Arch Lab 6

MIPS 多周期流水线处理器的设计与实现

Alex Chi

日期：2020 年 5 月 15 日

目录

1 简介 2

2 实验目的 2

3 MIPS CPU 设计 2

4 从单周期到流水线：阶段的拆分 3

4.1 五级流水线 . . . 3

4.2 IF 阶段 . . . 3

4.2.1 IF 阶段数据流 . . . 3

4.2.2 IF 阶段控制流 . . . 4

4.3 ID 阶段 . . . 5

4.3.1 ID 阶段输入与输出 . . . 5

4.3.2 Forwarding 模块的接入 . . . 6

4.3.3 ID 阶段数据流：解码 . . . 6

4.3.4 ID 阶段数据流：读寄存器 . . . 7

4.3.5 ID 阶段数据流：分支 . . . 7

4.3.6 ID 阶段数据流：Forwarding . . . 8

4.3.7 ID 阶段数据流：内存 . . . 8

4.3.8 ID 阶段数据流：ALU . . . 8

4.3.9 ID 阶段数据流：其他数据 . . . 9

4.3.10 ID 阶段控制流 . . . 9

4.4 Forwarding 的实现 . . . 10

4.4.1 Forwarding 的数据流 . . . 10

4.4.2 Forwarding 的输入输出 . . . 10

4.4.3 Forwarding 的逻辑实现 . . . 10

4.5 EX 阶段 . . . 12

4.5.1 EX 阶段数据流 . . . 12

4.5.2 EX 阶段控制流 13
4.6 MEM 阶段 14
4.6.1 MEM 阶段数据流 14
4.6.2 MEM 阶段控制流 15
4.7 WB 阶段 15

5 Hazard 的处理 16
5.1 Data Hazard 16
5.2 Control Hazard 16

6 测试 16
6.1 基本功能测试 16
6.2 Hazard 测试 19

7 总结与感想 22

1 简介

在本次实验中，我完成了下面的任务。

- 在单周期 MIPS CPU 的基础上，完成了多周期流水线 MIPS CPU。
- 完成了一个支持几乎所有 MIPS 指令 (36 个) 的多周期 MIPS CPU。
- 通过汇编器自己编写了 3 个程序，测试 CPU 中的 Hazard，通过 Vivado 进行测试，和汇编器结果比较。

2 实验目的

- 完成多周期的类 MIPS 处理器。
- 通过 Forwarding 处理 Data Hazard。
- 通过 Predict-Not-Taken 处理 Control Hazard。
- 设计支持 36 条 MIPS 指令（具体指令见下文）的多周期 CPU。

3 MIPS CPU 设计

本次实验中，我编写的多周期 MIPS CPU 支持下面这些指令。

R-type	add	addu	and	noop	or	sllv	slt	sltu		srlv	subu	sub	xor
Immediate	addi	addiu	andi	lui	ori	sll	slti	sltiu	sra	srl			xori
Branch	beq	bgez	bgtz	blez	bltz	bne							
Jump and Link	j	jal	jr										
Memory	lb	lw	sb	sw									

4 从单周期到流水线：阶段的拆分

4.1 五级流水线

CPU 大致可以拆为五个阶段。它们分别是：Instruction Fetch，Instruction Decode，Execute，Memory，Write Back。

其中，各个阶段的任务是：

- IF: 从 Instruction Memory 中获取一条指令。
- ID：对指令进行解码，决定之后几个周期元器件的执行操作。
- EX：通过 ALU 进行计算。如果是分支指令，在这一阶段可以决定是否跳转。
- MEM：读写内存。
- WB：把数据写回 Register File。

在之前设计的单周期处理器中，数据流有复杂的依赖关系。理想中的拆分，是可以直接在两个阶段间加入阶段寄存器。但是单周期处理器中的数据流有跨越两级依赖的情况，很难拆分为五级流水线。这样的问题主要体现在：

1. 各个阶段都需要 ID 阶段生成的 opcode。
2. WB, ID 阶段都需要用到 RegisterFile。
3. WB, MEM 阶段都需要用到 ALU 的值。
4. IF 阶段需要 EX 阶段产生的跳转结果。

通过重新设计数据流和阶段寄存器，可以很好地解决这些依赖问题，从而完成一个五级流水线。

4.2 IF 阶段

4.2.1 IF 阶段数据流

```
module InstFetch(
    input [`WORD] if_pc,
    output [`WORD] inst,
    output [`WORD] pc,
    output [`WORD] next_pc
);
    wire [`WORD] imem_addr = if_pc;
    InstMemory imem(
        .address (imem_addr),
        .readData (inst)
    );
    assign pc = if_pc;
    assign next_pc = pc + 4;
endmodule
```

IF 阶段接收控制线上的 PC，从 Inst Memory 中读取指令，并写入阶段寄存器中。

4.2.2 IF 阶段控制流

IF 阶段的输入只有一个，就是 `if_pc`。它决定了指令的地址。如果没有 Control Hazard，它就是下一条指令。否则，则是应该跳转的地址。

IF 阶段有这些阶段寄存器。

寄存器	位数	内容
stage_if_inst	32	指令内容
stage_if_pc	32	指令对应 PC
stage_if_branch_taken	1	是否预测跳转

```
// --- STAGE ---
//   InstFetch

// --- INPUT ---
wire [`WORD] if_pc = !correct_branch_prediction ? branch_jump_target : pc;

// --- STAGE REGS ---
reg [`WORD] stage_if_inst;
reg [`WORD] stage_if_pc;
reg stage_if_branch_taken;

// --- OUTPUT ---
wire [`WORD] out_if_next_pc;
wire [`WORD] out_if_inst;
wire [`WORD] out_if_pc;

InstFetch instFetch(
    .if_pc (if_pc),
    .inst (out_if_inst),
    .pc (out_if_pc),
    .next_pc (out_if_next_pc)
);

wire out_id_stall;
wire [`WORD] if_next_pc = out_id_stall ? if_pc : out_if_next_pc;

// --- CLOCK ---
always @ (negedge clk) begin
    if (!out_id_stall) begin
        stage_if_inst <= out_if_inst;
        stage_if_pc <= out_if_pc;
        stage_if_branch_taken <= 0; // branch prediction: always not taken
        pc <= if_next_pc;
    end
end
```

4.3 ID 阶段

4.3.1 ID 阶段输入与输出

在 ID 阶段，我们需要用到 CPU 中的 RegisterFile。由于这一模块在 CPU 中，而不在 InstDecode 模块中，我们需要通过引脚将它接入 ID 模块。

```
module InstDecode(
    // 其他输入输出省略
    // MODULE: RegisterFile
    output wire [`REG] rf_src1,
    output wire [`REG] rf_src2,
    input wire [`WORD] rf_out1_prev,
    input wire [`WORD] rf_out2_prev
    // 其他输入输出省略
);
```

ID 阶段的输入输出包括上一级的阶段寄存器，和传递给 EX 阶段的寄存器。这些寄存器如下。

寄存器	位数	内容
stage_id_alu_op	6	ALU 操作 opcode
stage_id_alu_src1	32	ALU 操作数 1
stage_id_alu_src2	32	ALU 操作数 2
stage_id_opcode	1	当前指令的 opcode
stage_id_pc	32	当前指令的 PC
stage_id_alu_branch_mask	1	跳转正确的预计 ALU 输出
stage_id_branch_pc	32	跳转 PC
stage_id_next_pc	32	下一条指令的 PC
stage_id_rf_dest	5	目标寄存器
stage_id_mem_data	32	写入内存的数据
stage_id_branch_taken	1	IF 是否预测跳转
stage_id_force_jump	1	是否是强制跳转指令

```
module InstDecode(
    // 其他输入输出省略
    input [`WORD] inst,
    input [`WORD] if_pc,
    input if_branch_taken,
    output [`OP] alu_op,
    output [`WORD] alu_src1,
    output [`WORD] alu_src2,
    output [`OP] opcode,
    output [`WORD] id_pc,
    output alu_branch_mask,
    output [`WORD] branch_pc,
```

```
    output [`WORD] next_pc,
    output [`REG] rf_dest,
    output [`WORD] mem_data,
    output id_branch_taken,
    output force_jump,
    output stall,
    // 其他输入输出省略
);
```

4.3.2 Forwarding 模块的接入

ID 阶段可以通过将其他阶段的输出 forward 到这一阶段。这由一个 CPU 中专门的 Forwarding 单元实现。我们把两个 Forwarding 单元的结果输入 ID 阶段。

```
module InstDecode(
    // 其他输入输出省略
    // MODULE: Forward
    input forward_depends_1,
    input forward_depends_2,
    input forward_stalls_1,
    input forward_stalls_2,
    input [`WORD] forward_result_1,
    input [`WORD] forward_result_2,
    output wire [`REG] forward_op1,
    output wire [`REG] forward_op2,
    // 其他输入输出省略
);
```

4.3.3 ID 阶段数据流：解码

而后，我们可以把单周期处理器中的 ID 阶段剥离出来。

在解码部分的处理和之前一样，只需要把指令中各个位置的数据取出来即可。

```
wire [`WORD] pc = if_pc;
assign opcode = inst[31:26];
wire [`REG] rs = inst[25:21];
wire [`REG] rt = inst[20:16];
wire [`REG] rd = inst[15:11];
wire [`REG] shamt = inst[10:6];
wire [`OP] funct = inst[`OP];
wire [15:0] imm = inst[15:0];
wire [`WORD] imm_sign_ext;
wire [`WORD] imm_zero_ext;
wire [`WORD] shamt_zero_ext = {{27'b0}, shamt};
SignExt signExt(.unextended (imm), .extended (imm_sign_ext));
ZeroExt zeroExt(.unextended (imm), .extended (imm_zero_ext));
```

```
wire is_shift;
IsShift isShift(.funct (funct), .shift (is_shift));
wire is_type_R = (opcode == 0);
wire use_shamt = is_shift && is_type_R;
wire [`WORD] jump_target = {4'b00, inst[25:0], 2'b00} | (pc & 32'hf0000000);
wire is_branch;
wire is_memory;
```

4.3.4 ID 阶段数据流：读寄存器

之后，我们要获取寄存器中对应的值。

```
// MODULE: Register File
assign rf_src1 = rs;
assign rf_src2 = is_type_R || is_branch || is_memory_store ? rt : 0;
assign rf_dest =  is_type_R ? rd : (
                        opcode == 3 ? 31 : rt);
wire [`WORD] rf_out1 = forward_depends_1 ? forward_result_1 : rf_out1_prev;
wire [`WORD] rf_out2 = override_rt ? branch_alu_rt_val :
                        (forward_depends_2 ? forward_result_2 : rf_out2_prev);
```

4.3.5 ID 阶段数据流：分支

而后，我们要为 ALU 阶段的分支判断做好准备。

```
// MODULE: Branch
wire [`WORD] imm_offset = imm_sign_ext <<< 2;
assign branch_pc = pc + 4 + imm_offset;
assign next_pc = (opcode == 2 || opcode == 3) ? jump_target :
                        (opcode == 0 && funct == 8 ? rf_out1 : pc + 4);
wire override_rt;
wire [`WORD] branch_alu_rt_val;
BranchOp branchOp(
    .opcode (opcode),
    .branch_op (is_branch),
    .override_rt (override_rt),
    .rt_val (branch_alu_rt_val)
);
BranchOut branchOut(
    .opcode (opcode),
    .rt (rt),
    .alu_branch_mask (alu_branch_mask)
);
assign force_jump = opcode == 2 || opcode == 3 || (opcode == 0 && funct == 8);
```

4.3.6 ID 阶段数据流：Forwarding

如果数据可以通过 Forwarding 而非从寄存器中读取的方式获得，我们需要用最新的 Forwarding 的数据。如果出现了 load-use hazard，我们还需要停下流水线。

```
// MODULE: Forward
assign forward_op1 = rf_src1;
assign forward_op2 = rf_src2;
wire alu_use_rf_out_1;
wire alu_use_rf_out_2;
assign stall = (alu_use_rf_out_1 && forward_stalls_1) ||
                (alu_use_rf_out_2 && forward_stalls_2) ||
                (is_memory_store && forward_stalls_2);
```

4.3.7 ID 阶段数据流：内存

在这一阶段，我们也要为之后的内存操作做好准备。比如确定内存中存入的数据等。

```
// MODULE: Memory
wire [`OP] mapped_op;
/* verilator lint_off PINMISSING */
ALUOp aluOp(
    .opcode (opcode),
    .ALUopcode (mapped_op));
/* verilator lint_on PINMISSING */
wire is_memory_load;
wire is_memory_store;
wire [2:0] memory_mode;
MemoryOp memoryOp(
    .opcode (opcode),
    .store (is_memory_store),
    .load (is_memory_load),
    .memory_op (is_memory),
    .memory_mode (memory_mode));
assign mem_data = forward_depends_2 ? forward_result_2 : rf_out2;
```

4.3.8 ID 阶段数据流：ALU

在 EX 阶段中，ALU 所需要的操作数也应当在这一阶段生成。

```
// MODULE: ALU
wire ext_mode;
ExtMode extMode (
    .opcode (opcode),
    .signExt (ext_mode));
assign alu_op = is_type_R ? funct : mapped_op;
wire [`WORD] alu_imm = ext_mode ? imm_sign_ext : imm_zero_ext;
```

```
assign alu_use_rf_out_1 = !use_shamt && opcode != 3;
assign alu_use_rf_out_2 = is_type_R || is_branch;
assign alu_src1 = use_shamt ? shamt_zero_ext :
                        (opcode == 3 ? pc : rf_out1);
assign alu_src2 = alu_use_rf_out_2 ? rf_out2 :
                        (opcode == 3 ? 4 : alu_imm);
```

4.3.9 ID 阶段数据流：其他数据

IF 阶段是否跳转，以及这条指令对应的 PC，在之后也会被用到。ID 阶段只需要不加处理直接传递即可。

```
// Other Modules
assign id_pc = pc;
assign id_branch_taken = if_branch_taken;
```

4.3.10 ID 阶段控制流

ID 阶段可能会被 stall 或 bubble。在这种情况下，我们需要在分支预测错误或是需要 stall 的时候清空阶段寄存器的值 (bubble)。

```
always @ (negedge clk) begin
    if (!correct_branch_prediction || out_id_stall) begin
        // Bubble
        stage_id_alu_op <= 0;
        stage_id_alu_src1 <= 0;
        stage_id_alu_src2 <= 0;
        stage_id_opcode <= 0;
        stage_id_pc <= 0;
        stage_id_alu_branch_mask <= 0;
        stage_id_branch_pc <= 0;
        stage_id_next_pc <= 0;
        stage_id_rf_dest <= 0;
        stage_id_mem_data <= 0;
        stage_id_branch_taken <= 0;
        stage_id_force_jump <= 0;
    end else begin
        stage_id_alu_op <= out_id_alu_op;
        stage_id_alu_src1 <= out_id_alu_src1;
        stage_id_alu_src2 <= out_id_alu_src2;
        stage_id_opcode <= out_id_opcode;
        stage_id_pc <= out_id_pc;
        stage_id_alu_branch_mask <= out_id_alu_branch_mask;
        stage_id_branch_pc <= out_id_branch_pc;
        stage_id_next_pc <= out_id_next_pc;
        stage_id_rf_dest <= out_id_rf_dest;
```

```
        stage_id_mem_data <= out_id_mem_data;
        stage_id_branch_taken <= out_id_branch_taken;
        stage_id_force_jump <= out_id_force_jump;
    end
end
```

4.4 Forwarding 的实现

4.4.1 Forwarding 的数据流

Forwarding 模块起到的作用很简单。就是根据所需要的寄存器，收集 EX、MEM、WB 阶段的信息，并作出对应的反应。

- 如果 EX 阶段的目标寄存器和所需要的寄存器相同，且是一个数值运算指令则 forward。
- 如果 EX 阶段的目标寄存器和所需要的寄存器相同，且是内存指令则 stall。这是 load-use hazard。
- 如果 MEM 阶段的目标寄存器和所需要的寄存器相同，则 forward。如果是内存指令，forward 内存结果。如果是数值指令，forward ALU 结果。
- 如果 WB 阶段的目标寄存器和所需要的寄存器相同，则 forward。

4.4.2 Forwarding 的输入输出

Forward 阶段需要获取各个阶段的操作数。因此它的输入包括每个阶段的 `opcode`, `dest`, `val`。

```
module Forward(
    input [`OP] ex_opcode,
    input [`REG] ex_dest,
    input [`WORD] ex_val,
    input [`OP] mem_opcode,
    input [`REG] mem_dest,
    input [`WORD] mem_alu_val,
    input [`WORD] mem_val,
    input [`OP] wb_opcode,
    input [`REG] wb_dest,
    input [`WORD] wb_val,
    input [`REG] src,
    output reg [`WORD] data,
    output reg depends,
    output reg stall
);
```

4.4.3 Forwarding 的逻辑实现

通过组合电路逻辑，我们可以得到三个信号：

- `stall`: 是否需要暂停流水。
- `data`: 得到的数据。
- `depends`: 所需要的寄存器是否可以 forward。

```
wire ex_is_arithmetic_op;
wire ex_is_link_op = ex_opcode == 3;
wire ex_is_memory_load;
wire mem_is_arithmetic_op;
wire mem_is_link_op = mem_opcode == 3;
wire mem_is_memory_load;
wire wb_is_arithmetic_op;
wire wb_is_link_op = wb_opcode == 3;
wire wb_is_memory_load;

/* verilator lint_off PINMISSING */
ALUOp ex_aluop(.opcode (ex_opcode), .arithmetic_op (ex_is_arithmetic_op));
ALUOp mem_aluop(.opcode (mem_opcode), .arithmetic_op (mem_is_arithmetic_op));
ALUOp wb_aluop(.opcode (wb_opcode), .arithmetic_op (wb_is_arithmetic_op));

MemoryOp ex_memop(.opcode (ex_opcode), .load (ex_is_memory_load));
MemoryOp mem_memop(.opcode (mem_opcode), .load (mem_is_memory_load));
MemoryOp wb_memop(.opcode (wb_opcode), .load (wb_is_memory_load));
/* verilator lint_on PINMISSING */

always @(*) begin
    data = 0;
    depends = 0;
    stall = 0;
    if (src == 0) begin
        data = 0;
        depends = 0;
        stall = 0;
    end else if (ex_dest == src && (ex_is_arithmetic_op || ex_is_link_op)) begin
        data = ex_val;
        stall = 0;
        depends = 1;
    end else if (ex_dest == src && ex_is_memory_load) begin
        data = 0;
        stall = 1;
        depends = 1;
    end else if (mem_dest == src && (mem_is_arithmetic_op || mem_is_link_op)) begin
        data = mem_alu_val;
        stall = 0;
        depends = 1;
    end else if (mem_dest == src && mem_is_memory_load) begin
        data = mem_val;
        stall = 0;
```

```
        depends = 1;
    end else if (wb_dest == src && (wb_is_arithmetic_op || wb_is_memory_load ||
        wb_is_link_op)) begin
        data = wb_val;
        stall = 0;
        depends = 1;
    end
end
```

4.5 EX 阶段

4.5.1 EX 阶段数据流

EX 阶段通过 ID 阶段提供的信息，进行具体的数值运算。它将这些内容传递给 MEM 阶段。

寄存器	位数	内容
stage_ex_alu_out	32	ALU 输出
stage_ex_opcode	6	指令 opcode
stage_ex_pc	32	指令对应 PC
stage_ex_rf_dest	5	ID 阶段写入寄存器编号
stage_ex_mem_data	32	ID 阶段写入内存的值

这些输出和上一级的输入构成了 EX 阶段的引脚。

```
module Execute(
    input [`OP] alu_op,
    input [`WORD] alu_src1,
    input [`WORD] alu_src2,
    input [`OP] id_opcode,
    input [`WORD] id_pc,
    input alu_branch_mask,
    input [`WORD] branch_pc,
    input [`WORD] next_pc,
    input [`REG] id_rf_dest,
    input [`WORD] id_mem_data,
    input id_branch_taken,
    input force_jump,
    output [`WORD] alu_out,
    output [`OP] ex_opcode,
    output [`WORD] ex_pc,
    output [`REG] ex_rf_dest,
    output [`WORD] ex_mem_data,
    output correct_branch_prediction,
    output [`WORD] branch_jump_target
);
```

在 EX 阶段，需要通过 ALU 计算结果，并确定分支预测的结果是否正确。

ALU 计算部分如下。

```
assign ex_mem_data = id_mem_data;
assign ex_rf_dest = id_rf_dest;
assign ex_opcode = id_opcode;

wire alu_zero;

ALU alu (
        .ALUopcode (alu_op),
        .op1 (alu_src1),
        .op2 (alu_src2),
        .out (alu_out),
        .zero (alu_zero));
```

分支判断部分如下。

```
wire is_branch;
/* verilator lint_off PINMISSING */
BranchOp branchOp(
    .opcode (id_opcode),
    .branch_op (is_branch)
);
/* verilator lint_on PINMISSING */

wire take_branch = is_branch && (alu_zero ^ alu_branch_mask);
assign ex_pc = take_branch ? branch_pc : next_pc;

assign correct_branch_prediction =
    !((take_branch != id_branch_taken) || force_jump);

assign branch_jump_target = (take_branch != id_branch_taken) ? ex_pc :
    (force_jump ? next_pc : 0);
```

4.5.2 EX 阶段控制流

EX 阶段不存在 stall 的情况。如果出现分支预测错误的情况，branch 指令进入下一级流水线也不会存在副作用。因此，不会 bubble。所以控制十分简单，只需要把输出存入寄存器即可。

```
always @ (negedge clk) begin
    stage_ex_alu_out <= out_ex_alu_out;
    stage_ex_opcode <= out_ex_opcode;
    stage_ex_pc <= out_ex_pc;
    stage_ex_rf_dest <= out_ex_rf_dest;
    stage_ex_mem_data <= out_ex_mem_data;
end
```

4.6 MEM 阶段

4.6.1 MEM 阶段数据流

MEM 阶段根据之前阶段的输出从内存中读取、或向内存写入数据。

寄存器	位数	内容
stage_mem_pc	32	指令对应 PC
stage_mem_out	32	内存读取结果
stage_mem_alu_out	32	EX 阶段 ALU 运算结果
stage_mem_opcode	6	指令 opcode
stage_mem_rf_dest	5	ID 阶段写入寄存器

由于 DataMemory 不在 MEM 阶段模块中，需要通过引脚连接。

```
module Memory(
    input [`WORD] ex_alu_out,
    input [`OP] ex_opcode,
    input [`WORD] ex_pc,
    input [`REG] ex_rf_dest,
    input [`WORD] ex_mem_data,
    output [`WORD] mem_pc,
    output [`WORD] mem_out,
    output [`WORD] mem_alu_out,
    output [`OP] mem_opcode,
    output [`REG] mem_rf_dest,
    // MODULE: Data Memory
    input [`WORD] dmem_out,
    output [`WORD] dmem_addr,
    output [`WORD] dmem_in,
    output dmem_write,
    output dmem_read,
    output [2:0] dmem_mode
);
```

而 MEM 阶段的逻辑只需要进行连线即可。

```
assign mem_pc = ex_pc;
assign mem_out = dmem_out;
assign mem_alu_out = ex_alu_out;
assign mem_opcode = ex_opcode;
assign mem_rf_dest = ex_rf_dest;

/* verilator lint_off PINMISSING */
MemoryOp memoryOp(
    .opcode (ex_opcode),
    .store (dmem_write),
    .load (dmem_read),
```

```
    .memory_mode (dmem_mode));
/* verilator lint_on PINMISSING */
assign dmem_addr = ex_alu_out;
assign dmem_in = ex_mem_data;
```

4.6.2 MEM 阶段控制流

MEM 阶段仅需要刷新阶段寄存器即可。

```
always @ (negedge clk) begin
    stage_mem_pc <= out_mem_pc;
    stage_mem_out <= out_mem_out;
    stage_mem_alu_out <= out_mem_alu_out;
    stage_mem_opcode <= out_mem_opcode;
    stage_mem_rf_dest <= out_mem_rf_dest;
end
```

4.7 WB 阶段

WB 阶段只需要根据 `rf_dest` 把数据写入对应寄存器即可。由于 RegisterFile 不在这一模块中，我们同样需要用引脚把数据导出。

```
module WriteBack(
    input [`WORD] pc,
    input [`WORD] mem_out,
    input [`WORD] alu_out,
    input [`REG] mem_rf_dest,
    input [`OP] opcode,
    output [`REG] rf_dest,
    output [`WORD] rf_data,
    output rf_write
);
    wire is_branch;
    /* verilator lint_off PINMISSING */
    BranchOp branchOp(
        .opcode (opcode),
        .branch_op (is_branch)
    );
    /* verilator lint_on PINMISSING */

    wire is_mem_store;
    wire is_mem_load;

    /* verilator lint_off PINMISSING */
    MemoryOp memoryOp(
        .opcode (opcode),
```

```
        .store (is_mem_store),
        .load (is_mem_load));
    /* verilator lint_on PINMISSING */

    assign rf_data = is_mem_load ? mem_out : alu_out;
    assign rf_write = !is_branch && !is_mem_store && opcode != 2;
    assign rf_dest = mem_rf_dest;
endmodule
```

5 Hazard 的处理

5.1 Data Hazard

在上一章中，我们已经讨论了通过 Forwarding 模块处理 Data Hazard 的可能。只有 load-use hazard 会导致流水暂停。其他情况对流水都没有影响。

5.2 Control Hazard

Control Hazard 可能在两个模块中被检测到。

- `j`，`jal`，`jr` 无条件跳转导致的 Control Hazard，在 ID 阶段被监测。
- `beq`，`bne`，`bgez`，`bgtz`，`blez`，`bltz` 预测错误导致的跳转，在 EX 阶段被检测。

其中，

- ID 阶段检测到的 Hazard，通过 `force_jump` 信号传入 EX 阶段，在下一周期中处理。
- EX 阶段检测到的 Hazard，在当前周期处理。

虽然检测发生的时间不同，两个 Control Hazard 都在到达 EX 阶段才处理。

EX 阶段产生 `correct_branch_prediction` 和 `branch_jump_target` 信号。分支预测结果错误和无条件跳转都会导致 `correct_branch_prediction` 为 0。这一信号被 IF 阶段读取，并调整下一条指令的 PC 信号 `if_pc`。相关代码在前文已作分析，在此不再解释。

通过调整 IF 阶段的预测策略，我们可以很轻松地进行其他类型的分支预测 (比如 Two-level adaptive predictor)。在这个实验中，`stage_if_branch_taken` 始终为 0。故这个 CPU 实现的是 always-not-taken 的分支预测策略。

6 测试

本实验中，依然采用了我自己编写的 5 个样例，外加 3 个 hazard 测试样例。

6.1 基本功能测试

在上一份实验报告中，本人已经解释了相关的代码。在此，通过预期结果和实验结果的比较验证 36 个指令的实现情况。预期结果如下。

	zero	**at**	**v0**	**v1**	**a0**	**a1**	**a2**	**a3**
数值运算测试	0	FFFF	0	0	FFFE0000	FFFE0000	FFFF01D2	FFFF0000
比较指令测试	0	FFFE0000	0	0	F4240	1E8480	14	FFFE7960
分支测试	0	0	64	32	0	FFFFFFCE	0	0
循环 + 内存 + 数值运算测试	0	1	0	0	0	0	0	0
跳转指令测试	0	0	0	0	0	91D	E9	0
	t0	**t1**	**t2**	**t3**	**t4**	**t5**	**t6**	**t7**
数值运算测试	E90000	FFFF0000	FFFF0000	E9FFFE	E90000	0	FF170000	FF170000
比较指令测试	3D090000	24000000	3D0	0	3D0	0	1	0
分支测试	0	0	0	0	0	0	0	0
循环 + 内存 + 数值运算测试	0	0	0	0	0	0	0	0
跳转指令测试	0	0	0	0	0	0	0	0
	s0	**s1**	**s2**	**s3**	**s4**	**s5**	**s6**	**s7**
数值运算测试	0	FFFF0001	FFFF0001	FF160000	E9FFFF	FFFF	0	0
比较指令测试	1	0	0	0	0	0	0	0
分支测试	0	0	0	0	0	0	0	0
循环 + 内存 + 数值运算测试	0	2000	201	2200	0	200	0	FFFFFF00
跳转指令测试	5B25	0	0	0	0	0	0	0
	t8	**t9**	**k0**	**k1**	**gp**	**sp**	**fp**	**ra**
数值运算测试	0	0	0	0	0	0	0	0
比较指令测试	0	0	0	0	0	0	0	0
分支测试	0	0	0	0	0	0	0	0
循环 + 内存 + 数值运算测试	0	0	0	0	0	0	0	0
跳转指令测试	0	0	0	0	0	0	0	4

实验结果如下。

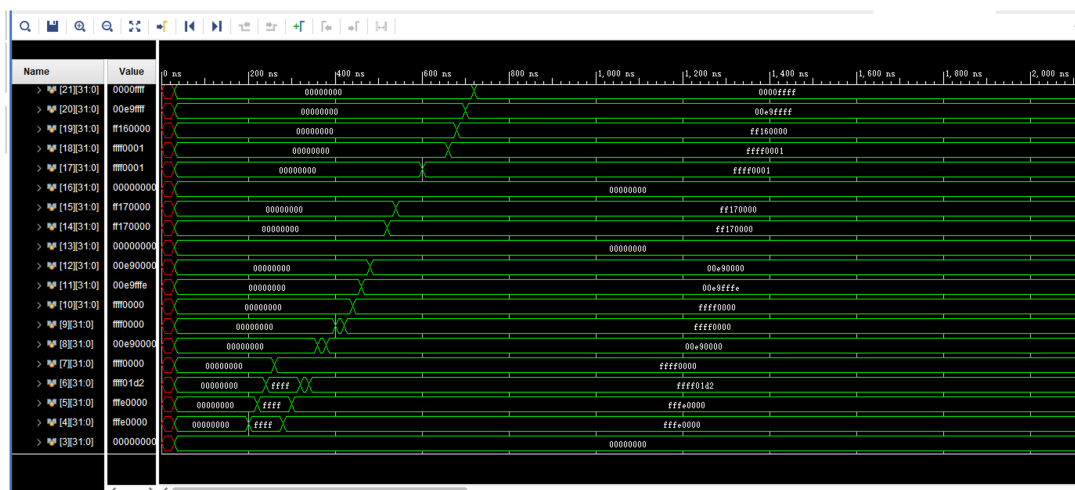

图 1: 数值运算测试结果

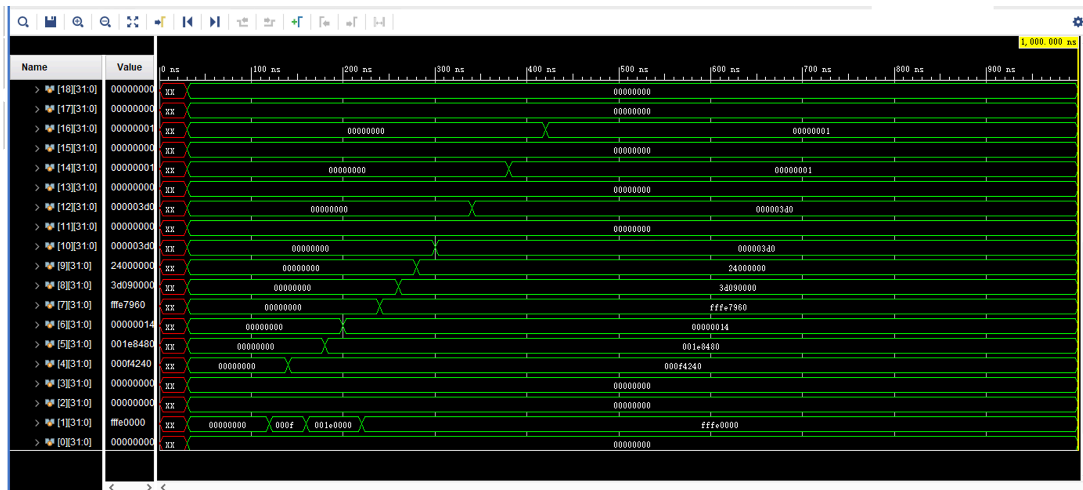

图 2: 比较指令测试结果

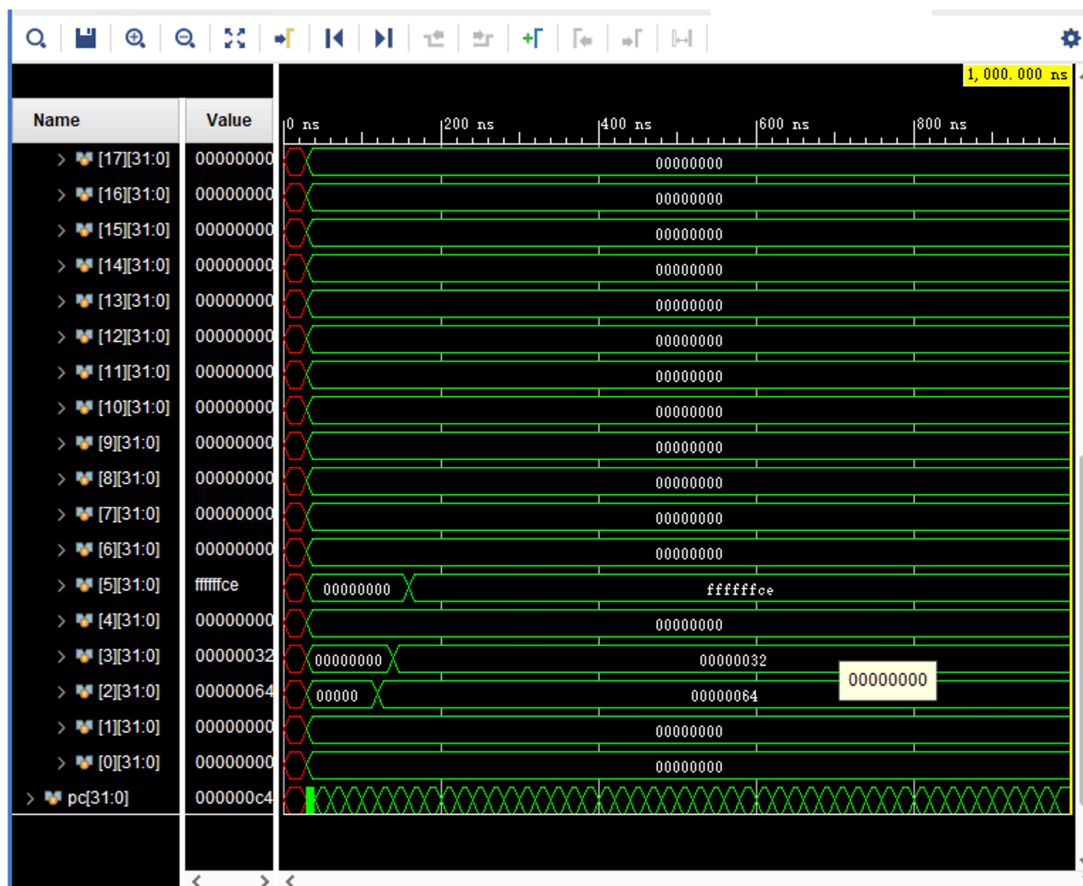

图 3: 分支测试结果

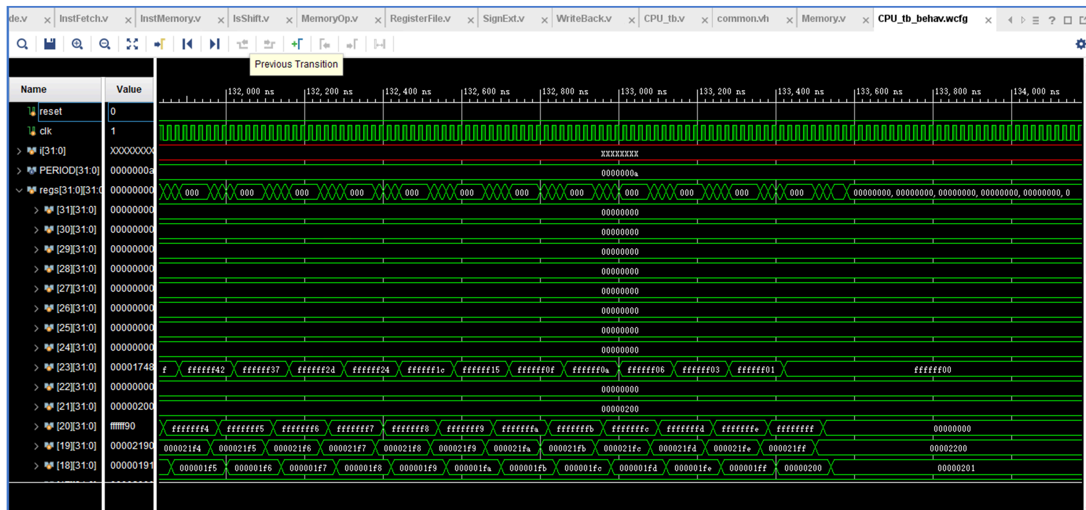

图 4: 循环 + 内存 + 数值运算测试结果

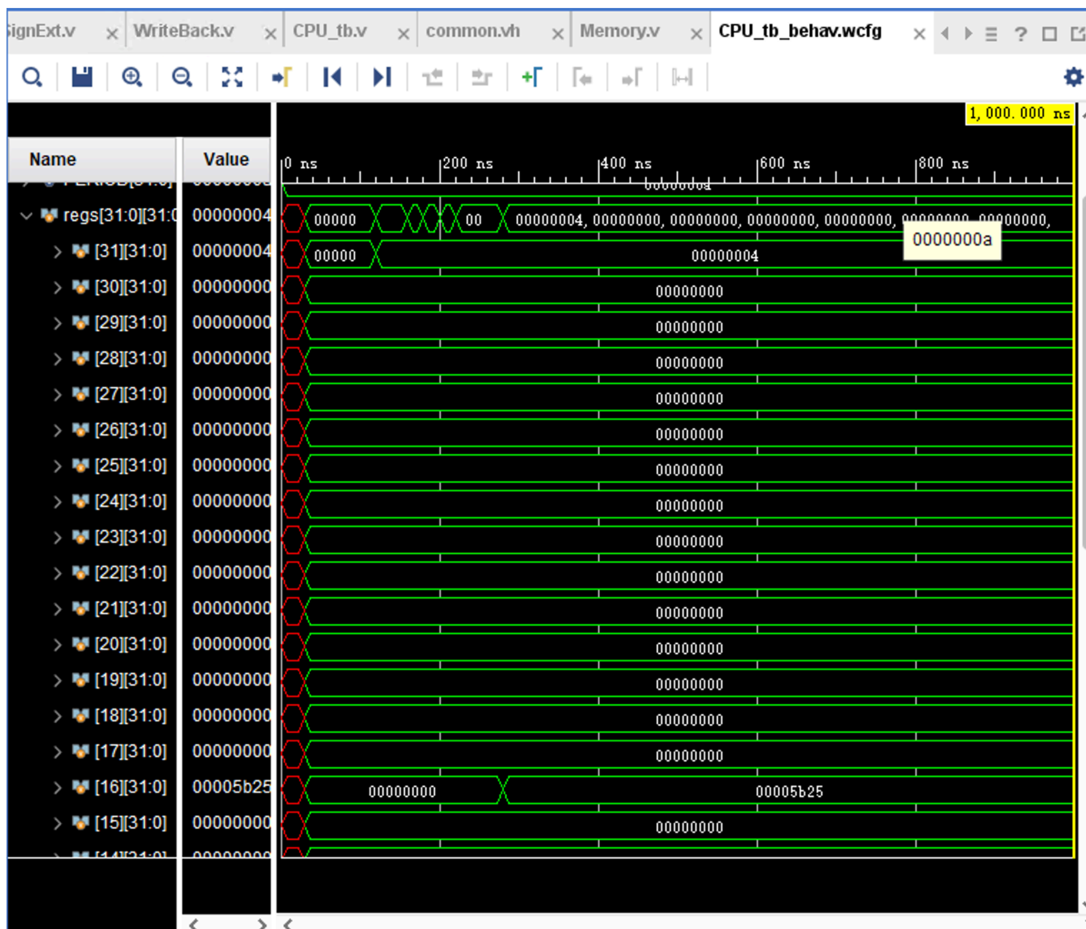

图 5: 跳转指令测试结果

6.2 Hazard 测试

首先测试 Data Hazard。

```
li $a0, 100
li $a1, 200
```

```
add $a2, $a0, $a1
add $a1, $a1, $a1
add $a2, $a2, $a2
add $a1, $a1, $a1
add $a2, $a2, $a2
add $a1, $a1, $a1
add $a2, $a2, $a2
add $a1, $a1, $a1
add $a2, $a2, $a2
add $a1, $a1, $a1
add $a2, $a2, $a2
add $a1, $a1, $a1
add $a2, $a2, $a2
add $a1, $a1, $a1
add $a2, $a2, $a2
```

寄存器	a0	a1	a2
预期输出	0x64	0x6400	0x9600

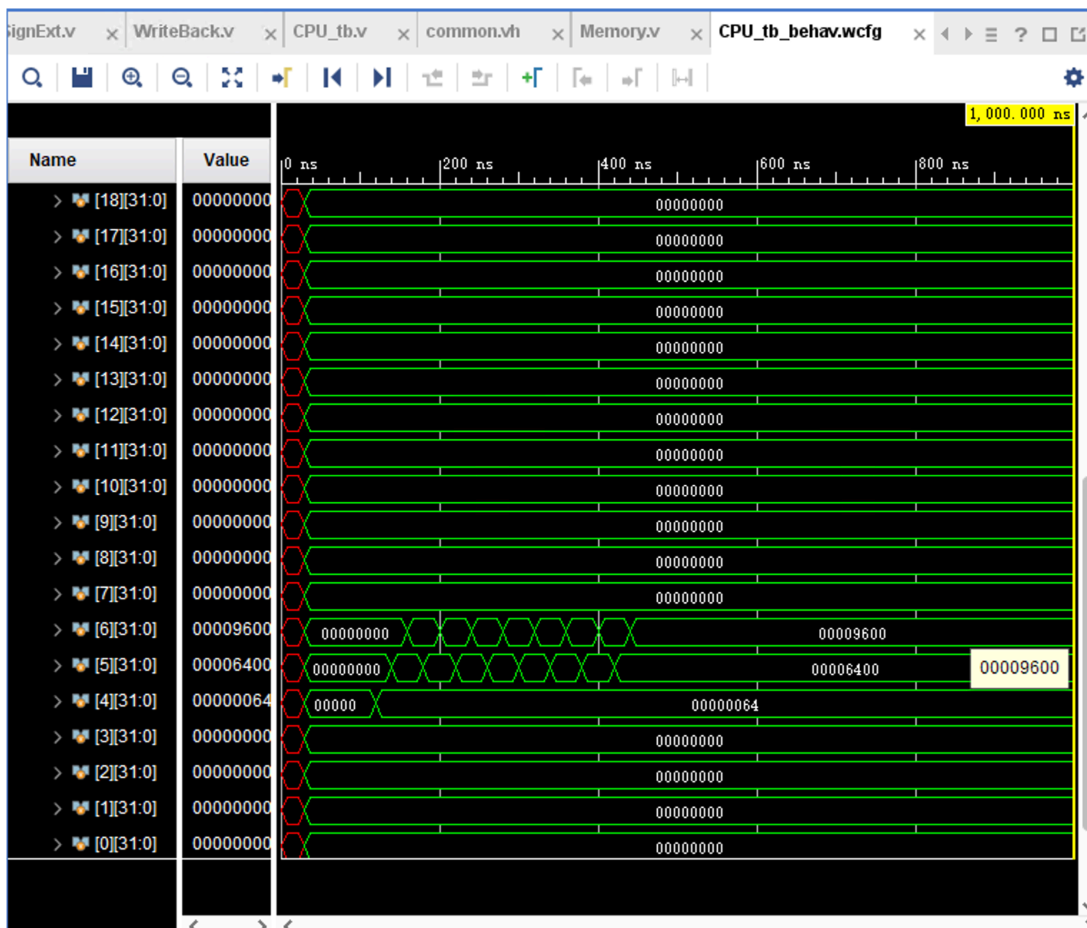

图 6: Data Hazard 测试结果

```
jal test1
test1:
jal test2
test2:
jal test3
test3:
jal test4
test4:
jal test5
test5:
jal test6
test6:
li $a0, 1000
```

寄存器	a0	ra
预期输出	0x3e8	0x18

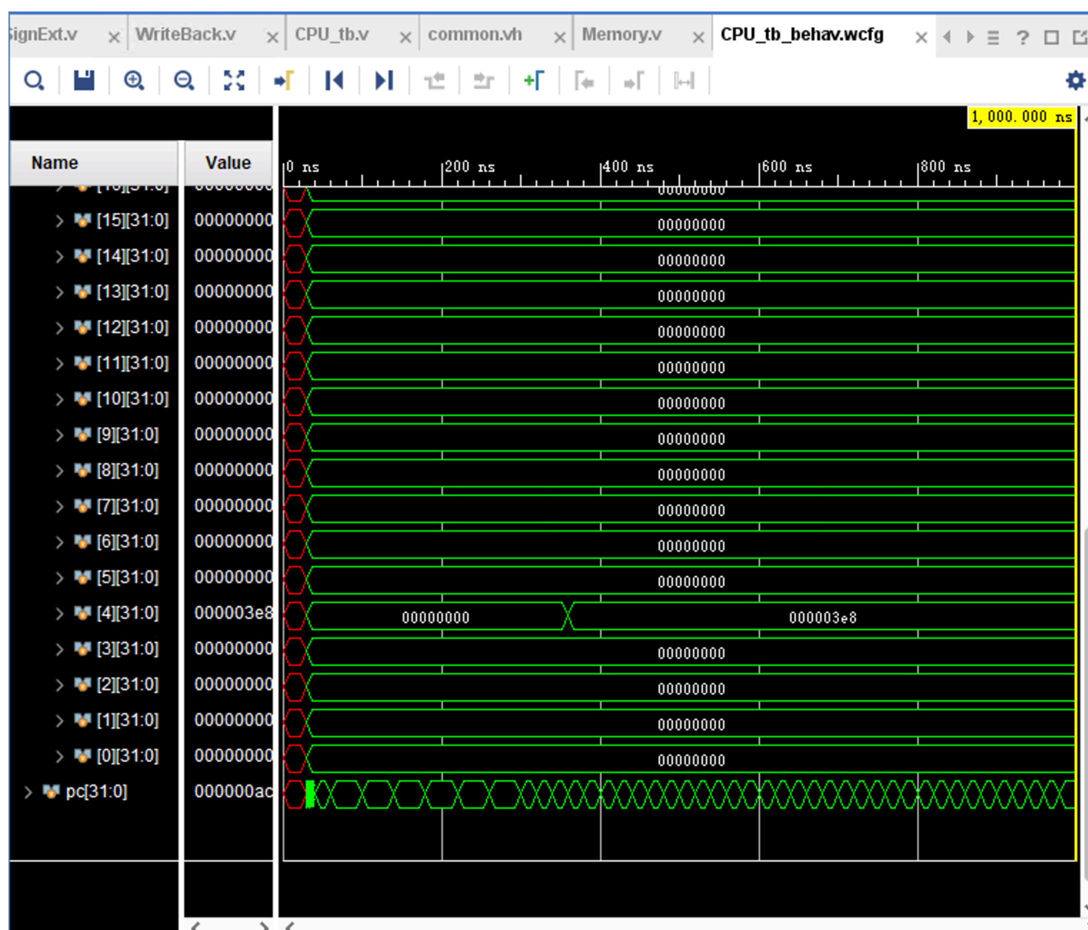

图 7: Control Hazard 测试结果

```
li $a0, 100
li $a2, 0x2000
sb $a0, ($a2)
```

```
lb $a1, ($a2)
addi $a1, $a1, 2000
lb $a0, ($a2)
add $a0, $a0, $a1
```

寄存器	a0	a1	a2
预期输出	0x898	0x834	0x2000

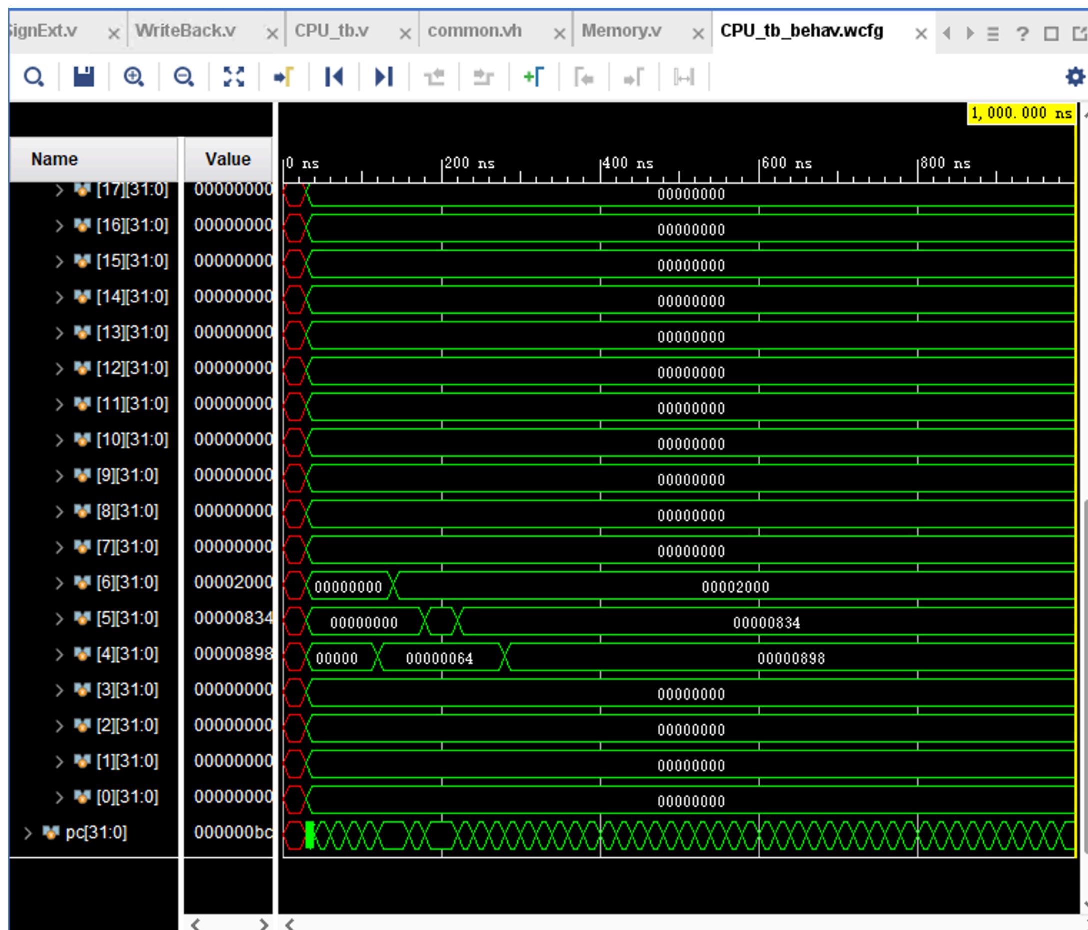

图 8: Load-Use Hazard 测试结果

7 总结与感想

This part is not made public.