MIPS32 CPU

韩周吾

第一部分：测试流程

通过 `instruction_memory` 模块将测试命令集加载到指令存储器中，在 Vivado 中运行仿真后，可以通过观察生成的`.wcfg` 文件中的波形图来验证指令执行的正确性。以下是详细的测试流程：

1. **加载指令集**：将编写的指令集通过`$readmemh` 方法加载到指令存储器模块中（在 instr_memory.sv 中选择需要运行的测试）。
2. **运行仿真**：在 Vivado 中启动仿真，并观察生成的波形图文件（`.wcfg`）。
3. **验证结果**：通过分析波形图中的信号变化，确认各指令的执行是否符合预期。例如，可以检查寄存器文件的值、ALU 结果、内存访问等。

第二部分：代码解释

包括以下关键模块和功能：

1. **指令取指单元（IUnit.sv）**：基于 PC 值获取当前指令，并将 PC 值加 4。
2. **控制单元（controller.sv）**：基于指令的操作码和功能码生成控制信号。
3. **算术逻辑单元（ALU）（ALU.sv）**：根据控制信号执行算术和逻辑操作。
4. **数据通路（DataPath.sv）**：管理流水线各阶段之间的数据流动。
5. **寄存器文件（Regfile.sv）**：存储 32 个寄存器的值。
6. **转发单元（FwdUnit.sv）**：通过转发数据解决数据冒险问题。
7. **加法器（Adder32bits.sv 和 Adder.sv）**：执行加法运算。
8. **Load-Use 检测单元（Load_Use_Det.sv）**：检测 Load-Use 冒险并在必要时暂停流水线。
9. **数据内存（data_memory.sv）**：模拟加载和存储指令的数据内存。
10. **符号扩展单元（EXT.sv）**：将 16 位立即数扩展为 32 位。

1. **分支预测实现**

通过 state 信号和 predPC 信号实现。state 信号表示当前的分支预测状态，predPC 信号表示预测的下一条指令的地址。

具体逻辑如下：

a) 在取指阶段，根据当前指令的操作码判断是否为分支指令。
b) 根据分支预测状态和条件预测下一条指令的地址。
c) 如果预测错误，清除流水线中的错误指令，确保后续指令执行的正确性，并通过 IDEx_PCpl4 或 Btar 纠正下一条指令的地址，并更新分支预测状态。

2. **转发逻辑**

a) ForwardA 和 ForwardB 的计算：

- ForwardA[1] 和 ForwardB[1] 用于检测并解决 EX 阶段的数据冒险。当 EX/MEM 阶段的指令需要写回寄存器，并且写回的目标寄存器与当前指令的源寄存器匹配时，触发转发。
- ForwardA[0] 和 ForwardB[0] 用于检测并解决 MEM 阶段的数据冒险。当 MEM/WB 阶段的指令需要写回寄存器，并且写回的目标寄存器与当前指令的源寄存器匹配时，触发转发。

b) 操作过程：

- 在执行单元（ExecUnit）中，根据 ForwardA 和 ForwardB 的值，选择适当的操作数源。
- 如果需要转发，则使用前一个阶段的结果作为当前指令的操作数，避免数据冒险。

第三部分：代码评估

支持如下指令：

lw, sw, addi, addiu, slti, sltiu, andi, ori, xori, beq, bne, j, sllv, add, addu,

sub, subu, slt, sltu, and, or, xor, nor, nop，共 24 条指令。

功能

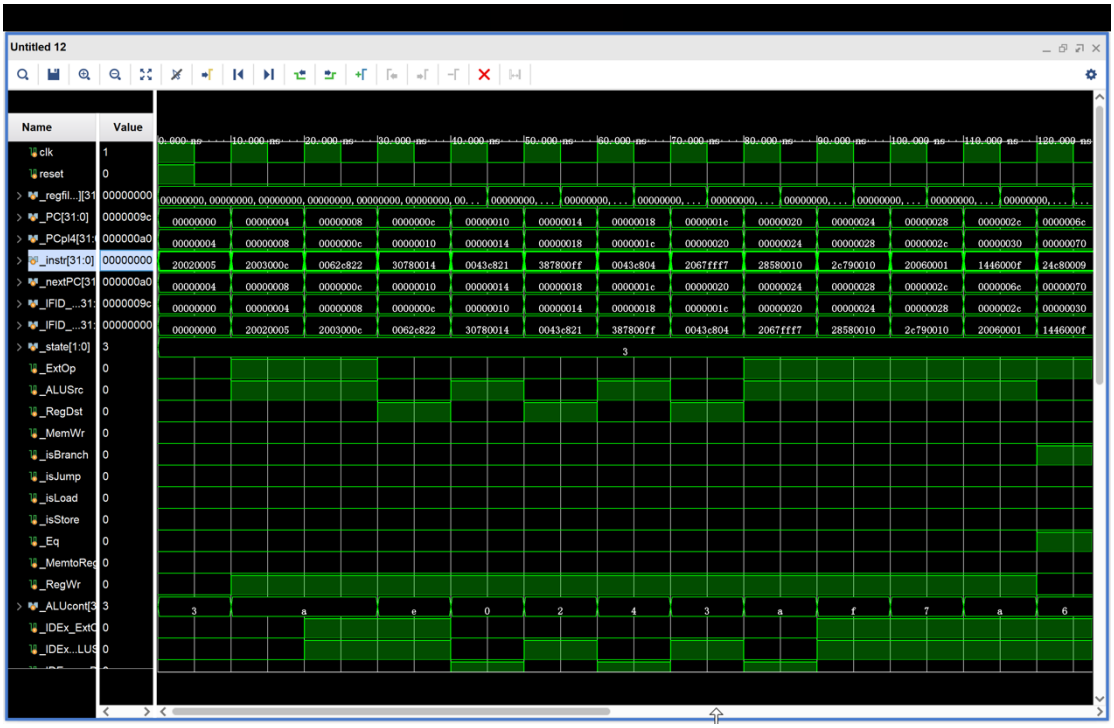
1. **数据冒险处理**：通过转发单元（Forward Unit）和 Load-Use 检测单元（Load-Use Detector）有效地处理了数据冒险问题。
2. **异常处理机制**：包含基本的异常处理机制，能有效处理指令错误、溢出和数据存储错误。
3. **动态分支预测**：通过在译码阶段和执行阶段对分支指令的预测和校验机制，实现了简单的动态分支预测。预测机制根据前一个分支的执行结果进行预测，并通过比较预测结果和实际执行结果来更新预测状态。

缺陷

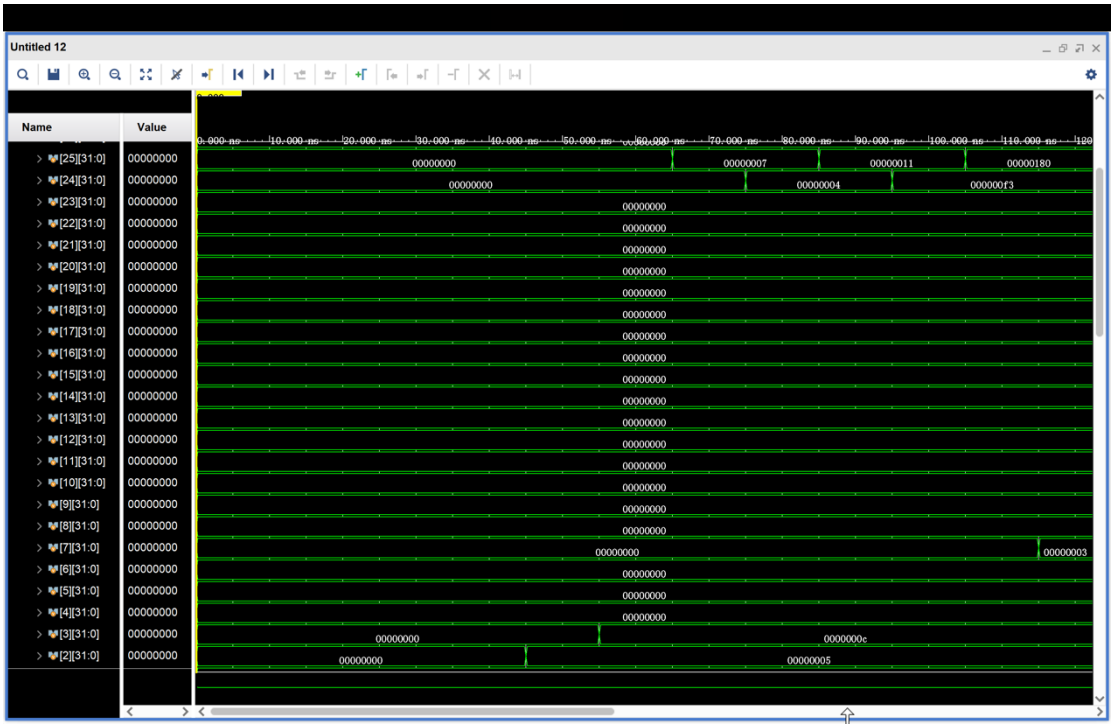
1. **缺乏扩展性**：当前实现的指令集较为基本，部分指令尚未实现，需要进一步扩展。
2. **性能优化不足**：未实现更高级的性能优化技术，如超标量执行等。
3. **测试覆盖率有限**：测试用例相对简单，未覆盖所有可能的指令组合和边界情况，测试覆盖率有待提高。

正确性验证

测试一：指令测试（包含实现的所有指令）



寄存器：



测试一命令：

20020005	ADDI $2, $0, 0x0005	读取$0，运算 0+5，将 0x5 写入$2
2003000c	ADDI $3, $0, 0x000C	读取$0，运算 0+c，将 0xc 写入$3
0062C822	SUB $25, $3, $2	读取$3 $2，运算 c-5，将 0x7 写入$25
30780014	ANDI $24, $3, 0x14	读取$3，运算 c&14，将 0x4 写入$24

0043C821　　ADDU $25, $2, $3　　读取$2 $3, 运算 5+c，将 0x11 写入 $25
387800FF　　XORI $24, $3, 0xff　　读取$3, 运算 c^ff，将 0xf3 写入$24
0043C804　　SLLV $25, $3, $2　　读取$3 $2, 运算 c<<5，将 0x300 写入 $25
2067fff7　　ADDI $7, $3, 0xFFF7　　读取$3, 运算 c-9，将 0x3 写入$7
28580010　　SLTI $24, $2, 0x10　　读取$2, 判断 5<0x10，将 1 写入$24
2C790010　　SLTIU $25, $3, 0x10　　读取$3, 判断 c<0x10，将 0 写入$25
20060001　　ADDI $6, $0, 0x0001　　读取$0, 运算 0+1，将 1 写入$6
1446000f　　BNE $2, $6, 0x000F　　判断$2 != $6，若是跳转到 pc+0x000F
00e22025　　OR $4, $7, $2　　读取$7 $2, 运算 3|5，将 0x7 写入$4
00642824　　AND $5, $3, $4　　读取$3 $4, 运算 c&7，将 0x4 写入$5
00a42820　　ADD $5, $5, $4　　读取$5 $4, 运算 4+7，将 0xb 写入$5
10a7000b　　BEQ $5, $7, 0x000B　　判断$5 == $7，若是跳转到 pc+0x000B
0064202a　　SLT $4, $3, $4　　读取$3 $4, 判断 c<7，将 0 写入$4
10800001　　BEQ $4, $0, 0x0001　　判断$4 == $0，若是跳转到 pc+0x0001
20050000　　ADDI $5, $0, 0x0000　　读取$0, 运算 0+0，将 0 写入$5
00e2202b　　SLTU $4, $7, $2　　读取$7 $2, 判断 3<5，将 1 写入$4
00853820　　ADD $7, $4 $5　　读取$4 $5, 运算 1+0，将 1 写入$7
00e23822　　SUB $7, $7, $2　　读取$7 $2, 运算 1-5，将 0xfffb 写入 $7
ac670044　　SW $7, 0x0044, $3　　将$7 写入内存地址 c+0x0044
8c020050　　LW $2, 0x0050, $0　　将内存地址 0x0050 数据写入$2
20470001　　ADDI $7, $2, 0x0001　　读取$2, 运算 0+1，将 1 写入$7
0800001c　　J 0x000001C　　跳转到地址 0x000001C
20020001　　ADDI $2, $0, 0x0001　　读取$0, 运算 0+1，将 1 写入$2
24c80009　　ADDIU $8, $6, 0x0009　　读取$6, 运算 1+9，将 0xa 写入$8
2409fffd　　ADDIU $9, $0, 0xFFFD　　读取$0, 运算 0xfffd，将 0xfffd 写入 $9
350a0005　　ORI $10, $8, 0x0005　　读取$8, 运算 a|5，将 0xf 写入$10
010a5823　　SUBU $11, $8, $10　　读取$8 $10, 运算 a-f，将 0x5 写入 $11
01096026　　XOR $12, $8, $9　　读取$8 $9, 运算 a^fffd，将 0xfffffff7 写入$12
016c6827　　NOR $13, $11, $12　　读取$11 $12, 运算~(5|fffffff7)，将 0xfffffff8 写入$13
15a00001　　BNE $13, $0, 0x0001　　判断$13 != $0，若是跳转到 pc+0x0001
15a8ffe9　　BNE $13, $8, 0xFFE9　　判断$13 != $8，若是跳转到 pc+0xFFE9
ac020054　　SW $2, 0x0054, $0　　将$2 写入内存地址 0x0054
00000000　　NOP　　空操作
00000000　　NOP　　空操作
00000000　　NOP　　空操作

00000000　　NOP　　空操作

部分指令详细分析（i、r、j 型指令各一条）

指令 1：`20020005`（addi $2, $0, 5）

这条指令的含义是将立即数 5 加到寄存器$0 的值，并将结果存入寄存器$2。具体步骤如下：

1. **取指阶段（IF）：**
   - PC：初始值为 0x00000000。
   - PC+4：计算得到 0x00000004。
   - Instr Memory：从指令存储器中读取指令 20020005。
2. **译码阶段（ID）：**
   - Instr：20020005 被译码为 addi 指令。
   - rs：$0（源寄存器），rt：$2（目标寄存器），imm：5（立即数）。
   - 控制信号：生成控制信号，如 ALUSrc = 1，RegDst = 0，RegWr = 1 等。
   - 执行阶段（EX）：
   - ALU 操作数：SrcA = $0 的值（0），SrcB = 5（立即数扩展后）。
   - ALU 运算：执行加法，结果为 5。
   - 目标寄存器：根据控制信号，目标寄存器为$2。
3. **访存阶段（MEM）：**
   - 无访存操作：因为是算术指令，无需访存。
4. **写回阶段（WB）：**
   - 写寄存器：将 ALU 结果 5 写回到寄存器$2。

在这一过程中，段寄存器传递如下：

- IF/ID 段寄存器：存储 PC+4（0x00000004）和 Instr（20020005）。
- ID/EX 段寄存器：存储译码后的控制信号和操作数。
- EX/MEM 段寄存器：存储 ALU 结果（5）和目标寄存器信息。
- MEM/WB 段寄存器：存储写回的值（5）和目标寄存器信息。

总结：

- PC 变化：从 0x00000000 到 0x00000004。
- 寄存器$2：值变化为 5。

指令 2：`0043C821`（addu $25, $2, $3）

这条指令的含义是将寄存器$2 和寄存器$3 的值相加，并将结果存入寄存器$25。具体步骤如下：

1. **取指阶段（IF）：**
   - PC：从 0x0000000C 开始。
   - PC+4：计算得到 0x00000010。
   - Instr Memory：从指令存储器中读取指令 0043C821。
2. **译码阶段（ID）：**
   - Instr：0043C821 被译码为 addu 指令。
   - rs：$2（源寄存器），rt：$3（源寄存器），rd：$25（目标寄存器）。
   - 控制信号：生成控制信号，如 ALUSrc = 0，RegDst = 1，RegWr = 1 等。
3. **执行阶段（EX）：**
   - ALU 操作数：SrcA = $2 的值（5），SrcB = $3 的值（12）。
   - ALU 运算：执行加法，结果为 0x11。
   - 目标寄存器：根据控制信号，目标寄存器为$25。
4. **访存阶段（MEM）：**
   - 无访存操作：因为是算术指令，无需访存。
5. **写回阶段（WB）：**
   - 写寄存器：将 ALU 结果 0x11 写回到寄存器$25。

在这一过程中，段寄存器传递如下：

- IF/ID 段寄存器：存储 PC+4（0x00000010）和 Instr（0043C821）。
- ID/EX 段寄存器：存储译码后的控制信号和操作数。
- EX/MEM 段寄存器：存储 ALU 结果（17）和目标寄存器信息。
- MEM/WB 段寄存器：存储写回的值（17）和目标寄存器信息。

总结：

- PC 变化：从 0x0000000C 到 0x00000010。
- 寄存器$25：值变化为 17。

指令 3：`0800001c`（J 0x000001C）

这条指令的含义是无条件跳转到地址 0x000001C。具体步骤如下：

1. **取指阶段（IF）：**
   - PC：当前 PC 值为 0x00000078。
   - PC+4：计算得到 0x0000007C。

- Instr Memory：从指令存储器中读取指令 0800001c。

2. **译码阶段（ID）：**
   - Instr：0800001c 被译码为 J 型指令。
   - op：操作码为 0x2，表示 J 指令。
   - 地址：指令中的地址字段为 0x000001C。
3. **执行阶段（EX）：**
   - 计算跳转地址：将当前 PC 的高 4 位与指令中的地址字段组合，并左移 2 位，形成目标地址。
     - 目标地址计算：目标地址 = {PC[31:28], 地址[25:0], 2'b00}。
     - 计算得到：目标地址 = {0x0, 0x000001C, 2'b00} = 0x0000070。
4. **访存阶段（MEM）：**
   - 无访存操作：因为是跳转指令，无需访存。
5. **写回阶段（WB）：**
   - 无写回操作：因为是跳转指令，无需写回寄存器。

在这一过程中，段寄存器传递如下：

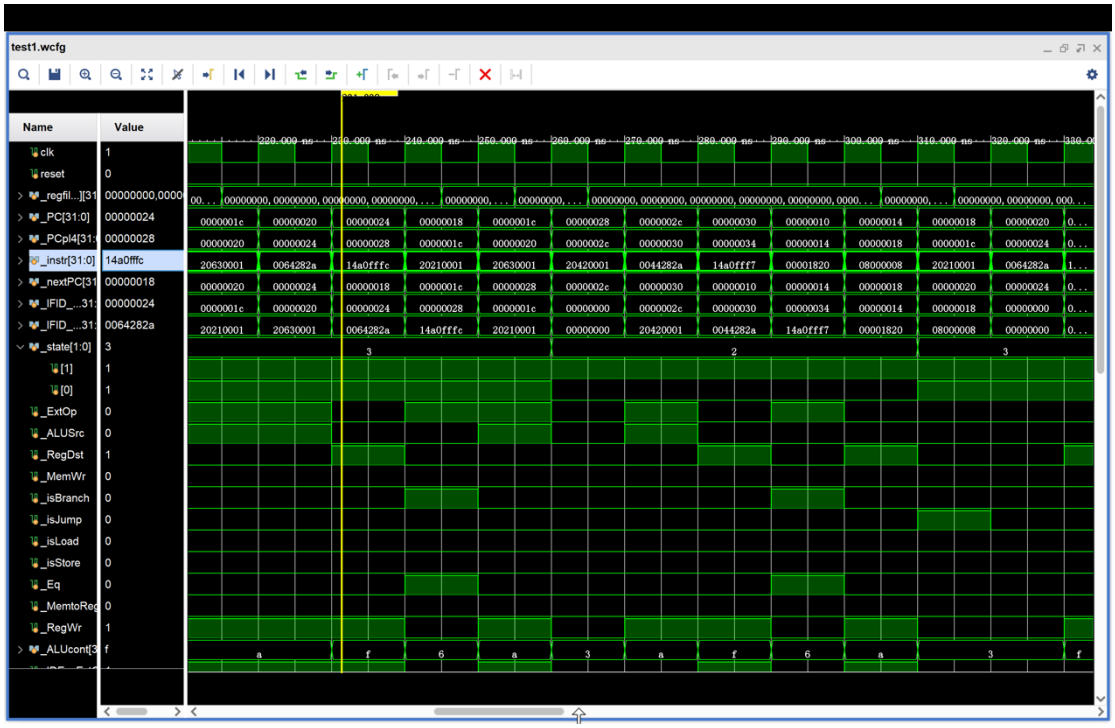
- IF/ID 段寄存器：存储 PC+4（0x0000007C）和 Instr（0800001c）。
- ID/EX 段寄存器：存储译码后的控制信号和目标地址。
- EX/MEM 段寄存器：无关紧要，因为没有访存操作。
- MEM/WB 段寄存器：无关紧要，因为没有写回操作。

总结：

- PC 变化：从 0x00000078 跳转到 0x0000070。
- 寄存器：无变化，因为没有寄存器写回。

与图中结果一致。

测试二：分支测试（包含分支预测的测试）

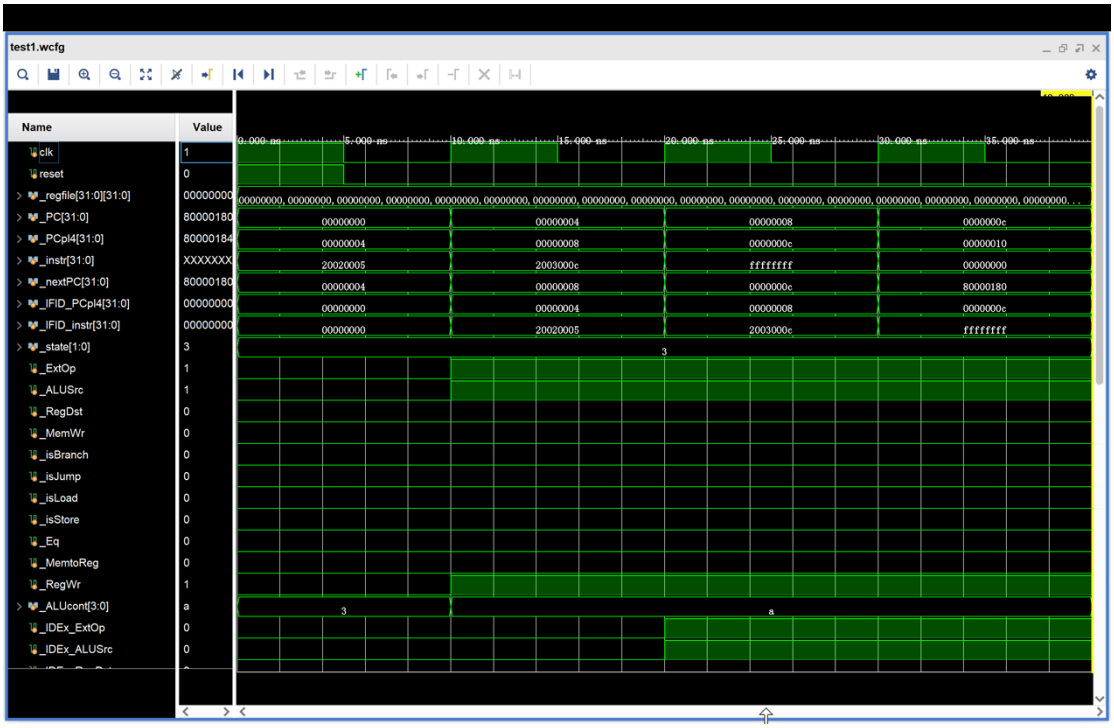


局部测试代码：

```
20210001    0x18    addi $at, $at, 1

20630001    0x1c    addi $v1, $v1, 1

0064282a    0x20    slt $a1, $v1, $a0

14a0fffc    0x24    bne $0, $a1, -4

20420001    0x28    addi $v0, $v0, 1
```

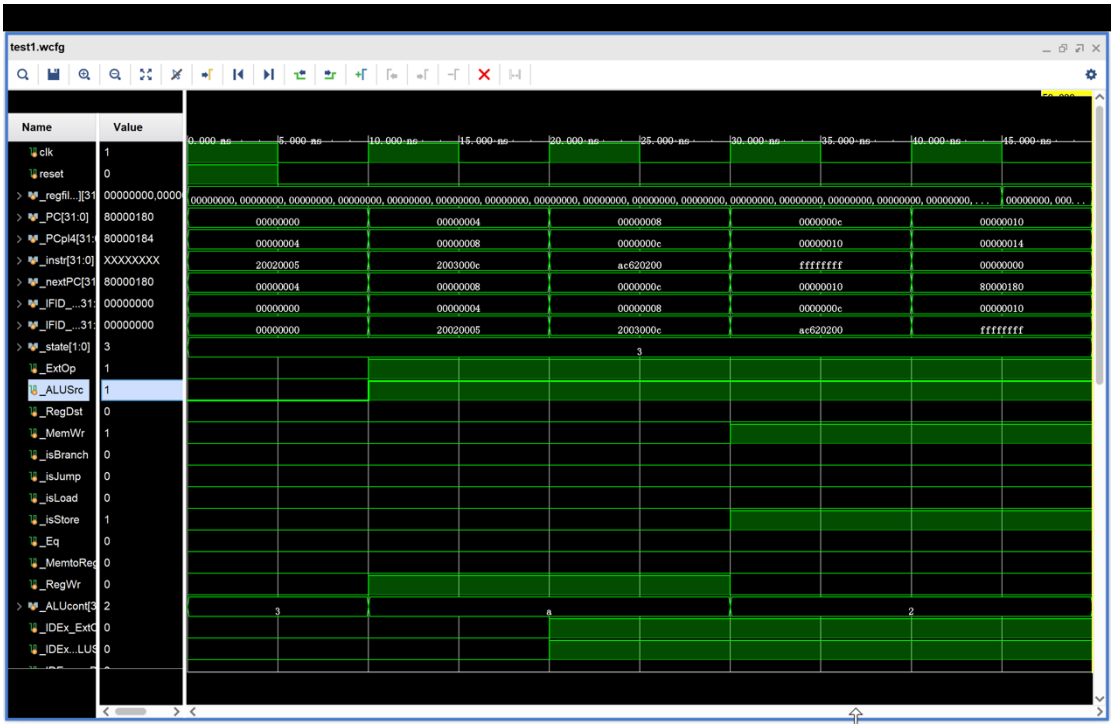
标杆处为测试的分支指令（0x24），由于之前 state 为 11（强跳转），所以本次预测为跳转（本指令为循环逻辑的跳转，故跳转为常态）。然而，执行两个周期后（分支指令执行完 exe 阶段），分支结果为不跳转，并写入了新的 PC，此时将取指、译码阶段的寄存器 flush（不涉及写入，故只 flush 这两个阶段内部即可），即可保证程序正确运行。这时，state 被修改为 10（弱跳转）。

测试三：指令错误测试



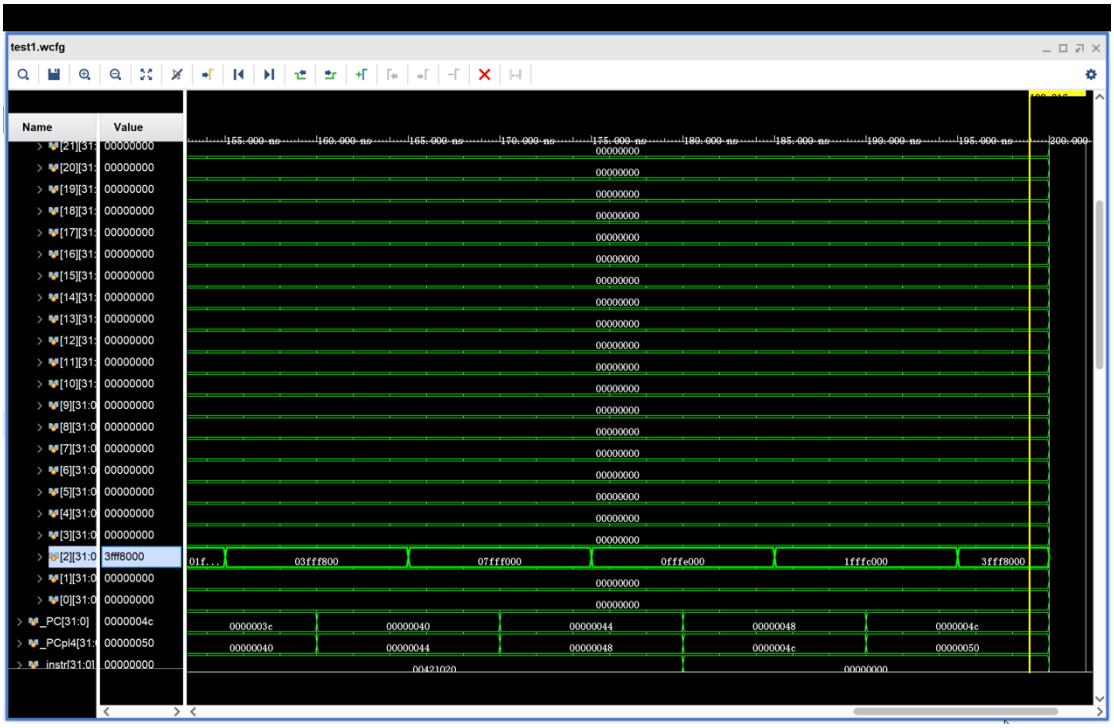
当出现 instruction_error 时，段寄存器会进行 flush，并且自动调节 pc 到 80000180，调用异常处理程序（未实现），本例中会因为 PC 超过 160 自动退出。

测试四：内存地址错误测试



会自动将 dmem_error 设为 1，并跳转到 0x80000180。

测试五：溢出测试



overflow 会被 set，然后跳转到 0x80000180。