DiffTest - 一种高效的处理器验证方法

余子濠　王华强

提纲

- 调试理论 - 为什么调bug这么难
- DiffTest - 实践调试理论
- DiffTest应用案例
- DiffTest代码导读

调bug是码农的宿命

- 大家从学编程开始, 就注定要学会和bug相处
  - 程序设计作业bug -> Wrong Answer
  - 操作系统大作业bug -> kernel panic/QEMU崩溃或神秘重启
  - 组成原理大作业bug -> CPU跑飞/卡死
  - 编译原理大作业bug -> 生成的程序段错误
  - 真实项目的bug -> ???
- 调试哲学
  - 机器永远是对的 - 错的是你的代码
  - 未测试代码永远是错的 - bug很可能出现在你觉得“应该没问题”的代码

- 你是否思考过, bug是如何产生的?

调试理论

- 需求 -> 设计 -> 代码 -> Fault -> Error -> Failure
- 两种bug
  - 正确理解需求(specification), 但代码实现和需求不一致
    - 你知道xxx(局部变量要初始化, i和j不能搞错, 数组访问不能越界...), 但就是忘了/搞错了
  - 正确地实现了错误的需求
    - 你想当然地认为xxx(某些指令的结果需要零扩展/符号扩展, 某些标志应该更新/不更新...)

- 避免第二类bug的唯一方法 – 仔细RTFM
  - Read The Friendly Manual
  - 无论是写代码还是调试, 都有意识地RTFM, 确认自己正确理解了需求
    - 开销很低, 但能节省好几天的调试时间
  - 觉得少看几句话问题不大 -> 调试好几天
    - 某同学实现printf, 真的因为少看手册1句话, 调试了4天

调试理论

- 需求 -> 设计 -> 代码 -> Fault -> Error -> Failure
- 软件工程领域中调试相关的三种 “错误”
  - Fault - 有bug的代码, 例如数组访问越界
  - Error - 程序运行时刻的非预期状态, 如某些内存的值被错误改写
  - Failure - 可观测的致命结果, 如输出乱码/assert失败/段错误

- 第一类bug(代码实现和需求不一致)的传播路径
  - 代码 (手滑/眼瞎/脑抽)-> Fault (不一定)-> Error (不一定马上)-> Failure

- 调试 = 从Failure回溯到Fault
  - 距离越远, 调试越难

专业的调试方法

- 需求 -> 设计 -> 代码 -> Fault -> Error -> Failure

- 添加断言(assert), 把Error转变成Failure
  - 把需求(specification)直接写出来, 运行时检查
- 进行测试, 把Fault转变成Error
  - 单元测试, 随机测试, DiffTest
- 用lint工具检查代码, 暴露Fault
  - -Wall, -Werror
- 编写可读, 可维护, 易验证的代码(不言自明, 不言自证)
  - 从源头消灭bug

添加断言(Error -> Failure)

```
if (!cond) {
  // observable failure
  report_and_exit();
}
```

- 断言背后就是一个if语句
  - 关键是条件
- 如果条件是需求的代码表达, assert就是在检查程序行为是否符合需求
  - 双向循环链表
    - assert(p != NULL); assert(p->next != NULL); assert(p->prev != NULL);
    - assert(p->next->prev == p); assert(p->prev->next == p);
  - 组相连cache命中
    - assert(PopCount(HitVector) <= 1);
- 代码实现和需求不一致时, assert就会报错
  - 如果代码里面到处都是描述需求的assert, 仍然没有报错, 那你就对代码有很大信心
- 但并不是所有需求都很容易用代码表达来作为assert的条件

进行测试(Fault -> Error)

- 测试需要测试用例
  - 单元测试 - 与具体模块相关, 自行编写
  - 集成测试 - AM上的各种应用, riscv-tests
  - 系统测试 - RT-Thread

- 随机测试 - 随机产生测试用例
  - riscv-torture - 产生随机的riscv指令序列
    - https://github.com/ucb-bar/riscv-torture
  - 好处: 不用自己写了, 写测试很累的
  - 坏处: 边界情况的覆盖率比较低, 需要添加一定规则进行指导

- DiffTest - 今天的主题

使用lint工具(Fault -> Failure)

- 通过分析代码(静态程序分析), 提示编译通过但有潜在错误风险的代码

- 编译器一般自带lint工具, 如gcc
  - -Wall, -Werror

```
if (p = NULL) {
   // ...
}
```

- 电路仿真器也有lint工具
  - Verilator也有-Wall
  - 还有专门进行linting的商业工具

- 综合工具的检查更加严格
  - 规范的芯片设计流程一定要清除工具报告的所有warning
  - 任何已知风险带来的后果可能都是无法承担的

编写可读, 可维护, 易验证的代码(少写Fault)

- 正确的代码 != 好的代码
- 好代码的两个重要属性
  - **不言自明** – 仅看代码就能明白是做什么的(specification)
  - **不言自证** – 仅看代码就能验证实现是对的(verification)

- **防御性编程 - 假设你经常手滑/眼瞎/脑抽, 你应该如何编程才能尽最大可能保护你?**

- 写过一定规模的代码就知道, 编码风格不是虚的
  - 相反, 如果没写过一定规模的代码, 则不易理解
- 专业的代码能以更高的概率在意外情况下存活

例子: 编写译码器

- 你愿意通过阅读以下代码, 检查实现是否正确吗?
  - 不言自证?

```
492         assign SB = (IR[31] & ~IR[30] & IR[29] & ~IR[28] & ~IR[27] & ~IR[26]);
493         assign SH = (IR[31] & ~IR[30] & IR[29] & ~IR[28] & ~IR[27] & IR[26]);
494         assign SWR = (IR[31] & ~IR[30] & IR[29] & IR[28] & IR[27] & ~IR[26]);
495         assign SWL = (IR[31] & ~IR[30] & IR[29] & ~IR[28] & IR[27] & ~IR[26]);
496         assign SRL = (~IR[31] & ~IR[30] & ~IR[29] & ~IR[28] & ~IR[27] & ~IR[26] & ~IR[5] & ~IR[4] & ~IR[3] & ~IR[2] & IR[1] & ~IR[0
497         assign SRA = (~IR[31] & ~IR[30] & ~IR[29] & ~IR[28] & ~IR[27] & ~IR[26] & ~IR[5] & ~IR[4] & ~IR[3] & ~IR[2] & IR[1] & IR[0]
498         assign Ext0 = (~IR[31] & ~IR[30] & IR[29] & IR[28] & ~IR[27] & IR[26]) | (~IR[31] & ~IR[30] & IR[29] & IR[28] & ~IR[27] & ~
499         assign Wtrs = ((Zero == 1'b0) & (~IR[31] & ~IR[30] & ~IR[29] & ~IR[28] & ~IR[27] & ~IR[26] & ~IR[5] & ~IR[4] & IR[3] & ~IR[
500         assign LWLR1 = (IR[31] & ~IR[30] & ~IR[29] & ~IR[28] & IR[27] & ~IR[26]) | (IR[31] & ~IR[30] & ~IR[29] & IR[28] & IR[27] & 
501         assign LWLR2 = (IR[31] & ~IR[30] & ~IR[29] & ~IR[28] & IR[27] & ~IR[26]);
502         assign LWLR3 = (IR[31] & ~IR[30] & ~IR[29] & IR[28] & IR[27] & ~IR[26]);
503         assign LHU = (IR[31] & ~IR[30] & ~IR[29] & IR[28] & ~IR[27] & IR[26]);
504         assign LH = (IR[31] & ~IR[30] & ~IR[29] & ~IR[28] & ~IR[27] & IR[26]);
505         assign LBU = (IR[31] & ~IR[30] & ~IR[29] & IR[28] & ~IR[27] & ~IR[26]);
506         assign LB = (IR[31] & ~IR[30] & ~IR[29] & ~IR[28] & ~IR[27] & ~IR[26]);
507         assign ZeroJg = (~IR[31] & ~IR[30] & ~IR[29] & ~IR[28] & ~IR[27] & IR[26] & ~IR[20] & ~IR[19] & ~IR[18] & ~IR[17] & IR[16])
508         assign RegDst = (~IR[31] & ~IR[30] & IR[29] & ~IR[28] & ~IR[27] & IR[26]) | (IR[31] & ~IR[30] & ~IR[29] & ~IR[28] & IR[27] 
509         assign ALUSrc = (~IR[31] & ~IR[30] & ~IR[29] & IR[28] & ~IR[27] & IR[26]) | (~IR[31] & ~IR[30] & ~IR[29] & ~IR[28] & ~IR[27
510         assign PCSrc = ((~(Zero == 1)) & (~IR[31] & ~IR[30] & ~IR[29] & IR[28] & ~IR[27] & IR[26])) | ((Zero == 1) & (~IR[31] & ~IR
511 //      assign RegWrite = (~IR[31] & ~IR[30] & IR[29] & ~IR[28] & ~IR[27] & IR[26]) | (IR[31] & ~IR[30] & ~IR[29] & ~IR[28] & IR[27
512         assign Jmp = (~IR[31] & ~IR[30] & ~IR[29] & ~IR[28] & IR[27] & ~IR[26]) | (~IR[31] & ~IR[30] & ~IR[29] & ~IR[28] & IR[27] &
513 //      assign MemWrite = (IR[31] & ~IR[30] & IR[29] & ~IR[28] & IR[27] & IR[26]) | (IR[31] & ~IR[30] & IR[29] & ~IR[28] & ~IR[27] 
514 //      assign MemRead = (IR[31] & ~IR[30] & ~IR[29] & ~IR[28] & IR[27] & IR[26]) | (IR[31] & ~IR[30] & ~IR[29] & ~IR[28] & ~IR[27]
515         assign MemtoReg = (IR[31] & ~IR[30] & ~IR[29] & ~IR[28] & IR[27] & IR[26]) | (IR[31] & ~IR[30] & ~IR[29] & ~IR[28] & ~IR[27
516         assign SLL = (~IR[31] & ~IR[30] & ~IR[29] & ~IR[28] & ~IR[27] & ~IR[26] & ~IR[5] & ~IR[4] & ~IR[3] & ~IR[2] & ~IR[1] & ~IR[
517         assign JAL1 = (~IR[31] & ~IR[30] & ~IR[29] & ~IR[28] & IR[27] & IR[26]) | (~IR[31] & ~IR[30] & ~IR[29] & ~IR[28] & ~IR[27] 
518         assign JAL2 = (~IR[31] & ~IR[30] & ~IR[29] & ~IR[28] & IR[27] & IR[26]);
519         assign JR = (~IR[31] & ~IR[30] & ~IR[29] & ~IR[28] & ~IR[27] & ~IR[26] & ~IR[5] & ~IR[4] & IR[3] & ~IR[2] & ~IR[1] & ~IR[0]
520         assign Write_strb[3] = (IR[31] & ~IR[30] & IR[29] & ~IR[28] & IR[27] & IR[26]) | (ALUResult[1] & ALUResult[0] & IR[31] & ~I
521         assign Write_strb[2] = (IR[31] & ~IR[30] & IR[29] & ~IR[28] & IR[27] & IR[26]) | (ALUResult[1] & ~ALUResult[0] & IR[31] & ~
522         assign Write_strb[1] = (IR[31] & ~IR[30] & IR[29] & ~IR[28] & IR[27] & IR[26]) | (~ALUResult[1] & ALUResult[0] & IR[31] & ~
523         assign Write_strb[0] = (IR[31] & ~IR[30] & IR[29] & ~IR[28] & IR[27] & IR[26]) | (~ALUResult[1] & ~ALUResult[0] & IR[31] & 
524         assign ALUctr[3] = (~IR[31] & ~IR[30] & ~IR[29] & ~IR[28] & ~IR[27] & ~IR[26] & IR[5] & ~IR[4] & ~IR[3] & IR[2] & IR[1] & I
525         assign ALUctr[2] = (~IR[31] & ~IR[30] & ~IR[29] & IR[28] & ~IR[27] & IR[26]) | (~IR[31] & ~IR[30] & ~IR[29] & IR[28] & ~IR[
526         assign ALUctr[1] = (~IR[31] & ~IR[30] & IR[29] & ~IR[28] & ~IR[27] & IR[26]) | (IR[31] & ~IR[30] & ~IR[29] & ~IR[28] & IR[2
527         assign ALUctr[0] = (~IR[31] & ~IR[30] & ~IR[29] & ~IR[28] & ~IR[27] & ~IR[26] & IR[5] & ~IR[4] & ~IR[3] & IR[2] & ~IR[1] & 
```

写了上千行

```
1336        6'b001101:// ORI
1337        begin
1338                RegDst <= 0;
1339                ALUSrc <= 1;
1340                Branch <= 0;
1341                RegWrite <= 0;
1342                RegRead <= 1;
1343                MemtoReg <= 0;
1344                //MemRead <= 0;
1345                //MemWrite <= 0;
1346                Write_strb <= 4'd0;
1347                ALUcon_in <= 3'b101;
1348                Jump <= 0;
1349                JumpReg <= 0;
1350                JAL <= 0;
1351                LUI <= 0;
1352                SL <= 0;
1353                Sign <= 0;
1354                BEQ <= 0;
1355                BGEZ <= 0;
1356                BGTZ <= 0;
1357                BLEZ <= 0;
1358                BLTZ <= 0;
1359                Logic <= 0;
1360                Left <= 0;
1361                JALR <= 0;
1362                Align <= 0;
1363                Part_data <= 32'd0;
1364        end
1365
1366        6'b001110:// XORI
1367        begin
1368                RegDst <= 0;
1369                ALUSrc <= 1;
1370                Branch <= 0;
1371                RegWrite <= 0;
1372                RegRead <= 1;
1373                MemtoReg <= 0;
1374                //MemRead <= 0;
1375                //MemWrite <= 0;
```

```
case(op)
        6'b001001://addiu
                {RegDst,ALUSrc,MemtoReg,RegWrite,Branch,ALUOp,Jump,PCWrite} <= {2'b00,2'b01,4'b0000,1'b0,3'b000,3'b
        6'b000100://beq
                {RegDst,ALUSrc,MemtoReg,RegWrite,Branch,ALUOp,Jump,PCWrite} <= {2'b00,2'b00,4'b0000,1'b0,3'b001,3'b
        6'b000101://bne
                {RegDst,ALUSrc,MemtoReg,RegWrite,Branch,ALUOp,Jump,PCWrite} <= {2'b00,2'b00,4'b0000,1'b0,3'b010,3'b
        6'b000110://blez
                {RegDst,ALUSrc,MemtoReg,RegWrite,Branch,ALUOp,Jump,PCWrite} <= {2'b00,2'b00,4'b0000,1'b0,3'b100,3'b
        6'b000111://bgtz
                {RegDst,ALUSrc,MemtoReg,RegWrite,Branch,ALUOp,Jump,PCWrite} <= {2'b00,2'b00,4'b0000,1'b0,3'b110,3'b
        6'b000010://j
                {RegDst,ALUSrc,MemtoReg,RegWrite,Branch,ALUOp,Jump,PCWrite} <= {2'b00,2'b00,4'b0000,1'b0,3'b000,3'b
        6'b000011://jal
                {RegDst,ALUSrc,MemtoReg,RegWrite,Branch,ALUOp,Jump,PCWrite} <= {2'b10,2'b00,4'b0010,1'b1,3'b000,3'b
        6'b001111://lui
                {RegDst,ALUSrc,MemtoReg,RegWrite,Branch,ALUOp,Jump,PCWrite} <= {2'b00,2'b00,4'b0011,1'b0,3'b000,3'b
        6'b001010://slti
                {RegDst,ALUSrc,MemtoReg,RegWrite,Branch,ALUOp,Jump,PCWrite} <= {2'b00,2'b01,4'b0000,1'b0,3'b000,3'b
        6'b001011://sltiu
                {RegDst,ALUSrc,MemtoReg,RegWrite,Branch,ALUOp,Jump,PCWrite} <= {2'b00,2'b01,4'b0000,1'b0,3'b000,3'b
        6'b001100://andi
                {RegDst,ALUSrc,MemtoReg,RegWrite,Branch,ALUOp,Jump,PCWrite} <= {2'b00,2'b10,4'b0000,1'b0,3'b000,3'b
        6'b100100://lbu
                {RegDst,ALUSrc,MemtoReg,RegWrite,Branch,ALUOp,Jump,PCWrite} <= {2'b00,2'b01,4'b0101,1'b0,3'b000,3'b
        6'b100001://lh
                {RegDst,ALUSrc,MemtoReg,RegWrite,Branch,ALUOp,Jump,PCWrite} <= {2'b00,2'b01,4'b0110,1'b0,3'b000,3'b
```

南京大学龙芯杯经验

特色6 - 用脚本生成译码模块

ADD	SP	100000	REG	ALU	X	RD	RS	X	X	X	X	0	100000
ADDU	SP	100001	REG	ALU	X	RD	RS	X	X	X	X	0	100001
AND	SP	100100	REG	ALU	X	RD	RS	X	X	X	X	0	100100
ANDI	CO	1100	IMM	ALU	0	RT	RS	X	X	X	X	X	X
OR	SP	100101	REG	ALU	X	RD	RS	X	X	X	X	0	100101
ORI	CO	1101	IMM	ALU	0	RT	RS	X	X	X	X	X	X
BGEZAL	RI	10001	X	BRU	X	31	RS	X	X	10001	X	X	X
BEQ	CO	100	REG	BRU	X	X	RS	X	X	X	X	X	X
J	CO	10	X	BRU	X	X	X	X	X	X	X	X	X
JALR	SP	1001	X	BRU	X	RD	RS	X	X	0	X	0	1001
MUL	SP2	10	REG	MDU	X	RD	RS	X	X	X	X	0	10
DIV	SP	11010	REG	MDU	X	HILO	RS	X	X	X	0	0	11010
LB	CO	100000	X	LSU	+	RT	RS	X	X	X	X	X	X
SWR	CO	101110	REG	LSU	+	X	RS	X	X	X	X	X	X
MFC0	COP	0	X	ALU	X	RT	CP0	X	0	X	X	0	000xxx
MTC0	COP	100	REG	ALU	X	RD	X	X	100	X	X	0	000xxx
ERET	COP	10000	X	BRU	X	X	EPC	ERET	10000	0	0	0	11000
SYSCALL	SP	1100		BRU	X	X	X	SYSCALL	X	X	X	X	1100
BREAK	SP	1101		BRU	X	X	X						
MOVN	SP	1011	REG	ALU		RD	RS						
MOVZ	SP	1010	REG			RD	RS						

自动生成
Verilog

VIVADO™

20

```
11  object Instructions {
12    def LUI    = BitPat("b00111100000????????????????????????")
13    def ADD    = BitPat("b000000????????????????00000100000")
14    def ADDU   = BitPat("b000000????????????????00000100001")
15    def SUB    = BitPat("b000000????????????????00000100010")
16    def SUBU   = BitPat("b000000????????????????00000100011")
17    def SLT    = BitPat("b000000????????????????00000101010")
18    def SLTU   = BitPat("b000000????????????????00000101011")
```

```
72  // instruction decode stage
73  val csignals = ListLookup(instr.instr,
74    List(N, FU_TYPE_PRU, FU_OP_PRU_X, B_SRC_X, EXT_TYPE_X, RD_SEL_X, A_SRC_X), Array(
75      /* instr_valid  |  fu_type   |     fu_op        |  b_src   | ext_type |   rd_sel      |   a_src  |        */
76      // ALU instructions
77      ADD     -> List(Y, FU_TYPE_ALU, FU_OP_ALU_ADD_OV, B_SRC_REG, EXT_TYPE_X,    RD_SEL_RD,   A_SRC_RS),
78      ADDU    -> List(Y, FU_TYPE_ALU, FU_OP_ALU_ADD,    B_SRC_REG, EXT_TYPE_X,    RD_SEL_RD,   A_SRC_RS),
79      ADDI    -> List(Y, FU_TYPE_ALU, FU_OP_ALU_ADD_OV, B_SRC_IMM, EXT_TYPE_SIGN, RD_SEL_RT,   A_SRC_RS),
80      ADDIU   -> List(Y, FU_TYPE_ALU, FU_OP_ALU_ADD,    B_SRC_IMM, EXT_TYPE_SIGN, RD_SEL_RT,   A_SRC_RS),
81      SUB     -> List(Y, FU_TYPE_ALU, FU_OP_ALU_SUB_OV, B_SRC_REG, EXT_TYPE_X,    RD_SEL_RD,   A_SRC_RS),
82      SUBU    -> List(Y, FU_TYPE_ALU, FU_OP_ALU_SUB,    B_SRC_REG, EXT_TYPE_X,    RD_SEL_RD,   A_SRC_RS),
83      AND     -> List(Y, FU_TYPE_ALU, FU_OP_ALU_AND,    B_SRC_REG, EXT_TYPE_X,    RD_SEL_RD,   A_SRC_RS),
84      ANDI    -> List(Y, FU_TYPE_ALU, FU_OP_ALU_AND,    B_SRC_IMM, EXT_TYPE_ZERO, RD_SEL_RT,   A_SRC_RS),
```

2017年, 通过表格和python脚本,
实现译码逻辑的快速精准开发

2018年, 使用Chisel开发,
直接把表格嵌入代码中

http://www.nscscc.org/uploads/soft/171010/1-1G010133147.pdf

https://github.com/nju-mips/noop-lo

提纲

- 调试理论 - 为什么调bug这么难
- DiffTest - 实践调试理论
- DiffTest应用案例
- DiffTest代码导读

在CPU上执行程序, 结果错, 应该如何调试?

- 假设程序本身是对的

- 回顾bug的传播路径
  - 代码 (手滑/眼瞎/脑抽)-> Fault (不一定)-> Error (不一定马上)-> Failure
- 可能的Failure
  - 结果错
  - 卡死
  - assert失败

- 问题: 如何回溯到第一次触发Error的指令?
  - 这才是调试处理器最大的挑战(可能执行了成千上万的指令)

assert的启发

- 放任CPU运行 = 执行到最后才进行人工判断
  - 结果错 = manually_assert(结果应该正确)失败
  - 卡死 = manually_assert(程序应该输出)失败

- 如果我们希望马上把Error的指令转变成Failure, 就不应该最后才assert
- 能不能在每条指令执行之后插入一个特殊的assert?
  - 如果assert失败, 我们就找到了第一条出错的指令!

- 听上去很棒!
- 但这个特殊的assert应该检查什么?

回到计算机的本质

- CPU = 一个执行指令的死循环

```
while (1) {
    取出PC指向的指令;
    指令译码;
    指令执行;
    更新PC;
}
```

- 如果我们要检查指令，应该检查什么？
- 更关键地：执行一条指令之后，计算机究竟有什么变化？

- 电路视角的计算机 = 组合逻辑电路 + 时序逻辑电路 = 一个巨大状态机
  - 我们可以检查计算机的状态！
  - 状态 = 时序逻辑电路 = 寄存器 + PC + 内存

- 那怎么知道一个状态是否正确的呢？

秘诀 - DiffTest

- 源于软件工程领域, 全称Differential Testing(差分测试)
- 核心思想: 对于根据同一规范的两种实现, 给定相同的有定义的输入, 它们的行为应当一致

- 回到处理器设计: 对于根据riscv手册的两种实现, 给定相同的正确程序, 它们的状态变化应当一致

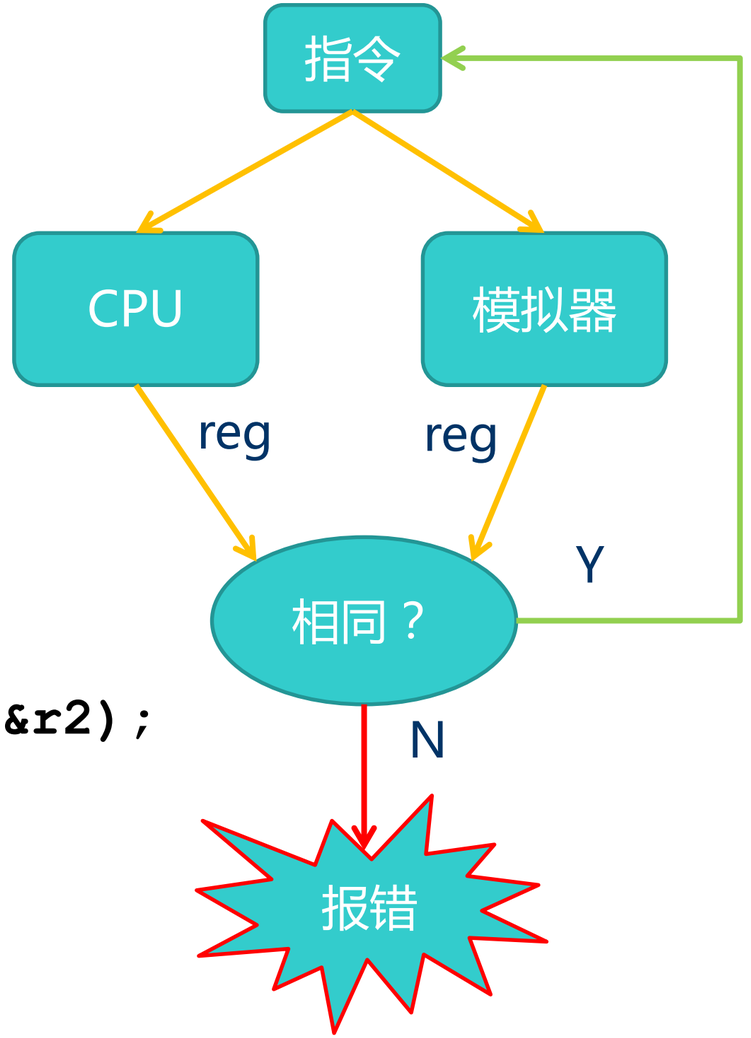
- 两种实现, 其中一种是我们的CPU
- 另一种实现选择什么呢?
  - 选一个简单的, 模拟器就可以了

与模拟器进行DiffTest

- 选一个模拟器作为参考实现(REF)
  - QEMU/Spike/NEMU...
- 为模拟器添加以下API

API	说明
difftest_memcpy(dest, src, size)	从src复制size字节到REF的内存地址dest中
difftest_getregs(r)	获得REF的寄存器状态
difftest_setregs(r)	设置REF的寄存器状态
difftest_exec(n)	令参考实现执行n条指令

```
while (1) {
  cpu_step();
  difftest_exec(1);
  cpu_getregs(&r1);
  difftest_getregs(&r2);
  assert(r1 == r2);
}
```



- 让仿真框架可以获得CPU的寄存器状态
- 有了这些API, 就可以在仿真框架中很容易实现DiffTest了

DiffTest的意义

```
while (1) {
  cpu_step();
  difftest_exec(1);
  cpu_getregs(&r1);
  difftest_getregs(&r2);
  assert(r1 == r2);
}
```

- DiffTest = 在线指令级行为验证方法
  - 在线 = 边跑程序边验证
  - 指令级 = 执行的每条指令都验证
- 能把任意程序转化为指令级别的测试, 对状态进行断言
  - 支持不会结束的程序, 例如OS
- 无需提前得知程序的结果
  - 因为我们对比的是指令执行的行为, 而不是程序的语义

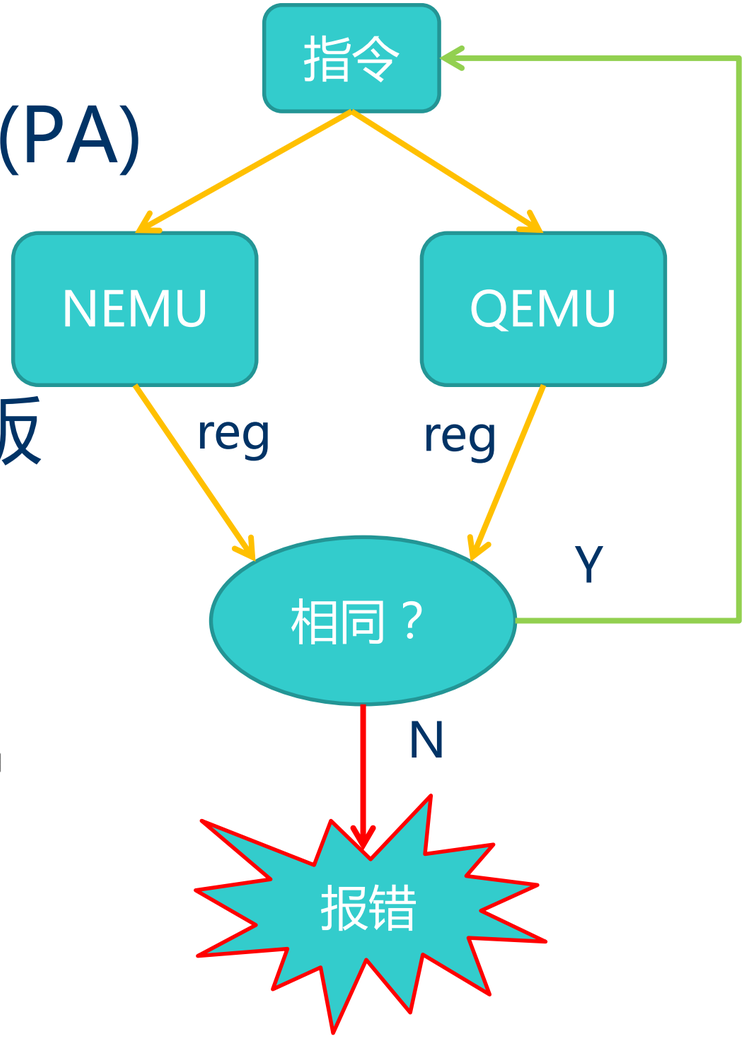
- riscv-torture通过比较signature(最终的寄存器状态)来判断执行结果
  - 比较signature = 离线程序级行为验证方法
  - 从Failure回溯到Error还是很困难
  - 如果程序不能结束, 就无法比较

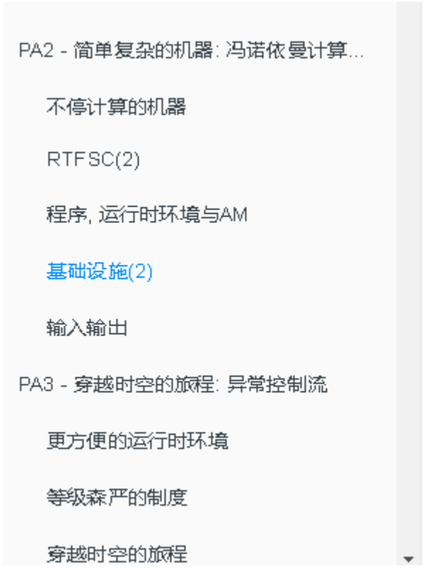
提纲

- 调试理论 - 为什么调bug这么难
- DiffTest - 实践调试理论
- DiffTest应用案例
- DiffTest代码导读

2017年: DiffTest起源, 引入南京大学PA实验

- 2017年秋季正式引入南京大学计算机系统基础实验(PA)
  - 学生要求实现NEMU模拟器, 和QEMU进行DiffTest
  - 结束了指令bug调试极其困难的黑暗时代
- 2017年的PA从多种意义上来说都是PA现代版的初版
  - 引入AM, DiffTest, 中间语言





Differential Testing

如果你在程序设计课上听说过上述这些建议, 相信你几乎不会遇到过运行时错误. 然而回过头来看上文提到的指令实现的bug, 我们会发现, 这些工具还是不够用: 我们很难通过 `assert()` 来表达指令的正确行为来进行自动检查, 而 `printf()` 和GDB实际上并没有缩短error和failure的距离.

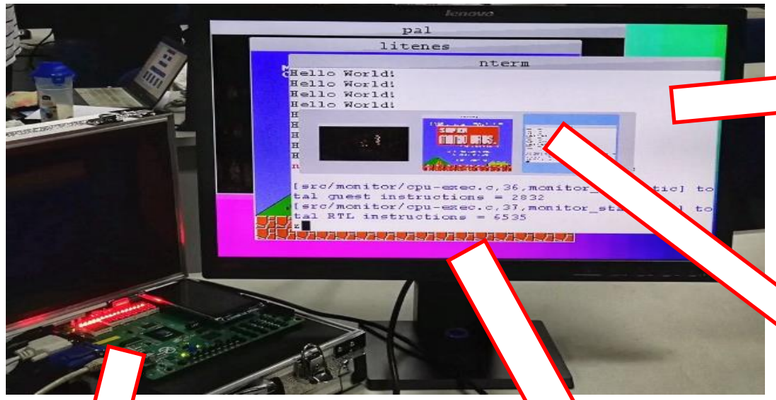
如果有一种方法能够表达指令的正确行为, 我们就可以基于这种方法来进行类似 `assert()` 的检查了. 那么, 究竟什么地方表达了指令的正确行为呢? 最直接的, 当然就是i386手册了, 但是我们恰恰就是根据i386手册中的指令行为来在NEMU中实现指令的, 同一套方法不能既用于实现也用于检查. 如果有一个i386手册的参考实现就好了. 嘿! 我们用的真机不就是根据i386手册实现出来的吗? 我们让在NEMU中执行的每条指令也在真机中执行一次, 然后对比NEMU和真机的状态, 如果NEMU和真机的状态不一致, 我们就捕捉到error了!

这实际上是一种非常奏效的测试方法, 在软件测试领域称为differential testing. 我们刚才提到了"状态", 那"状态"具体指的是什么呢? 我们在PA1中已经认识到, 计算机就是一个数字电路. 那么, "计算机的状态"就恰恰是那些时序逻辑部件的状态, 也就是寄存器和内存的值. 其实仔细思考一下, 计算机执行指令, 就是修改这些时序逻辑部件的状态的过程. 要检查指令的实现是否正确, 只要检查这些时序逻辑部件中的值是否一致就可以了!

https://nju-projectn.github.io/ics-pa-gitbook/ics2017/2.4.html

2018年: DiffTest助力NOOP获得龙芯杯第二名

- 首个采用DiffTest的FPGA项目: 先实现NEMU, 作为REF与NOOP进行对比



x86
模拟器

nterm
终端模拟器

图形应用程序
NES模拟器、仙剑

NWM 窗口管理器

libos, libc, libndl (图形)

功能测试

基准程序

打字游戏

Nanos
操作系统

AM (抽象计算机) = TRM + IOE + [CTE] + [VME]

mips32

NOOP 乱序处理器

NEMU全系统模拟器

应用

运行库

操作系统

抽象层

ISA

微结构

2018年: DiffTest助力NOOP获得龙芯杯第二名

- 南京大学书写了

  一周正确实现一个乱序发射乱序执行处理器，成功运行自制分时多任务操作系统Nanos和仙剑奇侠传

  的神话

取反之前忘记零扩展

```
306  307    def tlb_write(i:UInt):Unit = {
307  308      val entry = tlb(i)
     309  +   val mask = ~(pagemask.mask.ZExt(32))
308  310      if(param.pipelineDebug) {
```

Fixed bug of insufficient bits of pagemask
141242068-ouxianfei authored 1 year ago

differ with nemu when running nanos at 922149 cycle
141242068-ouxianfei authored 1 year ago

基础设施：效果展示

突破限制，改进工具

	Verilog	Chisel	NEMU
实现类似功能	500行	150行 (代码密度x3)	N/A
性能仿真时间	20min (大赛官方平台)	30s (速度提升40倍)	0.5s (提升2400倍)

七天实现乱序处理器

差分测试：三天内修复了6个乱序执行在特定复杂条件下触发的non-trivial bug
在2分钟内定位了某TLB bug

When cache instr trigger
Set bht 1 history length
Added --no-gen to avoid
Added page fault excepti
Fixed bug of insufficier

决赛现场最快速度完成指令添加

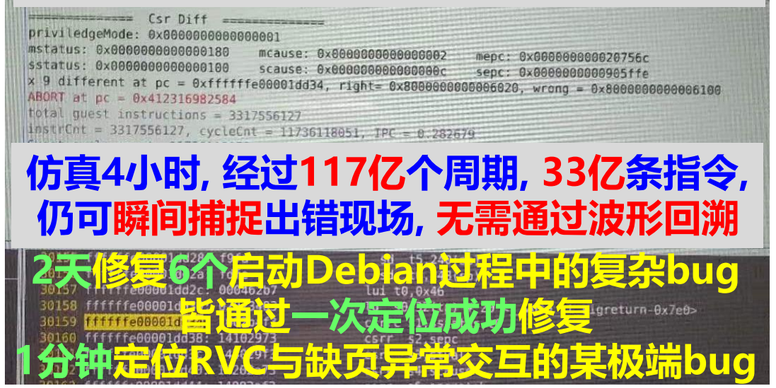
5分钟完成代码开发和验证（第1支完成的队伍）
综合完毕后一次通过验收

2018年龙芯杯南京大学二队决赛答辩 报告
http://www.nscscc.org/a/wangjie/NSCSCC2018/2018/1010/46.html

2019年: DiffTest助力首期一生一芯项目进行硅前验证

- 首个采用DiffTest的流片项目
  - RV64IMAC处理器
  - 仿真起Linux + Debian, 跑GCC/QEMU

与NEMU模拟器进行指令级别差分测试 [1]
捕捉了99%的处理器功能bug
5天启动Linux + 运行Busybox
4天启动Debian + 运行GCC/QEMU



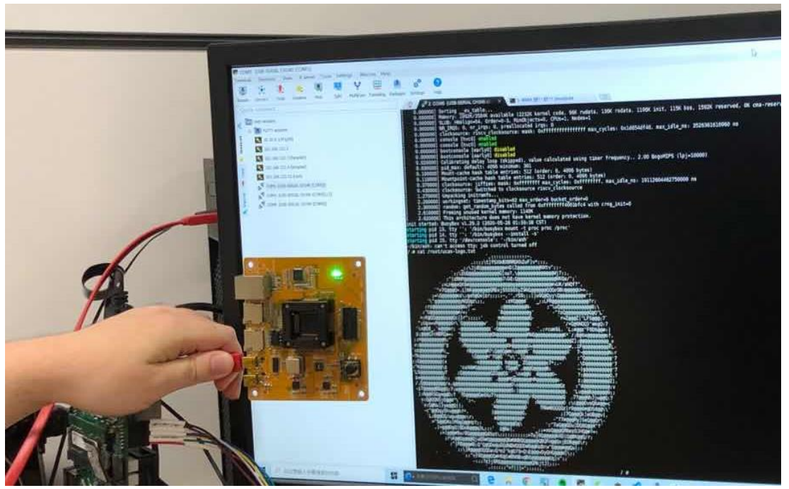
仿真4小时，经过117亿个周期，33亿条指令，
仍可瞬间捕捉出错现场，无需通过波形回溯

2天修复6个启动Debian过程中的复杂bug
皆通过一次定位成功修复
1分钟定位RVC与缺页异常交互的某极端bug

[1] Yu, EasyDiff: An Effective and Efficient Framework forProcessor Verification, CRVF 2019 https://crvf2019.github.io/pdf/14.pdf

投片生产后完成封装的芯片

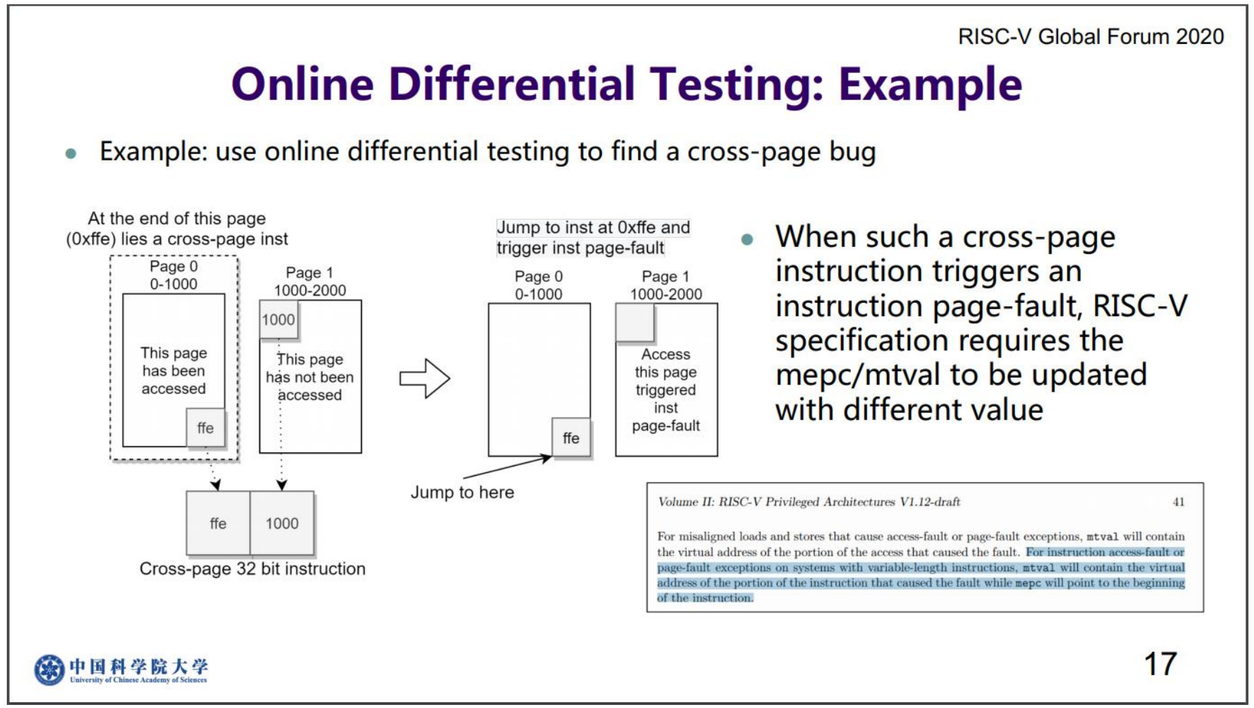
本科生带着自己设计的芯片
参加毕业远程答辩并展示



在芯片上运行经适配的Linux
系统，并展示中科院logo

2019年: DiffTest助力首期一生一芯项目进行硅前验证

- 学生在CRVF 2020和RISC-V global forum介绍果壳项目时, DiffTest均作为关键技术进行介绍



果壳调试Linux时, 在DiffTest的强力帮助下, 两天修复12个bug, 以至于没有留下印象深刻的bug

DiffTest捕捉到在Debian上运行GCC时的bug, 1分钟后定位到原因: 跨页指令取指缺页处理不正确

http://crva.ict.ac.cn/crvs2020/index/slides/1-2.pdf

https://static.sched.com/hosted_files/riscvglobalforum2020/9c/RISC-V_Global_Forum_2020-HuaqiangWang-NutShell.pdf

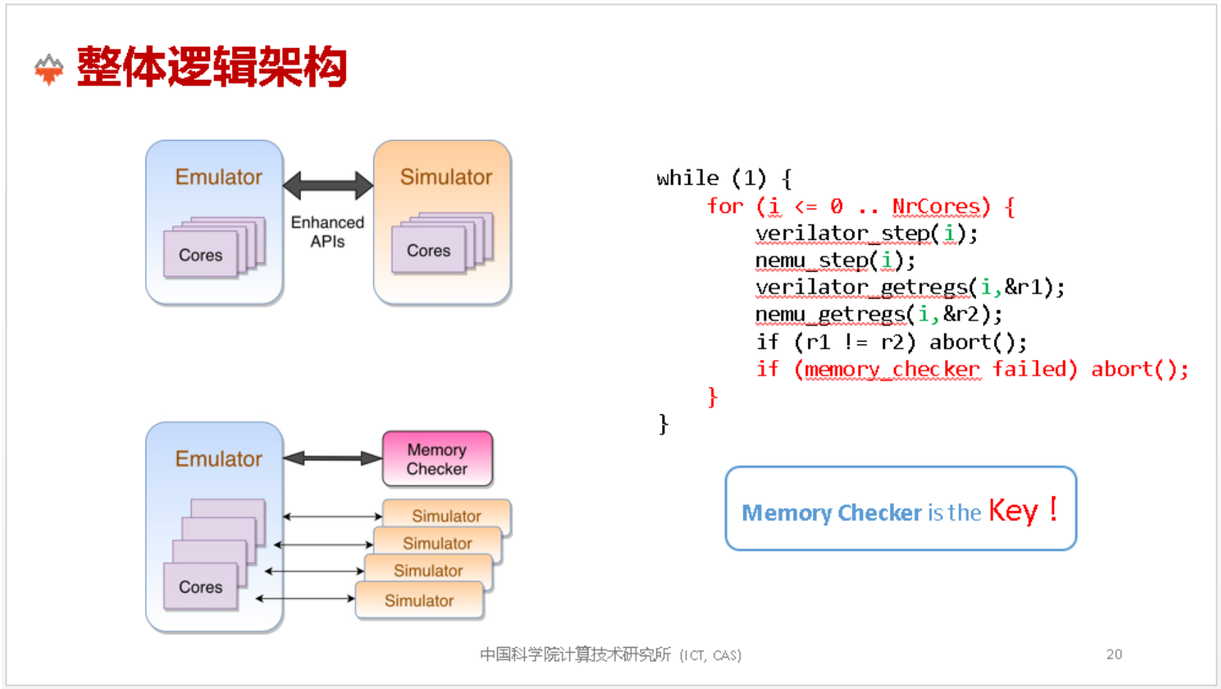
2020年: DiffTest助力开源高性能处理器香山进行硅前验证

- 在香山开发早期(第二周)就搭建DiffTest框架, 之后便一直开启
  - https://github.com/OpenXiangShan/XiangShan/pull/27

https://www.bilibili.com/video/BV19X4y1w7EB

2021年: 香山团队实现了支持多核的DiffTest

- https://github.com/OpenXiangShan/XiangShan-doc/tree/main/slides
  - 20210624-RVWC-SMP-Difftest 支持多处理器的差分测试方法.pdf



调试理论视角:
新增memory checker的assert

SMP DiffTest报告了新的硬件bug,
修复后香山成功启动多核Linux

告别波形，让处理器调试变得轻松

- 波形的信息密度很低，对人的理解不友好
- 香山团队开发了“波形终结者”

https://www.bilibili.com/video/BV1QV411H7gN

DiffTest给处理器验证方法带来新的可能性

	项目	效果
2017	南京大学PA实验	大幅降低了调试指令bug的难度
2018	南京大学 参加龙芯杯	**一周**正确实现**乱序发射乱序执行**处理器, 并运行自制**分时多任务**操作系统和复杂应用**仙剑奇侠传**
2019	首期一生一芯 果壳处理器	**5天**成功启动**Linux**运行**Busybox**, **4天**成功启动**Debian**运行**GCC/QEMU**
2020	开源高性能 RISC-V处理器香山	项目启动后**第3周**成功运行**coremark**,第5周成功运行**仙剑奇侠传**, **第3个月**成功启动**Linux**, **第4个月**成功启动**Debian**
2021	开源高性能 RISC-V处理器香山	成功启动**SMP Linux**; 环境就绪后首次上FPGA即可正确跑完**所有SPEC 2006 REF**测试, 无需在板卡上调试任何处理器相关的bug

- 没用过的不知道, 用过的都说好!

其它领域的DiffTest

- 对于根据同一规范的两种实现, 给定相同的有定义的输入, 它们的行为应当一致

- 编译器 - gcc vs. clang
  - 规范 = C语言标准
- 工具链生成的调试信息 - gcc/gdb vs. clang/lldb vs. rustc/lldb
  - 规范 = 标准UNIX调试格式(DWARF)
- 文件系统 - ext4, btrfs, NFS...
  - 规范 = VFS(虚拟文件系统)
- 航空航天容错 - 处理器三副本同时执行, 少数服从多数

提纲

- 调试理论 - 为什么调bug这么难
- DiffTest - 实践调试理论
- DiffTest应用案例
- DiffTest代码导读(请王华强主持)

小结

- 调试理论: 需求 -> 设计 -> 代码 -> Fault -> Error -> Failure
  - 添加断言(assert), 把Error转变成Failure
  - 进行测试, 把Fault转变成Error

- DiffTest = 在线指令级行为验证方法
  - 对于根据riscv手册的两种实现, 给定相同的正确程序, 它们的状态变化应当一致
  - CPU vs. 模拟器

- DiffTest给处理器验证方法带来新的可能性
  - 大幅降低指令/处理器调试的难度
  - 已在南大PA实验, 龙芯杯, 一生一芯和香山处理器开发过程中取得了很好的效果

参考资料

- DiffTest概述
  - https://nju-projectn.github.io/ics-pa-gitbook/ics2020/2.4.html
- DiffTest选讲(习题课)
  - 习题课ppt - http://jyywiki.cn/ICS/2020/slides/12.slides#/
  - B站录播 - https://www.bilibili.com/video/BV1qa4y1j7xk?p=11