计算机体系结构实验课程期末综合实验报告

实验名称	优化 CPU 系统			班级	李雨森班
学生姓名	姚知言	学号	2211290	指导老师	董前琨
实验地点	实验楼 A306		实验时间	2024 年 11—12 月	

1、 实验目的

加深对计算机组成原理和体系结构理论知识的理解。

培养对 CPU 设计的兴趣，在理解现有 CPU 架构的基础上，引发对体系结构的思考和创新。

培养创新思维能力，并通过实践验证新想法。

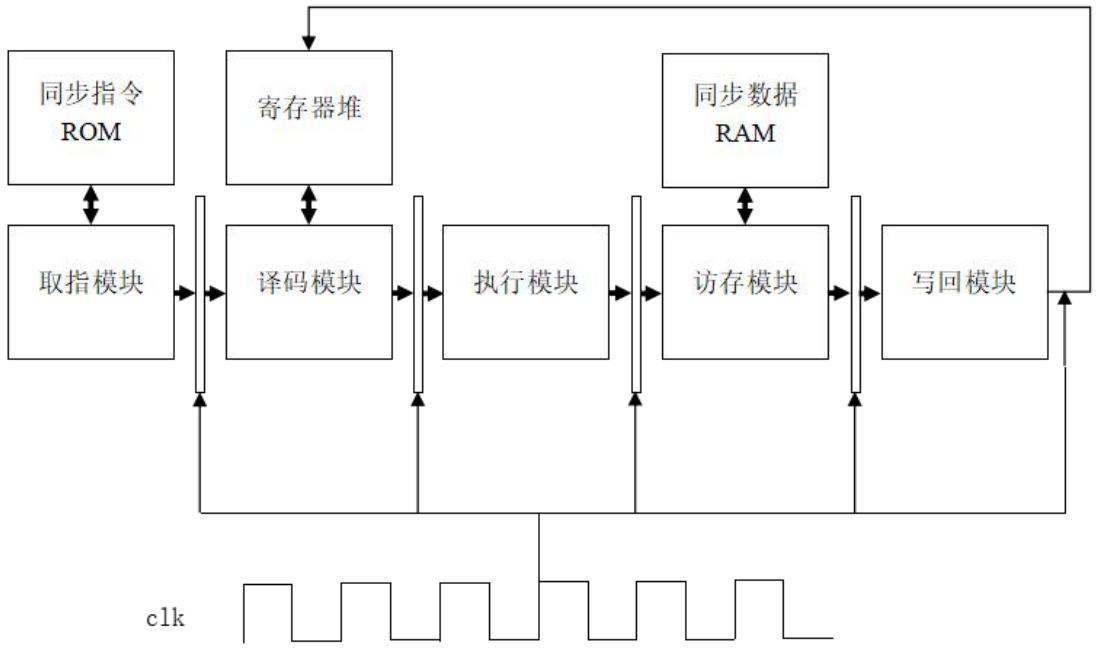
2、 实验内容说明

（1）分析静态 5 级流水 CPU 中的流水线阻塞情况，包括数据相关、控制相关、结构相关等，使用前递技术优化流水线设计，尽可能减少流水线阻塞情况。

（2）扩展 MIPS 指令集，针对 R 型，I 型，J 型指令均进行扩展，并验证结果。

（3）分析 MEM 阶段的执行流程，设计 cache，减少 MEM 阶段对内存访问的延迟。

3、 实验原理图



4、 实验步骤

（1）前递技术优化流水线设计

(a)pipeline_cpu.v

```
decode ID_module(                    // 译码级
    .rf_rs_value    (rf_rs_value    ),   // I, 32
    .rf_rt_value    (rf_rt_value    ),   // I, 32
    ...
    //forwarding
    .EXE_f_result   (EXE_f_result ),
    .EXE_f_valid    (EXE_f_valid),
    .EXE_f_wdest    (EXE_f_wdest),
```

```
        .MEM_f_result    (MEM_f_result ),
        .MEM_f_valid     (MEM_f_valid),
        .MEM_f_wdest     (MEM_f_wdest)
    );
```

将寄存器传入的 rs/rt value 改为 rf_rs/rt_value，在后续选择把 rf 的结果或者前递结果赋值。

在 decode 阶段增加 EX 和 MEM 的前递输入信号，valid 表示前递有效的使能信号，wdest 表示前递的寄存器号，result 表示前递的结果。

```
    wire [31:0] rf_rs_value;
    wire [31:0] rf_rt_value;
    regfile rf_module(              //  寄存器堆模块
        ...
        .rdata1 (rf_rs_value ),    // O, 32
        .rdata2 (rf_rt_value ),    // O, 32
        ..
    );
```

同步调整文件中的 wire 命名和 rf 端口绑定。

```
    exe EXE_module(                          //  执行级
        ...
        //forwarding
        .EXE_result    (EXE_result)
    );
    mem MEM_module(                              //  访存级
        ...
        //forwarding
        .MEM_result     (MEM_result)
    );
```

在 exe 阶段和 mem 阶段增加结果的输出信号。（虽然这一结果也包括在总线中，但这样实现较为直观）

```
    //forwarding
    wire [31:0] EXE_result;
    wire [31:0] MEM_result;
    reg [31:0] EXE_f_result,MEM_f_result;
    reg [4:0] EXE_f_wdest,MEM_f_wdest;
    reg EXE_f_valid,MEM_f_valid;

    //ID 与 EXE、MEM、WB 交互
    wire [ 4:0] EXE_wdest;
    wire [ 4:0] MEM_wdest;
    wire [ 4:0] WB_wdest;
```

设计新的 wire 读取前递结果，设计新的 reg 锁存结果。将 wdest 的 wire 定义位置提前，以便使用。

```
    always @(posedge clk)
    begin
```

```
        //if (!resetn || cancel)
        //begin
        //      MEM_valid <= 1'b0;
        //end
        //else if (MEM_allow_in)
        //begin
        //      MEM_valid <= EXE_over;
        //end
        if(resetn && !cancel && MEM_allow_in && EXE_over) begin
            MEM_valid <= 1'b1;
            EXE_f_result <= EXE_result;
            EXE_f_valid <= 1'b1;
            EXE_f_wdest <= EXE_wdest;
        end
        else begin
            MEM_valid <= 1'b0;
            EXE_f_result <= 32'b0;
            EXE_f_valid <= 1'b0;
            EXE_f_wdest <= 5'b0;
        end
    end
```

更改 MEM 的 valid 赋值语句块，保留原 valid 赋值不变，增加前递结果的锁存。

```
    always @(posedge clk)
    begin
        //if (!resetn || cancel)
        //begin
        //      WB_valid <= 1'b0;
        //end
        //else if (WB_allow_in)
        //begin
        //      WB_valid <= MEM_over;
        //end
        if(resetn && !cancel && WB_allow_in && MEM_over) begin
            WB_valid <= 1'b1;
            MEM_f_result <= MEM_result;
            MEM_f_valid <= 1'b1;
            MEM_f_wdest <= MEM_wdest;
        end
        else begin
            WB_valid <= 1'b0;
            MEM_f_result <= 32'b0;
            MEM_f_valid <= 1'b0;
            MEM_f_wdest <= 5'b0;
        end
```

```
end
```

WB 类似。

(b)decode.v

```
module decode(                              // 译码级
    ...
    input         [ 31:0] rf_rs_value,      // 第一源操作数值(rf)
    input         [ 31:0] rf_rt_value,      // 第二源操作数值(rf)
    ...
    //forwarding
    input         [31:0] EXE_f_result,
    input         [4:0] EXE_f_wdest,
    input                EXE_f_valid,
    input         [ 31:0] MEM_f_result,
    input         [4:0] MEM_f_wdest,
    input                MEM_f_valid
);
```

同步进行输入端口的修改。

```
wire [31:0] rs_value,rt_value;
assign rs_wait = ~inst_no_rs & (rs!=5'd0)
                 & ( (rs==EXE_wdest) | (rs==MEM_wdest) | (rs==WB_wdest) )
                 & (~EXE_f_valid | ~(rs==EXE_f_wdest))
                 & (~MEM_f_valid | ~(rs==MEM_f_wdest));
assign rt_wait = ~inst_no_rt & (rt!=5'd0)
                 & ( (rt==EXE_wdest) | (rt==MEM_wdest) | (rt==WB_wdest) )
                 & (~EXE_f_valid | ~(rt==EXE_f_wdest))
                 & (~MEM_f_valid | ~(rt==MEM_f_wdest));

assign rs_value = (rs==5'd0) ? 32'd0 :
                   (EXE_f_valid    & (rs==EXE_f_wdest)) ? EXE_f_result :
                  (MEM_f_valid    & (rs==MEM_f_wdest)) ? MEM_f_result :
                  rf_rs_value;
assign rt_value =(rt==5'd0) ? 32'd0 :
                  (EXE_f_valid    & (rt==EXE_f_wdest)) ? EXE_f_result :
                  (MEM_f_valid    & (rt==MEM_f_wdest)) ? MEM_f_result :
                  rf_rt_value;
```

更改 wait 条件，增加前递找不到结果才等待。

定义 rs/rt_value，若前递可以获得结果则通过前递结果赋值，否则读取寄存器的值。

注意：这里对于 rs=0 or rt=0 的特殊判定很重要！因为在不需要写回的指令中会将对应的 EXE/MEM 的 wdest 清零，若此时 rs/rt 恰好为 0 号寄存器，会导致错误的匹配。但 0 号寄存器的数值应该一直保持为 0，所以也不需要去特意修改 wait，直接把 32’d0 赋值就可以。

(c)exe.v

```
module exe(                                 // 执行级
    ...
    //forwarding
```

```
    output        [31:0] EXE_result
);
```

还需要删除原有的 exe_result 定义，将后续 exe_result 更改为 EXE_result。

(d)mem.v

```
module mem(                                    // 访存级
    ...
    //forwarding
    output        [ 31:0] MEM_result
);
```

还需要删除原有的 mem_result 定义，将后续 mem_result 更改为 MEM_result。

（2）指令添加设计

在本实验中实现了以下五条指令。

a.R 型指令：

nand，nand rd,rs,rt：rd=^(rs&rt)

Bit	31-26	25-21	20-16	15-11	10-6	5-0
R-type	op	rs	rt	rd	shamt	func
nand	000000	rs	rt	rd	00000	111111

b.I 型指令：

subiu， rt <- rs - (zero-extend)immediate；

nori， rt <- not(rs | (zero-extend)immediate)。

Bit	31-26	25-21	20-16	15-0
I-type	op	rs	rt	immediate
subiu	111101	rs	rt	immediate
nori	111110	rs	rt	immediate

c.J 型指令：

jc，跳转到 address 置寄存器 31 为 0，reg[31]=0,PC={next_pc[31:28],target <<2}。

Bit	31-26	25-0
R-type	op	address
jc	111111	address

以下是代码实现：

(a)pipeline_cpu.v

```
wire [168:0] ID_EXE_bus;
reg [168:0] ID_EXE_bus_r;
```

因为 ID->EXE 的标记位增加，需要调整总线位宽。

(b)decode.v

```
module decode(                                 // 译码级
    ...
    output        [168:0] ID_EXE_bus,    // ID->EXE 总线
    ...
);
```

需要调整总线位宽，因为增加了一个标记位 INST_JC，同时 ALU 的控制信号也多了一位。

```
wire inst_NAND, inst_SUBIU, inst_NORI, inst_JC;
assign inst_NAND = op_zero & sa_zero       & (funct == 6'b111111);
assign inst_SUBIU = (op == 6'b111101);
```

```
assign inst_NORI = (op == 6'b111110);
assign inst_JC = (op == 6'b111111);
```

增加的指令以及指令译码。

```
 assign inst_jbr = inst_J | inst_JAL   | inst_jr
                   | inst_BEQ   | inst_BNE   | inst_BGEZ
                   | inst_BGTZ | inst_BLEZ | inst_BLTZ
                   | inst_JC;
```

跳转指令新增 JC。

```
wire inst_nand;
assign inst_nand = inst_NAND;
```

新增 nand 信号，用于 ALU 指令分类。

```
assign inst_imm_zero = inst_ANDI   | inst_LUI   | inst_ORI | inst_XORI | inst_NORI |
inst_SUBIU;
```

NORI 和 SUBIU 都是需要向前补充 0 来进行无符号扩展的。

注：原代码中 ADDIU 在有符号扩展分类中，不知道是设计错误还是故意为之，个人倾向于认为这是一处错误，但在此我并没有进行修改。

```
assign inst_wdest_rt = inst_imm_zero | inst_ADDIU | inst_SLTI
                       | inst_SLTIU | inst_load | inst_MFC0;
assign inst_wdest_31 = inst_JAL | inst_JC;
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
                       | inst_MFHI | inst_MFLO | inst_NAND;
```

填写目的寄存器信号，NAND 的目的寄存器是 rd，JC 的目的寄存器是 31，另外两条因为包括在 inst_imm_zero 中了，不需要再进行添加。

```
assign inst_no_rt = inst_ADDIU | inst_SLTI | inst_SLTIU
                    | inst_BGEZ   | inst_load | inst_imm_zero
                    | inst_J      | inst_JAL   | inst_MFC0
                    | inst_SYSCALL | inst_JC;
```

将 JC 加入指令不包括 rt 的集合，以免错误读取。SUBIU 和 NORI 同样是因为包括在 inst_imm_zero 中了，不需要再进行添加，而 NAND 的 rs 和 rt 位置都存储寄存器地址，不应该在此添加。

```
assign j_taken = inst_J | inst_JAL | inst_jr | inst_JC;
```

将 JC 加入无条件跳转的信号中。

```
wire [12:0] alu_control;
assign alu_control = {inst_nand,
                      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
                    };
```

添加 nand 到 alu_control 中，并更改其位宽。

```
assign ID_EXE_bus = {multiply,mthi,mtlo,
                     alu_control,inst_JC,alu_operand1,alu_operand2,
                     mem_control,store_data,
                     mfhi,mflo,
                     mtc0,mfc0,cp0r_addr,syscall,eret,
                     rf_wen, rf_wdest,
                     pc};
```

在总线中增加 jc 的信号，以便后续正确修改 31 寄存器，与 jal 区分开来。

(c)exe.v

```
module exe(
    ...
    input          [168:0] ID_EXE_bus_r,// ID->EXE 总线
    ...
);
...
wire [12:0] alu_control;
...
assign {multiply,
        mthi,
        mtlo,
        alu_control,
        jc,
        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;
```

与 decode 对应，更改总线位宽和信息对应。

```
assign EXE_result = mthi         ? alu_operand1 :
                    mtc0         ? alu_operand2 :
                    multiply ? product[63:32] :
                    jc ? 32'd0:
                    alu_result;
```

新增如果是 JC 指令，需要将 result 清零。

(d)alu.v

```
module alu(
    input    [12:0] alu_control,    // ALU 控制信号
    ...
 );
```

控制信号位宽对应调整。

```
wire alu_nand;
assign alu_nand = alu_control[12];
```

为 nand 增加并绑定新的控制信号。

```
wire [31:0] nand_result;
assign nand_result = ~and_result;
```

添加为 nand 计算结果。

```
assign alu_result = alu_nand                ? nand_result:
                    (alu_add|alu_sub) ? add_sub_result[31:0] :
                    alu_slt                 ? slt_result :
                    ...
                    32'd0;
```

在选择结果输出中增加 nand。

（3）cache 实现

在本实验中，我设计了一个较为简单的 cache，其技术细节为：

共有 32 块，每块 4 字节，采用直相联和写穿透的方式实现。

同时，仅在写指令的时候同步对 cache 进行刷新，对于读指令时的 cache miss 情况，暂时不做 cache 更新。

以下是代码实现：

(a)cache.v[新增]

```
module cache(
    input [31:0]addr,              //输入地址（读/写）
    input wen,                     //写使能
    input [31:0] w_data,           //写数据
    output now_valid,              //当前 cacheline 的有效位
    output [24:0] now_label,       //当前 cacheline 存储的地址
    output [31:0] r_data           //当前 cacheline 存储的数据
    );
```

//cacheline 定义：最高位表示 valid,中间 25 位表示地址标记（32-5-2），末 32 位表示存储的数据

```
reg [57:0] cacheline [31:0];//[57] valid,[56:32]addr,[31:0] data
```

//cache 初始化为 0

```
integer i;
```

```
initial begin
    for (i = 0; i < 32; i = i + 1) begin
        cacheline[i] = 57'b0;
    end
end
wire [4:0] now_line;            //当前输入对应的 cacheline
wire [24:0] write_label;        //对应的标记位
assign now_line = addr[6:2];
assign now_valid = cacheline[now_line][57];
assign write_label = addr[31:7];
assign now_label = cacheline[now_line][56:32];
assign r_data = cacheline[now_line][31:0];
always @(*) begin
    if(wen) begin
        cacheline[now_line] ={1'b1,write_label,w_data};
    end
end
//always 块，在写指令时更新 cache
endmodule
```

其实在一开始是打算把 hit 是否命中的判定也放在这个里面的，但是后续调试过程中，如果使用 reg 赋值，会导致多耽误一个周期（也就是并没有优化时钟周期，显而易见）。使用 wire 调试的过程中会导致仿真卡死（并没有报错），原因可能是赋值嵌套。最后选择将是否命中的判定拆到 mem.v 中进行。

(b)mem.v

```
wire wen;
wire [24:0] now_label;
assign wen = MEM_allow_in && inst_store;
```

写使能和 now_label 的定义，写使能初始化为 MEM 的第一拍(allow_in)和指令写。（其实这里也曾经考虑过把读 cache miss 的情况也加进去，但是最终没能成功）

```
wire    load_sign;
wire [31:0] load_result;
wire [31:0] mem_rdata,cache_rdata;
wire now_valid;
wire hit = inst_load && now_valid && (now_label == dm_addr[31:7]);
```

定义 hit 判定条件：读指令，当前 cacheline 的有效位为 1，且地址标签匹配。再定义一些与 cache 交互的端口。

```
assign mem_rdata = (hit) ? cache_rdata :dm_rdata;
assign load_sign = (dm_addr[1:0]==2'd0) ? mem_rdata[ 7] :
                   (dm_addr[1:0]==2'd1) ? mem_rdata[15] :
                   (dm_addr[1:0]==2'd2) ? mem_rdata[23] : mem_rdata[31] ;
assign load_result[7:0] = (dm_addr[1:0]==2'd0) ? mem_rdata[ 7:0 ] :
                          (dm_addr[1:0]==2'd1) ? mem_rdata[15:8 ] :
                          (dm_addr[1:0]==2'd2) ? mem_rdata[23:16] :
                          mem_rdata[31:24] ;
```

assign load_result[31:8]= ls_word ? mem_rdata[31:8] : {24{lb_sign & load_sign}};

新定义 mem_rdata，在 cache 命中的时候使用 cache 存储的数据，否则使用内存读取的数据。在后续 load_sign 和 load_result 中，把赋值的 dm_rdata 替换为 mem_rdata。

assign MEM_over = (inst_load && !hit) ? MEM_valid_r : MEM_valid;

MEM_over 判定更新，现在只有读指令 cache 未命中的时候才需要两个周期了，其他都只需要一个周期。

```
cache cache_module(
    .addr    (dm_addr ),
    .wen         (wen),
    .w_data (dm_wdata),
    .now_valid      (now_valid),
    .now_label      (now_label),
    .r_data(cache_rdata)
 );
```

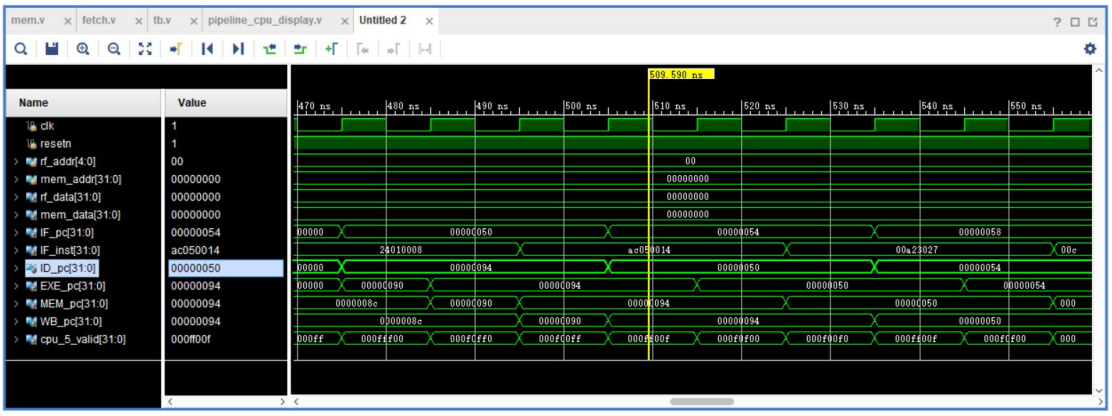
cache 模块的调用，在上一部分已经进行了解释。

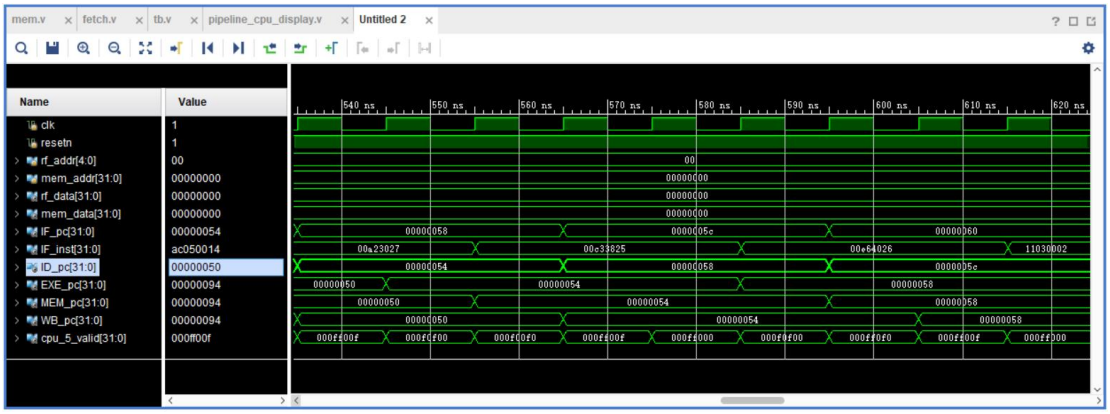
5、 实验结果分析

（1）前递技术优化对比

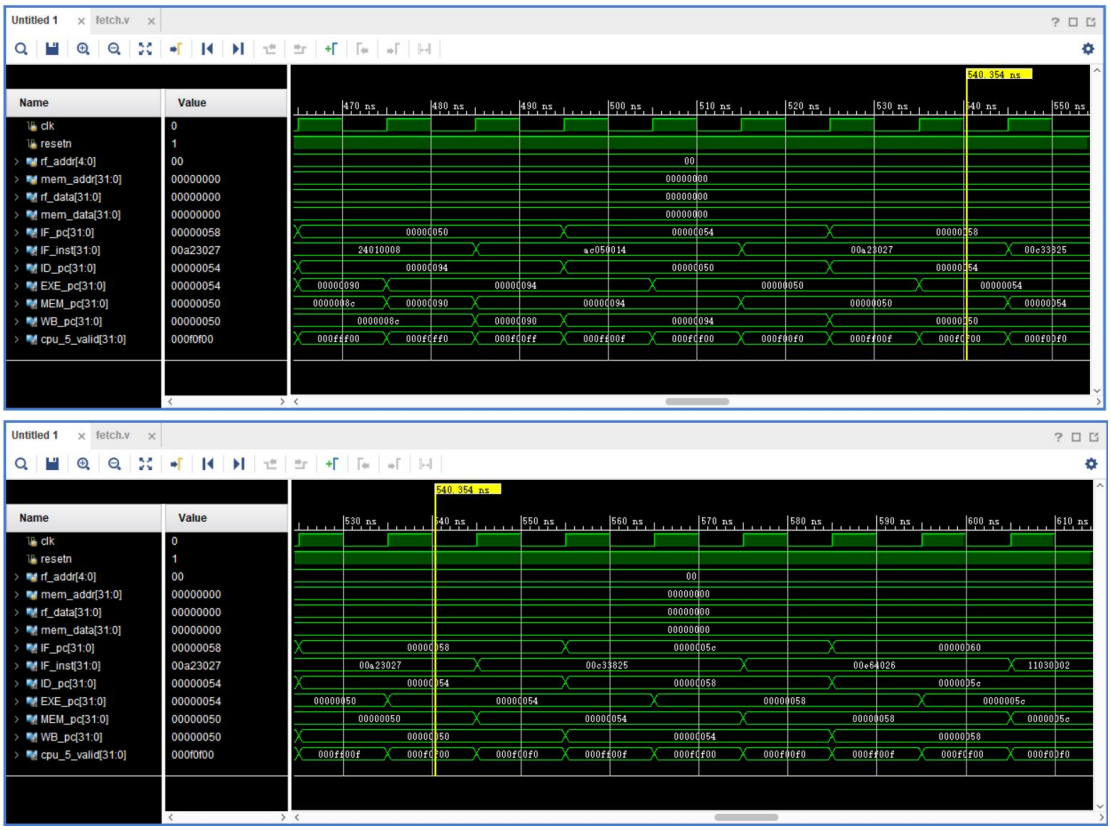
搭建前递前后，对 50H-60H 这五条指令的分析：

搭建前的仿真结果如下（这一结果也在上一次报告中展示）：

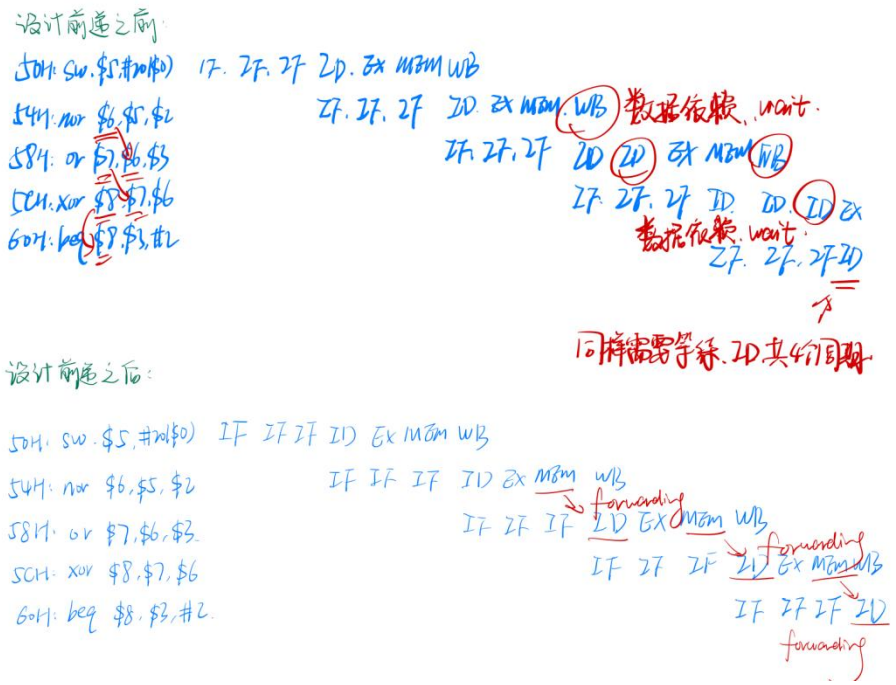




搭建后的仿真结果如下：



分析如下：



设计前递之前，如果是连续的数据依赖，每一条指令的 ID 都会多一个周期（因为需要等待上一条指令的 WB），从而造成 CPU 执行效率低下。

设计前递之后，由于前递结果的设计所有指令（若不涉及分支跳转，也不涉及读内存）均可在 7 周期内完成执行。在不涉及分支跳转下，平均每条指令的运行时间为耗时最长的 IF 阶段的 3 周期。前递的设计在数据依赖的情况下可以很好的提升 CPU 的运行性能。

（2）指令集扩展验证

在本部分中，通过我重新设计的.coe 文件作为 inst_rom 的指令，进行验证。

设计的指令如下：

nand $13,$25,$7　00000|11001|00111|01101|00000|111111，0327683F

subiu $13,$5,#5　111101|00101|01101|0000000000000101，F4AD0005

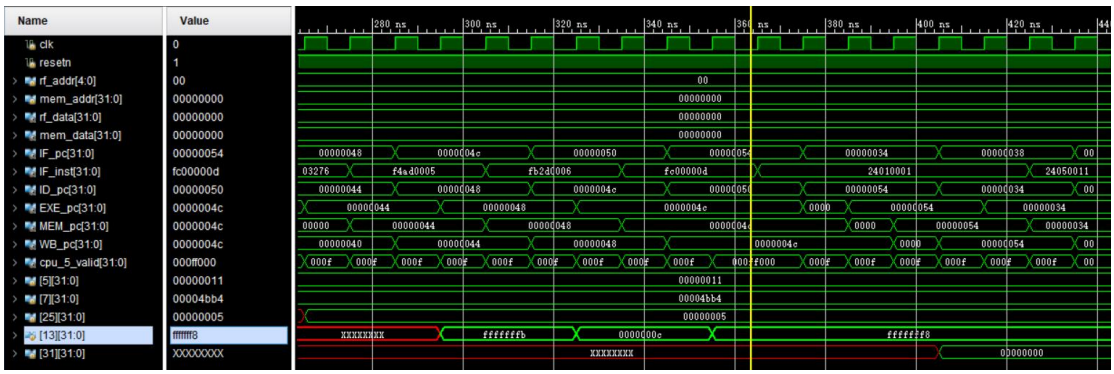
nori $13,$25,#6　111110|11001|01101|0000000000000110，FB2D0006

jc #34H　111111|00000000000000000000001101，FC00000D

设计文件如下：

```
; Initialization file for a
; 32-bit wide ROM
memory_initialization_radix = 16;
memory_initialization_vector =
; EX_JUMP_ADDR
24010001
24010001
24010001
24010001
24010001
24010001
24010001
24010001
24010001
24010001
24010001
24010001
24010001　开始地址之前的都用 addiu $1, $0,#1　即[$1]=1 填充
; START_ADDR
24010001　34H:addiu $1, $0,#1,[$1]=1
24050011　38H:addiu $5,$0,#17,[$5]=17 以下三条指令是为验证扩展指令准备寄存器
2407AAAA　3CH:addiu $7,$0,#0x4BB4,[$7]=0x4BB4
24190005　40H:addiu $25,$0,#5,[$25]=5
0327683F　44H:nand $13,$25,$7 以下四条指令是扩展指令集的指令
F4AD0005　48H:subiu $13,$5,#5
FB2D0006　4CH:nori $13,$25,#6
FC00000D　50H:jc #34H
24010001　由于程序存在跳转之前会多向后执行一条指令的 bug，因此多预留两条
24010001　addiu $1, $0,#1　即[$1]=1，以保证稳定运行
```

仿真结果如下：



44H 结束之后，$13 的结果为 0xfffffffb。

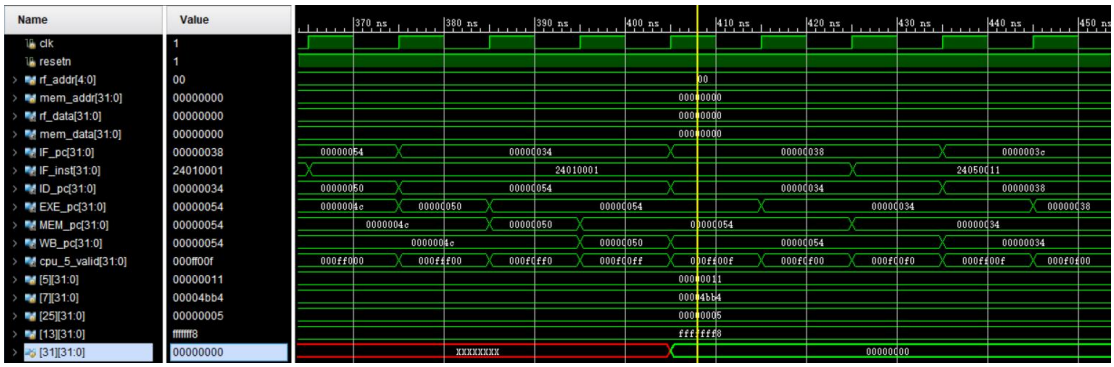
$25 为 0x00000005，$7 为 0x00004BB4，两者相与为 0x00000004，与非为 0xfffffffb，是正确的。

48H 结束之后，$13 的结果为 0x0000000c。

$5 为 0x00000011=17，17-5=12，因此也是正确的。

4CH 结束之后，$13 的结果为 0xfffffff8。

5 和 6 进行或运算，结果为 7，再进行非运算，结果为 0xfffffff8，同样是正确的。



50H 结束后，跳转到了 34H，$31 被清零，也是正确的。

至此，所有扩展的指令都得到了正确的验证。

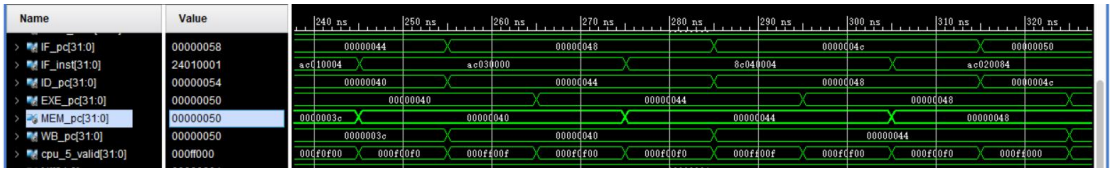
（3）cache 优化对比

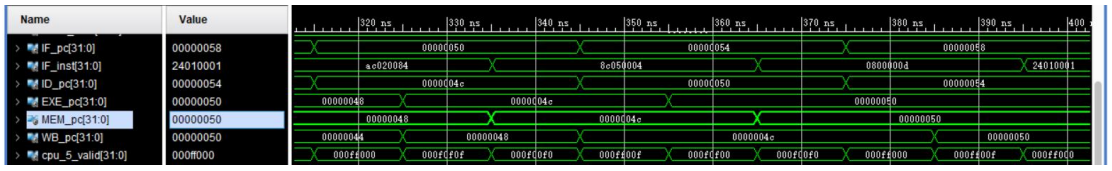
测试 coe 文件设计：

在 34H 之前的指令仍然通过 24010001 进行填充，不再赘述。

```
; START_ADDR
24010001    34H:addiu $1, $0,#1,[$1]=1
24020010    38H:addiu $2,$0,#0x10,[$2]=0x10
24030100    3CH:addiu $3,$0,#0x100,[$3]=0x100
AC010004    40H:sw $1,#4($0)      cache(1)更新
AC030000    44H:sw $3,#0($0)      cache(0)更新，不影响 cache(1)
8C040004    48H:lw $4,#4($0)      cache(1) hit!
AC020084    4CH:sw $2,#132($0)    #132 和#4 的映射 cache 块相同(1)，会换出#4
8C050004    50H:lw $5,#4($0)      cache(1) miss!
0800000D    54H:j 34H
24010001
24010001
```

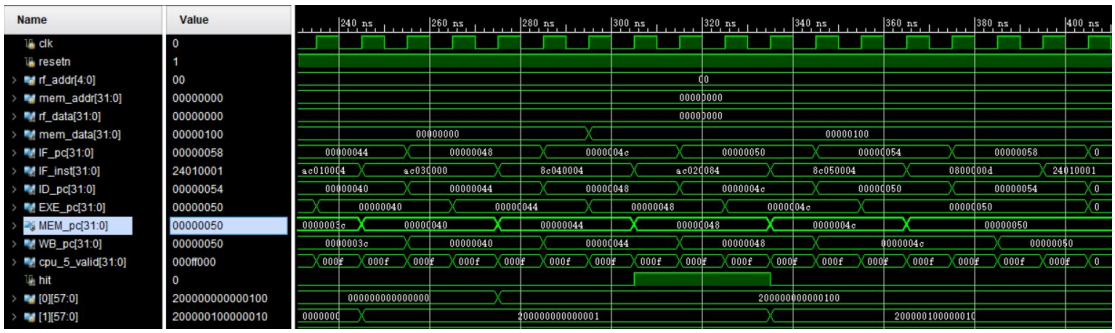
以下是优化之前的仿真结果：





可以看到写指令 40H，44H，4CH 的 MEM 阶段都只需要 1 周期，但 48H 和 50H 的读指令的 MEM 阶段却需要 2 周期。

以下是优化后的结果：可以看到 48H 的 cache HIT 情况需要 1 周期就可以完成，而 50H 的 cache MISS 则还需要 2 周期完成。



这说明我们的优化是很成功的，以下将对 cache 的运行过程进行详细分析。（在仿真结果图中也显示了对应信号，便于观察）

$1,$2,$3 的初始值分别为 0x1，0x10，0x100

40H:sw $1,#4($0)

Cache	valid[57]	addr[56:32]	data[31:0]
0	0	0	0
1	1	0	1

对 Cache[1]进行了刷新，addr 标记为 0，更新为寄存器$1 的值。

44H:sw $3,#0($0)

Cache	valid[57]	addr[56:32]	data[31:0]
0	1	0	100
1	1	0	1

对 Cache[0]进行了刷新，addr 标记为 0，data 更新为寄存器$3 的值。

48H:lw $4,#4($0)

检查 Cache0，发生 Cache Hit!

此时 Hit 为 1，直接使用 Cache 结果，有效节约了访存开销。

4CH:sw $2,#132($0)

Cache	valid[57]	addr[56:32]	data[31:0]
0	1	0	100
1	1	1	10

对 Cache[1]进行了刷新，addr 标记为 1，data 更新为寄存器$2 的值。

50H:lw $4,#4($0)

可以发现，cache[1]存储的数据并不是#4 的数据，发生 Cache Miss!

此时 Hit=0，只能从内存中读取结果。

6、 总结感想

在本次实验中，我对 5 级流水线 CPU 进行了改进，分别完成了前递的设计，指令集的扩展以及 cache 的设计。在实验过程中，我对流水线 CPU 有了更进一步的认识，对 Verilog 语法也加深了认识。通过一次一次的尝试与修改，在仿真中一个一个 wire or reg 变量的查询和 debug 我对 CPU 的运行过程也更加熟悉。

这学期体系结构实验，在先前组成原理实验的基础上，对 CPU 的运行流程的理解变得更加深刻，与理论课所讲授知识结合，增进了对体系结构的理解。