



Chisel — an Agile Hardware Description Language

An Introduction to Chisel

Shixin CHEN

November 6, 2021

Dept. ESE, Nanjing University

Dept. CSE, Chinese University of Hong Kong

Outline

Preliminary

Instances of Chisel

Chisel for Super Resolution on FPGA

Q & A

Preliminary

Integrated Circuit Design Flow

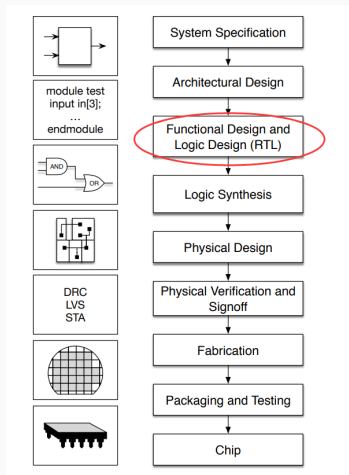


Figure 1: IC Design Flow



(a) Chisel in Reality



(b) Chisel in IC Design

Figure 2: Chisel

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 iterations
 - Low-level abstractions
 - Time-consuming development cycles

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>

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)
 - Extendibility
 - Simplicity
- ▶ **Chisel**, embedded in Scala, facilitating digital design
 - Higher-level abstractions
 - Time-saving and efficiency

How can Chise help?

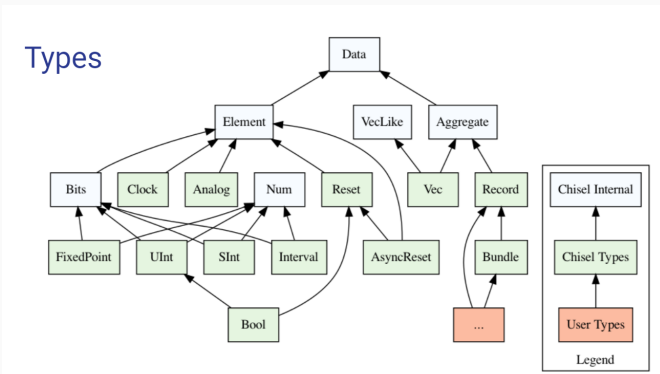
Motivation

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.

- ▶ Using modern programming styles to productively put IC elements, like Mux, Counter, and RAM together
 - Parameterized types
 - Object-oriented programming
 - Functional programming

Chisel Types Tree

- Providing a library of class and objects to representing hardware
 - Types: UInt, SInt, Vec, Bundle, Clock
 - Hardware: Reg, Wire, IO, Mux
 - Structure: Module, Blackbox



Instances of Chisel

Parameterized Adder in Chisel:

```
1 class ParamAdder(n: Int) extends Module {
2   val io = IO (new Bundle{
3     val a = Input(UInt(n.W))
4     val b = Input(UInt(n.W))
5     val out = Output(UInt((n+1).W))
6   })
7   io.out:=io.a+io.b
8 }
9 //instantiate
10 val add8 = Module(new ParamAdder(8))
11 val add16 = Module(new ParamAdder(16))
```

```
1 module Adder(
2   input      clock,
3   input      reset,
4   input [8:0] io_in_a,
5   input [8:0] io_in_b,
6   output [9:0] io_out
7 );
8   assign io_out=io_in_a+io_in_b;
9 endmodule
10 //instantiate
11 Adder add(
12   .clock      (sys_clk),
13   .reset      (sys_rst),
14   .io_in_a    (a      ),
15   .io_in_b    (b      ),
16   .io_out     (out     )
17 );
```

Combinations in Chisel

```
1  class PORT_B extends Bundle{
2    //Bundle Defines the combinations of
   ↪  Modules
3    var b0=Input(UInt(2.W))
4    var b1=Input(UInt(2.W))
5    var b3=Output(UInt(2.W))
6  }
7
8  class PORT_C extends Bundle{
9    var c0=Input(UInt(2.W))
10   var c1=Output(UInt(2.W))
11 }
12
13 class PORT_A 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(4.W))
18   val a1=Output(UInt(4.W))
19 }
```

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

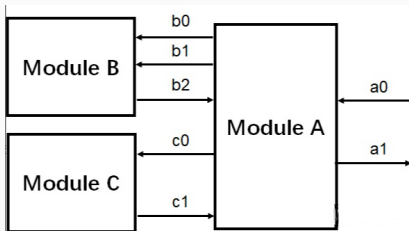


Figure 3: Modules Combinations

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()).io
31    val mb=Module(new Module_B()).io
32    val mc=Module(new Module_C()).io
33
34    ma.interface_b<>mb//Match the Input and the Corresponding Output
35    ma.interface_c<>mc
36    ma.a0 :=io.in
37    io.out:=ma.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), 2))}
```

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.

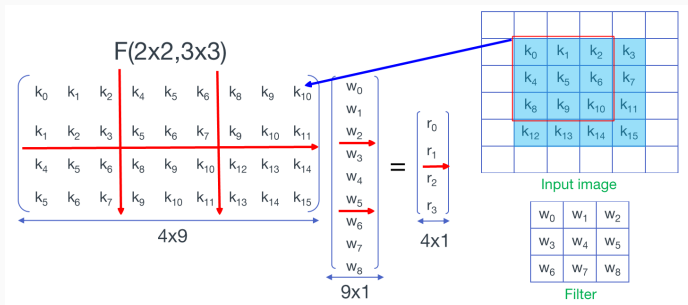


Figure 4: Classical Convolution based Matrix-Multiplying

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.

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

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

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

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

$$G = \begin{bmatrix} 1 & 0 & 0 \\ 0.5 & 0.5 & 0.5 \\ 0.5 & -0.5 & 0.5 \\ 0 & 0 & 1 \end{bmatrix} \quad (5)$$

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

Winograd Algorithm Flow

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

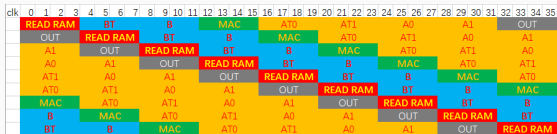
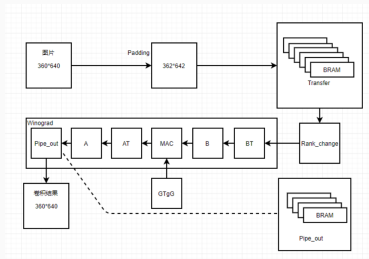


Figure 5: Data Flow of the Winograd

Figure 6: pipeline of the Winograd

Comparison: Length of Code

```
1  import ...
5
6  class SinglePortRAM extends Module {...}
26
27  trait Config {...}
52
53  class Transfer_IO extends Bundle with Config {...}
72
73  class Transfer
    extends Bundle
    with Config
    SRAM
76
77  class data_gen
83
84  class data_gen(IN_HEIGHT: Int, IN_WIDTH: Int) extends Module with Config {...}
109
110  class G_gen extends Module with Config {...}
151
152  class data_trans extends Module with Config {...}
217
218  object data_trans extends App{...}
```

(a) Chisel Code (218 LOC)

```
data_trans.v | Transformer.v | timing.edc | tb_top_behav.wcfg
D:\ChiselSRAM\data_trans.v

3218  always @(posedge clock) begin
3219  _T_31 <= transfer_io_out_valid;
3220  _T_33 <= _T_31;
3221  _T_35 <= _T_33;
3222  _T_37 <= _T_35;
3223  _T_39 <= _T_37;
3224  _T_42 <= transfer_io_out_valid;
3225  _T_44 <= _T_42;
3226  _T_46 <= _T_44;
3227  _T_48 <= _T_46;
3228  _T_50 <= _T_48;
3229  _T_52 <= _T_50;
3230  _T_54 <= _T_52;
3231  _T_56 <= _T_54;
3232  _T_58 <= _T_56;
3233  _T_60 <= _T_58;
3234  _T_62 <= _T_60;
3235  _T_64 <= _T_62;
3236  _T_66 <= _T_64;
3237  end
3238  endmodule
3239
```

(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

It is unavoidable for Chisel to consume more LUTs and FFs in the synthesis of Module than Verilog, because higher abstraction will take extra resources.

- ▶ More agile than Verilog
- ▶ More controllable than Chisel

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

Table 2: Resources of Conventional Convolution Implementation

- ▶ Utilize Chisel to enclose Verilog-based modules as hardware generators.
- ▶ Formulate the design space of hardware generators, which accept provided parameters to produce RTL flexibly.

Q & A
