


Chisel — an Agile Hardware Description Language

An Introduction to Chisel

Shixin CHEN

February 25, 2022

FPGA Group, Hardware Innovation Center

Outline

Preliminary

Instances of Chisel

Chisel for Super Resolution on FPGA

Q & A

Thanks for listening!

Preliminary

Integrated Circuit Design Flow

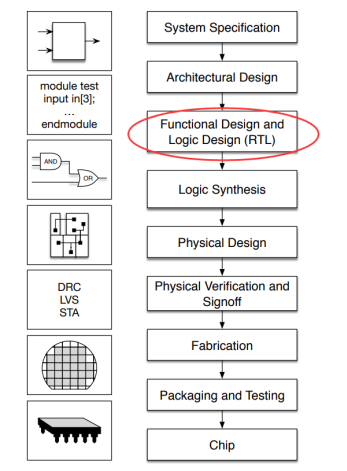


Figure 1: IC Design Flow

(a) Chisel in Reality

(b) Chisel in IC Design

Figure 2: Chisel

Hardware Description Languages (HDL)[1]

- Register Transfer Language (RTL)
  - **Verilog, VHDL**
- Meta HDL and Transpilers

C++	SystemC, VisualHDL
Python	PyRTL, Pyrope
Java	jhdl
Scala	Chisel, SpinalHDL

- High-Level Synthesis (HLS)
  - HLS
  - Legup

[1] `https://github.com/drom/awesome-hdl`

Drawbacks of Verilog

Verilog

The language has been dominant in IC design for more than 30 years.

- A classical but old-fashioned language
  - Incompatible with contemporary software development styles
- Unfriendly to design efficiency
  - Low-level abstractions
  - Time-consuming development cycles

Why is Scala? & Why is Chisel?

Chisel

Constructing **H**ardware **I**n **S**cala **E**mbedded **L**anguage

- **Scala**, an excellent host language used as Domain Specific Language (DSL)
  - Functional programming
  - Extensibility
- **Chisel**, embedded in Scala, facilitating digital design
  - Abstraction between Verilog and HLS
  - Higher efficiency

How can Chise help?

Motivation

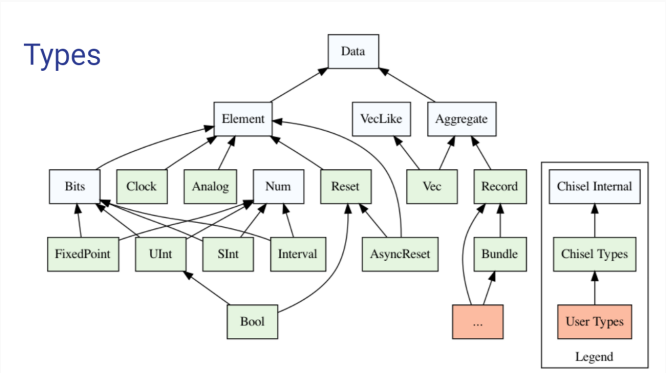
Through **Circuit Generators**, developers can leverage the hard work of design experts and raise the level of design abstraction to meet the demand of evolution in IC design, especially in SoC and Processors.

- Using **Modern Programming Styles** to productively arrange IC elements
  - Object-oriented programming
  - Functional programming
  - Parameterized types
- Using *Blackbox* to work with Verilog compatibly

Instances of Chisel

Chisel Types Tree

- Providing a library of class and objects to describe hardware flexibly
  - Structure: Module, Bundle, App, Object ...
  - Hardware: Reg, Wire, Mux, SRAM ...
  - Types: UInt, SInt, Vec, Clock ...



A Sample: Verilog Generated from Chisel

```
//define a class for parameterized adder
class ParamAddern(n: Int) extends Module {
  val io = IO (new Bundle{
    val a = Input(UInt(n.W))
    val b = Input(UInt(n.W))
    val out = Output(UInt((n+1).W))
  })
  io.out:=io.a+io.b
}
//instantiate
val add8 = Module(new ParamAdder(8))
add8.a :=a
add8.b :=b
out := add8.out
```

```
module ParamAddern(
  input         clock,
  input         reset,
  input  [7:0] io_in_a,
  input  [7:0] io_in_b,
  output [8:0] io_out
);
 assign io_out=io_in_a+io_in_b;
endmodule
//instantiate
ParamAddern add8(
    .clock    (sys_clk),
    .reset    (sys_rst),
    .io_in_a  (a      ),
    .io_in_b  (b      ),
    .io_out   (out    )
);
```

Combinations in Chisel

```
1  class PORT_B(n:Int) extends Bundle{
2  //Bundle Defines the combinations of
   ↪  Modules
3    var b0=Input(UInt(n.W))
4    var b1=Input(UInt(n.W))
5    var b3=Output(UInt(n.W))
6  }
7
8  class PORT_C(n:Int) extends Bundle{
9    var c0=Input(UInt(n.W))
10   var c1=Output(UInt(n.W))
11 }
12
13 class PORT_A(n:Int) extends Bundle {
14   //Exchange the Input and Output
15   val interface_b = Flipped(new PORT_B)
16   val interface_c = Flipped(new PORT_C)
17   val a0=Input(UInt(n.W))
18   val a1=Output(UInt(n.W))
19 }
```

Here are the *combinations* of *Modules*. With Chisel, implementation is efficient and time-saving.

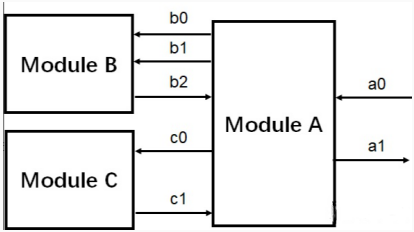


Figure 3: Modules Combinations

Combinations in Chisel

```
20  class Module_A extends Module{
21    val io=IO(new (PORT_A))
22    ...//Operations
23  }
24  ...//Module_B,Module_C
25  class PortReuse extends Module {
26    val io=IO(new Bundle{
27      val in = Input(UInt(4.W))
28      val out=Output(UInt(4.W))
29    })
30    val ma=Module(new Module_A())
31    val mb=Module(new Module_B())
32    val mc=Module(new Module_C())
33
34    ma.io.interface_b<>mb.io//Match the Input and the Corresponding Output
35    ma.io.interface_c<>mc.io
36    ma.a0.io :=io.in
37    io.out:=ma.io.a1
38  }
```

Object-oriented programming in Chisel

```
1  class Payload extends Bundle {
2      val data = UInt (16.W)
3      val flag = Bool ()
4  }
5  class Port[T <: Data ](private val dt: T) extends Bundle {
6      val address = UInt (8.W)
7      val data = dt. cloneType
8  }
9  class NocRouter2[T <: Data ](dt: T, n: Int) extends Module {
10     val io =IO(new Bundle {
11     val inPort = Input(Vec(n, dt))
12     val outPort = Output(Vec(n, dt))
13 })
14 // Route the payload according to the address
15 // ...
16 val router = Module(new NocRouter2 (new Port(new Payload), 4))}
```

Functional Abstraction in Chisel

```
1  val (cnt,cnt_valid)=Counter(io.input_valid,4)
2  //Counter is a module provided by Chisel
3  //We can define our own Module generators as well
4  class Counter_pulse extends Module{
5    val io=IO(new Bundle{
6      val valid=Input(Bool())
7      val goal_num=Input(UInt(8.W))
8      val pulse=Input(Bool())
9      val cnt=Output(UInt(8.W))
10     val out_valid=Output(Bool())
11   })
12   ...//Counter operations
13 }
14   object Counter_pulse{
15   def apply(valid:Bool,goal_num:UInt,pulse:Bool):(UInt,Bool)={
16      val inst=Module(new Counter_pulse())
17       inst.io.valid:=valid
18       inst.io.goal_num:=goal_num
19       inst.io.pulse:=pulse
20       (inst.io.cnt,inst.io.out_valid)
21     }
22  }
23 val (cnt2,cnt2_valid)=Counter_pulse(io.input_valid,6.U,cnt_valid)
```

Chisel for Super Resolution on FPGA

Conventional Convolution vs. Winograd Algorithm

Conventional Convolution Algorithm

The algorithm, consuming 9 DSPs in each convolutional computation, is demanding for DSPs on resource-limited FPGA.

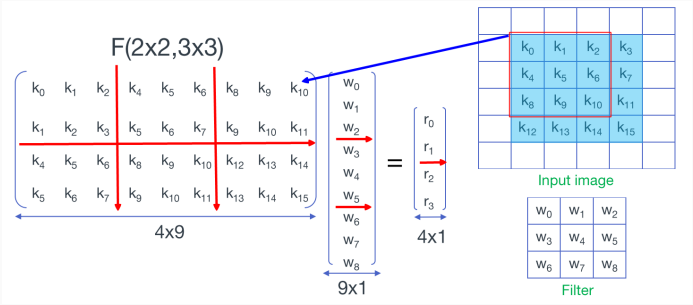


Figure 4: Classical Convolution based Matrix-Multiplying

Conventional Convolution vs. Winograd Algorithm

Winograd Algorithm

The algorithm, consuming only 4 DSPs in each computation on FPGA, works as a replacement of convolution operator.

$$S = A^T \left[\left(GgG^T \right) \odot \left(B^T dB \right) \right] A \tag{1}$$

$$g = \begin{bmatrix} wt_{00} & wt_{01} & wt_{02} \\ wt_{10} & wt_{11} & wt_{12} \\ wt_{20} & wt_{21} & wt_{22} \end{bmatrix} \tag{2}$$

$$d = \begin{bmatrix} x_{00} & x_{01} & x_{02} & x_{03} \\ x_{10} & x_{11} & x_{12} & x_{13} \\ x_{20} & x_{21} & x_{22} & x_{23} \\ x_{30} & x_{31} & x_{32} & x_{33} \end{bmatrix} \tag{3}$$

$$B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \tag{4}$$

$$G = \begin{bmatrix} 1 & 0 & 0 \\ 0.5 & 0.5 & 0.5 \\ 0.5 & -0.5 & 0.5 \\ 0 & 0 & 1 \end{bmatrix} \tag{5}$$

$$A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix} \tag{6}$$

Winograd Algorithm Flow

$$S = A^T \left[\left(GgG^T \right) \odot \left(B^T dB \right) \right] A \tag{7}$$

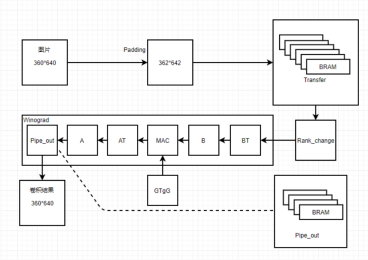


Figure 5: Data Flow of the Winograd

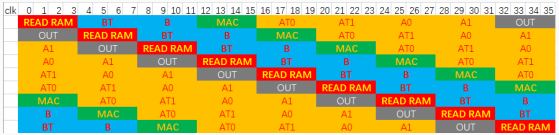
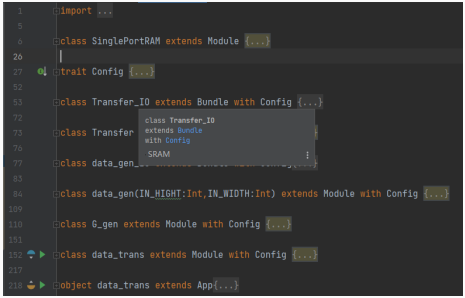
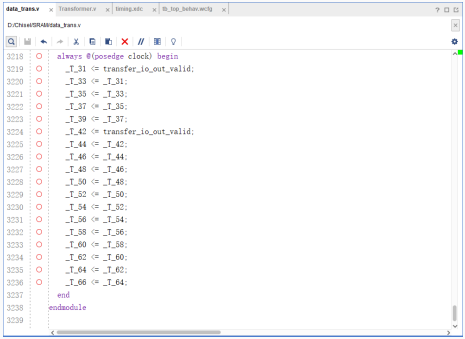


Figure 6: pipline of the Winograd

Comparison: Length of Code



(a) Chisel Code (218 LOC)



(b) Verilog Code (3239 LOC)

Figure 7: Chisel-generated Verilog

Comparison: Computing Resources

Compromise

DSPs is the bottleneck of computing resources while FFs and LUTs are sufficient in most FPGA platforms.
It is a rewarding strategy to use Winograd Algorithm on FPGA.

Case	Conventional Convolution	Winograd Algorithm
DSPs	9	4
LUTs	334	650
FFs	1110	1500

Table 1: Resources of Conventional Conv. Wino. Algo.

Chisel: Trade-off between Verilog and HLS

- More efficient than Verilog
- More controllable than HLS

Case	Verilog	Chisel
DSPs	9	9
LUTs	180	229
FFs	1156	1523

Table 2: Resources of Conventional Convolution Implementation

Chisel Re-Thinking

- Chisel is more agile in SoC and Processors design, while less suitable in ASIC than Verilog
- Testing Chisel code with cycle-accurate tools is still not easy
- Chisel->FIRRTL->**Verilog**: Verilog is still the basis of IC Design

Q & A

Thanks for listening!