整个过程不仅是对 Verilog 编程能力的考验，更是对计算机体系结构理解的一次升华。

经过这次课程设计，我深刻体会到“设计先行”的重要性。相较于直接编码，预先规划顶层模块的数据通路、明确定义各级总线（如 ID_EXE_bus）的内容，能有效避免后期集成的混乱，让编码过程事半功倍。这让我明白，优秀的硬件设计，其精髓在于对数据流与控制流的精确规划。

在攻坚阶段，跳转指令的控制尤为棘手。例如，最初 jal 指令虽能跳转，但其返回地址$ra 的保存值是错误的 PC。通过深入调试，我意识到必须在译码级为 ALU 规划一条专用通路来计算 PC+4，并将其结果与目标寄存器$ra 绑定，这让我懂得了 CPU 设计需对指令的每个副作用进行精密解耦。

同样，同步 RAM 的访存时序也是一大难点，我发现 lw 指令总读到无效值，其根源在于同步 RAM 需要额外一拍才能将数据稳定在 dm_rdata 总线上。为此，我通过寄存器 MEM_valid_r 来延迟完成信号，设计了 MEM_over 逻辑，在执行 load 指令时强制使 MEM 阶段多停留一个周期，以等待数据就绪。

这个过程让我对硬件时序与多部件协同工作的延迟问题有了切身体会。总而言之，本次课设将计算机逻辑基础、计算机组成原理以及计算机系统结构等课程的抽象知识具象化，让我从硬件层面理解了 CPU 的工作脉络。而在反复调试与修正中获得的成就感，让我真切感受到了计算机底层设计的魅力，也驱动我对探索 Verilog 硬件设计有了更浓厚的兴趣。