

重庆大学课程设计报告

课程设计题目： MIPS SOC 设计与性能优化

学 院: 计算机学院

学 生: ytz xxx

MIPS SOC 设计报告

ytz,xxx

1 设计简介

本设计实现了简单的 MIPS SOC, 主要是一个支持 AXI 总线的 CPU, 可以执行 57 条指令, 支持异常处理。

基于简易五级流水线 CPU, 通过改写 maindecoder 和 aludecoder 以及对应的 datapath 和 hazard 控制, 对冒险进行处理(暂停和前推), 从原先实现的几条简单指令拓展至 57 条指令并添加了异常处理模块, 通过了各组测试样例。在顶层添加了 sram 接口并通过了 SOC 测试(基础测试、延迟槽测试、异常测试, 共 89 个测试点)。将 sram 接口转为类 sram 接口, 采用龙芯杯提供的转接桥连接 AXI 总线, 通过了 AXI 测试(功能测试、性能测试)。

此外, 尝试了添加 cache, 但是卡在了第 77 个测试点。

1.1 小组分工说明

- ytz: 配置并测试实验环境。除逻辑、算术指令外的指令扩展, 数据通路的修改, 包含 hilo 寄存器、cp0 读写逻辑, 冒险处理逻辑, 添加异常处理模块。对 57 条指令的基本功能仿真测试, sram 接口的连线以及添加 sram 接口后的仿真测试, 添加 axi 接口后的功能测试, 部分 axi 性能测试。几乎全部的 debug 工作。部分实验报告的编写和校正。
 - xxx: 负责编辑器环境的配置, 逻辑、算术运算指令扩展, 部分 controller 和 alucontrol 接口修改, sram 接口和 axi 接口的提供, axi 接口的连线, axi 功能测试中 part2 的 debug, 部分 axi 性能测试, Cache 设计(最终未实现)。实验报告的编写。
- 其它没有提及的工作由两人共同完成。

2 设计方案

2.1 总体设计思路

设计的 CPU 用于支持基于 AXI 总线的片上系统。具体设计依照图1和硬综讲解 2.pptx 中的方式一, 将 CPU 封装成类 sram 接口并使用龙芯杯提供的类 sram-axi 转接桥。CPU 和 AXI 总线之间的通信都经过这两个接口的转换。将 CPU 封装成类 sram 接口又分为将 CPU 封装成 SRAM 接口和进行 SRAM 接口和类 SRAM 接口之间的转换这两个步骤。出入 CPU 的信号都要经过这些接口的转换。

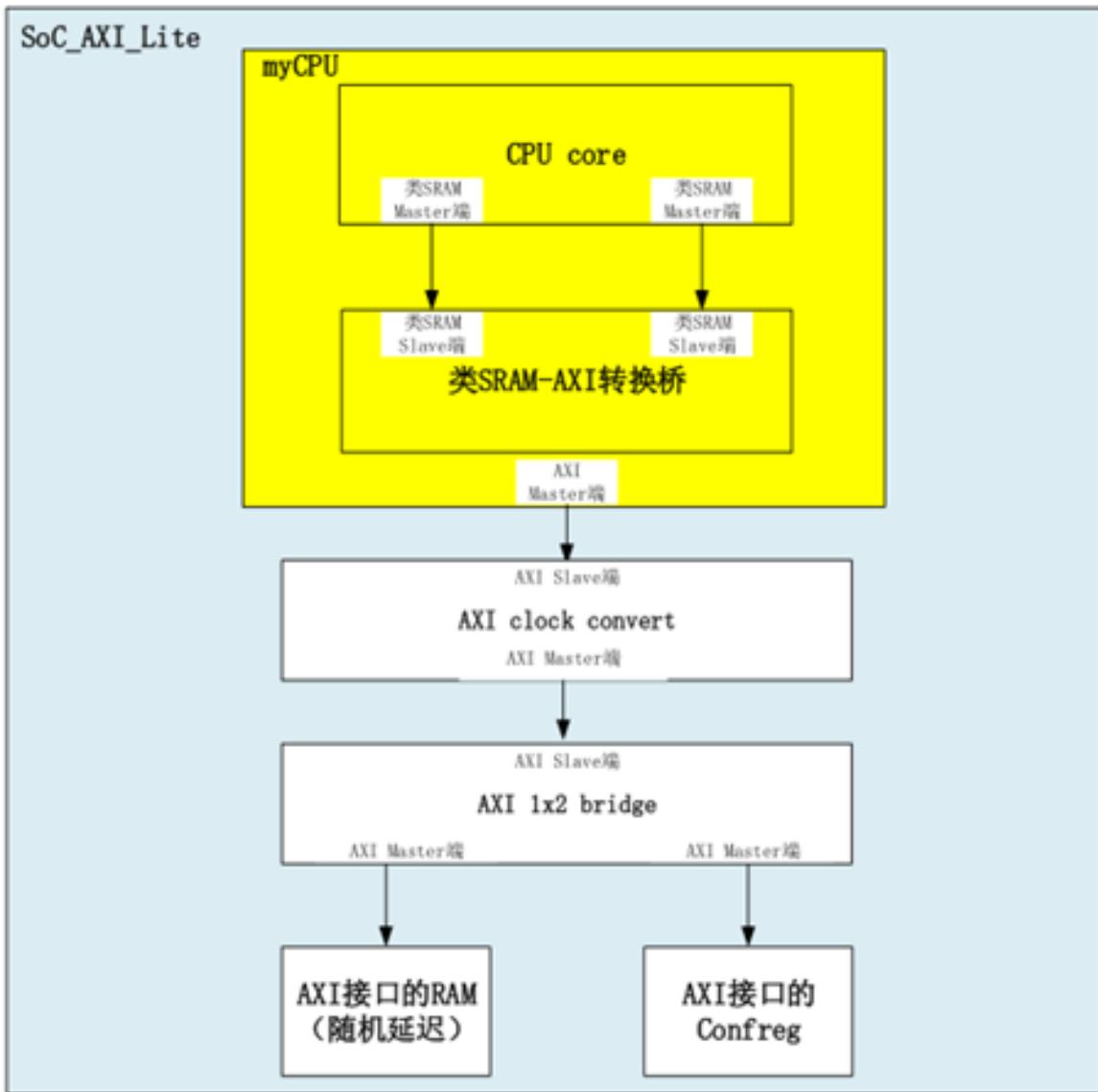


图: 龙芯杯SoC_axi_lite结构
(用于功能测试和性能测试)

图 1: 龙芯杯 SoC_axi_lite 结构 (图片来源:《硬综讲解 2.pptx》)

基于《组成原理》实验 4 实现的简易五级流水线 CPU, 通过修改 CPU 内部逻辑, 主要是 Controller 和 Datapath, 将 10 条指令扩展到 57 条指令, 使 CPU 能够正确执行 57 条指令, 并添加异常处理功能。虽然这些工作分散于整个 CPU 中, 由 Controller 和 Datapath 共同完成, 并不是像 main_decoder 这样的具体的电路模块, 但在设计时把它们抽象成不同的模块可以使设计思路更加清晰。

完成指令添加和异常处理后, 将 CPU 封装成 sram 接口, 并进行功能测试。

连接类 sram 转 axi 总线转接桥, 进行功能测试和性能测试。

2.2 扩展指令设计思路

具体按照《指令及其对应机器码查找表》中划分的模块的顺序来添加,添加时需修改 maindecoder 和 aludecoder 中对应的逻辑,修改 ALU 中对传入数据的处理逻辑,并对 CPU 各个模块的接口进行修改。有些指令还需要对 datapath 的逻辑进行修改与扩充,涉及到冒险的还需要扩充 hazard 模块逻辑。

要添加的 57 条指令被分成六个基本模块和异常处理模块(包含了内陷指令和特权指令的实现)。针对每个模块,下发的资料提供了相应测试文件,每添加一个模块,都有其对应的 1-2 个测试文件。将测试文件 (.coe) 导入实现了该功能模块的 cpu 中,观察输出结果是否与给出的指令一致,若该测试文件的输出波形图均符合预计,则视为该测试点通过,即功能模块已完成拓展。

课程中将 57 条指令划分为 8 个功能模块(详见《指令及对应机器码 _2018.pdf》):

逻辑运算指令:and,or,xor,nor,andi,ori,xori,lui

移位指令:sll,srl,sra,sllv,srlv,srav

数据移动指令:mfhi,mflo,mthi,mtlo

算数运算指令:add,addu,sub,subu,slt,sltu,mult,multu,div,divu,addi,addiu,slti,sltiu

分支跳转指令:jr,jalr,j,jal,beq,bgtz,blez,bne,bltz,bltzal,bgez,bgezal

访存指令:lb,lbu,lh,lhu,lw,sw,sh,sw

内陷指令:break,syscall

特权指令:mtc0,mfc0,eret

参考《MIPS 基准指令集手册 _v1.00.pdf》中提供的每条指令的功能说明以及 PPT 上给出的示例对译码器,alu 和数据通路进行修改,并在需要时引入新的模块。

除法、hilo 寄存器和 cp0 寄存器功能模块是直接调用的资料中提供的文件,在数据移动指令、算术运算指令中的除法指令、特权指令这些指令的添加时需要用到这些功能模块,需要在数据通路中对这三个功能模块进行连线。这部分参考了资料中提供的 ppt 和教学视频。

在信号的命名上,往往根据功能含义等的简写来命名,并添加相应字母(F、D、E、M、W,通常是大写)来标识信号所处的流水级。有些可以看出明显属于某一个流水级的信号没有在信号名中标注流水级。

代码中,由于最早测试每条指令添加是否正确时(人工比对波形图阶段),在仿真图中通过 Objects 添加的信号显示不出来,因此把所有要观察的信号都逐层往上连到了 testbench,后来添加了 sram 接口后可以通过 Objects 来添加变量观察信号了,最顶层的 top 模块接口也全改了,但底层接口就没有再改,还是包含了原来所有要观察的信号,因此 mips 模块中调用 datapath 接口时会有很多没有连接的线。这并不影响最终结果。

2.3 hilo 寄存器模块通路设计

这是一个抽象的、概念上的设计模块,具体实现分散在数据通路中。

在添加数据移动指令时,需要访问 hilo 寄存器,类似于 32 个基本寄存器,hilo 寄存器由 hilo_reg.v 文件提供。这里选择直接将 hilo 模块的相关控制逻辑添加到数据通路中。设计思路是把 hilo 寄存器看作是 32 个基本寄存器的扩展。与 32 个基本寄存器同步,在译码阶段读 hilo 寄存器,在写回阶段写 hilo 寄存器。设置 hl 信号来区分要访问的是 hi 寄存器还是 lo 寄存器,设置 hilo_readW 信号来表示是否需要读 hilo 寄存器。一些写回阶段用到的信号在译码阶段就传入了数据通路,因此需要在流水级间为对应信号添加寄存器,来把这些信号传递到写回阶段。涉及的信号见表 1,表中只列出了部分较重要的信号。而且这个模块实际实现上是分散在整个通路中的,没有单独的.v 文件,所以也没有输入、输出信号的概念。

信号名	位宽	功能描述
rsW	32-bit	rs 寄存器中的值,执行 mt 指令时用于移入 hilo 寄存器。
hlW	1-bit	在写回阶段 hlW=1 表示访问 hi,hlW=0 表示访问 lo。
hID	1-bit	在译码阶段 hID=1 表示访问 hi,hID=0 表示访问 lo。
hi	32-bit	经 hlW 信号选择过的要写入 hi 的值。
lo	32-bit	经 hlW 信号选择过的要写入 lo 的值。
hilo_writeW	1-bit	是否需要写 hilo 寄存器。
hilo_readW	1-bit	是否需要读 hilo 寄存器。
hilo_iW	64-bit	经 hilo_writeW 选择后的最终写入 hilo 寄存器的值,高 32 位写入 hi,低 32 位写入 lo。
hilo_oW	64-bit	从 hilo 寄存器中直接读出的值,高 32 位为 hi,低 32 位为 lo。
hiloresultD	32-bit	经 hID 信号选择过的 hilo 寄存器值。
hiloresultD	32-bit	hiloresultD 随流水线传递到写回阶段。
forwardHD	1-bit	当前 D 阶段的指令是否与前三条指令存在关于 hilo 的写后读数据相关。
hlwRef	32-bit	经过选择的前推的 hilo 中的数据。(名字由来:hilo writeback Result forwardhlo)

表 1: hilo 寄存器相关信号

设计该模块时用到的部分数据通路草图见图2。因为这只是设计时用的草图,所以会和最终代码中的数据通路有所出入。图中只简单画出了 hilo 部分相关逻辑控制数据通路图,为简洁起见,图中用 we 代表 hilo_writeD,用 hio 表示 hilo_oW[63:32],用 loo 表示 hilo_oW[31:0]。部分信号没有在图中标出。

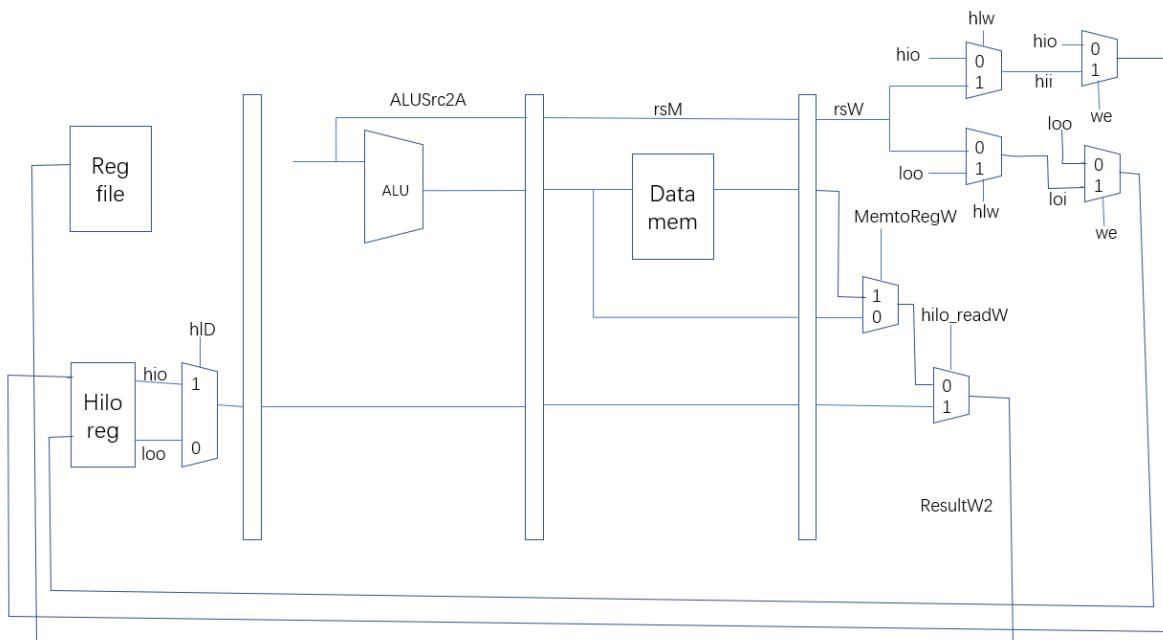


图 2: hilo 寄存器部分数据通路草图

需要注意的是, hilo 寄存器也存在数据冒险, 如在 D 阶段是读取 hilo 寄存器的时候, 若前三条指令存在写 hilo 寄存器的操作, 此时读取到的 hilo 寄存器的值还并没有在前面的写指令的写回阶段被正确写入, 故需要增加旁路将写回阶段的 hilo 值正确的送达读 hilo 的位置。添加了旁路和 hazard 控制的 hilo_hazard 模块如图

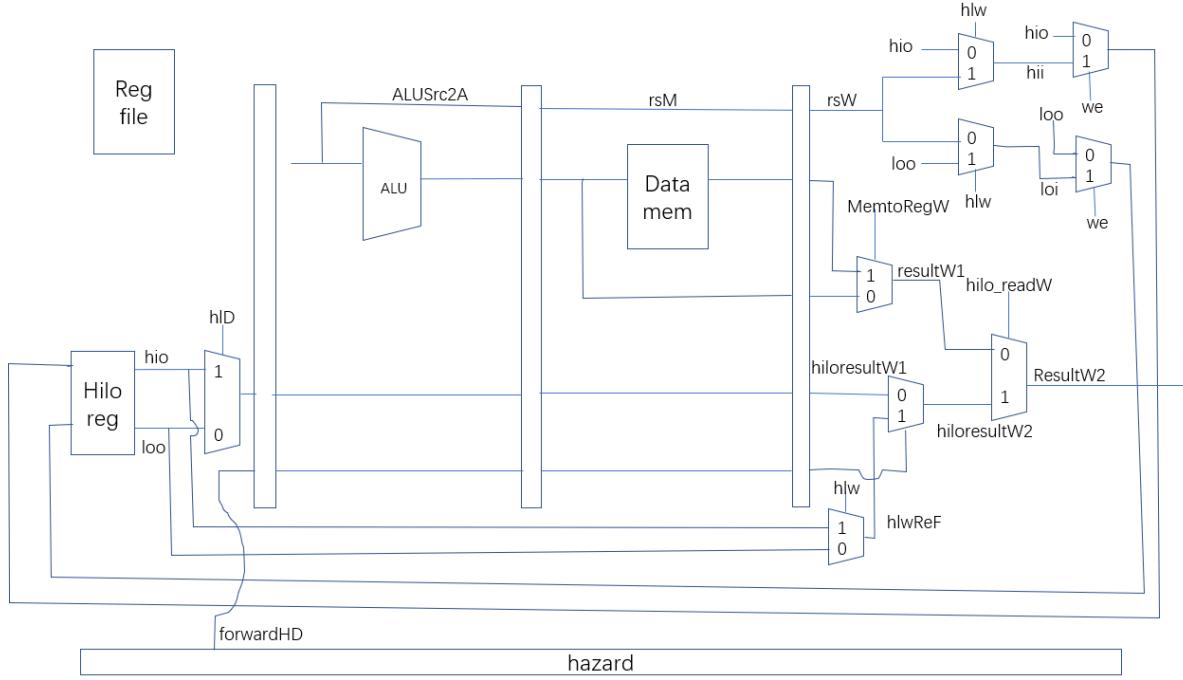


图 3: hilo_hazard 部分数据通路草图

其中, forwardHD 信号需要获取当前 decode 阶段的指令是否与前三条指令存在写后读的数据冒险, 即 $\text{forwardHD} = \text{hilo_writeE} \mid \text{hilo_writeM} \mid \text{hilo_writeM} \mid \text{multE} \mid \text{multM} \mid \text{multW} \mid \text{div_signalE} \mid \text{div_signalM} \mid \text{div_signalW}$ 。

将 hilo 寄存器写回阶段的值通过旁路和选择后作为前推数据 hlwReF, 由 forwardHW 信号控制, 若产生冒险, 则采用前推的数据, 若无冒险, 则采用正常的经过流水线两个阶段的 hiloresultW1 信号。最终得到从 hilo 寄存器中读出的数据 hiloresultW2。

2.4 div 功能实现设计

vivado 具有自带的除法器, 但是引入其自带的除法器会大大增加电路复杂度, 且由于其内部实现细节和参数传递的不一致性, 会影响实验结果。故本次实验采用了下发资料中提供的除法器模块来实现除法功能。需要注意的是除法器计算的结果需要保存到 hilo 寄存器中, 而且除法本身必然带来流水线的停顿。所以在除法的数据通路中需要涉及状态机的定义和流水线的停顿。除法器的状态机直接采用 ppt 上提供的。除法器接口的连线参考了吕学长视频中的做法。

除法器的接口说明 ppt 上已经提供了, 这里给出本设计中数据通路中除法功能的部分相关信号描述(表 2)。

因为只是通路中和除法功能相关的信号,不是单独的模块,所以没有输入输出的概念。

信号名	位宽	功能描述
signed_divE	1-bit	是否为有符号除法。
divSrcAE	32-bit	E 阶段除法器源操作数 A。
divSrcBE	32-bit	E 阶段除法器源操作数 B。
start_divE	1-bit	E 阶段除法开始信号。
annul_iE	1-bit	E 阶段强制结束除法信号。
div_readyE	1-bit	E 阶段除法完成信号。
div_signalE	1-bit	E 阶段是否执行除法。

表 2: 除法功能部分相关信号

除法器的连线本身并不复杂,复杂的是由除法引起的停顿及冒险的处理。当 E 阶段执行除法时,E 阶段本身需停顿,因此会引起 D 阶段停顿,F 阶段停顿。此外除法要写 hilo 寄存器,会涉及到 hilo 寄存器相关的冒险,设计中根据 div_signal 进行前推。div_signal 则根据 E 阶段的 ALUControl 来生成。另一处重要的地方是 annul_i 这个强制结束信号,在除法器该端口处接入 flush_except 信号,在发生异常,要清空流水线时,强制停止除法运算。此外,送入除法器的源操作数应该在除法执行的流水线暂停期始终保持不变,因为除法器在整个执行阶段都要用到它,因此就不能再是简单的一条线连进来,而是加个寄存器 (divSrcAE 和 divSrcBE) 来保留除法执行时的值,寄存器的更新(图4中 div src reg)根据除法暂停信号来决定。而传入 divSrcAE 和 divSrcBE 寄存器的源操作数和传入 ALU 的源操作数相同,都是经过旁路过的。

初步设计的除法器相关信号的数据通路如下图所示。

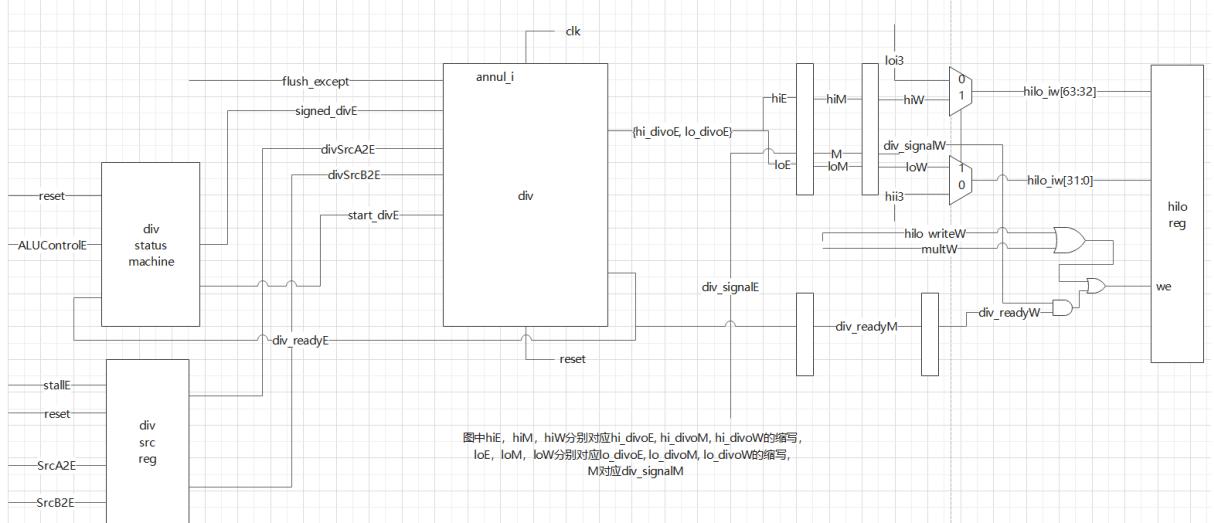


图 4: 除法器相关信号的通路草图

2.5 分支跳转指令实现设计

把分支跳转指令分为分支指令(B类)和跳转指令(J类)两类,J类指令不采用PPT中的控制信号,完全采用自己的设计。本设计中,把J类指令能执行的功能进行纵向划分,分别为跳转功能,写回延迟槽后的指令的地址功能,读寄存器功能(是否读寄存器),并分别用信号jump,jal,jr来标识(见表3),对于某条J类指令,

上述信号为 1 时表示该指令具备对应功能,否则不具备。通过这种方式可以区分出每条 J 类指令。其中 jalr 指令比较特殊,需根据指令的 rd 字段判断具体要将地址保存到哪个寄存器中。B 类指令较为简单,也是根据上述思路来设计的控制信号。具体信号设计见图5。

信号名	位宽	功能描述
branch	1-bit	是否有分支功能。
bal	1-bit	是否是分支指令且写回地址。
jump	1-bit	是否有跳转功能。
jal	1-bit	是否 bal 是跳转指令且写回地址。
jr	1-bit	是否是读寄存器跳转。

表 3: 分支跳转部分相关信号

	branch	jump	jal	ir	bal
J	0	1	0	0	0
Jal	0	1	1	0	0
Jr	0	1	0	1	0
Jalr	0	1	1	1	0

图 5: J 类指令的控制信号

下图是包括 jr,jal,jalr 指令相关控制信号的数据通路草图(不包含冒险)。

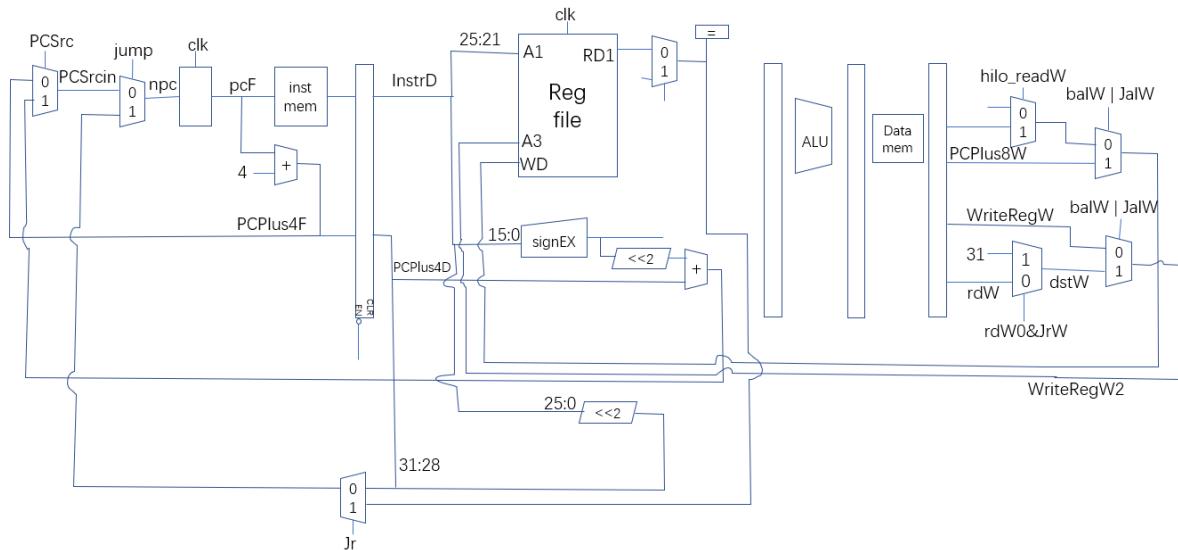


图 6: J 类指令数据通路草图

分支指令的执行需要读取 1-2 个寄存器,若之前存在对应寄存器的写入,就会形成写后读的冒险。mfhiilo 指令在译码阶段读出 hilo 中的值作为要写的结果,因此由它写寄存器产生的 RAW 只需要前推就可以解决。除 lw 指令外,其它所有写寄存器的指令在 EX 阶段产生要写的结果,lw 指令在 MEM 阶段获得要写的值,因此当前 D 流水级为分支指令需要读寄存器且 E 流水级由这些指令产生 RAW 时,必须暂停 D 流水级一周期,而且 M 流水级是 lw 指令且产生冒险时,也必须暂停流水级一周期。暂停过后就可以前推冒险的

数据了。

分支指令在 D 阶段读寄存器的前推逻辑见图7。由 al 系列写回通用寄存器可能引起的关于通用寄存器的冒险可以在 E 流水级就进行前推而不必暂停一周期,因为 al 系列指令要写回的值在当前 al 指令流经 D 阶段时就已经产生。

注意,因设计时就已开始报告编写,后续调试过程中又对代码进行了修改,所以报告截图中的代码可能会和最终通过测试版本的代码有所出入,这种情况下,请以最终提交的代码文件为准。

```
29      always @(*) begin
30          forwardaD = 2'b00;
31          forwardbD = 2'b00;
32          if(rsD!=0)begin
33              if(rsD == writeregE & regwriteE & (balE|jalE)) begin
34                  forwardaD = 2'b01;
35              end else if(rsD == writeregM & regwriteM) begin
36                  forwardaD = 2'b10;
37              end else if(rsD == writeregW & regwriteW) begin
38                  forwardaD = 2'b11;
39              end
40          end
41          if(rtD!=0)begin
42              if(rtD == writeregE & regwriteE & (balE|jalE)) begin
43                  forwardbD = 2'b01;
44              end else if(rtD == writeregM & regwriteM) begin
45                  forwardbD = 2'b10;
46              end else if(rtD == writeregW & regwriteW) begin
47                  forwardbD = 2'b11;
48              end

```

图 7: D 阶段读寄存器前推逻辑的代码

添加了前推逻辑的部分数据通路草图见图8。这里也只是设计时用到的草图,可能和最终代码中的通路有所出入。

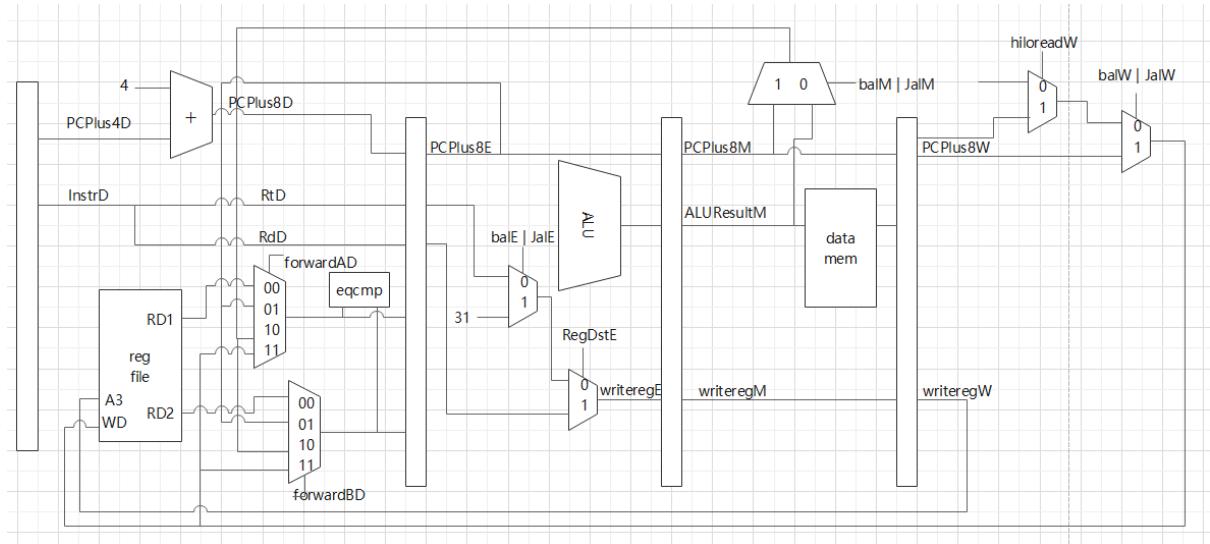


图 8: 带有前推逻辑的部分数据通路草图

2.6 cp0 模块调用以及异常处理

程序的异常与中断需要用到 cp0 寄存器来存储异常信息和跳转处理地址。cp0 寄存器直接调用资料中提供的选择把 eret 当作异常来处理。按照 ppt 以及学长讲解的异常处理步骤, 将异常处理分为三部分: 1. 识别异常, 传递异常信号。2. 访存阶段跳转至异常处理程序。3. 异常处理结束后返回产生异常的地方。

在 M 阶段处理异常是因为 M 阶段是可能产生异常的最后一个阶段, 在此阶段处理异常有利于精确异常处理。采用吕学长的方法, 用一个八位的信号记录八种(包含了 eret)异常, 该信号沿流水线流动的过程中逐级添加各级产生的异常信号。事实上, 这个八位的信号最低两位始终没有用到, 因此实际上只需要六位就足够了。但出于对吕学长原设计的尊重, 这里不再妄加修改。表4中是 cp0 和异常处理相关的部分信号说明。

信号名	位宽	功能描述
flush_except	1-bit	异常清空信号。
eretD	1-bit	D 阶段是否为 eret 指令。
exceptX	8-bit	异常信号记录, 用于在流水级中收集异常信号传递到 X 阶段处理。
syscallID	1-bit	D 阶段是否为 syscall 指令。
breakD	1-bit	D 阶段是否为 break 指令。
invalidD	1-bit	D 阶段是否为无效指令。
overflowE	1-bit	E 阶段是否产生溢出。
overflowE	1-bit	E 阶段是否产生溢出。
adelM	1-bit	M 阶段读数据的地址没对齐。
adesM	1-bit	M 阶段写数据的地址没对齐。
excepttypeM	32-bit	M 阶段 Mips 标准格式的异常类型。
cp0weX	1-bit	各流水级中的 cp0 使能信号。
cp0dataiX	32-bit	X 流水级中经旁路选择过的从 cp0 读出的数据。

表 4: cp0 和异常处理相关的部分信号

关于 eret 指令延迟槽的处理见图9, 修改 flushD 的逻辑, 在译码阶段的指令是 eret 时刷掉延迟槽, stallID 是

考虑到 eret 读 cp0 可能有数据冒险而导致停顿。

```
83 assign #1 flushD = flush_except|(eretD&~stallD); //flush delay slot after eret/
```

图 9: eret 延迟槽的处理

取指阶段可能产生 pc 没对齐的取指令地址错例外 adel(图10)。

```
171 assign exceptF=(pcF[1:0]==2'b00)?8'b0000_0000:8'b1000_0000;// adel
```

图 10: 取指 pc 没对齐的 adel

译码阶段可能产生系统调用例外,断点例外,保留指令例外和 eret 例外(把 eret 当例外处理)(图11)。

```
181 reE(clk,reset,~stallE,flushE,{exceptD[7],syscallD,breakD,eretD,invalidD,exceptD[2:0]},exceptM);
```

图 11: 译码阶段产生的例外

执行阶段可能产生算术溢出例外。加减法的进位就是运算结果在不溢出时的符号,如果运算结果的符号与进位不一致,则发生溢出(图12)。

```
183 flopr #(8) reM(clk,reset,{exceptE[7:3],overflowE,exceptE[1:0]},exceptM);
88 assign cbitr = ((F=='EXE_ADD_OP)|(F=='EXE_ADDI_OP))? {A[31],A}+{B[31],B}:
89             (F=='EXE_SUB_OP)? {A[31],A}-{B[31],B}:
90             32'b0;
91 assign overflowE = ((F=='EXE_ADD_OP)|(F=='EXE_ADDI_OP)|(F=='EXE_SUB_OP))?cbitr[N]^Y[N-1]:
92             1'b0;
```

图 12: 溢出例外

访问数据存储器的地址错例外由数据存储器外边包的那层处理半字和字节读写的逻辑产生(图13)。

```
89 case(ALUControlM)
90     `EXE_LW_OP:begin
91         sel<=4'b0000;
92         writedata2<=32'b0;
93         ReadData2<=ReadData;
94         // adel exception, copied from video
95         if(dataaddr[1:0]!=2'b00)begin
96             adelM<=1'b1;
97             bad_addrM <= dataaddr;
98         end
99     end
```

图 13: 访存阶段的地址错例外

参考 ppt 和吕学长视频中的 exception 模块,如图14,将例外信号转换为 MIPS 标准类型以便于写入 CP0,并

根据 CP0 寄存器的 status 和 cause 获得中断 INT 信号,所有转换过的例外信号由 excepttypeM 端口输出,该信号连入 CP0 寄存器的 excepttype_i 端口,至此全部八种(包含 eret)异常被成功捕获并送入 CP0 寄存器。

```
185 exception exp(reset,exceptM,ad1M,adesM,status_o,cause_o,excepttypeM);
```

图 14: 调用 exception 模块

添加选择器,从读指令地址和读写数据地址中选择出导致异常的坏地址(图15)。

```
186 mux2 #(32) bad_addr(i(pcM,bad_addrM,exceptM[7],bad_addrM));
```

图 15: 选择坏地址

跳转地址模块,除了 eret 外都是跳 32'hBFC00380,eret 跳转至从 CP0 中读出的 epc(图 16)。

```
197 always@(*)begin
198     if(excepttypeM!=32'b0)begin// not Int
199         case(excepttypeM)
200             32'h00000001:begin
201                 newpcM <=32'hBFC00380;
202             end
203             32'h00000004:begin// ad1
204                 newpcM <=32'hBFC00380;
205             end
206
207             32'h00000008:begin// syscall
208                 newpcM <=32'hBFC00380;
209             end
210             32'h00000009:begin// break
211                 newpcM <=32'hBFC00380;
212             end
213             32'h0000000a:begin// undefined inst
214                 newpcM <=32'hBFC00380;
215             end
216             32'h0000000c:begin// overflow
```

图 16: 异常跳转地址

在 top 接口添加硬件中断端口,将该信号接到 cp0 硬件中断输入端口上(图 17)。

```

24  module top(input clk,rst,output[31:0]instr,pc,output memtoreg,memwrite,pcsrc,alusrc,regdst,
25  zero, output[7:0]alucontrol,// new 2 to 7
26  output[31:0]dataaddr,WriteData,output reg[31:0]ReadData2, output [31:0]SrcAD,SrcBD,npc,Resu
27  output StallF,branchD,stallD,MemtoRegE,flushE,RegWriteM,MemtoRegM,RegWriteW,output[4:0]rsD,
28  output[1:0] forwardAD,forwardBD,forwardAE,forwardBE,output[31:0]SrcA2E,SrcB2E,InstrD,output
29  output[7:0]ALUControlE,// new 2 to 7
30  output[31:0]SignImmE,output[4:0]WriteRegM,WriteRegW,output MemtoRegW,
31  output[31:0]ReadDataW,ALUResultW,SrcA2D,SrcB2D,PCPlus4D,output[4:0]rdD,
32  output[31:0] ALUResultE,WriteDataE,SrcBE,wloi,whio,wloo,rsW,reg2,
33  output hilo_writeW, hilo_writeD,output[5:0]entermfhi,output[31:0]reg4,
34  output[63:0]hilo_tempE,output[31:0]reg1,reg3,hi_divoW,lo_divoW,output div_readyE,
35  output signed_divE,start_divE,stall_divE,h1W,multW,div_signalW,output[31:0]reg31,pcPlus8W,
36  output balW,jalW,balD,balE,balM,output[31:0]pcE,pcM,pcW,output branchJr stallD,bjfromE,
37  bjfromM,jalE,flushD,output[4:0]rdE,output[31:0]ALUResultM1,output memenD,
38  output reg[31:0]writedata2,output reg[3:0]sel,output [31:0]ReadData,output [31:0]addr,
39  output memenE,memenM,output[31:0]PCJump2

```

图 17: 添加硬件中断信号

添加选择器,在 M 阶段发生异常时选择异常产生的指令地址执行,eret 借此跳回(图18)。

```

158 mux2 #(32)pcjump2mux(SrcA2D,PCJump1,jrD,PCJump2);
159 mux2 Jumpm(PCJump2,PCSrcin,jump,pcJ);
160 mux2 excpm(newpcM,pcJ,excepttypeM!=32'b0,npC);

```

图 18: 异常地址跳转

在发生异常清空流水线时,原本 al 系列指令写寄存器相关的信号不能简单地添加 flush 信号清空,因为这样会导致不发生异常时这些信号也被清空,从而导致无法写回寄存器。

故对于 al 系列指令直接不清空后续阶段的信号,而不是原来的清空不需要的信号(图19)。

```

85 assign #1 flushE = (lwstallD | branchD & ~balD | jumpD & ~jalD) & ~stallE|flush_except;//
```

图 19: 针对 al 系列指令修改 flushE 信号

于是这些寄存器都可以加上 flushE 信号进行刷新了。

对于 mfc0 和 mtc0,和 mfthilo 不同,这次采用访存阶段写,执行阶段读的方式,这样做的话,关于 cp0 的冒险,读 mfc0 只需要前推上一条紧邻且写同样 cp0 寄存器的 mtc0 指令要写的值。但是 mtc0 要处理访存阶段读通用寄存器的前推,这种情况只发生在紧邻的上一条指令要写的通用寄存器恰好是当前 mtc0 指令读的通用寄存器。

hilo 寄存器设计为译码阶段读,写回阶段写,这样设计的冒险处理逻辑较复杂。设计 CP0 寄存器的读写时充分吸取前面 hilo 寄存器读写的教训,为了减少处理冒险的复读写设计杂度,选择在执行阶段读 CP0,访存阶段写 CP0。在译码阶段对 mftc0 指令进行译码,alu_dec 中获得 aluop,alu 中直接将 rt 传给结果,将 cp0

写使能信号 cp0we 逐级传递(图20)。

```
305  flopenc #(1) r21E(clk,reset,~stallE,flushE,cp0weD,cp0weE);  
330  flopenc #(1) r19M(clk,reset,~stallM,flushM,cp0weE,cp0weM);
```

图 20: 传递 cp0 写使能

向 cp0 中写入的值是经旁路过的 rtM 寄存器中的值。因为在访存阶段才用到 rtM 的值,如果紧邻的上一条指令是 lw 并产生数据依赖,那么可以直接从写回阶段前推,而不需要暂停。

修改由前面的载入指令引起的译码阶段的停顿,译码阶段是写 mtc0 指令时,可以不用暂停(图21),等到 M 阶段由 W 阶段前推(图22)。

```
85  assign #1 lwstallD = memtoregE & (rtE == rsD | rtE == rtD ) & ~cp0weD;// no need to add
```

图 21: 修改载入指令引起的译码阶段停顿

添加旁路逻辑,选择旁路过来的值写入 cp0(图22),这样 mtc0 就添加完了。

```
92  assign forwardrdM = cp0weM & memtoregW & (rtM==regwriteW);  
187 mux2 #(32) cp0dataiMmux(ResultW3,ALUResultM2,forwardrdM,cp0dataiM);  
188 cp0_reg cp0(.clk(clk),.rst(reset),.we_i(cp0weM),.waddr_i(rdm),.raddr_i(rdE),  
189 .data_i(cp0dataiM),
```

图 22: 前推载入的值给 mtc0

然后是 mfc0,译码已经做过了,不需要 alu 处理,在 E 阶段读 cp0,送进 cp0 的读地址是 rdE。接下来写通用寄存器在写回阶段,要把读出来的 cp0 中的值写进通用寄存器。只需要将读出来的值传到写回阶段,添加一个标识当前指令是 mfc0 的信号,并传递到 W 阶段,在 W 阶段根据该信号选择要写回通用寄存器的值即可。

译码时用 is_mfc0D 信号标识 mfc0(图23),将此信号逐级传递。

```
307  flopenc #(1) r22E(clk,reset,~stallE,flushE,is_mfc0D,is_mfc0E);  
334  flopenc #(1) r21M(clk,reset,~stallM,flushM,is_mfc0E,is_mfc0M);  
358  flopenc #(1) r18W(clk,reset,flushW,is_mfc0M,is_mfc0W);
```

图 23: 标识 mfc0

将从 cp0 中读出的数据逐级传递(图24)。

```
337  flopenc #(32) r22M(clk,reset,~stallM,flushM,cp0dataE,cp0dataM);  
362  flopenc #(1) r19W(clk,reset,flushW,cp0dataM,cp0dataW);
```

图 24: 传递 cp0 中读出的数据

根据 is_mfc0W 选择最终要写入通用寄存器的结果(图 25)。

```

94 mux2 #(32) resultW3(pcPlus8W,ResultW2,balW|jalW,ResultW3);
95 mux2 #(32) resultW4(cp0dataW,ResultW3,is_mfc0W,ResultW4);
96 regfile reg32(.clk(clk),.rst(reset),.we3(RegWriteW),.ra1(InstrD[25:21]),.ra2(InstrD[20:16]),
97 .wa3(WriteRegW),.wd3(ResultW4),.rd1(SrcAD),

```

图 25: 根据 is_mfc0W 选择结果

最后,当相邻两条指令是 mtc0,mfc0 且存在数据依赖时,会产生数据冒险,mfc0 在 E 阶段要读 cp0 时,mtc0 正在 M 阶段写。

添加 cp0 前推逻辑,添加选择器选择出旁路过的 cp0dataE(图26)。

```

92 assign forwardcp0E = is_mfc0E & cp0weM & (rdE==rdM);
198 mux2 #(32) cp0dataEmux(cp0dataiM,data_o,forwardcp0E,cp0dataE);

```

图 26: cp0 前推

至此内陷指令和特权指令添加完毕,其它三条指令只是打了标记传递给 cp0 并获得跳转地址。下面两张图分别是 cp0 和 exception 模块相关信号的数据通路草图。

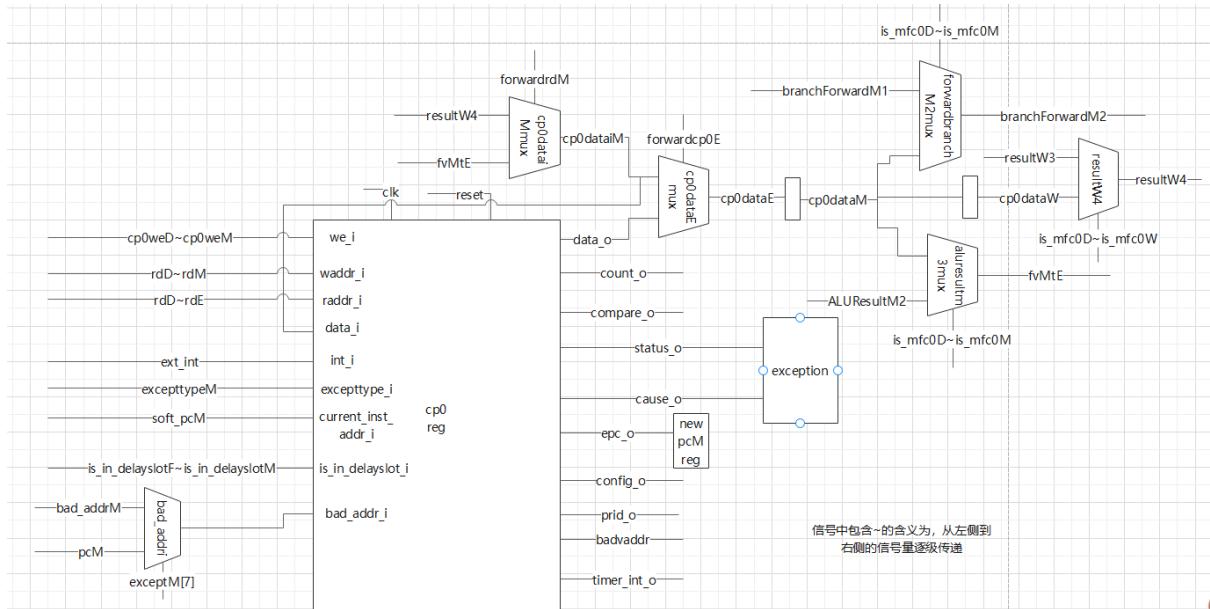


图 27: cp0 相关信号通路草图

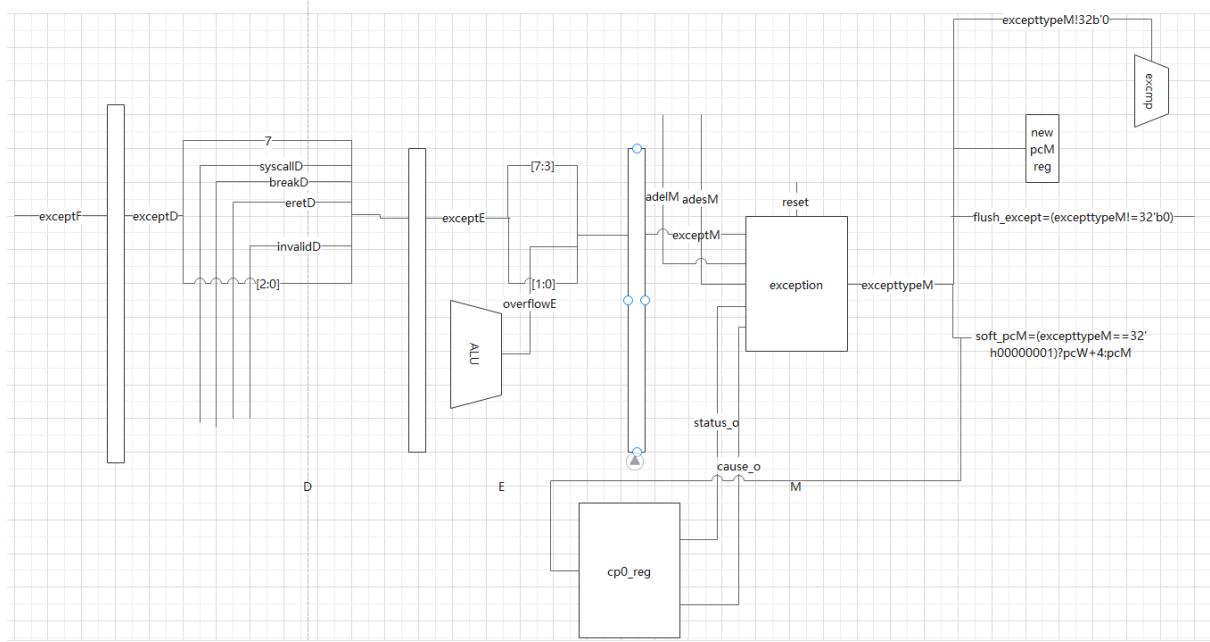


图 28: exception 相关信号通路草图

2.7 冒险模块设计

实际设计时, hazard 模块并不是单独拿出来预先设计的,而是在添加每类指令时,考虑添加的指令可能产生的冒险,设计相应的冒险逻辑并对 hazard 模块进行修改。最终当然还是通过进行功能测试,在不断 debug 的过程中逐步修改设计,才得到最终完善的版本。在前面有介绍实现某个功能时相应的冒险设计,前面已经写过的部分这里就不再重复。因为冒险的处理可能涉及很多不同功能,这些不同功能的冒险处理可能是交织在一起的,这里主要是这些不方便放在前面的冒险逻辑。

表5中是 hazard 模块端口信号说明。

信号名	方向	位宽	功能描述
stallX	output	1-bit	X 级暂停信号。
rsX	input	5-bit	X 级 rs。
rtX	input	5-bit	X 级 rt。
forwardaX	output	2-bit	X 级源操作数 A 的前推信号。
forwardbX	output	2-bit	X 级源操作数 B 的前推信号。
writeregX	input	5-bit	X 级要写的目的寄存器。
regwriteX	input	1-bit	X 级是否写通用寄存器信号。
memtoregX	input	1-bit	X 级是否从数据存储器中读数据写回通用寄存器信号。
hilo_writeX	input	1-bit	X 级读 hilo 寄存器信号。
hilo_readX	input	1-bit	X 级写 hilo 寄存器信号。
multX	input	1-bit	X 级是否为乘法指令信号。
forwardHD	output	1-bit	D 级读 hilo 寄存器的前推信号。
stall_divE	input	1-bit	E 级除法暂停信号。
div_signalX	input	1-bit	X 级是否是除法指令信号。
flushX	output	1-bit	X 级 flush 信号。
jumpD	input	1-bit	D 级指令是否有跳转功能。
balX	input	1-bit	X 级 bal。
jalX	input	1-bit	X 级 jal。
cp0weX	input	1-bit	X 级 cp0we 使能信号。
cp0weX	input	1-bit	X 级 cp0we 使能信号。
jrD	input	1-bit	D 级是否是寄存器跳转。
pcsrcD	input	1-bit	D 级 pc 的分支选择信号。
flush_except	input	1-bit	异常时的 flush 信号。
eretD	input	1-bit	D 级是否是 eret 指令。
forwardrdM	output	1-bit	M 级写 cp0 时的前推信号。
is_mfc0E	input	1-bit	E 阶段是否是 mfc0 指令。
forwardcp0E	output	1-bit	E 级读 cp0 时的前推信号。
flushpcE	output	1-bit	E 级专门用于冲刷 pc 的信号。
istall	input	1-bit	读指令停顿信号。
dstall	input	1-bit	读写数据停顿信号。

表 5: hazard 模块端口信号

在向处于 D 流水级的分支指令前推数据时,如果要前推的数据来自 mfhilo 指令,可能会面临 mfhilo 从 hilo 寄存器中读出的值本身也需要前推的问题,这样就会发生前推中套另一个前推,可以在流水线长度允许下一直套下去。

这种前推套前推的设计复杂且麻烦,如果 mthilo 指令在执行阶段写 hilo 寄存器,那么考虑以下情景,依次执行三条指令:mthi;mfhi \$1;beq \$1,\$2,offset,每相邻两条指令间都存在 RAW,beq 在译码阶段读 \$1,此时 mfhi 在 EX 阶段,还未写入 \$1,因此需从 EX 向译码阶段前推,而此时 mthi 已经执行到 MEM,由假设,mthi 在执行到 EX 阶段时就已经写入 hi 寄存器,也就是 beq 执行到取指阶段时,因此,mfhi 在 EX 向译码阶段的 beq 前推 hi 寄存器的值时,可以直接取当前从 hi 寄存器读出来的值,因为这个值在上一周期就已被 mthi 修改过,不再需要二次前推。

于是准备更改设计,把 mthilo 的写 hilo 寄存器由写回阶段前移到执行阶段。但如果把写 hilo 放到执行阶段的话,那么写 hilo 指令要读的通用寄存器就面临 RAW,而且因为在执行阶段就要用,所以还可能需要停顿。

这么看来,把写 hilo 提前也并没有带来太多好处,问题的根源应该在于数据移动指令直接读寄存器和写寄存器,两头都有冒险的可能,要对它进行旁路,只能直接旁路寄存器值,而不像其它 ALU 指令那样有个 ALUResult 可以用来旁路,旁路寄存器值就会引起要旁路的值本身也要旁路,从而引起套娃。这大概是个不能两全的问题,应该是要引入停顿才能解决。

于是还是不改变原来的设计,还是统统放在写回阶段和通用寄存器保持同步,如果要旁路的值由 mfhilo 指令产生,为了避免套娃,选择不进行旁路而是直接停顿到 mfhilo 指令执行完毕。

修改 branchstallD,确保前两条 mfhilo 指令引起 raw 时暂停,而往前数第三条指令如果是 mfhilo 且引起了 raw,则可以通过前推解决,这时前推的值一定是正确的,不会触发套娃。修改前的分支指令在 D 阶段由于 RAW 引起暂停的代码见图29,修改后的代码见图30。

```
71 ~ assign #1 branchstallD = branchD &
72           | (regwriteE &
73           |   (writeregE == rsD | writeregE == rtD) |
74 ~           |   memtoregM &
75           |   (writeregM == rsD | writeregM == rtD));
```

图 29: 修改前的 branchstallD

```
72 assign #1 branchstallD = branchD &
73           | (regwriteE & (writeregE == rsD | writeregE == rtD) |
74           |   (memtoregM | hilo_readM) & (writeregM == rsD | writeregM == rtD));
```

图 30: 修改后的 branchstallD

sram 接口的添加,向 sram 接口的转换,以及连接 axi 转接桥部分主要是修改接口进行连线,几乎不存在需要设计的地方,不再单独开设模块进行介绍。这部分的主要难点在于进行调试以通过测试,调试过程中做出的改动请参见第 3 部分设计过程中的错误记录。

连接 axi 转接桥后,从存储器中读写数据以及读指令时会有较大延迟,在完成读写请求前,流水线应暂停而不是继续运行,因此需引入 dstall 和 istall 信号来标识读写数据和读指令延迟引起的流水线暂停。

对于读指令延迟,这里采取的方案是只暂停前两个流水级,后面流水级继续运行。因此引入 istall 更新了 F 和 D 的暂停逻辑以及 E 的冲刷逻辑。

对于读写数据延迟,暂停整个流水线,因此所有流水级的暂停信号都引入 dstall。

3 设计过程

3.1 设计流水账

12月27日,15:00-23:00

xxx 配置了 VSCode 编辑器,使得两人可以通过 VSCode 同时在线编写代码,同时具有代码高亮和语法检测功能。添加了逻辑指令组,修改了 aludecoder,controller 和 alu 的部分接口。

ytz 将提供的 test_bench.v 文件添加到计组实验四项目中,调整了接口,使其可以进行仿真测试。修改了 maindecoder 的接口和各级模块(从 datapath 到 testbench)接口,添加了移位指令组,编写了部分实验报告。

通过了测试点 1。

12 月 28 日,8:30-23:00

xxx 添加了部分算术运算指令,绘制了 hilo 寄存器的数据通路图。

ytz 添加了 hilo 数据移动指令。

通过了测试点 2,3 逻辑指令组和移位指令组添加验证完成。

12 月 29 日,10:00-23:00

通过了测试点 4,hilo 指令组添加验证完成。

12 月 30 日,15:00-20:00

添加了算术指令的加减、大小判断、乘法功能模块,通过了测试点 5。添加了除法指令,但并未正确通过测试点 6

12 月 31 日 11:00-23:00

完善了除法器,通过了测试点 6。更换了新的测试文件,通过进一步调试后通过了 ArithmeticTest, DataMoveInstTest, LogicInstTest, ShiftInstTest 四个测试点。

1 月 1 日 10:00-23:00

添加了分支跳转指令,实现了基本的分支跳转指令执行。

1 月 2 日 10:00-23:00

完善了分支跳转指令,增加了分支跳转指令的数据冒险通路。

1 月 3 日 9:00-23:00

完善了分支跳转指令的冒险逻辑,通过测试点 7,8,添加了访存指令。

1 月 4 日 9:00-23:00

完善了访存指令和冒险,通过了测试点 9,10,添加了 cp0 和 exception 指令。

1 月 5 日 8:00-23:00

完善了异常处理模块,通过了测试点 11,12,尝试连接 sram,进行 soc 功能测试。

1月6日 9:00-1月7日 2:00

通过不断的调试 debug,过了全部的 89 个功能测试点。尝试连接类 sram,进行 axi 功能测试。

1月7日 9:00-1月8日 4:00

通过不断的调试 debug,通过了全部了 89 个 axi 功能测试点,通过了 10 个 axi 性能测试,添加了 cache 模块,但是可惜没有通过全部的 axi 功能测试,卡在了第 77 个功能测试点。

3.2 错误记录

3.2.1 错误 1

- (1) 错误现象: 端口定义和模块接口标红,报错‘redeclaration of ansi port ’XX’ is not allowed’。
- (2) 分析定位过程: 检查报错位置,发现模块接口中出现的端口在后面被重新声明了。
- (3) 错误原因: Verilog 语法中不允许多次声明同一个端口,在模块接口中声明的端口不需要在模块中被再次声明。
- (4) 修正效果: 注释掉模块中相应端口的声明,问题解决。
- (5) 归纳总结(可选): Verilog 语法规范问题,通常是调试时修改端口导致的,不影响正常仿真,可根据报错提示立即解决(即使不解决也不会影响结果)。

3.2.2 错误 2

- (1) 错误现象: 仿真图中符号扩展后的立即数为 XXXX。
- (2) 分析定位过程: 找到立即数符号扩展模块,发现问题。
- (3) 错误原因: 符号扩展模块中,新添加的判断进行有符号扩展还是无符号扩展的 type 信号忘记连线。
- (4) 修正效果: 在 datapath 中将指令的 opcode 的中间两位连入符号扩展模块的 type 接口。问题解决。
- (5) 归纳总结(可选): 修改原有模块时忘记改接口。在对模块进行修改时应牢记检查接口,确保所有接口都对的上,有来源。

3.2.3 错误 3

- (1) 错误现象: 波形图中,ALU 的两个源操作数 SrcA2E,SrcB2E 均正确,但 ALU 的运算结果不正确。
- (2) 分析定位过程: 判断是 ALU 出了问题,定位到 ALU 模块。
- (3) 错误原因: 为添加移位指令,在 ALU 模块中添加了 sa 端口,但忘记在 datapath 中调用 ALU 时传递该信号。
- (4) 修正效果: 在 datapath 中调用 ALU 时将指令中 shift 对应的 5 位传递给 sa 端口。问题部分解决。
- (5) 归纳总结(可选): 修改原有模块时忘记改接口。在对模块进行修改时应牢记检查接口,确保所有接

口都对的上,有来源。

3.2.4 错误 4

- (1) 错误现象: 波形图中 ALU 的运算结果仍然不完全正确。
- (2) 分析定位过程: 经分析,问题应仍出在 ALU 模块,仔细检查后将问题定位在 ALUControl 信号的位数上。检查过各级(从 datapath 到 testbench)模块后又去 datapath 中寻找出现的每一处 ALUControl,最终发现用于在流水级间传递该信号的寄存器位数忘记修改。
- (3) 错误原因: ALUControl 信号由实验 4 中的 3 位改为 8 位,忘记在各级模块的接口中进行修改。忘记修改在流水级间传递该信号的寄存器位数。
- (4) 修正效果: 对各级模块接口中 ALUControl 信号的位宽进行修改。修改后仍有问题。再对 datapath 中在流水级间传递该信号的寄存器位数进行修改,修改后问题解决。
- (5) 归纳总结(可选): 修改信号位宽后应接着修改接口,确保各级接口同步。还应检查每一处用到该信号的地方,确保位数同步。

3.2.5 错误 5

- (1) 错误现象: 测试逻辑运算指令时,andi 指令的结果不对。
- (2) 分析定位过程: 首先检查源操作数,发现 SrcB2E 不对,再去检查产生该源操作数的选择信号 alusrc,发现不对,将问题定位到 alusrc 的产生上,即 maindecoder 模块中。
- (3) 错误原因: maindecoder 模块中,根据 opcode 进行的 case 选择分支中忘记添加 ANDI 指令的情况,导致执行 ANDI 指令时,采取的是默认分支。
- (4) 修正效果: 在 maindecoder 中的 case 块中添加 ANDI 对应的分支,问题解决。
- (5) 归纳总结(可选): 写代码时不够仔细,没有对每一条指令是否正确添加进行检查。应按指令一条一条检查以确保正确。

3.2.6 错误 6

- (1) 错误现象: 测试数据移动指令时,仿真图中,前三条 mthi 指令对 hilo 的修改是正确的,但后面 hilo 中 hi 的值一直在被修改,即使执行的指令是 mfhi。mfhi 指令从 hilo 寄存器中读出的值不正确。
- (2) 分析定位过程: 起初问题定位错误,误以为是第二条 mthi 指令就出错了,检查过 2 号寄存器之后才发现是眼花了,看的是写回阶段的信号,和取值阶段的地址错开了几个周期。这样才成功找到问题是出在 mfhi 指令上。先去检查 maindecoder 模块 case 块中 mfhi 分支,未找到问题。在波形图中发现 hi 在不希望被修改的时候被修改了,于是去检查 hilo 寄存器写使能信号 hilo_writeW,发现该信号自从变为 1 后就再也没有回到过 0,一直错误地保持高电平。于是把问题定位到 main_decoder 模块中 hilo_writeW 信号的产生。修改以确保 hilo_writeW 信号正确后,问题仍未解决。在 case 块中 mfhi 对应的分支中应该将 hilo_writeW 置零,但波形图显示仍为 1,怀疑该分支根本没有被执行,在该分支中专门添加了一个该分支独有的信号,用于测试该分支是否被执行,仿真图中,该信号始终为 XXX,该

分支果然没有被执行。盯着 case 块看了足够长时间后,想到 funct 变化并没有对分支选择造成影响,果然 always 块敏感列表中忘记添加 funct。

- (3) 错误原因: main_decoder 模块中最开始只添加了 ppt 中提示的 hilo_writeW 信号,后来设计 hilo 寄存器访问逻辑时发现需要一个信号对访问 hi 还是 lo 进行区分,还需要一个信号判断是否读 hilo。因此对应添加了 hlW 信号和 hilo_readW 信号,但忘记同步到 maindecoder 模块中了,具体来说,只在该模块的接口中添加了这两个信号,忘记在 case 块中每一个分支对这两个信号赋值。且忘记在 always 块敏感列表中添加 funct
- (4) 修正效果: 在 case 块每个分支中对新加的两个信号赋值,在 always 块敏感列表中添加 funct。问题解决。
- (5) 归纳总结(可选): 一方面要确保新加的信号都应被赋值。另一方面对 verilog 语法不够熟悉,always 块中所有用于条件判断的信号都应添加到敏感列表中。

3.2.7 错误 7

- (1) 错误现象: add 指令产生算术溢出时未能按照算术溢出异常处理经过处理输出正确的预期结果
- (2) 分析定位过程: 观察仿真产生的波形图,波形图运行到 add 指令时,输出的波形并没有按照预期的输出 $32'b0$
- (3) 错误原因: 算术溢出需要实现异常处理模块,指令添加到当前阶段还未引入异常处理模块
- (4) 修正效果: 由于目前还未引入异常处理模块,为了通过该测试点,暂时采用打表的方法在 alu 时就判断溢出并输出 $32'b0$ 。需要注意的是后续实现异常处理模块时要记得回过头来处理 add 等这些有溢出的异常。
- (5) 归纳总结(可选): 对于有符号的运算需要注意在 32 位的限制下可能产生溢出,对这种溢出异常 cpu 应该能检测并正确处理,以防止出现后续的问题。

3.2.8 错误 8

- (1) 错误现象: 在执行乘法指令时,乘法运算结果没有正确写入 hilo 寄存器中
- (2) 分析定位过程: 通过观察波形图可知,运算结果已经正确在写回阶段输入到 hilo 寄存器,但是后续 hilo 寄存器的值并没有根据输入的值进行更新。有输入的值但没有正确写入,说明是 hilo 寄存器的写使能没有生效。经过查看 hilo 寄存器的写使能信号赋值发现确实没有添加乘法时激活写使能的条件。



图 31: hilo 寄存器未正确写入乘法结果

- (3) 错误原因: 输入 hilo 寄存器的值需要有写使能才能正确写入 hilo 寄存器,而在实现乘法指令的时候并没有同步在 hilo 的写使能信号判断条件中加入乘法条件,导致乘法运算结果不能正确写入 hilo 寄存器中。
- (4) 修正效果: 给 hilo 寄存器写使能信号判断条件中增加乘法写回阶段的写使能之后,问题得到解决。

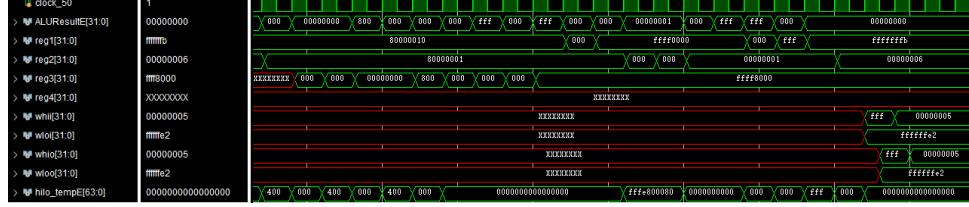


图 32: hilo 寄存器正确写入乘法结果

- (5) 归纳总结(可选): 添加的指令如果需要引用到额外的模块,需要注意:为了确保引用的额外模块正确执行功能,这个模块所需的各类信号条件都要能够从这条指令处得到反馈。不能出现像这个错误中出现的这样,该功能模块需要某个使能信号,但是指令未能正确控制该使能信号转变为正确的真,导致功能模块未能正确执行。

3.2.9 错误 9

- (1) 错误现象: 仿真运行失败,显示 result_o 端口没有正确连接。

```

    ▼ Simulation (2 errors)
      ▼ sim_1 (2 errors)
        ! [VRFC 10-3423] illegal output port connection to 'result_o' [datapath.v:146]
        ! [XSIM 43-3322] Static elaboration of top level Verilog design unit(s) in library work failed.
  
```

图 33: 控制台错误信息

- (2) 分析定位过程: 根据控制台输出的错误定位到 datapath 中的除法模块引用,仔细对比后发现 datapath 中提供的信号和 div.v 中定义的信号没有完全对齐,因信号位宽不同,引入的信号位宽大于定义的位宽导致报错。
- (3) 错误原因: datapath 中引用除法器时在引入 signed_div_i 后多引入了一位控制信号 ALUControlE 导致后面的信号都没有和 div.v 中定义的信号对齐,最终在 result_o 处报错。

```

146 div Div(clk,rst,signed_divE,ALUControlE,SrcA2E,SrcB2E,start_divE,1'b0,{hi_divoE,lo_divoE},di
module div(
    input wire                                     clk,
    input wire                                     rst,
    input wire                                     signed_div_i,
    input wire[31:0]                                opdata1_i,
    input wire[31:0]                                opdata2_i,
    input wire                                     start_i,
    input wire                                     annul_i,
    output reg[63:0]                               result_o,
    output reg                                     ready_o
);

```

图 34: div 模块调用信号未对齐

- (4) 修正效果: 将 datapath 中调用 div 处的 ALUControlE 删除后报错解决。

```

146 div Div(clk,rst,signed_divE,SrcA2E,SrcB2E,start_divE,1'b0,{hi_divoE,lo_divoE},div_readyE);

```

图 35: 修正端口使信号对齐

- (5) 归纳总结(可选): 在引用功能模块的时候,一定要注意端口的对齐,对于.0 引用的方法可以不按照原设计的功能模块的顺序写,但也要保证每个端口都正确的输入和输出信号,而对于直接引用的方法,更是要仔细比对端口是否完全一致。

3.2.10 错误 10

- (1) 错误现象: 仿真运行失败,显示 div 模块运算结束标志 div_readyE 不合法。

```

▼ Simulation (2 errors)
  ▼ sim_1 (2 errors)
    ! [VRFC 10-3236] concurrent assignment to a non-net 'div_readyE' is not permitted [datapath.v:146]
    ! [XSIM 43-3322] Static elaboration of top level Verilog design unit(s) in library work failed.

```

图 36: 输出端口不合法

- (2) 分析定位过程: 追踪至 datapath 中发现原本应该以 wire 接出的信号 div_readyE 错误的写成了 reg 类型。

- (3) 错误原因: 将 div_readyE 的类型错误的设置为 reg 型

```

70 reg signed_divE,start_divE,div_readyE,stall_divE;// new

```

图 37: div_readyE 类型错误

- (4) 修正效果: 将 div_readyE 信号类型设置为 wire 型后问题解决。

```

70 reg signed_divE,start_divE,stall_divE;// new
71 wire div_signalE,div_signalM,div_signalW,div_readyE;// new

```

图 38: div_readyE 类型修正

- (5) 归纳总结(可选): 对于信号的 reg 还是 wire 类型需要更谨慎的进行判断后定义,以防止后续添加时再次出现类似的问题。

3.2.11 错误 11

- (1) 错误现象: 除法结果没有按照预期的在 36 个周期后写入 hilo 寄存器中。

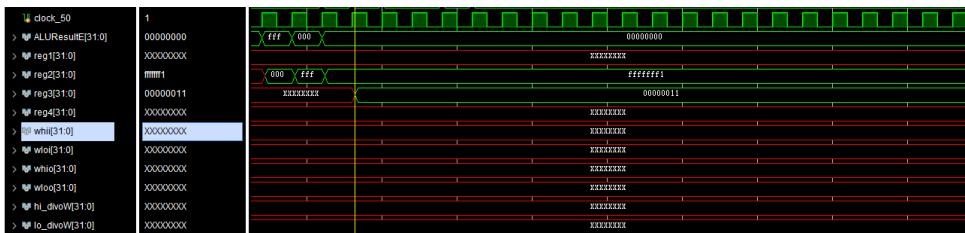


图 39: div 运算结果未正确写入 hilo 寄存器

在执行除法 36 个周期后 whii, wlio 应该变成除法结果,但是仿真图中一直为 xxxx。

- (2) 分析定位过程: 首先怀疑是除法与前面的运算指令存在数据相关没有处理这一问题导致。

ori \$2,\$2,0xffff1	# \$2 = -15
ori \$3,\$0,0x11	# \$3 = 17

div \$zero,\$2,\$3	# hi = 0xffffffff1
	# lo = 0x0

图 40: 数据相关

查看波形图发现送入除法器的值正确,说明数据冒险已经经过前推得到解决。

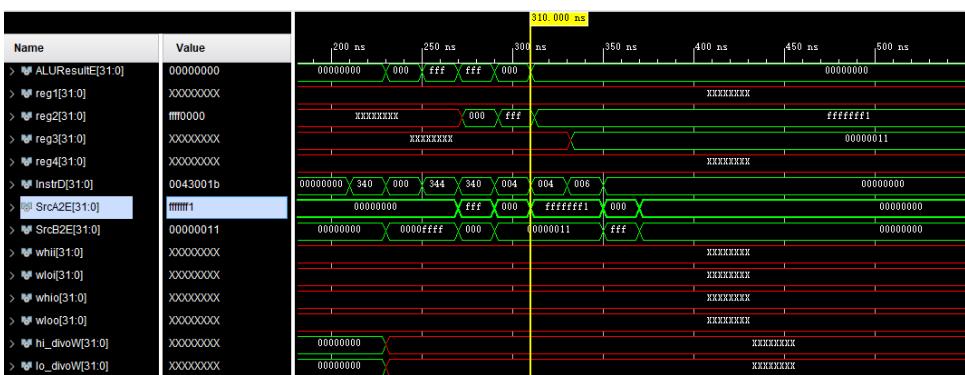


图 41: 除法器输入的值正确

传入除法器的值没有问题,那么问题应该出在除法器的控制信号上。经过进一步检查波形图发现

stallID 始终为 0, 也即 stall_divE 信号始终为 0。

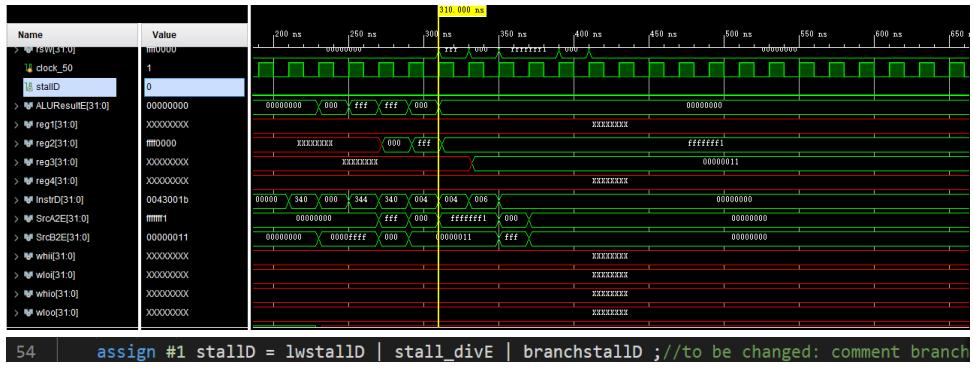


图 42: 除法器未正确停顿

推测为 div 状态机中 $\text{div_readyE} == 1'b0$ 的这个 if 块没有执行导致 stall_divE 未被正确赋值。

```

93 // divider state machine
94 always@(ALUControlE,div_readyE)begin
95   case(ALUControlE)
96     `EXE_DIV_OP:begin
97       if(div_readyE==1'b0)begin
98         start_divE <= 1'b1;
99         signed_divE <= 1'b1;
100        stall_divE <= 1'b1;

```

图 43: 除法器状态机未正确生效

检查 div_readyE , 发现该信号没有赋初值, 在执行除法时, 状态机对它的值做判断时, 其值可能因没赋初值而为 XXX, 从而没有进入上述 if 块。

检查仿真图发现检查 div_readyE 的值为 x。检查 datapath, div_readyE 来源于 div.v 模块中输出的 ready_o 端口, 在除法器中 ready_o 被初始化为 ‘DivResultNotReady’, 但事实上没有被正确赋值, 推测是 rst 信号没有正确生效导致。

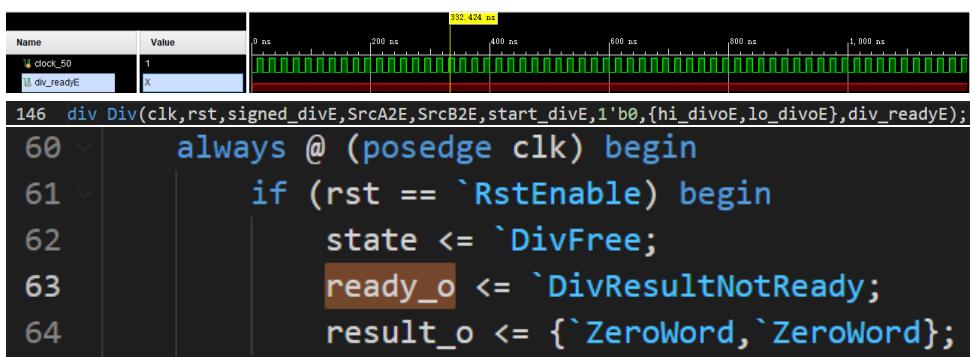


图 44: div_readyE 信号来源追踪

进行进一步检查发现 datapath 用的写法是 reset, 而连到除法器端口上的是 rst。问题应该就是出在这

里。

- (3) 错误原因: stallD 信号未能正确获取,导致除法器不能正确输出结果。
- (4) 修正效果: 将 div 端口调用的 reset 信号进行修正,修正后 div_readyE 信号能在仿真图上输出信号。但是除法器计算结果仍未正确输出。

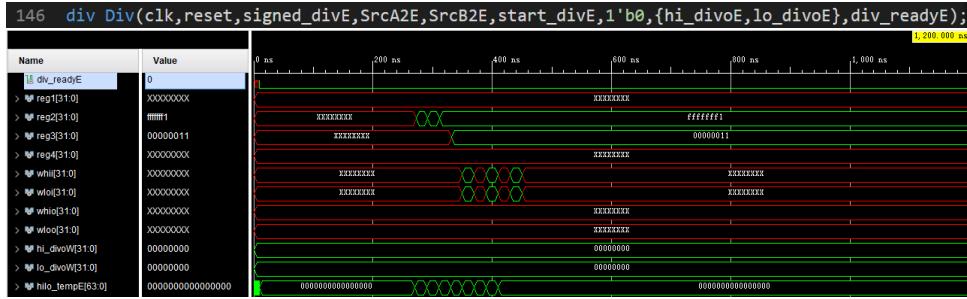


图 45: reset 信号修正

- (5) 归纳总结(可选): 对于接口处命名相近的信号名需要格外注意,尤其注意模块内定义和模块外输入的信号名称可能不一致,而赋值时可能习惯性用同一个变量名导致引用错误。故而在设计接口和信号时,要尽量保证两端一致,如果确实需要不用的变量名,应及时进行注释防止调用错误产生未知错误。

3.2.12 错误 12

- (1) 错误现象: 使用 vscode 更新 testbench 时若在 vivado 中也打开了这个 testbench,那么在 vscode 上对 testbench 的修改不能正确同步到 vivado 上。
- (2) 分析定位过程: 仿真时 vivado 会自动打开 testbench 文件,这个文件在 vscode 中也同样打开了,经过测试发现此时在 vscode 中的编辑不能同步保存,而 vivado 中的编辑能够正确同步保存。在 VSCode 中打开的仿真文件 test_bench 里添加了 div_readyE 信号并保存,然后又在 Vivado 中打开的仿真文件 test_bench 中修改了仿真结束时间并保存。再运行仿真发现之前添加的 div_readyE 信号不见了。
- (3) 错误原因: vscode 仅作为可视化编辑器使用,当 vivado 和 vscode 同时占用编辑时,vivado 具有更高的权限,故而导致 vscode 上的更新不能完成同步。
- (4) 修正效果: 直接在 vivado 工程上修改 testbench,或者将 vivado 上的 testbench 关闭后再在 vscode 上编辑。
- (5) 归纳总结(可选): 在使用可视化编辑器的时候要注意同一个文件的多处访问问题,防止出现更新不能正确同步的问题。

3.2.13 错误 13

- (1) 错误现象: 问题 11 解决了除法器运行控制信号的输入问题,解决后除法器应已经正确执行除法计算,但是除法仍未结束输出结果。
- (2) 分析定位过程: 推测可能是除法器控制逻辑依然存在问题。检查除法控制信号,发现停顿出了问题,

执行除法运算时,应该停顿直到除法执行完毕,而这里只停顿了一个周期。

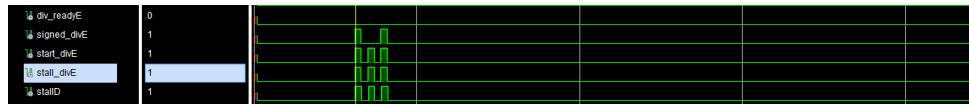


图 46: 停顿信号异常

首先检查状态机中的逻辑,没有发现问题,而且 div_readyE 也为 1,推测可能是 ALUControlE 信号没有停顿,导致触发器直接执行到 default 阶段。

```
always@(ALUControlE,div_readyE)begin
    case(ALUControlE)
        `EXE_DIV_OP:begin
            if(div_readyE==1'b0)begin
                start_divE <= 1'b1;
                signed_divE <= 1'b1;
                stall_divE <= 1'b1;
            end
        default:begin
            start_divE <= 1'b0;
            signed_divE <= 1'b0;
            stall_divE <= 1'b0;
        end
    end
end
```

图 47: ALUControlE 信号异常导致 case 执行异常

经过对比波形图发现确实是 ALUControlE 信号出现问题,是在流水线中被刷掉了。



图 48: ALUControlE 信号异常

- (3) 错误原因: flushE 信号用于决定是否刷掉 E 阶段的信号,以前实验四的时候,停顿只发生在 D 阶段,因此只要 D 停顿了,E 是一定要刷的,但是除法引起的停顿发生在 E 阶段,此时 E 的信号要保留,E 停顿也会引发 D 停顿,再用原来的冲刷逻辑就不对了。

56

```
assign #1 flushE = stallD; // to be changed
```

图 49: ALUControlE 信号异常

(4) 修正效果: 按照 ppt 的提示进行修改。

```
54 assign #1 stallD = lwstallD | stall_dive; // branchstallD ;//to be changed: comment branch
55 assign #1 stallF = stallD;
56 assign #1 flushE = lwstallD | branchD; // to be changed, to add |jumpD
```

图 50: stallD 信号修正

修改后结果正确,因显示不全,再次往后调了仿真结束时间。

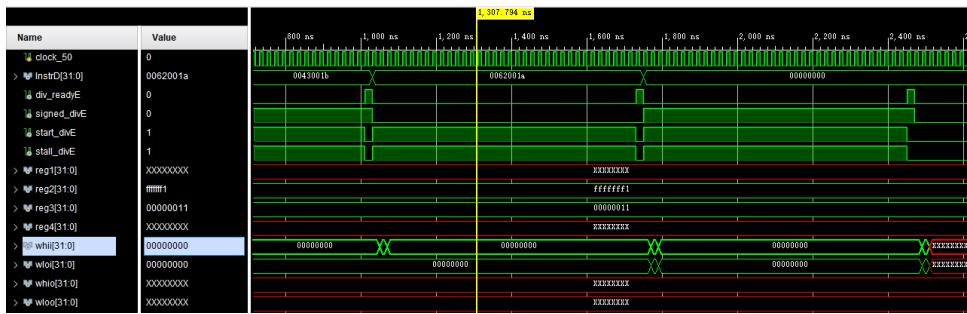


图 51: stallD 修正后波形图

(5) 归纳总结(可选): 计组实验 4 的实现逻辑中很多信号由于当时添加的指令数较少,不用加入过多的判断条件,在扩充指令的时候这些信号都需要得到及时的扩充以保证功能逻辑正确。

3.2.14 错误 14

(1) 错误现象: 错误 13 解决后,除法器能正确输出结果,但是没有正确保存至 hilo 寄存器中。

(2) 分析定位过程: 联想到乘法运算输出的时候也存在同样的问题,查找 hilo 寄存器写使能信号定义,发现果然也是没有添加除法判断条件导致。

```
275 hilo_reg hilo(clk,rst,hilo_writeW|multW|hilo_iW[63:32],hilo_iW[31:0],hilo_oW[63:32],hilo_oW[
276 mux2 #(32) resmux(hiloresultW2,ResultW1,hilo_readW,ResultW2);
277 assign whi=hilo_iW[63:32];
278 assign wlo=hilo_iW[31:0];
279 assign whio=hilo_oW[63:32];
280 assign wloo=hilo_oW[31:0];
```

图 52: hilo 寄存器写使能没有正确添加除法条件

(3) 错误原因: hilo 寄存器写使能没有正确添加除法条件。

(4) 修正效果: 处理方式同错误 8,在 hilo 寄存器写使能端口添加除法判断条件。

```
275 hilo_reg hilo(clk,rst,hilo_writeW|multW|div_signalW|hilo_iW[63:32],hilo_iW[31:0],hilo_oW[63:
```

图 53: hilo 寄存器写使能添加除法条件

添加后重新仿真,除法器正确计算出结果,cpu 将其写入 hilo 寄存器中。

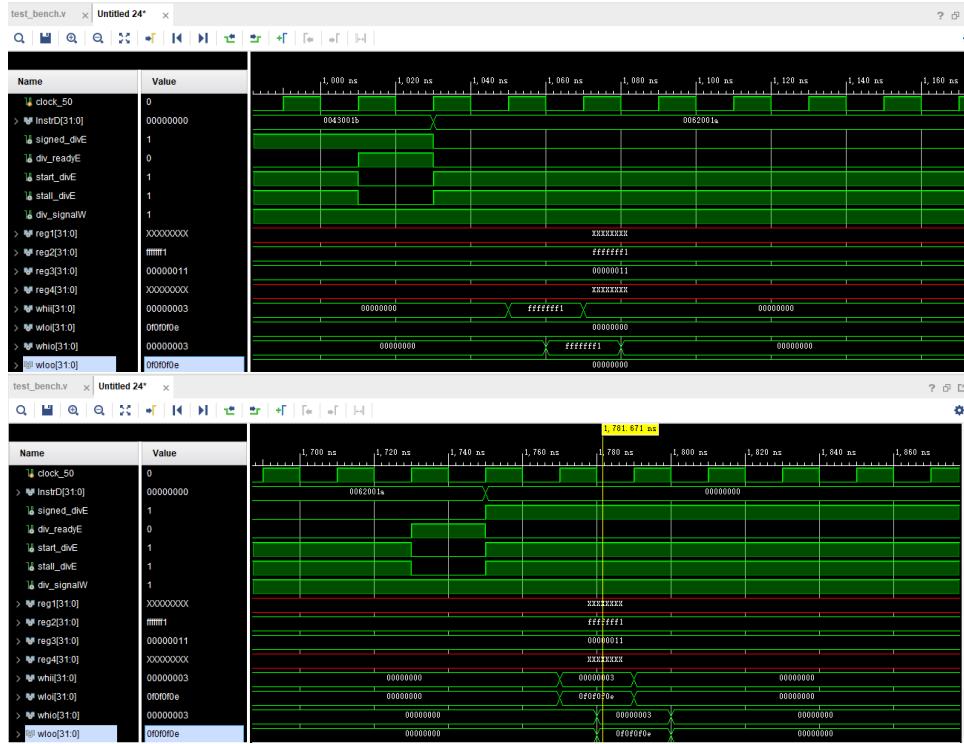


图 54: div 模块正确执行

3.2.15 错误 15

- (1) 错误现象: 用硬综设计资料 funcTest_independent 中的 ArithmeticTest 测试算术指令时,发现波形图中 hilo 寄存器存储的值在除法指令发射时跳变为 0。
- (2) 分析定位过程: 结合波形图推测是 hilo 寄存器的写使能定义不够严谨导致。
- (3) 错误原因: hilo 寄存器的写使能仅判断了除法指令,并没有除法执行完成的信号,也即在除法未完成的时候 hilo 寄存器的写使能就已经开放了,这在逻辑上存在瑕疵。虽然在当前流水线中执行除法时流水线停顿,除法执行完毕后写回的结果会把 0 覆盖,这个问题不会影响到最终结果,但安全规范起见还是进行修改。

```
275 hilo_reg hilo(clk,rst,hilo_writeW|multW|div_signalW,hilo_iW[63:32],hilo_iW[31:0],hilo_oW[63:
```

图 55: hilo 寄存器写使能添加除法条件不完善

- (4) 修正效果: 在 hilo 寄存器写使能判断条件中增加除法完成的信号后问题解决,hilo 寄存器不再在除法开始阶段跳变为 0。

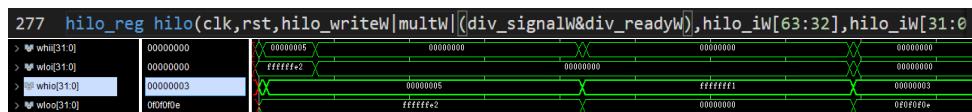


图 56: hilo 寄存器写使能添加除法完成判断条件

- (5) 归纳总结(可选): 对于需要定义的使能信号,需要严谨的考虑到各种运算指令的控制。

3.2.16 错误 16

- (1) 错误现象: 仿真图前半段没有使用 hilo 寄存器时 hilo 的输出为红色, 说明 rst 没有正确生效。
- (2) 分析定位过程: 查看 datapath, 发现 hilo 寄存器犯了和 div 模块调用时同样的错误, reset 信号错写成 rst 导致初始化信号没有正确发挥作用。在排查出这个错误的同时, 联想到 regfile 中也可以同样通过传入 reset 信号来初始化。

```
277 hilo_reg#(hilo(clk,rst,hilo_writew,multW)(div_signalW&div_readyW),hilo_iw[63:32],hilo_iw[31:0]
```

图 57: hilo 寄存器 reset 信号传入错误

- (3) 错误原因: reset 信号没有正确传入寄存器堆导致寄存器没有在开始完成初始化。
- (4) 修正效果: 在各个寄存器堆正确完成 reset 接口后, 波形图中对应寄存器在没有被调用过之前都能正确的被初始化, 在波形图中的表现即不再为红线。

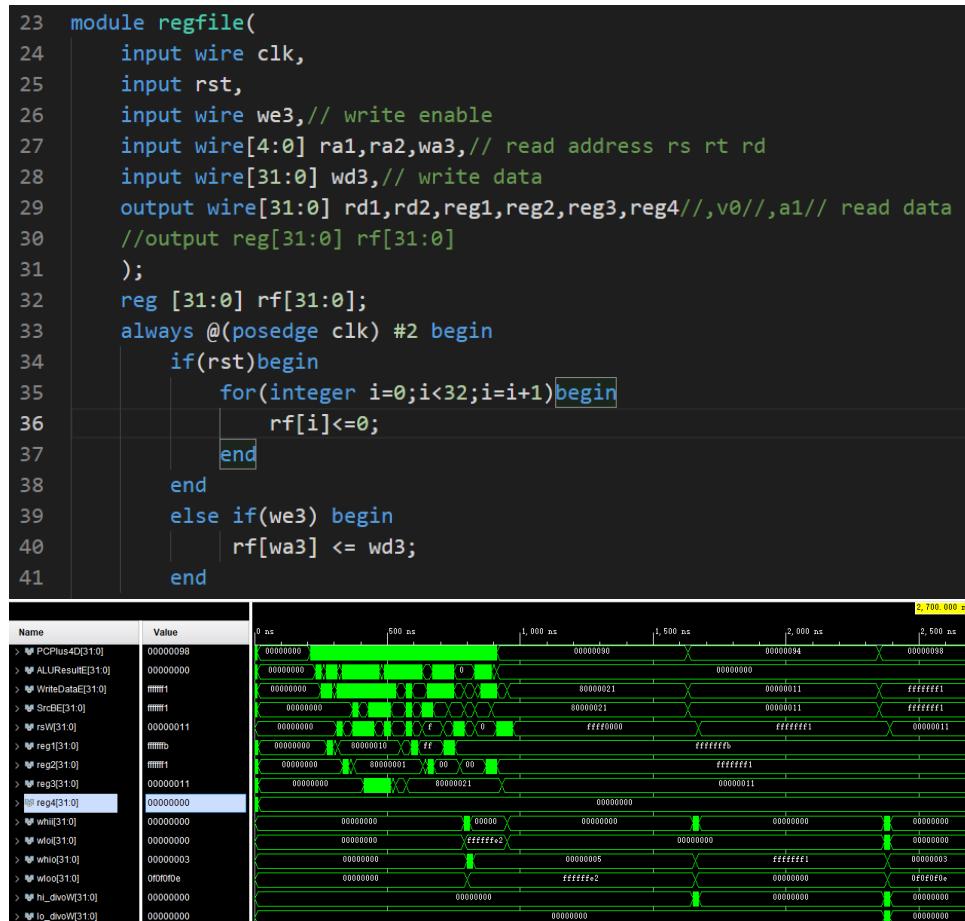


图 58: 修正寄存器的 reset 信号

- (5) 归纳总结(可选): 寄存器的初始化可能并不是必须的, 但是为了避免出现未知的错误, 保证 reset 功能的正确实现还是很有必要的。

3.2.17 错误 17

- (1) 错误现象: 经过排查, hilo 寄存器的前推逻辑没有增加除法的冒险前推。虽然在测试指令中没有涉及到除法的数据冒险,但是考虑到功能实现的完整性,添加完善除法的数据前推是必须的。

```
59 assign forwardHD = hilo_writeE | hilo_writeM | hilo_writeW | multE | multM | multW |
60     | div_signalE | div_signalM | div_signalW; // new, forward hilo
```

图 59: 数据前推信号未添加除法前推

- (2) 修正效果: 由于测试的指令集不涉及除法的数据冒险,故而修正前后的波形图没有区别。

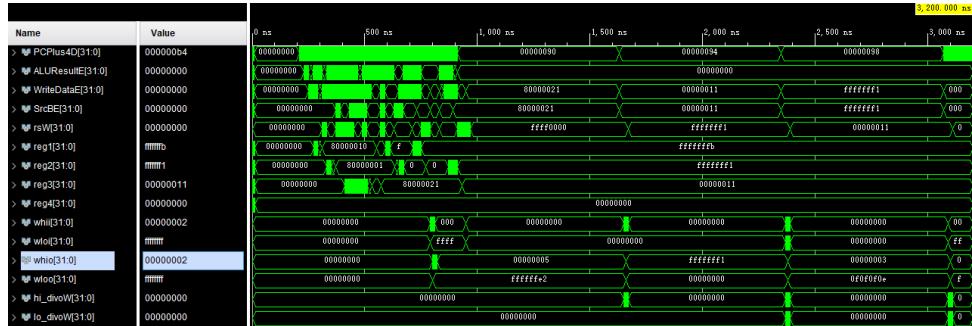


图 60: 测试波形图概览(后调了仿真结束时间)

3.2.18 错误 18

- (1) 错误现象: 跳转指令测试点测试执行跳转指令 2 周期后 pc 未能正确获得后续指令。
(2) 分析定位过程: 通过对比波形图和测试指令集发现是 ori 指令没有被正确加载读取导致。

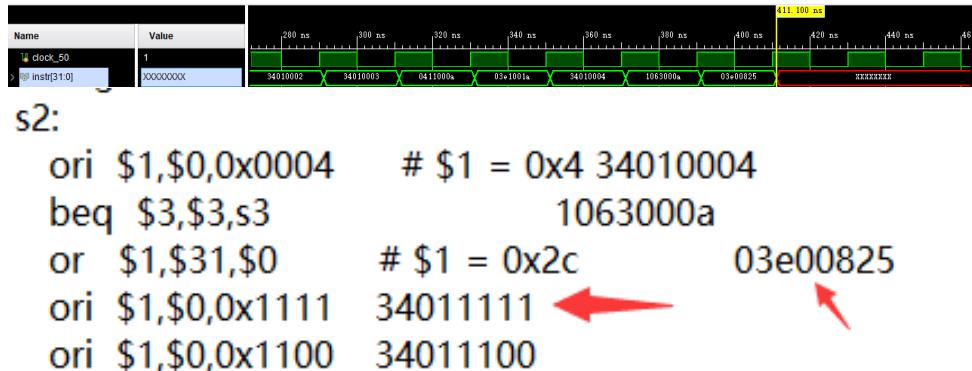


图 61: 结合波形图判断出错指令位置

查看 pc 值,发现此时确实没有获得正确的 pc 值。

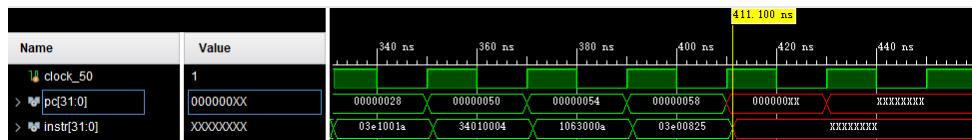


图 62: pc 获取异常

查看 pc 寄存器的赋值,检查 npc,在波形图上检查发现 npc 异常



图 63: npc 获取异常

查看 npc 的选择器,结合波形图,此时 jump 信号为 0,应选择 PCSrcin。

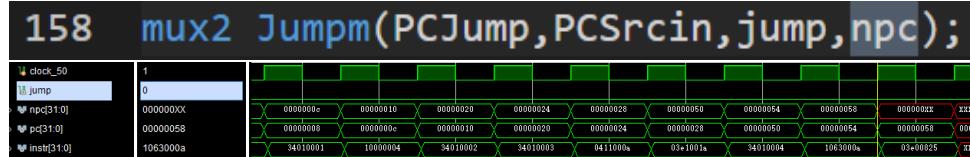


图 64: npc 信号来源

查看 PCSrc 的选择器和其输出对应波形图,发现 PCSrc 异常。

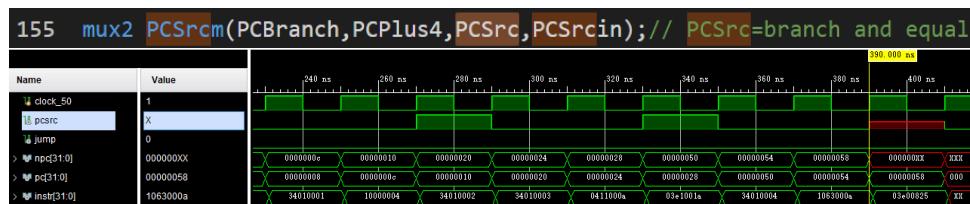


图 65: PCSrc 信号异常

查看 PCSrc 来源和其对应波形图,发现 zero 信号异常。

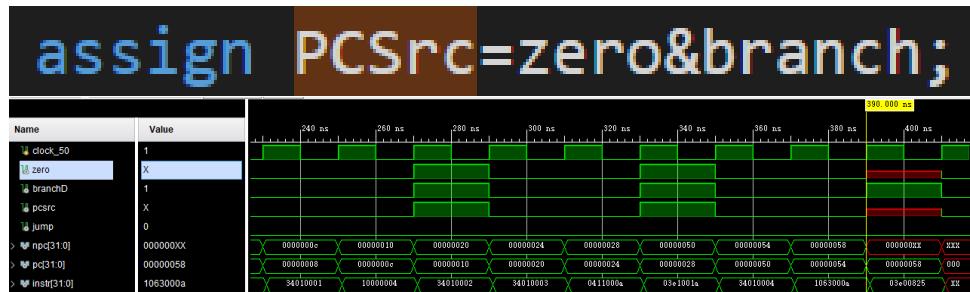


图 66: zero 信号异常

查看 zero 信号来源,发现来自 datapath 中的 eqcmp 产生,但是在 controller 中被错误的接成了输出信号。



图 67: zero 信号异常

修正 controller 中端口 zero 为输出信号,但问题并没有得到解决。

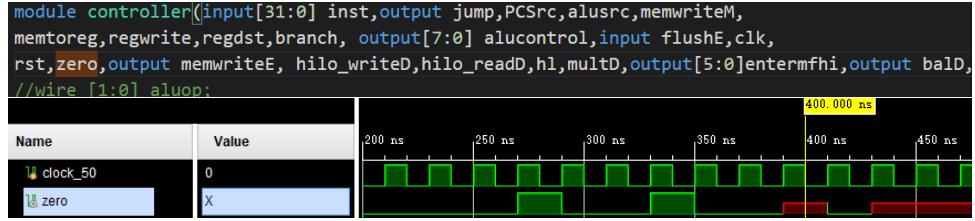


图 68: 修正 zero 信号类型后问题仍未得到解决

重新查看产生 zero 信号的 eqcmp, 发现 op 的值是正确的, 说明后续操作没有问题, 出错的应该是源操作数。查看波形图发现 SrcA2D 没有接上。

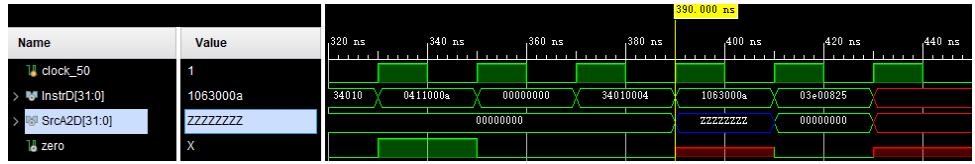


图 69: SrcA2D 信号没有正确接上

考虑 SrcA2D 的来源是 SrcAD 经过前推得到的信号, 而 SrcAD 读取的是 \$3 寄存器, 而波形图中显示这个寄存器没有被写入。

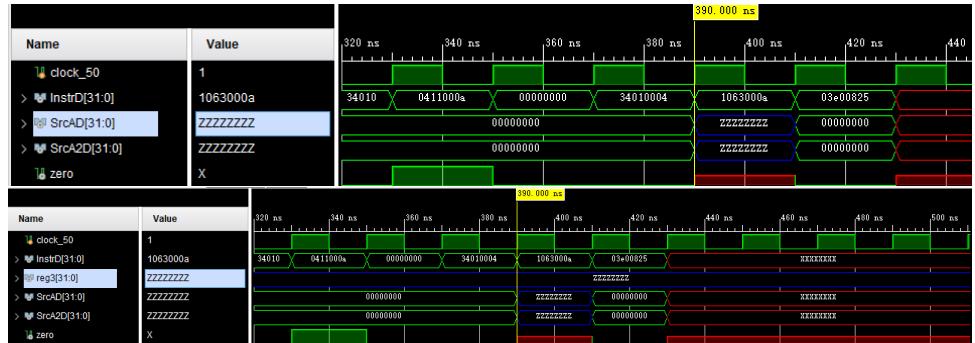


图 70: SrcAD 信号没有正确读取

根据测试指令集, \$3 寄存器应该已经被正确赋值, 但是波形图中始终为高阻态, 且写回基本寄存器的值 ResultW3 始终是高阻态。

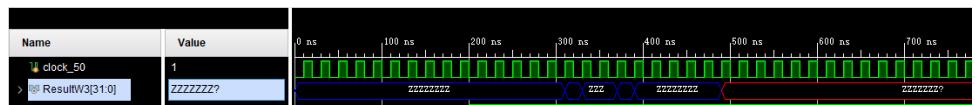


图 71: SrcAD 信号没有正确读取

准备查看 ResultW2 时发现接在 datapath 输出端口的 ResultW2 被改为 ResultW3, 导致 ResultW2 这个信号没有被声明过。

- (3) 错误原因: controller 中 zero 信号的输入输出类型设置错误, datapath 中输出端口的 ResultW2 修改后没有被正确声明导致无法被正常读取。
- (4) 修正效果: 能正常执行仿真, 波形图输出理想结果。



图 72: ori 指令处问题解决

- (5) 归纳总结(可选): 对数据通路中的各种信号可能以后也会涉及到更多的修改,每次修改的时候都要仔细查找这个信号在别的地方是否也有引用,并且要仔细斟酌信号的类型、位宽等属性,以防止出现不同信号表示同一逻辑含义但是其中部分信号缺少定义导致运行出错。

3.2.19 错误 19

- (1) 错误现象: 仿真时控制台输出无法获取 coe 文件
- (2) 分析定位过程: 根据控制台输出的错误信息,移除了部分报错的 coe 文件,此时通过了 Synthesis,但 simulation 依然报错。重启后 simulation 依然报错,尝试根据 Synthesis 中的 warning 进行修正,最终在将数据通路中重复定义的信号全都移除之后,仿正常执行。
- (3) 错误原因: 存在无法寻得路径的 coe 文件,数据通路中对部分信号进行了重复定义。
- (4) 修正效果: 删除错误的 coe 文件,对数据通路中重复定义的信号进行了袖中后,仿真运行正常执行
- (5) 归纳总结(可选): 在前面的仿真中数据通路同样也存在信号重复定义的 warning,但是由于没有影响仿正常执行,故而没有对其进行修正。在测试 j 类指令集的时候这一问题导致了仿真失败。所以在设计信号的时候,应该尽量按照规范,不要重复定义信号,以免出现不必要的问题。

3.2.20 错误 20

- (1) 错误现象: 测试 j 指令时仿真图显示执行了延迟槽中的指令。

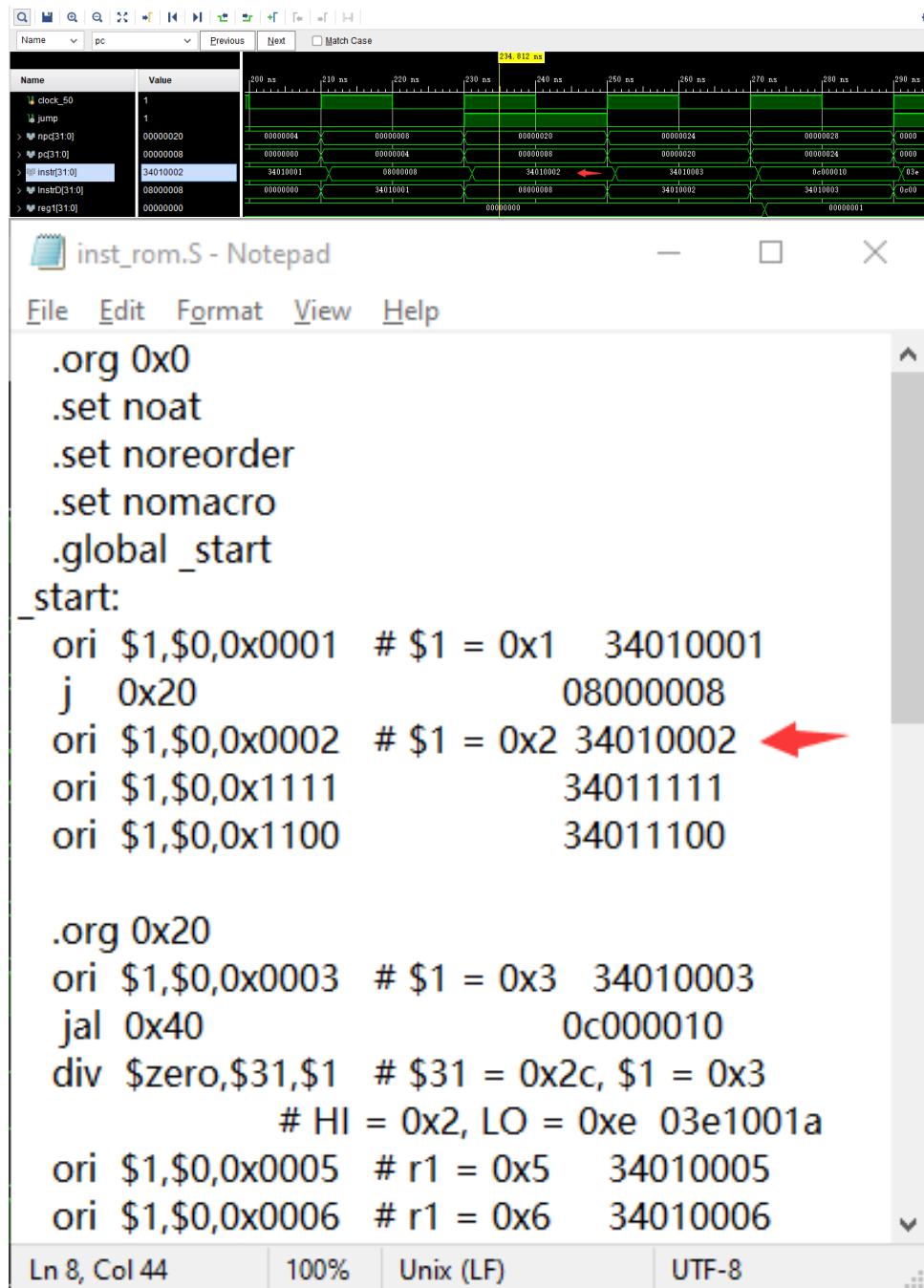


图 73: j 型指令延迟槽指令被执行

- (2) 分析定位过程: 返回查看 branch 对延迟槽的处理,结合波形图发现 branch 系列指令后面的延迟槽里的指令会被直接刷掉。由于本次实验 cpu 设计不涉及指令调度,所以对 j 类指令也同样处理。
- (3) 错误原因: 没有给 j 类指令刷掉延迟槽指令的信号。
- (4) 修正效果: 对信号进行部分筛选,让 jump 也刷掉延迟槽中的指令。找到 D 到 E 的信号,经过分析发现该信号不适用于 al 类型分支跳转,因为 al 类型需要把写的地址和寄存器传到写回阶段,而这些信号应该被保留。为最大限度减少分支跳转时无用的信号传到后面可能带来的不利影响,这里采取的措施是,对于 al 系列指令,只保留需要传到后面的信号,其它刷掉。使 j 系列指令能够刷掉后面的信号,但同时能保证需要传递到后面的信号得到保留。

```

57 ssign #1 flushE = lwstallD | branchD | jumpD; // |jumpD added
57 ssign #1 flushE = lwstallD | branchD | jumpD; // |jumpD added
218 fopenrc #(32) r10E(clk,reset,~stallE,flushE,hiloresultD,hiloresultE); //new
219 fopenrc #(1) r11E(clk,reset,~stallE,flushE,hilo_readD,hilo_readE); // new
220 fopenrc #(1) r12E(clk,reset,~stallE,flushE,forwardHD,forwardHW); //new
221 fopenrc #(1) r13E(clk,reset,~stallE,flushE,multD,multE); //new
222 fopenr #(2) r14E(clk,reset,~stallE,{balD,jalD},{balE,jalE}); //new for branch al, jump al
223 fopenr #(32) r15E(clk,reset,~stallE,pcD,pcE);
224 fopenr #(5) r16E(clk,reset,~stallE,rdD,rdE);
225 fopenr #(1) r17E(clk,reset,~stallE,jrD,jrE);

```

图 74: 修正 j 类指令控制信号刷新逻辑

- (5) 归纳总结(可选): 对于跳转系列的指令,事实上会有两类,一类是直接跳转的,一类是带地址跳转的。对于这两种类型的指令要进行仔细的区分,以确保二者的逻辑都能正确实现。

3.2.21 错误 21

- (1) 错误现象: 错误 20 经过修正后,波形图中显示 31 号寄存器仍未被正确写入,在写回阶段的写回寄存器的值 ResultW3 仍然为 0。
 (2) 分析定位过程: 观察波形图发现错误现象。



图 75: 31 号寄存器未被正确写入

于是找到产生该信号的选择器,尝试从相关的信号中寻找原因,发现 balD 的信号始终为 0。

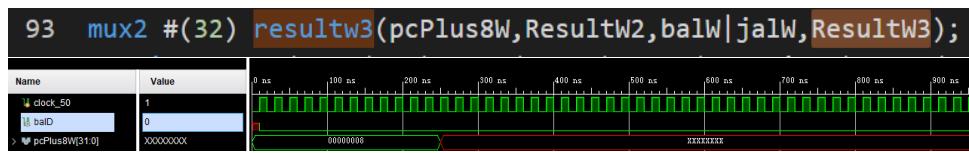


图 76: balD 信号异常

追踪到 maindecoder 中的信号产生发现由于后来对特殊指令的解码进行了额外实现,而在 maindecoder 中一开始使用了通用的解码,导致出现了问题。

- (3) 错误原因:maindecoder 中对于特殊指令的泛化定义在特殊指令进行特殊定义后应该删除,防止出现一个信号有两个不同的赋值方式导致结果混乱。
 (4) 修正效果: 删除泛化定义后当前问题还未完全解决。
 (5) 归纳总结(可选): 对于特殊指令的泛化定义在特殊指令进行特殊定义后应该删除,防止出现一个信号有两个不同的赋值方式导致结果混乱。这一问题在后续的代码编写中也需要特别注意。

3.2.22 错误 22

- (1) 错误现象: 错误 21 修正后 bal 信号仍没有正确显示。
 (2) 分析定位过程: 定位至错误指令 0411000a,手动译码后确定是 bgezal 类型指令,查看 maindecoder 中的信号赋值,发现 case 中使用了 rt,但是在 always 敏感块中没有添加。

添加了 rt 之后问题仍未解决,通过拉出一根信号 entermfhi 连接 EXE_BLTZAL 信号在波形图中观察发现一直为 xx,说明这个分支没有正确进入。

```
`EXE_BGEZAL, `EXE_BLTZAL:begin
    regwrite <= 1'b1;
    memtoreg<=1'b0;
    regdst<=1'b0;
    alusrc<=1'b0;
    branch<=1'b1;
    jump<=1'b0;
    memwrite<=1'b0;
    jal<=1'b0;
    jr<=1'b0;
    bal<=1'b1;
    hilo_writeD<=1'b0;
    hilo_readD<=1'b0;
    hl<=1'b0;
    multD<=1'b0;
    entermfhi<= `EXE_BLTZAL;
end
```

图 77: 分支没有正常进入

但是把这根线接到 case 外能正常显示,说明 rt 信号还是有问题。查找后发现是 controller 中没有正确给 maindecoder 接入 rt 信号导致。

- (3) 错误原因: maindecoder 中使用了 rt 作为 case 变量,但是在 always 敏感块中没有定义。同时,在 controller 中也没有正确给 maindecoder 传入 rt 值。
- (4) 修正效果: always 块添加 rt,在 controller 中给 maindecoder 接入 rt 信号,修正后 bal 信号显示正常。
- (5) 归纳总结(可选): 接口定义在涉及修改的每一个地方都要同步更新,作为 case 里的变量要注意添加到 always 敏感变量中。

3.2.23 错误 23

- (1) 错误现象: PCPlus8W 信号没有正确读入。
- (2) 分析定位过程: 检查生成 pcPlus8W 的 pc 信号,发现 pcE 没有连线。

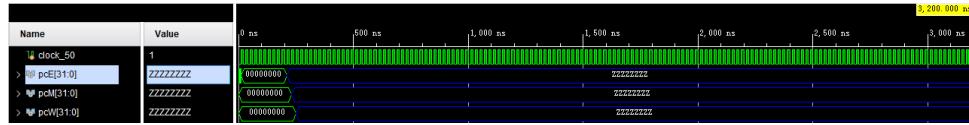


图 78: 没有正确连线

检查 datapath 中发现 pcF 的名字叫 cpc。

- (3) 错误原因: 变量名使用错误。
- (4) 修正效果: 将 pcD 修改为 pcF 后 ResultW3 信号正常,但 reg31 仍存在问题。

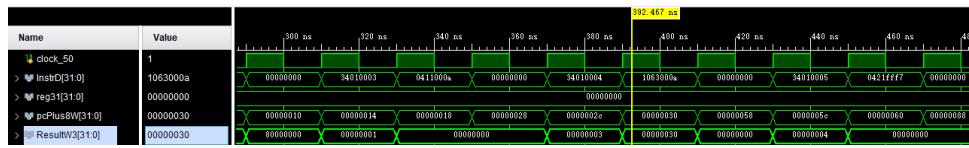


图 79: ResultW3 信号正常

3.2.24 错误 24

- (1) 错误现象: 承接错误 23 修正后波形图,reg31 未正确写入值。
- (2) 分析定位过程: 检查写 reg31 的相关信号,写入的值对了,只可能是要写的寄存器号不对。查看波形图发现写的寄存器号 WriteRegW2 不对。查看多选器逻辑发现 al 系列指令的目标寄存器的选择逻辑有误。

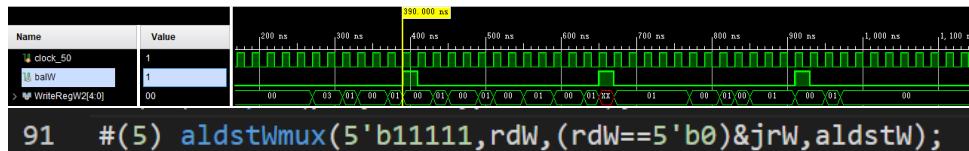


图 80: al 系列指令的目标寄存器选择逻辑出错

- (3) 错误原因: al 系列指令的选择逻辑颠倒。
- (4) 修正效果: 修正选择逻辑,仅在 jalr 指令指定 rd 寄存器时目标寄存器才是 rd,否则都是 31 号,更正后目标寄存器 WriteRegW2 正常。

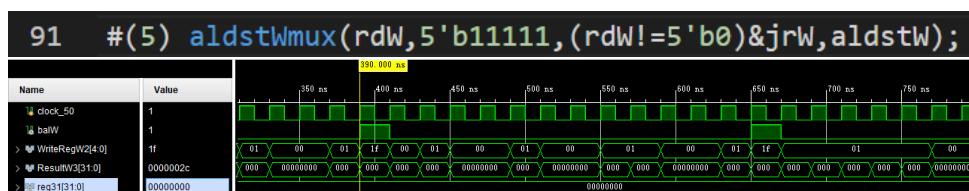


图 81: 修正选择逻辑

3.2.25 错误 25

- (1) 错误现象: 承接错误 24, reg31 仍未被正确写入。
- (2) 分析定位过程: 要写的值和寄存器号都是对的, 没有正确写入寄存器, 只能是寄存器没有使能。查看 regE 写使能发现改分支跳转相关的信号的 flush 时忘记了 rewrite, 导致它还是会被刷掉。
- (3) 错误原因: 寄存器写使能没有被正确接入。
- (4) 修正效果: 修正后 31 号寄存器正确写入信号。

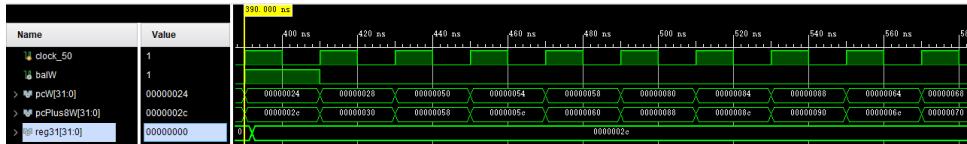


图 82: 修正选择逻辑

- (5) 归纳总结(可选): 做逻辑修改时, 没有把相关的所有信号都考虑进来, 导致出现疏漏。避免这样的错误很困难, 要求思路必须十分清晰。

3.2.26 错误 26

- (1) 错误现象: branch 暂停时刷掉延迟槽的信号 flushE 没被暂停。

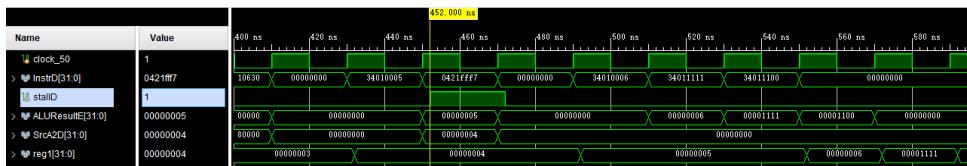


图 83

- (2) 分析定位过程: 修正暂停逻辑。

```
76 | assign #1 flushE = (lwstallD | branchD | jumpD) & ~stallD; // | jumpD added
```

图 84

修正后 flushE 仍未被正确暂停。

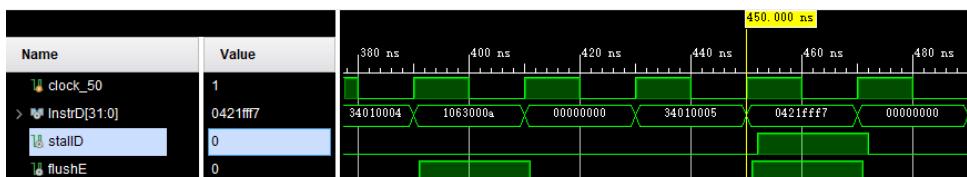


图 85

找到 instrD 的更改逻辑, 发现应该修改的是 flushD 而不是 flushE。

- (3) 修正效果: 将 flushD 按之前的逻辑进行正确修改后问题解决。

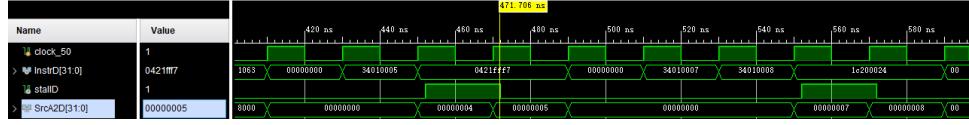


图 86

3.2.27 错误 27

- (1) 错误现象: j 类指令测试集中存在的 raw 没有被正确解决。
- (2) 分析定位过程: 检查产生 SrcA2D 相关的信号,发现 forwardAD 不正确。

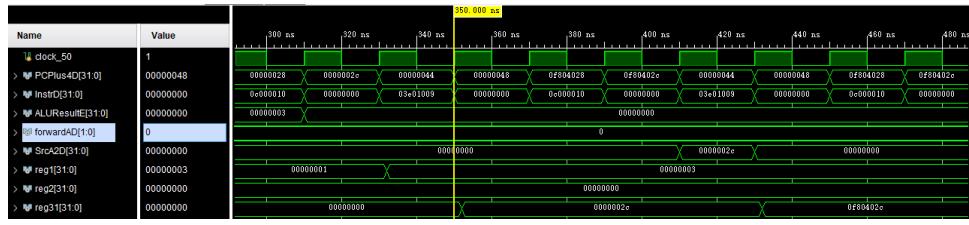


图 87

找到产生 forwardAD 的地方,查找与之相关的信号,结合波形图发现 WriteRegE 不对。查看产生 WriteRegE 的选择器,它在设计中被修改,但在代码中忘记修改了。

- (3) 错误原因: 设计中修改的 WriteRegE 没有在代码中被修正。
- (4) 修正效果: 修正后问题解决。

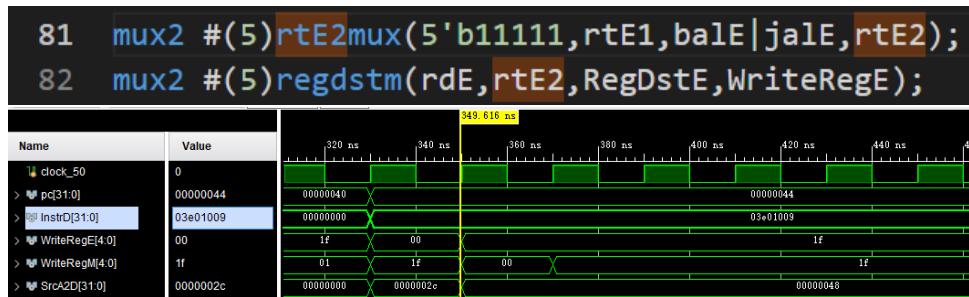


图 88

- (5) 归纳总结(可选): 设计时的逻辑在改代码时忘记修改,因为一处设计可能涉及几处改动,往往改过一处后会忘记改另一处,可以在着手修改前先记录下要改哪几个地方。

3.2.28 错误 28

- (1) 错误现象: 错误 27 解决后暂停信号异常,导致指令加载异常。
- (2) 分析定位过程: 查看波形图发现出现了错误的暂停信号,此时跳转读 31 号寄存器的冒险可以直接前推不需要暂停。

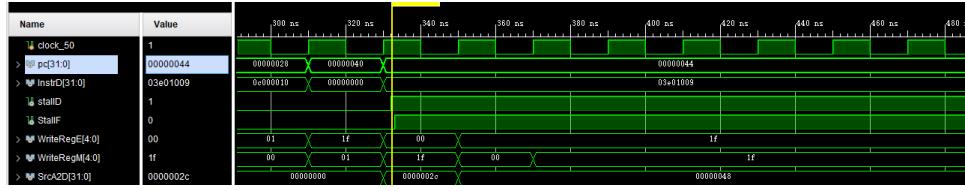


图 89

查找到此处发现问题，设计时忽略了 jal 可以直接前推，不需要暂停。

```
71 assign #1 branchJrstallD = (branchD|jrD) &
72           (regwriteE & (writeregE == rsD | writeregE == rtD) |
73           (memtoregM | hilo_readM) & (writeregM == rsD | writeregM == rtD));
```

图 90

(3) 错误原因：暂停信号逻辑存在问题。

(4) 修正效果：修正逻辑后能正确加载下一条指令。

```
71 assign #1 branchJrstallD = (branchD|jrD) &
72           (regwriteE & (writeregE == rsD | writeregE == rtD) & ~balE & ~jalE |
73           (memtoregM | hilo_readM) & (writeregM == rsD | writeregM == rtD));
```

Name	Value
clock_50	1
> InstrD[31:0]	03e01009

图 91

3.2.29 错误 29

(1) 错误现象：错误 28 修正后问题仍未得到完全的解决。指令执行陷入循环。

(2) 分析定位过程：查看波形图和代码发现是由 E 阶段的条件引起的。

```
73 assign #1 bjfromE = regwriteE & (writeregE == rsD | writeregE == rtD) & ~balE & ~jalE;
74 assign #1 bjfromM = (memtoregM | hilo_readM) & (writeregM == rsD | writeregM == rtD);
```

Name	Value
clock_50	1
> InstrD[31:0]	03e01009
bjfromE	1
stallD	0

图 92

开始的猜测是 E 阶段那条指令被刷掉了，但和 jal 相关的一部分信号因为 jal 写地址要用，所以被保留下，但 jal 却被刷掉了，所以为 1。结合代码实现发现这个猜测并不正确，但是发现 datapath 中没有声明 jalE 但是仿真却没有报错。

基于这一现象，推测是 al 系列有些指令不需要检验 rt，因此产生了误导。E 阶段这条是 j 延迟槽中刷掉的空指令 0+0，而 rtD 恰好为零，所以导致错误。

(3) 修正效果：添加 flushedM 信号记录当前 EX 阶段的指令是不是被冲刷掉的指令，不从被冲刷掉的指令那儿旁路，不再暂停。

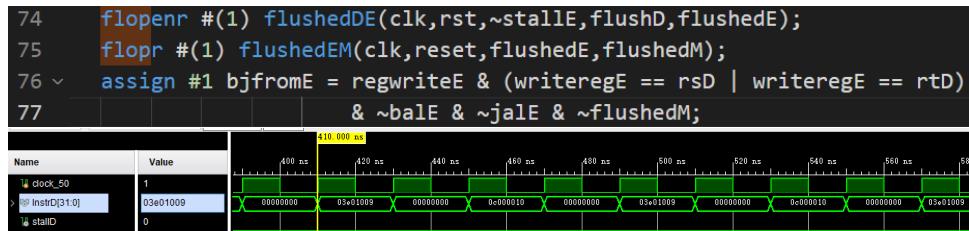


图 93

(4) 归纳总结(可选): 冲刷处理有所欠缺。

3.2.30 错误 30

- (1) 错误现象: 错误 29 修正后没有正确加载除法。
- (2) 分析定位过程: 结合波形图,jalr 指令从 \$31 加载的地址是对的,应该是选择哪个地址进行跳转时出了问题。

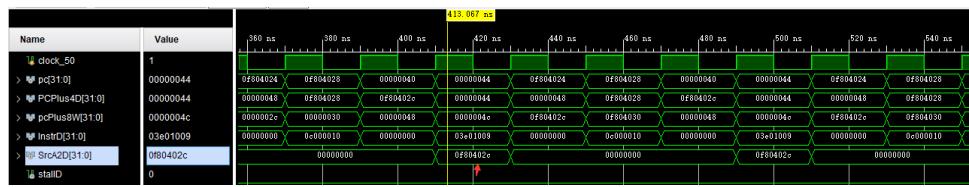


图 94

检查数据通路代码后发现忘记按照新设计的通路在代码中添加选择器了。

- (3) 修正效果: 修正后可以发现波形图中可以看到没有加载除法。

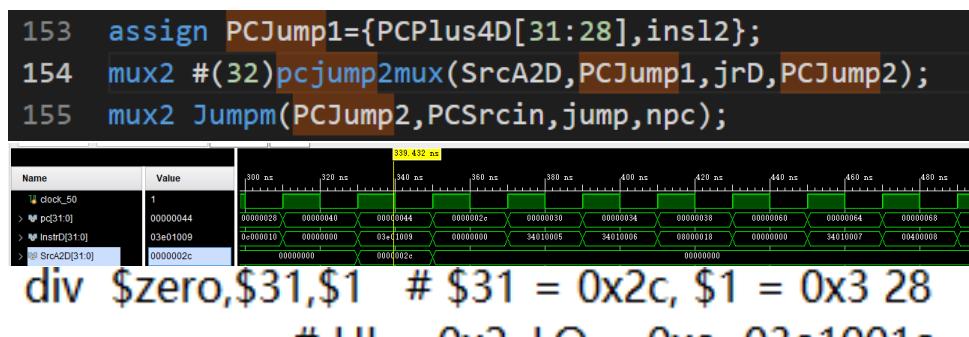


图 95

这是因为除法在 jalr 的延迟槽中,pc+8,跳过了。

```
jal 0x40          0c000010 24
div $zero,$31,$1 # $31 = 0x2c, $1 = 0x3 28
                  # HI = 0x2, LO = 0xe 03e1001a
ori $1,$0,0x0005 # r1 = 0x5 → 34010005 2c
```

图 96

波形图中可以看到加载了除法下面一条 ori 指令,这是正确的。但是 jalr 似乎没有能把它 pc+8 写到 \$2 中。

3.2.31 错误 31

- (1) 错误现象: 承接错误 31。
- (2) 分析定位过程: 推测是 regdstD 到 RegDstE 的那个触发器出了问题。检查旁路逻辑后修改如下。

```
279 mux2 #(32) alurestlm2mux(pcPlus8M,ALUResultM1,ba1M|jalM,ALUResultM2);
280 mux3 #(32) forwarddaemux([SrcAE,ResultW3,ALUResultM2,forwardAE,SrcA2E]); // 1 to 3
```

图 97

修改后波形图显示如下指令写回阶段写回的值似乎没连上。

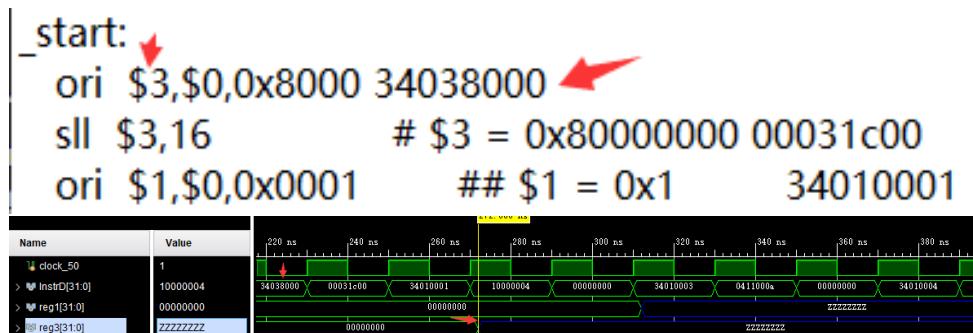


图 98

查看波形图发现 ResultW1、3 信号为全蓝。检查产生 ResultW1 的地方,发现是 ALUResult 出了问题,检查输入到 ALU 的源操作数,SrcB2E 不对。

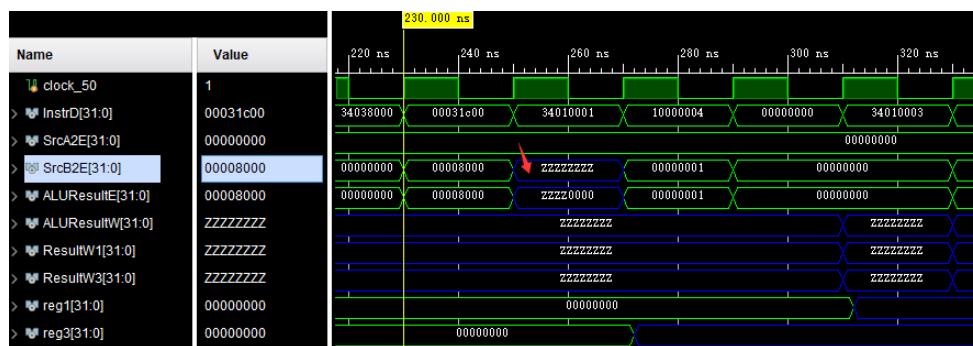


图 99

找到波形图中箭头标注处 SrcB2E 应该选为 rtE,应该是选择过程出了问题。



图 100

SrcB2E 是选择 rtE 对应的寄存器中存储的值, 在这里是 WriteDataE, 该信号出了问题。

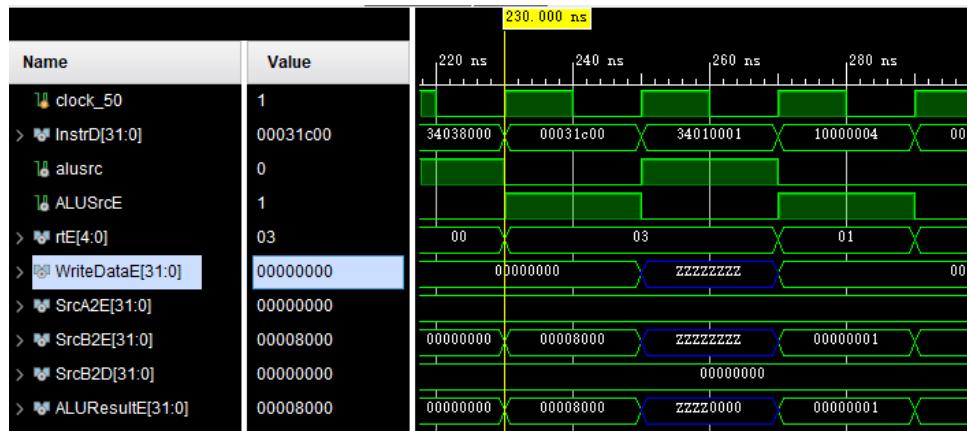


图 101

查找设计的数据通路发现其由三路选择器产生。

```
283 mux3 #(32) Forwardbemux(SrcBE,ResultW3,ALUResultM2,forwardBE,WriteDataE); // 1 to 3
```

图 102

结合波形图此时 forwardBE 的值为 2, 结合指令测试集对比此时确实存在 \$3 的数据冒险, 应选择设计的数据通路图中的 ALUResultM1, 在数据通路图中对比发现这个信号在 datapath 中没有被声明。这一现象可能是因为之前只有一个 ALUResultM 信号, 加了选择器后分成了 ALUResultM1 和 ALUResultM2, 之前声明的 ALUResultM 改成了 M2, M1 就忘记了。

(3) 修正效果: 修正后此处问题得到解决。

3.2.32 错误 32

(1) 错误现象: j 指令延迟槽中的指令没有被刷掉。

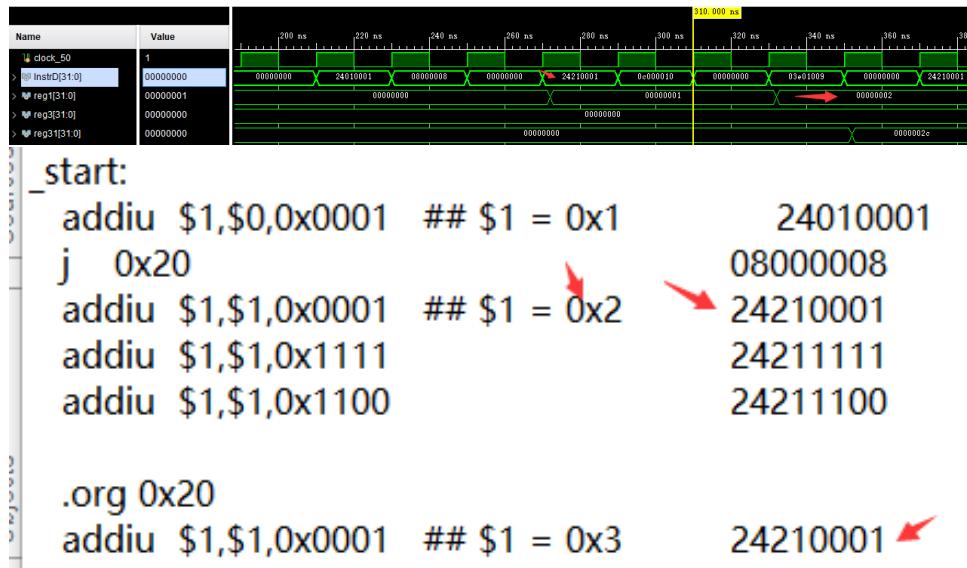


图 103

(2) 分析定位过程: 之前的 branch 指令也存在类似的问题。不过 branch 出现问题是因为发生数据冒险需要暂停,而此时不需要暂停。

但是经过仔细比对后发现并不是延迟槽中的指令没有被刷掉,而是跳转目的地的指令 (0x20) 和延迟槽 (0x08) 中的指令完全相同。都是 addiu 1,1,0x0001, 而.S 中 0x20 处的指令执行完后的结果是 \$1=3, 也就是说认为执行了延迟槽 (0x08) 中的指令,即延迟槽中的指令确实不需要刷,而一开始的设计把延迟槽中的指令刷掉了,因此与答案不符。

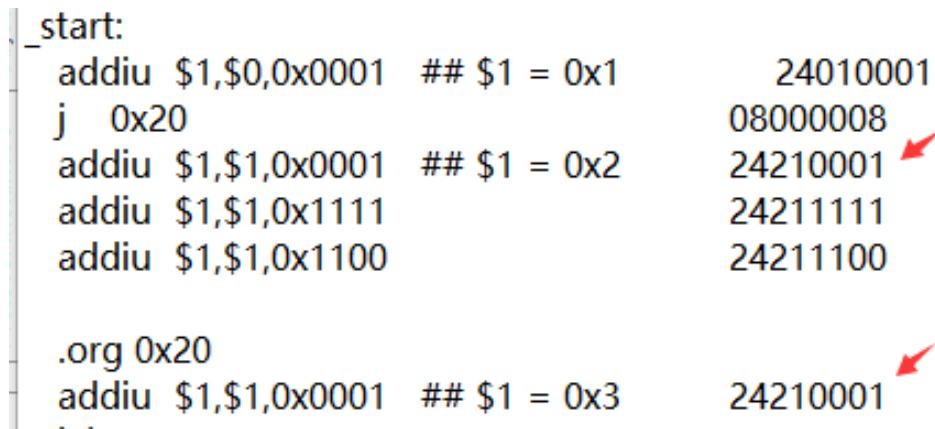


图 104

所以,需要对之前的设计进行修改,使得不再刷掉延迟槽的指令。

(3) 修正效果: 修改为不带 flush 信号的触发器。

```

159 //flopenrc#(32)r2D(clk,reset,~stallD,flushD,Instr,InstrD);
160 flopennr#(32)r2D(clk,reset,~stallD,Instr,InstrD);

```

图 105

依测试集的情况来看,延迟槽中指令与后面跳转目标指令的冒险是要处理的,于是在旁路逻辑中去掉 flushedM。

```

76      assign #1 bjfromE = regwriteE & (writerregE == rsD | writerregE == rtD)
77          & ~balE & ~jalE & ~flushedM;

```

图 106

改过后波形图正确了,因为连着取了两条同样的指令,所以看上去两个周期是连在一起的 24210001。

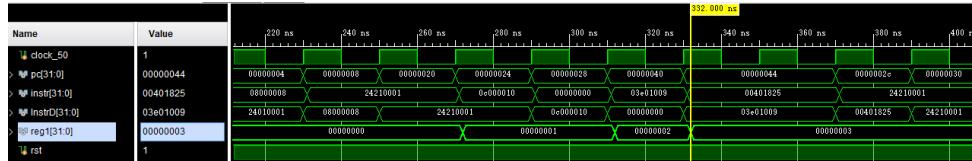


图 107

3.2.33 错误 33

(1) 错误现象: 空指令冒险导致流水线暂停。

(2) 分析定位过程:

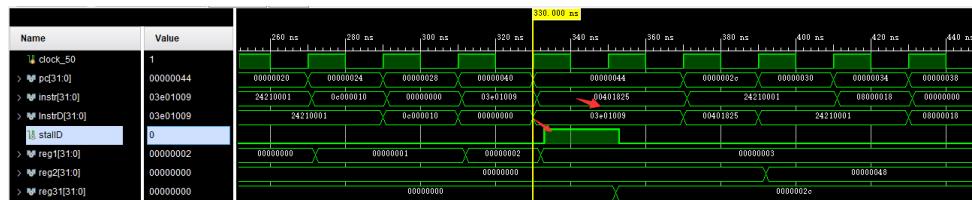


图 108

波形图中产生停顿,因为 0c000010 这条 jal 指令的延迟槽中是空指令,以 \$0 为目标寄存器,而 jalr 的 rt 字段恰好是 0,因此会引发数据冒险,导致停顿。

jal 0x40	0c000010
nop ←	
addiu \$1,\$1,0x0001 ## r1 = 0x4	24210001
addiu \$1,\$1,0x0001 ## r1 = 0x5	24210001
j 0x60	08000018
nop	
 .org 0x40	
 jalr \$2,\$31	03e01009

图 109

(3) 修正效果: 修改指令如下图。

```

76 jfromE = regwriteE & (writerregE == rsD | writerregE == rtD)
77     & ~balE & ~jalE & writerregE!=5'b0;// & ~flushedM;
78 jfromM = (memtoregM | hilo_readM) & (writerregM == rsD | writerregM == rtD) & writerregM!=5'b0;

```

图 110

修改后不再产生停顿。

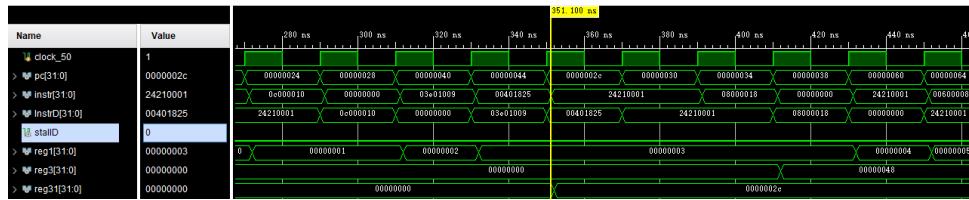


图 111

3.2.34 错误 34

(1) 错误现象: j 类指令集测试时发现延迟槽中的除法只停顿了一个周期。

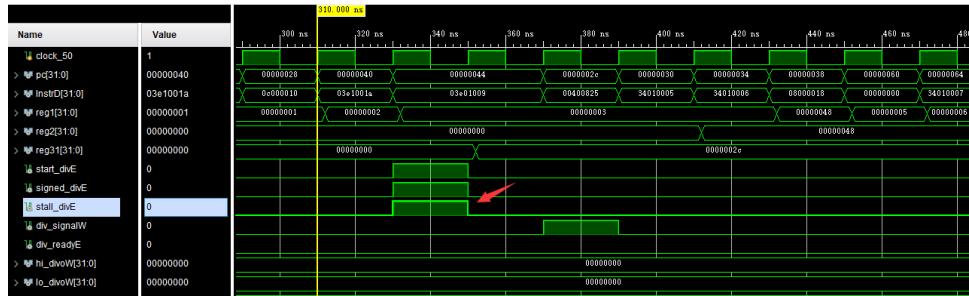


图 112

(2) 分析定位过程: 这个问题在之前设计除法的时候碰到过类似的。查找后发现问题也是出在 ALUControlE 上。

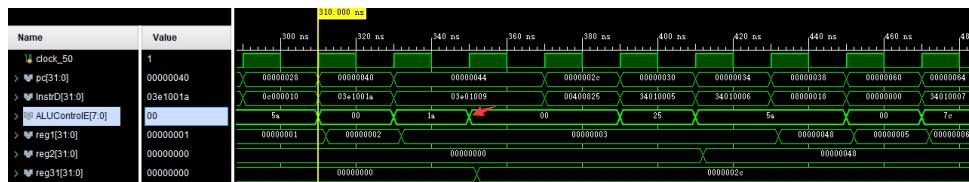


图 113

div 所在的延迟槽上下两条指令都是跳转指令,jalr 加载进来会导致 jumpD 为 1,从而 flushE 为 1,导致 E 阶段正在执行的除法指令被刷掉。

原来考虑的情况是,D 暂停(或是分支跳转指令)后,E 阶段执行完毕流走了,D 因为暂停不能流向 E,所以刷掉 E,在 E 不会出现暂停的时候这样做是没有问题的,但是现在引入了除法,除法在 E 阶段暂停。这时如果 D、E 恰好都要暂停,那么按照原来的逻辑,会错误地认为 D 停顿一周期后 E 已经执行完毕流向下一流水级,从而刷掉 E,但实际上 E 也发生暂停,这就会导致错误地刷掉暂停的 E。

(3) 修正效果: 处理方法就是向处理 flushD 一样,在 E 暂停时禁止刷掉 E。

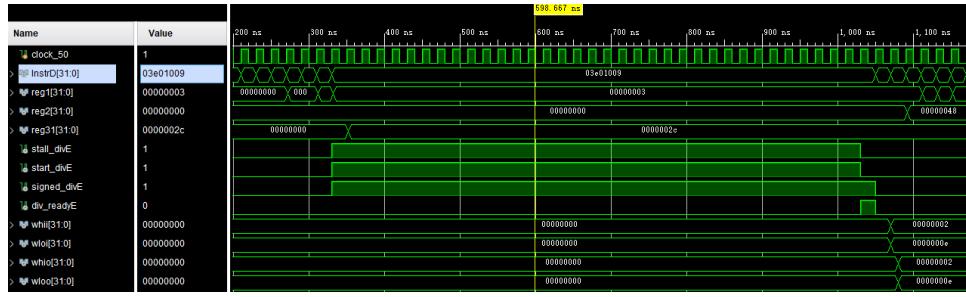


图 114

修正后除法能正确停顿。

3.2.35 错误 35

- (1) 错误现象: 仿真报错。
- (2) 分析定位过程: 根据控制台打印的错误信息查找出错信息。共找到 3 处错误。

```
24 module top(input clk,rst,output[31:0]instr,pc,output memtoreg,memwrite,pcsrc,alusrc,regdst,r
25 zero, output[7:0]alucontrol,// new 2 to 7
26 output[31:0]dataaddr,WriteData,output reg[31:0]ReadData2, output [31:0]SrcAD,SrcBD,npc,Resu
```

图 115: top 端口修改的信号名在 testbench 中没有同步修改

```
76 flopr alucontrollem(clk,rst,ALUControlE,ALUControlM);
```

图 116: 触发器忘记赋名

```
24 module top(input clk,rst,output[31:0]instr,pc,output memtoreg,memwrite,pcsrc,alusrc,regdst,r
25 zero, output[7:0]alucontrol,// new 2 to 7
26 output[31:0]dataaddr,WriteData,output reg[31:0]ReadData2, output [31:0]SrcAD,SrcBD,npc,Resu
```

图 117: 修改 top 端口时误加 reg 类型

- (3) 修正效果: 修正后仿正常出波形图而不是报错。

3.2.36 错误 36

- (1) 错误现象: 仿真图中 readdata 信号异常。

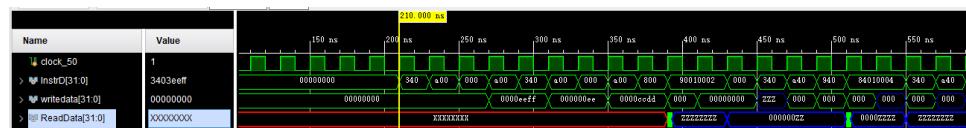


图 118

- (2) 分析定位过程: 红 × 说明没有进行初始化,先将 readdata 信号初始化。初始化后仍然存在高阻态。



图 119

查看测试指令集,发现 sb 指令要存的数据 writedata2M 不对。

start:

<pre>ori \$3,\$0,0xeeff sb \$3,0x3(\$0) # [0x3] = 0xff</pre>	3403eeff a0030003
---	----------------------

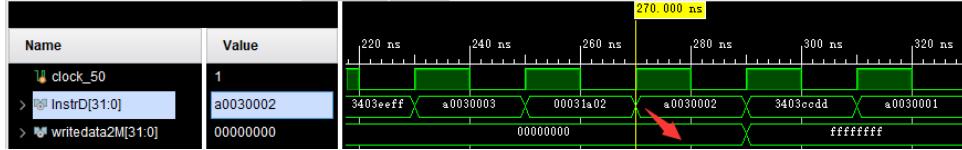


图 120

查看设计 writedata2M 由 WriteData 的低八位重复组成,而波形图中显示 writedata 是对的, ALUControl 也是对的,最后在 always 块中找到字节的 case 逻辑中误将 ALUControlM 写成 ALUControlE。

```

76 // logic for byte and half word
77 flopr alucontrolem(clk,rst,ALUControlE,ALUControlM);
78 // Parts of codes below are copied from ppt
79 always@(*)begin
80   if(rst)begin
81     ReadData2<=32'b0;
82     writedata2<=32'b0;
83   end
84   else begin
85     case(ALUControlE)
86       `EXE_LW_OP:begin
87         sel<=4'b0000;
88         writedata2<=32'b0;
89         ReadData2<=ReadData;
90       end
91       `EXE_LB_OP:begin
92         sel<=4'b0000;
93         writedata2<=32'b0;

```

图 121

同时,也发现 sel 没有正确进行初始化。修改后 ALUResultE 没有输出正确的值,对比发现是 ALU 中没有添加使之能识别这些新的读写指令。

(3) 修正效果: 修正后 dataaddr 输出正确的值。

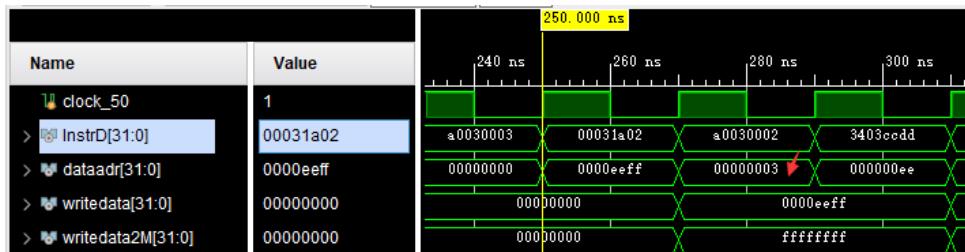


图 122

3.2.37 错误 37

(1) 错误现象: 取字节 lb 的结果错误。同时, 承接错误 36, readdata 依然为高阻态。

srl \$3,\$3,8	00031a02
sb \$3,0x0(\$0)	# [0x0] = 0xcc a0030000
lb \$1,0x3(\$0)	# \$1 = 0xffffffff 80010003 ←
lbu \$1,0x2(\$0)	# \$1 = 0x000000ee 90010002
nop	

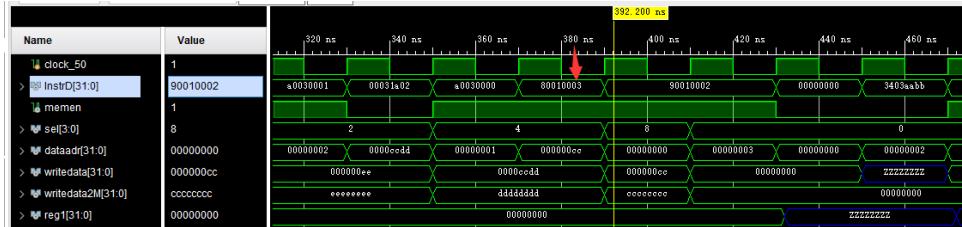


图 123

(2) 分析定位过程: 从读字节的处理逻辑可以看出, 实际读写的时候是以字为单位, 也就是说送入数据存储器的地址应该是按字对齐的, 故对地址进行对齐处理。

修改对齐后错误仍未得到解决, 查看 memen 使能信号发现其用的是译码阶段产生的, 忘记添加流水级间的寄存器把它沿流水级逐级下传。

修改后使能信号正确输出, 但是 readdata 依旧高阻态。

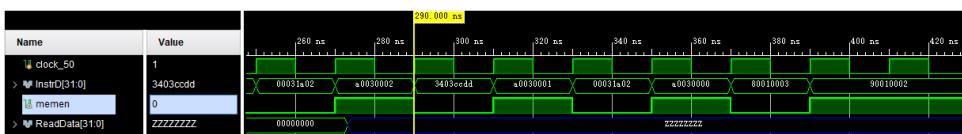


图 124

测试集中相邻两条读指令中, 第二条读指令产生了停顿, 但它们之间没有数据冒险。

lb \$1,0x3(\$0)	# \$1 = 0xffffffff 80010003
lbu \$1,0x2(\$0)	# \$1 = 0x000000ee 90010002
nop	

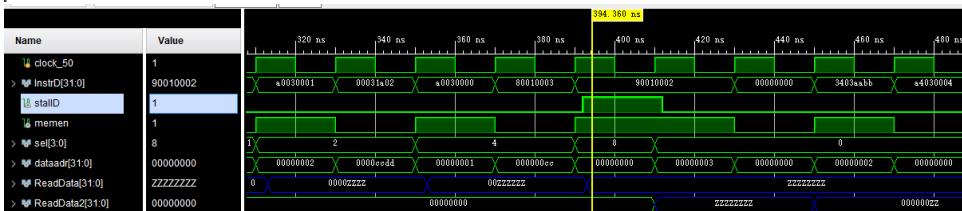


图 125

判断载入停顿的逻辑默认译码阶段读 rt, 实际上第二条载入指令在译码阶段写 rt, 因此误把 WAW 当成 RAW。

只有 branch 系列指令会在译码阶段用到读出来的 rt, 而且似乎也不是全部 branch 指令, 如果对非 branch 指令不在译码阶段暂停的话, 那么 R 型指令又会在执行阶段用到 rt 读出来的值, 此时 lw 正在访存, 仍需暂停, 因此要捋清这个是否真的需要在译码阶段暂停的逻辑颇为不易, 而连着两条写同一个寄存器的取字指令应该颇为罕见, 因为这意味着第一条取字指令取出来的值还没有被使用就被覆盖掉了。所以这样一个不必要的停顿带来的代价相比移除它所带来的复杂度而言是可以接受的。

3.2.38 错误 38

- (1) 错误现象: 承接上述错误, 高阻态的问题仍未得到解决。
- (2) 分析定位过程: 将 top 模块换回计组 4 的版本, 发现 writedata 出现问题。



图 126

结合测试指令集, 发现是前推逻辑存在问题, 导致没有正确执行前推。

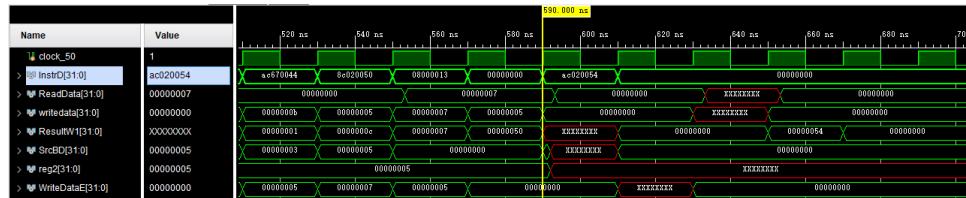


图 127

reg2 变成 xxx 是因为 lw 指令没能把读到的载入进去。

检查生成写回值的选择器, 结合波形图发现发现 ReadDataW 出现问题, 信号为全红 x。

找到 ReadDataW 的选择器, 只能是 datapath 中的 ReadData 出了问题, 发现 mips 中传回去的还是改过的 ReadData2, 修改成 ReadData 后 reg2 能正确写入。

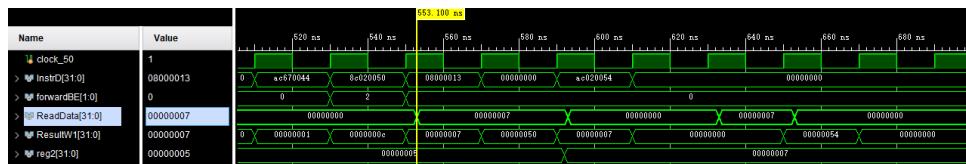


图 128

检查发现实验四 test 指令集中, 最后一条 sw 由 j 跳转过来, 刚好跳过 addi, 而实验四又是采取的刷掉延迟槽中的指令的策略, 因此 addi 得不到执行, 所以实验四的结果是正确的。

接下来尝试把 top 中的逻辑替换成可以处理字节和半字的逻辑。

首先替换 addr, 换完后仍然正确。

替换 sel 后仍然正确。

换完 writedata2 后不再正确, 于是问题出在 writedata2 上。

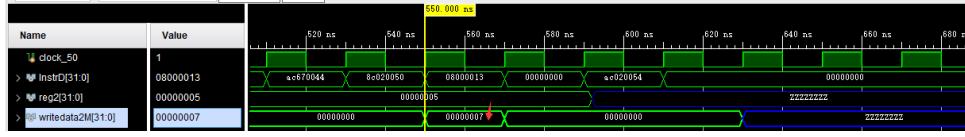


图 129

但 SW 指令 ac670044 对应的 M 阶段的 wridata2M 却是正确的。

想到这个信号是寄存器类型，直接被接到了数据存储器的写端口上，可能要连根线，但是连根线后还是不行。

最终经过仔细查找发现是 wridata2 小写而连入存储器端口的是大写 WriteData2。高阻态问题终于得到解决。

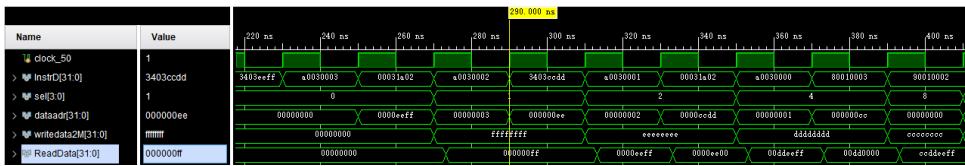


图 130

3.2.39 错误 39

(1) 错误现象: \$1 载入错误。

```
.set nomacro
.global _start
_start:
    ori $3,$0,0xeeff
    sb $3,0x3($0)      # [0x3] = 0xff          a0030003
    srl $3,$3,8
    sb $3,0x2($0)      # [0x2] = 0xee          a0030002
    ori $3,$0,0xccdd
    sb $3,0x1($0)      # [0x1] = 0xdd          a0030001
    srl $3,$3,8
    sb $3,0x0($0)      # [0x0] = 0xcc          a0030000
    lb $1,0x3($0)      # $1 = 0xffffffff        80010003
    lbu $1,0x2($0)     # $1 = 0x000000ee        90010002
    nop
```

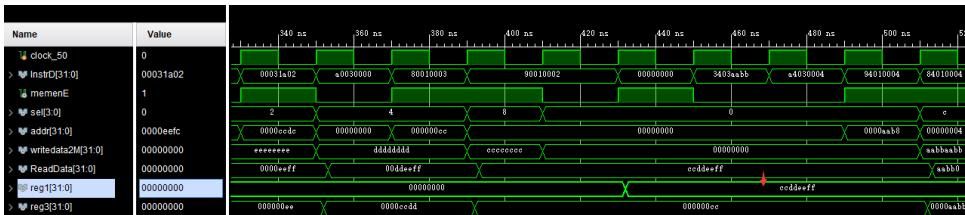


图 131

- (2) 分析定位过程: 原因大概是之前跑实验四测试的时候改过的 mips 的 ReadData 端口忘记改回来了。
将其修改后能正确载入。

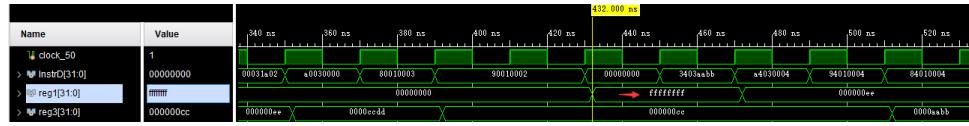


图 132

3.2.40 错误 40

- (1) 错误现象: cp0 测试指令集 mtc0 的 alucontrol 信号出错。

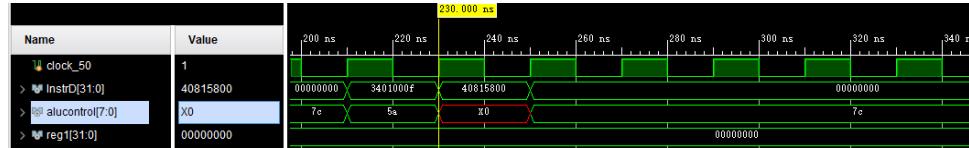


图 133

- (2) 分析定位过程: 检查 aludec 和宏定义没有发现问题,结合波形图发现 alu 执行正确,但是没有把结果传递到 M 阶段,应该是 flushM 出错,被误刷掉了。

继续查看波形图发现此时未定义指令异常信号不为全 0,而事实上不存在未定义指令,所以是 maindec 中没有更新 mtc0 的判断导致其被归类为 default 中的未定义指令。

修改 maindec 和 always 敏感块后仍然被判定为未定义指令。继续测试是否进入了 mtc0 的 case 分支,结果是已经进入,所以错误的来源是 excepttype 的信号产生有问题。

结合波形图发现 nop 指令被视作 invalid。检查 maindecoder 发现是没有译码移位指令。加上后,由于 nop 的 funct 字段和 sll 是一样的,因此会被当成 sll 来译码,而 nop 可以视作 0+0,一种特殊的加法,这样译码是正确的。

修改后不再触发 exception。但是 \$2 没写进去。

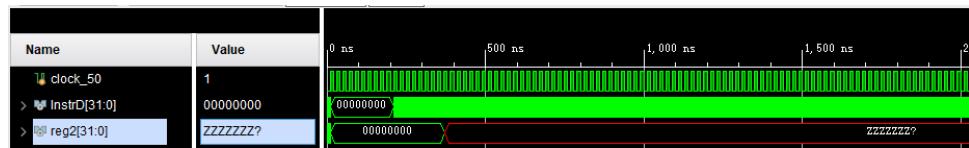


图 134

结合波形图,发现往 cp0 中写的数据有误,aluresultE 和 alucontrol 信号有问题。

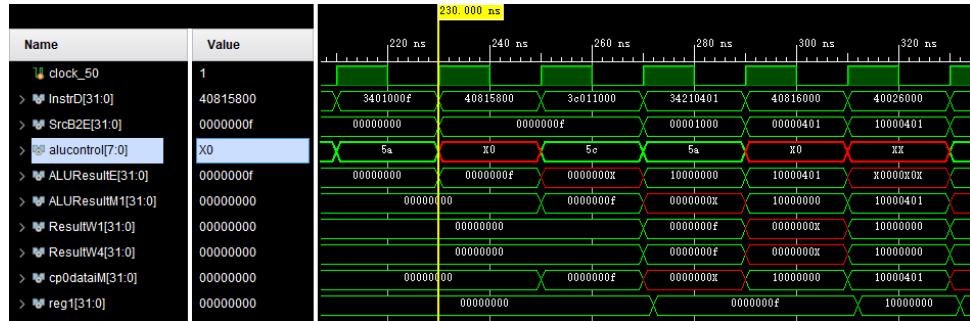


图 135

检查 aludec 发现 rsD 的位宽设置错误(5 位设成了 6 位),修改为 5 位位宽后 alucontrol 正确输出。

3.2.41 错误 41

(1) 错误现象: ResultW4 波形图中存在高阻态。

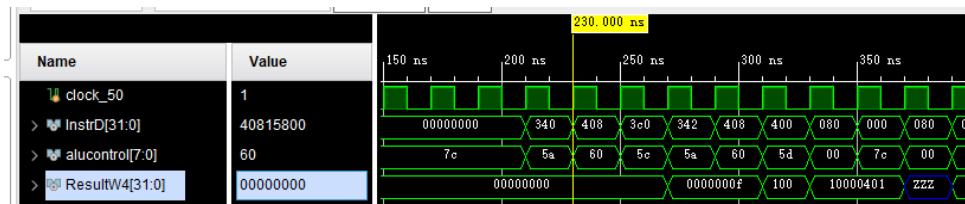


图 136

(2) 分析定位过程: 查看测试指令集发现存在关于 cp0status 寄存器的冒险。

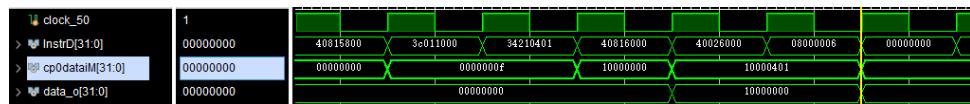


图 137

波形图中可以看到 cpodataM 是正确的,而和 ResultW4 相关的信号 cpodataW 出错为高阻态。

寻找 cpodataW 相关的两个信号 cpodataM 和 flushW,在波形图中都显示正确,最终发现是寄存器的位数出错导致。

修正后问题得到解决。

```
365 floprc #(1) r19W(clk,reset,flushW,cp0dataM,cp0dataW);
```

图 138

3.2.42 错误 42

(1) 错误现象: 测试第 12 个测试点时, stallD 信号始终为 1,而理论上应该只停顿一个周期。

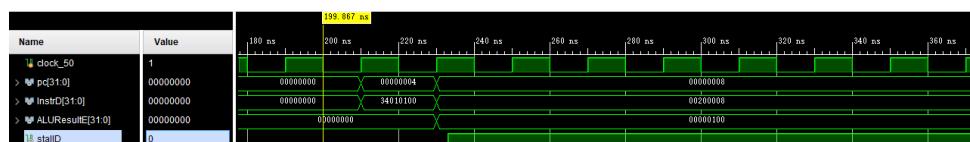


图 139

(2) 分析定位过程: 检查 stallID 的产生逻辑。波形图中显示 regwriteE 流走后没有被刷掉, 所以需要将无关信号刷掉。



图 140

因为 E 暂停了, 所以没能刷掉 E。

```
87 | assign #1 flushE = ((lwstallD | branchD & ~balD | jumpD & ~jalD) & ~stallE)|flush_except
```

图 141

而 E 暂停是因为 D 暂停了, 这逻辑不对, D 暂停没有理由暂停 E, 而是只需刷掉 E 就可以。

```
88 | assign #1 stallE = stall_divE|lwstallD|branchJrstallD;
```

图 142

去掉 D 暂停导致 E 暂停的逻辑。

```
88 | assign #1 stallE = stall_divE;//|lwstallD|branchJrstallD;
```

图 143

(3) 修正效果: 修正后 stallID 信号正常只停顿一个周期。

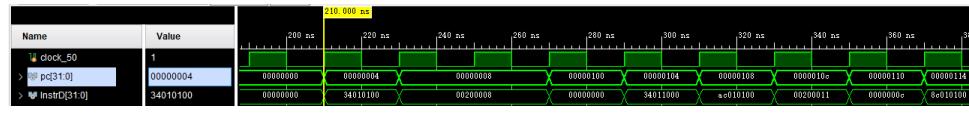


图 144

3.2.43 错误 43

(1) 错误现象: sram-soc 测试输出 pcF 一直为 0xbfc0038c。

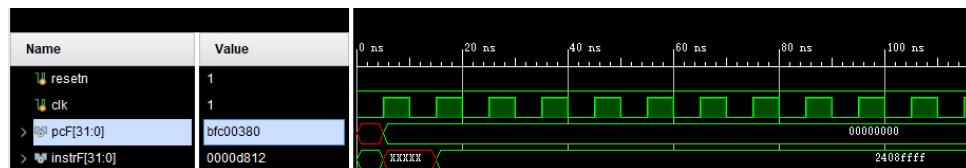
```

Test begin!
[ 22000 ns] Test is running, debug_wb_pc = 0xbfc0038c
[ 32000 ns] Test is running, debug_wb_pc = 0xbfc0038c
[ 42000 ns] Test is running, debug_wb_pc = 0xbfc0038c
[ 52000 ns] Test is running, debug_wb_pc = 0xbfc0038c
[ 62000 ns] Test is running, debug_wb_pc = 0xbfc0038c
[ 72000 ns] Test is running, debug_wb_pc = 0xbfc0038c
[ 82000 ns] Test is running, debug_wb_pc = 0xbfc0038c
[ 92000 ns] Test is running, debug_wb_pc = 0xbfc0038c
[102000 ns] Test is running, debug_wb_pc = 0xbfc0038c
[112000 ns] Test is running, debug_wb_pc = 0xbfc0038c
[122000 ns] Test is running, debug_wb_pc = 0xbfc0038c
[132000 ns] Test is running, debug_wb_pc = 0xbfc0038c
[142000 ns] Test is running, debug_wb_pc = 0xbfc0038c

```

图 145

- (2) 分析定位过程: 查看波形图中 pcW 也一直为 0。说明取出的指令有问题, 检查发现 ip 核没有进行更改。更改后读出了地址, 但 pc 还是不对。



Test begin!

```

[ 22000 ns] Test is running, debug_wb_pc = 0xbfc00384
[ 32000 ns] Test is running, debug_wb_pc = 0xbfc00388
[ 42000 ns] Test is running, debug_wb_pc = 0xbfc00380
[ 52000 ns] Test is running, debug_wb_pc = 0xbfc00384
[ 62000 ns] Test is running, debug_wb_pc = 0xbfc00388
[ 72000 ns] Test is running, debug_wb_pc = 0xbfc00380
[ 82000 ns] Test is running, debug_wb_pc = 0xbfc00384
[ 92000 ns] Test is running, debug_wb_pc = 0xbfc00388
[102000 ns] Test is running, debug_wb_pc = 0xbfc00380

```

图 146

查看波形图, pc 发生了改变, 但是此时取出的指令没有发生改变。

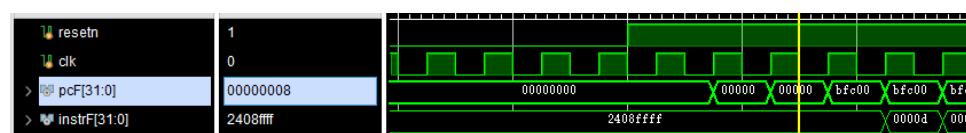


图 147

根据波形图中第一个周期的 instrF 为 xxx, 推测是出现了取指令的延迟。

尝试将时钟信号取反, 情况依然没有得到改善, 取值地址没有随着 pcF 变化。

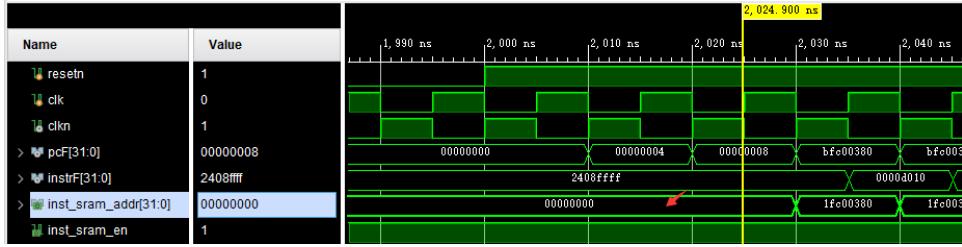


图 148

多次取指令失败, 推测是地址映射的逻辑有问题。

开始通过打表跳过这一错误, 但是后来还是遇到了很多次问题都与地址虚拟映射有关。请教了同学和助教后, 最终通过添加 mmu.v 模块进行虚拟地址映射后, 这一系列问题才最终得到解决。

- (3) 错误原因: 没有实现正确的虚拟地址映射逻辑。
- (4) 修正效果: 添加 mmu 模块进行虚拟地址映射后问题得到解决。
- (5) 归纳总结(可选): 虚拟地址映射是进行测试所必需的一个关键点, 关系到能否通过映射正确访存。

3.2.44 错误 44

- (1) 错误现象: 修正错误 43 虚拟地址异常后, 仿真 tcl 打印新的错误。

```
[ 2287 ns] Error!!!
reference: PC = 0x9fc00704, vb_rf_vnum = 0x1f, vb_rf_wdata = 0x9fc0070c
mycpu : PC = 0x9fc00704, vb_rf_vnum = 0x1f, vb_rf_wdata = 0xxxxxxxxX
$finish called at time : 2327 ns : File "E:/University/JuniorYear/First/Hardware/hardwaredatas_full_v0.04/test/func_test_v0.03/soc_sram_func/testbench/mycpu_tb.v" Line 160
```

图 149

- (2) 分析定位过程: 检查波形图发现 datapath 中的 resultW4 是高阻态, 根据 resultW4 的来源逐级上推发现 resultW1 不对, 而根据多路选择器, 这个值来源于 alurestultM2。

```
370 ?: alurestultm2mux(pcPlus8M,ALUResultM1,balM|jalM,ALUResultM2);
```

图 150

查找波形图发现 pcplus8D 信号为高阻态, 说明这个信号没有接上。

- (3) 修正效果: 在 datapath 中修正接口, 接上 pcplus8D 后问题解决。

3.2.45 错误 45

- (1) 错误现象: 仿真 tcl 打印新的错误。

```

Tcl Console x Messages Log
Q | H | S | I | D | B | F | M | C |
INFO: [USF-XSim-9] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:13 : elapsed = 00:00:21 . Memory (MB): peak = 943.648 : gain = 73.117
run all
Test begin!
[ 14057 ns] Error!!!
reference: PC = 0x9fc07d9c, wb_rf_wnum = 0x08, wb_rf_wdata = 0x01000001
mycpu : PC = 0xbfc00380, wb_rf_wnum = 0x1a, wb_rf_wdata = 0x00000000
$finish called at time : 14097 ns : File "E:/University/JuniorYear/First/Hardware/hardwaredatas_full_v0.04/test/func_test_v0.03/soc_sram_func/testbench/mycpu_tb.v" Line 160

```

图 151

- (2) 分析定位过程: 0380 是异常处理地址, 而根据查找报错的 pc 值对应的指令为一条 or 指令, 一条正常的 r 型指令却触发了异常跳转, 说明 or 的 decode 没有成功。

经过检查, 果然发现一开始做 decode 的时候, 在 maindecoder 中将 and, or, xor, nor 归在 default 中了, 而后来的 default 是用来处理 57 条指令之外的, 所以出现了判断跳转异常的情况。

- (3) 修正效果: 在 maindecoder 中添加上述 4 个指令的译码后该处错误解决。

3.2.46 错误 46

- (1) 错误现象: 仿真 tcl 打印新的错误。

```

[ 367777 ns] Error!!!
reference: PC = 0xbfc6570c, wb_rf_wnum = 0x02, wb_rf_wdata = 0x555b05c6
mycpu : PC = 0xbfc6570c, wb_rf_wnum = 0x02, wb_rf_wdata = 0x00000000

```

图 152

- (2) 分析定位过程: 报错的指令经过查找后为 add 指令, 查看波形图发现 add 的计算结果变成全 0。说明 add 的实现有问题。以为此处测试点还不涉及异常处理, 所以 add 理论上出错只有计算逻辑有误, 而不是溢出异常。

到 alu 中查看 add 实现, 发现之前在做某个测试的时候修改了 add 的逻辑使之在某种特定情况下输出全 0 来打表过点, 后来这个点过了其实与 add 无关, 而忘了把这个地方的打表逻辑删除。

- (3) 修正效果: 删除了这段特殊的打表逻辑后问题解决。

3.2.47 错误 47

- (1) 错误现象: 仿真 tcl 打印新的错误。

```

[ 922227 ns] Error!!!
reference: PC = 0xbfc1c57c, wb_rf_wnum = 0x04, wb_rf_wdata = 0xbfc1c5a8
mycpu : PC = 0xbfc1c5a0, wb_rf_wnum = 0x1f, wb_rf_wdata = 0xbfc1c5a8

```

图 153

- (2) 分析定位过程: 根据出错 pc 查找到出错指令是 bltzal, 结合波形图和反编译文件发现 bltzal 的上一条 ori 与之产生数据冒险, bltzal 应该暂停一个周期。

结合波形图, blt 在译码阶段停顿的时候 regwrite 信号被往后传了。

```

bfc1c5a0: 0510ffff6 bltzal      t0|bfc1c57c <n41_bltzal_test+0x1c>
assign #1 flushE = ((lwstallD | branchD & ~balD | jumpD & ~jalD) & ~stallE)|flush_except

```

图 154

flushE 没有刷掉,因为是 al 指令。经过思考,就算是 al 系列指令,指令停顿了,信号传到后面也没用,所以还是要刷。

(3) 修正效果: 修正 hazard 里 flushE 逻辑如下图所示。

```

87 assign #1 flushE = (([lwstallD | branchD & ~balD | jumpD & ~jalD | branchJrstallD] & ~stallE)

```

图 155

后面检查的时候发现除法也出现了写使能被传递下去的问题,添加如图所示的逻辑。

```

89 #1 flushM = flush_except | stallE;

```

图 156

3.2.48 错误 48

(1) 错误现象: 仿真 tcl 打印新的错误。

```

----[1018095 ns] Number 8'd42 Functional Test Point PASS!!!
[1022000 ns] Test is running, debug_wb_pc = 0xbfc6e61c
[1032000 ns] Test is running, debug_wb_pc = 0x00000000
[1042000 ns] Test is running, debug_wb_pc = 0xbfc6e7e4
[1052000 ns] Test is running, debug_wb_pc = 0xbfc6f8c8
[1062000 ns] Test is running, debug_wb_pc = 0xbfc709ac
----[1070265 ns] Number 8'd43 Functional Test Point PASS!!!

[1070997 ns] Error!!!
reference: PC = 0xbfc3ef10, wb_rf_wnum = 0x16, wb_rf_wdata = 0x15b8b7a4
mycpu   : PC = 0xbfc3ef10, wb_rf_wnum = 0x16, wb_rf_wdata = 0x00000000

```

图 157

(2) 分析定位过程: 查看对应的出错指令为 mfhi 指令。发现就在 mfhi 和 mflo 的前面有一个除法运算,存在 hi 的冒险。

查看波形图发现 hilo 的前推信号有一段红色,且 forwardHE 为高阻态。



图 158

经过仔细对比,发现通路中有一处 forward 信号的传递误将 forwardHE 写成了 forwardHW,导致 forwardHE 实质上没有在任何地方得到赋值。

(3) 修正效果: 修正后问题得到解决。

```
303    flopenrc #(1) r12E[clk,reset,~stallE,flushE,forwardHD,forwardHE]; /
```

图 159

3.2.49 错误 49

- (1) 错误现象: 仿真 tcl 报错。
- (2) 分析定位过程: 根据报错找到对应指令,发现是 lb 指令,取字节错误。一开始通过打表跳过这个错误,但是后来又再次遇到了卡在下一个 lb,仔细对比两个 lb 的错误发现都是读取的字节错位导致。联想到之前吕学长的视频貌似提到过大小端的问题,于是尝试将大小端逻辑进行修改。

```
case (dataout[11:8])
  2'b00:ReadData2<={{24{0}},ReadData[31:24]};
  2'b01:ReadData2<={{24{0}},ReadData[23:16]};
  2'b10:ReadData2<={{24{0}},ReadData[15:8]};
  2'b11:ReadData2<={{24{0}},ReadData[7:0]};
```

图 160

- (3) 修正效果: 修改大小端(调换读取字节位置逻辑)之后 lb 指令正确执行。在后面又遇到了 lh 的问题,于是对 lh 的大小端进行了修改。鉴于读取大小端有误,联想到写入的大小端也会错误,故对写入的也同时进行了更新,以防止后续写入指令的出错。
- (4) 归纳总结(可选): 接上 sram 之后读取字节的大小端会发生变化,导致原本在自己 cpu 上正确的读写逻辑会颠倒。

3.2.50 错误 50

- (1) 错误现象: tcl 控制台报错。

```

---[1070265 ns] Number 8'443 Functional Test Point PASS!!!
---[1071967 ns] Error!!!
    reference: PC = 0xbfc3ef7c, wb_rf_vnum = 0x15, wb_rf_rdata = 0x0000000a
    mycpu : PC = 0xbfc3ef7c, wb_rf_vnum = 0x15, wb_rf_rdata = 0xfffffffffb

[1072000 ns] Test is running. debug_wb_pc = 0xbfc3ef88
$finish called at time : 1072007 ns : File "E:/University/JuniorYear/First/Hardware/hardwaredatas_full_v0.04/test/func_test_v0.03/soc_sram_func/testbench/mycpu_tb.v" Line 160
) run: Time (s): cpu = 00:00:20 : elapsed = 00:00:16 . Memory (MB): peak = 1380.215 : gain = 0.000

```

图 161

- (2) 分析定位过程: 查看出错对应指令发现是除法后的 mfhl 冒险。查看波形图发现 forwardAE 信号在 E 暂停后会发生改变。

```

55      // forwardAE,forwardBE
56      always @(*) begin
57          forwardAE = 2'b00;
58          forwardBE = 2'b00;
59          if(rsE != 0) begin
60              if(rsE == writeregM & regwriteM) begin
61                  forwardAE = 2'b10;
62              end else if(rsE == writeregW & regwriteW) begin
63                  forwardAE = 2'b01;
64              end
65      end

```

图 162

修改 forwardAE 和 forwardBE 的生成后,该错误解决。

```

59 // forwardaE,forwardbE
60 always @(*) begin
61     if(rst)begin
62         forwardaE = 2'b00;
63         forwardbE = 2'b00;
64     end else if(~stallE)begin
65         forwardaE = 2'b00;
66         forwardbE = 2'b00;
67         if(rsE != 0) begin
68             if(rsE == writeregM & regwriteM) begin
69                 forwardaE = 2'b10;
70             end else if(rsE == writerew & regwriteW) begin
71                 forwardaE = 2'b01;
72             end
73         end
74         if(rtE != 0) begin
75             if(rtE == writeregM & regwriteM) begin
76                 forwardbE = 2'b10;
77             end else if(rtE == writerew & regwriteW) begin
78                 forwardbE = 2'b01;
79             end
80         end
81     end
82 end

```

图 163

3.2.51 错误 51

(1) 错误现象: tcl 控制台打印新的错误。

```

[ 14657 ns] Error!!!
reference : PC = 0xbfc00400, *b_rf_vnum = 0x1a, *b_rf_vdata = 0x00000020
mycpu   : PC = 0xbfc00678, *b_rf_vnum = 0x1a, *b_rf_vdata = 0x0000x120
) $finish called at time : 14697 ns : File "E:/University/JuniorYear/First/Hardware/hardwaredata_full_v0.04/test/func_t-test_v0.03/sec_sram_func/testbench/mycpu_tb.v" Line 160

```

图 164

(2) 分析定位过程: 定位指令发现是 bne 错误的跳转了,且 bne 和上一条 mfc0 存在关于 k0 的冒险。查看波形图发现 SrcA2D 的前推出现问题。bne 要读 k0 的时候,401a7000 这条 mfc0 正在读 cp0 到 k0,要把读出来的值旁路给 bne,需要停顿一周期,停顿一周期后,cp0 读出的值到 M 阶段,从 M 阶段旁路

回 D 阶段。而这个旁路在映像中好像并没有添加。

- (3) 修正效果: 加了从 M 阶段前推从 cp0 读出的数据到 D 的逻辑后问题得到解决。

```
261 mux2 #(32) forwardbranchM1mux(pcPlus8M,ALUResultM2,balM|jalM,branchForwardM1);//?
262 mux2 #(32) forwardbranchM2mux(cp0dataM,branchForwardM1,is_mfc0M,branchForwardM2);
263 mux4 #(32) forwardamux(SrcAD,pcPlus8E,branchForwardM2,ResultW4,forwardAD,SrcA2D);
264 mux4 #(32) forwardbmx(SrcBD,pcPlus8E,branchForwardM2,ResultW4,forwardBD,SrcB2D);
```

图 165

3.2.52 错误 52

- (1) 错误现象: tcl 打印新的错误信息。

```
[ 16047 ns] Error!!!
reference: PC = 0xbfc00380, wb_rf_wnum = 0x1a, wb_rf_wdata = 0x00000000
mycpu    : PC = 0xbfc04b74, wb_rf_wnum = 0x09, wb_rf_wdata = 0x02000000

[ 16065 ns] Error( 0)!!! Occurred in number 8' d02 Functional Test Point!
```

图 166

- (2) 分析定位过程: 找到这条出错的指令前面出错的地方,是一条 syscall,这条指令应该跳到异常处理程序首地址 bfc00380,这条指令应该跳到异常处理程序首地址 bfc00380,但波形图中显示 syscall 在译码阶段暂停了。查看上一条指令是 divu,所以引起了暂停,M 阶段的异常信号对,但下一条 pc 载入不正确,npc 正确,应该是这个时候恰好 D 阶段停顿,引起 F 停顿,导致 pc 没能跳转。这个问题寻找了很久的原因,一直没有找到逻辑上的问题,最后终于发现竟然是信号 excepttypeM 在 datapath 中没有声明的原因。

- (3) 修正效果: 添加声明后错误解决

- (4) 归纳总结(可选): 添加一个新的信号后一定要记得添加声明。

3.2.53 错误 53

- (1) 错误现象: tcl 控制台一直报 pc 为 0x00000000。

- (2) 分析定位过程: 查看出错的指令,发现是 lui 指令,而此时刚好偶然读出了 badvaddr 的值。波形图中 badvaddr 一直是 XXX 是因为没初始化,恰好读出它的时候就会是 XXX。

一开始怀疑是出现了环路,经过一次综合修正了部分位数没对齐的选择器之后问题依然没解决,说明不是环路的问题。继续检查波形图发现 adelM 始终为 1。如果时 pc 没对齐引发的 adel,pc 变成后 bfc00380 后,excepttype 就会归零。这里是 adel 忘记归零导致的问题,修改后依然一直打印 0x00000000。

怀疑是 (case(ALUControlM)) 中没有加 default,这样指令变过后,找不到能进入的分支,就什么都不做,寄存器 adel 里的值就保持不变。

- (3) 修正效果: 在 default 中添加 adelM 和 adesM 的归零后问题解决。
- (4) 归纳总结(可选): 这属于编写程序不规范, 写分支时考虑不周全。如果遵守规范, 老老实实写 default 分支, 就可以规避这种情况。

3.2.54 错误 54

- (1) 错误现象: tcl 控制台打印新的错误。

```

[ 78415 ns] Number 8'd11 Functional Test Point PASS!!!
[ 82000 ns] Test is running, debug_wb_pc = 0xbfc0038c
[ 85695 ns] Number 8'd12 Functional Test Point PASS!!!
[ 92000 ns] Test is running, debug_wb_pc = 0xbfc00594
[ 92375 ns] Number 8'd13 Functional Test Point PASS!!!

[ 93297 ns] Error!!!
reference: PC = 0xbfc005d0, wb_rf_vnum = 0x1a, wb_rf_vdata = 0xbfc03d14
mycpu : PC = 0xbfc005d0, wb_rf_vnum = 0x1a, wb_rf_vdata = 0x00000000

```

图 167

- (2) 分析定位过程: 查看出错指令为 mfc0, 对比波形图发现 epc 的值有问题, 可能是写的时候就出了问题。向前寻找这个写的来源, 发现触发了软中断, 查看波形图发现延迟槽没有被正确传递下去, 经过检查发现是控制条件 stallID 写成了 stall。
- 修正后延迟槽正常传递, 但错误依然存在。对比波形图发现 pcE 出错, 跳转指令把执行阶段的信号刷掉了。pc 信号比较特殊, 只在停顿的时候刷掉就可以, 跳转的时候可以不刷。

- (3) 修正效果: 加一个单独刷 pc 的信号。

```

64  wire stallE,stallF1,annul_iE,flushpcE;//,forwardBD;//,forwardBD1;//stallD,stallF
97  assign #1 flushE = ((lwstallD | branchD & ~balD | jumpD & ~jalD | branchJrstallD) & ~stallE) | flush_except;// for pc
98  ->assign #1 flushpcE = ((lwstallD | branchJrstallD) & ~stallE) | flush_except;// for pc

```

图 168

修正后问题解决。

3.2.55 错误 55

- (1) 错误现象: 添加 axi 接口后出现 pc 循环报 0x00000000。

```

Tcl Console  x Messages  Log
Q  |  M  |  D  |  II  |  R  |  B  |  C  |

[ 272000 ns] Test is running, debug_wb_pc = 0x00000000
[ 282000 ns] Test is running, debug_wb_pc = 0x00000000
[ 292000 ns] Test is running, debug_wb_pc = 0x00000000
[ 302000 ns] Test is running, debug_wb_pc = 0x00000000
[ 312000 ns] Test is running, debug_wb_pc = 0x00000000
[ 322000 ns] Test is running, debug_wb_pc = 0x00000000
[ 332000 ns] Test is running, debug_wb_pc = 0x00000000
[ 342000 ns] Test is running, debug_wb_pc = 0x00000000
[ 352000 ns] Test is running, debug_wb_pc = 0x00000000
[ 362000 ns] Test is running, debug_wb_pc = 0x00000000
run: Time (s): cpu = 00:00:12 ; elapsed = 00:00:09 . Memory (MB): peak = 875.168 ; gain = 0.000

```

图 169

(2) 分析定位过程: 查找后发现 clk 的信号名发生了变化, 将其修改成正确的 clkn。

查看波形图发现很多信号都是蓝红线, 发现很多信号在大小写上出现了错误, 还有很多信号没有成功连上。

(3) 修正效果: 将所有有问题的连线和端口接上修正后, 不再循环报 pc=0。

3.2.56 错误 56

(1) 错误现象: tcl 控制台打印新错误。

```
[ 2483 ns] Error!!!
reference: PC = 0xbfc00004, wb_rf_wnum = 0x08, wb_rf_wdata = 0xffffffff
mycpu : PC = 0xbfc00000, wb_rf_wnum = 0x08, wb_rf_wdata = 0xffffffff

$finish called at time : 2522500 ps : File "E:/University/JuniorYear/First/Hardware/hardwaredata_full_v0_04/test/func_test_v0_03/soc_axi_func/testbench/mycpu_tb.v" Line 168
```

图 170

(2) 分析定位过程: 对比波形图发现 regwrite 被传到了 W 阶段, 流水线暂停出了问题。初始化时流水线各级都是空指令 nop, 这个指令因为被当作 R-type0+0 处理, 所以会译码处 regwrite=1, 初始时 D 阶段的这个信号会在停顿信号出现前已经传到后面。尝试修改 nop 的译码, 不让它写寄存器。

查看 maindecoder 发现之前把 nop 归类的位置出错。修正了之后问题没有完全得到解决。

参考吕学长在视频中提到的译码阶段暂停, 发现本来理解的是不让 F 阶段的信号流到 D 阶段, 就是让译码阶段暂停了; 但实际上应该不让 D 阶段的信号留到 E 阶段, 即取指暂停时, 上一条指令应停在译码阶段。因此修改 stallE 逻辑如下图所示。

```
99 assign #1 stallE = stall_divE|dstall|istall;//|lwstallD|branchJrstallD; // new, add |lws
```

图 171

(3) 修正效果: 将 nop 的译码单列, 修正 stallE 逻辑后问题解决。

3.2.57 错误 57

(1) 错误现象: tcl 打印新的错误。

```
Test begin!

[ 4144 ns] Error!!!
reference: PC = 0xbfc006bc, wb_rf_wnum = 0x04, wb_rf_wdata = 0xbfaaff008
mycpu : PC = 0xbfc006bc, wb_rf_wnum = 0x04, wb_rf_wdata = 0x0000f008

$finish called at time : 4183500 ps : File "E:/University/JuniorYear/First/Hardware/ha
```

图 172

(2) 分析定位过程: 查看出错指令发现是 ori 指令写回的值不对。查看波形图发现 alurestM 在 M 阶段被刷掉了。尝试把 flushM 的 stallE 改成了 stallM, 又回到了之前时序对不上的问题。

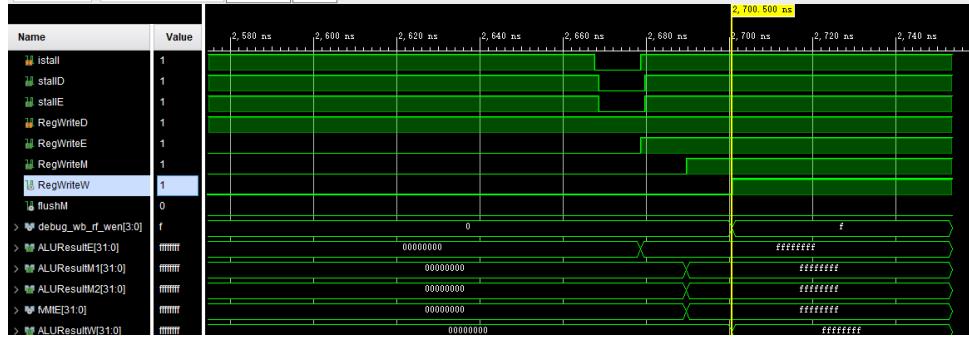


图 173

如上图所示,往 E 阶段放进来一条指令后,它流到 W,W 信号没有被冲刷掉。它流走后,现在指令并不是每一条紧跟着上一条在流水线中流动,而是要隔几个周期才会有一条指令流经流水线。这种情况下,每当有一条指令流过,而接下来没有紧跟着它的指令时,指令流走后的流水级就应该清空。因此只要流水级没有停顿,且下一周期没有指令要流进来,流水级就应清空。于是问题就变成,如何判断某一流水级中指令流走后下一周期有没有指令要流进来。没有指令要流进来,说明前面流水级暂停了。向前找到最后一级暂停的流水级,可以根据暂停信号清空暂停流水级的下一级,这样这个清空的效果就会随流水线一直流下去。所以下图所示位置应该在 D 暂停后清空 E,而更改 D 暂停逻辑时忘记同步到 flushE 中了。

```
97 assign #1 flushE = ((lwstallD | branchD & ~balD | jumpD & ~jalD | branchJrstallD |
98   | istall | dstall) & ~stallE)|flush_except;// |jumpD added // comm
```

图 174

之前对停顿信号的理解有偏差,当某一级停顿时,就是说这一级的数据就停在这一级不动了。之前理解成了某一级停顿就是禁止前一级向这一级传递信号,这是片面的,只是这一级停顿的一个必然要求。事实上一个流水级停顿了,就要求停顿级的信号不能被改变,所以前一级不能再传信号给它,而且停顿级的信号也不能再向下流动,于是下一级被刷新。

按照以上思路完善所有 stall 和 flush 信号的逻辑,为了思路清晰,引入了部分冗余信号。如下图所示。

```
93 assign #1 flushF = flush_except;
94 assign #1 stallD = lwstallD | stall_divE | branchJrstallD | istall | dstall;// ; comment br
95 assign #1 flushD = flush_except|((eretD|stallF)&~stallD); //flush delay slot after eret//(pc
96 assign #1 stallF = stallD | istall | dstall;
97 assign #1 flushE = ((lwstallD | branchD & ~balD | jumpD & ~jalD | branchJrstallD |
98   | istall | dstall) & ~stallE)|flush_except;// |jumpD added // comment
99 assign #1 flushpcE = ((lwstallD | branchJrstallD | istall | dstall) & ~stallE) | flush_except;
100 assign #1 stallE = stall_divE|dstall|istall;//|lwstallD|branchJrstallD; // new, add |lwstall
101 assign #1 flushM = flush_except | (stallE&~stallM); // add stallE for regwrite not flow// ch
102 assign #1 stallM = stall_divE|dstall;
103 assign #1 stallW = dstall;// axi new
104 assign #1 flushW = flush_except | (stallM&~stallW);
```

图 175

(3) 修正效果: 修正后该处错误解决。

3.2.58 错误 58

(1) 错误现象: tcl 控制台打印新的错误。

```
-----[32204335 ns] Number 8'd64 Functional Test Point PASS!!!
[32212000 ns] Test is running, debug_wb_pc = 0x00000000
-----
[32214718 ns] Error!!!
    reference: PC = 0xbfc00380, wb_rf_wnum = 0x1a, wb_rf_wdata = 0x00004000
    mycpu    : PC = 0xbfc00384, wb_rf_wnum = 0x1b, wb_rf_wdata = 0x00000000
-----
```

图 176

(2) 分析定位过程: 查看波形图,bfc00380 处取出来的地址不对,变成了全 0。如下图所示,原因是请求异常地址处的指令时,还没有读出上一条指令,指令 sram 繁忙,无视了 cpu 的请求。为了简化设计,不中断 sram 当前的工作,而是礼貌地等它读完这条无用的指令并忽视掉它返回的指令,然后再发起读请求。

拟添加一个状态机,其实就是一个排队信号,让中断地址排队等 isram 空闲,当 isram 读出当前指令后,如果有异常处理指令正在排队,则不把 isram 的状态变化通知给 cpu,让 cpu 继续保持原状。cpu 发出读异常处理程序指令的请求时,如果 isram 正在读,就让这次读完的信号不传给 cpu,接着读异常处理指令,等异常处理指令读完后再向 cpu 发送读完信号。

状态机逻辑添加如下图所示。

```
90  always@(rst,i_stall,posedge except)begin
91      if(rst)begin
92          exwa<=2'b00;
93          //exec<=1'b0;
94      end else begin
95          if(i_stall)begin
96              if(except)begin
97                  exwa<=2'b10;
98              end
99          end else begin
100             if(exwa==2'b10)begin
101                 exwa<=2'b01;
102             end else begin
103                 exwa<=2'b00;
104             end
105         end
106     end
107 end
```

图 177

修改后运行仿真再一次进入 pc 全 0 的问题。

继续查看波形图寻找存在的问题,发现等前面的指令读完后没有发送读异常指令的请求。发现 stallF 始终为 1,导致 pc 因为 stall 没有被正确传递下去。

发生例外的这条指令似乎没有暂停,查看前面几条指令发现设置了软中断,携带软中断的指令被暂停在 D,于是它的地址传不到 M,前面几条指令又设置好了软中断,所以 M 写到 epc 的地址是 0。如果成功触发软中断,那么要写到 etc 的指令地址是最后一条设置软中断的指令地址 +4,而最后一条设置软中断的指令地址是可以得到的,它就是触发中断的 M 阶段时的 pcW。

添加选择器如下图所示。

```
212 assign soft_pcM=(excepttypeM==32'h00000001)?pcW+4:pcM;
```

图 178

(3) 修正效果: axi 接口所有功能测试均通过。

4 设计结果

4.1 设计交付物说明

提交的目录最外层为学号姓名_myCPU,内部包含两个文件夹和一个 Readme.txt。Perf_test 文件夹中是最最终版本的 myCPU 代码,通过了 axi 十个性能测试。Others 文件夹中是先前版本的代码,分为三个文件夹,sram89 中是通过了 89 个测试点的 sram 接口版的 myCPU,axi89 中是通过了 89 个测试点的加了 axi 转接桥版的 myCPU,cache 中是加了写透 cache 卡在第 77 个测试点的版本的 myCPU。要进行仿真、综合,只需选择要测试的 myCPU 文件夹替换掉测试工程中的 myCPU 文件夹,并确保项目中载入了该文件夹,运行仿真或综合即可。不同 tb 文件可能存在接口中外部中断信号名称不一致的问题,存在 ext_int 和 int 两种写法,可能需要进行相应修改,同时修改该信号的连线。

4.2 设计演示结果

指令集测试首先采用了下发材料中的 12 个测试点测试各功能点的正确性。

1.Ori_Inst_Test	2017/12/16 17:54	文件夹
2.Logic_Inst_Test	2017/12/16 17:54	文件夹
3.Shift_Inst_Test	2017/12/16 17:54	文件夹
4.Move_Inst_Test	2017/12/16 17:54	文件夹
5.Simple_Arithmatic_Inst_Test	2017/12/16 17:54	文件夹
6.Div_Inst_Test	2017/12/16 17:54	文件夹
7.J_Inst_Test	2017/12/16 17:54	文件夹
8.B_Inst_Test	2017/12/16 17:54	文件夹
9.Memory_Inst_Test	2017/12/16 17:54	文件夹
10.M_Inst_Hazard_Test	2017/12/16 17:54	文件夹
11.CP0_Inst_Test	2017/12/16 17:54	文件夹
12.Exception_Inst_Test	2017/12/16 17:54	文件夹

图 179: 12 个测试点

1. 第一个测试点:对 ori 指令测试通过(测试指令如下)。图中显示信号的值与预期相同。

```
_start:
    ori $1,$0,0x1100      # $1 = $0 | 0x1100 = 0x1100
    ori $2,$0,0x0020      # $2 = $0 | 0x0020 = 0x0020
    ori $3,$0,0xffff00    # $3 = $0 | 0xffff00 = 0xffff00
    ori $4,$0,0xffff      # $4 = $0 | 0xffff = 0xffff
```

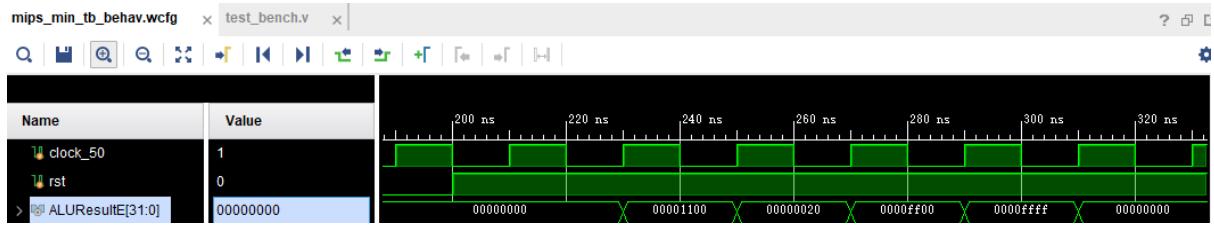


图 180: ori 指令测试通过

2. 第二个测试点:对 logic 指令集测试通过(测试指令如下)。图中显示的信号的值与预期相同。

```
_start:
    lui   $1,0x0101
    ori   $1,$1,0x0101
    ori   $2,$1,0x1100      # $2 = $1 | 0x1100 = 0x01011101
    or    $1,$1,$2          # $1 = $1 | $2 = 0x01011101
    andi $3,$1,0x00fe      # $3 = $1 & 0x00fe = 0x00000000
    and   $1,$3,$1          # $1 = $3 & $1 = 0x00000000
    xori $4,$1,0xff00      # $4 = $1 ^ 0xff00 = 0x0000ff00
    xor   $1,$4,$1          # $1 = $4 ^ $1 = 0x0000ff00
    nor   $1,$4,$1          # $1 = $4 ^ $1 = 0xfffff00ff    nor is "notUor"
```

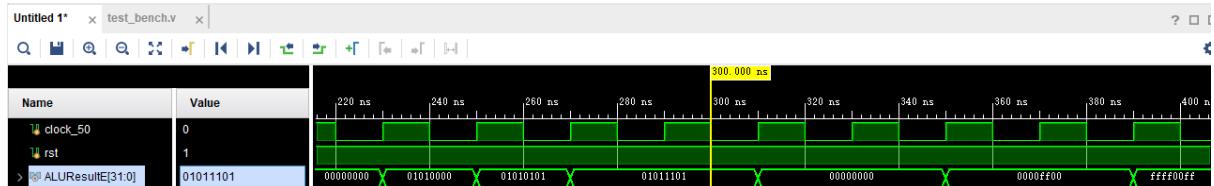


图 181: logic 指令集测试通过

3. 第三个测试点:对 shift 移位指令集测试通过(测试指令如下)。图中显示的信号的值与预期相同。

```
_start:
    lui   $2,0x0404
    ori   $2,$2,0x0404
    ori   $7,$0,0x7
    ori   $5,$0,0x5
    ori   $8,$0,0x8
    sync
    sll   $2,$2,8      # $2 = 0x40404040  sll 8 = 0x04040400
    sllv  $2,$2,$7      # $2 = 0x04040400  sll 7 = 0x02020000
    srl   $2,$2,8      # $2 = 0x02020000  srl 8 = 0x00020200
    srlv  $2,$2,$5      # $2 = 0x00020200  srl 5 = 0x00001010
    nop
    sll   $2,$2,19     # $2 = 0x00001010  sll 19 = 0x80800000
    ssnop
    sra   $2,$2,16     # $2 = 0x80800000  sra 16 = 0xffff8080
    srav  $2,$2,$8      # $2 = 0xffff8080  sra 8 = 0xfffffff80
```

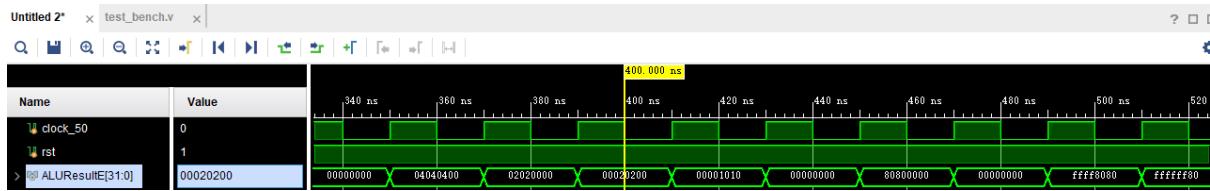


图 182: shift 移位指令集测试通过

4. 第四个测试点:对 hilo 指令集测试通过(测试指令如下)。图中显示的 hilo 寄存器中保存的值与预期相同。

```
_start:
    lui $1,0x0000      # $1 = 0x00000000
    lui $2,0xffff       # $2 = 0xffff0000
    lui $3,0x0505       # $3 = 0x05050000
    lui $4,0x0000       # $4 = 0x00000000

    mthi $0             # hi = 0x00000000
    mthi $2             # hi = 0xffff0000
    mthi $3             # hi = 0x05050000
    mfhi $4             # $4 = 0x05050000

    mtlo $3             # li = 0x05050000
    mtlo $2             # li = 0xffff0000
    mtlo $1             # li = 0x00000000
    mflo $4             # $4 = 0x00000000
```

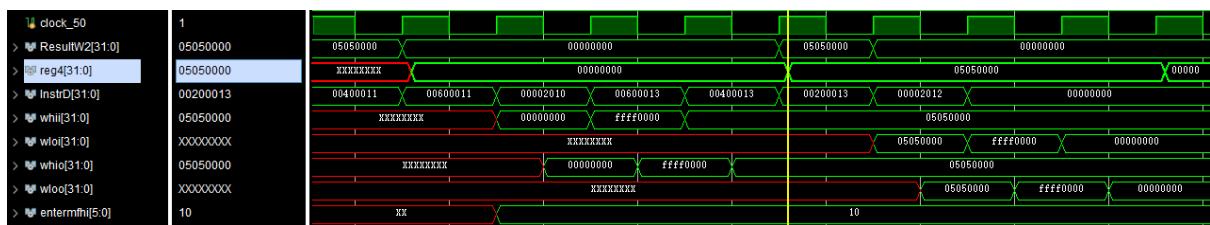


图 183: hilo 指令集测试通过

5. 第五个测试点:对 arithmetic 指令集测试通过(除法除外,测试指令如下)。通过截图丢失了,但由于后续已通过 soc 功能测试,故此处不再置成果图。

```
_start:
    ori $1,$0,0x8000      # $1 = 0x8000
    sll $1,$1,16           # $1 = 0x80000000
    ori $1,$1,0x0010       # $1 = 0x80000010

    ori $2,$0,0x8000      # $2 = 0x8000
    sll $2,$2,16           # $2 = 0x80000000
    ori $2,$2,0x0001       # $2 = 0x80000001

    ori $3,$0,0x0000      # $3 = 0x00000000
    addu $3,$2,$1          # $3 = 0x00000011
    ori $3,$0,0x0000       # $3 = 0x00000000
    add $3,$2,$1           # overflow ,$3 keep 0x00000000

    sub $3,$1,$3           # $3 = 0x80000010
    subu $3,$3,$2          # $3 = 0xF

    addi $3,$3,2            # $3 = 0x11
    ori $3,$0,0x0000       # $3 = 0x00000000
    addiu $3,$3,0x8000     # $3 = 0xffff8000

    or $1,$0,0xffff        # $1 = 0xffff
    sll $1,$1,16           # $1 = 0xffff0000
```

```

    slt  $2,$1,$0          # $2 = 1
    sltu $2,$1,$0           # $2 = 0
    slti $2,$1,0x8000       # $2 = 1
    sltiu $2,$1,0x8000      # $2 = 1

    ori   $1,$0,0xfffff
    sll   $1,$1,16
    ori   $1,$1,0xffffb      # $1 = -5
    ori   $2,$0,6             # $2 = 6
    mult  $1,$2               # hi = 0xffffffff
                                # lo = 0xffffffe2

    multu $1,$2              # hi = 0x5
                                # lo = 0xffffffe2
    nop
    nop

```

6. 第六个测试点:对除法指令集测试通过(测试指令如下)。通过截图丢失了,但由于后续已通过soc功能测试,故此处不再置成果图。

```

_start:
    ori   $2,$0,0xfffff
    sll   $2,$2,16
    ori   $2,$2,0xffff1      # $2 = -15
    ori   $3,$0,0x11          # $3 = 17

    div   $zero,$2,$3          # hi = 0xffffffff1
                                # lo = 0x0
    divu  $zero,$2,$3          # hi = 0x00000003
                                # lo = 0x0f0f0f0e

    div   $zero,$3,$2          # hi = 2
                                # lo = 0xffffffff

```

7. 第七个测试点:j类指令集测试通过(测试指令如下)。图中显示的信号的值与预期相同。

```

_start:
    ori   $1,$0,0x0001      # $1 = 0x1
    j     0x20
    ori   $1,$0,0x0002      # $1 = 0x2
    ori   $1,$0,0x1111
    ori   $1,$0,0x1100

    .org 0x20
    ori   $1,$0,0x0003      # $1 = 0x3
    jal   0x40
    div   $zero,$31,$1        # $31 = 0x2c, $1 = 0x3
                                # HI = 0x2, LO = 0xe
    ori   $1,$0,0x0005      # r1 = 0x5
    ori   $1,$0,0x0006      # r1 = 0x6
    j     0x60
    nop

    .org 0x40

    jalr $2,$31
    or   $1,$2,$0            # $1 = 0x48
    ori   $1,$0,0x0009      # $1 = 0x9
    ori   $1,$0,0x000a      # $1 = 0xa
    j   0x80
    nop

    .org 0x60
    ori   $1,$0,0x0007      # $1 = 0x7
    jr   $2
    ori   $1,$0,0x0008      # $1 = 0x8
    ori   $1,$0,0x1111
    ori   $1,$0,0x1100

    .org 0x80

```

```

nop

loop:
    j _loop
    nop

```



图 184: j 类指令集测试通过

8. 第八个测试点:b类指令集测试通过(测试指令如下)。图中显示的信号的值与预期相同。

```

_start:
    ori $3,$0,0x8000
    sll $3,16          # $3 = 0x80000000
    ori $1,$0,0x0001  # $1 = 0x1
    b    s1
    ori $1,$0,0x0002  # $1 = 0x2
1:
    ori $1,$0,0x1111
    ori $1,$0,0x1100

    .org 0x20
s1:
    ori $1,$0,0x0003  # $1 = 0x3
    bal s2
    div $zero,$31,$1  # $31 = 0x2c, $1 = 0x3
                           # HI = 0x2, LO = 0xe
    ori $1,$0,0x1100

```

```

ori $1,$0,0x1111
bne $1,$0,s3
nop
ori $1,$0,0x1100
ori $1,$0,0x1111

.org 0x50
s2:
ori $1,$0,0x0004      # $1 = 0x4
beq $3,$3,s3
or $1,$31,$0          # $1 = 0x2c
ori $1,$0,0x1111
ori $1,$0,0x1100

2:
ori $1,$0,0x0007      # $1 = 0x7
ori $1,$0,0x0008      # $1 = 0x8
bgtz $1,s4
ori $1,$0,0x0009      # $1 = 0x9
ori $1,$0,0x1111
ori $1,$0,0x1100

.org 0x80
s3:
ori $1,$0,0x0005      # $1 = 0x5
BGEZ $1,2b
ori $1,$0,0x0006      # $1 = 0x6
ori $1,$0,0x1111
ori $1,$0,0x1100

.org 0x100
s4:
ori $1,$0,0x000a      # $1 = 0xa
BGEZAL $3,s3
or $1,$0,$31            # $1 = 0x10c
ori $1,$0,0x000b      # $1 = 0xb
ori $1,$0,0x000c      # $1 = 0xc
ori $1,$0,0x000d      # $1 = 0xd
ori $1,$0,0x000e      # $1 = 0xe
bltz $3,s5
ori $1,$0,0x000f      # $1 = 0xf
ori $1,$0,0x1100

.org 0x130
s5:
ori $1,$0,0x0010      # $1 = 0x10
blez $1,2b
ori $1,$0,0x0011      # $1 = 0x11
ori $1,$0,0x0012      # $1 = 0x12
ori $1,$0,0x0013      # $1 = 0x13
bltzal $3,s6
or $1,$0,$31            # $1 = 0x14c
ori $1,$0,0x1100

.org 0x160
s6:
ori $1,$0,0x0014      # $1 = 0x14
nop

_loop:
j _loop
nop

```

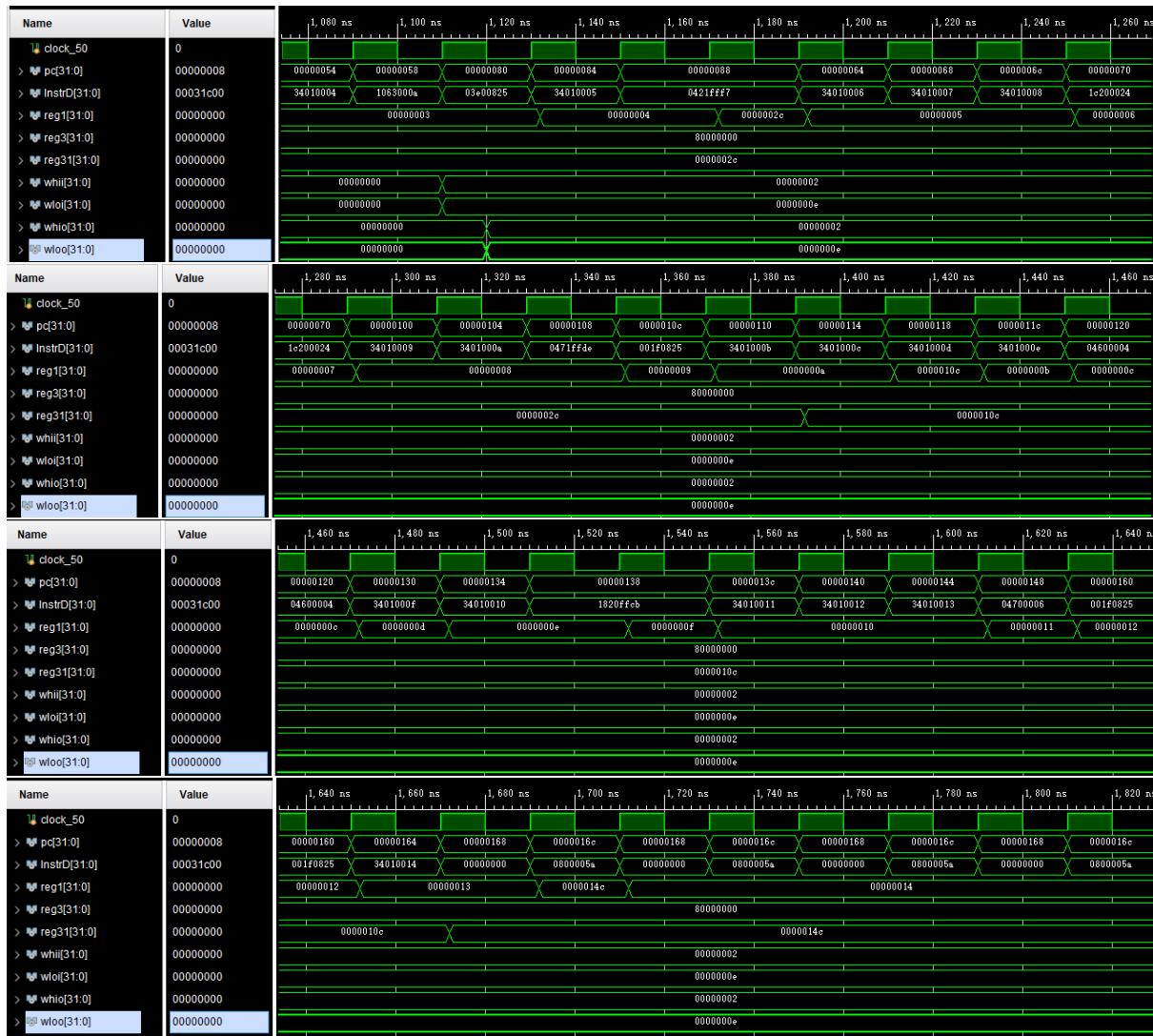


图 185: b 类指令集测试通过

9. 第九个测试点:memory 指令集测试通过(测试指令如下)。图中显示的信号的值与预期相同。

```

_start:
    ori $3,$0,0xeeff
    sb $3,0x3($0)      # [0x3] = 0xff
    srl $3,$3,8
    sb $3,0x2($0)      # [0x2] = 0xee
    ori $3,$0,0xccdd
    sb $3,0x1($0)      # [0x1] = 0xdd
    srl $3,$3,8
    sb $3,0x0($0)      # [0x0] = 0xcc
    lb $1,0x3($0)
    lbu $1,0x2($0)     # $1 = 0x000000ee
    nop

    ori $3,$0,0xaabb
    sh $3,0x4($0)      # [0x4] = 0xaa, [0x5] = 0xbb
    lhu $1,0x4($0)     # $1 = 0x0000aabb
    lh $1,0x4($0)      # $1 = 0xffffaabb

    ori $3,$0,0x8899
    sh $3,0x6($0)      # [0x6] = 0x88, [0x7] = 0x99
    lh $1,0x6($0)      # $1 = 0xffff8899
    lhu $1,0x6($0)     # $1 = 0x00008899

    ori $3,$0,0x4455

```

```

s11 $3,$3,0x10
ori $3,$3,0x6677
sw $3,0x8($0)      # [0x8] = 0x44, [0x9]= 0x55, [0xa]= 0x66, [0xb] = 0x77
lw $1,0x8($0)      # $1 = 0x44556677

nop

loop:
j _loop
nop

```

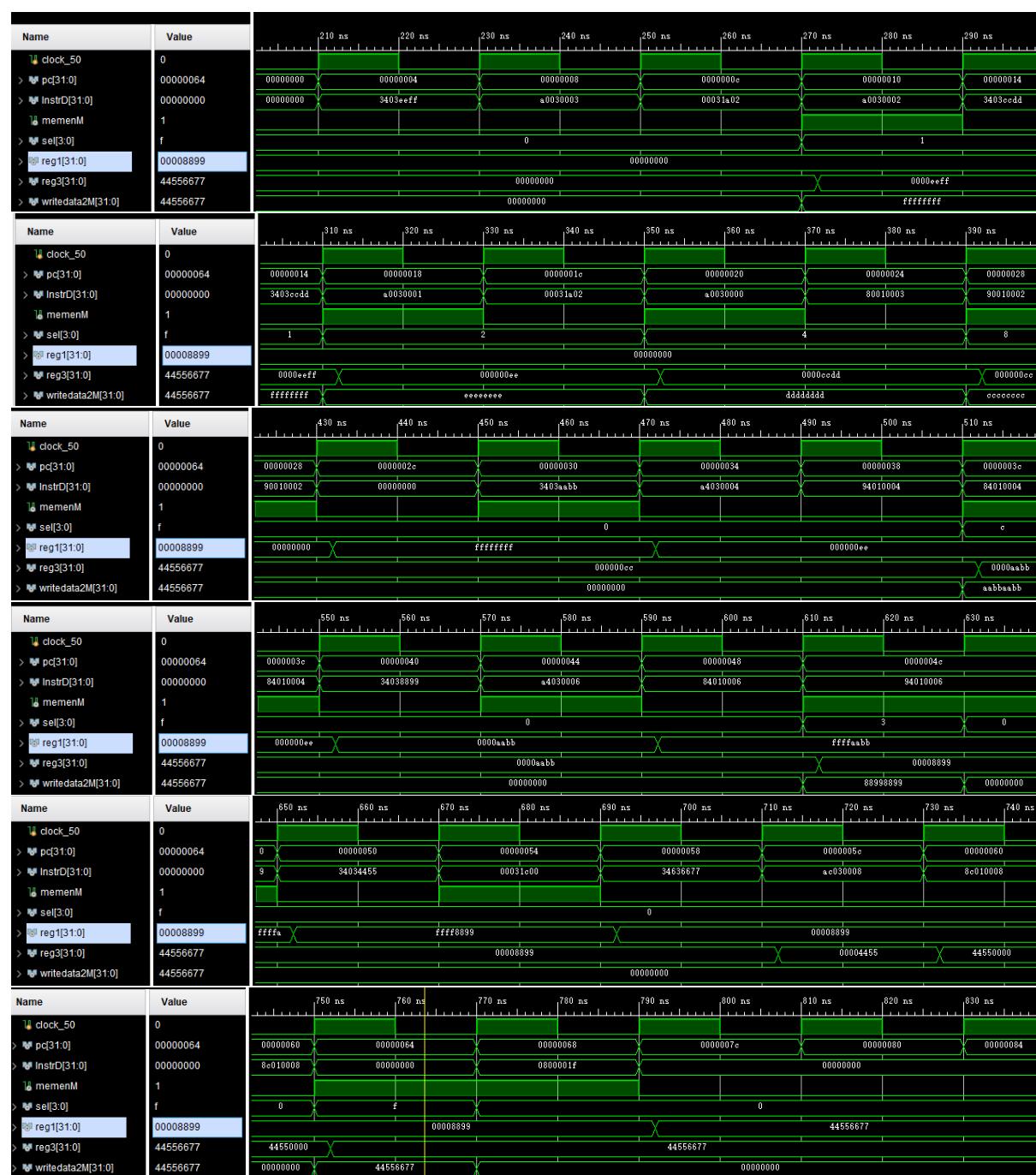


图 186: memory 指令集测试通过

10. 第十个测试点:hazard 指令集测试通过(测试指令如下)。图中显示的信号的值与预期相同。

_start:

```

ori $1,$0,0x1234      # $1 = 0x00001234
sw  $1,0x0($0)        # [0x0] = 0x00001234

ori $2,$0,0x1234      # $2 = 0x00001234
ori $1,$0,0x0          # $1 = 0x0
lw   $1,0x0($0)        # $1 = 0x00001234
beq $1,$2,Label
nop

ori $1,$0,0x4567
nop

Label:
ori $1,$0,0x89ab      # $1 = 0x000089ab
nop

_loop:
j _loop
nop

```

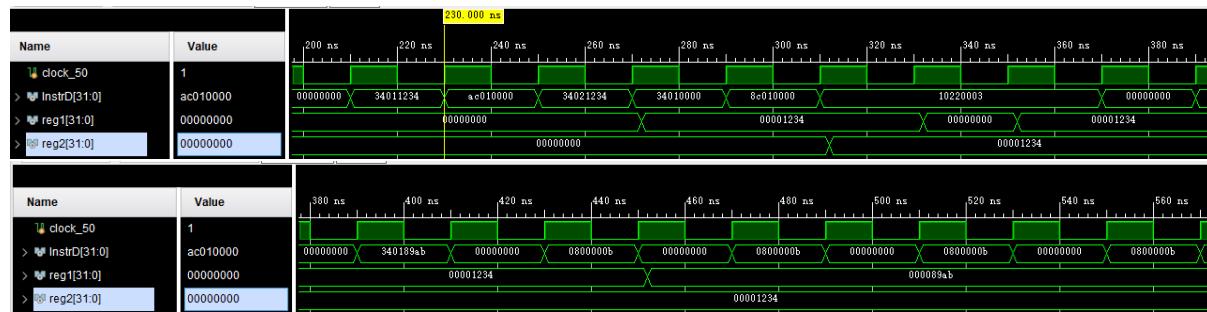


图 187: hazard 指令集测试通过

11. 第十一个测试点:cp0 指令集测试通过(测试指令如下)。图中显示的信号的值与预期相同。

```

_start:
ori $1,$0,0xf
mtc0 $1,$11,0x0 #写 compare 寄存器, 开始计时
lui $1,0x1000
ori $1,$1,0x401
mtc0 $1,$12,0x0 #将 0x401 写入 status 寄存器
mfco $2,$12,0x0 #读 status 寄存器, $2=0x401

_loop:
j _loop
nop

```

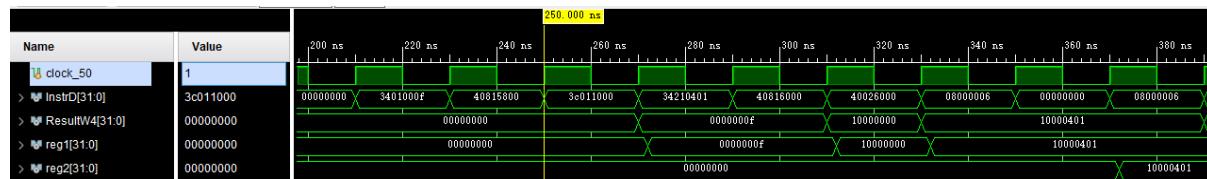


图 188: cp0 指令集测试通过

12. 第十二个测试点:exception 指令集测试通过(测试指令如下)。图中显示的信号的值与预期相同。

```

_start:
ori $1,$0,0x100      # $1 = 0x100
jr $1
nop

.org 0x40
ori $1,$0,0x8000      # $1 = 0x00008000

```

```

ori $1,$0,0x9000      # $1 = 0x00009000
mfc0 $1,$14,0x0       # $1 = 0x00000010c
addi $1,$1,0x4         # $1 = 0x000000110
mtc0 $1,$14,0x0
eret
nop

.org 0x100
ori $1,$0,0x1000      # $1 = 0x1000
sw $1, 0x0100($0)    # [0x100] = 0x00001000
mthi $1                # HI = 0x00001000
syscall
lw $1, 0x0100($0)    # $1 = 0x00001000
mfhi $2                # $2 = 0x00001000
_loop:
j _loop
nop

```



图 189: exception 指令集测试通过

后续又增加了新的测试指令集

ArithmeticTest	2019/12/4 18:59	文件夹
DataMoveInstTest	2019/12/5 15:40	文件夹
j_BTest	2019/12/4 16:40	文件夹
LogicInstTest	2019/12/5 15:40	文件夹
S_LInstTest	2019/12/5 15:40	文件夹
ShiftInstTest	2019/12/5 15:40	文件夹
Readme.txt	2019/12/5 15:52	文本文档
test_bench.v	2019/12/5 16:57	V 文件

图 190: 6 个测试点

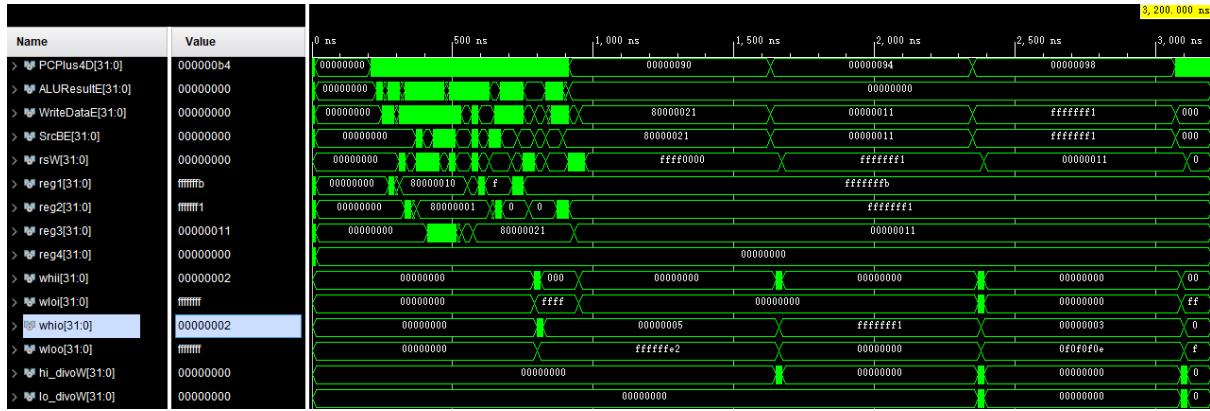


图 191: ArithmeticTest 测试点仿真

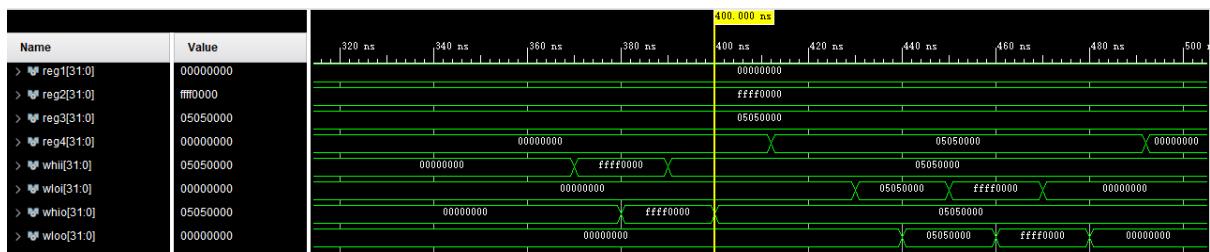


图 192: DataMoveTest 测试点仿真

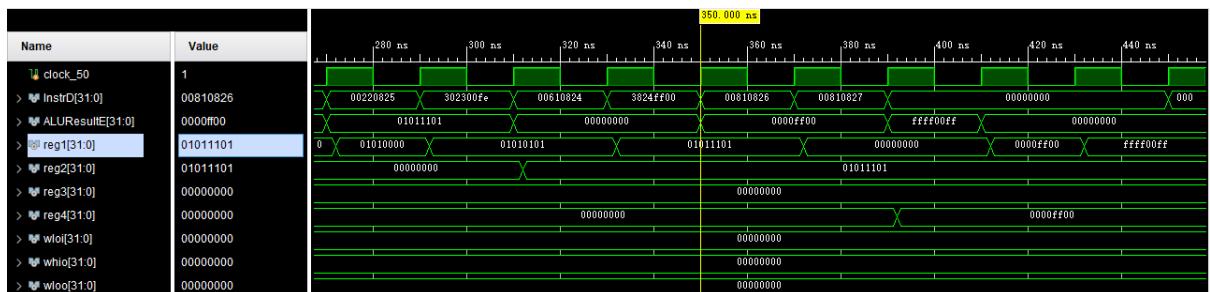


图 193: LogicInstTest 测试点仿真

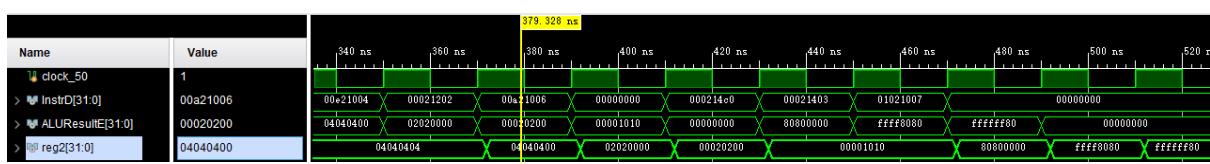


图 194: ShiftInstTest 测试点仿真



图 195: J_BTest 测试点 1 仿真



图 196: J_BTest 测试点 2 仿真

S_LInstTest 与之前的测试点 9 测试指令集基本一致,在此不再提供测试截图。

连接 sram-soc 之后通过全部的 89 个功能测试点, 连接类 sram 后 axi 通过全部的 89 个功能测试点。

sram 功能测试 89 个测试点全部通过。

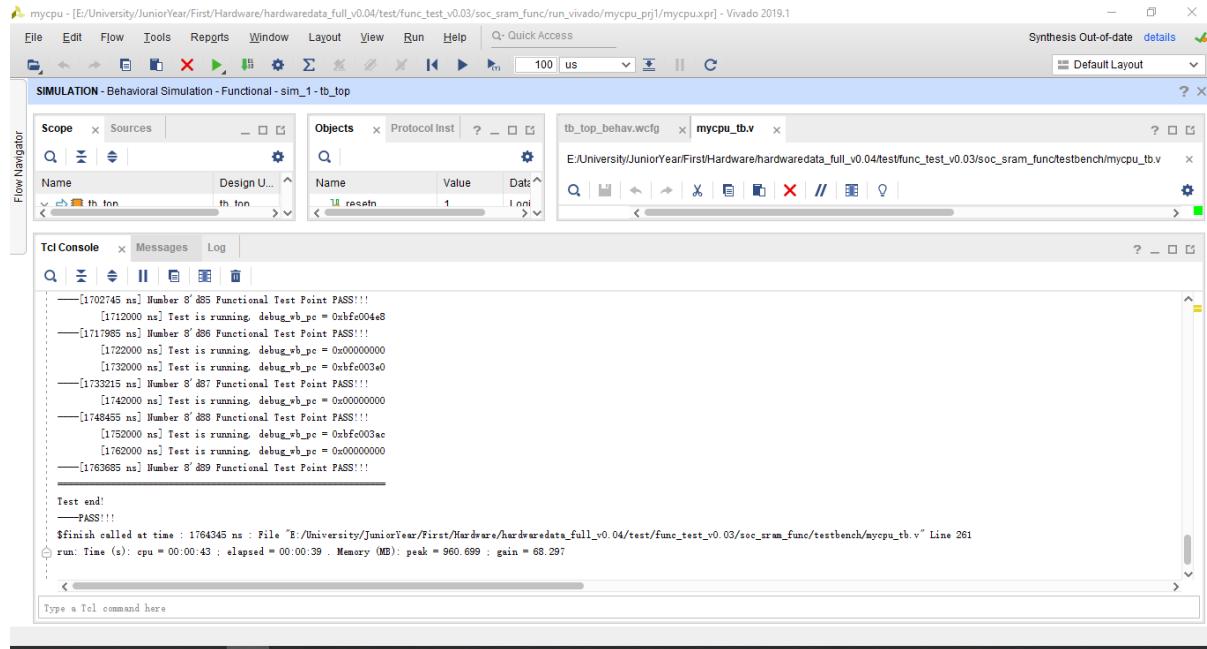


图 197: sram 功能测试 89 个测试点全部通过

axi 功能测试 89 个测试点全部通过。

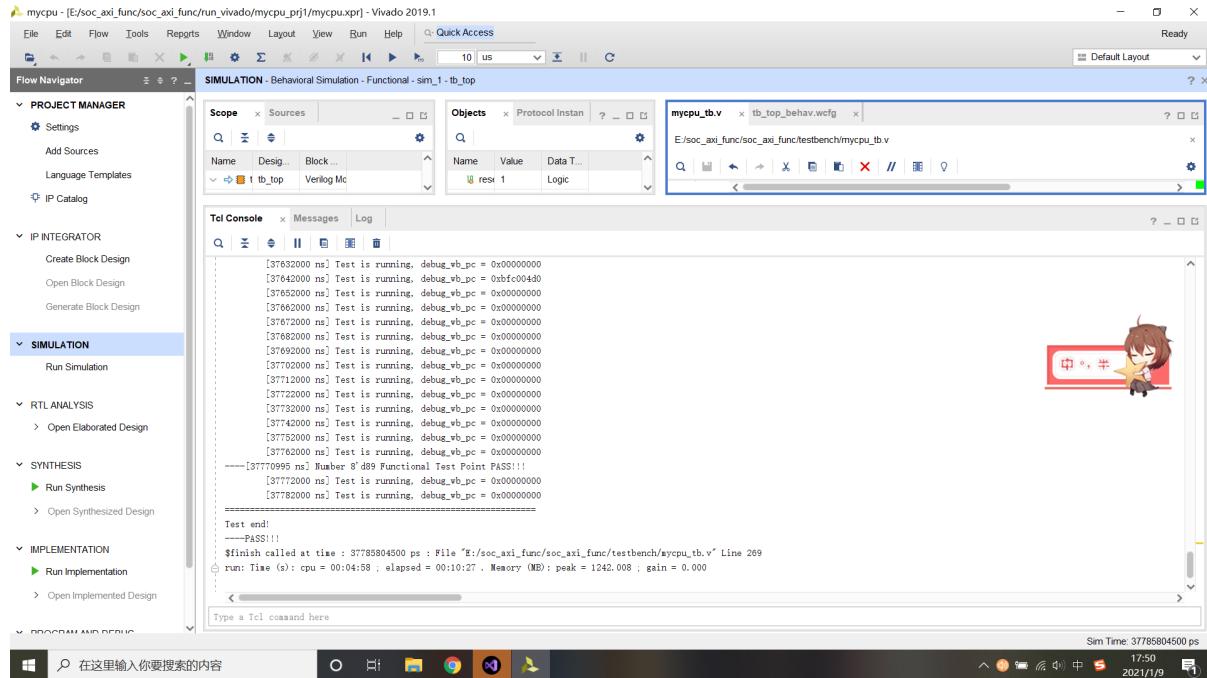


图 198: axi 功能测试 89 个测试点全部通过

通过 10 个性能测试, 全部运行成功。bitcount 运行成功。

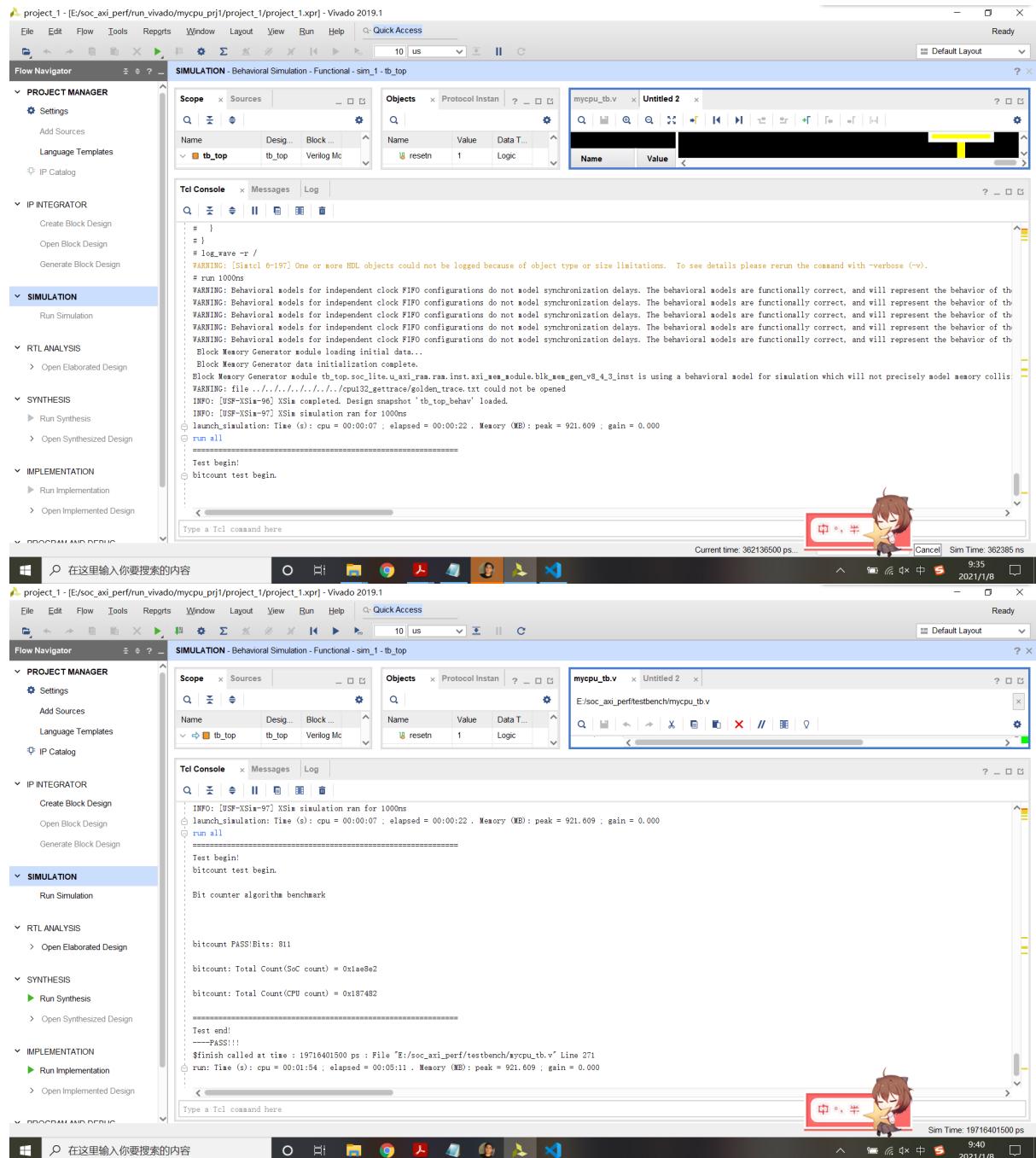


图 199: bitcount 测试通过

bubblesort 运行成功。

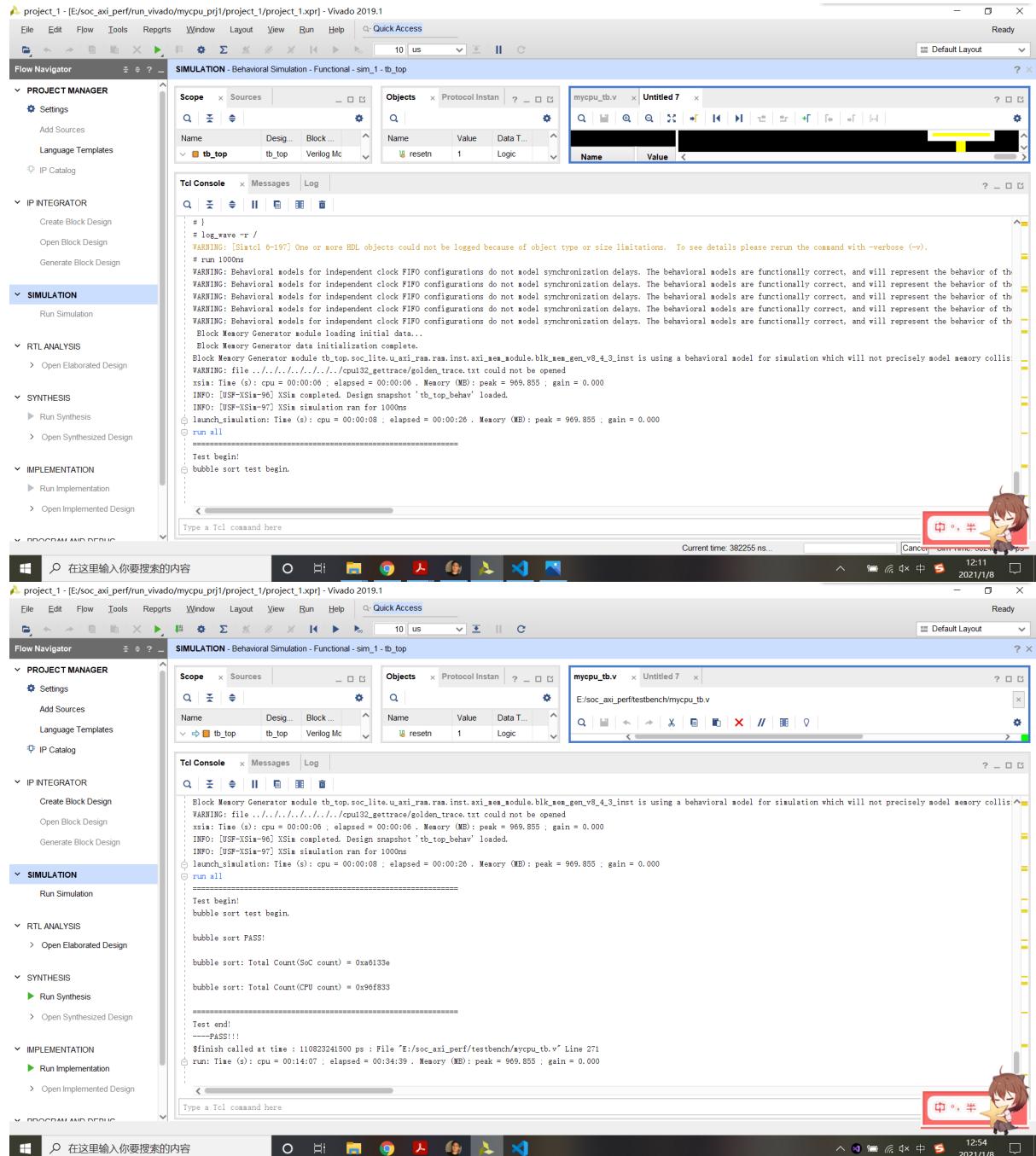


图 200: bubblesort 测试通过

coremark 运行成功。

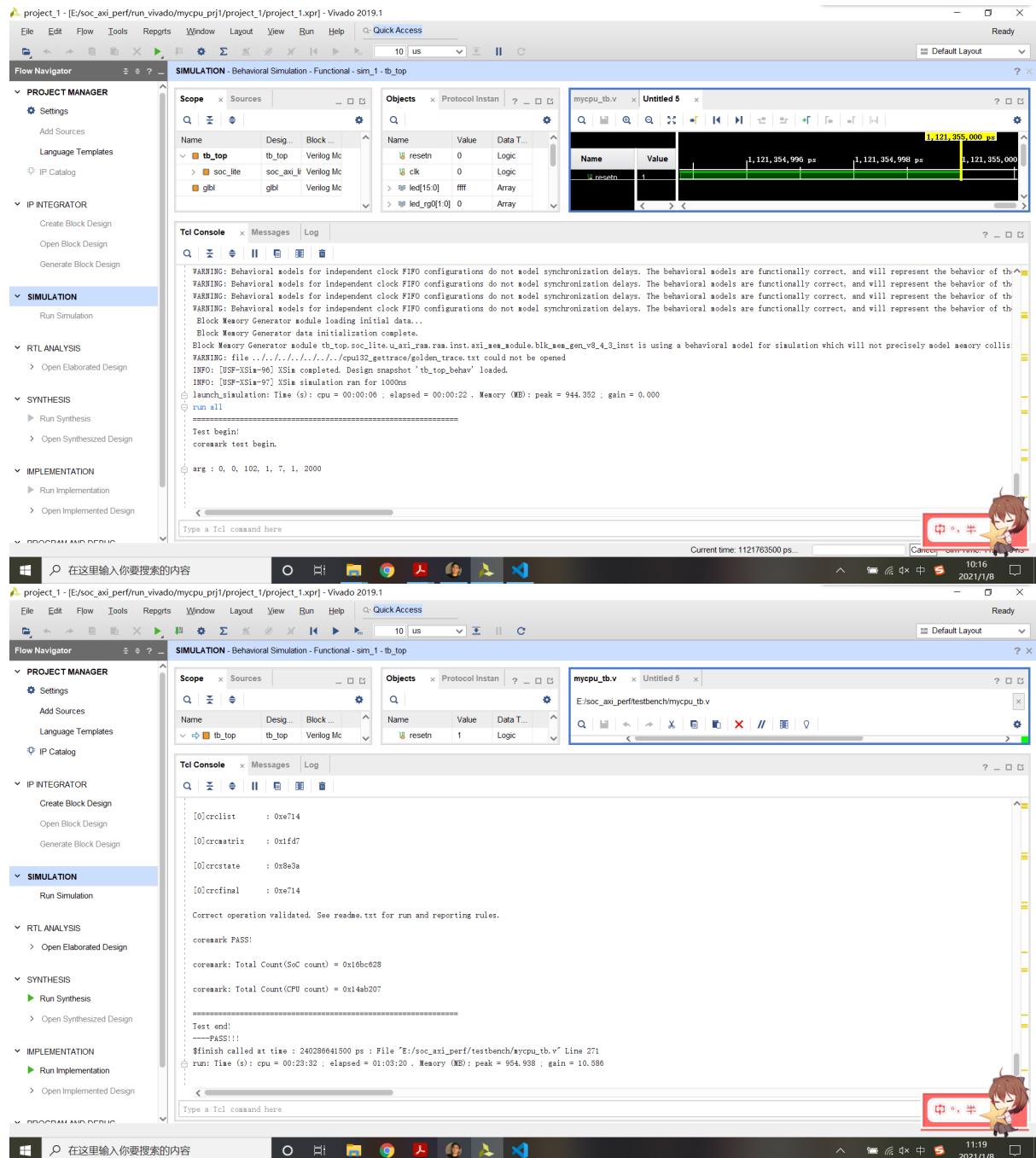


图 201: coremark 测试通过

crc32 运行成功。

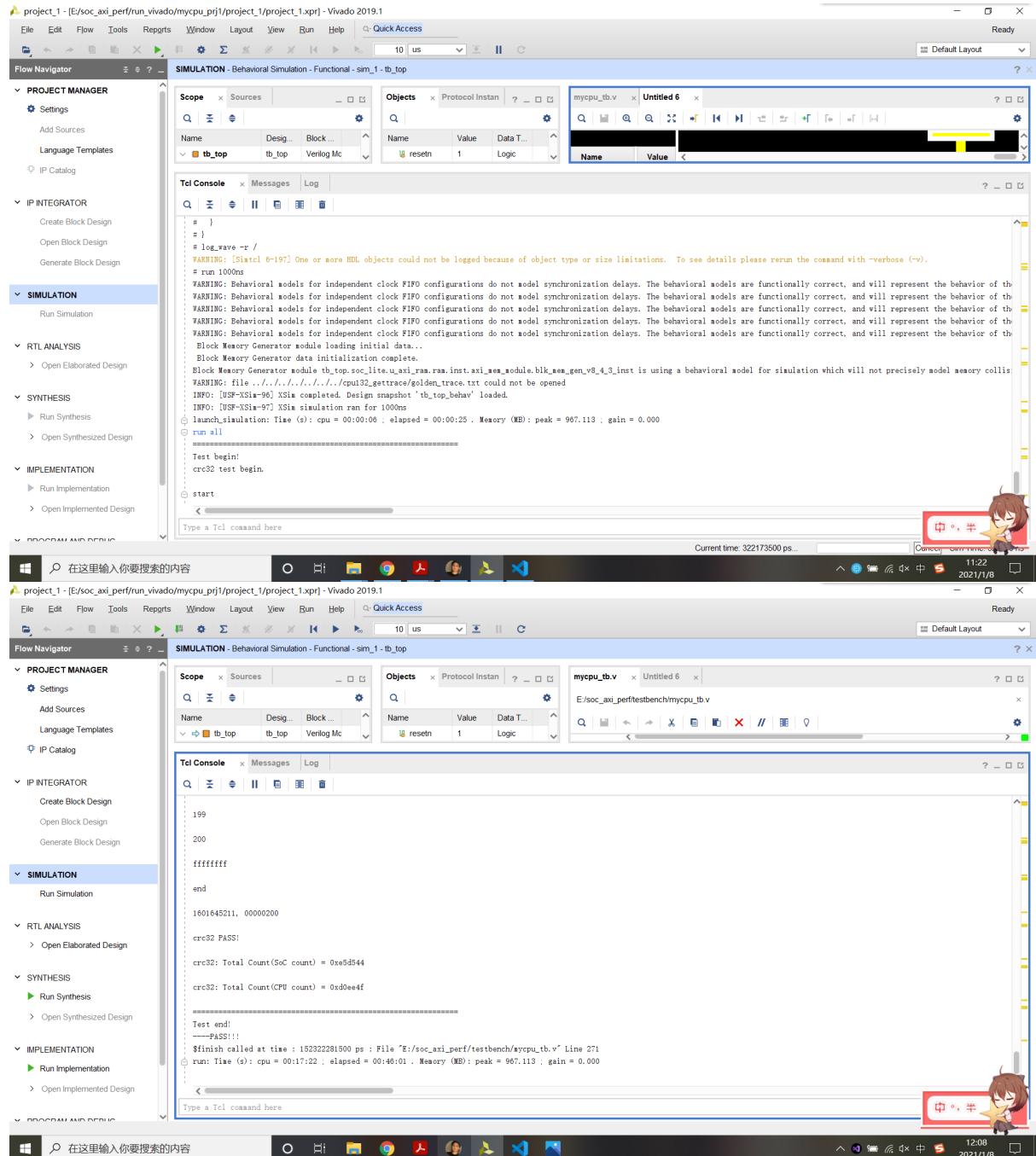


图 202: crc32 测试通过

dhrystone 运行成功。

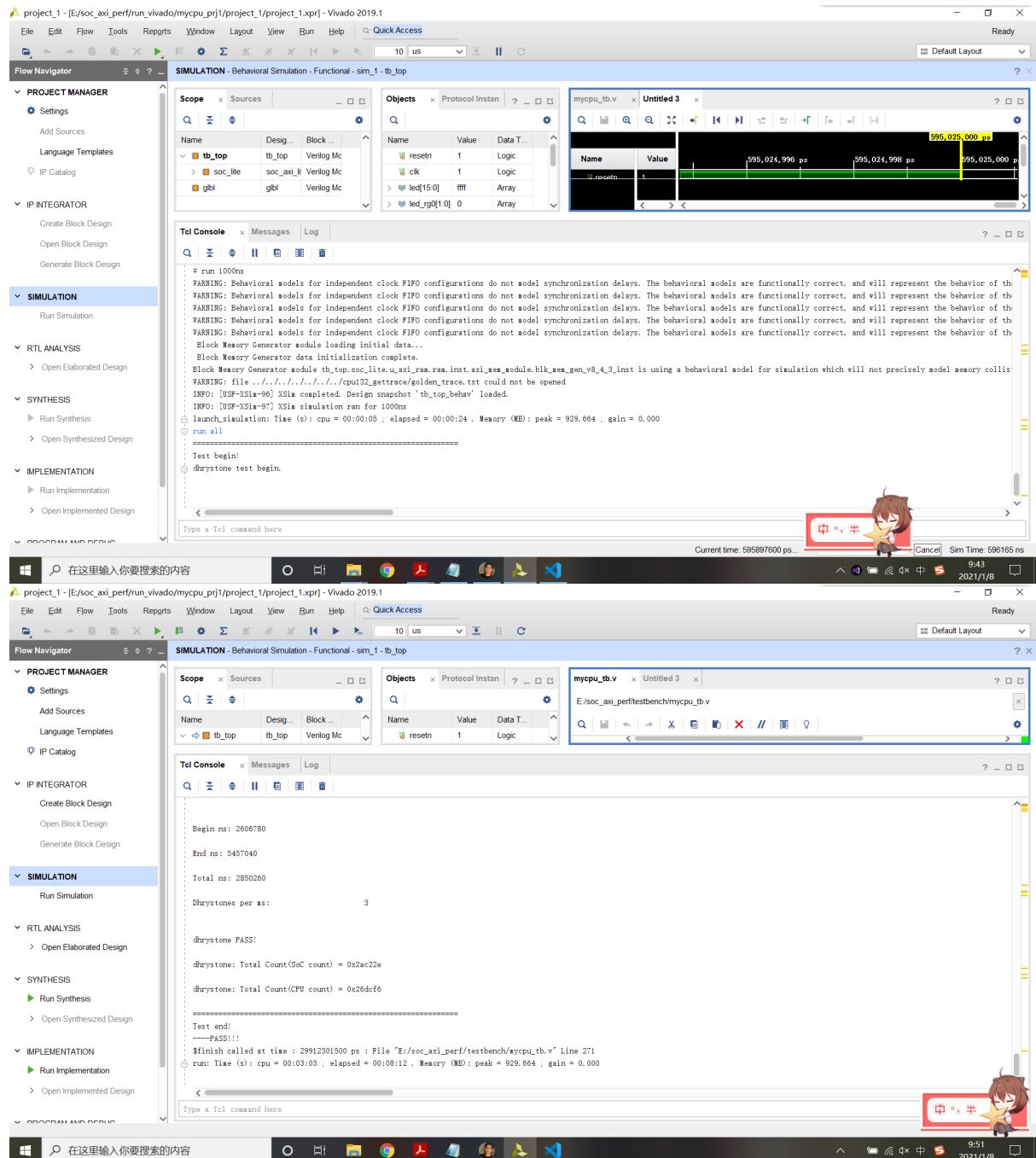


图 203: dhrystone 测试通过

quicksort 运行成功。

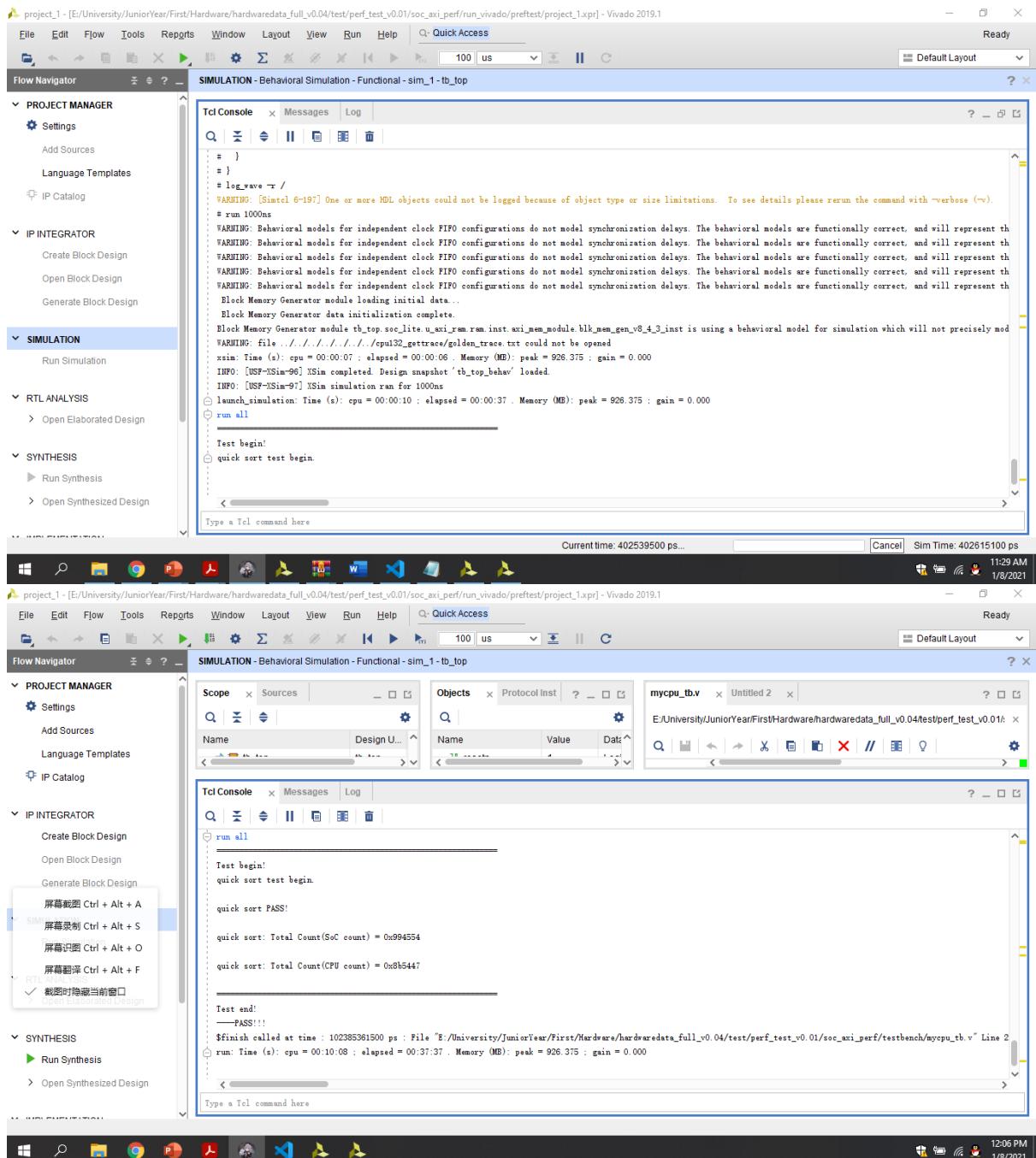


图 204: quicksort 测试通过

selectsort 运行成功。

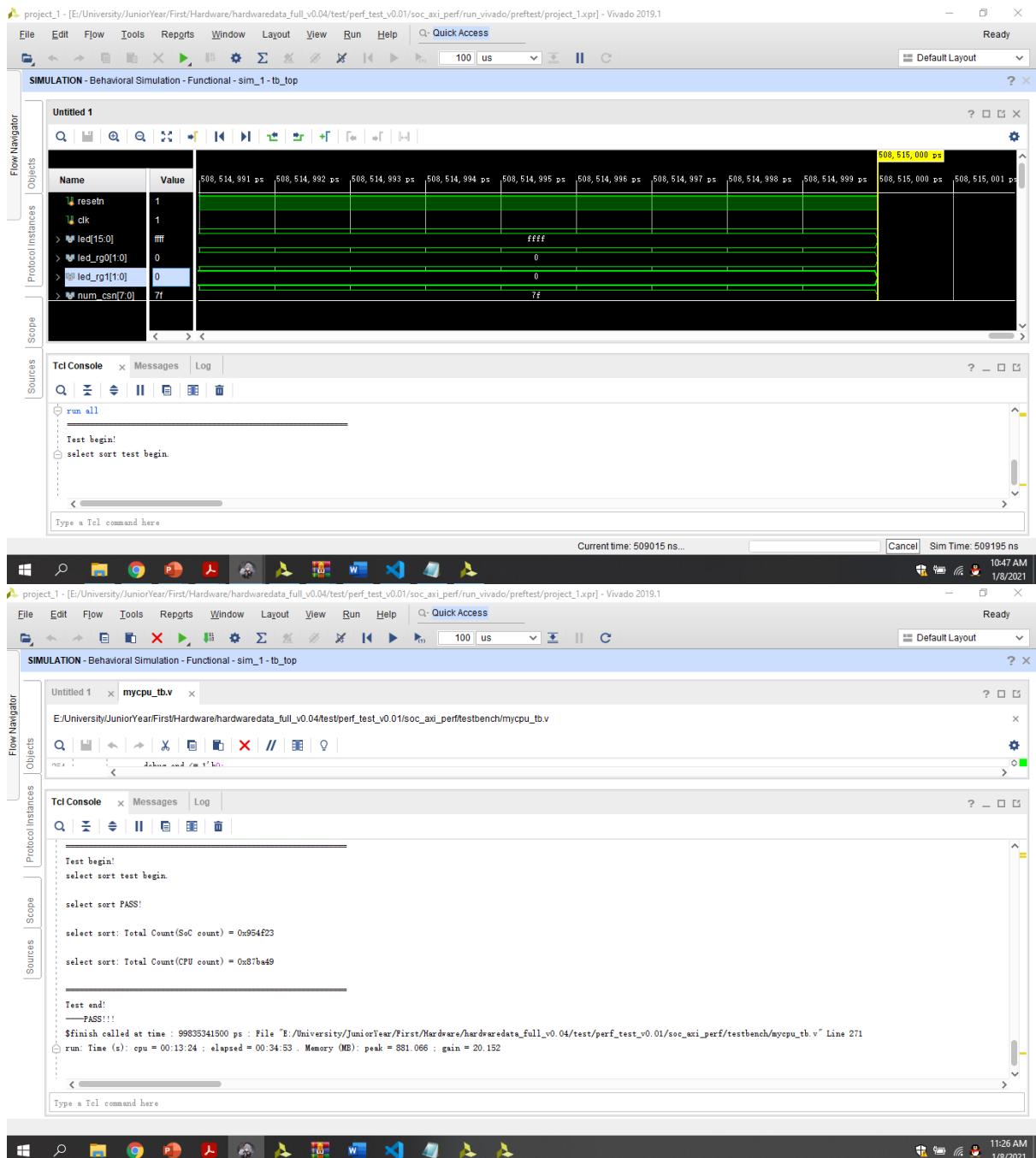


图 205: selectsort 测试通过

sha 运行成功。

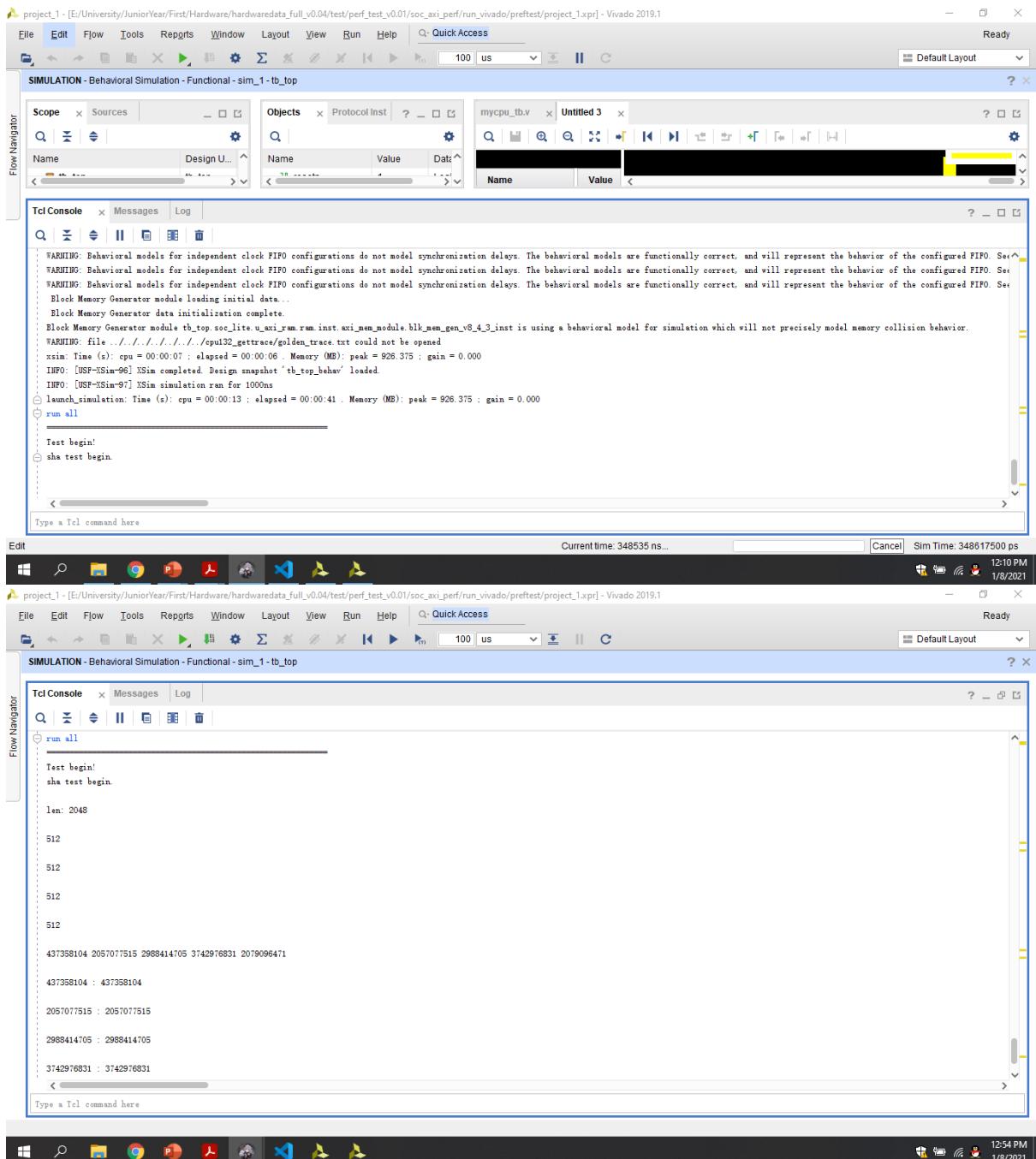


图 206: sha 测试通过

streamcopy 运行成功。

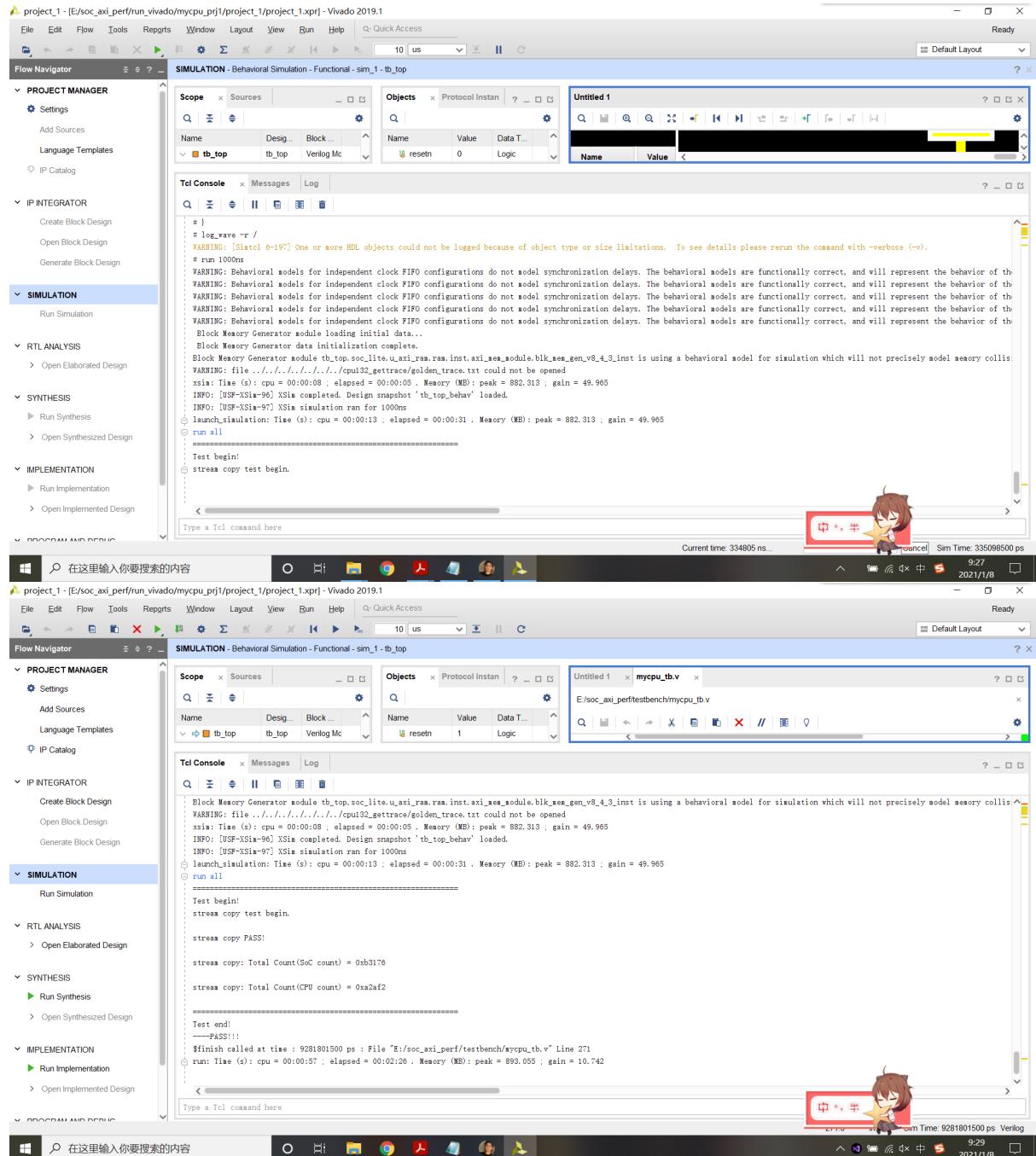


图 207: streamcopy 测试通过

stringsearch 运行成功。

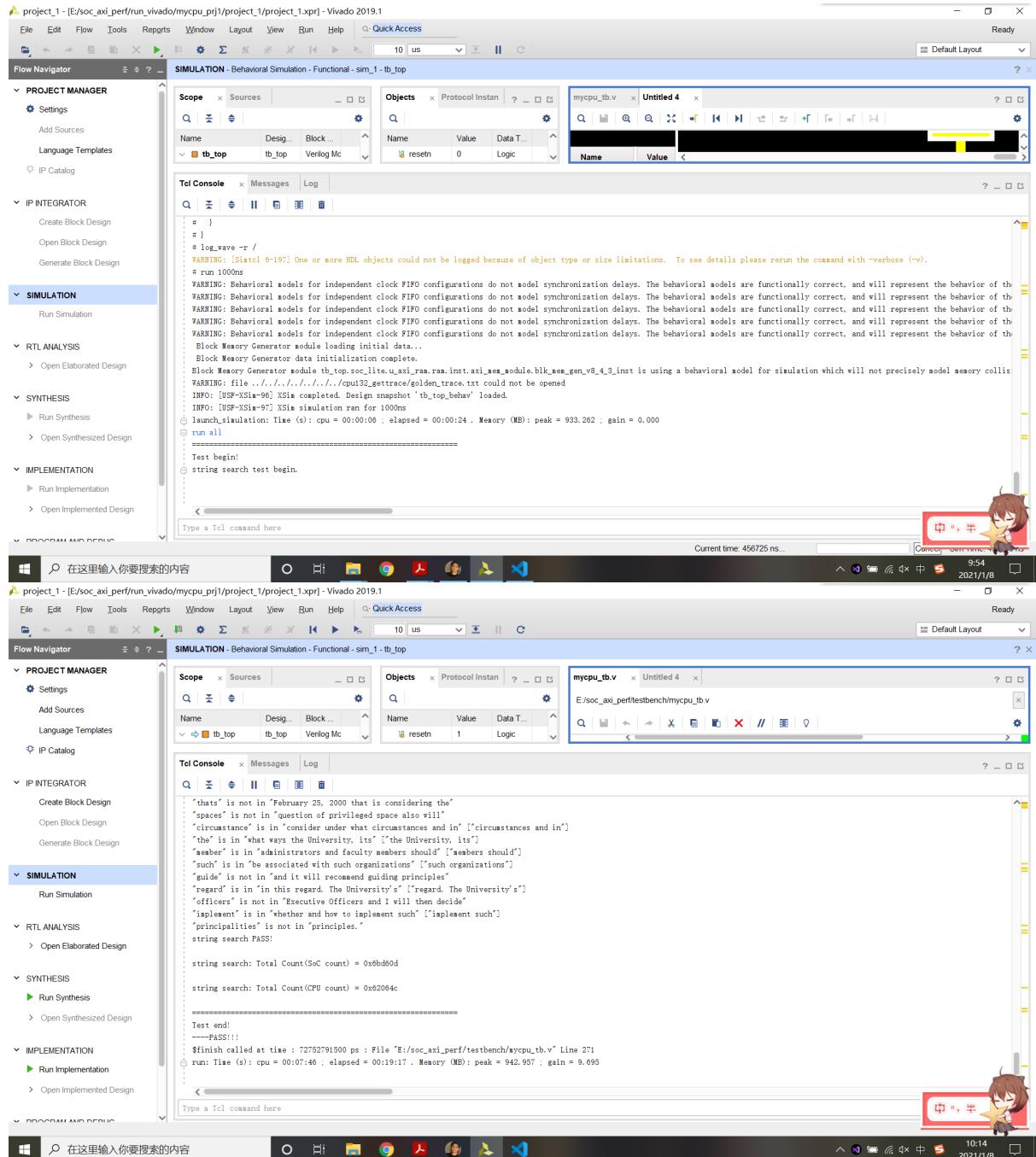


图 208: stringsearch 测试通过

5 参考设计说明

部分数据通路和译码的逻辑参考了提供的资料中的 PPT ISA 讲解 _2019.pptx、硬件综合设计讲解 2.pptx、硬综讲解 1.pptx、硬综讲解 2.pptx。

宏定义模块调用了提供的 defines.vh 和 defines2.vh

除法器的接口部分参考了吕学长的录制视频中的实现 <https://b23.tv/K7GFUg>

除法器调用了资料中提供的模块。

hilo 寄存器调用了资料中提供的模块。

载入存储指令对半字和字节的处理使用了吕学长视频中提供的方法,一种类似于掩码的思想。视频中吕学长只提供了一条指令的处理方法,设计中依样画葫芦把剩下的指令的处理补上了。

cp0 寄存器调用了资料中提供的模块。

异常处理模块异常信号传递方式和 exception 模块参考了吕学长的录制视频中的实现

<https://b23.tv/K7GFUg>

地址映射模块 mmu 使用了资料中提供的模块。

sram 接口设计参考了吕学长的录制视频中的实现,并针对自己的 cpu 信号进行了修改。

<https://b23.tv/K7GFUg>

类 sram 接口设计参考了提供的 PPT 资料和 cache0.06 版本中的 sram-sraml 转接口,使用了龙芯提供的 axi 转接桥。

cache 设计引用了这学期体系结构设计的写透 cache,修改端口适应此次实验。

6 总结

成果总结:

1. 成功添加了 57 条指令,通过了 57 条指令的单独功能测试点。
2. 成功添加 sram 进行 soc 功能测试,通过了 89 个测试点。
3. 成功完成了类 sram 转接 axi 功能测试,通过了 89 个测试点。
4. 成功完成了 axi 上的 10 个性能测试。
5. 添加了 cache,但是没有完全通过 axi89 个测试点,卡在了第 77 个测试点(软中断)。

心得总结:

这次硬综自己写了一个能接 AXI 的 cpu,对 MIPS 简单五级流水线有了很深刻的认识,清楚理解了 CPU 与其它外部设备通过总线交互的过程。同时提升了硬件设计的能力,可以很自然的从硬件的角度来思考问题。使用 Verilog 语言设计电路以及使用 Vivado 进行调试的技能也更加熟练。综合运用了数字逻辑、计算机组成原理以及计算机体系结构课堂上学到的理论知识,通过亲自实践,对这些理论的理解也更加明白了。

7 参考文献

- [1] A03_“系统能力培养大赛”MIPS 指令系统规范 _v1.00.pdf
- [2] A09_CPU 仿真调试说明 _v1.00.pdf
- [3] A11_Trace 比对机制使用说明 _v1.pdf
- [4] A12_类 SRAM 接口说明.pdf
- [5] AMBAaxi.pdf
- [6] 功能测试说明 _v0.01.pdf
- [7] 测试文件的使用.docx
- [8] Cache 实验指导书.pdf
- [9] 2017nscscc.pdf
- [10] 原始数据通路图 _2018.pdf
- [11] 指令及对应机器码 _2018.pdf
- [12] ISA 讲解 _2020.pptx
- [13] 硬件综合设计讲解 2.pptx
- [14] 硬综讲解 1.pptx
- [15] 硬综讲解 2.pptx