组成原理课程第 七 次实验报告

实验名称：五级流水线 CPU 实验

学号： 2312141 姓名： 张德民 班次： 李涛老师

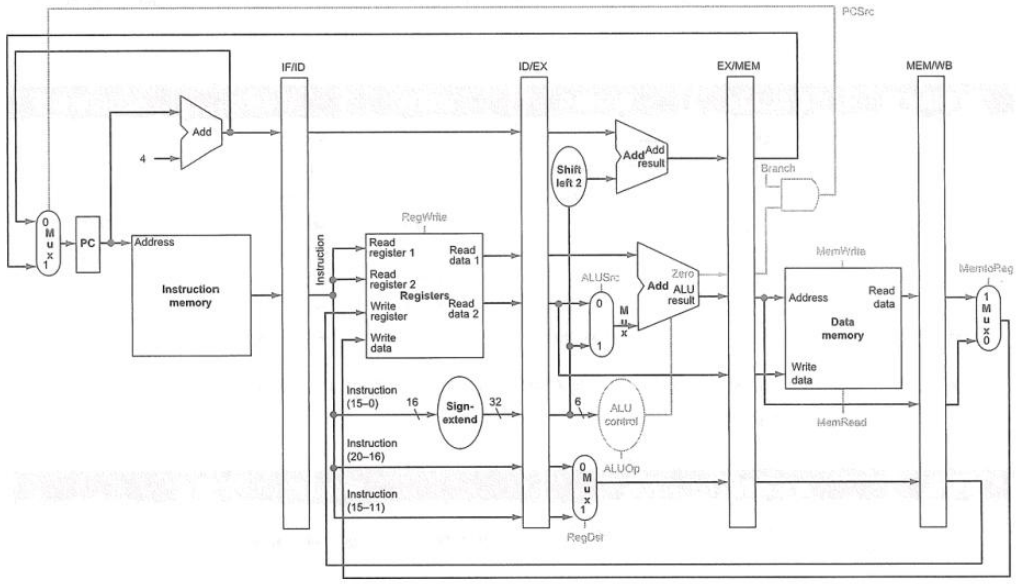
一、 实验目的

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。
6. 自行添加流水线指令。

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）

三、 实验原理图



上图是一个流水线数据通路示意图。

四、 实验步骤

1. 分析原代码功能

本次任务需要我去修改源代码的 bug，那么就必须了解本次实验各部分代码的功能。这里我主要分析在多周期实验中没有的代码模块的功能。

本次实验的多数模块和之前是一样的，包括寄存器，存储器，IF,ID,EX,MEM,WB 等基本上是一样的，主要区别在于 pipeline_cpu 文件，如下图，多了五个后缀是 in 的 wire 型数据，还有一个 cancel 数据。

```
//5模块的valid信号
reg IF_valid;
reg ID_valid;
reg EXE_valid;
reg MEM_valid;
reg WB_valid;
//5模块执行完成信号,来自各模块的输出
wire IF_over;
wire ID_over;
wire EXE_over;
wire MEM_over;
wire WB_over;
//5模块允许下一级指令进入
wire IF_allow_in;
wire ID_allow_in;
wire EXE_allow_in;
wire MEM_allow_in;
wire WB_allow_in;

// syscall和eret到达写回级时会发出cancel信号，
wire cancel;    // 取消已经取出的正在其他流水级执行的指令
```

并且设定了它们的赋值逻辑，即当本级无效（本级没有收到指令）或者本级完成运算并且下一级允许进入的时候，允许信号才为 1，否则为 0。

```
//各级允许进入信号:本级无效，或本级执行完成且下级允许进入
assign IF_allow_in  = (IF_over & ID_allow_in) | cancel;
assign ID_allow_in  = ~ID_valid  | (ID_over  & EXE_allow_in);
assign EXE_allow_in = ~EXE_valid | (EXE_over & MEM_allow_in);
assign MEM_allow_in = ~MEM_valid | (MEM_over & WB_allow_in );
assign WB_allow_in  = ~WB_valid  | WB_over;
```

并且还加了一个 cpu_5_valid 信号,用来展示当前那些阶段是可用的。

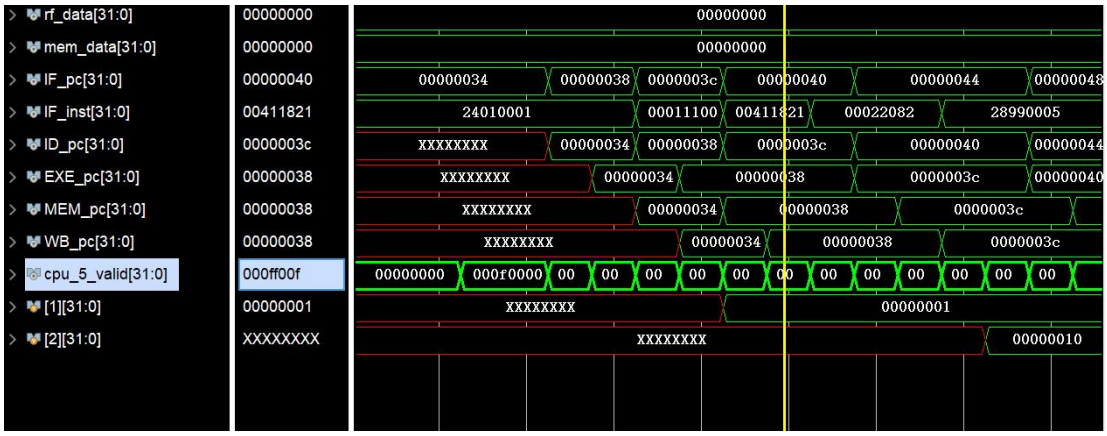
```
//展示5级的valid信号
assign cpu_5_valid = {12'd0          ,{4{IF_valid }},{4{ID_valid}},
                      {4{EXE_valid}},{4{MEM_valid}},{4{WB_valid}}};
```

那么这时候我基本就能明白这个五级流水线的实现逻辑了，它和之前的多周期基本上是一样的，区别就在于多了五个允许进入信号，来控制指令的允许，当前一个指令的某个阶段运行完以后，如果这个阶段允许进入，就可以允许下一个指令的该阶段。

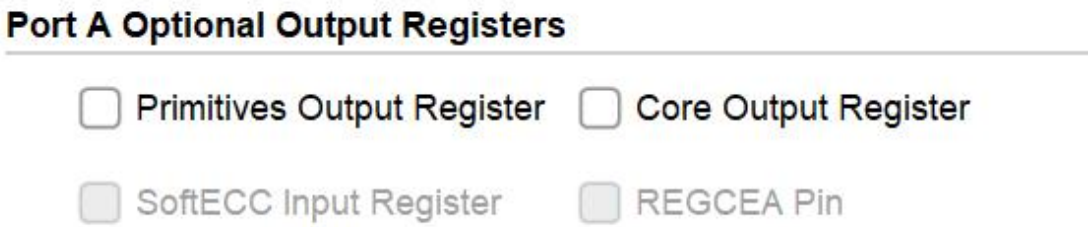
2. 分析修改原代码 BUG

2.1 与多周期一样的 cpu 周期推迟的 bug

这个问题老师在题目上说了，我这里先再去验证一下看看。

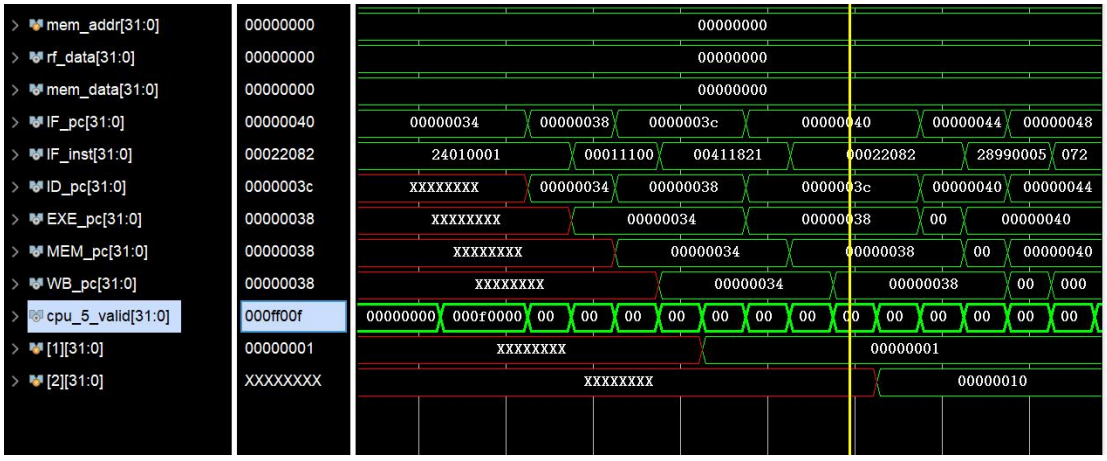


如上图，看 WB_pc,此时运行的是第二条指令(向 2 号寄存器写入的命令)，但是可以看见，在这个周期结束以后，2 号寄存器并没有写入任何的值，而是在第三条指令运行完以后，才被写入，因此相同的问题仍然存在，我使用一样的办法去解决即可。



如上图，把 primitives output register 选项去除就好了。这个选项会导致时钟被延迟一个周期。

改完后重新仿真，可以看见，时钟延迟一个周期的问题就没有了。



2.2 指令相关问题

这一部分根据课本上说的，主要应该是数据依赖，不过具体是什么问题还需要去具体分析。我一步一步修改指令进行测试：

数据冒险的问题：

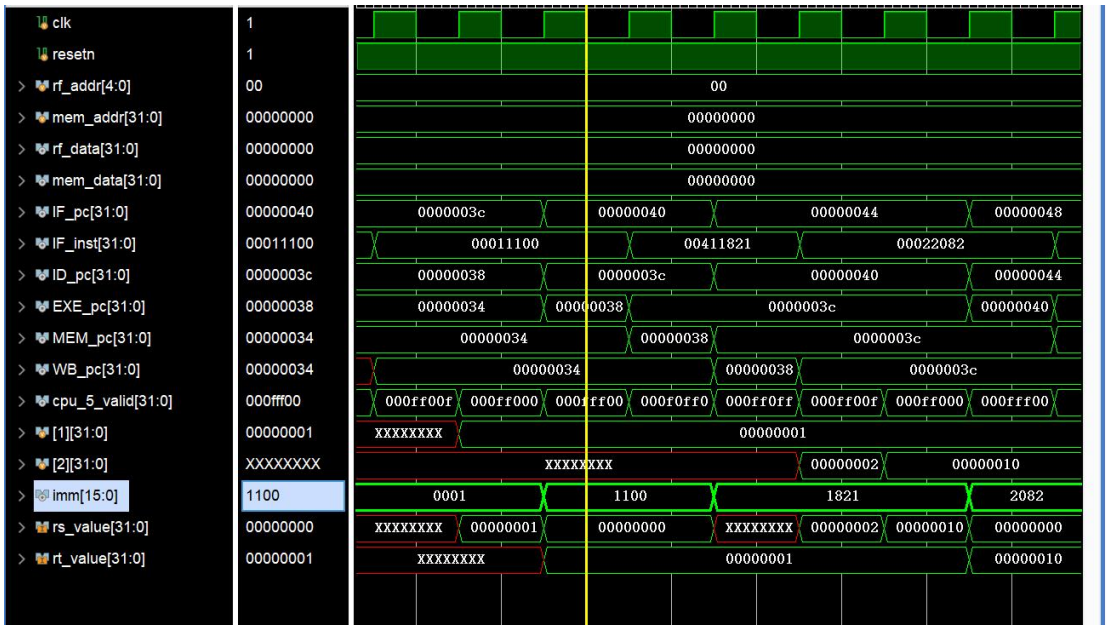
第一:读后写的问题，即后边指令要读取一个寄存器的值，而这个寄存器的值还没有写入。

24010001

24220001

我用上边的 24010001 和 24220001 来测试，其功能为 addiu　$1, $0,#1　　addiu　$2, $1,#1 测试结果如下：

首先，可以看见 cpu_5_valid 里，当第一条指令运行到 ID（译码与取值）阶段的时候，第二条指令开始了 IF（取指令），但是可以看见，当第一条指令开始 EX 阶段，第二条也没有开始第二步，因为需要的$1 还没有写入，当第一条指令结束 exe 阶段以后，第二条指令进入了第二阶段 ID，然后就一直停在了第二阶段，直到第一步完成了写回操作以后，第二个指令才进行 ex 阶段。这样就保证避免了写后读的数据冲突问题。



我也去看了第二步的 rs，rt 以及立即数的值，可以看见，当第一条指令的 wb 结束后，寄存器 1 的值被写入，然后 rs 的值就变成了 1，即正常读取到了需要的值，避免了相关问题。

```
assign rs_wait = ~inst_no_rs & (rs!=5'd0)
               & ( (rs==EXE_wdest) | (rs==MEM_wdest) | (rs==WB_wdest) );
assign rt_wait = ~inst_no_rt & (rt!=5'd0)
               & ( (rt==EXE_wdest) | (rt==MEM_wdest) | (rt==WB_wdest) );

//对于分支跳转指令，只有在IF执行完成后，才可以算ID完成；
//否则，ID级先完成了，而IF还在取指令，则next_pc不能锁存到PC里去，
//那么等IF完成，next_pc能锁存到PC里去时，jbr_bus上的数据已变成无效，
//导致分支跳转失败
//(~inst_jbr | IF_over)即是(~inst_jbr | (inst_jbr & IF_over))
assign ID_over = ID_valid & ~rs_wait & ~rt_wait & (~inst_jbr | IF_over);
```

在 ID 的代码里，我找到了两个新的值 rs_wait 和 rt_wait,我通过分析明白了，这两个值的作用是实现只要当前执行的指令前边需要向内存或寄存器里写入新的值，并且这个写入到地址和我们要读取的地址是一样的，那么 ID 阶段就不会结束，那么新的指令就无法进入到 ID 阶段。直到已经写入后。

因此写后读的冲突问题，原代码已经通过插入空指令的操作解决了。

第二，写后读的问题，在流水线里一般不会出现，这个一般在乱序执行里出现。

第三，是写后写的问题，这个也是，在流水线里一般不会出现，这个一般在乱序执行里出现。

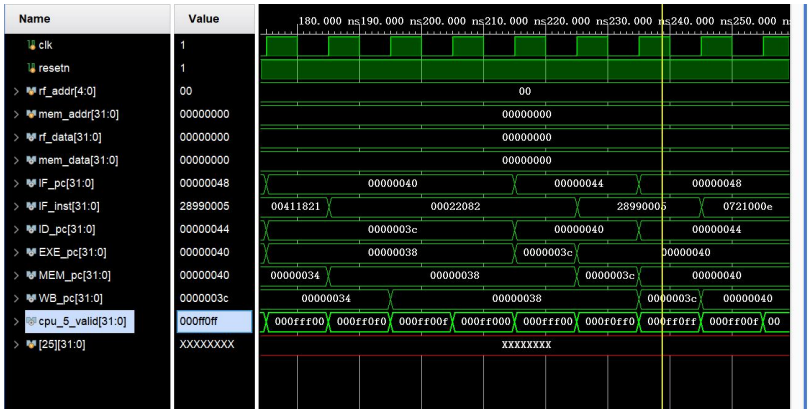
综上所述，目前的五级流水线没有发现数据依赖的指令相关问题。

2.3 流水线冲突的问题

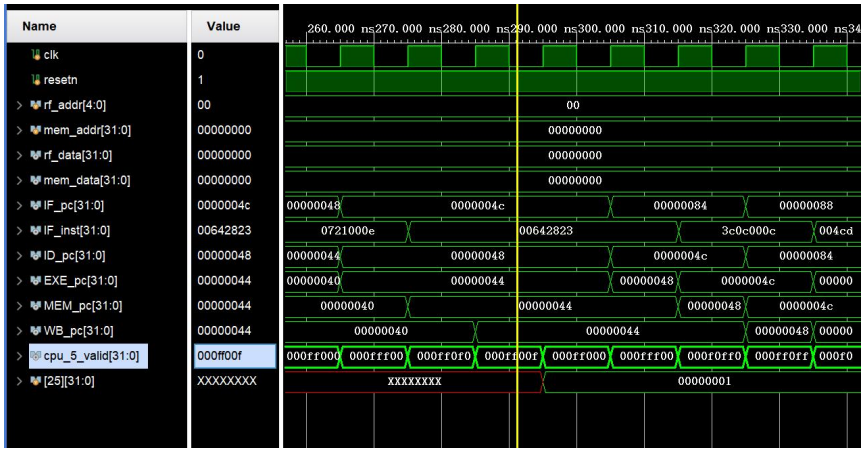
流水线冲突根据课本上讲的，主要就是数据冒险，控制冒险，还有结构冒险。而数据冒险就是我刚刚分析的指令相关问题，因此这里主要去分析控制冒险和结构冒险的问题。

控制冒险的问题：

控制冒险主要出现在跳转语句里，这俩我测试的是 0721000E，bgez　$25,#14，即比较$25 的值是不是大于 0，如果是则跳转，否则顺序执行。

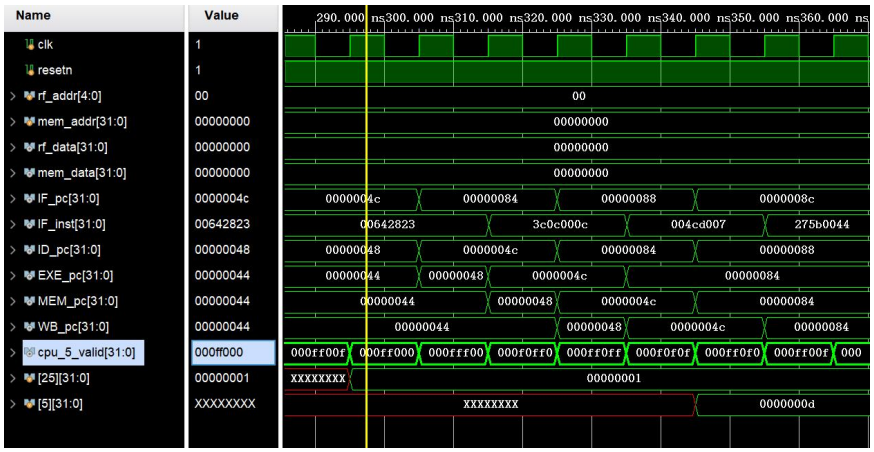


如上图，此时的跳转语句刚好到第一步取指，而计算$25 的指令才到第 2 阶段 ID，



如上图，在$25 写回前，可以看见跳转指令一直在 ID 阶段，即确定跳转指令的地址，而 IF 阶段一直可以进行，因此下一条指令就进行了顺序取值，看见，IF_PC 在 48（跳转指令的 pc 值）之后是 4c，即顺序读取了。但是当$25 的值写入以后，可以看见然后下一个取值的 PC 就是目标的跳转地址 84。

但是这样就产生了一个错误，那就是我并不想执行地址为 4c 的指令，但是 cpu 还是读取了它并且执行了，这个指令的功能是 subu　$5, $3, $4，即给$5 赋值为 d。如下图，可以看见，我们确实是实现了跳转的功能，但是却多往后执行了一条指令，导致$5 被赋值为 d。



综上所述，我们的 cpu 五级流水线是存在数据依赖的相关问题的，这个问题体现在当我们在运行一个跳转指令的时候，cpu 会顺序读取下一条指令，但是当我们的跳转语句的结果出来以后，即要跳转的指令地址已经确定，但是这个被顺序读取的指令仍然会被执行。

结构冒险的问题：

控制冒险主要是因为硬件资源不足的问题，例如取指和内存访问共享总线/端口，不能同时进行。但是本次的流水线中取指是随时进行的，而内存访问可以正常进行，因此暂时没有发现有结构冒险的问题。

2.4 关于修复思路

根据上边的分析测试，我找到了一个控制冒险的问题，但是我又查阅了一些资料，发现

这个严格上来说不能算是一个错误，这种策略叫做延迟槽，是指在一些早期的流水线 CPU 架构中，尤其是 RISC 架构（如 MIPS）中，为了解决分支指令（如跳转、函数调用）造成的流水线中断问题，在分支指令之后安排的一条指令，该指令在逻辑上仍然在分支前执行。也就是说，即使遇到跳转，延迟槽中的指令也会被执行，从而减少流水线空转带来的性能损失。

但是在我的角度来看，这样做可能是有一些风险的，比如多执行的指令如果影响了原程序的某些值，进而导致整个程序出错，因此有必要采取一些策略去避免这个错误。

解决延迟槽问题的常用方法是由编译器在延迟槽中填入一条对程序逻辑无影响、但能有效利用 CPU 资源的指令，如将原本应在分支前执行的有用指令移入延迟槽；若没有合适指令可填，则插入一条空操作（NOP）指令以保持程序正确性。同时，现代 CPU 架构则通过引入分支预测、乱序执行等机制来避免使用延迟槽，从根本上解决因分支带来的流水线中断问题。

在现代 CPU 中，为避免编程和编译上的复杂性，很多架构（如 x86、ARM 的后期版本）取消了延迟槽机制，转而使用分支预测、流水线回退和乱序执行技术。例如，如果预测一个分支将被执行，CPU 会预先加载对应路径上的指令，只有在预测错误时才进行回退处理。这使得程序不再需要显式管理延迟槽，程序更简洁、性能也更高。

3. 增加新指令

根据要求，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）。

3.1 基本算术指令

根据《cpu 设计实战》这本书，我选择添加原 cpu 没有的基本算术指令 ADD，即有符号加法，这主要是因为原代码已经比较全面了，可供选择的指令确实不多。不过原代码只实现了 ADDU，即无符号加法，因此我选择加入一个有符号加法。

首先，在实现指令列表里加入 ADD。

```
// 实现指令列表
wire inst_ADDU, inst_SUBU , inst_SLT , inst_AND;
wire inst_NOR , inst_OR   , inst_XOR , inst_SLL;
wire inst_SRL , inst_ADDIU, inst_BEQ , inst_BNE;
wire inst_LW  , inst_SW   , inst_LUI , inst_J;
wire inst_SLTU, inst_JALR , inst_JR  , inst_SLLV;
wire inst_SRA , inst_SRAV , inst_SRLV, inst_SLTIU;
wire inst_SLTI, inst_BGEZ , inst_BGTZ, inst_BLEZ;
wire inst_BLTZ, inst_LB   , inst_LBU , inst_SB;
wire inst_ANDI, inst_ORI  , inst_XORI, inst_JAL;
wire inst_MULT, inst_MFLO , inst_MFHI, inst_MTLO;
wire inst_MTHI, inst_MFC0 , inst_MTC0;
wire inst_ERET, inst_SYSCALL;
wire op_zero;  // 操作码全0
wire sa_zero;  // sa域全0
wire inst_ADD;
```

然后为有符号加法分配一个功能码：101001

```
assign inst_ADD= op_zero & sa_zero    & (funct == 6'b101001);//有符号加法
```

然后设定 alu 的运算，即加法：

```
assign inst_add = inst_ADDU | inst_ADDIU | inst_load
                | inst_store | inst_j_link |inst_ADD;
```

然后确定写入的地址，rd 寄存器：

```
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_ADD;
```

到这里这个基本运算指令有符号加法就完成了。

3.2 乘法与除法指令

首先我优化了原代码的乘法模块，将其由原来的一位乘法改为了两位乘法，优化了运行速度。

其次我增加了一个除法指令。

乘法优化

首先可以看见在原来的代码里是有乘法的部分的，但是我尝试运行了一下，发现是可以跑通的，但是存在一个问题是运算消耗的周期数太多了，效率很低。我这里主要是进行了一个效率优化。

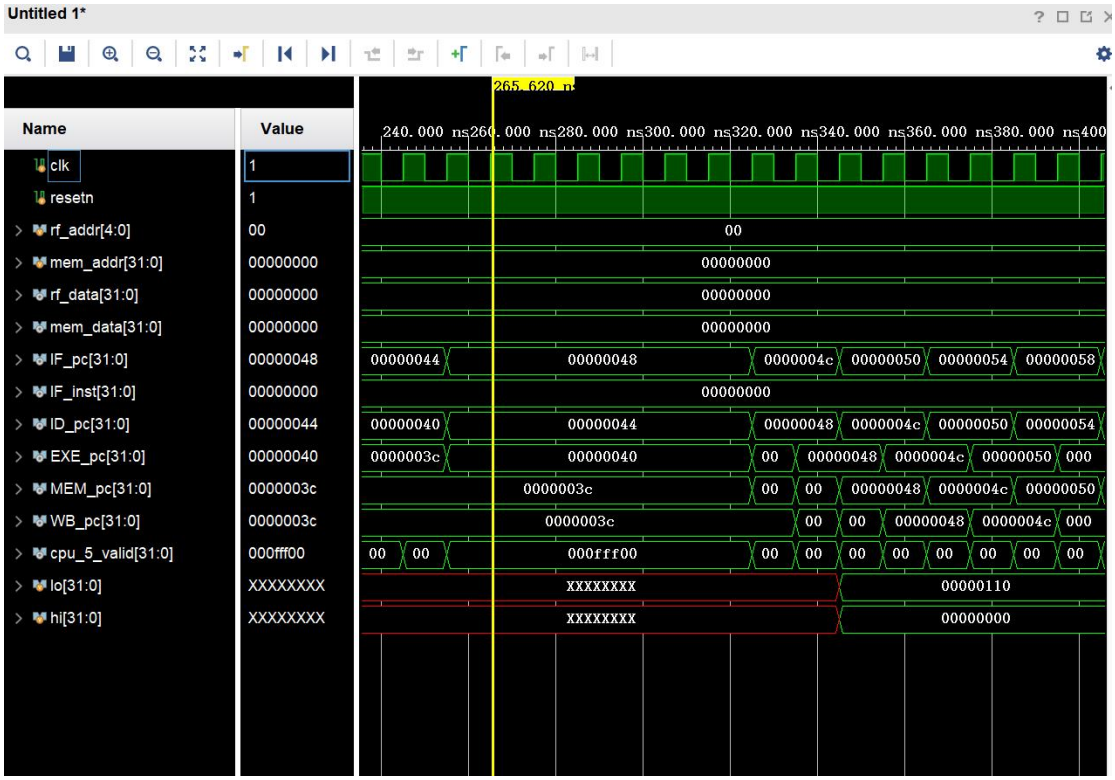
首先我根据代码可以知道，最后乘法的结果是在 WB 阶段写入到了 HI,Lo 两个 32 位的寄存器里:

```
//HI用于存放乘法结果的高32位
//LO用于存放乘法结果的低32位
reg [31:0] hi;
reg [31:0] lo;
```

然后看我运行的指令：前三条的功能是令$1=1,$2=2,$3=3,最重要的是第四条，其作用是计算$2*$3。

```
24010001
00011100
00221829
00430018
```

然后开始仿真测试：



如图可以看见，000fff00 即对应乘法运算在 exe 阶段，并且很明显可以看出来这个运算消耗的周期数要远远大于其他的周期数，一共用了 7 个时钟周期。

接下来我去进行优化：

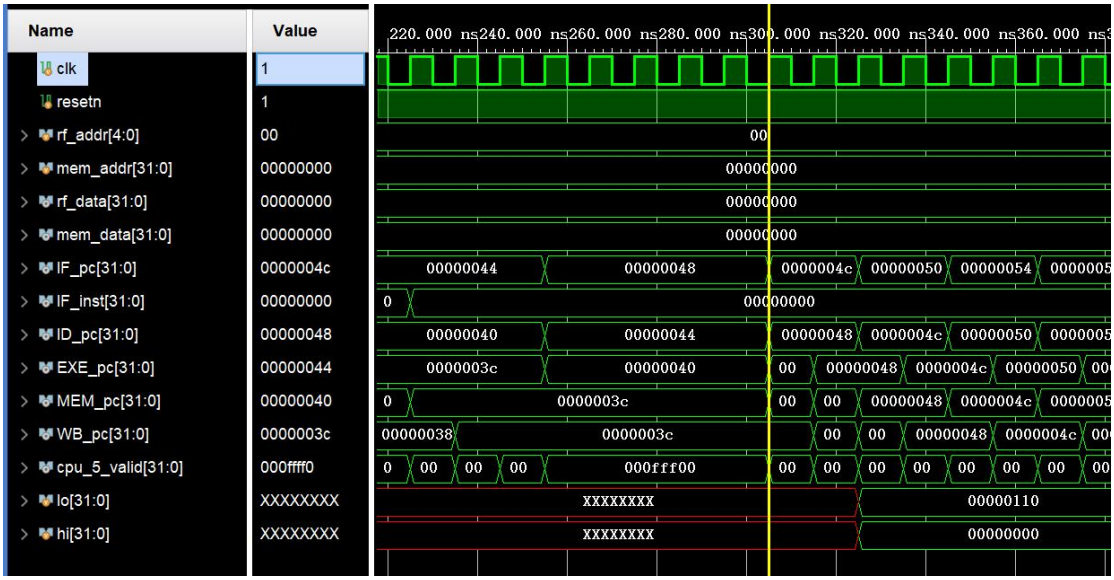
```
multiplicand <= {multiplicand[61:0],2'b00};

multiplier <= {2'b00,multiplier[31:2]};
```

首先把原来的一次移动一位改为一次移动两位。

```
wire [63:0] partial_product1;
wire [63:0] partial_product0;
wire [63:0] partial_product;
assign partial_product0=multiplier[0] ? multiplicand : 64'd0;
assign partial_product1=multiplier[1] ? {multiplicand[62:0],1'b0} : 64'd0;
assign partial_product = partial_product0+partial_product1;
```

然后修改部分积的运算逻辑，因为原先一位的话，只有 0，1 两种可能，而改为移动两位，就变成了 00，01，10，11 四种可能，需要修改计算逻辑。



修改以后的仿真图如上，可以看见，原来的 exe 阶段使用了 7 个时钟周期，而修改以后使用了 5 个时钟周期，运行效率确实提高了。

除法指令

接下来我添加一个除法指令：

```
wire inst_DIVI;
```

```
assign inst_DIVI=op_zero & (rd==5'd0)
                 & sa_zero & (funct == 6'b011001);
```

首先增加指令，并分配功能码。

```
//EXE需要用到的信息
wire multiply;          //乘法MULT
wire division;
```

```
assign division=inst_DIVI;
```

```
assign ID_EXE_bus = {division,multiply,mthi,mtlo,
```

```
output      [159:0] ID_EXE_bus,    // ID->EXE总线
```

然后多传一共 division 信号到 exe 阶段。注意对齐总线的宽度。

```
//-----{ID->EXE总线}begin
    //EXE需要用到的信息
    wire multiply;          //乘法
    wire division;
```

```
assign {division,
        multiply,
        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;
-----{ID->EXE总线}end
```

然后在 exe 阶段接收这个 division 信号。

```
wire [63:0] product1;
wire [63:0] product2;
```

用 product1 和 product2 来区分乘法除法的结果。

```
assign mult_begin = multiply & EXE_valid;
multiply multiply_module (
    .clk       (clk       ),
    .mult_begin(mult_begin  ),
    .mult_op1  (alu_operand1),
    .mult_op2  (alu_operand2),
    .product   (product1   ),
    .mult_end  (mult_end  )
);
```

最后设定一个 div_begin 信号并实例化一个除法器模块就可以了。

最后是我的除法模块的具体实现：

```
// 进行除法运算，每次循环将被除数减去除数，直到被除数小于除数
always @(posedge clk) begin
    if (div_valid) begin
        if (dividend >= divisor) begin
            dividend <= dividend - divisor;  // 被除数减去除数
            quotient_temp <= quotient_temp + 1;  // 商加1
        end
    end else if (div_begin) begin
        dividend <= op1_abs;  // 被除数加载
        divisor <= op2_abs;   // 除数加载
        quotient_temp <= 32'd0;  // 商清零
    end
end
```

其本质上就是进行了多次减法操作，这样可能会导致运行效率比较低，尤其是被除数远远大于除数的时候，不过考虑到这个添加比较复杂，为了避免出现更多问题，就先选择了这个比较直接的算法。

这时候就可以进行除法操作了，具体示例在第五节实验结果分析里。

3.3 转移指令

我添加的转移指令是 BLTZAL(小于零则跳转并链接（保存返回地址到地址为 31 的寄存器）。下边介绍修改的过程：

```
wire inst_BLTZAL;

assign inst_BLTZAL=(op == 6'b110000) & (rt==5'd0);
```

首先第一步还是定义指令以及分配操作码。

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

然后在分支跳转指令里加入 inst_BLTZAL,但需要注意的是，BLTZAL 不仅仅是分支跳转，它还有链接的功能。

```
assign inst_add = inst_ADDU | inst_ADDIU | inst_load
                | inst_store | inst_j_link |inst_ADD|inst_BLTZAL;
```

如上图，规定 inst_BLTZAL 进行加法运算，目的是求原地址，因此后边我还需要去修改一下 alu 的操作数 1 和操作数 2 的赋值逻辑。

```
assign inst_wdest_31 = inst_JAL|inst_BLTZAL;
```

如上图，规定把返回地址写入 31 号寄存器。

```
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;
```

如上图，确定分支跳转的条件，inst_BLTZAL 跳转的条件是如果 rs 寄存器的值小于 0 则跳转。

```
assign alu_operand1 = inst_j_link ? pc :
                      inst_BLTZAL ? pc :
                      inst_shf_sa ? {27'd0,sa} : rs_value;
assign alu_operand2 = inst_j_link ? 32'd8 :
                      inst_BLTZAL ? {16'd1,~sa+1}:
                      inst_imm_zero ? {16'd0, imm} :
                      inst_imm_sign ?  {{16{imm[15]}}, imm} : rt_value;
```

最后是确定操作数 1 和操作数 2 的值，如上图，操作数 1 的值是跳转后的地址，那么我设定操作数 2 的值就是偏移量的相反数，这样相加就能得到原地址了。

综上就实现了 BLTZAL 的功能，具体测试示例在第五节。

3.4 访存指令 1：访问

这一部分我选择添加的访问指令是 LHU，即加载一个半字（16 位），并且高 16 为用 0 填充。具体添加操作如下：

首先还是定义指令以及分配操作码：

```
wire inst_LHU;

assign inst_LHU    = (op == 6'b100101);              //load半字
```

接下来在 inst_load 里加入 inst_LHU 这一条指令。这是用来确定进行访问操作。

```
assign inst_load  = inst_LW | inst_LB | inst_LBU|inst_LHU;  // load指令
assign inst_store = inst_SW | inst_SB;                      // store指令
```

然后在传给 mem 的信号里加上一个 lh_sign:

```
assign lb_sign = inst_LB;
assign lh_sign =inst_LHU;
assign ls_word = inst_LW | inst_SW;
assign mem_control = {inst_load,
                      inst_store,
                      ls_word,
                      lb_sign,
                      lh_sign };
```

注意，这里需要修改 mem_control 的宽度，还有 ID 到 exe 的数据总线以及 exe 到 mem 的数据总线的宽度，这里需要修改的地方还是蛮多的，需要小心检查：

```
output      [160:0] ID_EXE_bus,   /
```

```
wire [4:0] mem_control;   //MEM需要使用的控制信号
```

```
output      [154:0] EXE_MEM_bus,  // EXE->MEM总线
```

最后在 mem 阶段接收这个信号：

```
//-----{load/store切存}begin
wire inst_load;   //load操作
wire inst_store;  //store操作
wire ls_word;     //load/store为字节还是字,0:byte;1:word
wire lb_sign;     //load一字节为有符号load
wire lh_sign;
assign {inst_load,inst_store,ls_word,lb_sign,lh_sign} = mem_control;
```

最后，要修改 mem_result 的赋值逻辑：

```
assign load_result[7:0] = (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] ;
assign load_result[15:8]= ls_word ? dm_rdata[15:8] :
                          lb_sign ? {8{lb_sign & load_sign}}:
                          dm_rdata[15:8];
assign load_result[31:16]= ls_word ? dm_rdata[31:16] :
                          lb_sign ? {16{lb_sign & load_sign}}:
                          {16'd0};
```

如上图，如果是 lh 指令，那么就只取后 16 位，前边的 16 位扩展位由 0 填充。

综上，一个取半字的访问无符号填充指令就实现了。具体的测试结果在第五节呈现。

3.5 访存指令 2：写入

这一部分我选择添加的写入指令是 SH，即写入一个半字（16 位）。具体添加操作如下：

首先第一步还是定义指令以及分配操作码：

```
wire inst_SH;
```

```
assign inst_SH    = (op == 6'b101101);                //向内存存储字节
```

然后接下来在 inst_store 里加入 inst_SH 这一条指令。这是用来确定进行写入操作。

```
assign inst_store = inst_SW | inst_SB|inst_SH;
```

然后在传给 mem 的信号里加上一个 sh_op,注意，这里需要修改 mem_control 的宽度，还有 ID 到 exe 的数据总线以及 exe 到 mem 的数据总线的宽度，这里需要修改的地方也很多，和之前的读取是一样的，需要小心检查:

```
wire lb_sign;  //load一字节为有符号load
wire lh_sign;
wire ls_word;  //load/store为字节还是字,0:byte;1:word
wire [5:0] mem_control;  //MEM需要使用的控制信号
wire [31:0] store_data;  //store操作的存的数据
wire sh_op;
assign sh_op=inst_SH;
assign lb_sign = inst_LB;
assign lh_sign =inst_LHU;
assign ls_word = inst_LW | inst_SW;
assign mem_control = {inst_load,
                      inst_store,
                      ls_word,
                      lb_sign,
                      lh_sign,
                      sh_op };
```

```
wire [ 63:0] IF_ID_bus;    // IF->ID级总线
wire [161:0] ID_EXE_bus;   // ID->EXE级总线
wire [155:0] EXE_MEM_bus;  // EXE->MEM级总线
wire [117:0] MEM_WB_bus;   // MEM->WB级总线

//锁存以上总线信号
reg [ 63:0] IF_ID_bus_r;
reg [161:0] ID_EXE_bus_r;
reg [155:0] EXE_MEM_bus_r;
reg [117:0] MEM_WB_bus_r;
```

然后在 mem 阶段接收接收这个信号：

```
wire inst_load;  //load操作
wire inst_store; //store操作
wire ls_word;    //load/store为字节还是字,0:byte;1:word
wire lb_sign;
wire lh_sign;    //load一字节为有符号load
wire sh_op;
assign {inst_load,inst_store,ls_word,lb_sign,lh_sign,sh_op} = mem_control;
```

最后是设置写使能信号，当 sh_op 为 1 的时候，设置写使能信号为 0011，意为只写入后 16 个比特的数据：

```
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 (sh_op)
        begin
            dm_wen <= 4'b0011;
        end
        else
        begin // SB指令，需要依据地址底两位，确定对应的写使能
```

综上，一个写入半字的写入指令就实现了。具体的测试结果在第五节呈现。

五、 实验结果分析

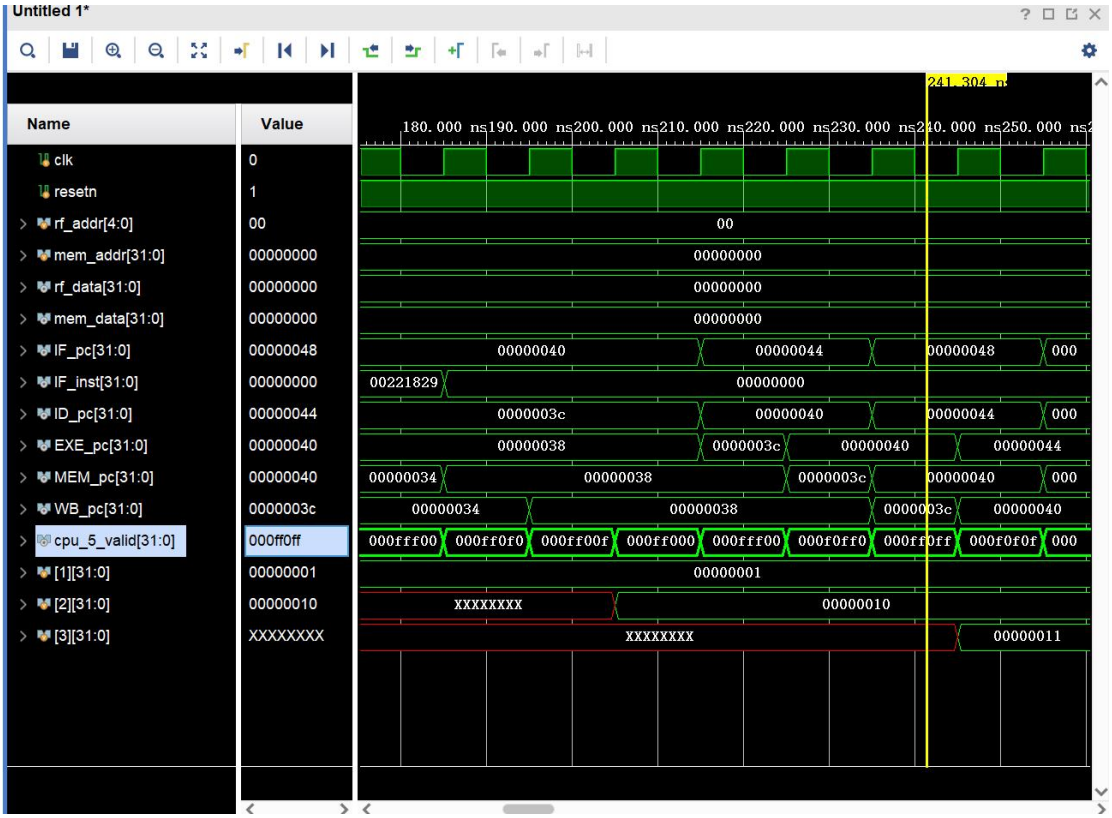
这一部分主要展示我补充的基本算术指令一条、乘除指令一条、转移指令一条、访存指令两条（一读一写）。

基本算术指令

这一部分我介绍我添加的有符号加法，首先是 coe 文件，其实现在功能是将寄存器 1 和寄存器 2 的值相加，存到寄存器 3 里：

```
; START_ADDR
24010001
00011100
00221829
```

如下图，可以看见，寄存器 1 的值是 1，寄存器 2 的值是 10，那么相加应该是 11。可以看见，当加法的 wb 阶段运行完以后，3 号寄存器的值变成了 11，和预测的值一样，因此加法的功能正确实现了。

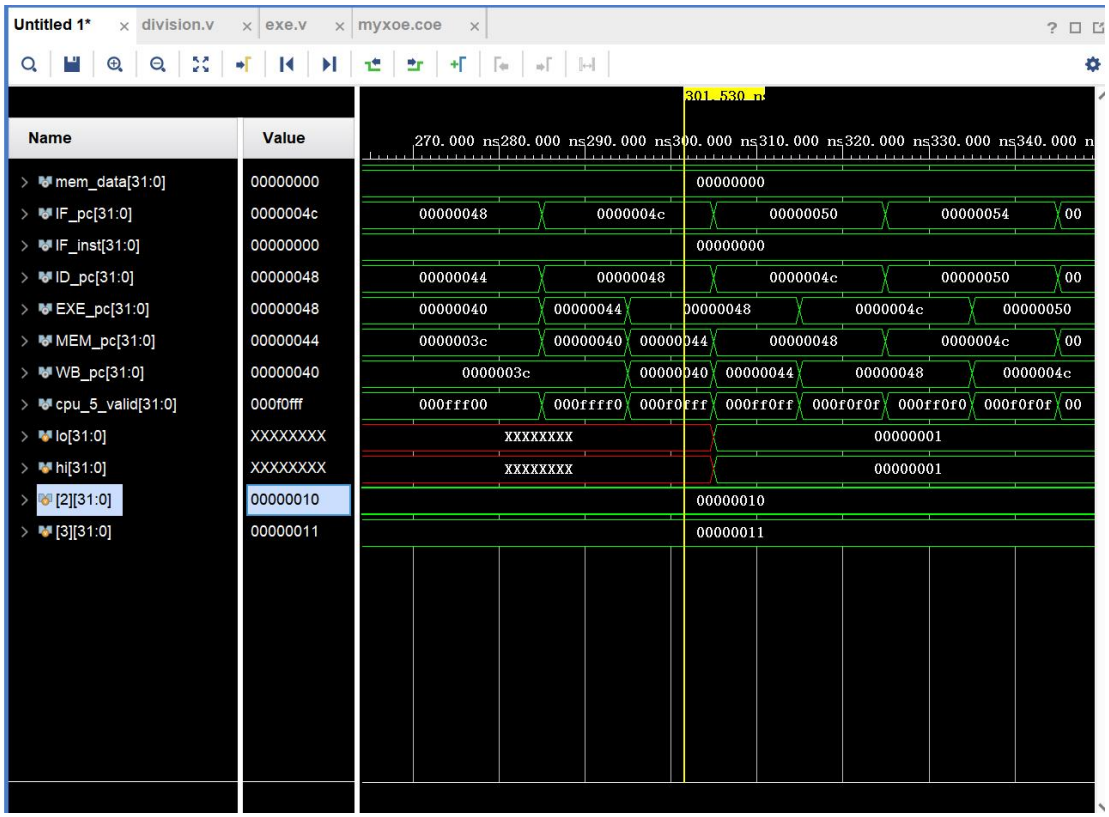


除法指令

这一部分我还优化了一下乘法指令，不过实验结果已经体现在第四节了，即运算周期数的减少，就不多介绍了，主要介绍除法指令。

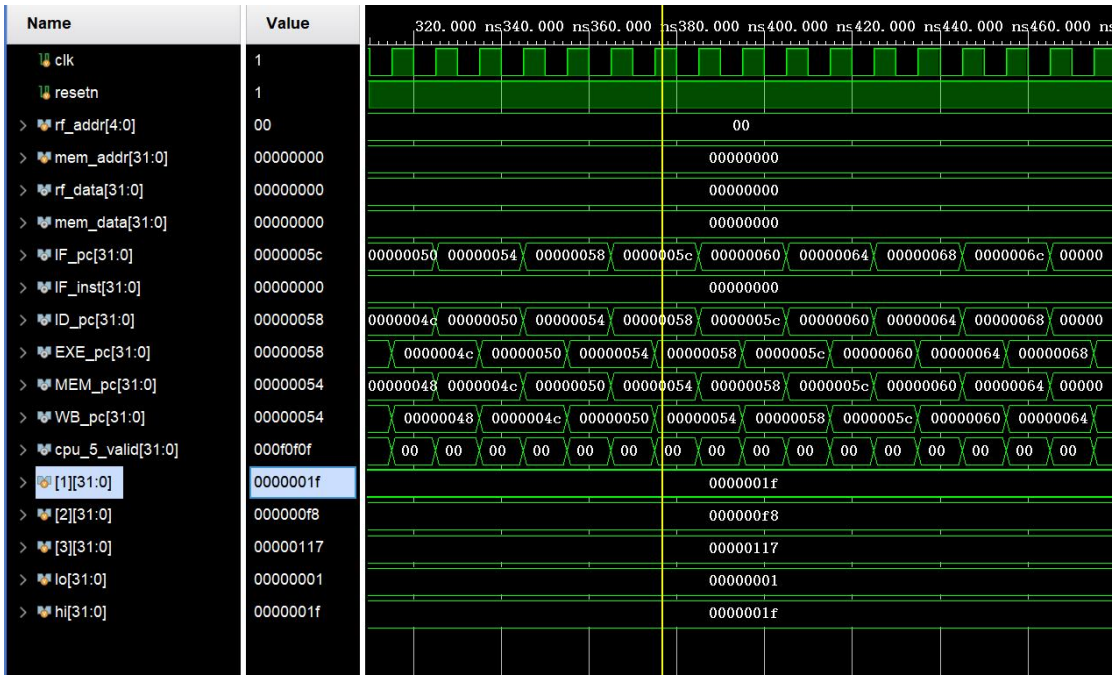
```
00620019
```

我的指令码是 00620019，其功能是将$3 寄存器的值除以$2 寄存器的值，最后的余数存在 hi 寄存器，商存在 lo 寄存器。



如上图，可以看见我的$3 寄存器的值是 3，$2 寄存器的值是 2，即计算 3 除以 2。如上图，很明显可以看见，当除法指令的 wb 阶段结束以后，lo 寄存器的值变成了 1，即商是 1，hi 寄存器的值是 1，即余数是 1.很明显 3 除以 2，商 1 余 1，答案正确。

当然这个可能有点太简单，所以我又测试了一个比较复杂的：



如上图,$3 是 117h，即 279，$2 是 f8h，即 248，除法结果应该是商 1 余 31。而图中 lo 寄存器的值是 1，即商 1，正确，hi 寄存器的值是 1f，即余 31，正确。

综上所述，除法指令的功能正常。

转移指令

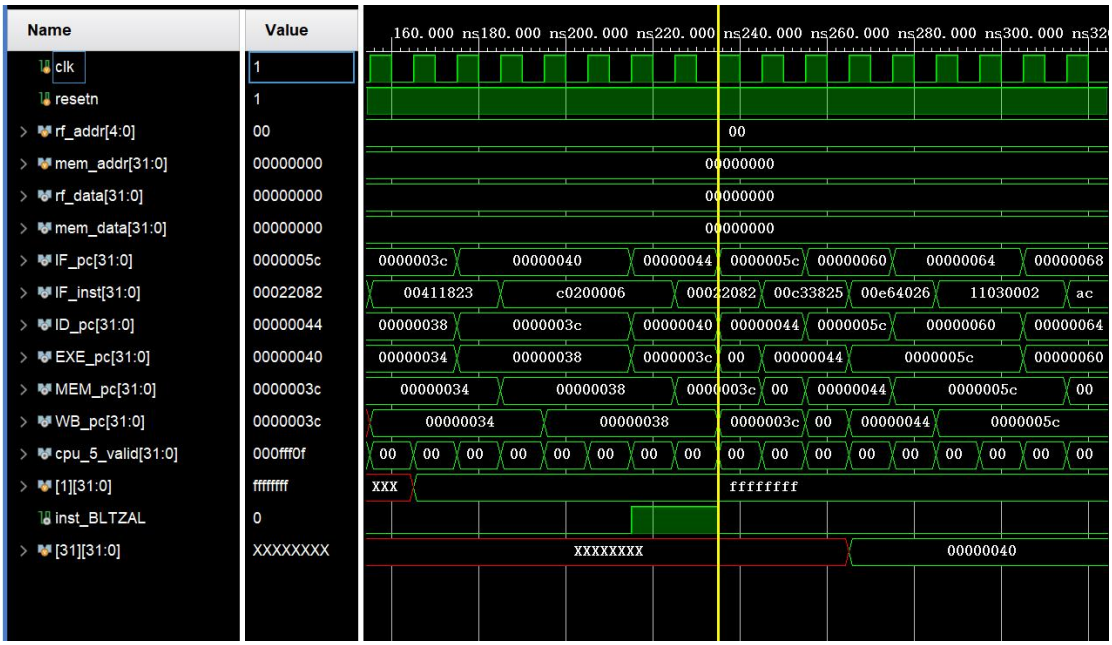
我添加的转移指令是 BLTZAL,即小于零则跳转并链接（保存返回地址到地址为 31 的寄存器）。

我的 coe 测试文件如下：

```
2401ffff
00011100
00411823
c0200006
```

重点是第一条和第四条指令，第一条指令的作用是将 1 号寄存器赋值为-1,第四条指令的作用是比较 1 号寄存器的值，如果小与 0，那么跳转到据当前指令地址为 6 的指令，我的跳转指令的 PC 值是 40H，那么如何跳转，应该跳转到 5cH。

如下图：



首先看 IF_PC,可以看见我的跳转指令 40H 之后是 44H，这个是延迟槽的问题，而 44H 之后，就变成了 5cH，即我们的跳转目的地址。因此跳转功能成功实现。

并且可以看见，我的 inst_BLTZAL 信号在跳转指令进行 ID 阶段的时候，一直是 1，因此可以确定运行的就是 BLTZAL 指令。

最后是链接功能，看图最下边的 31 号寄存器的值，很显然，当这个跳转指令执行完 WB 阶段以后，31 号寄存器的值变成了 40H，这刚好是我们的跳转指令的 PC 地址，执行成功。

综上所述，转移指令 BLTZAL,即小于零则跳转并链接（保存返回地址到地址为 31 的寄存器）的功能成功实现。

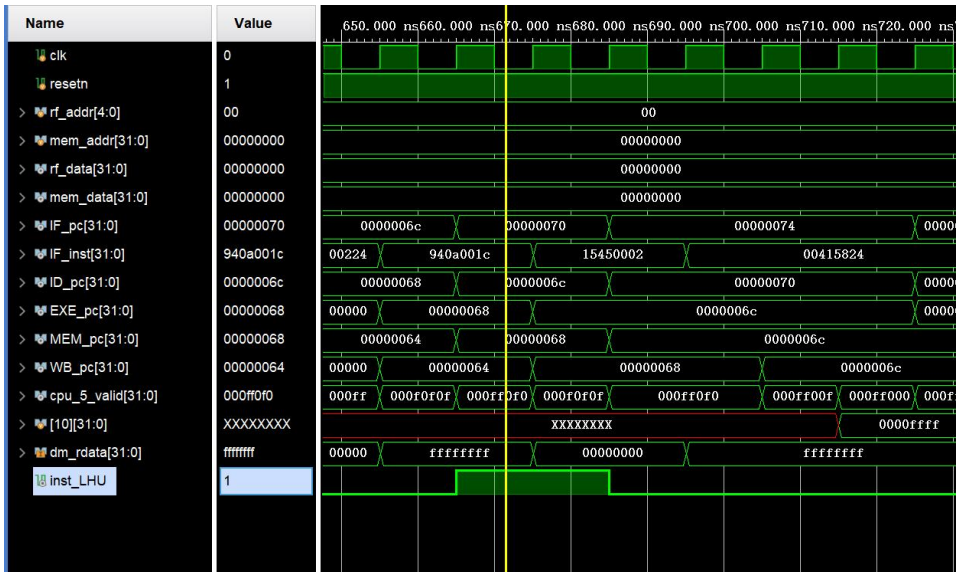
访存指令

读半字无符号扩展 LHU

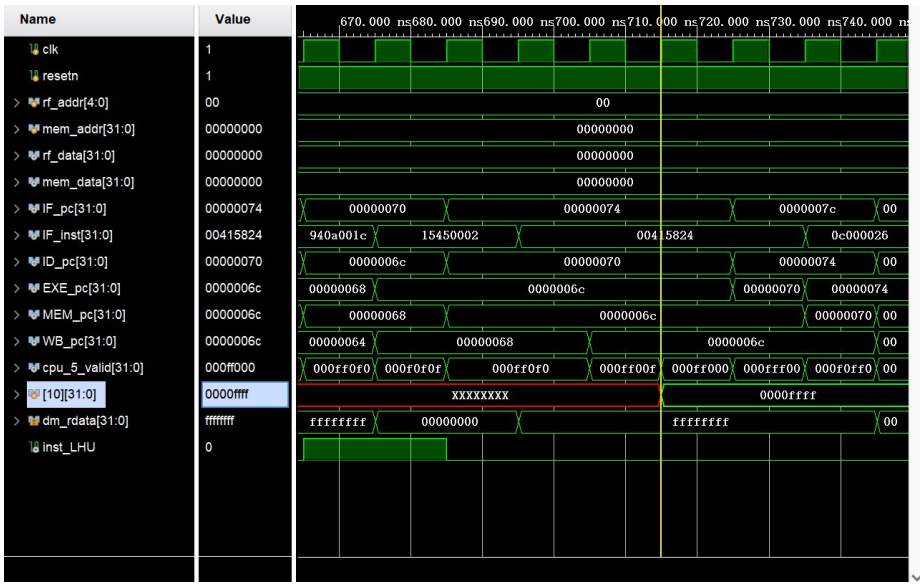
首先是我的 coe 文件，我这次测试使用的是原始的 coe 文件，并进行了一点修改，因为自己添加数据取读取与存储不太方便，如下图的 34 行，这个是我添加的读半字无符号扩展指令，前六位是 100101，刚好是我分配的 inst_LH 的编码，这一条指令的作用是读取地址为#28($0)的存储器的值，并将其保存在 10 号寄存器里：

```
31  11030002
32  AC08001C
33  0022482A
34  940a001C
35  15450002
```

如下图的仿照图，当这个读半字指令在译码阶段时，可以看见，inst_LHU 信号是 1，说明进行的就是读半字指令，没有问题：



如下图：



dm_rdata 信号展示的是读取的存储器的值，即 ffffffff，不过这并不是我要取的值，我只需要后 16 位，可以看见，当这个读半字的 wb 阶段结束以后，[10]，即 10 号寄存器的值变成了 0000ffff，正好是我要读取的后 16 位，而前 16 位用 0 填充。答案正确。

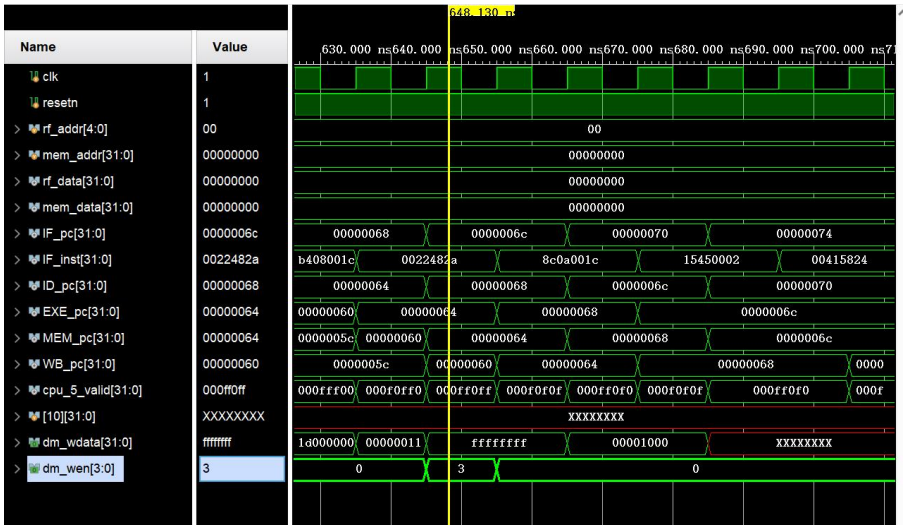
综上所述，添加的读半字无符号填充指令正常工作。

写入半字指令 SH

首先是我的 coe 文件，还是使用的是原始的 coe 文件，并进行了一点修改，如下图的 32 行，这个是我修改的写入指令，前六位是 101101，刚好是我分配的 inst_SH 编码 101101，这一条指令的作用是将 10 号寄存器里的值写入到地址为#28($0)的存储器，而第 34 行的指令会读取（完全读取，不是读半字）这个地址的值，我会结合这两个指令来验证我的指令的正确性：

```
32  b408001C
33  0022482A
34  8c0a001C
35  15450002
36  00415824
```

如下图，可以看见 dm_wdata,即要写入的值是 ffffffff，而写使能信号是 0011，即只写入后 16 位的值。到这里所有信号都是正确的没有问题，但是具体写入的是不是，还有往后看。



如下图，是 LW 指令，读取了相应地址的值，可以看见，dm_rdata，即从存储器里读取到的值就是 0000ffff,并且在 wb 阶段以后，写入到 10 号寄存器里的值也确实是 0000ffff，这就说明了原先存储地址里存的值就是 0000ffff，即 ffffffff 的后 16 位，答案是正确的：

综上所述，添加的写入半字指令正常工作。

六、 总结感想

通过本次实验，我深入理解了 CPU 流水线结构的工作机制及其存在的典型问题，并且掌握了如何通过设计测试程序发现和分析这些问题，如数据相关、结构冒险和控制冒险等。在调试过程中，我不仅理解了这些问题的成因，还学会了如何通过添加冒险检测逻辑、观察波形等方式定位错误。同时，在扩展指令集功能时，我尝试插入新的算术、访存和跳转类指令，并成功实现了它们的基本功能，这一过程让我更加熟悉 Verilog 的编写与调试，也加深了我对指令译码、控制信号生成和流水线阶段协作的理解。整个实验让我体会到了软硬件协同设计的重要性，也为今后深入学习 CPU 架构与数字系统设计打下了坚实的基础。