浙江大学

# 本科实验报告

课程名称：计算机体系结构

姓　　名：吴同

学　　院：计算机科学与技术学院

　　系：计算机科学与技术系

专　　业：计算机科学与技术

学　　号：3170104848

指导教师：陈文智

2019 年　　10 月　　16 日

# 浙江大学实验报告

专业：计算机科学与技术
姓名：吴同
学号：3170104848
日期：2019 年 10 月 16 日
地点：曹西 301

课程名称：计算机体系结构　指导老师：陈文智　电子邮件：wutongcs@zju.edu.cn
实验名称：带停顿的五级流水线 CPU 设计　实验类型：综合型　同组同学：徐欣苑

## 一、 实验目的和要求

### 1. 实验目的

- 理解流水线 CPU 的原理
- 理解流水线 CPU 的基本组成
- 理解五级流水的工作流程
- 理解流水线 CPU 停顿的原理
- 理解数据竞争的原理
- 掌握流水线 CPU 停顿检测的方法
- 掌握 CPU 内核模块的仿真
- 掌握带停顿的流水线 CPU 的程序验证方法

### 2. 实验要求

- 设计 CPU 控制器
- 基于 CPU 控制器，设计五级流水 CPU 的数据通路
- 设计五级流水 CPU 数据通路的停顿部分
- 增加停顿条件检测
- 仿真 CPU 内核模块
- 通过程序验证 CPU，观察程序的执行

## 二、 实验内容和原理

### 1. 实验内容

- 分析实验框架，补全代码，完成数据通路和控制器的设计
- 完成 CPU 内核模块的仿真
- 在 FPGA 上完成流水线 CPU 的硬件实现，并通过程序验证

## 2. 实验原理

图 1是五级流水 CPU 数据通路图。数据通路中各个阶段的逻辑如下：

**IF**

- 根据 PC 值从内存中读出指令。
- 将 PC+4 存到锁存器中。
- PC 的值根据控制信号，设置为 PC+4 或跳转到的地址。

**ID**

- 将 IF 阶段的指令地址、指令数据和下一条指令的地址存入 ID 阶段的锁存器中。
- 对指令进行译码，将控制信号存入锁存器中。
- 从寄存器堆中读出所取寄存器的值，和扩展后的立即数一并存入到锁存器中。
- 将写入寄存器的地址存到锁存器中。

**EXE**

- 将 ID 阶段的指令地址、指令数据、下一条指令地址、所取寄存器的值和控制信号存入 EXE 阶段的锁存器中。
- 根据控制信号，将运算数据送入 ALU 中，将计算结果写入到锁存器中。

**MEM**

- 将 EXE 阶段的各种数据存入 MEM 阶段的锁存器中。
- 根据控制信号和 ALU 计算结果，确定跳转地址的值，直接送回到 IF 阶段。
- 对内存数据进行读写，将从内存中取出的结果存入锁存器中。

**WB**

- 获取 MEM 阶段的 ALU 运算结果、从内存中取出的数据和控制信号，直接送回到 ID 阶段。

**停顿**

如果遇到数据竞争或跳转指令，则采取停顿的方式解决。

解决数据竞争的停顿的逻辑为，逐个判断指令所读取的寄存器，如果在 EXE 或 MEM 中的锁存器中的寄存器地址正是这一寄存器，且写寄存器的控制信号为高电平，则设置寄存器停顿。遇到这种停顿时，禁止 IF 和 ID 阶段的操作，并将 EXE 阶段的指令锁存器清零。

解决跳转指令的停顿的逻辑为，如果 ID、EXE 或 MEM 阶段中的任一一个指令是跳转指令且符合跳转条件，则设置跳转指令的停顿。遇到这种停顿时，将 ID 阶段的指令锁存器清零。

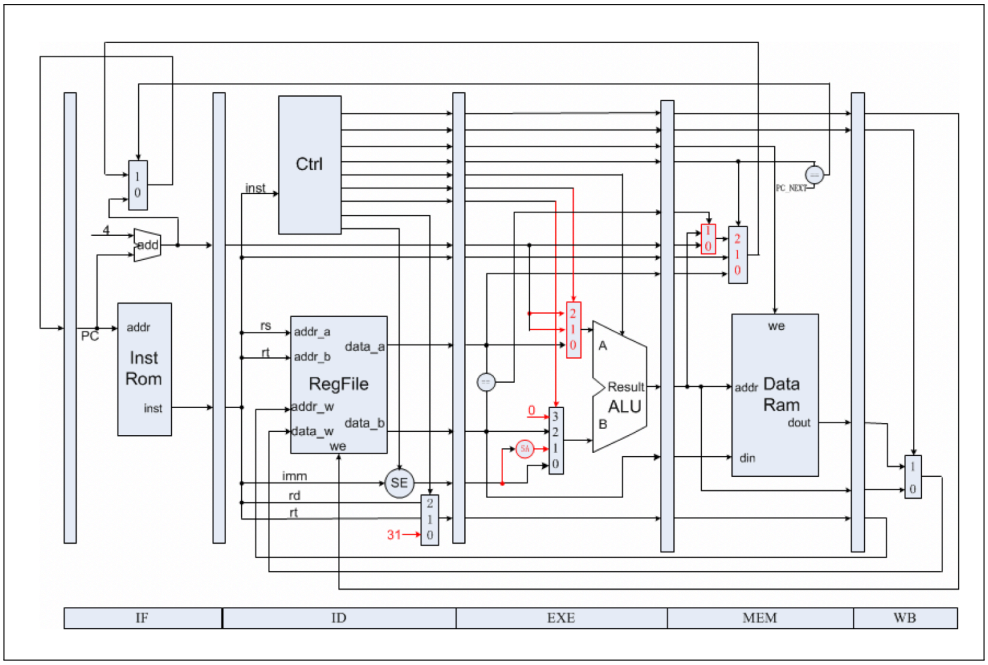


图 1: 五级流水数据通路图

## 三、 实验过程和数据记录

通过分析实验框架代码，对空缺部分进行如下的补全。

**controller.v**

```
case (inst[31:26])
   INST_R: begin
      case (inst[5:0])
         R_FUNC_JR: begin
            pc_src = PC_JR;
            rs_used = 1;
         end
         R_FUNC_ADD: begin
            exe_alu_oper = EXE_ALU_ADD;
            wb_addr_src = WB_ADDR_RD;
            wb_data_src = WB_DATA_ALU;
            wb_wen = 1;
            rs_used = 1;
            rt_used = 1;
         end
         R_FUNC_SUB: begin
            exe_alu_oper = EXE_ALU_SUB;
            wb_addr_src = WB_ADDR_RD;
            wb_data_src = WB_DATA_ALU;
            wb_wen = 1;
            rs_used = 1;
            rt_used = 1;
         end
         R_FUNC_AND: begin
            exe_alu_oper = EXE_ALU_AND;
            wb_addr_src = WB_ADDR_RD;
```

```
            wb_data_src = WB_DATA_ALU;
            wb_wen = 1;
            rs_used = 1;
            rt_used = 1;
         end
         R_FUNC_OR: begin
            exe_alu_oper = EXE_ALU_OR;
            wb_addr_src = WB_ADDR_RD;
            wb_data_src = WB_DATA_ALU;
            wb_wen = 1;
            rs_used = 1;
            rt_used = 1;
         end
         R_FUNC_SLT: begin
            exe_alu_oper = EXE_ALU_SLT;
            wb_addr_src = WB_ADDR_RD;
            wb_data_src = WB_DATA_ALU;
            wb_wen = 1;
            rs_used = 1;
            rt_used = 1;
         end
         default: begin
            unrecognized = 1;
         end
      endcase
   end
   INST_J: begin
      pc_src = PC_JUMP;
   end
   INST_JAL: begin
      pc_src = PC_JUMP;
      exe_a_src = EXE_A_LINK;
      exe_b_src = EXE_B_LINK;
      exe_alu_oper = EXE_ALU_ADD;
      wb_addr_src = WB_ADDR_LINK;
      wb_data_src = WB_DATA_ALU;
      wb_wen = 1;
   end
   INST_BEQ: begin
      pc_src = PC_BEQ;
      exe_a_src = EXE_A_BRANCH;
      exe_b_src = EXE_B_BRANCH;
      exe_alu_oper = EXE_ALU_ADD;
      imm_ext = 1;
      rs_used = 1;
      rt_used = 1;
   end
   INST_BNE: begin
      pc_src = PC_BNE;
      exe_a_src = EXE_A_BRANCH;
      exe_b_src = EXE_B_BRANCH;
      exe_alu_oper = EXE_ALU_ADD;
      imm_ext = 1;
      rs_used = 1;
```

```
            rt_used = 1;
        end
        INST_ADDI: begin
            imm_ext = 1;
            exe_b_src = EXE_B_IMM;
            exe_alu_oper = EXE_ALU_ADD;
            wb_addr_src = WB_ADDR_RT;
            wb_data_src = WB_DATA_ALU;
            wb_wen = 1;
            rs_used = 1;
        end
        INST_ANDI: begin
            imm_ext = 0;
            exe_b_src = EXE_B_IMM;
            exe_alu_oper = EXE_ALU_AND;
            wb_addr_src = WB_ADDR_RT;
            wb_data_src = WB_DATA_ALU;
            wb_wen = 1;
            rs_used = 1;
        end
        INST_ORI: begin
            imm_ext = 0;
            exe_b_src = EXE_B_IMM;
            exe_alu_oper = EXE_ALU_OR;
            wb_addr_src = WB_ADDR_RT;
            wb_data_src = WB_DATA_ALU;
            wb_wen = 1;
            rs_used = 1;
        end
        INST_LW: begin
            imm_ext = 1;
            exe_b_src = EXE_B_IMM;
            exe_alu_oper = EXE_ALU_ADD;
            mem_ren = 1;
            wb_addr_src = WB_ADDR_RT;
            wb_data_src = WB_DATA_MEM;
            wb_wen = 1;
            rs_used = 1;
        end
        INST_SW: begin
            imm_ext = 1;
            exe_b_src = EXE_B_IMM;
            exe_alu_oper = EXE_ALU_ADD;
            mem_wen = 1;
            rs_used = 1;
            rt_used = 1;
        end
        default: begin
            unrecognized = 1;
        end
    endcase

    always @(*) begin
        reg_stall = 0;
```

```
    if (rs_used && addr_rs != 0) begin
       if (regw_addr_exe == addr_rs && wb_wen_exe) begin
          reg_stall = 1;
       end
       else if (regw_addr_mem == addr_rs && wb_wen_mem) begin
          reg_stall = 1;
       end
    end
    if (rt_used && addr_rt != 0) begin
       if (regw_addr_exe == addr_rt && wb_wen_exe) begin
          reg_stall = 1;
       end
       else if (regw_addr_mem == addr_rt && wb_wen_mem) begin
          reg_stall = 1;
       end
    end
 end

 always @(*) begin
    branch_stall = 0;
    if (pc_src != PC_NEXT || is_branch_mem || is_branch_exe)
       branch_stall = 1;
 end
```

**datapath.v**

```
 always @(*) begin
    regw_addr_id = inst_data_id[15:11];
    case (wb_addr_src_ctrl)
       WB_ADDR_RD: regw_addr_id = addr_rd;
       WB_ADDR_RT: regw_addr_id = addr_rt;
       WB_ADDR_LINK: regw_addr_id = GPR_RA;
    endcase
 end

 always @(*) begin
    opa_exe = data_rs_exe;
    opb_exe = data_rt_exe;
    case (exe_a_src_exe)
       EXE_A_RS: opa_exe = data_rs_exe;
       EXE_A_LINK: opa_exe = inst_addr_next_exe;
       EXE_A_BRANCH: opa_exe = inst_addr_next_exe;
    endcase
    case (exe_b_src_exe)
       EXE_B_RT: opb_exe = data_rt_exe;
       EXE_B_IMM: opb_exe = data_imm_exe;
       EXE_B_LINK: opb_exe = 32'b0; // linked address is the next one
           of current instruction
       EXE_B_BRANCH: opb_exe = {data_imm_exe[29:0], 2'b0};
    endcase
 end

 always @(*) begin
    case (pc_src_mem)
```

```
        PC_JUMP: branch_target_mem <= {inst_addr_mem[31:28],
            inst_data_mem[25:0], 2'b0};
        PC_JR: branch_target_mem <= data_rs_mem;
        PC_BEQ: branch_target_mem <= rs_rt_equal_mem ? alu_out_mem :
            inst_addr_next_mem;
        PC_BNE: branch_target_mem <= rs_rt_equal_mem ?
            inst_addr_next_mem : alu_out_mem;
        default: branch_target_mem <= inst_addr_next_mem; // will
            never used
    endcase
end

always @(*) begin
    regw_data_wb = alu_out_wb;
    case (wb_data_src_wb)
        WB_DATA_ALU: regw_data_wb = alu_out_wb;
        WB_DATA_MEM: regw_data_wb = mem_din_wb;
    endcase
end
```

完成 CPU 控制器和数据通路的设计后，首先进行仿真验证，如图 2所示。图 3~ 图 7是仿真结果的详细情况

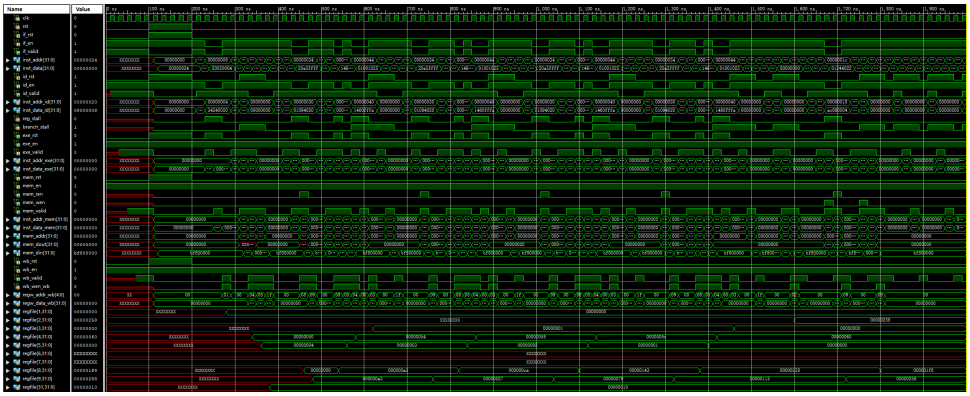

图 2: 仿真测试结果图（0~2000ns）

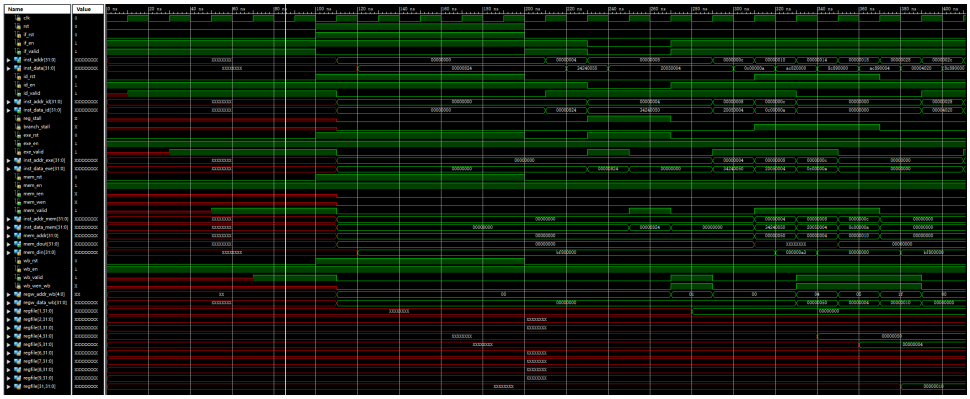

图 3: 仿真测试结果图（0~400ns）

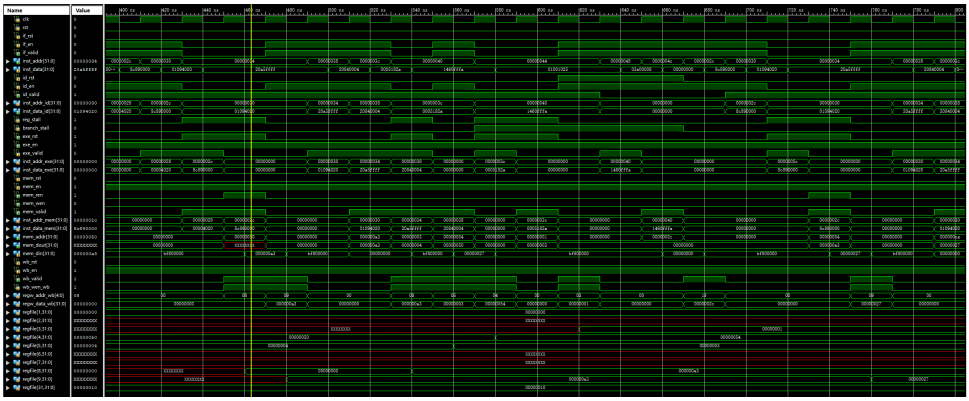

图 4: 仿真测试结果图（400∼800ns）

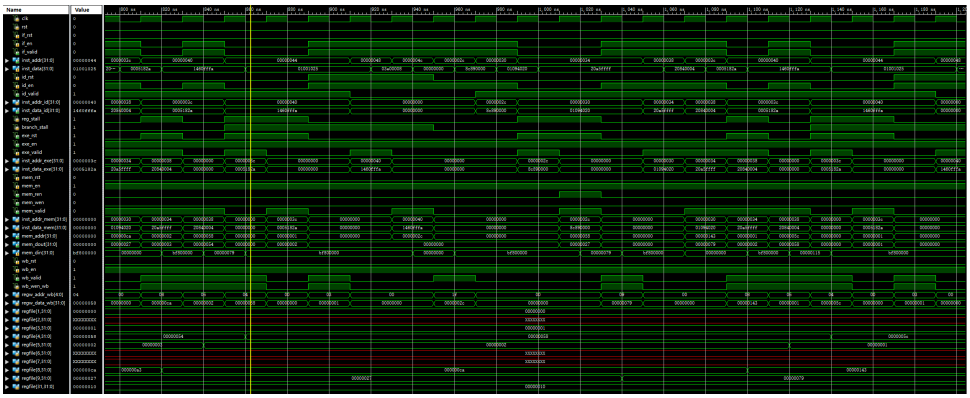

图 5: 仿真测试结果图（800∼1200ns）

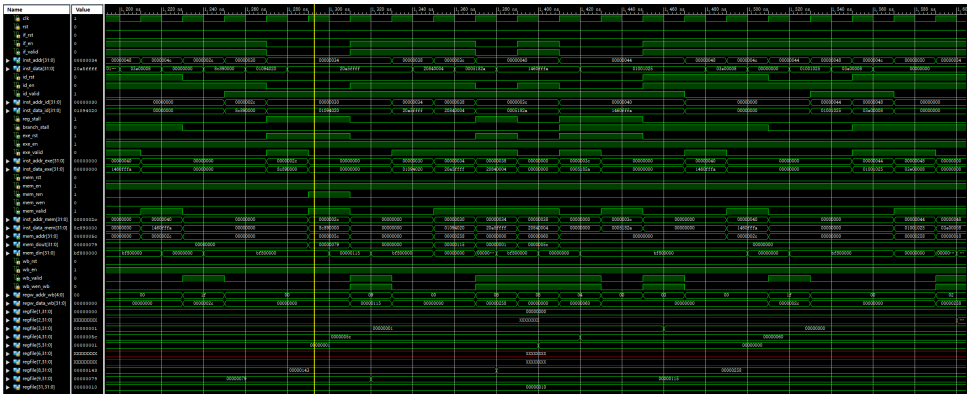

图 6: 仿真测试结果图（1200∼1600ns）

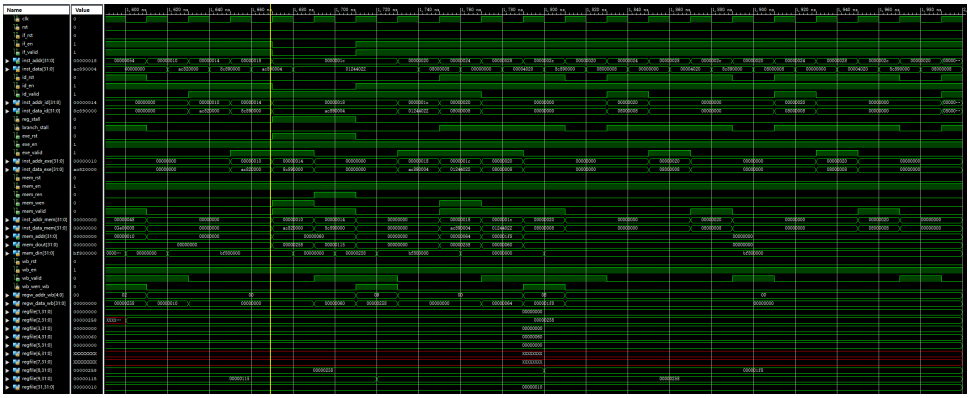

图 7: 仿真测试结果图（1600~2000ns）

仿真测试通过后，综合整个工程，下载到 SWORD 4.0 开发板上。开发板上可以选择正常时钟或单步调试。正常执行时，观察到显示屏上 IF 的地址和指令数据会停在一个值，每次重置 CPU 后其停在的值都不同。其原因是对于流水线 CPU，取指这一阶段会一直进行下去，之所以在 VGA 显示屏上观察不到变化，是因为显示器的刷新频率恰好等于此时 IF 值的变化频率，所以观察不到变化。

## 四、 实验结果分析

图 8是一处数据竞争导致停顿的实例。IF 取到 0xAC890004 指令，需要读取 9 号寄存器的值。此时 ID 指令为 0x8C890000，要向 9 号寄存器中写入，这引发了数据竞争。reg_stall 信号触发后，IF 不能继续取指，ID 不能从 IF 获取指令，EXE 指令在下一个节拍被置零，原本应为 0xAC890004。相应地，再下一个节拍的 MEM 指令也被置零。随后流水线恢复正常。

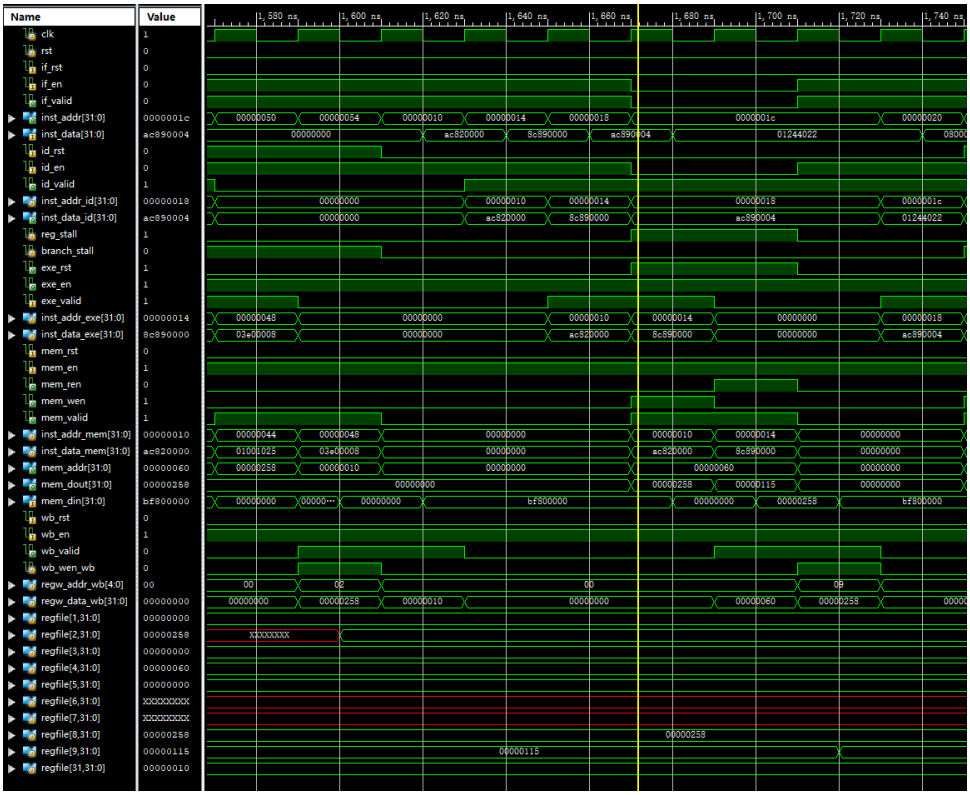


图 8: reg_stall 示例

图 9是一处跳转导致停顿的实例。IF 所取的指令为 0x08000008，为跳转指令。ID 指令被清零，相应地 EXE 和 MEM 指令被依次清零。随后流水线恢复正常。

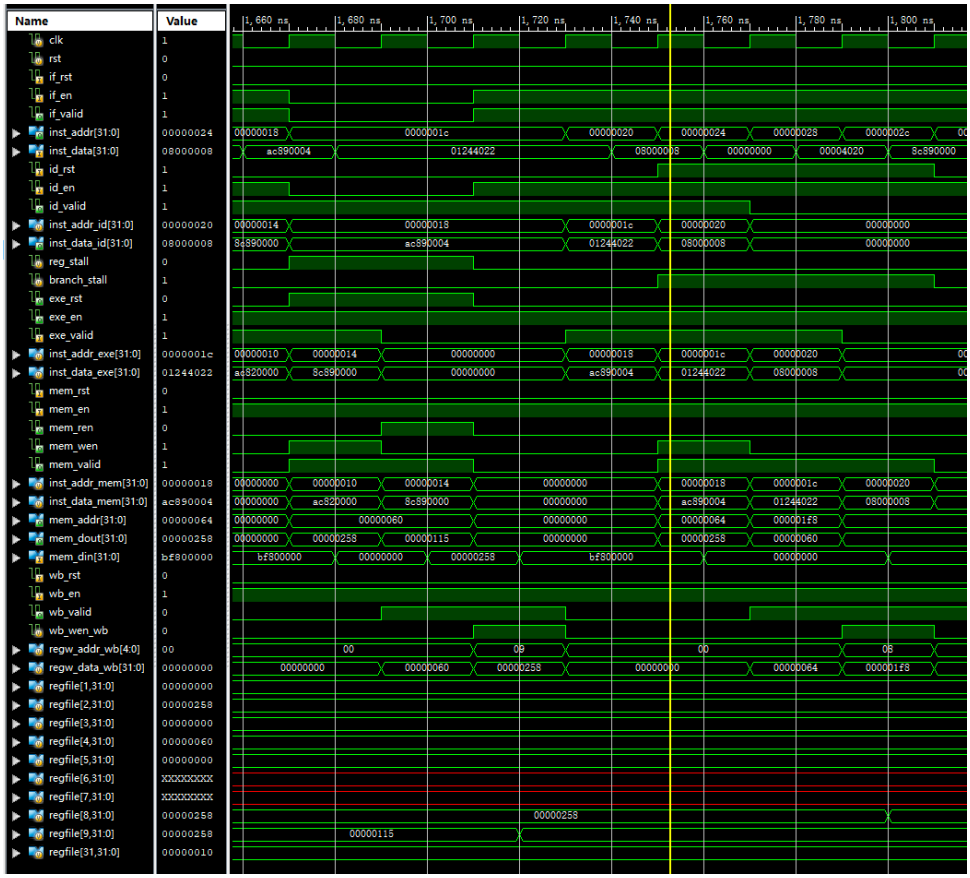


图 9: branch_stall 示例

## 五、 讨论与心得

本次实验是设计五级流水 CPU，实验在所给的框架下进行代码补全。通过分析所给的实验框架，我对流水线 CPU 的原理有了初步的认识。在计算机组成课程中，我接触了多周期 CPU 的设计。多周期 CPU 是将一条指令分成不同阶段来执行，这种分阶段执行的设计有利于流水的实现。在本次实验的流水线设计中，CPU 的各个部件在同一时间都得到了利用，这样 CPU 的执行效率得到了很大的提高。

在流水线 CPU 的设计中，会遇到数据竞争的问题，即后一条指令要读前一条指令写入寄存器的值。或遇到跳转指令，流水线不能正常取指令。为了解决这种问题，流水线中引入了停顿。这一设计保证了指令执行的正确性，但不能保证执行的效率。

在实验过程中，我遇到了执行结果错误的问题。在程序执行完毕后，8 号寄存器和 9 号寄存器的执行结果与理论值不符。在上次实验中，我对验证程序的执行过程进行了单步分析。通过对比程序执行情况与分析结果，我发现了问题出在执行 BNE 指令时，应该进行停顿时没有停顿。对相关的代码进行分析，发现问题不是出在逻辑，而是做相等判断时，将 1 位的信号直接与数字 1 作比较，这两者在 Verilog 中存在位宽不同的问题，比较结果恒为 flase。将这一问题解决后，程序的执行结果与理论值相同。