Introducing Chisel, a Hardware Construction Language

Bruno FERRES
(bruno.ferres@inria.fr)

28 March, 2023

Plan

Introducing Myself[1]

- **Engineering degree** (Ensimag 2015–2018) — *Embedded Systems and Software* (SLE)
- **Master degree** (UGA 2017–2018) — *CyberSecurity*
- **PhD candidate** (2018–2022) — TIMA (Grenoble)
  "Leveraging Hardware Construction Languages for Flexible Design Space Exploration on FPGA"
  Supervisors: Frédéric Rousseau, Olivier Muller
- **Post-doctoral researcher** (2022–2023) — LIP (Lyon)
  "Using Model Checking for Electrical Rule Checking of Integrated Circuits at Transistor Level"
  Collaborators: Matthieu Moy, Ludovic Henrio, Gabriel Radanne, Pascal Raymond, Oussama Oulkaid, Mehdi Khosravian

[1] Contact: www.ferres.me / bruno.ferres@inria.fr

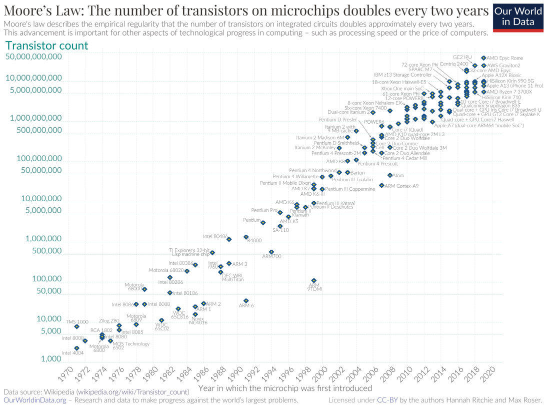
Introducing Myself[1]

- **Engineering degree** (Ensimag 2015–2018) — *Embedded Systems and Software* (SLE)
- **Master degree** (UGA 2017–2018) — *CyberSecurity*
- **PhD candidate** (2018–2022) — TIMA (Grenoble)
  "Leveraging Hardware Construction Languages for Flexible Design Space Exploration on FPGA"
  Supervisors: Frédéric Rousseau, Olivier Muller
- **Post-doctoral researcher** (2022–2023) — LIP (Lyon)
  "Using Model Checking for Electrical Rule Checking of Integrated Circuits at Transistor Level"
  Collaborators: Matthieu Moy, Ludovic Henrio, Gabriel Radanne, Pascal Raymond, Oussama Oulkaid, Mehdi Khosravian

Interests: Using high-level methods and tools for hardware design and verification.

[1] Contact: www.ferres.me / bruno.ferres@inria.fr

Digital Design



Moore's Law: The number of transistors on microchips doubles every two years

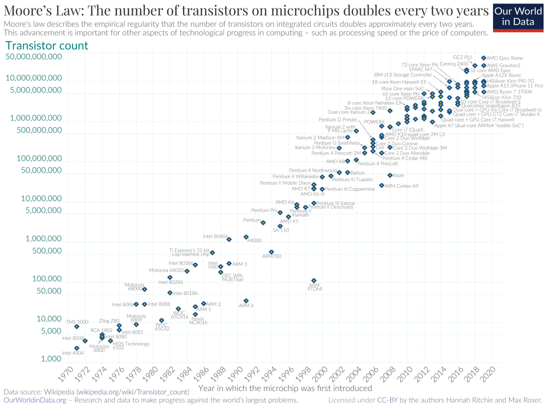
Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.

Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)

OurWorldinData.org – Research and data to make progress against the world's largest problems.

Licensed under CC-BY by the authors Hannah Ritchie and Max Roser.

Digital Design



How to design circuits efficiently?

Digital Design at Archi 2023

Overview of Design Methodologies Lectures

1. Monday (16h–17h30): **Introduction to Digital Design** (Steven Derrien)
2. Tuesday (9h–12h30): **Introduction to HLS** (Steven Derrien)
3. Tuesday (14h–17h30): **Introduction to** `Chisel`

Digital Design at Archi 2023

Overview of Design Methodologies Lectures

1. Monday (16h–17h30): **Introduction to Digital Design** (Steven Derrien)
2. Tuesday (9h–12h30): **Introduction to HLS** (Steven Derrien)
3. Tuesday (14h–17h30): **Introduction to** `Chisel`

Introduction to `Chisel`

- `Chisel` is a *Hardware Construction Language*. . .

Digital Design at Archi 2023

Overview of Design Methodologies Lectures

1. Monday (16h–17h30): **Introduction to Digital Design** (Steven Derrien)
2. Tuesday (9h–12h30): **Introduction to HLS** (Steven Derrien)
3. Tuesday (14h–17h30): **Introduction to** `Chisel`

Introduction to `Chisel`

- `Chisel` is a *Hardware Construction Language*...
  - ... an alternative to HDL, HLS, DSL!

Digital Design at Archi 2023

Overview of Design Methodologies Lectures

1. Monday (16h–17h30): **Introduction to Digital Design** (Steven Derrien)
2. Tuesday (9h–12h30): **Introduction to HLS** (Steven Derrien)
3. Tuesday (14h–17h30): **Introduction to** `Chisel`

Introduction to `Chisel`

- `Chisel` is a *Hardware Construction Language*. . .
  - . . . an alternative to HDL, HLS, DSL!
- **Lecture** (1h30): using `Chisel`: why, when and how?

Digital Design at Archi 2023

Overview of Design Methodologies Lectures

1. Monday (16h–17h30): **Introduction to Digital Design** (Steven Derrien)
2. Tuesday (9h–12h30): **Introduction to HLS** (Steven Derrien)
3. Tuesday (14h–17h30): **Introduction to `Chisel`**

Introduction to `Chisel`

- `Chisel` is a *Hardware Construction Language*. . .
  - . . . an alternative to HDL, HLS, DSL!
- **Lecture** (1h30): using `Chisel`: why, when and how?
- **Practical Work** (1h30): implementing dot product with `Chisel`

Objectives of the Lecture

- introduce **Hardware Construction Languages**
  - introduce `Chisel`

Objectives of the Lecture

- introduce **Hardware Construction Languages**
  - introduce `Chisel`
- give the basics to use `Chisel`
  - practical works are always better

Objectives of the Lecture

- introduce **Hardware Construction Languages**
  - introduce `Chisel`
- give the basics to use `Chisel`
  - practical works are always better
- provide some keys to **choose a design paradigm**
  - when should you use HLS or `Chisel`?

Overview of the Lecture

Main References

- Martin Schoeberl. *Digital design with chisel*. Kindle Direct Publishing, 2019
- Jean Bruant. "Abstracting Hardware Architectures for Agile Design of High-performance Applications on FPGA". PhD thesis. Université Grenoble Alpes, 2023
- Bruno Ferres. "Leveraging Hardware Construction Languages for Flexible Design Space Exploration on FPGA". PhD thesis. Université Grenoble Alpes, 2022

Plan

Existing Design Methodologies

- Hardware Description Languages: VHDL, (System)Verilog
- High Level Synthesis: `vivado HLS`, `altera`, `LegUp`, `AUGH`
- Domain Specific Languages: P4 (network), DFiant (dataflow)

Existing Design Methodologies

- Hardware Description Languages: VHDL, (System)Verilog
  - **Limitations**: expressivity, reusability, modularity
- High Level Synthesis: `vivado HLS`, `altera`, `LegUp`, `AUGH`
- Domain Specific Languages: P4 (network), DFiant (dataflow)

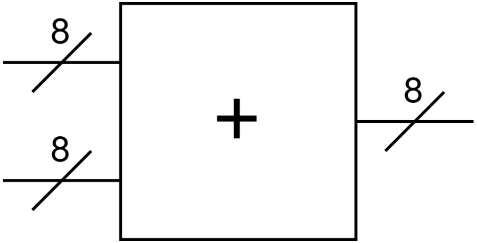
Existing Design Methodologies

- Hardware Description Languages: VHDL, (System)Verilog
  - **Limitations**: expressivity, reusability, modularity
- High Level Synthesis: `vivado HLS`, `altera`, `LegUp`, `AUGH`
  - **Limitations**: implementation details, portability, no real semantics
- Domain Specific Languages: P4 (network), DFiant (dataflow)

Existing Design Methodologies

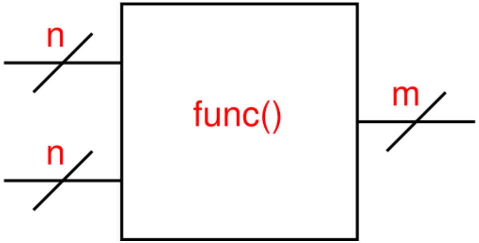
- Hardware Description Languages: VHDL, (System)Verilog
  - **Limitations**: expressivity, reusability, modularity
- High Level Synthesis: `vivado HLS`, `altera`, `LegUp`, `AUGH`
  - **Limitations**: implementation details, portability, no real semantics
- Domain Specific Languages: P4 (network), DFiant (dataflow)
  - **Limitations**: limited to specific domains, rely on (vendor) provided IPs

Motivating Example



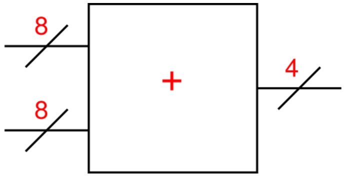
Simple 8 bit adder

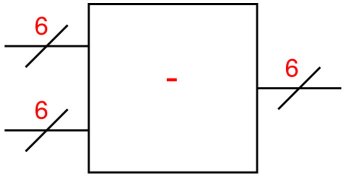
Motivating Example



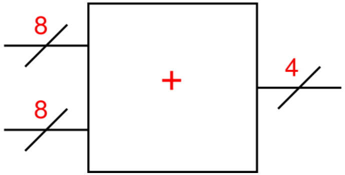
Parametrized functional block

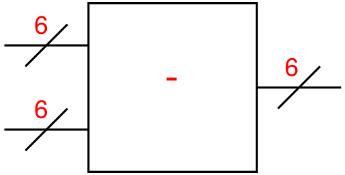
Motivating Example





Motivating Example

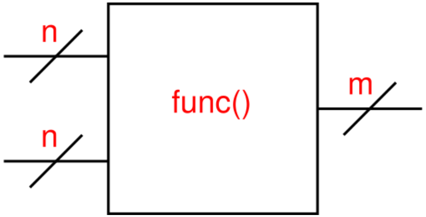




```
module adder (
    input [7:0] op1,
    input [7:0] op2,
    output [3:0] out
);
    wire [8:0] tmp;
    tmp = op1 + op2
    out = tmp > 15 ?
        15 : tmp[3:0] ;
endmodule
```

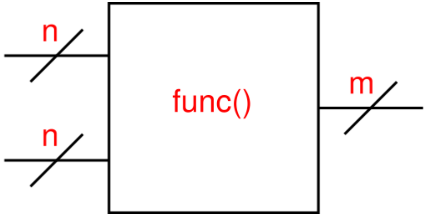
```
module sub (
    input [5:0] op1,
    input [5:0] op2,
    output [5:0] out
);
    out = op1 - op2;
endmodule
```

Motivating Example



```
class CustomBlock[T <: Data](
    inputBitwidth:  Int,
    outputBitwidth: Int,
    func:           (T, T) => T
) extends Module { ... }
```

Motivating Example



```
class CustomBlock[T <: Data](
    inputBitwidth:  Int,
    outputBitwidth: Int,
    func:           (T, T) => T
) extends Module { ... }

val adder   = new CustomBlock[UInt](8, 4, _ + _)
val sub     = new CustomBlock[UInt](6, 6, _ - _)
```

Some Requirement for High-Level Generation Features

Coping with limitations from other paradigms

- **generic usage**
  - no domain/vendor specific IPs

Some Requirement for High-Level Generation Features

Coping with limitations from other paradigms

- **generic usage**
  - no domain/vendor specific IPs
- **reusability** and **adaptability**

Some Requirement for High-Level Generation Features

Coping with limitations from other paradigms

- **generic usage**
  - no domain/vendor specific IPs
- **reusability** and **adaptability**
- fine control of **implementation details**
  - in a **portable**, **understandable** and **consistent** way!

Some Requirement for High-Level Generation Features

Coping with limitations from other paradigms

- **generic usage**
  - no domain/vendor specific IPs
- **reusability** and **adaptability**
- fine control of **implementation details**
  - in a **portable**, **understandable** and **consistent** way!
- **expressivity** and **modularity**

Some Requirement for High-Level Generation Features

Coping with limitations from other paradigms

- **generic usage**
  - no domain/vendor specific IPs
- **reusability** and **adaptability**
- fine control of **implementation details**
  - in a **portable**, **understandable** and **consistent** way!
- **expressivity** and **modularity**

Parallel with software programming

“Recent” programming paradigms improve the coding experience:

Some Requirement for High-Level Generation Features

Coping with limitations from other paradigms

- **generic usage**
  - no domain/vendor specific IPs
- **reusability** and **adaptability**
- fine control of **implementation details**
  - in a **portable**, **understandable** and **consistent** way!
- **expressivity** and **modularity**

Parallel with software programming

"Recent" programming paradigms improve the coding experience:

- **object-oriented programming**: modularity and reusability

Some Requirement for High-Level Generation Features

Coping with limitations from other paradigms

- **generic usage**
  - no domain/vendor specific IPs
- **reusability** and **adaptability**
- fine control of **implementation details**
  - in a **portable**, **understandable** and **consistent** way!
- **expressivity** and **modularity**

Parallel with software programming

"Recent" programming paradigms improve the coding experience:

- **object-oriented programming**: modularity and reusability
- **functional programming**: high-order parameters

Some Requirement for High-Level Generation Features

Coping with limitations from other paradigms

- **generic usage**
  - no domain/vendor specific IPs
- **reusability** and **adaptability**
- fine control of **implementation details**
  - in a **portable**, **understandable** and **consistent** way!
- **expressivity** and **modularity**

Parallel with software programming

"Recent" programming paradigms improve the coding experience:

- **object-oriented programming**: modularity and reusability
- **functional programming**: high-order parameters
- **reflexivity/introspection**: meta-programming

Some Requirement for High-Level Generation Features

Ultimately, should find a compromise between:

- **performance**: *control implementation details*
- **usability**: *ease describing new circuits*

Some Requirement for High-Level Generation Features

Ultimately, should find a compromise between:

- **performance**: *control implementation details*
- **usability**: *ease describing new circuits*
- **agility**: *existing code evolution*

`Chisel`: *Constructing Hardware in a Scala Embedded Language*

Existing HCLs (a selection)

- **python**: `Migen`, `pyMTL`, `MyHDL`
- **Haskell**: `Cλash`, `Lava`
- **OCaml**: `hardcaml`
- **Scala**: `Chisel`, `SpinalHDL`

Chisel: *Constructing Hardware in a Scala Embedded Language*

Existing HCLs (a selection)

- **python**: `Migen`, `pyMTL`, `MyHDL`
- **Haskell**: `C`λ`ash`, `Lava`
- **OCaml**: `hardcaml`
- **Scala**: `Chisel`, `SpinalHDL`

Why choose `Chisel`?

- support from both **academic** and **industrial** worlds
- **powerful language**
- **supportive community**
- documented **infrastructure**

Plan

Chisel Community

https://www.chisel-lang.org/community.html

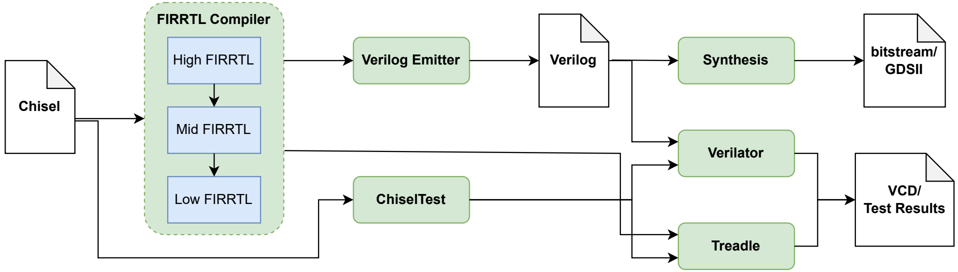
Chisel Community

https://www.chisel-lang.org/community.html

Chisel Community

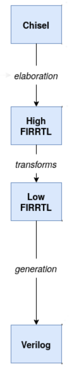
https://www.chisel-lang.org/community.html

Overview of the Generation Flow

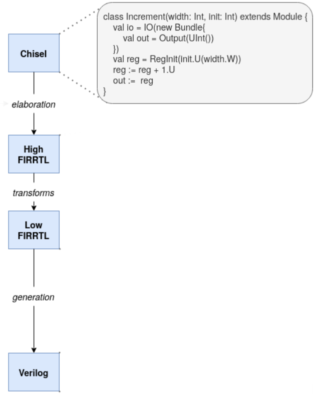


Andrew Dobis, Kevin Laeufer, Hans Jakob Damsgaard, Tjark Petersen, Kasper Juul Hesse Rasmussen, Enrico Tolotto, Simon Thye Andersen, Richard Lin, and Martin Schoeberl. "Verification of Chisel Hardware Designs with ChiselVerify". In: *Microprocessors and Microsystems* (2023)

Overview of the Generation Flow

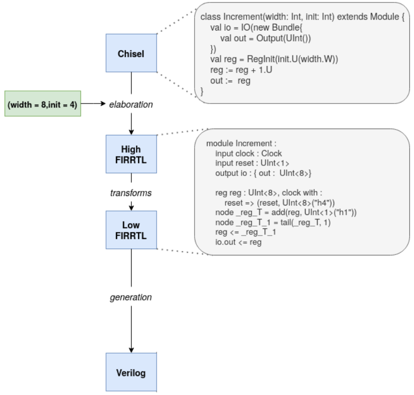


Overview of the Generation Flow



```
class Increment(width: Int, init: Int) extends Module {
  val io = IO(new Bundle{
    val out = Output(UInt())
  })
  val reg = RegInit(init.U(width.W))
  reg := reg + 1.U
  out :=  reg
}
```

Overview of the Generation Flow

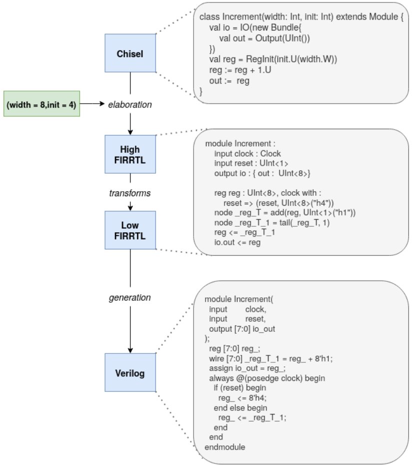


```
class Increment(width: Int, init: Int) extends Module {
   val io = IO(new Bundle{
      val out = Output(UInt())
   })
   val reg = RegInit(init.U(width.W))
   reg := reg + 1.U
   out :=  reg
}
```

```
module Increment :
   input clock : Clock
   input reset : UInt<1>
   output io : { out :  UInt<8>}

   reg reg : UInt<8>, clock with :
      reset => (reset, UInt<8>("h4"))
   node _reg_T = add(reg, UInt<1>("h1"))
   node _reg_T_1 = tail(_reg_T, 1)
   reg <= _reg_T_1
   io.out <= reg
```

Overview of the Generation Flow



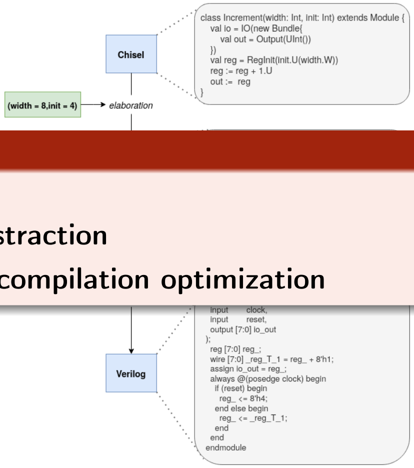
```
class Increment(width: Int, init: Int) extends Module {
  val io = IO(new Bundle{
    val out = Output(UInt())
  })
  val reg = RegInit(init.U(width.W))
  reg := reg + 1.U
  out :=  reg
}
```

```
module Increment :
  input clock : Clock
  input reset : UInt<1>
  output io : { out :  UInt<8>}

  reg reg : UInt<8>, clock with :
    reset => (reset, UInt<8>("h4"))
  node _reg_T = add(reg, UInt<1>("h1"))
  node _reg_T_1 = tail(_reg_T, 1)
  reg <= _reg_T_1
  io.out <= reg
```

```
module Increment(
  input       clock,
  input       reset,
  output [7:0] io_out
);
  reg [7:0] reg_;
  wire [7:0] _reg_T_1 = reg_ + 8'h1;
  assign io_out = reg_;
  always @(posedge clock) begin
    if (reset) begin
      reg_ <= 8'h4;
    end else begin
      reg_ <= _reg_T_1;
    end
  end
endmodule
```

Overview of the Generation Flow



```
class Increment(width: Int, init: Int) extends Module {
  val io = IO(new Bundle{
    val out = Output(UInt())
  })
  val reg = RegInit(init.U(width.W))
  reg := reg + 1.U
  out := reg
}
```

```
  input       clock,
  input       reset,
  output [7:0] io_out
);
  reg [7:0] reg_;
  wire [7:0] _reg_T_1 = reg_ + 8'h1;
  assign io_out = reg_;
  always @(posedge clock) begin
    if (reset) begin
      reg_ <= 8'h4;
    end else begin
      reg_ <= _reg_T_1;
    end
  end
endmodule
```

Particularity of the flow

- **generates** `verilog`
- multiple **levels of abstraction**
- custom **elaboration/compilation optimization**

Frontend: The `Chisel` Compiler

Scala meta language

- `Chisel` is a `scala` **library** ...

Frontend: The Chisel Compiler

Scala meta language

- Chisel is a scala **library** . . .
  - . . . can be extended

Frontend: The `Chisel` Compiler

`Scala` meta language

- `Chisel` is a `scala` **library** ...
  - ... can be extended
- code is **executed** to **elaborate** hardware

Frontend: The Chisel Compiler

Scala meta language

- Chisel is a scala **library** ...
  - ... can be extended
- code is **executed** to **elaborate** hardware

```
class Replication(nRep: Int, width: Int)
extends Module {
  val tpe = UInt(width.W)
  val io = IO(new Bundle{
    val i = Input(tpe)
    val o = Output(Vec(nRep, tpe))
  })
  for (i <- 0 until nRep) {
    io.o(i) := io.i
  }
}
```

Frontend: The `Chisel` Compiler

Scala meta language

- `Chisel` is a `scala` **library** . . .
  - . . . can be extended
- code is **executed** to **elaborate** hardware

```scala
class Replication(nRep: Int, width: Int)
extends Module {
  val tpe = UInt(width.W)
  val io = IO(new Bundle{
    val i = Input(tpe)
    val o = Output(Vec(nRep, tpe))
  })
  for (i <- 0 until nRep) {
    io.o(i) := io.i
  }
}
```

$$\xRightarrow[\text{width} = 8]{\text{nRep} = 4}$$

Frontend: The Chisel Compiler

Scala meta language

- Chisel is a scala **library** ...
  - ... can be extended
- code is **executed** to **elaborate** hardware

```
class Replication(nRep: Int, width: Int)
extends Module {
  val tpe = UInt(width.W)
  val io = IO(new Bundle{
    val i = Input(tpe)
    val o = Output(Vec(nRep, tpe))
  })
  for (i <- 0 until nRep) {
    io.o(i) := io.i
  }
}
```

$$\xrightarrow[\text{width} = 8]{\text{nRep} = 4}$$

```
module Replication(...);
  assign io_o_0 = io_i;
  assign io_o_1 = io_i;
  assign io_o_2 = io_i;
  assign io_o_3 = io_i;
endmodule
```

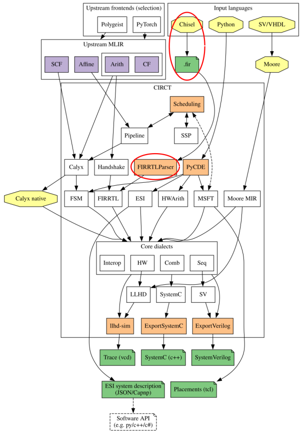
FIRRTL: *Flexible Intermediate Representation for RTL*

Defining an Intermediate Representation

- **well specified** IR for circuits
- multiple **abstraction levels**
  - *high* → *mid* → *low* → `verilog`
- custom **transforms**
  - optimization, estimations, monitoring

Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, et al. "Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations". In: *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2017

FIRRTL: *Flexible Intermediate Representation for RTL*



CIRCT: Circuit IR Compilers and Tools

Backend: Hardware Generation

Register-Transfer Level generation

- straightforward translation: *Low FIRRTL* $\mapsto$ `verilog`
  - simple `verilog` (no `parameters`)
- code generation: **how to track bugs to `Chisel` code?**

Backend: Hardware Generation

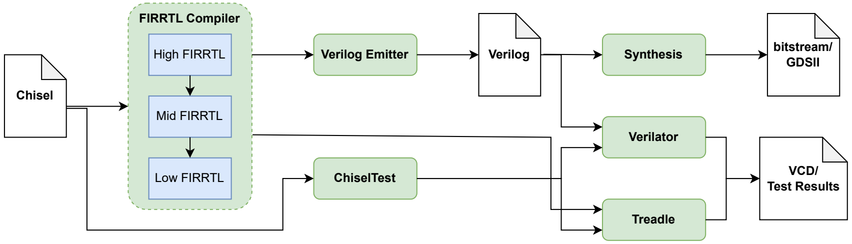
Register-Transfer Level generation

- straightforward translation: *Low FIRRTL* ↦ `verilog`
  - simple `verilog` (no `parameters`)
- code generation: **how to track bugs to `Chisel` code?**

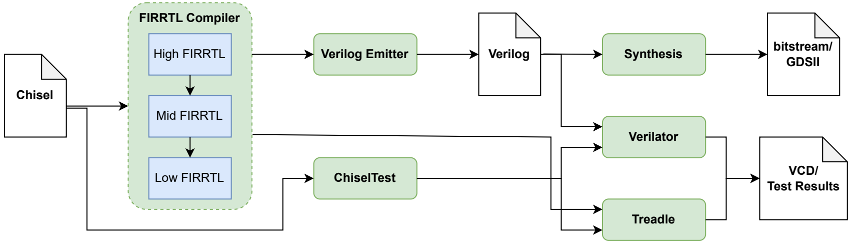
```
module Replication(
  input         clock,
  input         reset,
  input   [7:0] io_i,
  output  [7:0] io_o_0,
  output  [7:0] io_o_1,
  output  [7:0] io_o_2,
  output  [7:0] io_o_3
);
```

```
  assign io_o_0 = io_i; // @[Dummy.scala 13:18]
  assign io_o_1 = io_i; // @[Dummy.scala 13:18]
  assign io_o_2 = io_i; // @[Dummy.scala 13:18]
  assign io_o_3 = io_i; // @[Dummy.scala 13:18]
endmodule
```

Simulation Environment



Simulation Environment



Simulation

- treadle simulates **FIRRTL** directly
- verilator compiles verilog to machine code
- can be plugged to **any** verilog-**based simulator**

Summary of Chisel/FIRRTL Ecosystem

Chisel/FIRRTL infrastructure

- **compiler-like** architecture
  - **frontend**: execute Chiselcode to generate FIRRTL
  - **transforms**: modify FIRRTL (DCE, combinatorial checks, . . .)
  - **simulation**: integrated *cycle-accurate* simulators
  - **backend**: `verilog` emission
- *open-source* and *community-based*

Summary of Chisel/FIRRTL Ecosystem

Chisel/FIRRTL infrastructure

- **compiler-like** architecture
  - **frontend**: execute `Chisel`code to generate FIRRTL
  - **transforms**: modify FIRRTL (DCE, combinatorial checks, . . .)
  - **simulation**: integrated *cycle-accurate* simulators
  - **backend**: `verilog` emission
- *open-source* and *community-based*

Some initiatives around Chisel/FIRRTL compiler

- RISC-V generators with **parameter negociation**: *rocket-chip*, *BOOM*
- CIRCT integration
- `Chisel`-based estimators for DSE: `QECE` *my thesis*
- `sv2chisel`: porting legacy code to `Chisel` *Jean Bruant*

Plan

Chisel Syntax

Basic syntax

- Chisel is a library that **must be imported**
  - `import chisel3._`
- must **respect** `scala` **syntax** ($\simeq$ `java` syntax)
- **main rule**: partition between **software** and **hardware world**

Chisel Syntax

Basic syntax

- `Chisel` is a library that **must be imported**
  - `import chisel3._`
- must **respect** `scala` **syntax** (≃ `java` syntax)
- **main rule**: partition between **software** and **hardware world**

```
if (swVar == true) {
    io.out := io.in
} else {
    io.out := 0.U
}
```

Generate one branch only

Chisel Syntax

Basic syntax

- `Chisel` is a library that **must be imported**
  - `import chisel3._`
- must **respect** `scala` **syntax** ($\simeq$ `java` syntax)
- **main rule**: partition between **software** and **hardware world**

```
if (swVar == true) {
    io.out := io.in
} else {
    io.out := 0.U
}
```

Generate one branch only

```
when (hwVar === true.B) {
    io.out := io.in
}.otherwise {
    io.out := 0.U
}
```

Generate a Mux structure

Chisel Syntax

Basic syntax

- `Chisel` is a library that **must be imported**
  - `import chisel3._`
- must **respect** `scala` **syntax** ($\simeq$ `java` syntax)
- **main rule**: partition between **software** and **hardware world**

```
if (swVar == true) {
    io.out := io.in
} else {
    io.out := 0.U
}
```

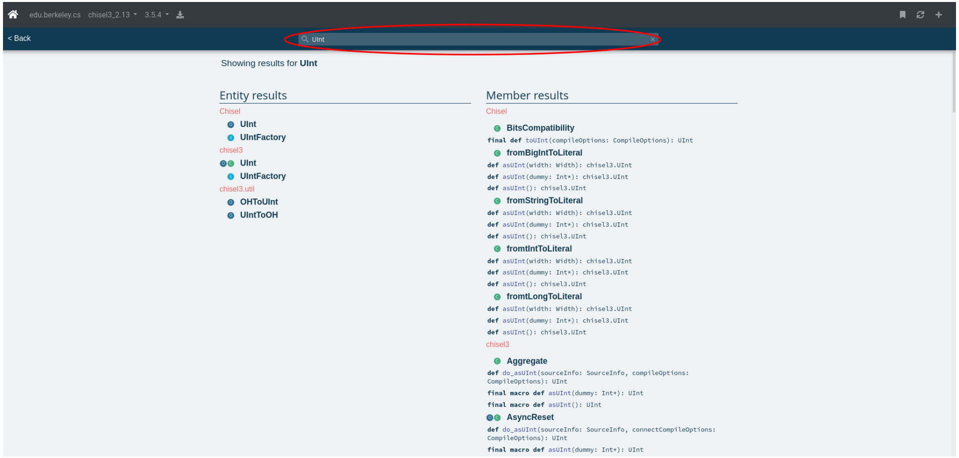
Generate one branch only

```
when (hwVar === true.B) {
    io.out := io.in
}.otherwise {
    io.out := 0.U
}
```

Generate a Mux structure

Must be wrapped in a Module anyway!

Application Programming Interface[2]



Showing results for **UInt**

Entity results

Chisel
- UInt
- UIntFactory

chisel3
- UInt
- UIntFactory

chisel3.util
- OHToUInt
- UIntToOH

Member results

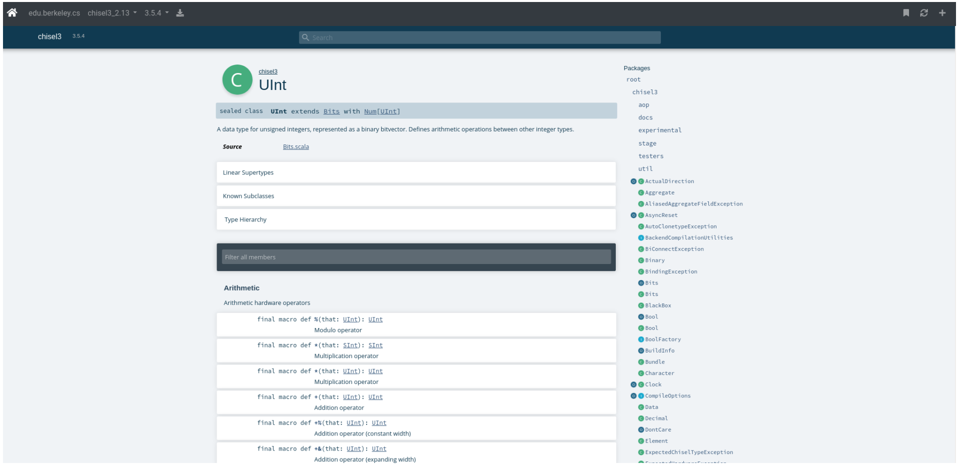
Chisel

BitsCompatibility
```
final def toUInt(compileOptions: CompileOptions): UInt
```
fromBigIntToLiteral
```
def asUInt(width: Width): chisel3.UInt
def asUInt(dummy: Int*): chisel3.UInt
def asUInt(): chisel3.UInt
```
fromStringToLiteral
```
def asUInt(width: Width): chisel3.UInt
def asUInt(dummy: Int*): chisel3.UInt
def asUInt(): chisel3.UInt
```
fromtIntToLiteral
```
def asUInt(width: Width): chisel3.UInt
def asUInt(dummy: Int*): chisel3.UInt
def asUInt(): chisel3.UInt
```
fromtLongToLiteral
```
def asUInt(width: Width): chisel3.UInt
def asUInt(dummy: Int*): chisel3.UInt
def asUInt(): chisel3.UInt
```
chisel3

Aggregate
```
def do_asUInt(sourceInfo: SourceInfo, compileOptions:
CompileOptions): UInt
final macro def asUInt(dummy: Int*): UInt
final macro def asUInt(): UInt
```
AsyncReset
```
def do_asUInt(sourceInfo: SourceInfo, connectCompileOptions:
CompileOptions): UInt
final macro def asUInt(dummy: Int*): UInt
```

[2] https://javadoc.io/doc/edu.berkeley.cs/chisel3_2.13/3.5.4/index.html

Application Programming Interface[2]



chisel3
UInt

```
sealed class UInt extends Bits with Num[UInt]
```

A data type for unsigned integers, represented as a binary bitvector. Defines arithmetic operations between other integer types.

Source Bits.scala

Linear Supertypes

Known Subclasses

Type Hierarchy

Filter all members

Arithmetic

Arithmetic hardware operators

```
final macro def %(that: UInt): UInt
```
Modulo operator

```
final macro def *(that: SInt): SInt
```
Multiplication operator

```
final macro def *(that: UInt): UInt
```
Multiplication operator

```
final macro def +(that: UInt): UInt
```
Addition operator

```
final macro def +%(that: UInt): UInt
```
Addition operator (constant width)

```
final macro def +&(that: UInt): UInt
```
Addition operator (expanding width)

Packages

```
root
  chisel3
    aop
    docs
    experimental
    stage
    testers
    util
    ActualDirection
    Aggregate
    AliasedAggregateFieldException
    AsyncReset
    AutoClonetypeException
    BackendCompilationUtilities
    BiConnectException
    Binary
    BindingException
    Bits
    Bits
    BlackBox
    Bool
    Bool
    BoolFactory
    BuildInfo
    Bundle
    Character
    Clock
    CompileOptions
    Data
    Decimal
    DontCare
    Element
    ExpectedChiselTypeException
```

[2]https://javadoc.io/doc/edu.berkeley.cs/chisel3_2.13/3.5.4/index.html

Combinational Logic

```
class Adder extends Module {
  val io = IO(new Bundle {
    val in1 = Input(UInt(8.W))
    val in2 = Input(UInt(8.W))
    val out = Output(UInt(8.W))
  })
  io.out := io.in1 + io.in2
}
```

Combinational Logic

```scala
class Adder extends Module {
  val io = IO(new Bundle {
    val in1 = Input(UInt(8.W))
    val in2 = Input(UInt(8.W))
    val out = Output(UInt(8.W))
  })
  io.out := io.in1 + io.in2
}
```

```verilog
module Adder(
  input        clock,
  input        reset,
  input  [7:0] io_in1,
  input  [7:0] io_in2,
  output [7:0] io_out
);
  assign io_out = io_in1 + io_in2;
endmodule
```

Combinational Logic

```
class ALU extends Module {
  val io = IO(new Bundle {
    val in1 = Input(UInt(8.W))
    val in2 = Input(UInt(8.W))
    val op  = Input(UInt(3.W))
    val out = Output(UInt(8.W))
  })
  when (io.op === 0.U) {
    io.out := io.in1 + io.in2
  }.elsewhen (io.op === 1.U) {
    io.out := io.in1 - io.in2
  }.elsewhen (io.op === 2.U) {
    io.out := io.in1 & io.in2
  }.otherwise {
    io.out := io.in1 ^ io.in2
  }
}
```

Combinational Logic

```scala
class ALU extends Module {
  val io = IO(new Bundle {
    val in1 = Input(UInt(8.W))
    val in2 = Input(UInt(8.W))
    val op  = Input(UInt(3.W))
    val out = Output(UInt(8.W))
  })
  when (io.op === 0.U) {
    io.out := io.in1 + io.in2
  }.elsewhen (io.op === 1.U) {
    io.out := io.in1 - io.in2
  }.elsewhen (io.op === 2.U) {
    io.out := io.in1 & io.in2
  }.otherwise {
    io.out := io.in1 ^ io.in2
  }
}
```

```verilog
module ALU(
  input        clock,
  input        reset,
  input  [7:0] io_in1,
  input  [7:0] io_in2,
  input  [2:0] io_op,
  output [7:0] io_out
);
  wire [7:0] _io_out_T_1 = io_in1 + io_in2;
  wire [7:0] _io_out_T_3 = io_in1 - io_in2;
  wire [7:0] _io_out_T_4 = io_in1 & io_in2;
  wire [7:0] _io_out_T_5 = io_in1 ^ io_in2;
  wire [7:0] _GEN_0 =
    io_op == 3'h2 ? _io_out_T_4 : _io_out_T_5;
  wire [7:0] _GEN_1 =
    io_op == 3'h1 ? _io_out_T_3 : _GEN_0;
  assign io_out =
    io_op == 3'h0 ? _io_out_T_1 : _GEN_1;
endmodule
```

Components as Modules

```
class MySubModule(width: Int)
extends Module {
  val tpe = UInt(width.W)
  val io = IO(new Bundle {
    val in1 = Input(tpe)
    val in2 = Input(tpe)
    val out = Output(tpe)
  })
  io.out := io.in1 + io.in2
}
```

Components as Modules

```
class MySubModule(width: Int)
extends Module {
  val tpe = UInt(width.W)
  val io = IO(new Bundle {
    val in1 = Input(tpe)
    val in2 = Input(tpe)
    val out = Output(tpe)
  })
  io.out := io.in1 + io.in2
}
```

```
class MyModule(nbSub: Int, width: Int)
extends Module {
  val tpe = UInt(width.W)
  val io = IO(new Bundle {
    val in1 = Input(Vec(nbSub, tpe))
    val in2 = Input(Vec(nbSub, tpe))
    val out = Output(Vec(nbSub, tpe))
  })
  for (i <- 0 until nbSub) {
    val mod = Module(new MySubModule(width))
    mod.io.in1 := io.in1(i)
    mod.io.in2 := io.in2(i)
    io.out(i)  := mod.io.out
  }
}
```

Components as Modules

```
class MySubModule(width: Int)
extends Module {
  val tpe = UInt(width.W)
  val io = IO(new Bundle {
    val in1 = Input(tpe)
    val in2 = Input(tpe)
    val out = Output(tpe)
  })
  io.out := io.in1 + io.in2
}
```

```
class MyModule(nbSub: Int, width: Int)
extends Module {
  val tpe = UInt(width.W)
  val io = IO(new Bundle {
    val in1 = Input(Vec(nbSub, tpe))
    val in2 = Input(Vec(nbSub, tpe))
    val out = Output(Vec(nbSub, tpe))
  })
  for (i <- 0 until nbSub) {
    val mod = Module(new MySubModule(width))
    mod.io.in1 := io.in1(i)
    mod.io.in2 := io.in2(i)
    io.out(i)  := mod.io.out
  }
}
```

(Sub)Circuits are **modules** AND **objects**

Sequential Logic

```
class SimpleRegister(width: Int)
extends Module {
  val tpe = UInt(width.W)
  val io = IO(new Bundle{
    val q   = Input(tpe)
    val en  = Input(Bool())
    val d   = Output(tpe)
  })
  val reg = RegEnable(io.q, io.en)
  io.d := reg
}
```

Functions as hardware constructs

Sequential Logic

```scala
class SimpleRegister(width: Int)
extends Module {
  val tpe = UInt(width.W)
  val io = IO(new Bundle{
    val q   = Input(tpe)
    val en  = Input(Bool())
    val d   = Output(tpe)
  })
  val reg = RegEnable(io.q, io.en)
  io.d := reg
}
```

Functions as hardware constructs

```verilog
module SimpleRegister(
  input        clock,
  input        reset,
  input  [7:0] io_q,
  input        io_en,
  output [7:0] io_d
);
  reg [7:0] reg_;
  assign io_d = reg_;
  always @(posedge clock) begin
    if (io_en) begin
      reg_ <= io_q;
    end
  end
end module
```

Finite State Machines

```
object State extends ChiselEnum {
  val sNone, sOne1, sTwo1s = Value
}

val state = RegInit(sNone)
io.out := (state === sTwo1s)
```

Finite State Machines

```scala
object State extends ChiselEnum {
  val sNone, sOne1, sTwo1s = Value
}

val state = RegInit(sNone)
io.out := (state === sTwo1s)
```

Expressivity

- similar to HDLs
- but enumeration can be parametric

Finite State Machines

```
object State extends ChiselEnum {
  val sNone, sOne1, sTwo1s = Value
}

val state = RegInit(sNone)
io.out := (state === sTwo1s)
```

Expressivity

- similar to HDLs
- but enumeration can be parametric

```
switch (state) {
  is (sNone) {
    when (io.in) {
      state := sOne1
    }
  }
  is (sOne1) {
    when (io.in) {
      state := sTwo1s
    } .otherwise {
      state := sNone
    }
  }
  is (sTwo1s) {
    when (!io.in) {
      state := sNone
    }
  }
}
```

BlackBox Modules

What is a black box?

- legacy and/or analog IP
- may use **analog** datatype
- cannot be described in `Chisel`
    - maybe it's not worth it

BlackBox Modules

What is a black box?

- legacy and/or analog IP
- may use **analog** datatype
- cannot be described in `Chisel`
  - maybe it's not worth it

Can be used in hardware projects to migrate to `Chisel`

Comparing Chisel and Hardware Description Languages

Chisel = HDL?

- can do *almost everything* that could be done in HDL
  - combinatorial/sequential logics
  - *Finite-State Machine*
  - module hierarchy with **integer parameters**

Comparing Chisel and Hardware Description Languages

Chisel = HDL?

- can do *almost everything* that could be done in HDL
  - combinatorial/sequential logics
  - *Finite-State Machine*
  - module hierarchy with **integer parameters**
  - **design semantics** $\neq$ **simulation semantics**

Comparing Chisel and Hardware Description Languages

Chisel = HDL?

- can do *almost everything* that could be done in HDL
  - combinatorial/sequential logics
  - *Finite-State Machine*
  - module hierarchy with **integer parameters**
  - **design semantics** $\neq$ **simulation semantics**
- can use **black boxes** if needed

Comparing `Chisel` and Hardware Description Languages

`Chisel` = HDL?

- can do *almost everything* that could be done in HDL
  - combinatorial/sequential logics
  - *Finite-State Machine*
  - module hierarchy with **integer parameters**
  - **design semantics $\neq$ simulation semantics**
- can use **black boxes** if needed
- **what about writing generators?**

Comparing Chisel and Hardware Description Languages

Chisel = HDL?

- can do *almost everything* that could be done in HDL
  - combinatorial/sequential logics
  - *Finite-State Machine*
  - module hierarchy with **integer parameters**
  - **design semantics** $\neq$ **simulation semantics**
- can use **black boxes** if needed
- **what about writing generators?**

would be a shame not to use **high-level features**

Plan

Hardware (Simple) Generators

```
val regVec = Reg(Vec(8, UInt(1.W)))
regVec(0) := io.din
for (i <- 1 until 8) {
    regVec(i) := regVec (i - 1)
}
io.dout := regVec(7)
```

Shift register with `for` loop

Hardware (Simple) Generators

```
val regVec = Reg(Vec(8, UInt(1.W)))
regVec(0) := io.din
for (i <- 1 until 8) {
    regVec(i) := regVec (i - 1)
}
io.dout := regVec(7)
```

Shift register with `for` loop

```
val mySignal = Wire(UInt(8.W))
initValue match
    case Some(value) =>
        mySignal := value.U
    case None =>
        mySignal := DontCare
```

Conditional assignment to signal

Hardware (Simple) Generators

```
val regVec = Reg(Vec(8, UInt(1.W)))
regVec(0) := io.din
for (i <- 1 until 8) {
    regVec(i) := regVec (i - 1)
}
io.dout := regVec(7)
```

Shift register with `for` loop

```
val mySignal = Wire(UInt(8.W))
initValue match
    case Some(value) =>
        mySignal := value.U
    case None =>
        mySignal := DontCare
```

Conditional assignment to signal

Using parameters for generation

- similar to VHDL `generic` and `verilog` `parameter`.
- code is **executed** during elaboration

Lightweight Components with Functions

```
def adder (x: UInt , y: UInt, max: Int) = {
    val tmp = Wire(x + y)
    Mux(tmp > max.U, max.U, tmp)
}
```

Lightweight Components with Functions

```
def adder (x: UInt , y: UInt, max: Int) = {
    val tmp = Wire(x + y)
    Mux(tmp > max.U, max.U, tmp)
}
```

```
def delay(x: UInt, n: Int) = {
    val reg = Reg(Vec(n, UInt()))
    reg(0) := x
    for (i <- 1 until n) {
        reg(i) := reg(i - 1)
    }
    reg(n-1)
}
```

Lightweight Components with Functions

```
def adder (x: UInt , y: UInt, max: Int) = {
    val tmp = Wire(x + y)
    Mux(tmp > max.U, max.U, tmp)
}
```

```
def delay(x: UInt, n: Int) = {
    val reg = Reg(Vec(n, UInt()))
    reg(0) := x
    for (i <- 1 until n) {
        reg(i) := reg(i - 1)
    }
    reg(n-1)
}
```

```
val io.out = delay(adder(io.in_a, io.in_b, 16), 4)
```

Lightweight Components with Functions

```
def adder (x: UInt , y: UInt, max: Int) = {
    val tmp = Wire(x + y)
    Mux(tmp > max.U, max.U, tmp)
}
```

```
def delay(x: UInt, n: Int) = {
    val reg = Reg(Vec(n, UInt()))
    reg(0) := x
    for (i <- 1 until n) {
        reg(i) := reg(i - 1)
    }
    reg(n-1)
}
```

```
val io.out = delay(adder(io.in_a, io.in_b, 16), 4)
```

Functions as components

- can use **functional libraries**
- can build **recursive definitions** (*e.g.* `reduceBin`)

Lightweight Components with Functions

```
def adder (x: UInt , y: UInt, max: Int) = {
    val tmp = Wire(x + y)
    Mux(tmp > max.U, max.U, tmp)
}
```

```
def delay(x: UInt, n: Int) = {
    val reg = Reg(Vec(n, UInt()))
    reg(0) := x
    for (i <- 1 until n) {
        reg(i) := reg(i - 1)
    }
    reg(n-1)
}
```

```
val io.out = delay(adder(io.in_a, io.in_b, 16), 4)
```

Functions as components

- can use **functional libraries**
- can build **recursive definitions** (*e.g.* `reduceBin`)
- **high-order functions as parameters**

Packaging I/Os with Bundles

```
class GcdInputBundle(val w: Int)
  extends Bundle {
    val value1 = UInt(w.W)
    val value2 = UInt(w.W)
}
```

```
class GcdOutputBundle(val w: Int)
  extends Bundle {
    val value1 = UInt(w.W)
    val value2 = UInt(w.W)
    val gcd    = UInt(w.W)
}
```

Packaging I/Os with Bundles

```
class GcdInputBundle(val w: Int)
  extends Bundle {
    val value1 = UInt(w.W)
    val value2 = UInt(w.W)
}
```

```
class GcdOutputBundle(val w: Int)
  extends Bundle {
    val value1 = UInt(w.W)
    val value2 = UInt(w.W)
    val gcd    = UInt(w.W)
}
```

```
val input = IO(Flipped(Decoupled(new GcdInputBundle(width))))
val output = IO(Decoupled(new GcdOutputBundle(width)))
```

Packaging I/Os with Bundles

```
class GcdInputBundle(val w: Int)
  extends Bundle {
    val value1 = UInt(w.W)
    val value2 = UInt(w.W)
}
```

```
class GcdOutputBundle(val w: Int)
  extends Bundle {
    val value1 = UInt(w.W)
    val value2 = UInt(w.W)
    val gcd    = UInt(w.W)
}
```

```
val input = IO(Flipped(Decoupled(new GcdInputBundle(width))))
val output = IO(Decoupled(new GcdOutputBundle(width)))
```

Handling I/Os and synchronization

- can use `Bundle` as **datatypes**
- built-in **synchronization primitives** — *e.g.* `Decoupled(bundle)`:
  - `bundle.bits`: data (output)
  - `bundle.valid`: bool (output)
  - `bundle.ready`: bool (input)

Simple Parameters for Generation

```
class ParamAdder(n: Int) extends Module {
    val io = IO(new Bundle {
        val a = Input(UInt(n.W))
        val b = Input(UInt(n.W))
        val c = Output(UInt())
    })
    io.c := io.a + io.b
}
```

Simple Parameters for Generation

```
class ParamAdder(n: Int) extends Module {
    val io = IO(new Bundle {
        val a = Input(UInt(n.W))
        val b = Input(UInt(n.W))
        val c = Output(UInt())
    })
    io.c := io.a + io.b
}

val add8 = Module(new ParamAdder(8))
val add16 = Module(new ParamAdder(16))
```

Simple Parameters for Generation

```
class ParamAdder(n: Int) extends Module {
    val io = IO(new Bundle {
        val a = Input(UInt(n.W))
        val b = Input(UInt(n.W))
        val c = Output(UInt())
    })
    io.c := io.a + io.b
}

val add8 = Module(new ParamAdder(8))
val add16 = Module(new ParamAdder(16))
```

Simple parameters

- once again, similar to `verilog parameter`/VHDL `generic`...

Simple Parameters for Generation

```
class ParamAdder(n: Int) extends Module {
    val io = IO(new Bundle {
        val a = Input(UInt(n.W))
        val b = Input(UInt(n.W))
        val c = Output(UInt())
    })
    io.c := io.a + io.b
}

val add8 = Module(new ParamAdder(8))
val add16 = Module(new ParamAdder(16))
```

Simple parameters

- once again, similar to `verilog` `parameter`/VHDL `generic`...
- ... except that **everything can be a parameter**

Plan

Functions with Type Parameters

```
def myMux[T <: Data](sel: Bool , tPath: T, fPath: T): T = {
    val ret = WireDefault(fPath)
    when (sel) {
        ret := tPath
    }
    ret
}
```

Functions with Type Parameters

```
def myMux[T <: Data](sel: Bool , tPath: T, fPath: T): T = {
    val ret = WireDefault(fPath)
    when (sel) {
        ret := tPath
    }
    ret
}

val resA = myMux(selA , 5.U, 10.U)
val resB = myMux(selB , 6.S, -4.S)
```

Functions with Type Parameters

```
def myMux[T <: Data](sel: Bool , tPath: T, fPath: T): T = {
    val ret = WireDefault(fPath)
    when (sel) {
        ret := tPath
    }
    ret
}

val resA = myMux(selA , 5.U, 10.U)
val resB = myMux(selB , 6.S, -4.S)
```

Type parameters

- base Chisel datatype: `Data`
- type **inheritance** and **traits**

Functions with Type Parameters

```
def myMux[T <: Data](sel: Bool , tPath: T, fPath: T): T = {
    val ret = WireDefault(fPath)
    when (sel) {
        ret := tPath
    }
    ret
}

val resA = myMux(selA , 5.U, 10.U)
val resB = myMux(selB , 6.S, -4.S)
```

Type parameters

- base Chisel datatype: `Data`
- type **inheritance** and **traits**
- **can reuse code for different data types**

Generate Combinational Logic

```
val squareROM = VecInit(0.U, 1.U, 4.U, 9.U, 16.U, 25.U)
val coeffROM  = VecInit.tabulate(nValues)(i => (math.cos(i).U, math.sin(i).U))
```

Generate Combinational Logic

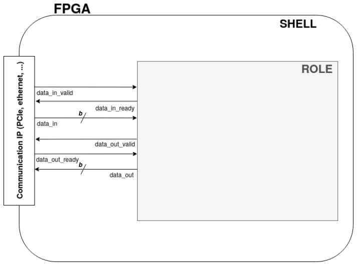
```scala
val squareROM = VecInit(0.U, 1.U, 4.U, 9.U, 16.U, 25.U)
val coeffROM  = VecInit.tabulate(nValues)(i => (math.cos(i).U, math.sin(i).U))
```

Signal and memory initialization

- can initialize any structure
- useful to set **ROM** with constants
- can use **software functions** for initialization

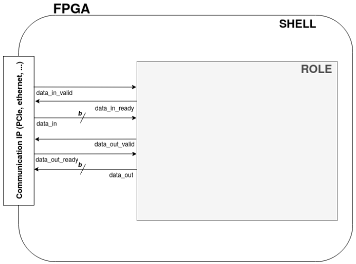
Hardware Inheritance

```
abstract class Role(val width: Int)
extends Module {
  val tpe = UInt(width.W)
  val io = IO(new Bundle {
    val data_in  = Flipped(Decoupled(tpe))
    val data_out = Decoupled(tpe)
  })
}
```



Hardware Inheritance

```
abstract class Role(val width: Int)
extends Module {
  val tpe = UInt(width.W)
  val io = IO(new Bundle {
    val data_in  = Flipped(Decoupled(tpe))
    val data_out = Decoupled(tpe)
  })
}
```



```
class Shifter(width: Int)
extends Role(width) {
  io.data_in.ready := true.B
  when (io.data_out.ready) {
    io.data_out.bits :=
      io.data_in.bits << 2
    io.data_out.valid :=
      io.data_in.valid
  }
}
```

Hardware Inheritance

```
trait HasMemoryAccess {
  def read(addr: UInt): UInt
  def write(addr: UInt, value: UInt): Unit
}
```

Hardware Inheritance

```
trait HasMemoryAccess {
  def read(addr: UInt): UInt
  def write(addr: UInt, value: UInt): Unit
}

class BlockRAM(width: Int, nElem: Int, nBank: Int)
extends Role(width) with HasMemoryAccess {
  // has access to data_in and data_out
  // must define read and write
}
```

Hardware Inheritance

```
trait HasMemoryAccess {
  def read(addr: UInt): UInt
  def write(addr: UInt, value: UInt): Unit
}

class BlockRAM(width: Int, nElem: Int, nBank: Int)
extends Role(width) with HasMemoryAccess {
  // has access to data_in and data_out
  // must define read and write
}
```

Inheritance and data types

- can also use inheritance for **bundles** and **data types**
  - base class: `Data`
- **structured data types**

Functional Programming for Hardware Generation

```
val vec = Vec(nElem, UInt(8.W))
```

Functional Programming for Hardware Generation

```
val vec = Vec(nElem, UInt(8.W))
```

```
def add(a: UInt, b: UInt) = a + b
```

```
// generate a comb reduction tree
val sum = vec.reduce(add)
```

Functional Programming for Hardware Generation

```
val vec = Vec(nElem, UInt(8.W))

def add(a: UInt, b: UInt) = a + b

// generate a comb reduction tree
val sum = vec.reduce(add)

// generate a balanced reduction tree
val sumB = vec.reduceTree(add)
```

Functional Programming for Hardware Generation

```scala
val vec = Vec(nElem, UInt(8.W))
```

```scala
def add(a: UInt, b: UInt) = a + b
```

```scala
def addReg(a: UInt, b: UInt) = RegNext(a + b)
def idReg(a: UInt) = RegNext(a)
```

```scala
// generate a comb reduction tree
val sum = vec.reduce(add)


// generate a balanced reduction tree
val sumB = vec.reduceTree(add)
```

Functional Programming for Hardware Generation

```
val vec = Vec(nElem, UInt(8.W))
```

```
def add(a: UInt, b: UInt) = a + b

// generate a comb reduction tree
val sum = vec.reduce(add)


// generate a balanced reduction tree
val sumB = vec.reduceTree(add)
```

```
def addReg(a: UInt, b: UInt) = RegNext(a + b)
def idReg(a: UInt) = RegNext(a)

// balanced tree with delay
val sumR = vec.reduceTree(addReg, idReg)
```

Functional Programming for Hardware Generation

```
val vec = Vec(nElem, UInt(8.W))
```

```
def add(a: UInt, b: UInt) = a + b

// generate a comb reduction tree
val sum = vec.reduce(add)


// generate a balanced reduction tree
val sumB = vec.reduceTree(add)
```

```
def addReg(a: UInt, b: UInt) = RegNext(a + b)
def idReg(a: UInt) = RegNext(a)

// balanced tree with delay
val sumR = vec.reduceTree(addReg, idReg)

// searching minimum in vec
val min = vec
    .reduceTree((x, y) => Mux(x < y, x, y))
```

Functional Programming for Hardware Generation

```
val vec = Vec(nElem, UInt(8.W))
```

```
def add(a: UInt, b: UInt) = a + b

// generate a comb reduction tree
val sum = vec.reduce(add)

// generate a balanced reduction tree
val sumB = vec.reduceTree(add)
```

```
def addReg(a: UInt, b: UInt) = RegNext(a + b)
def idReg(a: UInt) = RegNext(a)

// balanced tree with delay
val sumR = vec.reduceTree(addReg, idReg)

// searching minimum in vec
val min = vec
    .reduceTree((x, y) => Mux(x < y, x, y))
```

Functional programming

- functions can be used as **arguments**

Functional Programming for Hardware Generation

```
val vec = Vec(nElem, UInt(8.W))
```

```
def add(a: UInt, b: UInt) = a + b

// generate a comb reduction tree
val sum = vec.reduce(add)


// generate a balanced reduction tree
val sumB = vec.reduceTree(add)
```

```
def addReg(a: UInt, b: UInt) = RegNext(a + b)
def idReg(a: UInt) = RegNext(a)

// balanced tree with delay
val sumR = vec.reduceTree(addReg, idReg)

// searching minimum in vec
val min = vec
    .reduceTree((x, y) => Mux(x < y, x, y))
```

Functional programming

- functions can be used as **arguments**
  - can be used in `Module` constructors

Functional Programming for Hardware Generation

```
val vec = Vec(nElem, UInt(8.W))
```

```
def add(a: UInt, b: UInt) = a + b

// generate a comb reduction tree
val sum = vec.reduce(add)


// generate a balanced reduction tree
val sumB = vec.reduceTree(add)
```

```
def addReg(a: UInt, b: UInt) = RegNext(a + b)
def idReg(a: UInt) = RegNext(a)

// balanced tree with delay
val sumR = vec.reduceTree(addReg, idReg)

// searching minimum in vec
val min = vec
    .reduceTree((x, y) => Mux(x < y, x, y))
```

Functional programming

- functions can be used as **arguments**
  - can be used in `Module` constructors
- can be **composed**

Elaboration and Generation

Reflective programming

- analyze signals and modules during elaboration
- modify the generated **FIRRTL**

Elaboration and Generation

Reflective programming

- analyze signals and modules during elaboration
- modify the generated **FIRRTL**

Custom FIRRTL transforms

- define **transforms** to modify the IR
- dependancy system to define ordering

Elaboration and Generation

Reflective programming

- analyze signals and modules during elaboration
- modify the generated **FIRRTL**

Custom FIRRTL transforms

- define **transforms** to modify the IR
- dependancy system to define ordering

Examples

- `PresetAnnotation` for FPGA designs
- resource estimators in `QECE`

Elaboration and Generation

Reflective programming

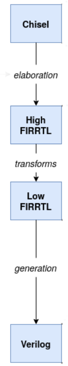
- analyze signals and modules during elaboration
- modify the generated **FIRRTL**

Custom FIRRTL transforms

- define **transforms** to modify the IR
- dependancy system to define ordering

Examples

- `PresetAnnotation` for FPGA designs
- resource estimators in `QECE`



Module with Type Parameters and Functional Programming

```
class GenericDotProduct[T <: Data with Num[T]](
  val tpe: T,
  val nElem: Int,
  val mulFunc: (T, T) => T,
  val addFunc: (T, T) => T,
  val latency : Int
) extends Module { ... }
```

Module with Type Parameters and Functional Programming

```
class GenericDotProduct[T <: Data with Num[T]](
  val tpe: T,
  val nElem: Int,
  val mulFunc: (T, T) => T,
  val addFunc: (T, T) => T,
  val latency : Int
) extends Module { ... }
```

Type parameters — again

- `[T <: Data with Num[T]]`: ensure that + and * exist
- **functional parameters** for generation

Module with Type Parameters and Functional Programming

```
class GenericDotProduct[T <: Data with Num[T]](
  val tpe: T,
  val nElem: Int,
  val mulFunc: (T, T) => T,
  val addFunc: (T, T) => T,
  val latency : Int
) extends Module { ... }
```

Type parameters — again

- `[T <: Data with Num[T]]`: ensure that + and ∗ exist
- **functional parameters** for generation
- *we'll see more in practical works*

Overview of Chisel Generation Possibilities

Interesting Features

In the library itself:

- Chisel execution for hardware elaboration
- high-level parametrization of modules, functions, bundles, . . .
- object-oriented and functional programming

Within the Chisel framework:

- reflexivity for code generation
- custom transforms on FIRRTL

References

- Martin Schoeberl. *Digital design with chisel*. Kindle Direct Publishing, 2019
- Bruno Ferres. "Leveraging Hardware Construction Languages for Flexible Design Space Exploration on FPGA". PhD thesis. Université Grenoble Alpes, 2022

Plan

Choosing a Design Paradigm

No generic answer

- depends on the algorithm
- depends on the target
- depends on the use case

Choosing a Design Paradigm

No generic answer

- depends on the algorithm
- depends on the target
- depends on the use case

Some key concerns

- are the kernels **highly regular**?
- do you need **performant interfaces**?
- are **domain specific IPs** available?

HLS *vs.* HCL

Cannot compare. . . but here are some keys

HLS *vs.* HCL

Cannot compare... but here are some keys

- HLS is (probably) better for
  - quick prototyping
  - regular (polyhedral) algorithms
  - automatic scheduling (dataflows, ...)
  - hardware design with little expertise

HLS *vs.* HCL

Cannot compare. . . but here are some keys

- HLS is (probably) better for
  - quick prototyping
  - regular (polyhedral) algorithms
  - automatic scheduling (dataflows, . . .)
  - hardware design with little expertise
- HCL are (probably) better for
  - tight control of **interfaces** (memory, comm., . . .)
  - IP integration
  - custom optimization
  - reusability and modularity

Overview of `Chisel`-based Projects

- RISC-V cores: *rocket-chip*, *Boom*, *RISC-V mini*
- AI cores: *Google TPU*, *DANA* co-processor
- network design: *Pipeline Automation Framework* (OVHcloud)
- design space exploration: `QECE` framework

https://www.chisel-lang.org/community.html

Practical Work: Designing a Dot Product Generator

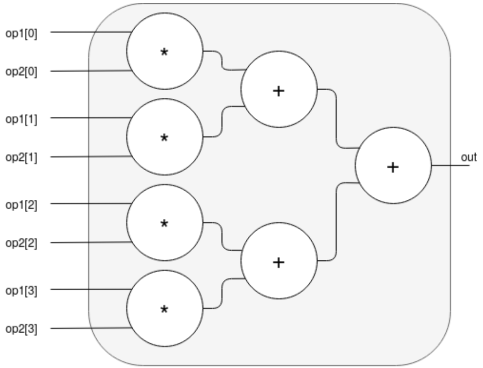
$$\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} \cdot \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} = a_0 b_0 + a_1 b_1 + a_2 b_2 + a_3 b_3$$

Dot Product algorithm

Practical Work: Designing a Dot Product Generator

$$\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} \cdot \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} = a_0 b_0 + a_1 b_1 + a_2 b_2 + a_3 b_3$$

Dot Product algorithm



Proposal of architecture

Practical Work: Designing a Dot Product Generator

Objectives of the Practical Work

- learn `scala/Chisel` syntax
- understand the **generation flow**
- experiment using **high level features** in design
- *compare with the **HLS lab** from this morning*

References (1/2)

- Chisel website: https://www.chisel-lang.org/
- Chisel bootcamp: https://github.com/ucb-bar/chisel-tutorial
- Martin Schoeberl. *Digital design with chisel*. Kindle Direct Publishing, 2019
- Jean Bruant. "Abstracting Hardware Architectures for Agile Design of High-performance Applications on FPGA". PhD thesis. Université Grenoble Alpes, 2023
- Bruno Ferres. "Leveraging Hardware Construction Languages for Flexible Design Space Exploration on FPGA". PhD thesis. Université Grenoble Alpes, 2022

References (2/2)

- J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R Avižienis, J. Wawrzynek, and K. Asanović. "Chisel: Constructing hardware in a Scala embedded language". In: *DAC Design Automation Conference 2012.* 2012
- Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, et al. "Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations". In: *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD).* 2017
- Andrew Dobis, Kevin Laeufer, Hans Jakob Damsgaard, Tjark Petersen, Kasper Juul Hesse Rasmussen, Enrico Tolotto, Simon Thye Andersen, Richard Lin, and Martin Schoeberl. "Verification of Chisel Hardware Designs with ChiselVerify". In: *Microprocessors and Microsystems* (2023)