Computer Organization Project 说明文档

开发者说明：

周益贤 12211825

- 负责工作：CPU 子模块设计（IF，ID，EX，MEM，WB）
- 贡献百分比：50%

张书玮 12212912

- 负责工作：CPU 架构设计（流水线）
- 贡献百分比：50%

CPU架构设计说明

CPU特性

- 指令：

  ISA: RISC-V.

  支持指令：add, sub, xor, or, and, sll, srl, sra, slt, sltu, addi, slti, sltiu, xori, ori, andi, slli, srli, srai, sb, sh, sw, lb, lh, lw, lbu, lhu, beq, bne, blt, bge, bltu, bgeu, jal, jalr, lui, auipc, ecall, ebreak

  编码与使用：所有指令编码均与 RV32I 编码一致，除 ecall 和 ebreak 外其他指令功能也与标准一致。ecall 仅实现了 1, 5, 10 三种调用。分别表示输出整数，输入整数和停机。ebreak 效果为一个暂停，按下开始键后继续运行。

- CPU时钟：23MHZ（流水线每个阶段的时钟）

  CPI：1

  周期类型：流水线（5级流水线）

  流水线冲突：采用 ID 后的 forward 模块和 EX 后的 forward 模块解决。

- 寻址空间：哈佛结构

  寻址单位：Byte

  指令空间/数据空间大小：65536 Bytes

  栈空间基地址：65536

- 外设IO：采用 MMIO 的方式（实际为寄存器模拟的虚拟地址），中断访问。

CPU接口

输入：

- **clk_in**：系统时钟
- **rst_in**：全部重置（本来设计的思路是想把memory重置掉，后面发现memory重置不了，就和局部重置一样了）
- **local_rst_in**：局部重置
- **start_in**：开始
- **pause_in**：暂停
- **uart_mode_in**：uart模式
- **display_mode_in**：显示模式（显示输入数据/CPU状态）
- **idata_in**：8bit，输入数据

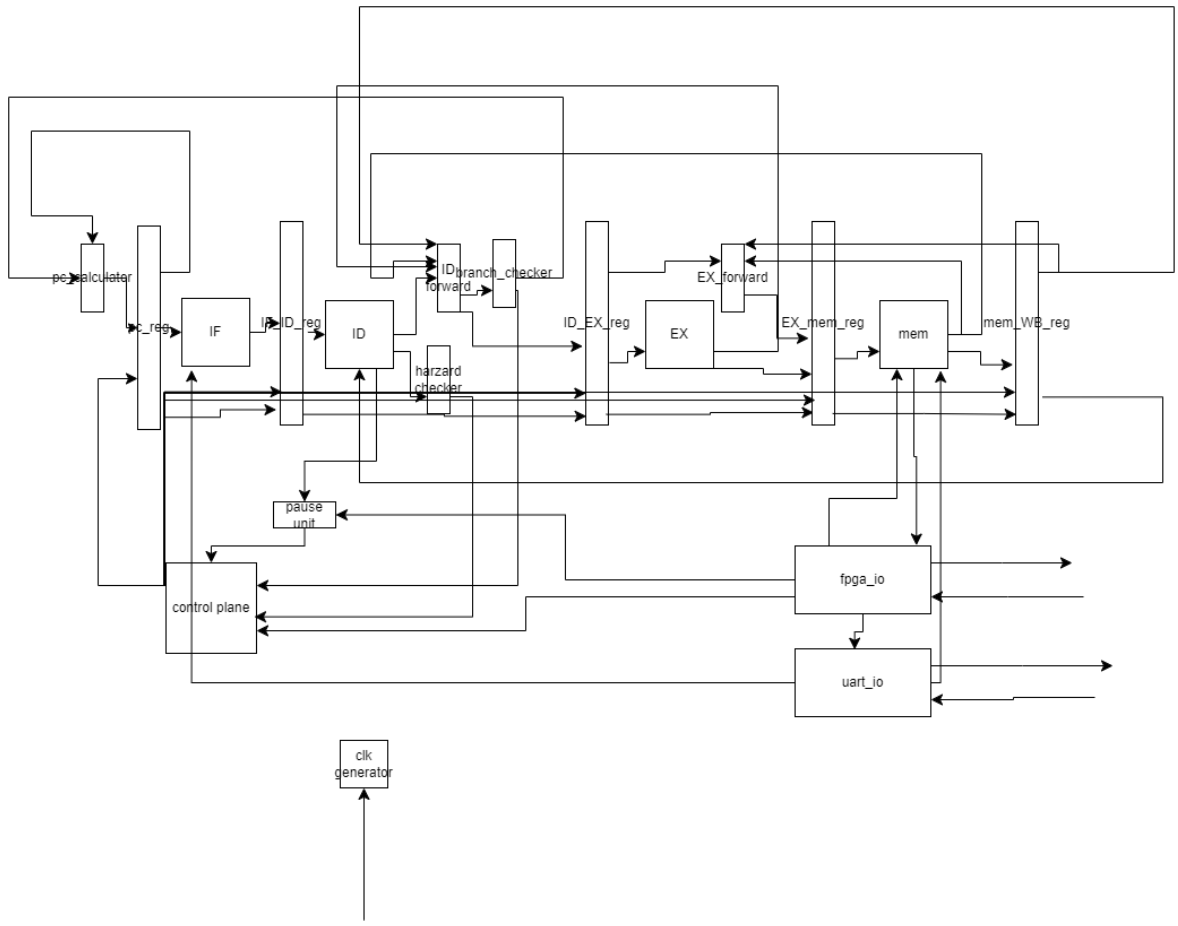
- **uart_rx**：uart 输入

输出：

- **led**：8bit，控制左侧 led 灯
- **debug_led**：8bit，控制右侧 led 灯（本来是 debug 用的，后面没改名字直接就继续用了）
- **sel1/sel2**：4bit，片选信号
- **seg_ctrl1/seg_ctrl2**：8bit，段选信号
- **uart_rx**：uart 输出

CPU内部结构

结构图（省略 clk 连线）



子模块设计说明

未说明则为 1bit。部分模块输入输出省略。

- **cpu_clk**

  生成时钟。

  输入：

  - **clk_in**：系统时钟

  输出：

  - **cpu_clk**：cpu 时钟
  - **ctr_clk**：控制时钟（和 cpu 时钟周期相同，3/4高电平，1/4低电平）
  - **uart_clk**：uart 时钟
- **fpga_io**

  和 fpga 开发板交互。

输入输出略。

- **uart_io**

  和 uart 模块交互。

  输入输出略。

- **pause_unit**：

  暂停和启动控制。

  输入：

  - **rst**：复位信号
  - **cpu_clk**：时钟
  - **ecall_pause**：[from Decoder] ecall 指令的暂停信号
  - **board_start**：[from fpga_io]板上输入的开始信号
  - **board_pause**：[from board_pause] 板上输入的暂停信号
  - **read_pause**：[from Decoder] 等待读入的暂停信号

  输出：

  - **start**：开始
  - **pause**：暂停

- **control_plane**

  控制器，维护状态自动机，控制整个流水线的状态。

  输入：

  - **ctr_clk**：控制时钟
  - **rst**：[from fpga_io]复位
  - **local_rst**：[from fpga_io] 局部复位
  - **start**：[from pause_unit] 开始
  - **pause**：[from pause_unit] 结束
  - **exit**：[from fpga_io] 退出
  - **nop_flag**：[from hazard_checker] 是否需要插入nop
  - **branch_flag:** [from branch_check] 是否需要跳转（用于分支预测）
  - **ID_nop**：[from ID_EX_reg] ID/EX stage register 是否nop
  - **EX_nop**：[from EX_mem_reg] EX/mem 是否nop
  - **mem_nop**：[from mem_WB_reg] mem/WB nop

  输出：

  - **pc_ctrl/IF_ctrl/ID_ctrl/EX_ctrl/mem_ctrl**：2bit，控制对应的stage register
  - **is_clear**：ID后面的流水线是否清空
  - **state_out**：控制器当前状态，用于输出展示

- **ALU**

  对操作数进行运算，根据参数选择对应的操作输出为结果。

  输入：

  - **func3:** [from ID_EX_reg] 3bits, 指令中的 func3 部分。
  - **func7:** [from ID_EX_reg] 7bits, 指令中的 func7 部分。
  - **use_imm:** [from ID_EX_reg] 是否使用立即数，若为 1 则第二个操作数被替换为立即数。
  - **data1:** [from ID_EX_reg] 32bits, 第一个操作数。
  - **data2:** [from ID_EX_reg] 32bits, 第二个操作数。
  - **imm:** [from ID_EX_reg] 32bits, 立即数。

  输出：

  - **res:** 32bits, 运算结果。

- **branch_checker**

  根据操作数判断是否需要分支或跳转。

  输入：

  - **branch_flag:** [from Decoder] 是否需要判断分支。
  - **jmp_flag:** [from Decoder] 是否需要判断跳转。
  - **data1:** [from ID_forward] 32bits, 第一个操作数。
  - **data2:** [from ID_forward] 32bits, 第二个操作数。
  - **imm:** [from Decoder] 32bits, 立即数。
  - **func3:** [from Decoder] 3bits, 指令中的 func3 部分。

  输出：

  - **branch:** 是否需要分支。
  - **jmp:** 是否需要跳转。
  - **offset:** 32bits, 分支或跳转偏移量。
- **pc_calculator**

  根据当前 pc 计算下一周期 pc 的值。若需要分支则 pc = pc + offset，若需要跳转则 pc = offset，否则 pc = pc + 4.

  输入：

  - **pc:** [from pc_reg] 32bits, pc 寄存器的值。
  - **branch:** [from branch_check] 是否需要分支。
  - **jmp:** [from branch_check] 是否需要跳转。
  - **offset:** [from branch_check] 32bits, 分支或跳转偏移量。

  输出：

  - **nxt_pc:** 32bits, 下一周期 pc 的值。
- **IFetch**

  读取 pc 处的指令，以及控制 uart 的指令写入。

  输入：

  - **clk:** 时钟。
  - **pc:** [from pc_reg] 32bits, pc 寄存器的值。
  - **uart_clk:** uart 时钟。
  - **uart_enable:** [from uart_io] 是否启用 uart.
  - **uart_addr:** [from uart_io] 32bits, uart 写入地址。
  - **uart_data:** [from uart_io] 32bits, uart 写入数据。
  - **uart_done:** [from uart_io] uart 写入是否完成。

  输出：

  - **inst:** 32 bits, 读取到的指令。
- **memory**

  处理内存写入和读取。同时也处理输入和输出。

  输入：

  - **clk, rst:** 时钟和重置。
  - **write_flag:** [from EX_mem_reg] 是否需要写入内存。
  - **mem_op:** [from EX_mem_reg] 3bits, 读取/写入操作类型（位宽&符号）
  - **addr:** [from EX_mem_reg] 32bits, 读取/写入地址。
  - **data_in:** [from EX_mem_reg] 32bits, 待写入的数据。
  - **input_data:** [from fpga_io] 32bits, 读入的数据。
  - **uart_clk:** uart 时钟。
  - **uart_enable:** [from uart_io] 是否启用 uart.

- **uart_addr:** [from uart_io] 32bits, uart 写入地址。
- **uart_data:** [from uart_io] 32bits, uart 写入数据。
- **uart_done:** [from uart_io] uart 写入是否完成。

输出：

- **res:** 32bits, 读取到的数据。
- **output_data:** 32bits, 待输出的数据。

- **Decoder**

  解码指令，维护寄存器。

  输入：

  - **clk, rst:** 时钟和重置。
  - **inst:** [from IF_ID_reg] 32bits, 指令。
  - **dst:** [from mem_WB_reg] 5bits, 写入寄存器的编号。
  - **alu_res:** [from mem_WB_reg] 32bits, ALU 运算结果。
  - **mem_res:** [from mem_WB_reg] 32bits, 内存读取结果。
  - **pc:** [from IF_ID_reg] 32bits, pc 寄存器的值。
  - **reg_write_flag:** [from mem_WB_reg] 3bits, 写回类型，指示把哪个结果写回寄存器（或者不写入）
  - **ecall_enable:** [from control_plane] 是否准备好进行 ecall 调用。

  输出：

  - **imm:** 32bits, 立即数。
  - **use_imm:** ALU 是否使用立即数。
  - **use_alu:** 是否需要在 ALU 之前用到数据。
  - **func3:** 3bits, 指令中的 func3 部分。
  - **func7:** 7bits, 指令中的 func7 部分。
  - **data1_idx：** 5bits, 第一个操作数的寄存器编号，0为不使用，下同。
  - **data2_idx：** 5bits, 第二个操作数的寄存器编号。
  - **sw_idx：** 5bits, 需要被写入内存的寄存器编号。
  - **dst_idx：** 5bits, 写回寄存器编号。
  - **data1_data：** 32bits, 第一个操作数的值，0为不使用，下同。
  - **data2_data：** 32bits, 第二个操作数的值。
  - **sw_data：** 32bits, 需要被写入内存的值。
  - **reg_write:** 3bits, 写回类型，指示把哪个结果写回寄存器（或者不写入）
  - **mem_write:** 是否需要写入内存。
  - **mem_op:** 3bits, 读取/写入操作类型（位宽&符号）
  - **branch:** 是否需要判断分支。
  - **jmp:** 是否需要判断跳转。
  - **ecall_pause:** 是否需要因 ecall 暂停。
  - **read_pause:** 是否需要因输入暂停。
  - **exit:** 是否结束。

- **pc_reg**

  pc寄存器，维护当前pc

  输入：

  - **cpu_clk/rst**
  - **next_pc**：[from pc_calculator] 下一个pc
  - **sr_ctrl**：[from control_plane] 2bit，模式控制信号

  输出：

  - **pc_out**：32bit，输出pc
  - **nop_out**：是否为nop

- **IF_ID_reg**

  IF 和 ID 之间的 stage register

  输入：

  - **cpu_clk/rst**
  - **nop_in**：[from pc_reg] 上一个stage register是否为nop
  - **sr_ctrl**
  - **inst_in**：[from pc_reg] 32bit，指令输入
  - **pc_in**：[from pc_reg] 32bit，地址输入

  输出：

  - **nop_out**
  - **inst_out**：32bit，指令输出
  - **pc_out**：32bit，地址输出
- **ID_forward**

  用于处理 ID 后面的forward

  输入：

  - **ID_idx**：[from Decoder] 5bit，ID 使用的寄存器编号
  - **ID_data**：[from Decoder] 32bit，ID 使用的寄存器的值
  - **EX_dst**：[from ID_EX_reg] 5bit，EX 阶段的寄存器目标编号
  - **EX_alu_result**：[from alu] 32bit，EX 阶段的算数运算结果
  - **EX_reg_write**：[from ID_EX_reg] 2bit，EX 阶段写回方式
  - **mem_dst**：[from EX_mem_reg] 5bit，memory阶段寄存器目标编号
  - **mem_alu_result** : [from EX_mem_reg] 32bit，memory阶段算数运算结果
  - **mem_mem_result**: [from memory] 32bit，memory阶段lw结果
  - **mem_reg_write**: [from EX_mem_reg] 32bit，memory阶段写回方式
  - **WB_dst**：[from mem_WB_reg] 5bit，WB阶段寄存器目标编号
  - **WB_alu_result** :[from mem_WB_reg] 32bit，WB阶段算数运算结果
  - **WB_mem_result**: [from mem_WB_reg] 32bit，WB阶段lw结果
  - **WB_reg_write**: [from mem_WB_reg] 32bit，WB阶段写回方式

  输出：

  - **res_data**：32bit，forward 之后的结果
- **harzard_checker：**

  用于判断是否需要插入nop

  输入：

  - **ID_idx1**：[from decoder] 5bit，ID使用的寄存器1编号
  - **ID_idx2**：[from decoder] 5bit，ID使用的寄存器2编号
  - **alu_op_flag**：[from decoder] 是否需要在EX阶段使用寄存器的结果
  - **EX_dst**：[from ID_EX_reg] 当前EX阶段的目标寄存器地址
  - **EX_reg_write**：[from ID_EX_reg] EX阶段的写回方式

  输出：

  - **nop_flag**：是否需要插入nop
- **ID_EX_reg**

  ID和EX间的stage register，实际上仅起到将ID的输出往后传递的作用，控制信号等与IF_ID_reg相同。输入输出略。

- **EX_forward**

  EX后的forward模块，和ID_forward功能类似，输入输出略。

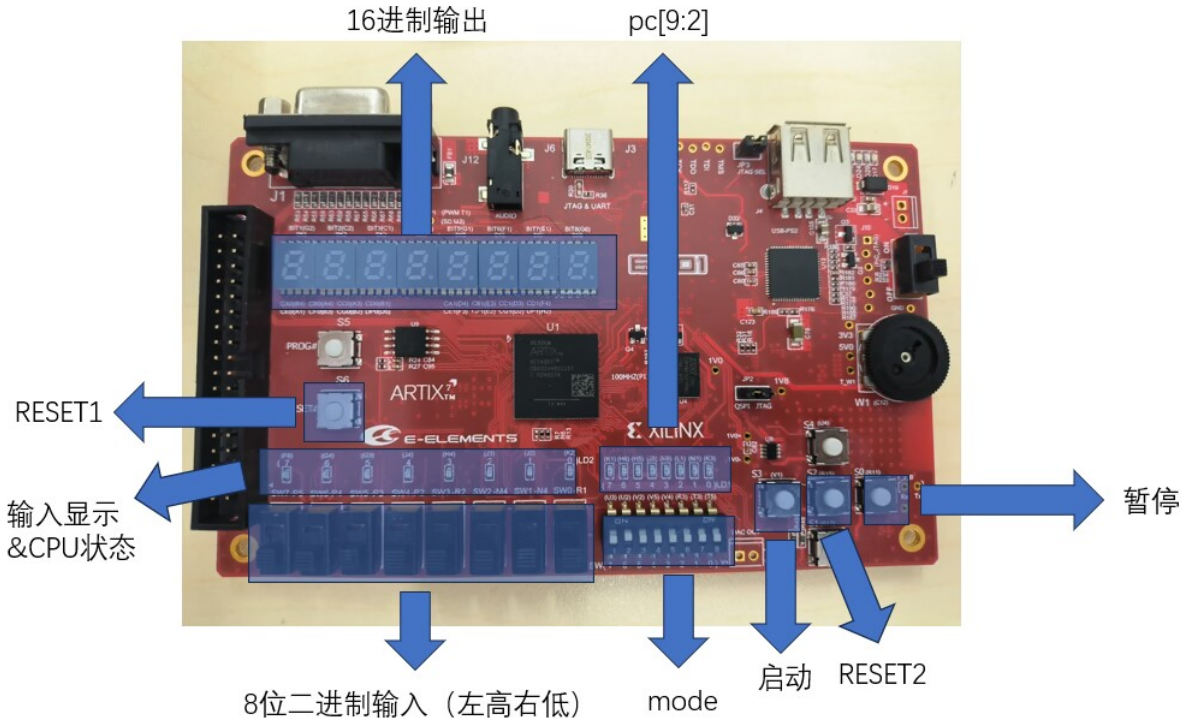
- **EX_mem_reg**

  同上，输入输出略。

- **mem_WB_reg**

  同上，输入输出略。

系统上板使用说明



图中 RESET1 和 RESET2 两个按钮为复位按钮，效果相同，按下任一个即可复位。按下启动按钮即可开始或继续运行程序，按下暂停按钮可暂停程序（如果其在运行）。mode 区域最左侧开关为状态切换开关，拨上为 UART 模式，拨下为正常模式；最右侧按钮为显示切换按钮，拨上时大拨码开关上方 LED 显示输入信息，拨下时其最右侧两个 LED 显示 CPU 状态（00-初始，01-运行，10-暂停，11-结束）。CPU 输出可通过 8 位数码管观测，为其 16 进制形式。

自测试说明

测试内容	测试类型	结果
ALU，Decoder 功能测试	单元，仿真	通过
lb, lh, sb, sh 效果测试	单元，仿真	通过
基础模块综合功能测试	集成，仿真	通过
pipeline 效果测试	集成，仿真	通过
分支预测测试	集成，仿真	通过
FPGA IO 测试	单元，上板	通过
各类型指令专题测试	集成，上板	通过
多种 hazard 测试	集成，上板	通过
任意大规模汇编代码测试	集成，上板	通过

结论：CPU 各种指令执行无误，可以执行较复杂的代码，通用性良好。

Bonus

流水线及分支预测

stage register 和时钟周期

为了实现流水线，我们在每个模块之间插入stage register，接收上一级的输入并将输出传递给下一级。除此之外，stage register还能存储一些和当前层指令相关的额外信息，如目标寄存器地址、写回方式等。这些信息能够帮助流水线进行 data forward 以及 hazard 判断。

在每个时钟上升沿，stage register 基于输入更新输出，将新的信息给到后面的 CPU 子模块中。而 IF 以及 memory 模块在每个时钟下降沿进行数据读写，从而保证数据的时效性。

control plane 和流水线控制

stage register 当然不止起到数据存储/传递的功能，它的一个更为重要的功能是控制流水线的流动。

在每个 stage register 中，我们还额外传递两个信息：`nop_in`和 `sr_ctrl`，分别表示上一级流水线是否为nop（空操作）以及 control plane 发来的控制信号。在每个时钟上升沿，stage register 有三种行为模式：

- 通常模式（normal）：当控制信号为 `SR_NORMAL`，正常传递指令。
- 无操作模式（nop）：当控制信号为 `SR_NOP`或者 `nop_in`为 1，将输出重置为默认值，并且将 `nop_out`设置为 1。
- 保持模式（remain）：当控制信号为 `SR_REMAIN`，忽略输入，保持输出不变。

以下为展示了 `IF_ID_reg`的具体实现：

```
module IF_ID_reg(
    input rst, input cpu_clk, input nop_in, input [1:0]sr_ctrl,
    input [31:0]inst_in, input [31:0] pc_in,
    output reg nop_out,
    output reg [31:0] inst_out, output reg [31:0] pc_out
    );
    `include "constants.v"
    always @(posedge cpu_clk or negedge rst)
        begin
            if (~rst) begin
                inst_out <= 0;
                nop_out <= 1;
                pc_out <= 0;
            end
            else begin
                case (sr_ctrl)
                SR_NOP: begin
                    inst_out <= 0;
                    nop_out <= 1;
                    pc_out <= 0;
                end

                SR_REMAIN: begin
                //remain unchanged
                end
                default: begin
                    if (nop_in) begin
```

```
                        inst_out <= 0;
                        nop_out <= 1;
                        pc_out <= 0;
                    end
                    else begin
                        inst_out <= inst_in;
                        nop_out <= 0;
                        pc_out <= pc_in;
                    end

                end
                endcase

            end
        end

endmodule
```

每个stage register 的 `sr_ctrl` 信号由 control plane 提供。control plane 在每个时钟下降沿（然而这种方式有问题，下文会提出解决方案）更新状态以及流水线控制信号。实际上，control plane 定义了流水线的四种状态：

`ST_REST` 、 `ST_RUN`、 `ST_PAUSE`、 `ST_EXIT`。

具体而言：

- `ST_REST`：初始状态，每个 stage_register 都无操作

  ```
  pc_ctrl <= SR_NOP;
  IF_ctrl <= SR_NOP;
  ID_ctrl <= SR_NOP;
  EX_ctrl <= SR_NOP;
  mem_ctrl <= SR_NOP;
  ```

- `ST_RUN`：正常运行

  ```
  pc_ctrl <= SR_NORMAL;
  IF_ctrl <= SR_NORMAL;
  ID_ctrl <= SR_NORMAL;
  EX_ctrl <= SR_NORMAL;
  mem_ctrl <= SR_NORMAL;
  ```

- `ST_REMAIN`：暂停状态，仅暂停 IF 和 ID 阶段。

  ```
  pc_ctrl <= SR_REMAIN;
  IF_ctrl <= SR_REMAIN;
  ID_ctrl <= SR_NOP;
  EX_ctrl <= SR_NORMAL;
  mem_ctrl <= SR_NORMAL;
  ```

- `ST_EXIT`：结束阶段，程序停止运行。

```
pc_ctrl <= SR_NOP;
IF_ctrl <= SR_NOP;
ID_ctrl <= SR_NOP;
EX_ctrl <= SR_NORMAL;
mem_ctrl <= SR_NORMAL;
```

`ST_REAMIN` 及 `ST_EXIT` 后面几个stage正常运行的原因是，通常和控制流水线行为有关的信号都是从 Decoder 发出来的，暂停应该是暂停**当前的**指令，而让之前的指令正常运行。

除此之外，在 `ST_RUN` 状态时，流水线还可能有两种特殊行为：

- 当 control plane 收到 `nop_flag` 信号时，流水线在 ID 后面插入一条 nop 指令。这对应 forward 无法解决的 data hazard。

```
pc_ctrl <= SR_REMAIN;
IF_ctrl <= SR_REMAIN;
ID_ctrl <= SR_NOP;
EX_ctrl <= SR_NORMAL;
mem_ctrl <= SR_NORMAL;
```

- 当 control plane 收到 `branch_flag` 信号时，流水线将当前 IF 的指令撤回。这代表分支预测出错。

```
pc_ctrl <= SR_NORMAL;
IF_ctrl <= SR_NOP;
ID_ctrl <= SR_NORMAL;
EX_ctrl <= SR_NORMAL;
mem_ctrl <= SR_NORMAL;
```

状态自动机的切换比较简单，这里就不再赘述了。唯一要提的一点是，decoder 会在执行 ecall 10，或者读到空指令后向 control plane 发送 exit 信号，表示程序结束，control plane 跳转到 `ST_EXIT` 状态。

forward 处理

我们需要在两个地方进行 forward 处理：ID 阶段的末尾和 EX 阶段的末尾。通过分析后面几个阶段的 stage register 寄存器写回方式以及写回地址可以实现对当前寄存器的 forward。zero 寄存器不进行 forward。这里展示 ID_forward 的实现方式：

```
module ID_forward(
    input [4:0] ID_idx, input [31:0] ID_data,
        input [4:0] EX_dst, input [31:0] EX_alu_result, input [2:0]
EX_reg_write,
        input [4:0] mem_dst, input [31:0] mem_alu_result, input [31:0]
mem_mem_result, input [2:0] mem_reg_write,
        input [4:0] WB_dst, input [31:0] WB_alu_result, input [31:0]
WB_mem_result, input [2:0] WB_reg_write,
    output reg [31:0] res_data
    );
    `include "constants.v"
    always @(*)
    begin
        if (ID_idx != 0) begin
            if (EX_dst == ID_idx && EX_reg_write == REG_ALU_W)
                res_data <= EX_alu_result;
```

```
            else if (mem_dst == ID_idx) begin
                res_data <= (mem_reg_write == REG_ALU_W) ? mem_alu_result :
mem_mem_result;
            end
            else if (WB_dst == ID_idx) begin
                res_data <= (WB_reg_write == REG_ALU_W) ? WB_alu_result :
WB_mem_result;
            end
            else
                res_data <= ID_data;
        end
        else
            res_data <= ID_data;

    end

endmodule
```

实际上，我们可以发现除了 **decoder 的数据需要在 EX 阶段使用，且 EX 阶段当前的指令需要在 memory 阶段才能拿到结果（即 load 指令）**之外，所有的 data hazard 都可以使用 forward 解决。而对于上述 forward 解决不了的情况，必须在 IF 后面插入一条 nop 语句。于是可以给出下面的 hazard 检测方式：

```
module hazard_checker(
    input [4:0] ID_idx1, input [4:0] ID_idx2, input alu_op_flag,
          input [4:0] EX_dst, input [2:0] EX_reg_write,
    output nop_flag
    );
    `include "constants.v"
    assign nop_flag = (EX_dst != 0) && alu_op_flag &&
                    (ID_idx1 == EX_dst || ID_idx2 == EX_dst) &&
                    (EX_reg_write == REG_MEM_W);

endmodule
```

`nop_flag` 直接连给 control plane，实现 nop 的插入。

跳转和分支预测

如果在 EX 阶段拿到比较的结果进行跳转显然太晚了，更好的方式是在 ID 阶段就拿到比较的结果。这可以通过将跳转检测模块接到 IF_forward 后面实现（这也是我们需要在 ID 末尾而不是 EX 开头拿到 forward 的结果的原因）。

此时出现了一个问题，我们是在 ID 拿到跳转结果的，但是此时 IF 已经是被自动更新到 IF 的下一条指令了。我们可能需要 “撤回” IF 的错误结果。解决方法也非常简单，当 IF 检测到需要跳转时，给 control plane 发一个信号 `branch_flag`，control plane 就能通过 nop 的方式清空 IF 的结果从而撤回不正确的指令。令人惊喜的是，这种机制也自动帮我们实现了分支预测（虽然是预测不会跳转）。

然而，还有一个细节问题需要处理。前面我们说到，memory 是在**下降沿**读取数据，而 control plane 也是在**下降沿**更新信号的。假设memory的数据需要forward 到 ID 里用于分支判断，这时候 control plane 在下降沿是不能拿到正确的分支判断结果的。解决方案也很简单，就是让错开这两个更新的时间点。我们采取了重新设计 control plane 时钟信号，使其在 3/4 时钟周期时才由高电平转为低电平。

ecall 指令的流水线处理

我们对 ecall / ebreak 的指令理解是，中断流水线运行，将控制权转交给上一级控制系统，等到上级控制系统处理完毕后再恢复流水线运行。而当上级控制系统对 cpu 进行操作时，必须保证 cpu 的各个模块的数据都是**最新版本**的（同时 ecall 也需要最新版本的寄存器数据进行指令翻译）。这意味着我们在转交控制权之前需要先等待**之前的指令执行完毕。**整个过程如下：ID 读到 ecall 指令后，发送 `ecall_pause` 信号给 pause_unit；该信号为最高优先级信号，pause_unit 收到后向 control plane 发送暂停信号；control plane 检测后三级流水线状态，当执行完毕后向 ID 发送 `ecall_enable` 信号；ID 清除 `ecall_pause` 信号，翻译指令并将指令交给上级控制系统执行，由上级控制系统决定是否要继续暂停流水线；上级控制系统执行完毕后恢复流水线运行。

由于我们并没有实现如字符串读写、动态内存分配之类的复杂 ecall 指令，所以指令的翻译以及暂停的处理直接写在了 ID 里。

lui、auipc、ecall、ebreak指令的翻译

lui&auipc: 在decoder中将操作数1替换为 0 或 pc，并将 ALU 操作替换为 addi.

ecall&ebreak: 读到该指令时，申请暂停，待该指令前的流水线完成后处理调用。若为 ecall，a7=1则翻译为 sw，由 memory 处理输出。若 a7=5 则翻译为 lw，并申请读入暂停，由 memory 处理输入。若 a7=10 则发出结束信号。若为 ebreak，仅申请读入暂停，不进行读入。

ecall 指令的翻译如下：

```
OP_ENV:begin
        imm = $signed(inst[31:20]);

        if(imm == 0)begin
            if(!ecall_enable)begin
                ecall_pause = 1;
            end else begin
                case(reg_data[REG_A7])
                    10: begin
                        exit = 1;
                        ecall_pause = 0;
                        read_pause = 0;
                    end
                    5:begin
                        mem_op = MEM_IN;
                        dst_idx = REG_A0;
                        read_pause = 1;
                        ecall_pause = 0;
                        reg_write = REG_MEM_W;
                    end
                    1:begin
                        mem_op = MEM_OUT;
                        sw_idx = REG_A0;
                        sw_data = reg_data[REG_A0];
                        mem_write = 1;
                        ecall_pause = 0;
                        read_pause = 0;
                    end
                endcase
            end
        end
        else begin
            if (!ecall_enable) begin
```

```
                ecall_pause = 1;
            end else begin
                read_pause = 1;
                ecall_pause = 0;
            end
//            ecall_pause = 1;
        end
    end
```

测试

由于 data hazard 的处理和 ecall/ebreak 已经在答辩时展示过了，这里只展示分支预测以及auipc / lui 的结果。

测试代码：

```
.text
    li t0, 2
L0:
    addi t0, t0, -1
    bnez t0, L0
    lui t1, 0xff77
    auipc t1, 0x22
```

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x00200293	addi x5,x0,2	2: li t0, 2
	0x00400004	0xfff28293	addi x5,x5,0xffffffff	4: addi t0, t0, -1
	0x00400008	0xfe029ee3	bne x5,x0,0xfffffffc	5: bnez t0, L0
	0x0040000c	0x0ff77337	lui x6,0x0000ff77	6: lui t1, 0xff77
	0x00400010	0x00022317	auipc x6,0x00000022	7: auipc t1, 0x22

运行结果预期：

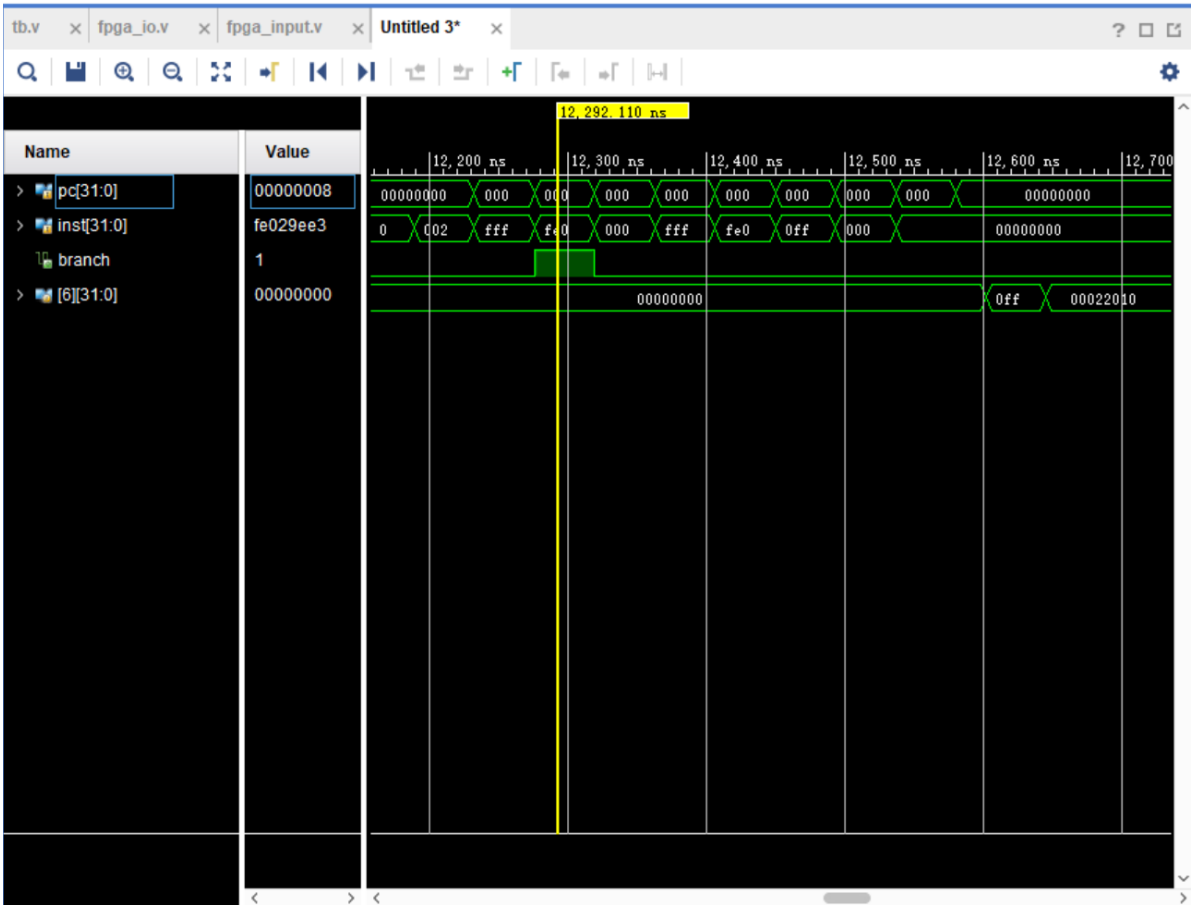
第一次执行bnez时进行跳转，control plane 将当前 IF 的指令替换为 nop；第二次执行 bnez 时不进行跳转，不进行指令替换。

lui 执行后 t1 的值为 0xff77000，auipc 执行后 t1 的值为 0x22010（和 rars 结果不同，因为 rars 的 pc 下标是从 0x400000 开始的）。

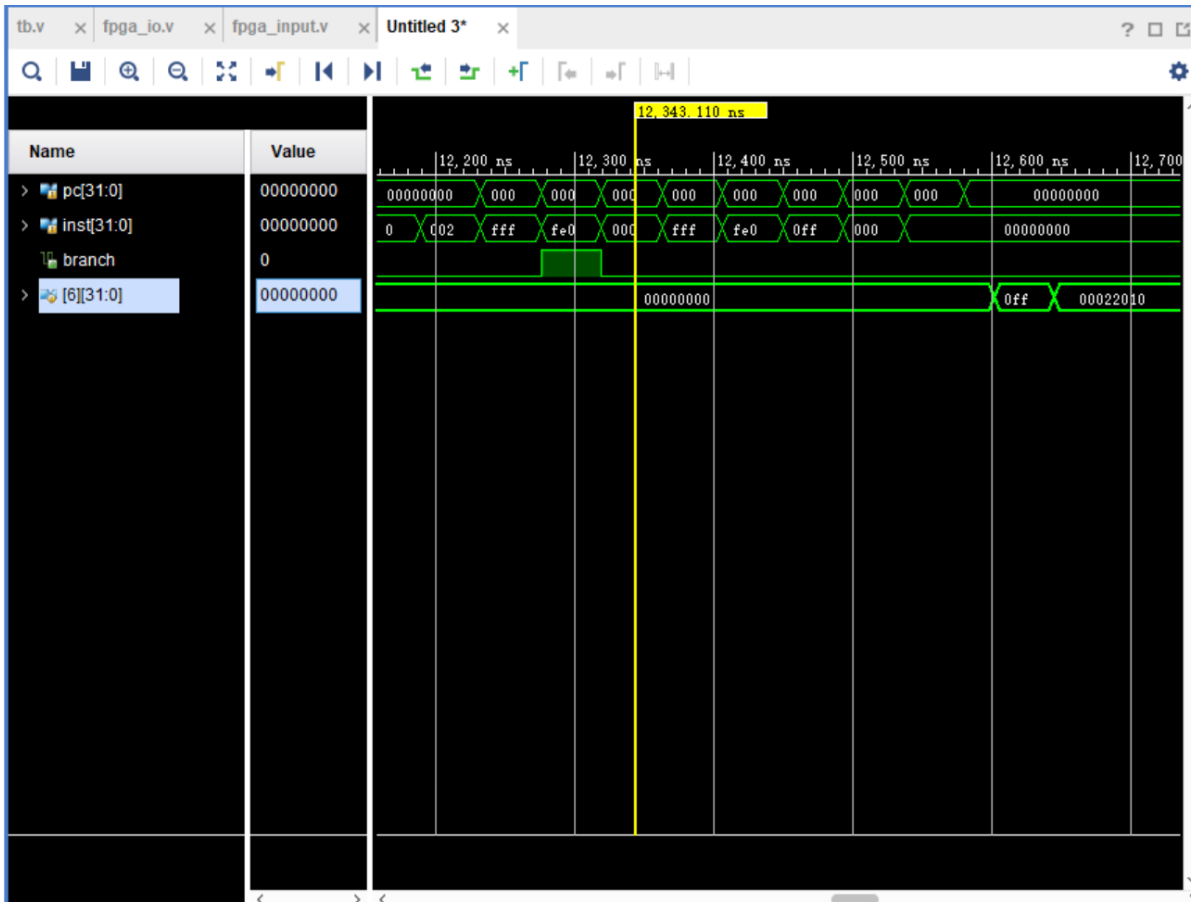
仿真结果展示：

pc，inst 为 ID 的输入；branch 为 branch check 模块的输出；reg[6] 是 ID 模块的寄存器值，代表 x6。

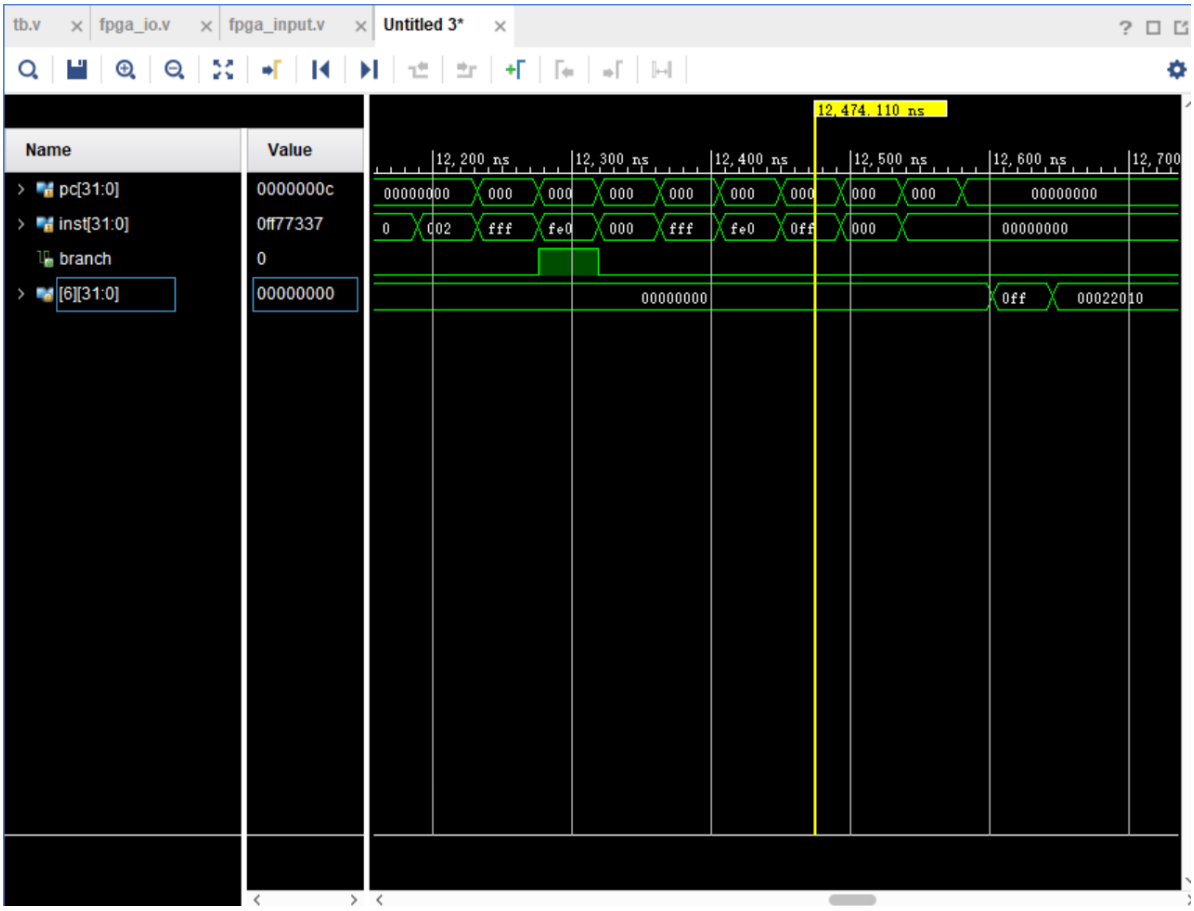
branch 信号：



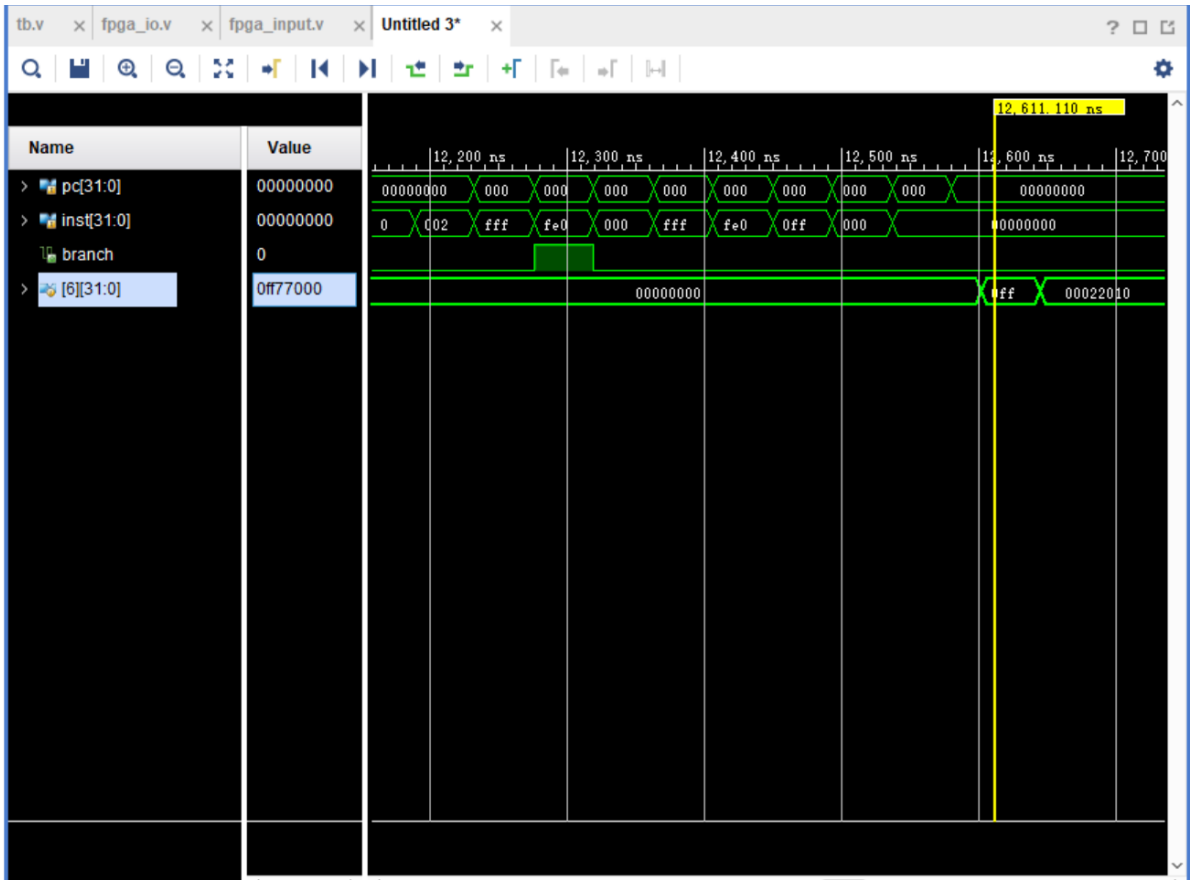
指令替换（预测失败）：



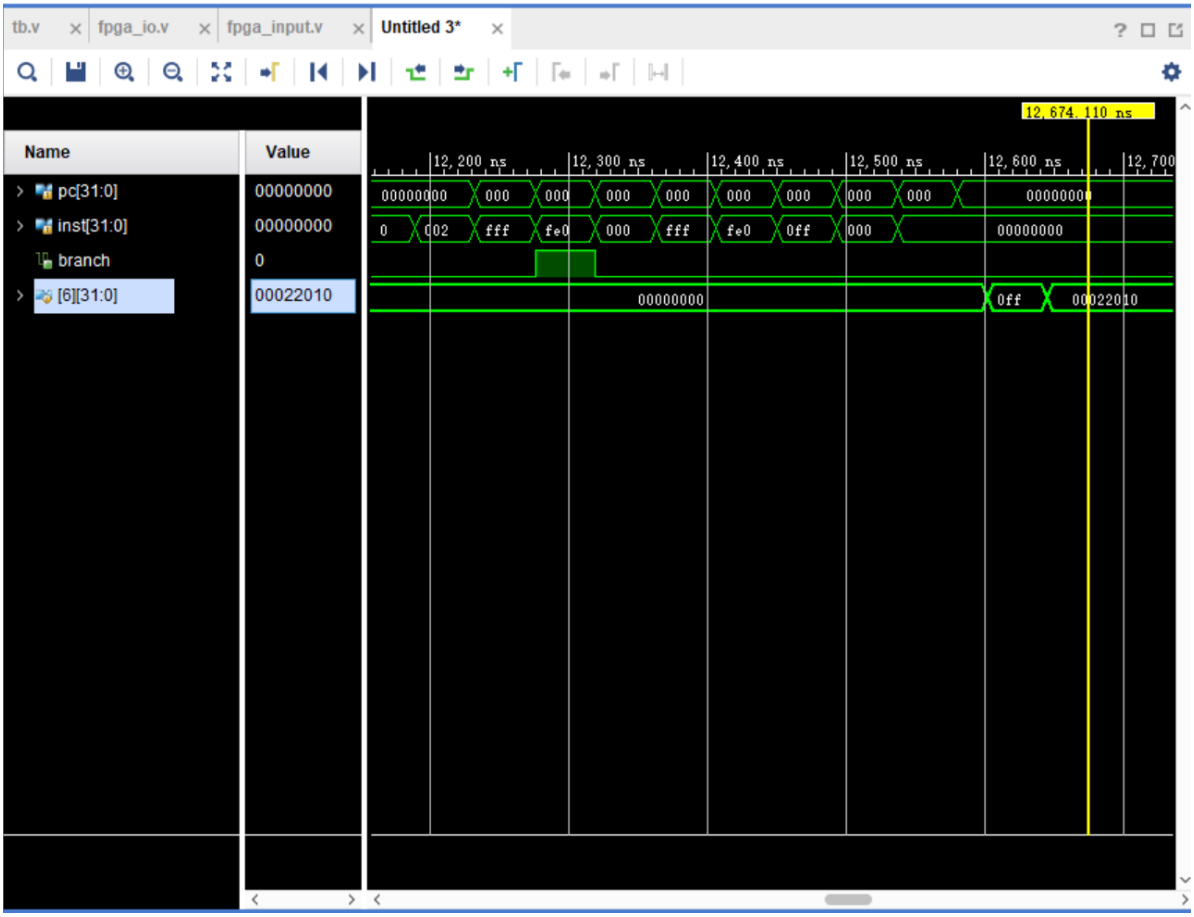
不替换（预测成功）：



lui 结果：



auipc 结果：



问题与总结

实际上，这次 project 的开发过程相当的顺利。除去两三个不太聪明的错误（高电平 reset / 低电平 reset、按键消抖写错），基本没有遇到任何模块/架构上的问题。甚至一解决那几个小错误之后就能过所有测试了。我们算了一下，两个人写代码的时间（不包含顶层模块连线，那玩意太阴间了）加起来可能也就 5~6 个小时。之前经常听到有同学说 verilog 难写，实际上，只要把整个框架的设计思路搞清楚了，写起来是相当简单的（上学期我们也只用了 2~3 天就速通了）。

不过，这次 project 也有一些可以改进的地方：

- 更高效的 verilog 代码编写方式：比如用 Scala 的 Chisel 框架写代码，然后再调用框架将 Scala 编译运行然后生成 Verilog 代码。
- 更强大的 UART 交互模块：实现开发板向 PC 端的数据传输，从而真正做到把 FPGA 作为一个外接 CPU 芯片运行 PC 程序。
- 更丰富的 ecall 指令：比如实现字符串读写（连续内存写入/读取）等。