

# 基于Labeled RISC-V的 芯片敏捷开发实践

余子濠, 刘志刚, 李一苇, 黄博文,  
王卅, 孙凝晖, 包云岗

2019.08@上海



中国科学院计算技术研究所  
Institute Of Computing Technology Chinese Academy Of Sciences



鹏城实验室

# Chisel – 面向敏捷开发的HDL



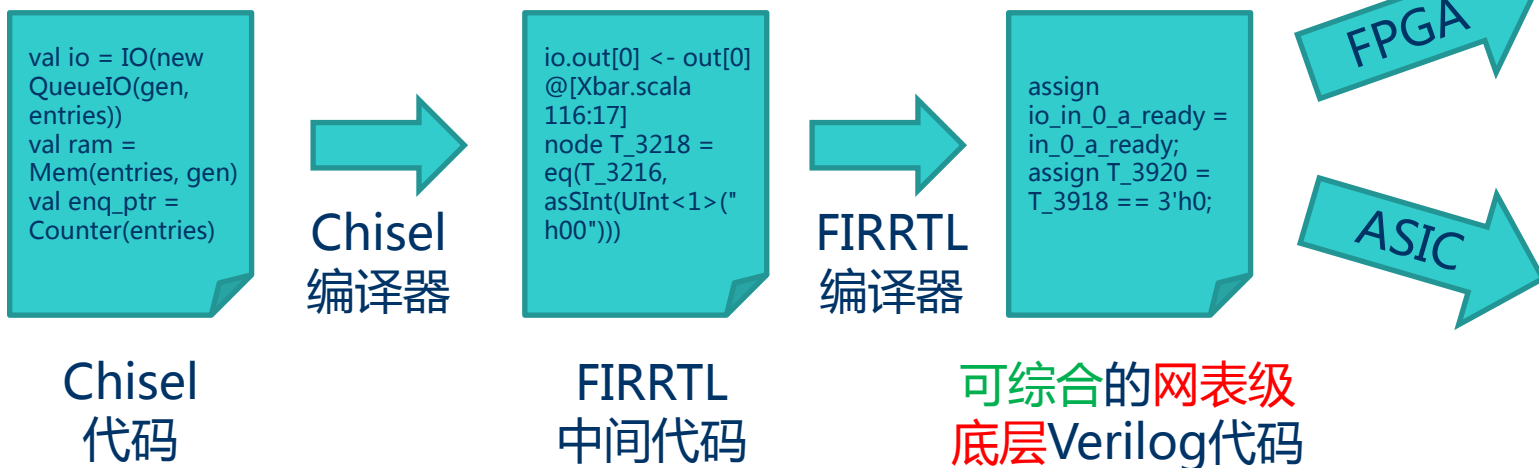
## ▶ 面向敏捷开发的新语言Chisel

- 宗旨 – 多快好省地描述电路(不是HLS)

  - ▶ 减少重复代码, 提升开发效率, 提升代码可读性和易维护性

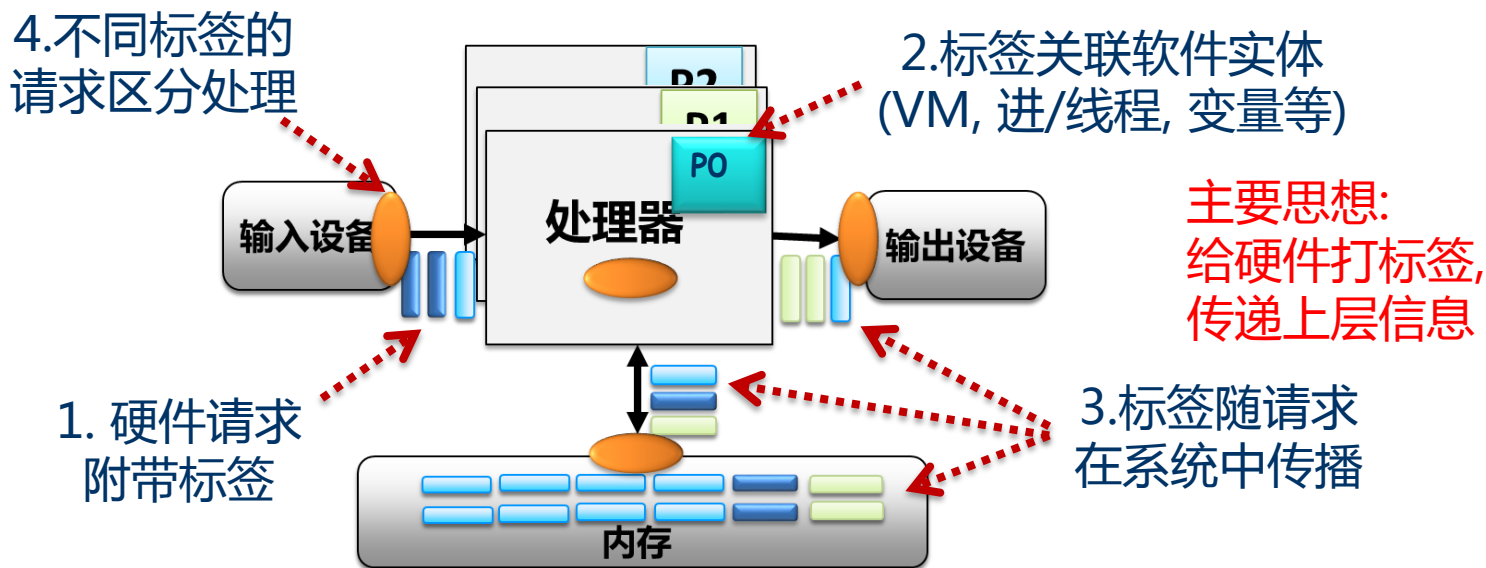
- 卖点 – 信号整体连接, 基于Scala的元编程, 面向对象编程, 函数式编程

## ▶ 开发流程



# 背景 - Labeled RISC-V项目

## ▶ 基于Rocket Chip的标签化体系结构[1]的实现



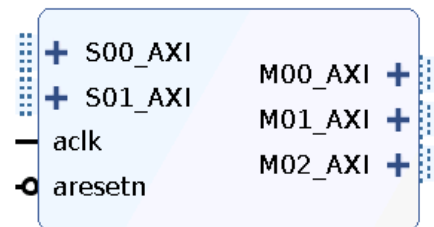
## ▶ 一个应用 - 性能调控[2]

## ▶ 今天介绍项目中敏捷开发的那些事

[1] Bao and Wang, Labeled von Neumann Architecture for Software-Defined Cloud, Journal of Computer Science and Technology, 2017 Vol. 32 (2): 219-223

[2] Ma et. al, Supporting Differentiated Services in Computers via Programmable Architecture for Resourcing-on-Demand (PARDo), ASPLOS, 2015

# Verilog编码“基本功” – 连线



来源: Vivado IP核配置界面

## ▶ 大部分模块之间通过总线连接

- 完整的AXI总线有40多个信号: 有点麻烦, 耐心点连, 还行
- 实例化一个2进3出的AXI crossbar: 要连 $40 * 5 = 200$ 根信号, 忍!
- 实例化3个这样的crossbar: 你肯定受不了

## ▶ 别高兴太早, 还会连错呢

- 连线错误经常发生在模块集成的过程中

- ▶ input/output弄反了
- ▶ 不小心漏了个信号没连
- ▶ arvalid和rvalid只差一个a
- ▶ 信号位宽定义疏忽



```
.mem_axi4_0_aw_ready(dut_mem_axi4_0_aw_ready),  
.mem_axi4_0_aw_valid(dut_mem_axi4_0_aw_valid),  
.mem_axi4_0_aw_bits_id(dut_mem_axi4_0_aw_bits_id),  
.mem_axi4_0_aw_bits_addr(dut_mem_axi4_0_aw_bits_addr),  
.mem_axi4_0_aw_bits_len(dut_mem_axi4_0_aw_bits_len),  
.mem_axi4_0_aw_bits_size(dut_mem_axi4_0_aw_bits_size),  
.mem_axi4_0_aw_bits_burst(dut_mem_axi4_0_aw_bits_burst),  
.mem_axi4_0_aw_bits_lock(dut_mem_axi4_0_aw_bits_lock),  
.mem_axi4_0_aw_bits_cache(dut_mem_axi4_0_aw_bits_cache),  
.mem_axi4_0_aw_bits_prot(dut_mem_axi4_0_aw_bits_prot),  
.mem_axi4_0_aw_bits_qos(dut_mem_axi4_0_aw_bits_qos),
```

## ▶ 手工连线 = 毫无技术含量的重复性脏活

# 信号整体连接

- ▶ Chisel可以定义一组信号的类型
  - 通过运算符<>对同类型的两组信号进行整体连接
    - ▶ core(0).out <> arbiter.in(0)
- ▶ Labeled RISC-V中添加标签并传播只要4行代码
  - 原因 – 代码中使用<>对总线进行连接
    - ▶ 修改总线定义时, 一改全改

```
+trait HasDsid extends HasTileLinkParameters {  
+  val dsid = UInt(width = tlDsidBits)  
+}  
  class AcquireMetadata(implicit p: Parameters) extends  
ClientToManagerChannel  
  with HasCacheBlockAddress  
+  with HasDsid  
  with HasClientTransactionId
```

```
core(0).out <> arbiter.in(0)
```

- ▶ 展示了需求变更时可快速拥抱变化
  - Verilog需要进行全局修改, 非常麻烦
  - SystemVerilog有interface特性, 但无法嵌套

# 基于Scala的元编程 – 模板

- ▶ 借助Scala特性对抽象出Chisel代码的共性部分
  - 具体的Chisel代码由Scala特性生成
  - 进一步减少冗余代码
  - 例子 – 用模板实现队列的原型

```
1 class Queue[T <: Data](gen: T, val entries: Int) extends
Module() {
2   val io = IO(new QueueIO(gen, entries))
3   val ram = Mem(entries, gen)
4   // ...
   val q = Module(new Queue(new TileLinkBundle, 8))
```

- ▶ 我们希望标签随请求一同穿过各种缓冲队列
  - 但无需修改任何代码
  - 原因 – 队列原型可适用于任意类型的元素
  - 若使用Verilog, 需全局手动增加队列元素的宽度, 十分繁琐
  - SV也支持模板, 但代码不可综合[1], 大多用于编写测试激励

[1] <https://stackoverflow.com/questions/20312810/what-systemverilog-features-should-be-avoided-in-synthesis>

# 面向对象编程 – 继承

- ▶ 重用父类的特性, 减少冗余代码
  - 还能让类型检查的过程更严格
- ▶ 例子 – 继承总线的各种参数

```
1 class TileLinkTrafficGenerator(implicit p: Parameters) extends
  TLModule() (p) {
2   val io = IO(new Bundle {
3     val out = new ClientTileLinkIO
4     val traffic_enable = Bool().asInput
5   })
20 // ...
```

- ▶ 负载发生器模块**自动**拥有TLModule的所有特性
  - 包括总线的大量参数, 如地址位宽, 数据位宽, ID位宽等
  - 可在第3行直接定义TileLink端口, **无需显式指定参数**
- ▶ Verilog不支持继承, 需用**10倍代码**编写此模块
- ▶ SystemVerilog部分支持继承, 但代码**不可综合**[1]

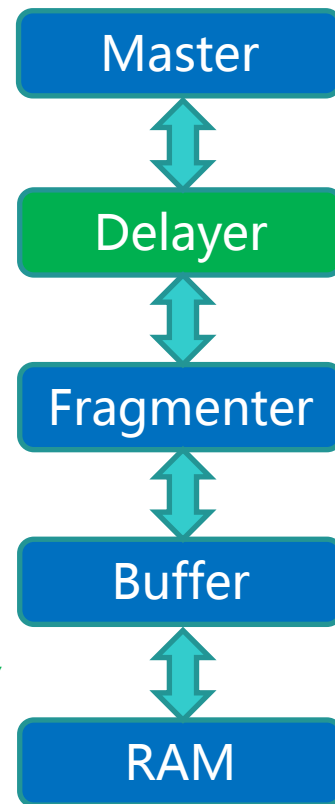
[1] <https://stackoverflow.com/questions/20312810/what-systemverilog-features-should-be-avoided-in-synthesis>

# 面向对象编程 – 重载

- ▶ 运算符重载可重新定义运算符的行为
  - 提高代码的可读性
- ▶ 例 – Diplomacy对":="运算符进行重载
  - 让其两侧AXI4节点的主从端口使用<>连接
- ▶ 在Labeled RISC-V中, 可通过少量代码在数据通路上添加延迟器

```
val node = AXI4MasterNode(List(edge.master))
val sram = LazyModule(new AXI4RAM(...))
sram.node := AXI4Buffer() := AXI4Fragmenter() := AXI4Delayer(0,
150) := node
```

- ▶ Verilog和SystemVerilog均不支持重载
  - 只能用模块来实例化, 并引入大量连线

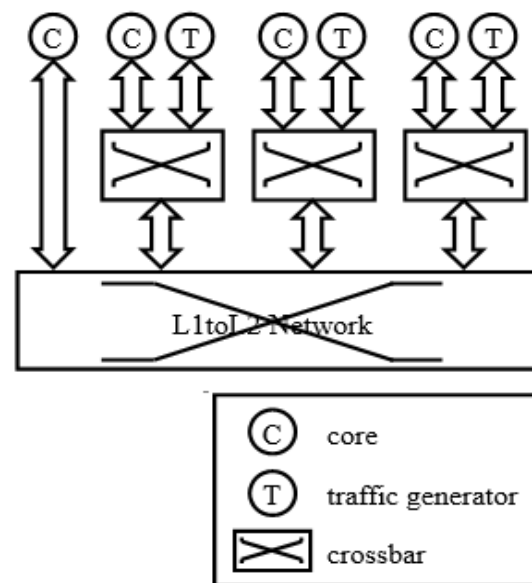




# 函数式编程

- ▶ 编写更紧凑, 可读性更好的代码
  - 使用容器(collection)来抽象电路元素
    - ▶ 容器中可以是信号, 寄存器, 端口, 模块, 映射等等
    - ▶ 或者是这些元素的复合
  - 使用map算子对容器中的对象进行批量操作
    - ▶ 操作可以是连接, 归约, 算术和逻辑运算, 选择, 实例化, 函数调用, 计算新映射等等
    - ▶ 或者是这些操作的复合
    - ▶ 操作结果返回1个新的容器

- ▶ 例子 – 如图所示接入负载发生器
  - 只在第2~n个核外接入
  - n可变

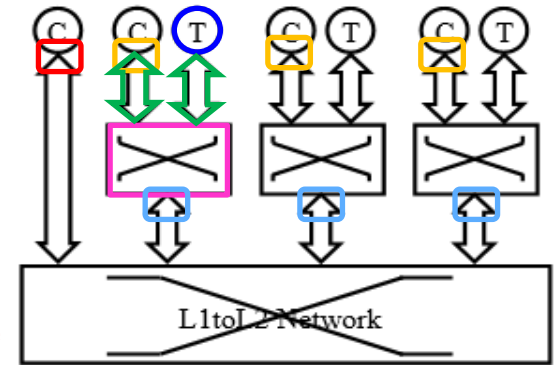


# 函数式编程

- ▶ 可通过10行Chisel代码实现

```
1 val cachedPorts = cachedPortsBeforeGenerator.take
2   cachedPortsBeforeGenerator.drop(1).map {
3     case p => {
4       val trafficGenerator = Module(new
TileLinkTrafficGenerator()(p_alter))
5       trafficGenerator.io.traffic_enable := io.traffic_enable
6       val trafficGeneratorArb = Module(new
ClientTileLinkIOArbiter(2)(p_alter))
7       trafficGeneratorArb.io.in <> List(p, trafficGenerator.io.out)
8       trafficGeneratorArb.io.out
9     }
10  }
```

- ▶ 展示了Chisel确实是硬件构造语言, 而非HLS
  - 容器中的对象和map算子的操作都是电路中的概念
- ▶ 传统HDL难以实现
  - for和generate机制只能对整数迭代



# Chisel特性小结

- ▶ 上述例子解读可参考《芯片敏捷开发实践：标签化RISC-V》[1]

特性	Chisel	SystemVerilog	Verilog
信号整体连接	支持	支持, 但有限制	不支持
元编程	支持	部分支持, 且不可综合	不支持
面向对象编程	支持	部分支持, 且不可综合	不支持
函数式编程	支持	不支持	不支持

- ▶ 特性通常混合使用
- ▶ 敏捷开发需要有一门这样的语言, 来
  - 减少重复代码
  - 提升开发效率
  - 提升代码可读性和易维护性

# 开发效率对比案例

## ▶ 实现一个简单的L2 cache

	一位工程师	一位本科生
项目经验	熟悉OpenSparc T1, 修改过Xilinx Cache	做过CPU课程设计, 有9个月Chisel开发经验
开发模式	传统开发	敏捷开发
开发语言	Verilog	Chisel
是否复用已有代码/测试环境	否, 独立开发/构建测试环境(花费约3周)	是, 使用Chisel库和Labeled RISC-V项目的测试环境
周期	6周	3天
有效代码/行	~1700	~350
效果	目前仍无法启动Linux	可启动多核Linux, 支持DMA模式的以太网[1]

## ▶ 敏捷开发的效率是传统开发的14倍!

- 代码量约为传统开发的1/5
- [1] <https://github.com/LvNA-system/labeled-RISC-V/blob/master/src/main/scala/l2cache/TLSimpleL2.scala>

# 花絮

- ▶ 在开发过程中实现"数量可配的总线连线"功能

	Verilog	Chisel
代码量/行	~250	2
实现周期	1~2天	<10秒
Bug数量/个	2	0
调试周期	3天	无需调试
可读性	编写了注释, 但一周后也要思考一段时间才明白代码如何工作	一目了然

- ▶ 工程师自评

- 用Verilog实现这一功能实在太繁琐了




- ▶ 要数量可配, 数组下标和连线又多, 还要顾及握手协议

- 就算实现了, 代码可读性也不好

- ▶ 结论 - Chisel开发效率高, 不易出错, 可读性好

# 开发质量对比案例

- ▶ 让另一名Chisel零基础的本科生, 来翻译工程师的核心模块并评估
  - Vivado 2017.01, FPGA型号xc7v2000tfhg1716-1

	Verilog	Chisel (直接翻译)	Chisel-opt (使用高级特性和库)
最高频率/MHz	135.814	136.388 (+0.42%)	154.107 (+13.47%)
功耗/W	0.770	0.749 (-2.73%)	0.749 (-2.73%)
LUT逻辑	5676	6422 (+13.14%)	2594 (-54.30%)
LUT存储	1796	1264 (-29.62%)	1492 (-16.93%)
FF	4266	3638 (-14.72%)	747 (-82.49%)
有效代码/行	618	470 (-23.95%)	155 (-74.92%)
工程师 心路历程			

# 花絮

```
wire _T_588; // @[Monitor.scala 73:14:freechip  
wire _T_590; // @[Monitor.scala 73:14:freechip  
wire _T_611; // @[Monitor.scala 80:25:freechip  
wire [7:0] _T_731; // @[Monitor.scala 85:30:fre  
5.8]  
wire [7:0] _T_732; // @[Monitor.scala 85:28:fre  
6.8]  
wire _T_734; // @[Monitor.scala 85:37:freechip
```

- ▶ 工程师一开始并不相信这一评估数据
  - 怀疑是Chisel代码有错 -> 非预期的优化
  - 但代码通过了工程师编写的测试
- ▶ 看到生成的Verilog代码, 更加不敢相信这一结果
  - "这样的代码, 评估结果应该很差劲才对"
  - "也许是代码太混乱了, 才正巧匹配上一些复杂原语"
- ▶ 但最后也不得不承认
  - "除非FPGA工程师直接调用原语, 不然正常情况下只会跟Chisel的资源消耗持平, 或者结果只会更差"
  - "如果ASIC也是这样的趋势, Chisel肯定是下一代HDL的强力竞争者"

# 案例小结

- ▶ 一个本科生的Chisel新手
  - 可以在更短的时间内编写更少的代码
  - 代码质量就能达到和工程师相当的水平
  - 甚至还可以超越工程师
- ▶ 即使有20%的差距, 敏捷开发仍有优势
  - 节省人力和时间
  - 能快速构建原型是很有意义的
  - 后续迭代优化
    - ▶ Chisel和FIRRTL编译器均开源, 可自由定制
- ▶ 敏捷开发确实大大降低了硬件设计的门槛

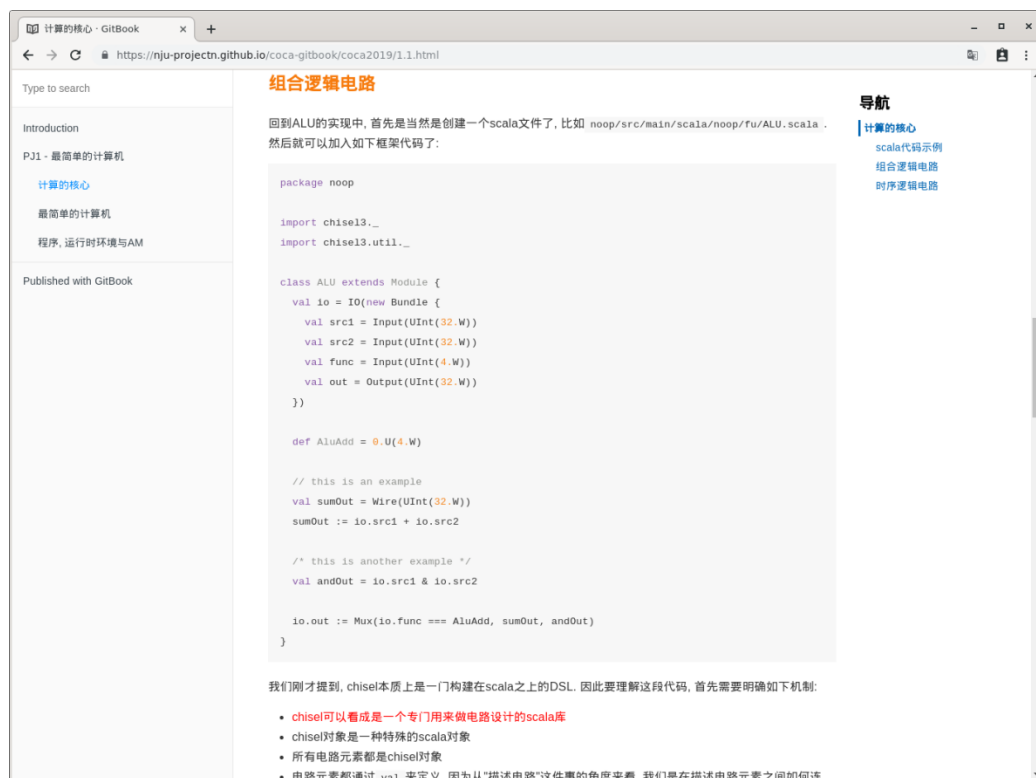


# 一些问题 – 前端

- ▶ Chisel – 学习曲线陡峭
  - 大多硬件开发人员未接触过OOP, FP
  - 需要区分Scala和Chisel的机制
  - 需要了解编译相关的基本知识才能更快地理解和排除错误
- ▶ Rocket Chip
  - 代码机制复杂, 不适合新手阅读
    - ▶ 坚持3天, 从入门到放弃
  - 缺少相关文档, 不易上手
  - 迭代速度过快, 需要花费时间跟进
- ▶ 需要更友好的学习资料

# 国内首个基于chisel的RISC-V教学实验

- ▶ 将由计算所和南京大学联合推出
  - 在线讲义 [nju-projectn.github.io/coca-gitbook](https://nju-projectn.github.io/coca-gitbook)
- ▶ 计算所的一批学生将开展实验并计划流片
  - 带着自己设计的RISC-V芯片毕业, 同时迭代改进实验方案



组合逻辑电路

回到ALU的实现中, 首先是当然是创建一个scala文件了, 比如 `noop/src/main/scala/noop/fu/ALU.scala`. 然后就可以加入如下框架代码了:

```
package noop

import chisel3._
import chisel3.util._

class ALU extends Module {
  val io = IO(new Bundle {
    val src1 = Input(UInt(32.W))
    val src2 = Input(UInt(32.W))
    val func = Input(UInt(4.W))
    val out = Output(UInt(32.W))
  })

  def AluAdd = 0.U(4.W)

  // this is an example
  val sumOut = Wire(UInt(32.W))
  sumOut := io.src1 + io.src2

  /* this is another example */
  val andOut = io.src1 & io.src2

  io.out := Mux(io.func == AluAdd, sumOut, andOut)
}
```

我们刚才提到, chisel本质上是一门构建在scala之上的DSL. 因此要理解这段代码, 首先需要明确如下机制:

- chisel可以看成是一个专门用来做电路设计的scala库
- chisel对象是一种特殊的scala对象
- 所有电路元素都是chisel对象
- 电路元素都通过 `val` 来定义. 因为从“描述电路”这件事的角度来看, 我们是在描述电路元素之间如何连

# 一些问题 - 后端

```
wire _T_588; // @[Monitor.scala 73:14:freechip
wire _T_590; // @[Monitor.scala 73:14:freechip
wire _T_611; // @[Monitor.scala 80:25:freechip
wire [7:0] _T_731; // @[Monitor.scala 85:30:fre
5.8]
wire [7:0] _T_732; // @[Monitor.scala 85:28:fre
6.8]
wire _T_734; // @[Monitor.scala 85:37:freechip
```

- ▶ 生成的Verilog代码可读性弱
  - 大量名字无意义的中间变量, 如\_T\_588
  - 后端团队对路径的理解变得困难[1]
- ▶ 改动少量Chisel代码, 大量无关中间变量命名受到影响
  - \_T\_588 -> \_T\_592
  - 即使小规模迭代, 后端团队难以精准定位某信号/路径[1]
- ▶ Chisel和FIRRTL的优化对生成Verilog代码也有影响
  - 去掉某信号 -> 直接修改Verilog, 而不是改Chisel [1]
- ▶ 缺少EDA工具的支持
- ▶ 仍然有不少问题待解决

# 一些尝试

## 改进中间变量的命名

- `_T_xxx = a + b;` => `a_add_b = a + b;`
- 大大降低了无关中间变量命名受到的影响
- 但编译Labeled RISC-V速度慢了76.6%
  - 94s => 166s

```
assign dsid_eq_ULit2_2_and_fire_mux_size_mux_ULit0 = dsid_eq_ULit2_2_and_fire ? bucketIO_2_size : 32'h0; // @[ControlPlane.scala
assign dsid_eq_ULit2_3 = bucketIO_3_dsid == 5'h2; // @[ControlPlane.scala 158:54:freechips.rocketchip.system.LvNAFPGAConfigzcu10
assign dsid_eq_ULit2_3_and_fire = dsid_eq_ULit2_3 & bucketIO_3_fire; // @[ControlPlane.scala 158:62:freechips.rocketchip.system.
assign dsid_eq_ULit2_3_and_fire_mux_size_mux_ULit0 = dsid_eq_ULit2_3_and_fire ? bucketIO_3_size : 32'h0; // @[ControlPlane.scala
assign dsid_eq_ULit2_and_fire_mux_size_mux_ULit0_add_dsid_eq_ULit2_1_and_fire_mux_size_mux_ULit0_tail_ILit1 = dsid_eq_ULit2_and_
assign dsid_eq_ULit2_and_fire_mux_size_mux_ULit0_add_dsid_eq_ULit2_1_and_fire_mux_size_mux_ULit0_tail_ILit1_add_dsid_eq_ULit2_2_
assign dsid_eq_ULit2_and_fire_mux_size_mux_ULit0_add_dsid_eq_ULit2_1_and_fire_mux_size_mux_ULit0_tail_ILit1_add_dsid_eq_ULit2_2_
assign inc_2 = $signed(bucketParams_2_inc); // @[ControlPlane.scala 161:60:freechips.rocketchip.system.LvNAFPGAConfigzcu102.fir@
assign counter_geq_freq_5_mux_inc_2_mux_SLit0 = counter_geq_freq_4 ? $signed(inc_2) : $signed(32'sh0); // @[ControlPlane.scala 1
assign nToken_add_counter_geq_freq_5_mux_inc_2_mux_SLit0_tail_ILit1 = $signed(bucketState_2_nToken) + $signed(counter_geq_freq_5
```

## 欢迎讨论

- 是否有更好的命名方式?
- 中间变量的可读性是否必要?
  - 如何为汇编代码的每处寄存器引用命名?

# 黄金时代

- ▶ 伯克利研究团队的实践
  - 在5年时间内进行了11次投片[1]
    - ▶ 平均每5~6个月完成一款芯片的设计
  - 与传统的两年一款芯片相比, 设计效率提高了4~5倍
- ▶ 有的小芯片市场周期只有半年[2], 更需要敏捷开发
- ▶ 通过敏捷开发来将芯片设计门槛降低几个数量级, 是有可能实现的
- ▶ 我们即将迎来芯片设计的黄金时代

[1] Lee, et al. An Agile Approach to Building RISC-V Microprocessors. IEEE Micro, 2016, 2(March 2016): 8-20

[2] Global Foundry 团队经验交流

# 总结

- ▶ Chisel代码对项目的敏捷开发提供帮助
  - 减少重复代码, 提升开发效率, 提升代码可读性和易维护性
  - 但和后端配合还有不少问题需要改进
- ▶ 与传统开发对比, 敏捷开发
  - 编码效率提升1个数量级, 达到与传统模式相当甚至更优的性能, 功耗与面积
- ▶ 开源社区还有不少需要改进的地方
  - 我们在努力, 欢迎一同完善开源芯片社区的发展!

**谢谢大家**