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

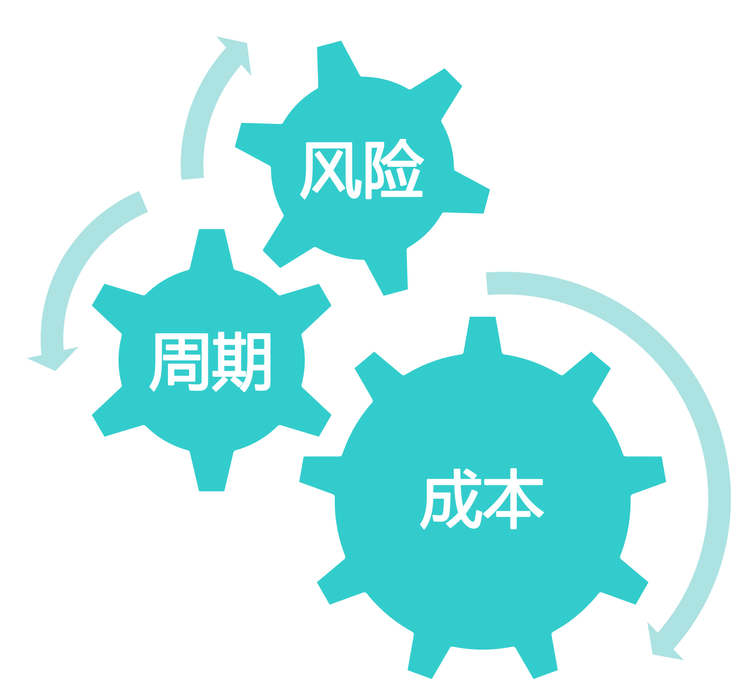
中科院计算所

2018.16@杭州

芯片设计门槛极高

	人员	周期	资金	风险	门槛
传统开发	20~50人	1~2年	上亿元	高	极高

- 还有"死锁"



- 只有少数大公司才玩得起

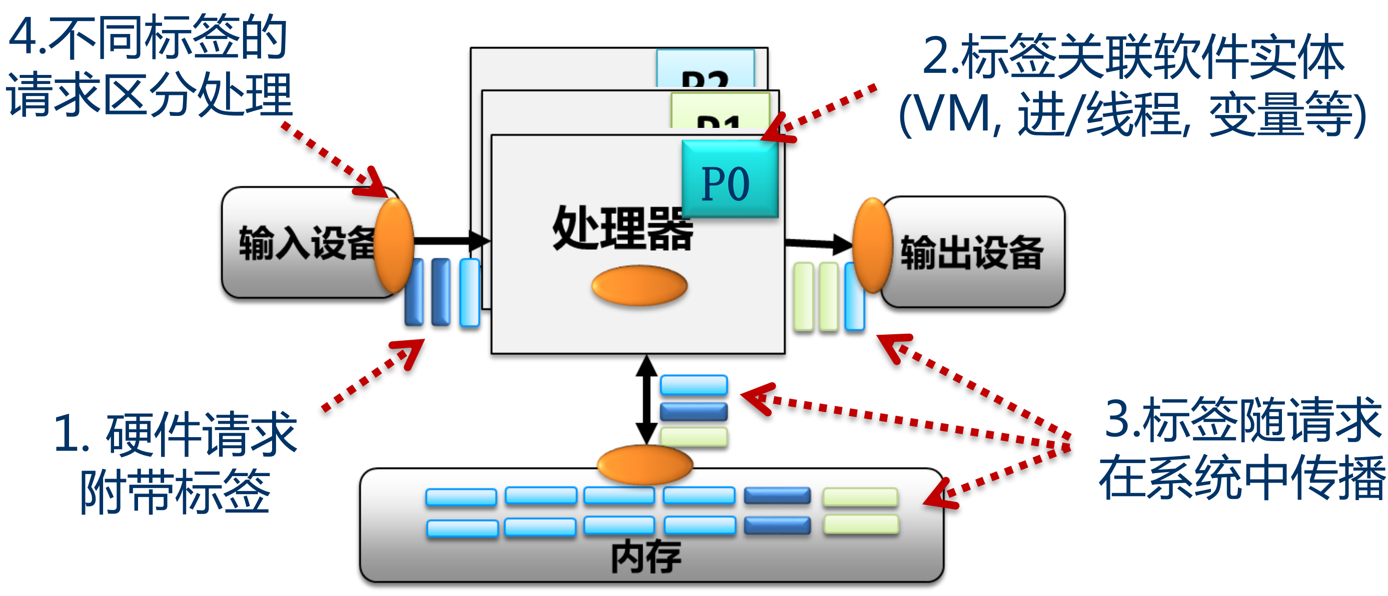
一个解决方案 – 芯片敏捷开发

	人员	周期	资金	风险	门槛
传统开发	20~50人	1~2年	上亿元	高	极高
芯片敏捷开发	3~5人	3~4月	<10万元	中	低

- 给芯片领域注入新生命
- 但这首先需要
  - 一个开放的指令集
  - 一个开源的微结构实现
  - 一门面向敏捷开发的设计语言
- 伯克利的"新三驾马车"
  - RISC-V, Rocket Chip, Chisel

今天分享一些"马车"的驾驶经验

- 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 (PARD), ASPLOS, 2015

开放不活跃的OpenSPARC T1

- 2013年7月选择OpenSPARC T1
  - 基于SPARC V9指令集
  - 2006年开源的工业级设计(听上去就很帅气)
- 但很快就碰壁
  - 开发工具不维护, 需要手动适配老版本的库
  - 代码多文档少, 30万行代码难理解
  - 生态不完善, 难运行真实app
  - 社区不活跃, 提问零回复

- 8C32T -> 1C1T, 尝试半年后弃坑

活跃不开放的MicroBlaze

- 生态和社区由Xilinx维护, 发展健康
- 我们在2016年6月实现了标签化MicroBlaze
  - 但期间也遇到各种困难

原因	举例	解决方法
复杂度	Xilinx Cache模块有6万行代码, 理解困难	投入时间(半年)
代码闭源	AXI Data Width Converter截断标签的传播	绕开
	进程级标签无法在核内实现	
	核心性能一般	无, 但可接受
	不支持多核OS	无
	无法流片	

- 需要选择新指令集
  - 还调研过ARM, 但实在高攀不起

开放又活跃的RISC-V

- 2016年7月, 有了Labeled RISC-V
  - 依托开放活跃的RISC-V及开源设计Rocket Chip

举例	MicroBlaze 解决方法	RISC-V/ Rocket Chip	好处
Cache模块 有6万行代码	投入时间 (半年)	~1千行代码, 3天实现所需功能 (效率提升50x)	实现简单 (敏捷开发相关)
截断标签的传播	绕开	可修改总线模块	修改灵活
标签无法在核内实现		可在核内添加CSR	
核心性能一般	无, 但可接受	无此问题	性能可选
不支持多核OS	无		支持多核
无法流片			允许流片

指令集小结

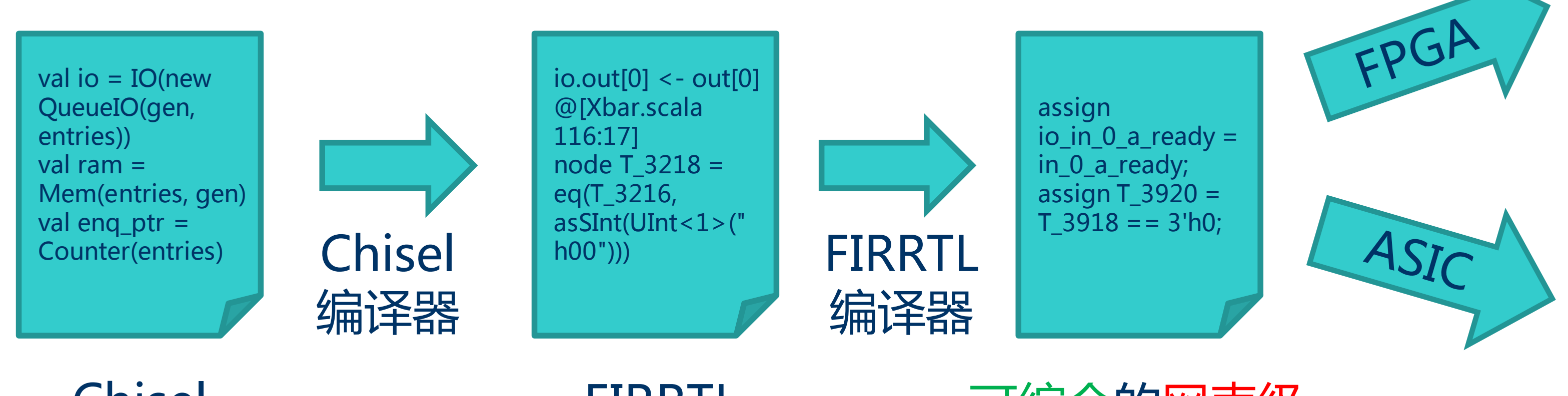
	OpenSPARC T1	MicroBlaze	RISC-V/Rocket Chip
设计开源度	√√√	√	√√√
定制灵活性	√	√	√√√
生态完整性	√	√√	√√√
社区活跃度	√	√√√	√√√

- 不活跃 – 难以运行真实app, 难以寻求帮助
- 不开放 – 难以定制, 部分前沿研究无法开展

- 开放活跃的指令集及其开源的微结构实现, 是降低芯片设计门槛的必要条件

Chisel – 面向敏捷开发的HDL

- 总有需要自己设计开发的时候
- 于是有了面向敏捷开发的新语言Chisel
  - 宗旨 – 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



Verilog编码"基本功" – 连线

- 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
  - 你肯定受不了了
- 别高兴太早, 还会连错呢
  - 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
- 手工连线 = 毫无技术含量的重复性脏活

S00_AXI
S01_AXI
aclk
aresetn
M00_AXI
M01_AXI
M02_AXI

```
.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),
```

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

```
+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)
```

信号整体连接

- Labeled RISC-V中添加标签并传播只要4行代码

```
+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)
```

- 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

- 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
  - 队列元素类型并未事先确定
  - 可用于实例化各种类型元素的队列

```
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 enq_ptr = Counter(entries)
5   val deq_ptr = Counter(entries)
6   // ...

  val TileLinkQueue = Module(new Queue(new TileLinkBundle, 8))
```

基于Scala的元编程 – 模板

```
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)

  val TileLinkQueue = Module(new Queue(new TileLinkBundle, 8))
```

- 在Labeled RISC-V中，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实现队列
  - 需要全局手动增加队列元素的宽度，十分繁琐
- SystemVerilog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

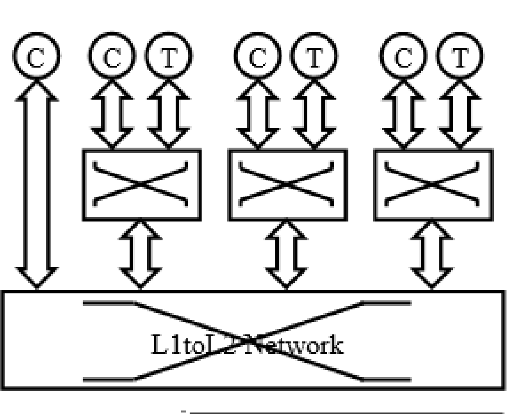
```
val node = AXI4MasterNode(List(edge.master))
val sram = LazyModule(new AXI4RAM(...))
sram.node := AXI4Buffer() := AXI4Fragmenter() := AXI4Delayer(0, 150) := node
```

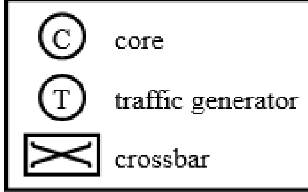
Master
Delayer
Fragmenter
Buffer
RAM

- 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

L1toL2Network

- 可通过10行Chisel代码实现

```
1 val cachedPorts = cachedPortsBeforeGenerator.take(1) ++
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HSL
  - 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

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

- 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

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	135.410	134.445 (-0.71%)	141.423 (+4.44%)
功耗/W	0.770	0.747 (-2.99%)	0.712 (-7.53%)
LUT逻辑	5685	6422 (+12.96%)	3007 (-47.11%)
LUT存储	1796	1264 (-29.62%)	1232 (-31.40%)
FF	4266	3638 (-14.72%)	754 (-82.33%)
有效代码/行	618	470 (-23.95%)	282 (-54.37%)
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

- Rocket Chip
  - 迭代速度过快 – 需要花费时间跟进
  - 没有整理稳定的发型版本 – 新手不易获得
- Chisel
  - 学习曲线陡峭 – 大多硬件开发人员未接触过OOP, FP
  - 缺少EDA工具的支持 – 需阅读网表级Verilog代码
- 开源EDA工具也很重要

- 但我们相信, 这个方向是对的
  - 给充分时间可以解决
  - DARPA牵头的电子复兴计划[1]

[1] https://www.darpa.mil/work-with-us/electronics-resurgence-initiative

硬件敏捷开发宣言

- 传统观点 – 硬件设计无法借鉴软件敏捷开发
  - 因为硬软件开发的特点有较大差异
- 伯克利观点 – 为了弥补这些差异, 更需要借鉴, 来提升硬件开发效率

- 伯克利团队提出"硬件敏捷开发宣言"[1]
  - 优先开发未完成但容易改造的原型, 而非构建功能齐全却难以扩展的模型
  - 优先组建灵活协作的团队, 而非强调各司其职的分工
  - 优先完善工具和生成器, 而非改进独立的设计实例
  - 优先拥抱变化, 而非遵循计划

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

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
- 最近睿思芯科也公布了第一款芯片[2]
  - 从零开始到设计完成, 只用了7个月 (复用RISC-V项目)
  - 是传统芯片设计效率的3倍
- 有的小芯片的市场周期只有半年, 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] https://riscv.org/2018/11/sohu-article-pygmy-soc/

总结

- 芯片敏捷开发的必要条件
  - 开放又活跃的指令集
  - 及其开源的微结构设计
- 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
- 与传统开发对比
  - 敏捷开发的编码效率提升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

谢谢大家