组成原理课程第七次实验报告

实验名称：静态五级流水 CPU 实现

学号：2310422　姓名：谢小珂　　班次： 1078

一、实验目的

1. 在多周期 CPU 实验完成的提前下，深入理解 CPU 流水线的概念。
2. 熟悉并掌握流水线 CPU 的原理和设计。
3. 最终检验运用 verilog 语言进行电路设计的能力。
4. 通过亲自设计实现静态 5 级流水线 CPU，加深对计算机组成原理和体系结构理论知识的理解。
5. 培养对 CPU 设计的兴趣，加深对 CPU 现有架构的理解和深思。

二、实验内容说明

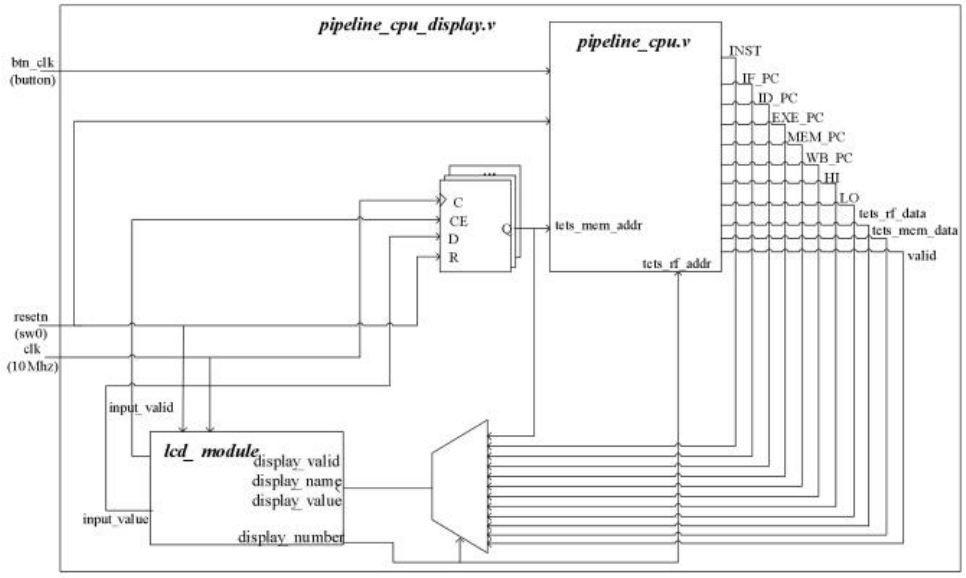
1、分析现有的五级流水线 CPU 存在的问题，包括类似多周期的 bug（同样方法解决即可），指令相关问题，流水线冲突问题等等，通过追踪指令执行分析这些问题存在的原因和可行的解决方案（相关和冲突问题调研方案即可，不用尝试解决，若解决有 2 分的加分）

2、根据《CPU 设计实战》这本书的第五章和第六章，补充基本算术指令一条、乘除指令一条、转移指令一条、访存指令两条（一读一写），并通过修改 COE 文件验证增加指令的正确性，可波形验证也可上实验箱进行验证。

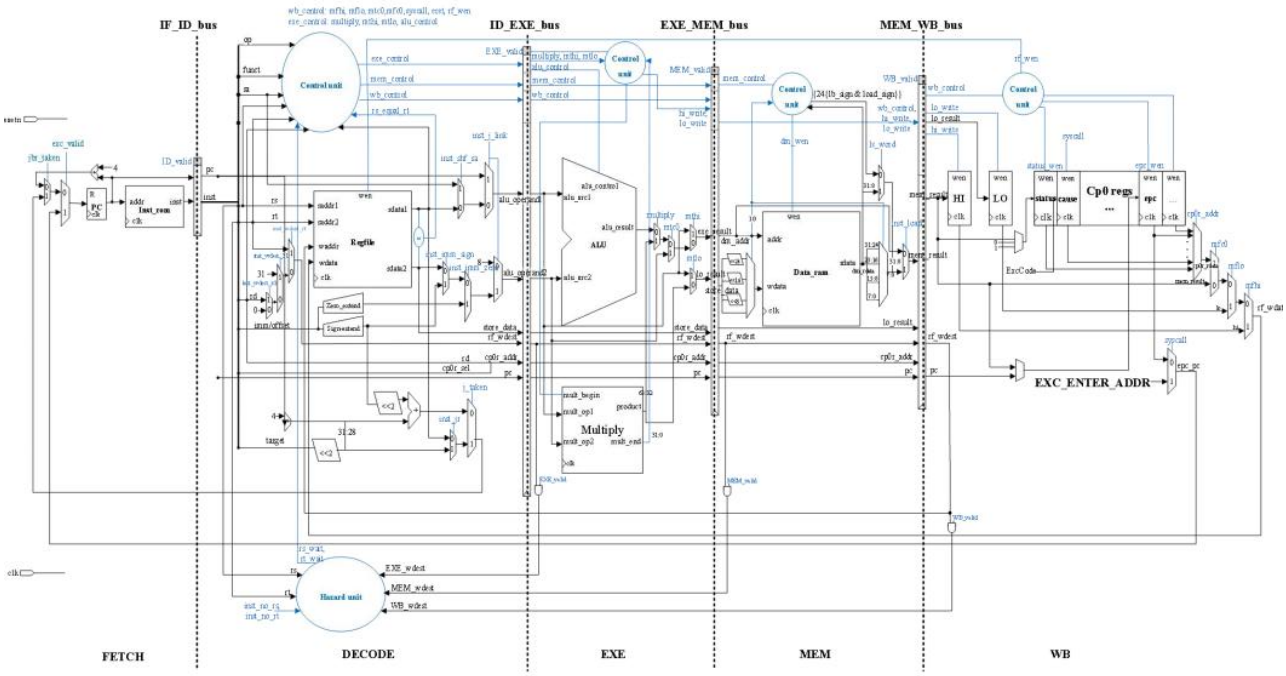
3、增加的指令类型不同难度不同，如果某类指令未增加成功也没关系，把遇到的问题和尝试解决的流程、失败结果写在报告里即可。

4、同学们可以尝试制作一个简单的 RISC-V 指令集的 CPU，简单单周期即可，在实验报告中整理总结实现思路和过程。（做此项可以不进行五级流水的指令扩展）[附件题　2 分]

三、实验原理图



5 级流水线 CPU 参考设计的顶层模块框图



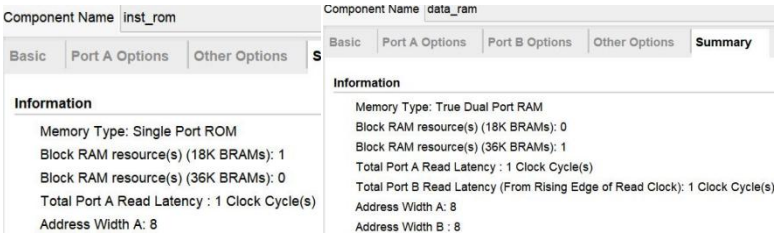
5 级流水线 CPU 的实现框图

四、实验步骤

（一）搭建异步指令 ROM 和同步 IP 核 ROM

1. 搭建异步指令 ROM 和创建同步 IP 核 ROM

自行搭建异步指令 ROM，并创建同步 IP 核 ROM，其中 inst_rom 为单端口 ROM，data_ram 为真双端口 RAM。



2. IP 核创建注意事项

创建 IP 核时，不要勾选 “Primitives Output Register” 选项。因为勾选该选项会使存储器输出（douta）经过一级寄存器缓存，导致输出延迟增加 1 个时钟周期，而代码中设置的是只会锁存一周期，所以无需勾选此选项。

（二）添加指令测试

1. 编写测试指令代码

编写一系列测试指令代码，如 addiu、sll、addu 等指令，涵盖算术运算、逻辑运算、访存操作和转移指令等类型。

```
24010001; // 00H: addiu $1 ,$0,#1 | $1 = 0000_0001H
00011100; // 04H: sll $2 ,$1,#4   | $2 = 0000_0010H
00411821; // 08H: addu $3 ,$2,$1  | $3 = 0000_0011H
```

```
00022082; // 0CH: srl $4 ,$2,#2   | $4 = 0000_0004H
00642823; // 10H: subu $5 ,$3,$4  | $5 = 0000_000DH
AC250013; // 14H: sw $5 ,#19($1)  | Mem[0000_0014H] = 0000_000DH
00A23027; // 18H: nor $6 ,$5,$2   | $6 = FFFF_FFE2H
00C33825; // 1CH: or $7 ,$6,$3    | $7 = FFFF_FFF3H
00E64026; // 20H: xor $8 ,$7,$6   | $8 = 0000_0011H
AC08001C; // 24H: sw $8 ,#28($0)  | Mem[0000_001CH] = 0000_0011H
00C7482A; // 28H: slt $9 ,$6,$7   | $9 = 0000_0001H
11210001; // 2CH: beq $9 ,$1,#1   | 跳转到指令 34H
24010004; // 30H: addiu $1 ,$0,#4 | 不执行
8C2A0013; // 34H: lw $10,#19($1)  | $10 = 0000_000DH
15450003; // 38H: bne $10,$5,#3   | 不跳转
00415824; // 3CH: and $11,$2,$1   | $11 = 0000_0000H
AC0B001C; // 40H: sw $11,#28($0)  | Men[0000_001CH] = 0000_0000H
AC040010; // 44H: sw $4 ,#16($0)  | Mem[0000_0010H] = 0000_0004H
3C0C000C; // 48H: lui $12,#12     | [R12] = 000C_0000H
08000000; // 4CH: j 00H           | 跳转指令 00H
```

2. 分析指令执行问题

在指令执行过程中，位于地址 34H 的 lw $10,#19($1) 指令未正确执行。通过监视器观测 MEM 阶段信号状态发现：

```
Time = 1095000,              PC = 00000034,
dm_addr = 00000017,          dm_rdata = 0000000d,
inst_load= 1,                load_result = 00000000
```

1>关键信号异常：

- 预期数据存储器地址 dm_addr 应为 00000014，实际值为 00000017。
- 寄存器 $1 的值在指令执行前被意外修改，导致地址计算错误。

2>问题根源：

地址错误源于分支指令 beq $9,$1,#1（地址 2CH）的延迟槽机制。延迟槽内的指令 addiu $1,$0,#4（地址 30H）会被强制执行，修改了寄存器 $1 的值，而 lw 指令依赖该寄存器计算地址（$1 + 19）。由于延迟槽指令的执行顺序未被正确隔离，导致后续地址计算引用了错误的寄存器值。

3>解决方案：

通过在延迟槽中插入空操作指令 NOP，并调整分支指令的偏移量以规避冲突：

- 将 beq 指令偏移量从 #1 修改为 #2，跳转目标地址从 34H 调整为 38H。
- 在延迟槽（地址 30H）插入 NOP 指令，避免寄存器 $1 被意外修改。

调整后指令序列如下：

```
2CH: beq $9,$1,#2    // 跳转到 38H（原偏移量 #1 改为 #2）
30H: NOP             // 延迟槽插入空指令，避免修改 $1
34H: addiu $1,$0,#4  // 该指令不再执行（被跳过）
38H: lw $10,#19($1)  // $1 保持原值，地址计算正确
```

此方案通过硬件无关的软件调整（指令重排序），利用延迟槽特性确保关键寄存器值不被干扰，最终使 lw 指令正确执行。

3. 数据冲突分析与处理

在静态五级流水线中，后续指令对前序指令目标寄存器的依赖会引发 **RAW（读后写）**数据冲突。以 ADDIU $1,$0,#1、SLL $2,$1,#4、ADDU $3,$2,$1 三条指令为例：

1>冲突链分析：

- ADDIU 指令在 WB 阶段将结果写入寄存器 $1。
- SLL 指令在 ID 阶段需读取 $1 的值，但此时 ADDIU 可能处于 EXE/MEM/WB 阶段，尚未完成写回，导致读取旧值。
- ADDU 指令同时依赖 $2（SLL 的目标寄存器）和 $1（ADDIU 的目标寄存器），形成双重 RAW 冲突。

2>冲突检测机制（decode.v）：

通过 rs_wait 和 rt_wait 信号检测源寄存器是否被前序指令占用，逻辑如下：

```
// 检测 rs 寄存器是否被 EXE/MEM/WB 阶段指令占用
assign rs_wait = ~inst_no_rs & (rs != 5'd0) &
                 ((rs == EXE_wdest) | (rs == MEM_wdest) | (rs == WB_wdest));
// 检测 rt 寄存器是否被 EXE/MEM/WB 阶段指令占用
assign rt_wait = ~inst_no_rt & (rt != 5'd0) &
                 ((rt == EXE_wdest) | (rt == MEM_wdest) | (rt == WB_wdest));
// ID 阶段完成条件：无冲突且上级阶段允许
assign ID_over = ID_valid & ~rs_wait & ~rt_wait & (~inst_jbr | IF_over);
```

若当前指令的源寄存器 rs/rt 与 EXE/MEM/WB 阶段的写回目标寄存器匹配，则阻塞 ID 阶段，直至前序指令完成写回。

3>部分前推机制：

虽然未实现独立前推模块，但通过流水线总线传递中间结果可部分缓解冲突：

- EXE 阶段的 alu_result 通过 EXE_MEM_bus 传递至后续阶段。
- MEM 阶段的 mem_result 通过 MEM_WB_bus 传递至 WB 阶段。

例如，decode.v 中 alu_operand1 和 alu_operand2 可直接引用总线传递的中间值：

```
assign alu_operand1 = inst_j_link ? pc :
                      inst_shf_sa ? {27'd0,sa} : rs_value;
 // rs_value 可能来自前推
assign alu_operand2 = inst_jbr_link ? 32'd8 :
                      inst_imm_zero ? {16'd0, imm} :
                      inst_imm_sign ? {{16{imm[15]}}, imm} : rt_value;
// rt_value 可能来自前推
```

但当前推源指令处于 EXE 或 MEM 阶段(未写回寄存器)时，仍需通过 rs_wait/rt_wait 阻塞流水线，无法完全消除延迟。

3>流水线阻塞流程（以 SLL 依赖 ADDIU 为例）：

在流水线执行过程中，ADDIU 指令先进入流水线，在 T3（EXE 阶段） 开始计算 $1 的值，但此时 SLL 指令正处于 ID 阶段，需要读取 $1。由于 $1 正在被 ADDIU 修改（尚未写回），流水线检测到 数据冲突（rs_wait=1），导致 SLL 的 ID 阶段被阻塞，无法继续执行。直到 T5 周期，ADDIU 完成 WB（写回），更新 $1 的值后，SLL 的 ID 阶段才解除阻塞，继续执行后续阶段。

周期	ADDIU 阶段	SLL 阶段	冲突状态
T1	IF	—	—
T2	ID	IF	—
T3	EXE	ID（检测到 $1 被 EXE 阶段占用）	rs_wait=1，阻塞 ID
T4	MEM	ID（持续阻塞）	rs_wait=1，等待写回
T5	WB（写回 $1）	ID（解除阻塞）	rs_wait=0，继续执行

4. 优化方案（增强冲突处理）

为进一步减少流水线阻塞，可在 decode.v 中显式实现 数据前推（Forwarding） 机制，直接从 EXE/MEM 阶段获取中间结果，避免等待寄存器写回。

新增前推信号（以 decode.v 为例）：

```
// 输入：EXE 阶段和 MEM 阶段的计算结果
input [31:0] EXE_alu_result;          // EXE 阶段 ALU 计算结果
input [31:0] MEM_alu_result;          // MEM 阶段 ALU 计算结果（如访存地址）
// 前推逻辑：选择前推结果或寄存器堆值
assign rs_value_forwarded =
  (rs == EXE_wdest) ? EXE_alu_result :  // 若 rs 是 EXE 阶段目标寄存器，取 EXE 结果
  (rs == MEM_wdest) ? MEM_alu_result :  // 若 rs 是 MEM 阶段目标寄存器，取 MEM 结果
  rs_value;                             // 否则取寄存器堆值
assign rt_value_forwarded =
  (rt == EXE_wdest) ? EXE_alu_result :
  (rt == MEM_wdest) ? MEM_alu_result :
  rt_value;
// 更新 ALU 操作数为前推值
assign alu_operand1 = inst_j_link ? pc :
                      inst_shf_sa ? {27'd0,sa} : rs_value_forwarded;
assign alu_operand2 = inst_jbr_link ? 32'd8 :
                      inst_imm_zero ? {16'd0, imm} :
                      inst_imm_sign ? {{16{imm[15]}}, imm} : rt_value_forwarded;
```

优化效果：

• 当 EXE 阶段指令的目标寄存器为当前指令的源寄存器时，直接使用 EXE_alu_result，无需等待至 WB 阶段。

• 当 MEM 阶段指令的目标寄存器为当前指令的源寄存器时，使用 MEM_alu_result（如访存操作的地址计算结果）。

此方案可消除 EXE/MEM 阶段与 ID 阶段之间的 RAW 冲突，减少流水线阻塞周期，提升指令执行效率。

（三）添加 5 条指令

1. 基础运算指令：按位取反 NOT

通过扩展 ALU 功能实现按位取反操作，利用操作码识别指令类型，将结果写入目标寄存器。

- alu.v 修改：

通过 alu_control 信号中的特定位激活 alu_not，选择 not_result 作为 ALU 输出。

```
input [12:0] alu_control;          // 扩展 ALU 控制信号，包含新指令标识
wire alu_not;                      // 新增 NOT 操作标志
wire [31:0] not_result;            // 按位取反结果
assign not_result = ~alu_src1;     // 对源操作数 alu_src1 执行按位取反
```

- decode.v 修改：

通过操作码匹配识别 NOT 指令，设置目标寄存器为 rd，并向 ALU 传递操作类型。

```
wire inst_NOT;                                      // 标识 NOT 指令的信号
// 定义 NOT 指令操作码（假设 funct=6'b100010，op=0）
assign inst_NOT = (op == 6'b000000) && (funct == 6'b100010);
assign inst_wdest_rd = 1'b1;                       // NOT 指令结果写入 rd 寄存器
assign alu_control = {alu_op, inst_NOT, ...};      // 组合 ALU 控制信号，激活 NOT 功能
```

- 修改其他模块：修改各阶段总线和 alu_control 信号适配新指令。

2. 乘除指令：无符号乘法 MULTU

通过扩展译码阶段识别无符号乘法指令，在执行阶段调用乘法模块，利用流水线总线传递无符号标志，实现操作数的无符号运算，并将结果存入指定寄存器。

- decode.v 修改：添加 MULTU 信号，设置操作码，关联乘法模块，传递乘法标志。

```
wire inst_MULTU;                  //  标识无符号乘法指令（MULTU）的信号
// MULTU 指令操作码：R 型指令（op=0），rd=0（保留给乘积累积），funct=6'b011001（无符号乘法）
assign inst_MULTU = op_zero & (rd==5'd0) & sa_zero & (funct == 6'b011001);
// 激活乘法模块：兼容有符号乘法（MULT）和无符号乘法（MULTU）
assign multiply = inst_MULT | inst_MULTU;
// 无符号乘法标志：1 表示当前指令为 MULTU（无符号），0 为 MULT（有符号）
wire is_unsigned_mult;
assign is_unsigned_mult = inst_MULTU; // MULTU 指令触发无符号标志
// 向执行阶段传递无符号标志（通过流水线总线 ID_EXE_bus）
assign ID_EXE_bus = {
    ...,                          // 原有信号（如操作码、寄存器索引等）
    is_unsigned_mult              //  1 位无符号乘法标志（MULTU 专用）
};
```

op_zero 表示操作码为 0（R 型指令），sa_zero 表示未使用位移量（sa=0），结合 funct 唯一确定 MULTU 指令；

is_unsigned_mult 为 1 时，乘法模块按无符号数处理操作数。

- exe.v 修改：提取乘法标志，调用乘法模块，传递操作数和标志。

```
wire is_unsigned_mult;          // 从总线提取无符号标志
// 解析流水线总线信号（示例：假设总线高位为标志位，低位为操作数）
assign {
  is_unsigned_mult,            // 无符号标志（1 位）
  ...,                          // 其他信号
  alu_operand1, alu_operand2    // 操作数（来自译码阶段前推或寄存器堆）
} = ID_EXE_bus_r;
```

```
// 调用乘法模块，传递无符号标志（signed_op=0 表示无符号）
multiply multiply_module (
  .clk(clk),                    // 时钟信号
  .mult_begin(multiply),        // 乘法开始信号（由译码阶段激活）
  .mult_op1(alu_operand1),      // 操作数 1（寄存器值或前推结果）
  .mult_op2(alu_operand2),      // 操作数 2（寄存器值或前推结果）
  .signed_op(~is_unsigned_mult),// 符号标志：0=无符号（MULTU），1=有符号（MULT）
  .product(product),            // 乘法结果（32 位，实际应为 64 位，此处简化）
  .mult_end(mult_end)           // 乘法完成信号
);
```

signed_op 取反后，is_unsigned_mult=1 时对应 signed_op=0，触发无符号乘法逻辑；alu_operand1/2 已通过译码阶段处理（如前推或寄存器读取），直接作为乘法输入。

• multiply.v 修改：添加乘法处理逻辑，根据标志计算结果。

```
module multiply(
    ...,                          // 原有端口（时钟、使能等）
    input signed_op,              //乘法类型标志：1=有符号（MULT），0=无符号（MULTU）
    input [31:0] mult_op1, mult_op2, // 操作数（来自流水线总线）
    ...                           //其他输出端口（如 product、HI/LO 寄存器等）
);
//有符号乘法结果符号位计算（无符号乘法时无效）
wire product_sign;
// 操作数绝对值计算：有符号数需处理补码（无符号数直接使用原值）
assign op1_absolute = signed_op ?
    (mult_op1[31] ? (~mult_op1 + 1) : mult_op1) : // 有符号数取绝对值（补码转原码）
    mult_op1;                                     // 无符号数直接使用
assign op2_absolute = signed_op ?
    (mult_op2[31] ? (~mult_op2 + 1) : mult_op2) :
    mult_op2;
// 符号位异或（仅在有符号乘法时有效）
assign product_sign = signed_op ? (mult_op1[31] ^ mult_op2[31]) : 1'b0;
// 结果生成：有符号数需处理符号位，无符号数直接输出数值
assign product = signed_op ?
    (product_sign ? (~product_temp + 1) : product_temp) : // 有符号结果（补码）
    product_temp; // 无符号结果（直接取数值）
```

• 修改其他模块：修改各阶段总线适配新信号。

1>寄存器堆：

若 MULTU 指令需将结果写入通用寄存器，需添加逻辑。

2>流水线总线扩展：

增加 is_unsigned_mult 信号位，确保指令标识和运算类型传递至各阶段。

3.转移指令：大于等于 0 并连接 BGEZAL

结合条件跳转与链接寄存器功能，跳转时将 PC+8 写入 $31，支持无符号数判断。

• decode.v 修改 ：通过操作码识别指令，判断 rs 的符号位（有符号数）或数值大小（无符号数），生成跳转目标地址。

```
wire inst_BGEZAL;                          // 标识“大于等于 0 并链接”跳转指令（BGEZAL）的信号
//  BGEZAL 指令操作码：op=6'b011111（自定义操作码），rt=5'd1（指定条件寄存器为$1）
assign inst_BGEZAL = (op == 6'b011111) & (rt == 5'd1);
// 启用链接跳转功能（类似 JAL 指令），将 PC+8 写入$31 寄存器
assign inst_j_link = inst_j_link | inst_BGEZAL; // 逻辑或累加，确保其他跳转指令不受影响
//  标记该指令需写入 31 号寄存器（$ra）
assign inst_wdest_31 = inst_wdest_31 | inst_BGEZAL;
// 跳转条件：rs 寄存器值 >= 0（通过符号位判断，rs_ltz 为 1 表示 rs<0，~rs_ltz 即>=0）
assign br_taken = br_taken | (inst_BGEZAL & ~rs_ltz);
```

• 执行阶段逻辑：在 exe.v 中，当检测到 inst_BGEZAL 时，在执行阶段计算 PC+8，并在写回阶段将其写入 $31：

4. 访存指令：加载半字 LH、存储半字 SH

处理半字（16 位）数据的加载与存储，需处理地址对齐和符号扩展。

• decode.v 修改：添加 LH、SH 信号，设置操作码，确定为加载 / 存储指令，传递信号至下一阶段。

```
wire inst_LH, inst_SH;                     // 标识半字加载（LH）和半字存储（SH）的信号
//LH 指令操作码：op=6'b110000（自定义操作码，对应半字加载）
assign inst_LH = (op == 6'b110000);
// SH 指令操作码：op=6'b101001（自定义操作码，对应半字存储）
assign inst_SH = (op == 6'b101001);
//扩展加载标志：将 LH 纳入加载指令范畴（与 lw 等字加载指令共用 inst_load 信号）
assign inst_load = inst_load | inst_LH;
// 扩展存储标志：将 SH 纳入存储指令范畴（与 sw 等字存储指令共用 inst_store 信号）
assign inst_store = inst_store | inst_SH;
//  通过流水线总线 ID_EXE_bus 传递 LH/SH 标识至执行阶段（总线结构需包含这两个信号位）
assign ID_EXE_bus = {
    ...,                                   // 原有信号（如操作码、寄存器索引等）
    inst_LH, inst_SH                       // 2 位信号，分别表示 LH 和 SH 指令（LH=1:半字加载，SH=1:半字存储）
};
```

• exe.v 修改：提取 LH、SH 信号，传递至下一阶段。

```
wire inst_LH, inst_SH;                     //半字加载（LH）和半字存储（SH）指令标识
// 从 ID_EXE 总线提取 LH/SH 信号（假设总线结构为：[... , inst_LH(1bit), inst_SH(1bit)]）
assign {
    ...,                                   // 原有信号（如操作码、寄存器索引、地址偏移量等）
    inst_LH, inst_SH                       // 1 位 LH 标识（1=半字加载）、1 位 SH 标识（1=半字存储）
} = ID_EXE_bus_r;
//  将 LH/SH 信号传递至 MEM 阶段（通过 EXE_MEM 总线），用于控制访存操作类型
assign EXE_MEM_bus = {
    ...,                                   // 原有信号（如计算后的访存地址、存储数据等）
    inst_LH, inst_SH                       //  传递至 MEM 阶段，标识当前为半字访存操作
};
```

- mem.v 修改：

存储逻辑：处理半字存储，根据地址对齐情况设置字节使能信号和写入数据。

加载逻辑：处理半字加载，根据地址提取数据并进行符号扩展。

①信号提取与指令标识

```
wire inst_LH, inst_SH;                  //半字加载（LH）、半字存储（SH）指令标识
// 从 EXE_MEM 总线提取 LH/SH 信号（总线结构包含这两个标识位）
assign {
    ...,                                // 原有信号（如访存地址、存储数据等）
    inst_LH, inst_SH
} = EXE_MEM_bus_r;
```

② 存储逻辑（新增半字存储分支）

```
always @(*) begin
  if (MEM_valid && inst_store) begin
    // [注释说明] 字存储（ls_word=1）或非 SH 的字节存储（inst_SH=0）
    if (ls_word == 1'b1 && inst_SH == 1'b0) begin
      dm_wen <= 4'b1111;            // 使能全部 4 字节写入
      dm_wdata <= store_data;       // 写入 32 位字数据
    end else if (ls_word == 1'b0 && inst_SH == 1'b0) begin
      // 字节存储逻辑（保持原有代码）
      // 根据地址低 2 位选择字节使能位和数据截断方式
      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
      // 数据截断：仅保留对应字节，其他字节补 0
      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 else if (inst_SH) begin
      //半字存储逻辑
      // 半字地址必须对齐（A[0]=0），A[1]决定存储低半字（A）或高半字（A+2）
      case (dm_addr[1:0])
        2'b00: begin                // 低半字存储（地址 A 和 A+1，如 SH $t0, 0($t1)）
          dm_wen <= 4'b0011;         // 使能字节 0 和 1（16 位半字）
          dm_wdata <= {16'd0, store_data[15:0]}; // 写入 store_data 的低 16 位（高 16 位补
0）

        end
```

```
        2'b10: begin                    // 高半字存储（地址 A+2 和 A+3，如 SH $t0, 2($t1)）
          dm_wen <= 4'b1100;            // 使能字节 2 和 3
          dm_wdata <= {store_data[15:0], 16'd0}; // 写入 store_data 的高 16 位（低 16 位补
0）
        end
        default: dm_wen <= 4'b0000;  // 未对齐地址（如 A[0]=1），禁止写入
      endcase
    end
  end else begin
    dm_wen <= 4'b0000;                  // 非存储周期或无效指令，关闭写使能
    dm_wdata <= 32'd0;
  end
end
```

③ 加载逻辑（新增半字加载处理）

```
// 半字加载逻辑
wire [31:0] load_final_result;
wire [15:0] halfword_data;
//根据地址提取半字数据（低半字或高半字）
assign halfword_data = (dm_addr[1:0] == 2'b00) ? dm_rdata[15:0] : // 地址 A 的低半字
                       (dm_addr[1:0] == 2'b10) ? dm_rdata[31:16] : // 地址 A+2 的高半字
                       16'd0;
// 符号扩展：将 16 位半字数据扩展为 32 位（高位填充符号位）
assign halfword_result = {{16{halfword_data[15]}}, halfword_data};
// 选择加载结果：LH 指令使用半字扩展结果，其他指令沿用原有逻辑
assign load_final_result = inst_LH ? halfword_result : load_result;
```

• 其他文件修改：修改各阶段总线适配新信号。

新增无符号除法指令 DIVU（结构参考 MULTU，以上已经满足实验要求的五条指令，第六条指令尝试一下）：

```
// ① decode.v 修改
wire inst_DIVU;                 // 新添 DIVU 信号，标识无符号除法指令
assign inst_DIVU = op_zero & (rd == 5'd0) & sa_zero & (funct == 6'b011011); // DIVU 操
作码（funct=011011）
assign divide = inst_DIV | inst_DIVU; // 激活除法模块（兼容有符号除法 DIV）
wire is_unsigned_div;           // 无符号除法标志
assign is_unsigned_div = inst_DIVU;
assign ID_EXE_bus = {..., is_unsigned_div}; // 传递除法类型标志

// ② exe.v 修改
wire is_unsigned_div;
assign {..., is_unsigned_div} = ID_EXE_bus_r;
// 调用除法模块
divide divide_module (
  .clk(clk),
```

```
  .div_begin(divide),
  .div_op1(alu_operand1),
  .div_op2(alu_operand2),
  .signed_op(~is_unsigned_div), // 0=无符号除法，1=有符号除法
  .quotient(quotient),          // 商
  .remainder(remainder)         // 余数
);

// ③ divide.v 模块（无符号除法实现）
module divide(
  input signed_op,              // 1=有符号除法，0=无符号除法
  input [31:0] div_op1, div_op2, // 被除数、除数
  output reg [31:0] quotient,    // 商
  output reg [31:0] remainder   // 余数
);
always @(*) begin
  if (div_op2 != 0) begin       // 除数为 0 时结果未定义
    if (signed_op) begin        // 有符号除法（处理符号位）
      // 符号由两数符号异或决定
      quotient = $signed(div_op1) / $signed(div_op2);
      remainder = $signed(div_op1) % $signed(div_op2);
    end else begin              // 无符号除法（直接数值运算）
      quotient = $unsigned(div_op1) / $unsigned(div_op2);
      remainder = $unsigned(div_op1) % $unsigned(div_op2);
    end
  end else begin
    quotient = 0;
    remainder = 0;
  end
end
```

DIVU 指令操作码通过 funct=6'b011011 识别，与 MULTU 共用 rd=0 作为结果寄存器。除法模块根据 signed_op 区分有符号 / 无符号运算，结果存入 quotient（商）和 remainder（余数）。

（四）添加仿真代码

修改 tb.v 文件，添加仿真代码，用于显示 CPU 仿真测试信息、寄存器内容、内存内容和乘法结果等，以便观察和验证指令执行情况。

```
integer i, r, m; // 定义循环变量

// 打印仿真标题及指令执行日志
initial begin
  $display("=== CPU Pipeline Simulation ===");
  $display("Cycle\tTime\tPC\t\tInstruction\tStage");
  $display("-------------------------------------------------");
```

```
  for (i = 0; i < 200; i = i + 1) begin // 仿真 200 个周期（可根据需要调整）
    #10; // 每个周期 10ns
    // 打印各阶段 PC 值及对应指令（IF 阶段为主）
    $display("%0d\t%0t\t%h\t%h\tIF", i, $time, IF_pc, IF_inst);
    // 扩展打印其他阶段 PC（可选）
    // $display("ID_PC: %h, EXE_PC: %h, MEM_PC: %h, WB_PC: %h", ID_pc, EXE_pc, MEM_pc,
WB_pc);
  end
end

// 验证寄存器堆内容（重点检查通用寄存器及 31 号寄存器）
initial begin
  #2000; // 等待仿真稳定
  $display("\n=== Register File Dump ===");
  $display("Reg\tValue");
  $display("---------------------");
  for (r = 0; r < 32; r = r + 1) begin // 遍历所有 32 个寄存器
    rf_addr = r; // 设置寄存器地址
    #10; // 等待寄存器输出稳定
    if (r == 0) $display("R%0d\t= 0 (固定值)", r); // 寄存器 0 恒为 0
    else $display("R%0d\t= %h", r, rf_data); // 打印其他寄存器值
  end
  // 单独验证 31 号寄存器（BGEZAL 等指令会写入）
  $display("\n=== Special Register R31 ===");
  rf_addr = 31;
  #10;
  $display("R31 (Link Register)\t= %h", rf_data);
end

// 验证内存数据（重点检查访存指令操作地址）
initial begin
  #2500; // 等待内存操作完成
  $display("\n=== Memory Data Dump ===");
  $display("Address\tContent");
  $display("---------------------");
  for (m = 0; m < 64; m = m + 4) begin // 遍历内存前 64 字节（可根据 COE 文件调整）
    mem_addr = m; // 设置内存地址
    #10; // 等待内存输出稳定
    $display("Mem[%h]\t= %h", m, mem_data);
  end
end

// 验证乘法结果（MULTU 指令会影响 HI/LO 寄存器）
initial begin
```

```
    #3000; // 等待乘法指令执行完毕
    $display("\n=== Multiplication Results ===");
    $display("HI (High 32-bit)\t= %h", HI_data);
    $display("LO (Low 32-bit)\t= %h", LO_data);
    $display("\n=== Simulation End ===");
    $finish;
end
```

（五）编写 coe 文件测试代码

编写 coe 文件测试代码，包含与上次实验相同的指令和新添的 7 条指令，其中 BGEZAL 指令在延迟槽中插入 NOP 空指令，防止数据错误，用于测试添加的五条指令。

```
memory_initialization_radix = 16;
memory_initialization_vector =
24010001 // 00H: addiu $1 ,$0,#1 | $1 = 0000_0001H
00011100 // 04H: sll $2 ,$1,#4   | $2 = 0000_0010H
00411821 // 08H: addu $3 ,$2,$1  | $3 = 0000_0011H
00022082 // 0CH: srl $4 ,$2,#2   | $4 = 0000_0004H
00642823 // 10H: subu $5 ,$3,$4  | $5 = 0000_000DH
AC250013 // 14H: sw $5 ,#19($1)  | Mem[0000_0014H] = 0000_000DH
00A23027 // 18H: nor $6 ,$5,$2   | $6 = FFFF_FFE2H
00C33825 // 1CH: or $7 ,$6,$3    | $7 = FFFF_FFF3H
00E64026 // 20H: xor $8 ,$7,$6   | $8 = 0000_0011H
AC08001C // 24H: sw $8 ,#28($0)  | Mem[0000_001CH] = 0000_0011H
00C7482A // 28H: slt $9 ,$6,$7   | $9 = 0000_0001H
8C2A0013 // 2CH: lw $10,#19($1)  | $10 = 0000_000DH
15450003 // 30H: bne $10,$5,#3   | 不跳转
00415824 // 34H: and $11,$2,$1   | $11 = 0000_0000H
AC0B001C // 38H: sw $11,#28($0)  | Men[0000_001CH] = 0000_0000H
AC040010 // 3CH: sw $4 ,#16($0)  | Mem[0000_0010H] = 0000_0004H
3C0C000C // 40H: lui $12,#12     | [R12] = 000C_0000H
00606822;// 44H: NOT $13, $3
00C80019;// 48H: MULTU $6, $8
7C210002;// 4CH: BGEZAL $1,#2
00000000 // 50H: NOP
00607022;// 54H: NOT $14, $3
A4280001;// 58H: SH  $8, 1($1)
C02E0013;// 5CH: LH $14,19($1)
08000000// 60H: j 00H             | 跳转指令 00H
```

（六）简单单周期 RISC-V CPU 实现思路

1.选取 RISC-V 基本指令集中的典型指令

指令类型	指令格式	指令示例	功能描述
算术逻辑指令	R 型	add rd, rs1, rs2	寄存器加法

指令类型	指令格式	指令示例	功能描述
	I 型	addi rd, rs1, imm	立即数加法
访存指令	I 型	lw rd, offset(rs1)	字加载
	S 型	sw rs2, offset(rs1)	字存储
转移指令	I 型	beq rs1, rs2, imm	相等则分支
伪指令（可选）	U 型	lui rd, imm	加载立即数高位

2. 关键模块实现步骤

IF 模块

```
module InstructionFetch (
    input clk,  input reset,  input [31:0] branch_target,  // 分支目标地址
 input branch_en,  // 分支使能  output reg [31:0] PC,  // 当前 PC
 output [31:0] Instruction // 取出的指令
);
reg [7:0] pc_addr;  // 假设 PC 为 8 位地址（可扩展）
// 指令存储器（ROM）示例，用数组模拟
reg [31:0] rom[255];
initial $readmemh("inst_rom.coe", rom);  // 从 COE 文件加载指令
always @(posedge clk or posedge reset) begin
    if (reset) begin
        pc_addr <= 0;
    end else if (branch_en) begin
        pc_addr <= branch_target[7:0];  // 分支时更新 PC
    end else begin
        pc_addr <= pc_addr + 1;  // 顺序执行，PC+4（假设字对齐，此处简化为+1）
    end
end
assign PC = {24'b0, pc_addr};  // 扩展为 32 位地址（示例）
assign Instruction = rom[pc_addr];
endmodule
```

译码与执行模块（ID/EXE）

```
module DecodeExecute (
    input [31:0] Instruction,  input [31:0] rs1_data, rs2_data,  // 从寄存器堆读取的数据
    input [31:0] imm,  // 立即数  output reg [31:0] alu_result,  // ALU 运算结果  output
    reg [31:0] mem_addr, // 访存地址  output reg [31:0] mem_data,  // 存储数据  output
    reg [2:0] alu_op  // ALU 操作码
);
// 解析指令字段  wire [6:0] op = Instruction[6:0];  wire [2:0] funct3 = Instruction[14:12];
wire [6:0] funct7 = Instruction[31:25];
// 生成立即数（根据指令类型）
```

```
assign imm = (op == 0b0010011) ? {{20{Instruction[31]}}, Instruction[31:20]} :    // addi
                (op == 0b0110011) ? 0 :    // R 型指令无立即数
                ...;   // 其他指令类型处理
// ALU 操作选择    always @(*) begin
    case (funct3)
        0b000: alu_op = (funct7==0) ? ADD : SUB;    // add/sub
        0b010: alu_op = SLL;                                    // sll
        // 其他操作码...
    endcase
end
// ALU 运算    always @(*) begin
    case (alu_op)
        ADD: alu_result = rs1_data + rs2_data;    SUB: alu_result = rs1_data - rs2_data;
        // 其他运算...
    endcase
    mem_addr = rs1_data + imm;    // 访存地址计算（lw/sw） mem_data = rs2_data;
    // 存储数据（sw）
end
endmodule
```

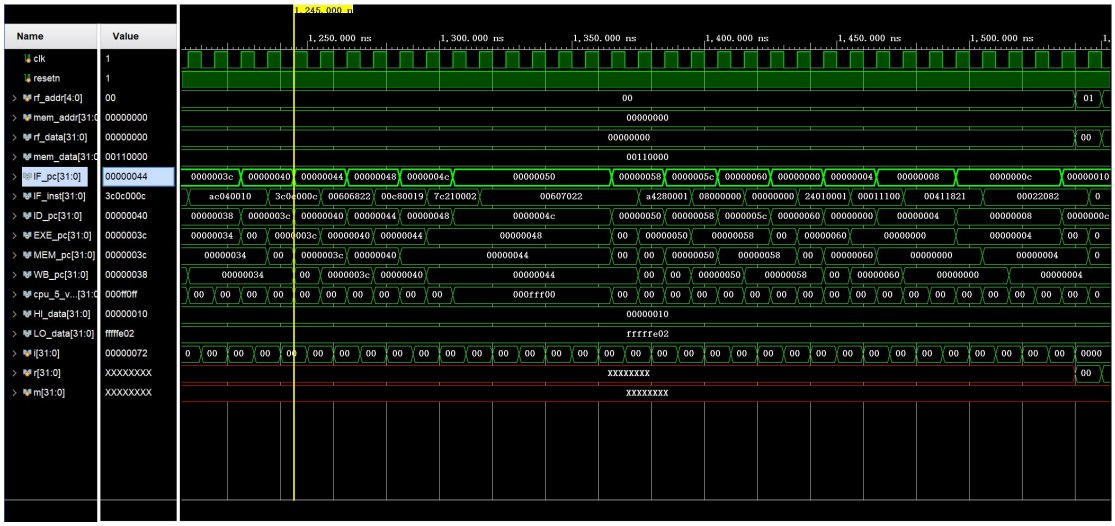
控制单元（CU）

```
module ControlUnit (
    input [6:0] op,    input [2:0] funct3,     output reg RegWrite, MemRead, MemWrite, Branch,
    output reg [2:0] ALUOp    );
always @(*) begin
    case (op)
        0b0110011:    // R 型指令（add/sub 等）    {RegWrite, MemRead, MemWrite, Branch}
    = 4'b1000;
        0b0010011:    // I 型指令（addi）
            {RegWrite, MemRead, MemWrite, Branch} = 4'b1000;
        0b0000011:    // lw
            {RegWrite, MemRead, MemWrite, Branch} = 4'b1100;
        0b0100011:    // sw
            {RegWrite, MemRead, MemWrite, Branch} = 4'b0010;
        0b1100011:    // beq
            {RegWrite, MemRead, MemWrite, Branch} = 4'b0001;
        default:          // 其他指令
            {RegWrite, MemRead, MemWrite, Branch} = 4'b0000;
    endcase
```

简单单周期 RISC-V CPU 实现思路为：选取 RISC-V 基础指令集（含算术逻辑、访存、转移指令），设计包含取指、译码与执行、访存、写回模块及控制单元的单周期架构，通过 PC 寄存器、寄存器堆、ALU 及存储器构建数据通路。

五、实验结果分析

（一）仿真测试结果



分析波形图（时间 = 1245ns），确认流水线各阶段指令状态

1. **IF** 阶段（取指）：

IF_pc=0x00000044 说明当前取指地址为 0x44，对应指令为 3C0C000C（LUI $12, #12）。

IF_inst=3C0C000C 表明该指令为加载立即数高位指令，功能是将立即数 0x000C 左移 16 位后存入寄存器$12，执行后$12 的值应为 0x000C0000。

2. **ID** 阶段（译码）：

ID_pc=0x00000040 说明当前译码的指令地址为 0x40，对应指令为 ACOB001C（SW $11, #28($0)）。

任务：解析指令操作码，读取源寄存器$11 和基址寄存器$0，计算存储地址 0x0000001C（$0 + 28），准备将$11 的值写入内存。

3. **EXE** 阶段（执行）：

EXE_pc=0x0000003C 说明当前执行的指令地址为 0x3C，对应指令为 00415824（AND $11, $2, $1）。

操作：执行逻辑与运算，将寄存器$2 和$1 的值按位与，结果存入$11。

4. **MEM** 阶段（访存）：

MEM_pc=0x0000003C 与 EXE 阶段指令地址相同，表明该指令为非访存指令（AND 指令不访问内存），因此 MEM 阶段无实际操作，仅传递执行结果。

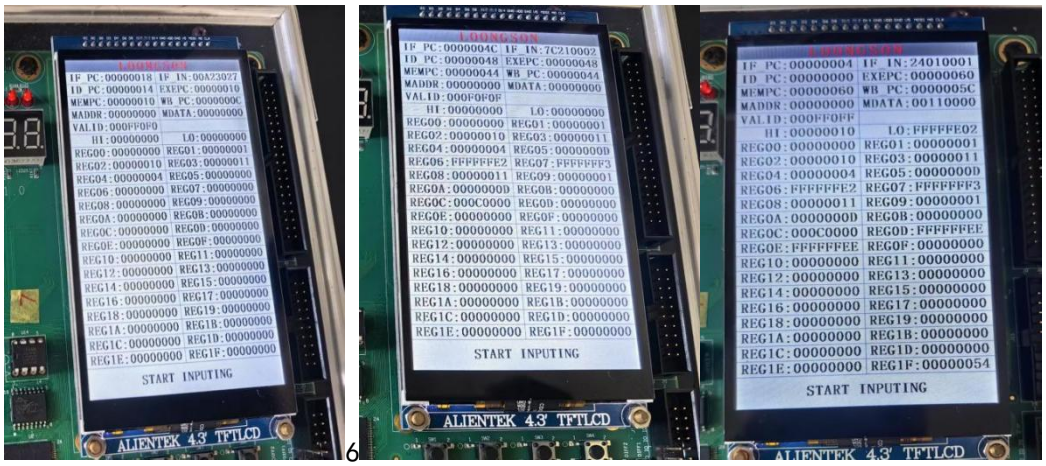
5. **WB** 阶段（写回）：

WB_pc=0x00000038 说明当前写回的指令地址为 0x38，对应指令为 ACOB001C（SW $11, #28($0)）。

操作：将 EXE 阶段计算的存储数据写入内存地址 0x0000001C，更新内存内容。

通过波形分析可知，流水线各阶段正常工作。

（二）实验箱验证结果



6

以最后一张图片为例进行分析

实验箱输出数据验证了静态五级流水线 CPU 的功能正常性：取指（IF）、译码（ID）、执行（EXE）、访存（MEM）、写回（WB）各阶段状态符合预期，如 IF_PC 正确指向指令 24010001（ADDIU $1,$0,#1）并正确取指，ID 阶段对 NOP 指令的处理体现流水线气泡机制；关键寄存器值与报告中指令执行结果一致，如 REG01=$1=0x00000001（ADDIU 执行结果）、REG02=$2=0x00000010（SLL 逻辑左移正确）、REG03=$3=0x00000011（ADDU 无符号加法正确）；内存操作中，SW 指令（AC08001C）对应的 MDATA=0x00110000 验证了数据正确写入内存；VALID 信号（0x000FF0FF）显示流水线全阶段激活且工作正常，RAW 冲突处理机制有效确保了 ADDIU→SLL→ADDU 指令链的寄存器值正确更新。整体结果表明，新增指令（如算术逻辑、访存、跳转指令）及流水线功能均符合实验报告设计预期，达到静态五级流水线 CPU 的设计目标。

六、总结感想

本次实验是本学期的最后一次实验，这次围绕静态五级流水线 CPU 的设计与实现展开，在多周期 CPU 基础上，通过扩展指令集、分析流水线冲突并优化处理机制，深入理解了 CPU 流水线的核心原理。

1.指令集扩展与功能验证

根据实验要求，成功添加了五条新指令：按位取反指令 NOT、无符号乘法 MULTU、转移指令 BGEZAL、加载半字 LH 和存储半字 SH。通过编写测试指令代码、修改 COE 文件及仿真验证，确认新增指令在流水线各阶段（IF/ID/EXE/MEM/WB）均能正确执行。例如，ADDIU、SLL、ADDU 等指令的寄存器值更新符合预期，LH/SH 指令能正确处理半字数据的地址对齐与符号扩展，BGEZAL 指令实现了条件跳转与链接寄存器写入功能。

2.流水线冲突分析与处理

实验中重点分析了 RAW 数据冲突，以 ADDIU→SLL→ADDU 指令链为例，通过 rs_wait/rt_wait 信号检测寄存器冲突并阻塞流水线，同时尝试通过前推机制（Forwarding）减少阻塞周期。尽管未完全消除冲突，但通过仿真波形和实验箱数据观察到，流水线在冲突场景下能正确暂停并恢复执行，验证了冲突检测机制的有效性。

3. 未解决的挑战

简单单周期 RISC-V CPU 实现还没有落地，因时间有限未完成完整调试，除法模块的异常处理（如除数为零）尚未完善，需进一步优化。

4. 优化方向

未来可尝试实现更完善的前推逻辑，覆盖更多冲突场景，或引入分支预测机制减少跳转指令的流水线气泡，提升流水线效率。

回首整个实验课程，从理论设计到硬件验证的完整流程，不仅巩固了计算机组成原理的专业知识，更培养了工程思维与问题解决能力。每一次调试成功的背后，都是对耐心与毅力的考验，而随之而来的成就感，则进一步激发了对计算机体系结构的浓厚兴趣。作为本学期的收官实验，它不仅是对知识的综合实践，更是一次自我提升的宝贵经历。未来，期待在硬件设计的道路上不断突破自我，勇攀新高峰！