

组成原理课程第六次实验报告

实验名称：多周期 CPU 实验

学号：2310422 姓名：谢小珂 班次：1078

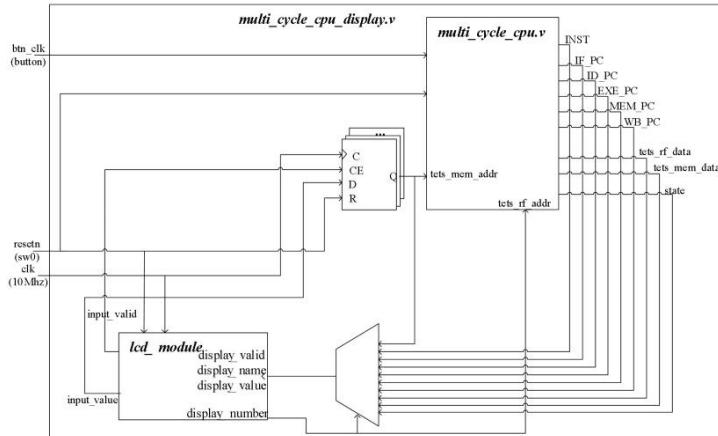
一、实验目的

- 在单周期 CPU 实验完成的前提下，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。

二、实验内容说明

- 多周期 CPU 实验使用同步 IP 核构造 data_ram 和 inst_rom，原始 source_code 中的同名.v 文件和 ngc 文件不要导入到项目。
- 多周期 CPU 运行的指令在 inst_rom 中，这里面的指令须导入 coe 文件。
- 请把 ALU 实验中添加的三个运算，自行定义类似 MIPS 指令格式的指令，把对应的指令和功能增加到多周期 CPU 中，并自行在 coe 文件中添加指令，然后进行运行验证（仿真波形验证或实验箱验证即可）。
- 实验报告中可以不放原理图，关于验证结果的图片（仿真图片或实验箱图片）需要详细介绍图中的信息和对指令验证的情况。

三、实验原理图



多周期 cpu 的顶层模块框图

四、实验步骤

(一) 搭建指令 ROM 并配置 IP 核

1. 创建同步 IP 核 ROM：

使用同步 IP 核构造 data_ram 和 inst_rom，避免导入原始 source_code 中的.v 和.ngc 文件。

Component Name: inst_rom

Basic | Port A Options | Other Options | **S**

Information

Memory Type: Single Port ROM
 Block RAM resource(s) (18K BRAMs): 1
 Block RAM resource(s) (36K BRAMs): 0
 Total Port A Read Latency : 1 Clock Cycle(s)
 Address Width A: 8

inst_rom

关键配置：创建 IP 核时不勾选 Primitives Output Register，防止输出延迟增加 1 个时钟周期（避免总延迟达到两周期）。

2. 添加指令文件：

在 inst_rom 中导入. coe 文件，指令与单周期 CPU 实验相同，新增以下三条指令：

```
0065682F // SGT $13, $3, $5
0065702C // SGTU $14, $3, $5
00A0782D // NOT $15, $5
```

(二) 实现新增指令功能

1. 修改 ALU 模块 (alu.v)

- 扩展 alu_control 信号至 15 位：

```
input [14:0] alu_control, // 扩展为 15 位，低 3 位用于新增指令
```

- 新增运算逻辑实现

<1>NOT 指令（按位取反）

```
wire [31:0] not_result;
assign not_result = ~alu_src1; // 对 alu_src1 按位取反
```

<2>SGTU 指令（无符号大于置位）

```
wire [31:0] sgtu_result;
assign sgtu_result = {31'd0, adder_cout & (~adder_result)};
```

<3>SGT 指令（有符号大于置位）

```
wire [31:0] sgt_result;
assign sgt_result = {31'd0, adder_cout & (~adder_result)};
```

- 更新 ALU 结果选择逻辑，优先匹配新增指令

```
assign alu_result = alu_sgt ? sgt_result :
alu_sgtu ? sgtu_result :
alu_not ? not_result :
```

2. 修改译码模块 (decode.v)

- 新增指令定义

```
assign inst_SGTU = op_zero & sa_zero & (funct == 6'b101100);
assign inst_SGT = op_zero & sa_zero & (funct == 6'b101111);
assign inst_NOT = op_zero & sa_zero & (funct == 6'b101101);
```

- ALU 操作分类映射

```
assign inst_sgtu = inst_SGTU; // 无符号大于置位
assign inst_sgt = inst_SGT; // 有符号大于置位
assign inst_not = inst_NOT; // 按位取反
```

- 扩展 alu_control 信号

```
assign alu_control = { ..., inst_sgtu, inst_sgt, inst_not };
```

扩展 alu_control 信号，将新增指令分类并映射到对应控制位。

- 更新寄存器写入目标逻辑

```
assign inst_wdest_rd = ... | inst_SGTU | inst_SGT | inst_NOT;
```

更新寄存器写入目标逻辑 (inst_wdest_rd)，包含新增指令。

3. 调整总线位宽

修改 multi_cycle_cpu.v 及相关文件，将 ID_EXE_bus 扩展至 153 位以适应 alu_control 信号。

```
wire [152:0] ID_EXE_bus;
reg [152:0] ID_EXE_bus_r;
wire [14:0] alu_control;
```

(三) 仿真验证

- 修改测试文件 (tb.v) :

添加仿真代码，显示指令执行过程、寄存器及内存内容：

```
initial begin
    // 打印仿真开始标志
    $display("== CPU Simulation Test ==");
    // 打印表头：时间、PC 值、当前指令
    $display("Time\tPC\t\tInstruction\tState");

    for (i = 0; i < 144; i = i + 1) begin
        // display_state - CPU 状态 (1=IF, 2=ID, 3=EX, 4=MEM, 5=WB)
        $display("%0t\t%h\t%h\t%d",
            $time, IF_pc, IF_inst, display_state);
    end

    $display("\n== Register File Contents ==");
    for (r = 1; r < 16; r = r + 1) begin
        rf_addr = r;          // 设置要读取的寄存器地址
        #10;                  // 等待寄存器读取稳定
        $display("R%0d\t= %h", r, rf_data);
    End

    $display("\n== Memory Contents ==");
    for (m = 0; m < 32; m = m + 4) begin
        mem_addr = m;        // 设置内存地址
```

```

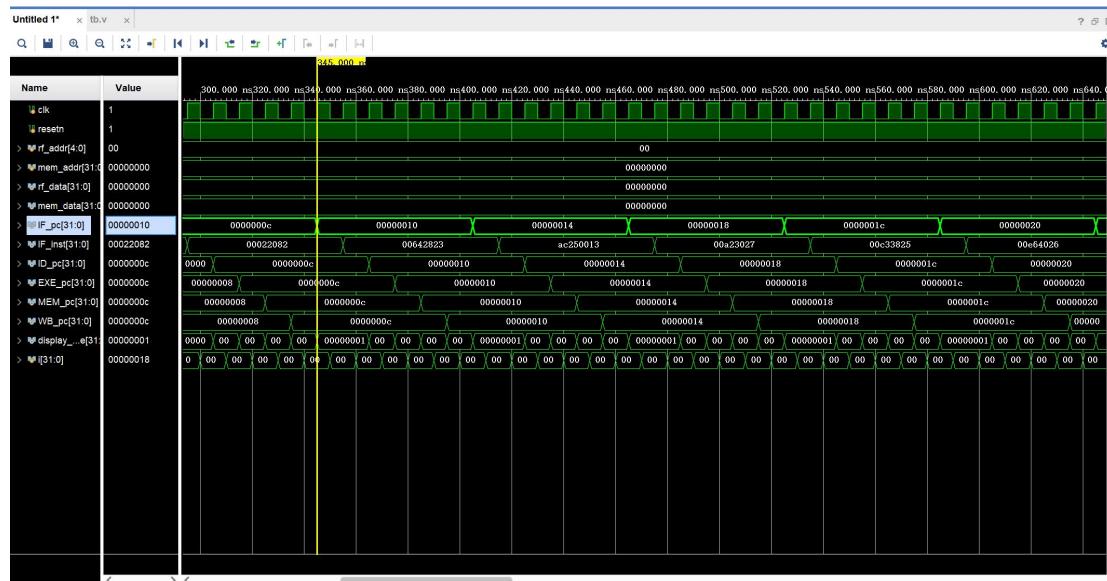
#10; // 等待内存读取稳定

$display("Mem[%h] = %h", m, mem_data);
end
$display("\n==== Simulation Complete ===");
$finish;
end

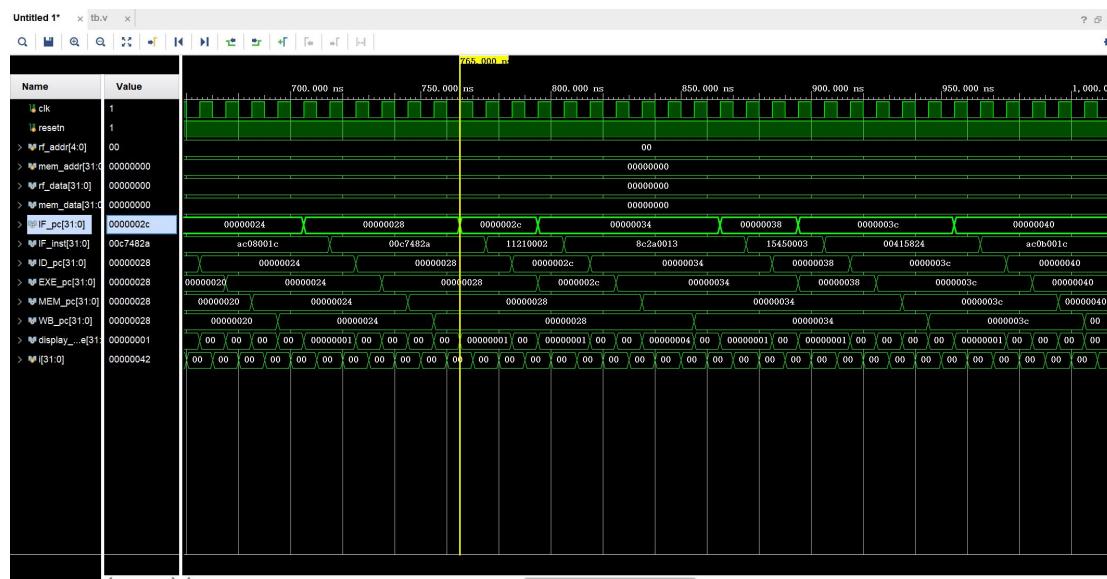
```

五、实验结果分析

(一) 仿真测试结果



图一



图二

通过波形分析可确认，所有指令均严格遵循五级流水线顺序执行：

IF（取指）→ ID（译码）→ EX（执行）→ MEM（访存）→ WB（写回）

每个时钟周期推进一个流水线阶段，各阶段寄存器正确锁存传递数据。

指令类型周期差异

指令类型	总周期数	关键阶段说明
R型/I型	6	完整经历5个阶段+1个结果传递周期（如：add, slt, ori）
条件跳转	3	在ID阶段完成： <ul style="list-style-type: none">条件判断（如beq \$9,\$1,#2）跳转地址计算
J型跳转	3	ID阶段直接计算目标地址（如j 00H）
Load指令	7	额外增加2个周期： <ul style="list-style-type: none">MEM阶段发地址下一周期才能读取同步RAM数据

图一仿真分析：

1. 当前指令执行状态

- IF阶段：PC值：00000010（16）

当前指令：00022082（对应指令：srl \$4,\$2,#2）

说明：CPU正在从地址0x10处取出右移指令

- ID阶段：PC值：0000000C（12）

说明：正在解码地址0x0C处的指令（上一条指令）

- EXE/MEM/WB阶段：PC值均为0000000C

2. 关键信号分析

时钟 clk=1（上升沿）

复位 resetn=1（未复位）

显示状态 display_state=1（IF阶段）

图二仿真分析：

1. 当前指令执行状态

- IF阶段：PC值：0000002C（44）

当前指令：00C7482A（对应指令：slt \$9,\$6,\$7）

指令功能：比较\$6和\$7，若\$6<\$7则\$9=1

- ID阶段：PC值：00000028（40）

对应指令：11210002（beq \$9,\$1,#2）

- EXE/MEM/WB阶段：PC值均为00000028

2. 关键信号分析

时钟 clk=1（上升沿）

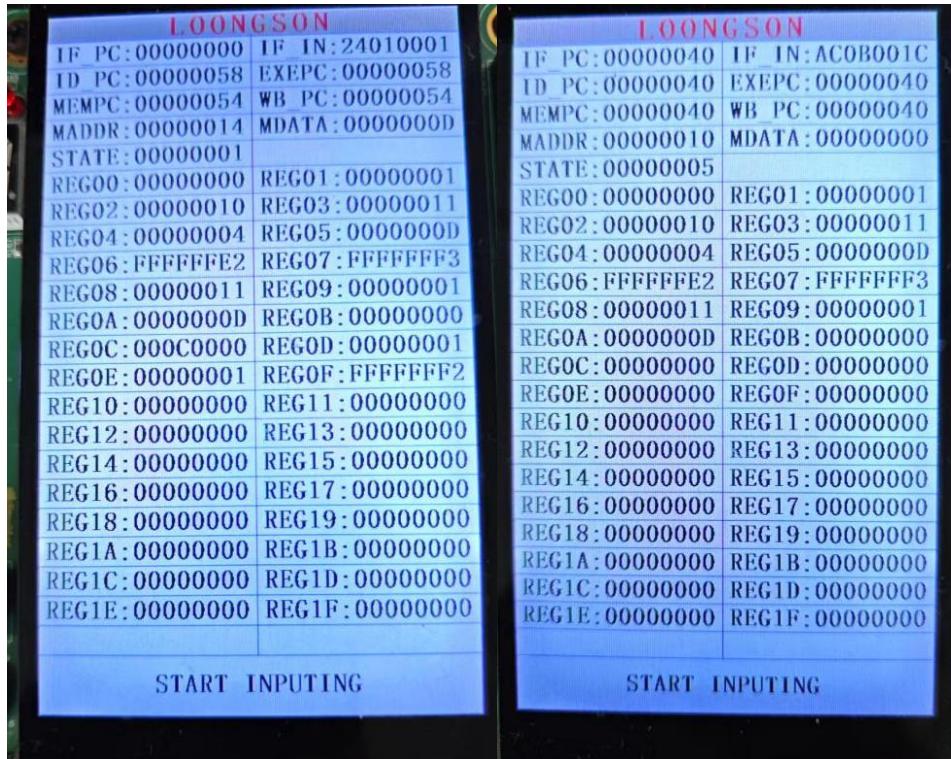
复位 resetn=1（未复位）

显示状态 display_state=1（IF阶段）

多周期特性正确表现

现象	符合预期的设计特性
各阶段 PC 独立推进	IF/ID/EXE/MEM/WB 可同时处理不同指令
分支指令快速判断	beq 在 ID 阶段完成比较（图 2 中 ID_pc=28H 时 IF 已取 2CH 指令）
同步存储器接口	mem_addr 和 mem_data 信号严格对齐时钟边沿

（二）实验箱验证结果



图左分析

1. 指令执行状态

当前指令：执行指令为 24010001 (ADDIU \$1, \$0, #1)。

2. 流水线状态：

ID 阶段：PC=00000058 (上一条指令地址)

EXE/MEM/WB 阶段：仍在处理更早的指令 (PC=00000054)。

图右分析

1. 指令执行状态

当前指令：执行指令为 AC0B001C (SW \$11, #28(\$0))。

2. 流水线状态：

IF/ID/MEM 阶段：PC 均为 00000040，表明是单指令执行完成后的状态（无重叠）。

经验证，结果均符合预期。

六、总结感想

(一) 知识理解模块：从单周期到多周期的深入认知

1. 多周期设计的核心价值

通过实验深刻理解多周期 CPU 与单周期 CPU 的本质差异：多周期设计将指令执行拆解为 IF（取指）→ ID（译码）→ EX（执行）→ MEM（访存）→ WB（写回）五级流水线，每个阶段在独立时钟周期内完成特定功能。这种“分阶段协作”模式使 CPU 可同时处理不同指令的不同阶段（如 IF 阶段取当前指令时，ID 阶段正在译码前一条指令），显著提升了指令执行效率。

2. 指令执行的时序逻辑

观察到不同类型指令的周期差异：

R 型 / I 型指令需 6 个周期（完整经历 5 阶段 + 1 个结果传递周期），例如 ADD 指令需依次完成取指、译码、算术运算、访存（若需）、写回；

条件跳转指令（如 BEQ）仅需 3 个周期，在 ID 阶段完成条件判断和跳转地址计算，提前终止流水线后续阶段，体现了分支指令的优化设计；

Load 指令因同步 RAM 访问特性需 7 个周期（MEM 阶段发地址后需下一周期读取数据），深刻理解了存储器接口时序对 CPU 性能的影响。

3. 模块化协作的系统性思维

从顶层模块框图（如 multi_cycle_cpu.v）到各子模块（ALU、译码器、寄存器堆等），认识到 CPU 是一个高度协同的系统。例如，ALU 模块的运算结果需通过总线（如 ID_EXE_bus）准确传递至后续阶段，译码模块需根据指令类型生成正确的控制信号（如 alu_control），各模块间的信号传递和时序配合直接决定系统功能的正确性。

(二) 实践能力：Verilog 设计与调试的全流程锻炼

1. Verilog 模块化编程能力

- ALU 模块扩展：将 alu_control 信号从单周期的低位宽扩展至 15 位，新增 NOT（按位取反）、SGT（有符号大于置位）、SGTU（无符号大于置位）运算逻辑。通过 assign 语句实现组合逻辑（如 not_result = ~alu_src1），并在结果选择逻辑中优先匹配新增指令（alu_result = alu_sgt ? sgt_result : ...），熟练掌握了 Verilog 中条件语句和逻辑表达式的应用。

- 译码模块指令映射：通过 op_zero 和 funct 字段定义新增指令（如 inst_NOT = op_zero & sa_zero & (funct == 6'b101101)），并将指令类型映射至 alu_control 控制位（如 assign inst_not = inst_NOT），深化了对 MIPS 指令格式和译码逻辑的理解。

2. 仿真与调试技能提升

- 波形分析能力：通过仿真工具观察 IF_pc、ID_pc 等信号，验证指令按流水线顺序执行（如图 1 中 IF 阶段处理 0x10 地址指令时，ID 阶段处理前一条 0x0C 指令）；通过 display_state 信号（1-5 分别对应 IF 至 WB 阶段）确认各阶段状态机正确跳转。

- 问题排查经验：在调试中遇到因 IP 核输出寄存器勾选导致的时序延迟问题，通过对比文档关键配置（不勾选 Primitives Output Register）解决；通过仿真日志打印（如 \$display 输出 PC 值、指令内容）快速定位指令执行异常点。

3. 实验箱验证的工程思维

在硬件验证中，通过观察 LCD 显示的寄存器值（如 REG01=0x00000001）和内存地址内容（如 MADDR=0x00000014、MDATA=0x0000000D），验证 ADDIU、SW 等指令的正确性。这一

过程强化了“软件设计→硬件实现→实测验证”的完整工程链路意识，认识到理论设计需与硬件特性（如同步 RAM 的访问延迟）相结合。

（三）挑战与方法论：细节把控与系统调试

1. 关键细节的决定性作用

- IP 核配置陷阱：若未注意 inst_rom 配置中不勾选输出寄存器，会导致指令输出延迟 1 个时钟周期，使流水线各阶段 PC 值错位（如 IF 与 ID 阶段 PC 值不连续），最终引发指令执行错乱。这一问题提醒我在硬件设计中需严格遵循文档要求，关注每一个配置参数对时序的影响。

- 总线位宽扩展：将 ID_EXE_bus 从单周期的低位宽扩展至 153 位以容纳 alu_control[14:0]，若遗漏此步骤会导致控制信号截断，新增指令无法正确执行。这体现了系统设计中“牵一发而动全身”的特性，需全面评估模块修改对全局的影响。

2. 分阶段调试方法论

采用“自上而下、分模块验证”策略：

先验证指令 ROM 加载（通过 coe 文件导入指令，确认 IF_inst 正确读取）；

再验证 ALU 运算功能（单独测试 not_result、sgt_result 等信号是否符合预期）；

最后进行全流水线联合仿真，观察各阶段 PC 值、控制信号和数据传递是否连贯。这种分层验证方法可有效缩小问题范围，提升调试效率。

（四）未来拓展：从多周期到流水线的进阶准备

本次实验为后续流水线 CPU 设计奠定了核心基础：

1. 流水线思想的预演：多周期 CPU 的五级流水线阶段划分与流水线 CPU 的阶段逻辑高度相似，通过本次实验已熟悉指令在各阶段的拆分与传递机制（如 PC 值在 IF/ID/EX 等阶段的独立锁存）。

2. 数据冲突与控制冲突处理伏笔：实验中观察到分支指令（如 BEQ）在 ID 阶段提前决定跳转方向，这与流水线 CPU 中分支预测机制具有相似性；而 Load 指令的延迟特性则提示未来需处理“Load-Use”数据冲突。

3. 模块化设计的复用价值：本次实验中实现的 ALU 模块、译码模块等可直接作为流水线 CPU 的子模块，只需进一步增加流水线寄存器和冲突检测逻辑，即可完成从多周期到流水线的升级。

（五）总结：从理论到实践的闭环成长

通过本次实验，我完成了从“理解多周期概念”到“动手设计模块→仿真验证→硬件实现”的完整闭环，深刻体会到计算机组成原理的“系统性”与“工程性”：

1. 知识层面：不再局限于单周期 CPU 的简单模型，而是从时序、模块协作、指令集扩展等维度构建了更立体的 CPU 认知；

2. 能力层面：Verilog 设计从“单模块编写”进阶到“多模块协同开发”，调试技能从“语法纠错”提升至“时序分析与系统级故障排查”；

3. 思维层面：培养了“细节决定成败”的工程思维（如 IP 核配置、总线位宽）和“从全局到局部”的系统思维（如流水线阶段划分、指令周期优化）。