



2. 参考的ISA为RISC-V。
3. 基于参考ISA，本次作业的更新为删减了一些在testcases中不需要使用的指令。

**寄存器（位宽和数目）等信息）；**

re:

1. 寄存器的位宽为32bit，共有32个寄存器
2. 寄存器中设置了zero, sp寄存器

**对于异常处理的支持情况。**

re:

1. 没有设置异常处理。

**CPU 时钟、CPI，属于单周期还是多周期CPU，是否支持 pipeline（如支持，是几级流水，采用什么方式解决的流水线冲突问题？**

re:

1. 为了保证板子的正常交互，本项目的CPU时钟为500Hz。
2. CPI为 5.
3. 多周期CPU，支持pipeline，五级流水线（IF, ID, EX, MEM, WB）。
4. 采用数据前递的方式来解决数据冒险（详细介绍在后面bonus介绍部分）。
5. 使用分支预测的方式来解决控制冒险（详细介绍在后面bonus介绍部分）。

**寻址空间设计：属于冯·诺依曼结构还是哈佛结构；**

re:

1. 寻址空间的架构为哈佛架构

**寻址单位，指令空间、数据空间的大小，栈空间的基地址。**

re:

1. 寻址单位：word
2. 指令空间：32\*16384
3. 数据空间：32\*16384 (bit)
4. 栈空间的基地址：0x2fff

**对外设IO的支持：采用单独的访问外设的指令（以及相应的指令）还是MMIO（以及相关外设对应的地址），采用轮询还是中断的方式访问IO。**

re:

IO: 采用单独的访问外设指令MMIO，数据流走向为：

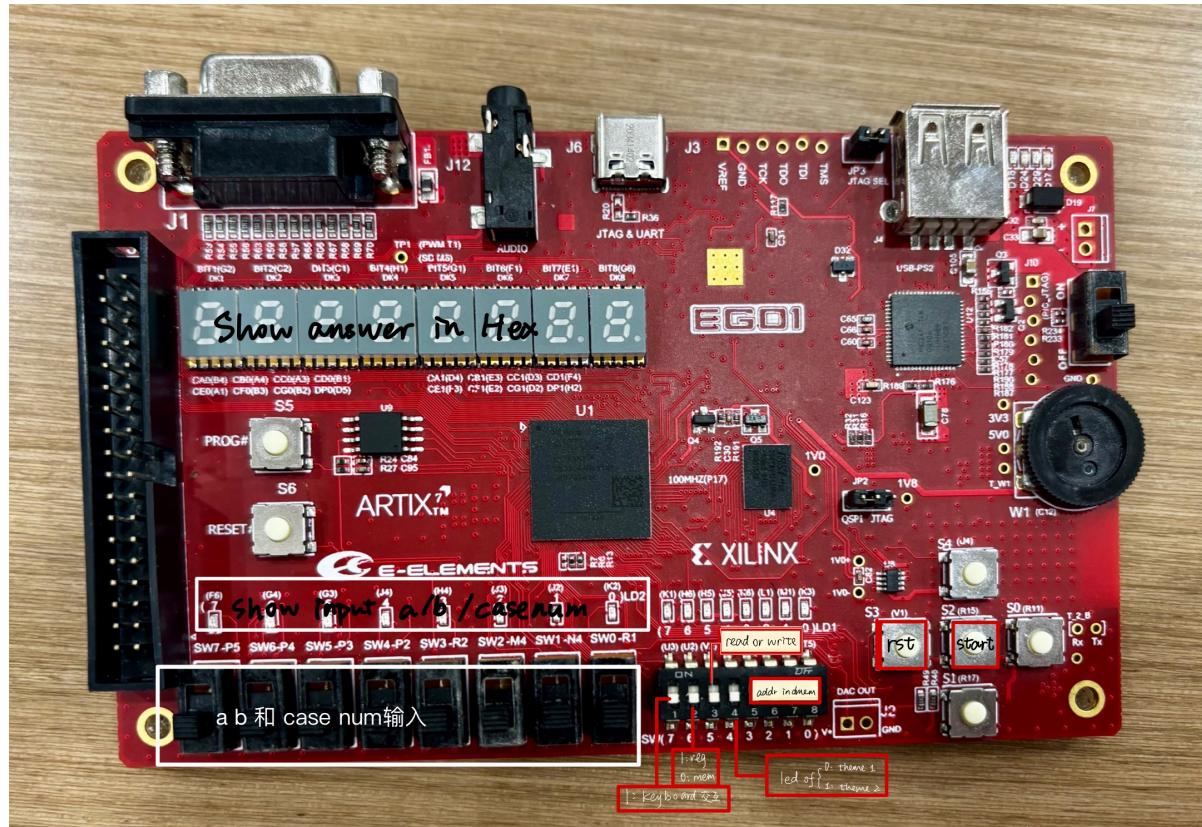
IN: board --> dmem特定地址 --> 通过load指令将数据搬运至reg做运算；

OUT: (1) 从a0 reg --> board LED； (2) 从特定几个dmem --> board Seg.

访问方式：中断方式访问IO，需要在运行coe文件前按顺序输入caseNum, a, b; 后按下start按钮，执行coe文件。

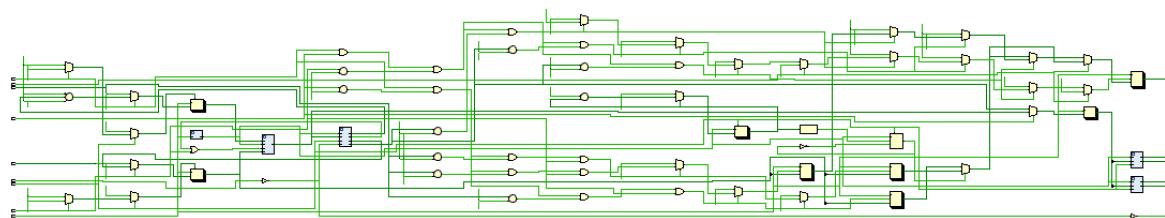
## CPU接口

CPU接口：时钟、复位、uart接口、其他常用IO接口说明。



## CPU内部结构

CPU内部各子模块的接口连接关系图



CPU内部子模块的设计说明（子模块端口规格及功能说明）

re:

```

    ▼ uCPU : CPU_Main_Basic (CPU_Main_Basic.v) (11)
      ▼ IM : Instr_Memory (Instr_Memory.v) (1)
        > urom : program (program.xci)
        ▼ ControllerU : Controller (Controller.v)
      ▼ IF : IFetch (IFetch.v) (1)
        ▼ pc_reg_inst : PC_Reg (PC_Reg.v)
        ▼ IF_ID_Reg : IF_ID_Reg (IF_ID_Reg.v)
      ▼ ID : Instruction_decode (Instruction_decode.v) (2)
        ▼ register : Registers (Registers.v)
        ▼ imm_G : Imm_gen (Imm_gen.v)
        ▼ ID_EX_Reg : ID_EX_Reg (ID_EX_Reg.v)
      ▼ EX : Execute (Execute.v) (4)
        ▼ alu : ALU (ALU.v)
        ▼ forward_unit_inst : forward_unit (forward_unit.v)
        ▼ mux3_1_forwardA : mux3_1 (mux3_1.v)
        ▼ mux3_1_forwardB : mux3_1 (mux3_1.v)
        ▼ EX_MEM_Reg : EX_MEM_Reg (EX_MEM_Reg.v)
      ▼ MEM : memory (memory.v) (1)
        ▼ mem_mux : mux (MUX_32.v)
        ▼ MEM_WB_Reg : MEM_WB_Reg (MEM_WB_Reg.v)
        ▼ WB : Write_Back (Write_Back.v)

```

为区别下面流水线寄存器这里只对每个阶段的输入输出端口：

```

module Instr_Memory(
  input clk,
  input [31:0] pc,
  output [31:0] instruction
);

module IFetch(
  input clk,
  input rst,
  input start,
  input pc_jump,
  input jalr,
  input [31:0] ALU_result_mem_wb_o,
  input [31:0] imm_jump,
  input [31:0] pc_if_i,
  output [31:0] pc_if_o,
  output reg [31:0] pc_next,
  input load_use_flag
);

module Instruction_decode(
  input wire clk1, rst1,
  input wire Reg_write1,//sign
  input wire [4:0] Rd_reg_i,
  input wire [31:0] write_data1,
  input wire [31:0] instruction,
  input wire jal,
  input wire jalr,
  input wire [2:0] funct3_mem_wb_o,
  input wire load,
  output wire [2:0] func3,

```

```

output wire [6:0] func7,
output wire [31:0] Read_data_1,
output wire [31:0] Read_data_2,
output wire [31:0] immediate,
output wire [4:0] write_reg,
output wire [6:0] opcode,
output wire [4:0] Read_reg1,
output wire [4:0] Read_reg2,
output wire [31:0] reg_to_top//传到顶层的寄存器
);

module Execute(
    input clk,
    input rst,
    input [31:0] pc_ex_i,
    input [31:0] imme_ex_i,
    input [31:0] Rd_data1_ex_i,
    input [31:0] Rd_data2_ex_i,

    output [31:0] ALU_result_ex_o,
    output [31:0] Rd_data2_ex_o,
    output [31:0] imme_ex_o,
    output pc_jump,

    //下面为数据前递判断所需的
    input Branch_ex_i,
    input MemRead_ex_i,
    input MemtoReg_ex_i,
    input [1:0] ALUOp_ex_i,
    input MemWrite_ex_i,
    input ALUSrc_ex_i,
    input RegWrite_ex_i,
    input [2:0] funct3_ex_i,
    input [6:0] funct7_ex_i,
    input jal_ex_i,
    input jalr_ex_i,

    input [4:0]Rs1_ex_i,
    input [4:0]Rs2_ex_i,
    input [4:0]Rd_ex_mem_o,
    input [4:0]Rd_mem_wb_o,
    input RegWrite_ex_mem_o,
    input RegWrite_mem_wb_o,
    input MemWrite_id_ex_o,
    input MemRead_ex_mem_o,
    input [31:0] ALU_result_ex_mem_o,
    input [31:0] ALU_result_mem_wb_o,
    output forwardC,

    input [4:0]Rs1_id_ex_i,
    input [4:0]Rs2_id_ex_i,
    input [4:0]Rd_id_ex_o,
    input MemRead_id_ex_o,
    input MemWrite_id_ex_i,
    input RegWrite_id_ex_o,

```

```

    output load_use_flag
);

module data_Memory(
    input wire clk,
    input mem_write,
    input [31:0] address,
    input [31:0] write_Data,
    output[31:0] read_Data
);

module write_Back //该模块真正的作用为确定写回的数据，并将该数据传递到相应的ID阶段进行寄存器写回操作
    input MemtoReg,
    input jal,
    input jalr,
    input [31:0] ALU_result_wb_i,
    input [31:0] loaddata_wb_i,
    input [31:0] imme_wb_i,
    input [31:0] pc_wb_i,
    output reg [31:0] wr_reg_data_wb_o
);

```

说明：上图中的MEM仅为数据前递的读入数据的判断，真正的MEM在顶层IO顶层例化。

上面端口输入输出端口采取了pipeline设计与单周期CPU的输入输出有较大的不同，对于每个信号都进行了阶段的区分。

需要额外声明的是对于WB阶段，因为寄存器子模块是在ID阶段例化，所以WB实际执行代码在ID模块中，这里体现了流水线中指令并行执行的同步思想。除此之外，各模块均为常规设计，仅在IF阶段中加入PC寄存器子模块为了让整个CPU更贴近于实际使用。

### 3 系统上板使用说明

**开发板上与系统操作相关输入、输出操作说明。（如复位使用的输入设备、如何实现复位；CPU工作模式切换的按键及如何实现模式选择；输出信号的观测区域，与输出数据的对应关系等）**

re:

输入操作说明：

(注：输入只能从board到dmem, seg\_ctrl(小拨码开关U2)一直保持低电平)

**输入一个case：**

1. 上电，打开FPGA电源开关，烧比特流
2. 将keyboard\_input\_enable(小拨码开关U3)置于高电平
3. 输入caseNum
  - a. 将address\_ctrl(小拨码开关右边四个)置于4'b0000
  - b. 将boardCtrlRW(小拨码开关V2)置于高电平
  - c. 波动左侧8bits大拨码开关作为输入caseNum
  - d. 当七段数码管显示输入的值即为已经读入操作数

- e. 关闭boardCtrlRW
4. 输入a
    - a. 将address\_ctrl(小拨码开关右边四个)置于4'b0001
    - b. 将boardCtrlRW(小拨码开关V2)置于高电平
    - c. 波动左侧8bits大拨码开关作为输入二进制a的值
    - d. 当七段数码管显示输入的值即为已经读入操作数
    - e. 关闭boardCtrlRW
  5. 输入b
    - a. 将address\_ctrl(小拨码开关右边四个)置于4'b0010
    - b. 将boardCtrlRW(小拨码开关V2)置于高电平
    - c. 波动左侧8bits大拨码开关作为输入二进制b的值
    - d. 当七段数码管显示输入的值即为已经读入操作数
    - e. 关闭boardCtrlRW
  6. 关闭keyboard\_input\_enable

#### **运行与结果观测：**

1. 长按start(右侧按键中心键R15)运行coe
2. 将keyboard\_input\_enable(小拨码开关U3)置于高电平
3. 若需要读取dmem的值
  - a. 将seg\_ctrl(小拨码开关U2)置于低电平
  - b. 将boardCtrlRW置于低电平
  - c. 将address\_ctrl(小拨码开关右边四个)置于目标地址 (如测试场景一的case001的答案需要拨至4'b0100; case010的答案需要拨至4'b1000)
  - d. 对应dmem的值会显示在七段数码管上
4. 若需展示读入a, b的LED交替显示 (测试场景一的case000, 以及测试场景二的操作数LED输出)
  - a. 将seg\_ctrl(小拨码开关U2)置于高电平
  - b. 大拨码开关上方的LED灯会亮起, 交替展示 (a, b) 或 (前8bits, 后8bits的浮点数)
5. 若需展示测试场景一中后5个case
  - a. 将seg\_ctrl(小拨码开关U2)置于高电平
  - b. 若比较关系成立, 大拨码开关上方的LED灯会点亮8'b1000\_0000
  - c. 若比较关系不成立, 大拨码开关上方的LED灯会点亮8'b0000\_1000

#### **不同case的切换：**

1. 按下rst按键(右侧按键左键V1)清空寄存器  
输入下一个case, 步骤同前面所说, 若a, b其中某值不变则可省略其输入步骤。

## 4 自测试说明

以表格的方式罗列出测试方法（仿真、上板）、测试类型（单元、集成）、测试用例（除本文及OJ以外的用例）描述、测试结果（通过、不通过）；以及最终的测试结论。

re:

## testcase2部分：

测试编号	测试方法	测试类型	测试用例	用例描述	预期结果	实际结果	测试结论(通过/不通过)	问题总结
1	仿真	单元	00010010	测试 test2的 000功能	3	3	通过	
2	仿真	单元	0x3400	测试 test2的 001功能	1	1	通过	
3	仿真	单元	0x3400	测试 test2的 010功能	0	0	通过	
4	仿真	单元	0x3400	测试 test2的 011功能	0	0	通过	
5	仿真	单元	11111011、01111011	测试 test2的 100功能	10001000	10001000	通过	
6	仿真	单元	0x3AC	测试 test2的 101功能	C03	C03	通过	
7	仿真	单元	5	测试 test2的 011功能	0x22	0x22	通过	
8	仿真	单元	5	测试 test2的 100功能	将a0的值存到 内存地址为0xc 的位置	存储成功	通过	
9	上板	集成	前8个测试用例的 testcase number 和测试数据	能正常实现功能	第八个 testcase未能实现功能	不通过	顶层未做 FSM，无法实现轮流展示	

## 5 Bonus说明

### pipeline的设计说明

pipeline的总体设计采用流水线寄存器处理，共分为五级流水线，分别为IF、ID、EX、MEM、WB五个阶段。pipeline中对控制冒险采用了分支预测的方式来处理，对于数据冒险采用了数据前递的方式处理。本项目中流水线CPU应用于整个项目，包括两个基本测试场景。

图示为流水线CPU的各模块结构图，其中在IM中独立进行pc地址到指令的获取，然后依次进行对单个指令依次进行IF、ID、EX、MEM、WB。所有后续要使用到的信号都由前面的reg寄存器依次传递，确保在正确的时序逻辑下获取到相应的信号。该部分较前面的主要区别即添加了流水线寄存器，以及在EX阶段进行数据前递判断，在EX、MEM阶段增加读入数据判断。

```
✓ uCPU : CPU_Main_Basic (CPU_Main_Basic.v) (1)
  ✓ IM : Instr_Memory (Instr_Memory.v) (1)
    > urom : program (program.xci)
    ✓ ControllerU : Controller (Controller.v)
  ✓ IF : IFetch (IFetch.v) (1)
    ✓ pc_reg_inst : PC_Reg (PC_Reg.v)
    ✓ IF_ID_Reg : IF_ID_Reg (IF_ID_Reg.v)
  ✓ ID : Instruction_decode (Instruction_decode.v) (2)
    ✓ register : Registers (Registers.v)
    ✓ imm_G : Imm_gen (Imm_gen.v)
    ✓ ID_EX_Reg : ID_EX_Reg (ID_EX_Reg.v)
  ✓ EX : Execute (Execute.v) (4)
    ✓ alu : ALU (ALU.v)
    ✓ forward_unit_inst : forward_unit (forward_unit.v)
    ✓ mux3_1_forwardA : mux3_1 (mux3_1.v)
    ✓ mux3_1_forwardB : mux3_1 (mux3_1.v)
    ✓ EX_MEM_Reg : EX_MEM_Reg (EX_MEM_Reg.v)
  ✓ MEM : memory (memory.v) (1)
    ✓ mem_mux : mux (MUX_32.v)
    ✓ MEM_WB_Reg : MEM_WB_Reg (MEM_WB_Reg.v)
    ✓ WB : Write_Back (Write_Back.v)
```

### 控制冒险

实现了分支预测，对于PC地址每次做+4操作，当遇上跳转指令时会冲刷掉被多取出来的那条指令（指令的冲刷即在ID阶段将所有控制信号重置为0，这样往下传递在后面的各个阶段不会进行任何操作），并且跳转到正确的PC地址。即在IF阶段优先进行地址+4操作，通过同一时刻在EX阶段生成的 `pc_jump` 信号实现指令的跳转（跳转指令要在EX阶段计算出指定地址）。该方面的处理支持beq、jal、jalr等多种指令。

```
module IFetch(
  input clk,
  input rst,
  input start,
  input pc_jump,
  input jalr,
  input [31:0] ALU_result_mem_wb_o,
  input [31:0] imm_jump,
  input [31:0] pc_if_i,
  output [31:0] pc_if_o,
  output reg [31:0] pc_next,
  input load_use_flag
);
```

```

PC_Reg pc_reg_inst (
    .clk(clk),
    .rst(rst),
    .pc(pc_if_i),
    .pc_out(pc_if_o),
    .load_use_flag(load_use_flag)
);

always@(posedge clk) begin
    if(rst || !start)
        pc_next <= 32'hffffffff;
    else if(load_use_flag)
        pc_next <= pc_if_i;
    else if(jalr)
        pc_next <= ALU_result_mem_wb_o;
    else if(pc_jump)
        pc_next <= pc_if_i - 4 + imm_jump;
    else //add 4
        pc_next <= pc_if_i + 4;
end

endmodule

```

## 数据冒险

数据冒险采用数据前递方式，根据指令类型分为load类型处理、其他运算类型处理。

- load类型：load类型在流水线中的MEM中得到所需数据，此时下一条指令在EX阶段。我们将load指令称为指令1，load指令的下一条指令称为指令2(例如 `lw t0, 0(t1)`      `addi t2, t0, t3`)。该数据冒险发生在指令2的运算阶段需要使用到指令1 load到的数据进行运算，该类型的数据冒险需要进行pipeline的延时处理，即在指令2的执行过程中停顿一个周期，确保指令1的MEM阶段执行后，指令2才开始执行EX阶段。此处用到的处理依旧是选择冲刷掉指令1后一条指令来达到延时效果（指令的冲刷方式与上述方式相似）。
- 其他：对于连续对相同寄存器进行写后读的操作，因为写操作发生在WB阶段而读操作发生在EX阶段，数据往往还没有写入到相应寄存器中就要进行读操作。本项目中处理该类型的数据冒险问题采用数据前递的方式，通过前递数据来处理冒险。

数据冒险的主要判断信号生成模块如下

```

module forward_unit(
    input [4:0]Rs1_id_ex_o,
    input [4:0]Rs2_id_ex_o,
    input [4:0]Rd_ex_mem_o,
    input [4:0]Rd_mem_wb_o,
    input RegWrite_ex_mem_o,
    input RegWrite_mem_wb_o,
    input MemWrite_id_ex_o,
    input MemRead_ex_mem_o,

    output [1:0]forwardA,
    output [1:0]forwardB,
    output forwardC,
);

```

```

    input [4:0]Rs1_id_ex_i,
    input [4:0]Rs2_id_ex_i,
    input [4:0]Rd_id_ex_o,
    input MemRead_id_ex_o,
    input MemWrite_id_ex_i,
    input RegWrite_id_ex_o,

    output load_use_flag

);

assign forwardA[1]=(RegWrite_ex_mem_o &&(Rd_ex_mem_o!=5'd0)&&
(Rd_ex_mem_o==Rs1_id_ex_o));
assign forwardA[0]=(RegWrite_mem_wb_o && (Rd_mem_wb_o !=5'd0) &&
(Rd_mem_wb_o==Rs1_id_ex_o));

assign forwardB[1]=(RegWrite_ex_mem_o &&(Rd_ex_mem_o!=5'd0)&&
(Rd_ex_mem_o==Rs2_id_ex_o));
assign forwardB[0]=(RegWrite_mem_wb_o && (Rd_mem_wb_o !=5'd0) &&
(Rd_mem_wb_o==Rs2_id_ex_o));

//load后紧跟sw但是不需要停顿
assign forwardC=(RegWrite_ex_mem_o &&(Rd_ex_mem_o!=5'd0)&&
(Rd_ex_mem_o!=Rs1_id_ex_o)&& (Rd_ex_mem_o==Rs2_id_ex_o)&& MemWrite_id_ex_o &&
MemRead_ex_mem_o );

//load-use
assign load_use_flag= MemRead_id_ex_o & RegWrite_id_ex_o &
(Rd_id_ex_o!=5'd0) //load
& (!MemWrite_id_ex_i) //非store
& ((Rd_id_ex_o ==Rs1_id_ex_i) | (Rd_id_ex_o
==Rs2_id_ex_i))
;

endmodule

```

下面为利用数据前递信号进行数据选择的实现

```

mux3_1 mux3_1_forwardA (
    .din1(ALU_result_ex_mem_o),
    .din2(ALU_result_mem_wb_o),
    .din3(Rd_data1_ex_i),
    .sel(forwardA),
    .dout(A)
);

```

## 测试说明

该部分实现了自测，具体的上板测试结果在视频中有所展示，这里只展示case1的仿真波形图来证明控制冒险以及数据冒险的正确实现。而在测试场景的case1、case2、case6均有相关数据冒险的发生，最终结果也准确无误。

```
lw x1, 0(x0)
addi x10, x0, 1
addi x11, x0, 2
beq x1, x10, case1
beq x1, x11, case2
```

case1: #数据冒险可能发生的两种情况:

- # 1: 发生在前一条指令的MEM时期
- # 2: 发生在前面隔一条指令的WB时期

```
addi x2, x0, 2
addi x3, x0, 3
add x4, x2, x3 # ----- 1、2都存在
add x5, x3, x4 # ----- 1、2都存在
sw x2, 4(x0)
sw x3, 8(x0)
sw x4, 12(x0)
sw x5, 16(x0)
```

```
j end
```

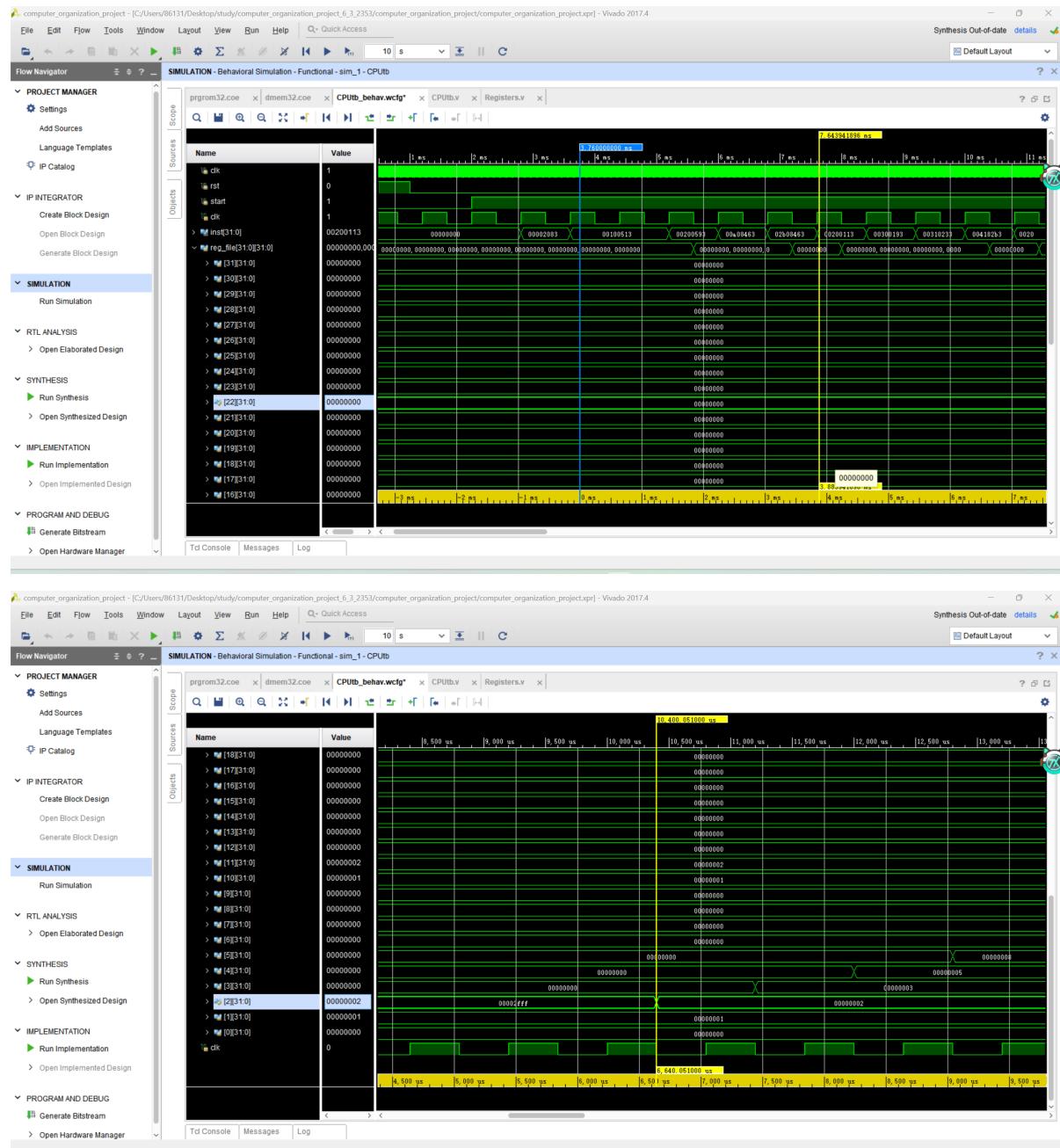
case2: # 数据冒险可能发生的另外两种情况关于lw指令

- # 1: load指令的下一条指令在EX阶段要使用到所load到的值，此时load指令在MEM阶段
- # 2: load指令的下一条指令在MEM阶段要使用到所load到的值， 此时load指令在WB阶段

```
addi x2, x0, 2
sw x2, 4(x0)
lw x3, 4(x0)
sw x3, 8(x0) #----- 2
lw x4, 4(x0)
add x5, x3, x4 #----- 1
sw x5, 12(x0)
```

```
j end
```

```
end:
```



## 问题及总结

本项目先基于单周期CPU进行IO的实现以及测试场景的汇编代码编写，项目初期进程缓慢，成员之间的交流频率较低，因此在项目前中期的协同上表现较差，该情况直接导致了后续项目的进展。在项目后期，因沟通回复不及时，导致CPU的流水线完成度不够，即对于一些指令如jal、jalr没有实现，导致在项目后期的时间压力较大。

除此之外，对于IO方案，组内成员分歧严重，虽然最终基于个人版本实现，但是同样延缓项目进程。

在项目进程后期有很大的时间压力，无论是CPU的流水线版本因为实现细节的问题导致一些指令无法正常实现如lb、lbu指令，因此增加了CPU的debug成本，也因此延缓了上板测试的时间。

对于IO模块的处理不清晰导致了一些测试结果能得到答案却无法正常展示出来，令人十分遗憾。