组成原理实验课程第6次实验报告

实验名称： 五级流水线CPU

学生姓名：许洋 学号：2313721 班级：张金老师

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

请根据实验指导手册及source_code中的五级流水源码，分析现有的五级流水线存在的问题，并实现五级流水线CPU实验的bug修复，然后参考《CPU设计实战》这本书完成对五级流水线的指令扩展，满分100分，细分要求如下：

1、针对现有五级流水线存在问题的分析，不只是bug，包括指令相关、流水线冲突等各种能分析的问题进行分析（20分）

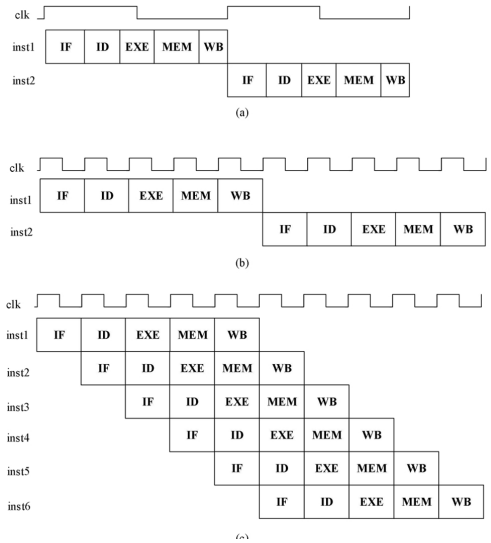
2、五级流水线指令运行时的bug修复（20分）

3、五级流水线指令扩展，运算类指令扩展至少一条（10分），乘除类指令至少一条（20分），转移指令至少一条（10分），访存指令一读一写两条（20分）

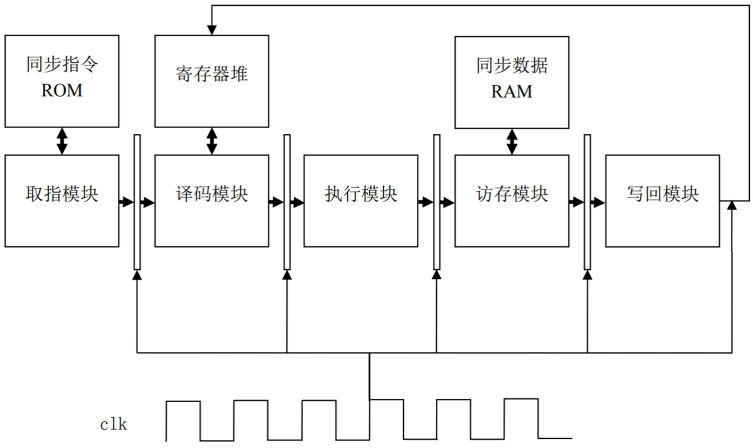
4、实验报告中针对bug修复过程、指令添加过程进行关键说明，并最终验证，验证需要有波形图或实验箱照片，并对波形图和实验箱照片进行分析解释。

5、实在无法完成某些指令扩展也没关系，把遇到的问题和失败的尝试写入报告，也会有相应分数。

三、实验原理图

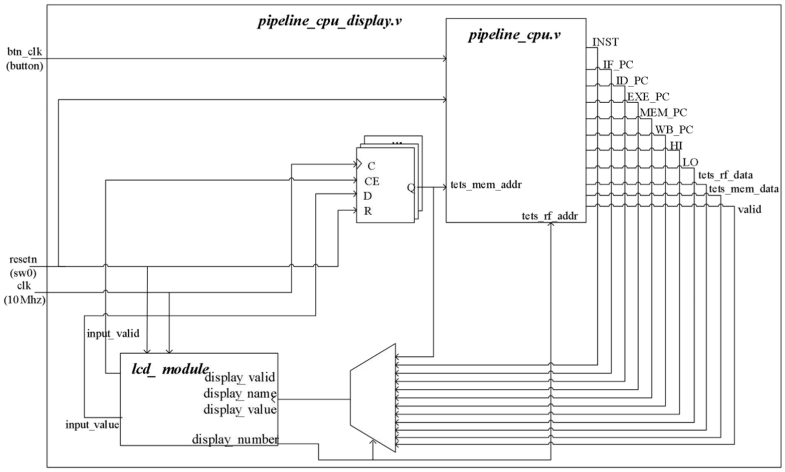


如图展示了单周期、多周期、5级流水的基本原理。我们可以很形象的看到，在单周期cpu中， 每一个时钟周期会完成一个指令的取指、译码、执行、访存、写回的操作；而在多周期cpu中，我们 在单周期基础上提高了时钟频率，并且在一个时钟周期内，只会完成取指、译码、执行、访存、写回这 五个操作中的一个,但并没有改善执行一条指令的时间，且存在资源闲置的问题；在5级流水cpu中， 我们添加流水线的逻辑，当第一条指令从取指级转换到下一级译码时，第二条指令进入取指级，当第 一条指令完成译码进入执行级时，第二条指令进入译码级，第三条指令进入取指级。



在流水CPU中，当在一时钟周期内完成了某一条指令的全部执行时（写回级完成），则有望在下 一时钟周期内完成下条指令的执行，因此依然相当于是一个周期完成一条指令，而时钟频率更高，因 此CPU可以运行的更快。

5 级流水cpu 的模块图如下所示：

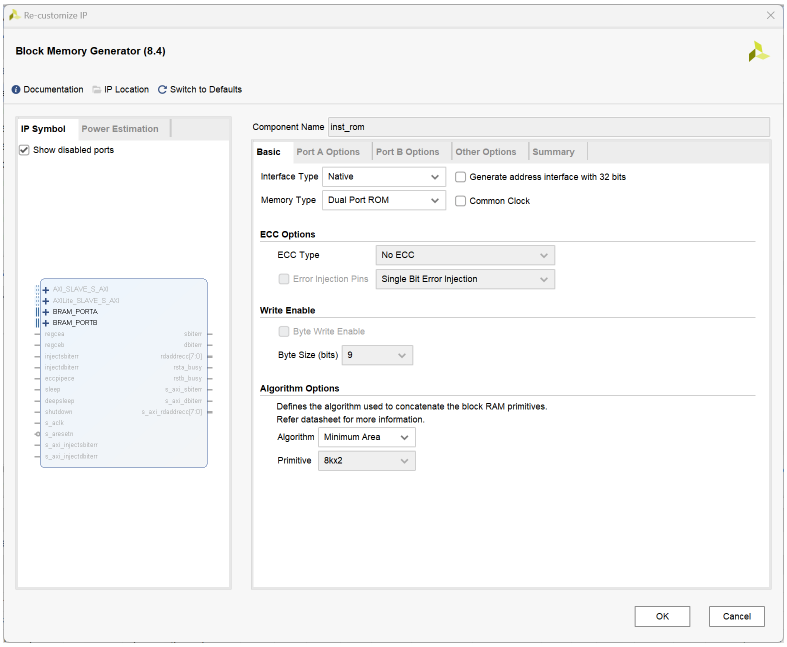


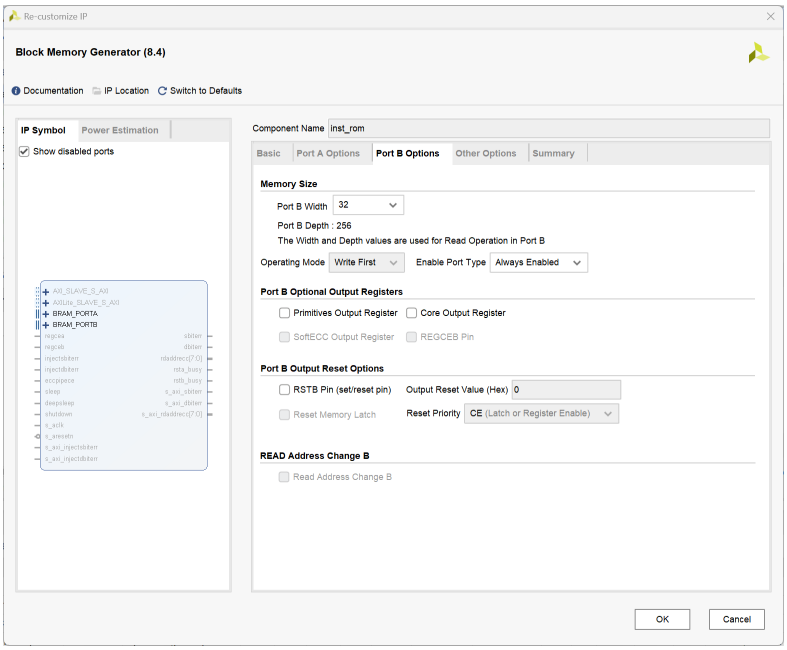
四、实验步骤

bug 分析与修复

首先我们先导入文件进行初步的测试，看看现有程序是否有bug，运行之后，我们发现，仿真中出现X的数据，而这个问题正是我们上次出现的问题，就是当输出寄存器（勾选Primitives Output Register）的时钟（clk_out） 与数据来源的时钟（clk_in）不同步时，直接传输数据可能导致亚稳态，导致仿真中数据变为X。 按之前的操作配置ip核，但注意不能勾选primitives output register 这个选项，否则会引入一个 时钟周期的延迟。

在这个基础上，如果利用多周期的ip核，依旧还有问题。我们查看给的inst_rom.v和data_ram.v文件发现，ip核发生了变化，需要按照下面两张图片重新配置。





问题分析

流水线冲突

在计算机体系结构中，流水线冲突指的是在指令流水线中由于某些原因导致指令不能按预期顺序连续执行，从而影响流水线性能。分析流水线冲突时，通常从三种类型出发：结构冲突、数据冲突、控制冲突。

结构冲突

结构冲突的核心成因是硬件资源共享不足，当两条或多条指令在同一时钟周期内争夺相同硬件资源（如内存、寄存器堆、算术逻辑单元 ALU 等）时，指令便无法按预期顺序连续执行，进而导致流水线性能下降。

在我们的程序中，采用了两个存储器，分别是指令存储器和数据存储器，这样就解决同时访问时的资源冲突。如果我们把指令和数据放在同一个存储器中，当我们在进行存储器的读取时，如果另一个操作需要进行取指操作，那么这样就会导致资源冲突。

数据冲突

数据冲突（Data Hazard）的核心成因是指令间存在寄存器访问的时序依赖，当前指令需读取的寄存器值尚未由前序指令完成写入。这类冲突主要发生在涉及寄存器读写的运算步骤中，可分为以下三种类型：

- **读后写冲突（RAW，Read After Write）**：指令 B 读取指令 A 写入的寄存器时，A 尚未完成写回。

- **写后读冲突（WAR，Write After Read）**：指令 B 写入指令 A 读取的寄存器时，A 尚未完成读取。
- **写后写冲突（WAW，Write After Write）**：两条指令写入同一寄存器时，写回顺序与指令顺序不一致。

其中最常见的就是读后写。

1. Load-Use 冲突（访存指令与后续指令的数据依赖）

```
lw $10, 0x1C($0)     # 将内存地址0x1C($0)的值读入寄存器$10（需经历5级流水：IF/ID/EX/MEM/WB）
bne $10, $5, offset # 在译码阶段（ID）需读取$10的值进行比较
```

冲突分析：

- `lw`指令的结果需在 **第 5 阶段（WB）** 才能写入$10寄存器，但`bne`指令在**第2阶段（ID）**就需读取$10 的值。此时读取的是旧值，导致判断错误。
- **本质**：访存指令的写回滞后于后续指令的读取需求，形成**跨阶段的数据依赖断层**。

解决方案：

- **插入空指令（nop）**：在两条指令间强制插入 2 条`nop`，延迟后续指令执行，等待`lw`完成写回。
- **流水线暂停（Bubble 插入）**：通过控制单元检测冲突，在 ID/EX 阶段暂停 2 个周期，避免读取旧值。
- **前递机制（Forwarding）**：将`lw`在访存阶段（MEM）计算出的值直接转发给`bne`的译码阶段（ID），无需等待写回（实际实现需结合硬件通路设计）。

2. ALU-ALU 冲突（算术指令间的数据依赖）

```
addiu $1, $0, 1      # 将1存入寄存器$1（写回在WB阶段）
sll $2, $1, 4        # 在执行阶段（EX）需读取$1的值进行左移
```

冲突分析：

- `addiu`的写回发生在**第 5 阶段（WB）**，而`sll`在 **第 3 阶段（EX）** 就需读取 $1 的值，导致读取早于写入。

解决方案：

- **前递机制**：若存在前递通路，可将`addiu`在执行阶段（EX）计算出的 $1 值，直接从**EX/MEM 寄存器**或**MEM/WB 寄存器**转发至`sll`的 EX 阶段输入端，避免等待。
- **插入 nop**：若无前递机制，需在两条指令间插入 1 条`nop`，确保`sll`执行时 $1 已完成写回。

3. ALU-Branch 冲突（算术指令与分支指令的数据依赖）

```
xor $8, $7, $6      # 计算$8的值（写回在WB阶段）
beq $8, $3, label   # 在译码阶段（ID）需读取$8的值进行相等判断
```

冲突分析：

- `xor`的写回在**第 5 阶段（WB）**，但`beq`在 **第 2 阶段（ID）** 就需读取 $8 的值。若未启用分支前递，判断时使用的是旧值，可能导致跳转逻辑错误。

解决方案：

- **分支前递（Branch Forwarding）**：专门设计硬件通路，将`xor`在执行阶段（EX）计算出的 $8 值直接转发至`beq`的 ID 阶段，用于条件判断。
- **流水线暂停**：若无前递机制，需插入气泡（Bubble）或延迟分支执行，等待`xor`完成写回。

控制冲突

控制冲突（Control Hazard）的核心矛盾源于**分支跳转指令的决策延迟与流水线预取机制的不匹配**。当 CPU 在取指阶段（IF）需要确定下一条指令地址时，分支指令的跳转决策（是否跳转、跳转到何处）尚未在译码阶段（ID）完成，导致可能预取错误的指令，进而引发流水线清空或错误执行。

在五级流水线 CPU 中，控制冲突主要发生在遇到分支跳转类指令的时候。

1. 条件分支指令（如 BGTZ、BLTZ、BNE）

```
bgtz $13, offset  # 若$13 > 0则跳转（决策在ID阶段完成）
sllv $14, $7, $6   # IF阶段已预取该指令，若跳转则需清空
```

冲突过程：

- **IF 阶段**：CPU 按顺序预取下一条指令`sllv`。
- **ID 阶段**：解析`bgtz`指令，判断是否跳转。若决定跳转，已预取的`sllv`指令需从流水线中清除（Flush），导致性能损失。

核心问题：分支决策晚于预取操作，造成**预取指令的无效化**。

2. 无条件跳转指令（如 J）

```
j 0x00000034  # 无条件跳转（跳转地址在ID阶段计算）
addiu $2, $2, 4  # IF阶段预取的顺序指令，跳转后需清空
```

冲突特点：

- 即使是无条件跳转，跳转目标地址的计算（如 PC + 偏移量）仍需在 ID 阶段完成，IF 阶段仍会预取顺序指令，导致无效指令进入流水线。

解决方案

1. 编译器层面：分支延迟槽（Branch Delay Slot）

原理：在分支指令后强制插入一个**延迟槽指令**，无论分支是否跳转，该指令都会被执行。通过编译器选择一条**与分支条件无关的安全指令**填充延迟槽，避免流水线气泡（Bubble）。

- ```
  bgtz $13, offset  # 分支指令
  sllv $14, $7, $6   # 延迟槽指令（无论是否跳转，均执行）
  ```

- **优势**：硬件无需修改，仅通过软件调度提升效率。
- **局限**：需编译器精准识别安全指令，若无法找到则只能插入空操作（nop）。

2. 硬件层面：分支预测技术

通过硬件预测分支是否跳转及目标地址，减少预取错误的概率：

- 静态分支预测：
  - **恒不跳转（Always Not Taken）**：默认执行顺序指令，适用于分支不跳转概率高的场景（如循环体结尾）。
  - **恒跳转（Always Taken）**：默认跳转，适用于条件成立概率高的分支。
  ```

- 动态分支预测：
    - **基于历史记录的预测**：使用分支历史表（BHT）记录过去的跳转行为，预测未来走向（如两位饱和计数器算法）。
    - **目标地址缓存**：用分支目标缓冲器（BTB）存储已执行过的分支目标地址，直接用于预取。
- **优势**：硬件自动优化，无需编译器干预，预测准确率可达 80% 以上。
- **局限**：增加硬件复杂度（如缓存、状态机），存在预测错误时仍需清空流水线。

指令添加

运算指令 ADDI

不同于之前多周期的指令添加，我在这里选择添加了一条立即数加法指令。

代码修改

decode.v

```
//立即数加法
wire inst_ADDI;
assign inst_ADDI  = (op == 6'b100001);
```

添加指令，以及操作码定义

```
assign inst_add = inst_ADDU | inst_ADDIU | inst_load
                | inst_ADDI | inst_store | inst_j_link;
 // 做加法
//....
assign inst_wdest_rt = inst_imm_zero | inst_ADDIU | inst_SLTI |
inst_ADDI
                       | inst_SLTIU | inst_load | inst_MFC0;
//....
assign inst_imm_sign = inst_ADDIU | inst_SLTI | inst_SLTIU |
inst_ADDI
                       | inst_load | inst_store;
```

将这条指令添加到add调用中。

转移指令 BGTZAL

转移指令比较简单，只需要修改decode模块即可。

代码修改

decode.v

```
// 新增转移指令
wire inst_BLTZAL;
assign inst_BLTZAL= (op == 6'b000001) & (rt==5'd16); //小于0跳转并链接
```

添加指令，以及操作码定义

```
assign inst_j_link = inst_JAL | inst_JALR | inst_BLTZAL;
assign inst_jbr = inst_J    | inst_JAL  | inst_jr
                  | inst_BEQ  | inst_BNE  | inst_BGEZ
                  | inst_BGTZ | inst_BLEZ | inst_BLTZ
                  | inst_BLTZAL;      // 添加BLTZAL到跳转分支指令
//....
assign inst_wdest_31 = inst_JAL | inst_BLTZAL;  // 添加BLTZAL到写31号
寄存器的指令
//....
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
                  | inst_BLTZAL & rs_ltz;                    //
添加BLTZAL条件判断
```

将这条指令添加到j跳转调用中。

乘除指令 MULTU

我们基于原来的有符号乘法MULT扩展我们的指令MULTU。二者区别在于符号位的判断，这里需要多加入标记。

代码修改

multply.v

```
module multiply(                // 乘法器
    input         clk,          // 时钟
    input         mult_begin,   // 乘法开始信号
    input  [31:0] mult_op1,     // 乘法源操作数1
    input  [31:0] mult_op2,     // 乘法源操作数2
    input         signed_mode,  // 是否为有符号乘法
    output [63:0] product,      // 乘积
    output        mult_end      // 乘法结束信号
);
//....
//仅在有符号模式下考虑符号位
assign op1_sign = signed_mode & mult_op1[31];
assign op2_sign = signed_mode & mult_op2[31];
//....
//若乘法结果为负数，则需要对结果取反+1
assign product = (signed_mode && product_sign) ? (~product_temp+1)
: product_temp;
//....
```

这里在乘法器的输入上增肌一位位宽，来判断是否是有符号乘法，仅在有符号模式下考虑符号位。

中间计算过程不变，在最后输出结果的过程进一步进行判断。

decode.v

```
//新增乘法指令
wire inst_MULTU;
assign inst_MULTU = op_zero & (rd==5'd0) & sa_zero & (funct ==
6'b011001); //无符号乘法
```

添加指令，以及操作码定义

```
wire signed_mode;
assign multiply = inst_MULT | inst_MULTU;
assign signed_mode = inst_MULT;
```

加入符号模式控制信号

```
assign ID_EXE_bus = {multiply,signed_mode,mthi,mtlo,
    //EXE需用的信息,新增
                     alu_control,alu_operand1,alu_operand2,//EXE需用
的信息
                     mem_control,store_data,               //MEM需用
的信号
                     mfhi,mflo,                            //WB需用
的信号,新增
                     mtc0,mfc0,cp0r_addr,syscall,eret,     //WB需用
的信号,新增
                     rf_wen, rf_wdest,                     //WB需用
的信号
                     pc};
```

将控制信号加入到总线中

exe.v

```
wire signed_mode;          //有符号乘法标志\
//....
assign {multiply,
            signed_mode,       // 新增有符号乘法标志
            mthi,
            mtlo,
            alu_control,
            alu_operand1,
            alu_operand2,
            mem_control,
            store_data,
            mfhi,
            mflo,
            mtc0,
            mfc0,
            cp0r_addr,
            syscall,
            eret,
            rf_wen,
            rf_wdest,
            pc          } = ID_EXE_bus_r;
```

```
//....
multiply multiply_module (
        .clk       (clk       ),
        .mult_begin(mult_begin  ),
        .mult_op1  (alu_operand1),
        .mult_op2  (alu_operand2),
        .signed_mode(signed_mode),
        .product   (product   ),
        .mult_end  (mult_end  )
    );
```

在exe模块需要对multiply模块进行调用，所以需要进行添加相应的控制信号。将signed_mode信号添加到总线中。

访存指令 LH和SH

访存指令相对较为麻烦，需要修改大量的逻辑。

代码修改

decode.v

```
// 新增访存指令
wire inst_LH;
assign inst_LH    = (op == 6'b100101);
wire inst_SH;
assign inst_SH    = (op == 6'b101001);
```

添加指令，以及操作码定义

```
assign inst_load  = inst_LW | inst_LB | inst_LBU | inst_LH;  //
load指令
assign inst_store = inst_SW | inst_SB | inst_SH;             //
store指令
//....
wire ls_half;    // half word访问标志
assign ls_half = inst_LH | inst_SH;
//....
assign mem_control = {inst_load,
                      inst_store,
                      ls_word,
                      ls_half,   // 新增ls_half控制信号
                      lb_sign };
```

将新增的指令添加到mem控制信号中

mem.v

```
wire ls_half;    // 新增half word标志
assign {inst_load,inst_store,ls_word,ls_half,lb_sign} =
mem_control;
//....
wire [15:0] load_half;    // 新增half word临时变量
```

添加指令信号

```
//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 if (ls_half)                  // 新增half word处理
        begin
            case(dm_addr[1:0])
                2'b00: dm_wen <= 4'b0011;  // 低半字
                2'b10: dm_wen <= 4'b1100;  // 高半字
                default: dm_wen <= 4'b0000;
```

```
            endcase
        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 @(*) begin
    if (MEM_valid && inst_store) begin
        if (ls_word) begin
            dm_wdata = store_data; // 全32位写入
        end
        else if (ls_half) begin
            // 半字写入：按地址选择低/高16位
            if (dm_addr[1] == 1'b0)
                dm_wdata = {16'd0, store_data[15:0]};
            else
                dm_wdata = {store_data[15:0], 16'd0};
        end
        else begin
            // 字节写入：按地址位置选择字节
            case (dm_addr[1:0])
                2'b00: dm_wdata = {24'd0, store_data[7:0]};
                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};
            endcase
        end
```

```
    end
    else begin
        dm_wdata = 32'd0;
    end
end
```

修改写使能和写数据

```
wire [15:0] load_half;    // 新增half word临时变量
// 根据地址获取字节和半字的符号位
assign load_sign = ls_half ? (dm_addr[1] ? dm_rdata[31] :
dm_rdata[15]) :  // 半字的符号位
                        (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];

// 获取半字数据
assign load_half = dm_addr[1] ? dm_rdata[31:16] : dm_rdata[15:0];
// 获取字节数据
assign load_byte = (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];

// 最终的load结果
assign load_result = ls_word ? dm_rdata :
 // 字
                     ls_half ? {{16{lb_sign & load_sign}},
// 半字(符号扩展)
                               load_half} :
                     {{24{lb_sign & load_sign}}, load_byte};
// 字节(符号扩展)
```

修改load读数据的操作，加入load_half

统一修改

最后是检查扩展指令后，整个程序传递的位宽。

pipeline_cpu.v

```
wire [169:0] ID_EXE_bus;  // ID->EXE级总线
wire [154:0] EXE_MEM_bus; // EXE->MEM级总线
//锁存以上总线信号
reg [169:0] ID_EXE_bus_r;
reg [154:0] EXE_MEM_bus_r;
```

men.v

```
input      [154:0] EXE_MEM_bus_r,// EXE->MEM总线
//....
wire [4 :0] mem_control;  //MEM需要使用的控制信号
```

exe.v

```
input      [169:0] ID_EXE_bus_r,// ID->EXE总线
output     [154:0] EXE_MEM_bus, // EXE->MEM总线
//....
wire [4:0] mem_control;  //MEM需要使用的控制信号
```

decode.v

```
output     [169:0] ID_EXE_bus,  // ID->EXE总线
```

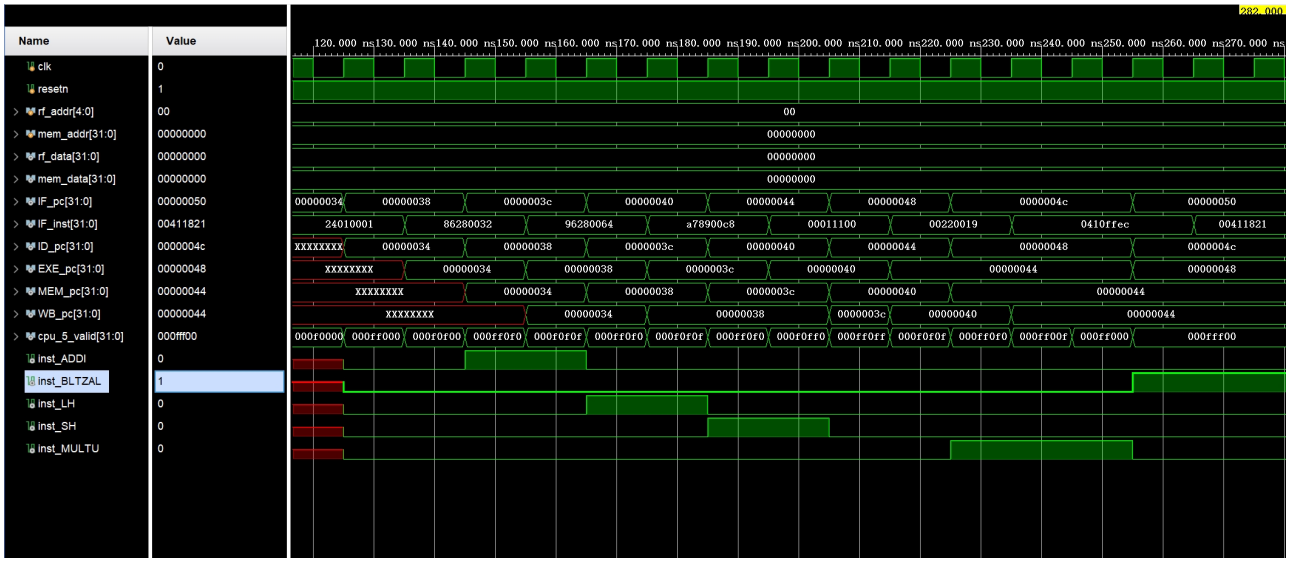
由于加入了signed_mode，以及访存指令，所以需要扩展ID_EXE和EXE_MEM的总线位宽。

五、实验结果分析

```
00100100000000010000000000000001
10000110001010000000000000110010
10010110001010000000000001100100
10100111100010010000000011001000
00000000000000010001000100000000
00000000010001000000000000011001
00000100000100001111111111101100
```

```
24010001
86280032
96280064
A78900C8
00011100
00220019
0410FFEC
```

为了尽早查看指令是否添加成功，我在开始运行时添加了指令，同时为了保证添加的指令不会对原程序造成影响，我在新添加的指令中间加入了原指令。



通过观察仿真图我们发现，IF_inst为24010001时是第一条指令ADDU，之后跳转到我们添加的指令86280032，96280064，分别调用了LH和SH，成功实现了指令的添加。

之后是我们为了验证原指令没有被破坏插入的一条原指令00011100，正常执行，之后再转到我们新添加的指令00220019，0410FFEC，分别调用了MULTU和BLTZAL，成功执行。

综上所述，表明我们的指令添加成功。

六、总结感想

本次实验通过设计五级流水线 CPU，深入理解了计算机组成原理，在硬件设计与指令扩展中收获颇丰：

1. 流水线冲突的核心收获

- **结构冲突**：指令 / 数据存储器分离可避免资源竞争；
- **数据冲突**：前递机制（Forwarding）能解决 RAW 冒险（如 Load-Use 冲突）；
- **控制冲突**：分支延迟槽和预测技术可减少流水线清空。

2. 指令扩展实践

- **运算指令（ADDI）**：实现立即数加法，修改译码与执行模块；
- **转移指令（BGTZAL）**：添加小于 0 跳转并保存返回地址；
- **乘除指令（MULTU）**：实现无符号乘法，区分符号位处理；
- **访存指令（LH/SH）**：完成半字读写，处理地址对齐与符号扩展。

3. 关键问题与解决

- **bug**：IP 核配置不勾选输出寄存器，解决时钟不同步导致的数据 X 态；
- **总线扩展**：新增控制信号后同步 ID_EXE/EXE_MEM 总线位宽，避免信号截断。

4. 实验感悟

流水线设计的本质是平衡并行效率与资源冲突，指令扩展需全流程适配（译码→执行→访存→写回）。通过 Verilog 编程将理论转化为硬件，深刻体会到计算机体系结构是性能、复杂度与成本的权衡艺术。