# Why Python

Chisel是Scala的一个库，Scala是一种多范式的编程语言，语法特别繁琐，入门以及精通难度比c++有过之无不及。

Python 有以下优点：

1. 语法简洁自然，符号简单易懂，学习门槛低
2. 动态类型，同时支持类型注释，即方便编程又便于阅读
3. 解释运行，不做编译优化，支持非常动态的元编程
4. 纯面向对象，编程范式统一，同时支持一定的函数式编程技巧

# Example

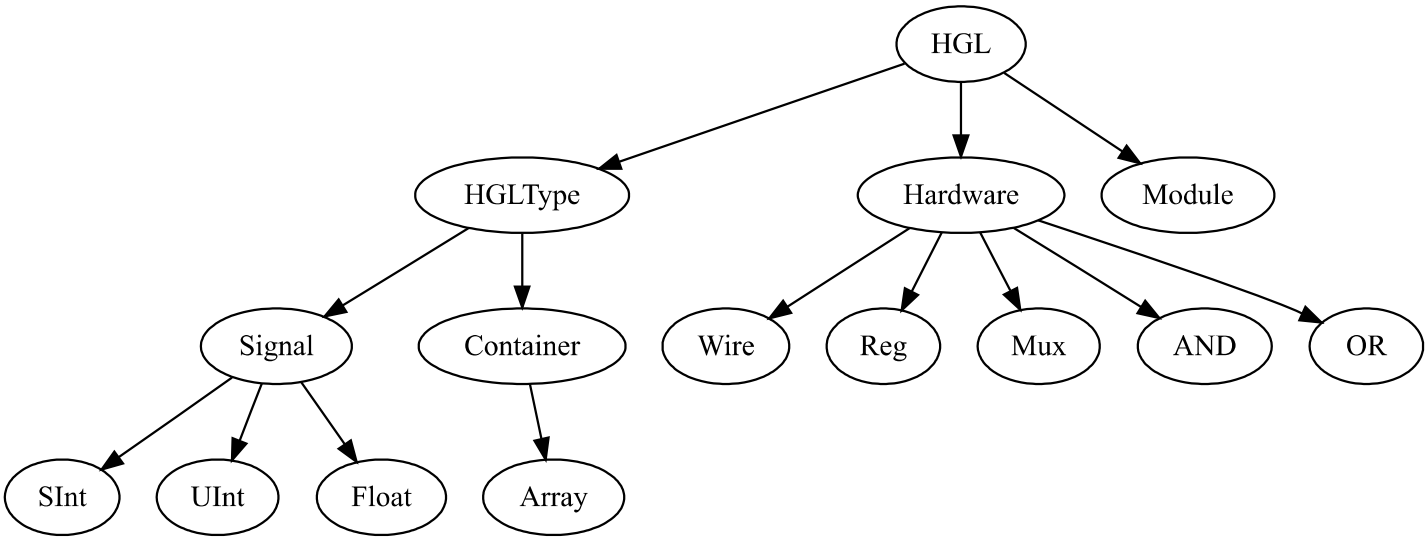
## chisel

```
// Chisel Code, but pass in a parameter to set widths of ports
class PassthroughGenerator(width: Int) extends Module {
  val io = IO(new Bundle {
    val in = Input(UInt(width.W))
    val out = Output(UInt(width.W))
  })
  io.out := io.in
}

// Let's now generate modules with different widths
println(getVerilog(new PassthroughGenerator(10)))
println(getVerilog(new PassthroughGenerator(20)))

test(new Passthrough()) { c =>
    c.io.in.poke(0.U)     // Set our input to value 0
    c.io.out.expect(0.U)  // Assert that the output correctly has 0
    c.io.in.poke(1.U)     // Set our input to value 1
    c.io.out.expect(1.U)  // Assert that the output correctly has 1
    c.io.in.poke(2.U)     // Set our input to value 2
    c.io.out.expect(2.U)  // Assert that the output correctly has 2
}
println("SUCCESS!!") // Scala Code: if we get here, our tests passed!
```

## pyhgl

```
from pyhgl import *

@module PassthroughGenerator(self, width):
    io = Array(
        i = Input(UInt(w=width)),
        o = Output(UInt(w=width))
    )
    io.o <== io.i


@testbench Tester(DUT):
    track(DUT.o)
    @initial task1:
        for i in range(10):
            DUT.i.setVal(i)
            yield 1
            print(f"{i} ==> {DUT.o.getVal(radix='b')}")

DUT = PassthroughGenerator(111)

verilog = Verilog(DUT)
verilog.emit('./test.v')

tester = Tester(DUT)
tester.run(1000)
```

# Types




- **HGLType**
  - 对 python 运算符进行重载，可以相互之间进行运算
- **Signal**
  - 指向硬件的某个端口
  - input can be `Signal` or `Gate` type
  - stores output value and width ( `int` type)
  - if reg, requires clock and/or reset signal
- **hardware**
  - 多输入多输出，输入 hardware，输出具体value
  - 无时钟（组合）或一个时钟（时序）
  - 如果有时钟，可选 reset 信号
  - 如果是组合逻辑，每个输出依赖于所有输入
  - 如果是时序逻辑，指定不依赖任何输入的输出，和不影响任何输出的输入
  - 并行化仿真带来的限制
    - 假设所有 clock/reset 都是同步复位/无复位寄存器

- Array
  - 类似 numpy，但是是树形结构，支持名字
  - 自动将运算 mapping 到元素
  - 如果维度不满足一一对应，自动 broadcast
  - 支持多种切片

### Array

```
b = Bundle(
    x = 1,
    y = 2,
    z = Bundle(
        m = ['a',2.0,'like'],
        n = [-1, 'xxx','yyy']
    ),
    zz = Bundle(
        m = ['a',2.0,'like'],
        n = [2, 'xxx','yyy']
    )
)
print(b[['z','zz'], :, -1])
b[['z','zz'], :, -1] = [['c','d']]
print(b)
```

## Global Parameter Tree

```
@conf Global(up):
    clocks = [Clock()._rename('clock50MHz')]
    resets = [UInt()._rename('reset_n')]
    @conf Top(up):
        width = 8

        @conf Top_A(up):
            width1 = 7
            width2 = up.width * 2
```

- 每个 module 都可以从外部指定 parameters
- 访问方式：全局变量 Conf, or 'self' ?
- 树形结构，lazy 求值
- 指定匹配的id : `@conf(Top.*)`
- 默认每层只在该层生效，不会进入下一层，但是参数会继承
- 可以指定某个 config 函数一直生效；以及设置不继承
- 一个 module 可以匹配多个

```
@module Top_A_X:
    x = UInt(5, w = Conf.width)
    y = UInt(5, w = Conf.width1)
    z = x & y
    out = ~ z
```

## Dynamic dispatch

- 全局参数有三个总是继承的属性：
  - `Conf.clocks: List[Clock]`
  - `Conf.resets: List[Signal]`
  - `Conf.F: Dispatcher`
- Dispatcher 对一元，二元函数进行动态派发，根据函数名和函数前两个 positional arg 的类型查找对应的函数
- 需要事先注册
- 可以通过注册 `Array` , `Array,Any` , `Any,Array` 实现函数自动向量化

用法

```
Conf.F.call('Xor', UInt('11111'))  # === Conf.F.Xor(UInt('11111'))
Conf.F.call('Xor', UInt('11111'), SInt('11000'))
Conf.F.Xor(1,2,3,4,5)  # 多个参数只根据前两个类型查找
```

# 弱类型

- 在需要的时候对 python 内置类型进行转换

```
8  -> UInt(8)
'111' -> UInt('111')
(2, '01', UInt('0000')) -> Cat(...)
[{'a':1, 'b':[1,2,3]},{}] -> Array(...)
```

- 字符串转换规则

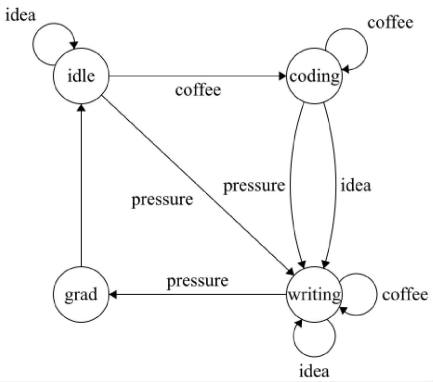
```
'u16:b1111'

'fp32: 11.1 km'

类型 位宽 : 前缀 值 单位
```

## 类型转换

```
UInt(a: Signal)  # 将 a 解释为 UInt

a = UInt('1111').to(Float)  # 生成新的电路，  需要实现注册转换函数
```

# 抽象

## condition




```
  val idle :: coding :: writing :: grad :: Nil = Enum(4)
  io.nextState := idle
  when (io.state === idle) {
    when      (io.coffee) { io.nextState := coding }
    .elsewhen (io.idea) { io.nextState := idle }
    .elsewhen (io.pressure) { io.nextState := writing }
  } .elsewhen (io.state === coding) {
    when      (io.coffee) { io.nextState := coding }
    .elsewhen (io.idea || io.pressure) { io.nextState := writing }
  } .elsewhen (io.state === writing) {
    when      (io.coffee || io.idea) { io.nextState := writing }
    .elsewhen (io.pressure) { io.nextState := grad }
  }
```

```
# first is default
state = State('idle', 'coding', 'writing', 'grad')

when state.at('idle'):
    when io.coffee:
        state.goto('coding')
    elsewhen io.idea:
        state.wait()
    elsewhen io.pressure:
        state.goto('writing')
elsewhen state.at('coding'):
    when io.coffee:
        state.wait()
    elsewhen [io.idea, io.pressure]:
        state.goto('writing')
elsewhen state.at('writing'):
    when [io.coffee, io.idea] :
        state.wait()
    elsewhen io.pressure:
        state.goto('grad')
otherwise:
    state.init()
```

bluespec

```
// 代码路径: src/3.SPIWriter/SPIWriter.bsv （部分）
module mkSPIWriter (SPIWriter);        // BSV SPI 发送（可综合！！），模块名称为 mkSPIWriter
   Reg#(bit) ss <- mkReg(1'b1);
   Reg#(bit) sck <- mkReg(1'b1);
   Reg#(bit) mosi <- mkReg(1'b1);
   Reg#(Bit#(8)) wdata <- mkReg(8'h0);
   Reg#(int) cnt <- mkReg(7);          // cnt 的复位值为 7

   FSM spiFsm <- mkFSM (               // mkFSM 是一个状态机自动生成器，能根据顺序模型生成状态机 spiFsm
      seq                              // seq...endseq 描述一个顺序模型，其中的每个语句占用1个时钟周期
         ss <= 1'b0;                   // ss 拉低
         while (cnt>=0) seq            // while 循环，cnt 从 7 递减到 0，共8次
            action                     // action...endaction 内的语句在同一周期内执行，即原子操作。
               sck <= 1'b0;            // sck 拉低
               mosi <= wdata[cnt];     // mosi 依次产生串行 bit
            endaction
            action                     // action...endaction 内的语句在同一周期内执行，即原子操作。
               sck <= 1'b1;            // sck 拉高
               cnt <= cnt - 1;         // cnt 每次循环都递减
            endaction
         endseq
         mosi <= 1'b1;                 // mosi 拉高
         ss <= 1'b1;                   // ss 拉高，发送结束
         cnt <= 7;                     // cnt 置为 7，保证下次 while 循环仍然正常循环 8 次
      endseq );                        // 顺序模型结束

   method Action write(Bit#(8) data);  // 当外部需要发送 SPI 时，调用此 method。参数 data 是待发送的字节
      wdata <= data;
      spiFsm.start();                  // 试图启动状态机 spiFsm
   endmethod

   method Bit#(3) spi = {ss,sck,mosi}; // 该 method 用于将 SPI 信号引出到模块外部
endmodule
```

## module

模块级抽象

- input, output 自动推导无需指定
- inout 必须指定，所有 inout 都会被引到 toplevel

- 每次调用都会生成一个 module 实例，如果多个 module 最终生成 verilog 代码相同，只生成一个 verilog module
- module 是 hardware 的集合

# 常用函数

```
a = UInt('1111')
# new wire from a
b = a.wire(w = 8)
# new reg from a
a.reg(w = 8, clock = clk50MHz, clock_edge = False, reset = rst_n, reset_level = False)

# reg has the same type of a, but self input
Reg(a)
# === a.reg()
RegNext(a)
# reg aa;
# aa <= cond ? a : aa ;
aa = RegCond(cond, a)
```

```
Cat(a, '111')  # {a, UInt(111)}
a ** 4         # {a,a,a,a}
# 切片，包括头尾
a[31:0]        # a[31:0]
a[0:31]        # reversed
a[31:0:1]    # Array(a[31], a[30], ...)
a.split(
  rs1 = '11111',
  rs2 = '11111',
  opcode = [7:0]
)
Array(
  rs1 = a[31:27],
  rs2 = a[26:22] ,
  opcode = a[7:0]
)
```

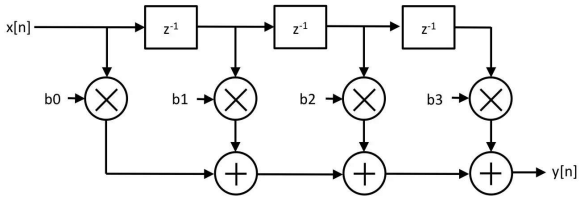
## 运算符

```
LOGICNOT                '!'
LOGICAND                '&&'
LOGICOR                 '||'
DOUBLEPLUS              '++'
SLEFTSHIFT              '<<<'
SRIGHTSHIFT             '>>>'
```

## InOut

```
b = InOut("4'b1111")
# sel1, sel2 需要互斥
b.when(sel1, UInt("4'b0001"))
b.when(sel2, UInt("4'b1111"))
```

## FIR Filter



```
class MyManyDynamicElementVecFir(length: Int) extends Module {
  val io = IO(new Bundle {
    val in = Input(UInt(8.W))
    val valid = Input(Bool())
    val out = Output(UInt(8.W))
    val consts = Input(Vec(length, UInt(8.W)))
  })

  // Such concision! You'll learn what all this means later.
  val taps = Seq(io.in) ++ Seq.fill(io.consts.length - 1)(RegInit(0.U(8.W)))
  taps.zip(taps.tail).foreach { case (a, b) => when (io.valid) { b := a } }

  io.out := taps.zip(io.consts).map { case (a, b) => a * b }.reduce(_ + _)
}

visualize(() => new MyManyDynamicElementVecFir(4))
```

```
module MyManyDynamicElementVecFir(self, length:int):

    x = UInt('8:0')
    valid = UInt()
    consts = [UInt('8:0') for _ in range(length)]

    x_delay = [x]
    for _ in range(length-1):
      x_delay.append(RegCond(valid, x_delay[-1]))

    y = Dot(x_delay, consts)
```

## Simulation