FPROSoC in Chisel

Chisel

Chisel is technically not a new language but a library written in and for Scala, more exactly it is domain specific language (DSL) for describing hardware circuits embedded in Scala, but given it's extensions it provides on top of Scala it's often described as open-source hardware description language.

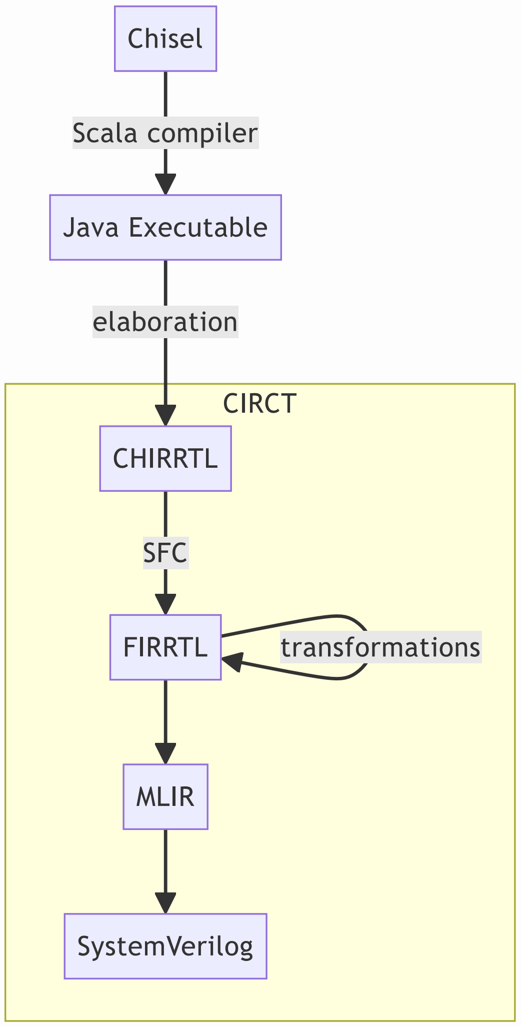
Hardware generation

In the early days, scripting languages like Perl and Python were used to generate VHDL/Verilog code (sometimes from excel spreadsheets), but the problem with such generators were that they worked on strings, without real understanding (typification) of ingested/generated code. SystemVerilog tried to migrate the this by offering some generational constructs (`#(parameter N=4)`, `generate`) and even some high level constructs like classes that are only available for verification (are typically not synthesizable).

Hardware construction language

Chisel is actually hardware construction language, that allows us to write hardware generators instead of describing hardware directly. Because it uses Scala each module can be parametrized/generalized as any Scala class (using various parameters and generics) and then use those arguments for "generation" of modules. Using Scala loops or even higher order functions it is possible to create pretty advance generation. To understand how and why this works, we will take a look in Chisel to Verilog compilation.

Chisel to Verilog



Chisel gets compiled (as Scala) to JVM, which is then executed (this is usually done by `sbt run` command).

Is execution on Chisel compiled program, which during construction of `Top` class (and it's subsequent classes) add hardware AST (CHIRRTL) to `ChiselStage`. Each `Module` class gets compiled separately (and it will get lowered into separate SystemVerilog module). Other classes are inlined into first parent `Module` class. This part does hardware generation, from parametrized constructs to concrete hardware AST (modules). Each iteration of loops just adds more hardware AST nodes to stage.

Scala FIRRTL (SFC) lowers CHIRRTL to FIRRTL, on which FIRRTL compilers apply some transformations (optimizations; ex: for unused signals) and lower it to MLIR that get optimized further and compiled into targeted language (usually SystemVerilog).

Elaboration

```
// The body of a Scala class is the default constructor
// MyModule's default constructor has a single Int argument
// Superclass Module is a chisel3 Class that begins construction of a hardware moc
```

```scala
// Implicit clock and reset inputs are added by the Module constructor
class MyModule(width: Int) extends Module {
  // io is a required field for subclasses of Module
  // new Bundle creates an instance of an anonymous subclass of Chisel's Bundle (l
  // When executing the function IO(...), Chisel adds ports to the Module based or
  val io = IO(new Bundle {
    val in = Input(UInt(width.W)) // Input port with width defined by parameter
    val out = Output(UInt()) // Output port with width inferred by Chisel
  })

  // A Scala println that will print at elaboration time each time this Module is
  // This does NOT create a node in the Module AST
  println(s"Constructing MyModule with width $width")

  // Adds a register declaration node to the Module AST
  // This counter register resets to the value of input port io.in
  // The implicit clock and reset inputs feed into this node
  val counter = RegInit(io.in)

  // Adds an addition node to the hardware AST with operands counter and 1
  val inc = counter + 1.U // + is overloaded, this is actually a Chisel function c

  // Connects the output of the addition node to the "next" value of counter
  counter := inc

  // Adds a printf node to the Module AST that will print at simulation time
  // The value of counter feeds into this node
  printf("counter = %d\n", counter)
}
```

Design flow in Chisel

Chisel Tree

```
.
├── build.sbt
├── src/
|   ├── main/
|   |   ├── resources/
|   |   |   └ *.sv
|   |   └── scala/cpro/
|   |       └ Top.scala
|   └── test/scala/cpro/
|       └ *.scala
└── target_sv/
    └ *.sv
```

Firstly, I imported all files from vaja6 into template Chisel tree (under resource directory), this way all sources are in one place (single source of truth).

`BlackBox`

Secondly, I created `BlackBox` for Top.sv file. `BlackBox` are like header files in C, they only define interface/signature not actual implementation and are not type checked (mismatch between Verilog and Chisel `BlackBox` will cause error in simulation/synthesis if you are lucky).

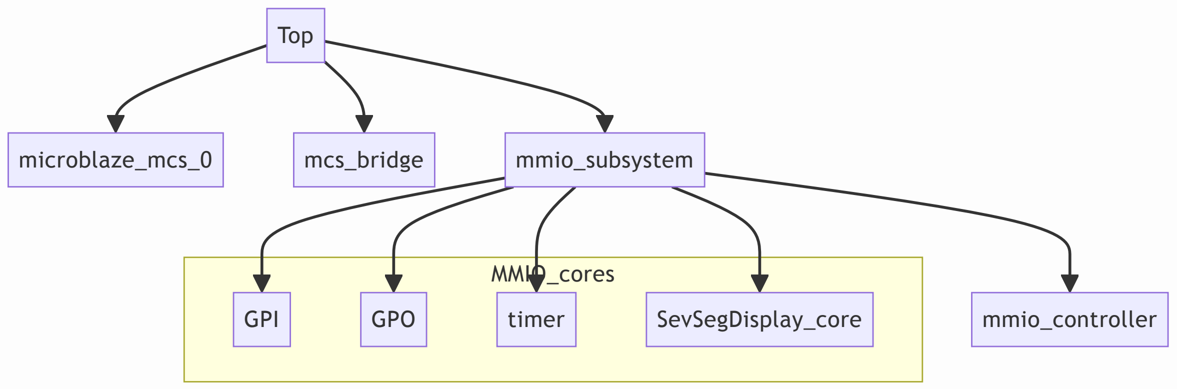
```
/*
module Top(
  input         clock,
                reset,
  input  [15:0] sw,
  output [15:0] led,
  output [7:0]  anode_assert,
  output [6:0]  segs
);
*/
class Top extends BlackBox with HasBlackBoxResource {
  val io = IO(new Bundle {
    // in blackbox we need to explicitly provide clock&reset
    val clock = Input(Clock())
    val reset = Input(Reset())
    val sw = Input(UInt(16.W))
    val led = Output(UInt(16.W))
    val anode_assert = Output(UInt(8.W))
    val segs = Output(UInt(7.W))
  })
  // will copy resource files into target_sv
  addResource("/top.sv")
  addResource("/mcs_bridge.sv")
  addResource("/mmio_controller.sv")
  addResource("/mmio_cores.sv")
  addResource("/mmio_subsystem.sv")
}

object Top extends App {
  ChiselStage.emitSystemVerilogFile(
    new Top,
    Array("--split-verilog", "--target-dir", "target_sv/"),
    firtoolOpts = Array("-disable-all-randomization", "-strip-debug-info") // make
  )
}
```

Lastly I setup `ChiselStage` to emit SystemVerilog files into target_sv directory.

Now if we run `sbt run` Chisel/Scala gets compiled, elaborated and lowered into SystemVerilog files that are located in target_sv. This folder can now be loaded into vivaldo project as sources folder (make sure that "copy to project" is not checked). Then we simply associate ELF file and generate bitstream and test on the board.

Incremental rewrite



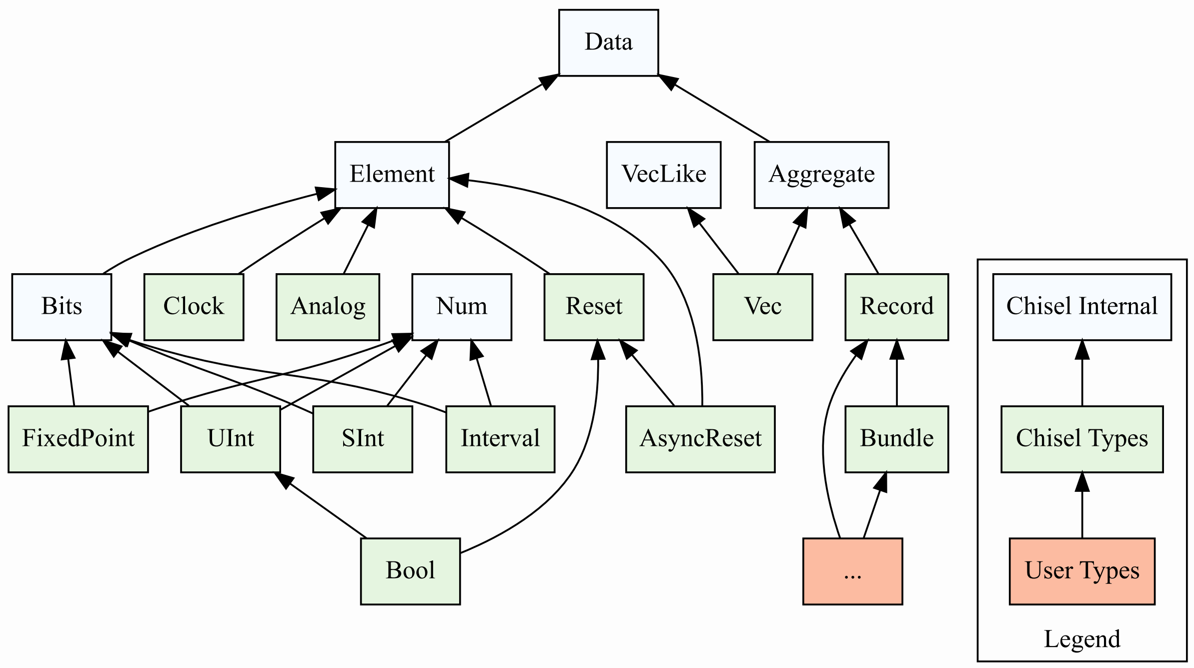
Using top-down approach we can incrementally rewrite Modules into Chisel and/or blackbox them. Then regenerate bitstream and retest, this way we know we did not break anything in this step of rewriting. We can also write tests in Chisel for modules to ensure no behavior changes (it make sense to write test for blackboxes and later when doing rewrite we expect that tests will still pass). Test are run using `sbt test` and they require verilator to be installed (WSL on Windows).

For `Top` class we need to change it from `BlackBox` to `Module`, then create blackboxes for all of it's children (`microblaze_mcs_0`, `mcs_bridge`, `mmio_subsystem`) and write actual Chisel implementation of `Top` (which is mostly just connecting wires between it's children) and finally remove Top.sv file from resources.

After whole rewrite was done it was time to write more idiomatic Chisel, more on that later.

Verilog -> Chisel

`Data`



All Chisel types have `Data` for superclass. Every `Data` class can be represented as a bit vector in a hardware design.

`Bool`

`Bool()` represents single bit that can be `true.B` or `false.B`. Chisel does not have representation of Z or X states.

`UInt`

`value.U(width.W)`

```
0xA0.U(16.W) // this can be problematic if we overflow Scala integer
"hA0".U(16.W) // so string literals are preferred
```

slices

Verilog	Chisel
`a[15]`	`a(15)`
`a[32:16]`	`a(32,16)`

slice assignment

Chisel does not support slice/subword assignment, but that limitation can be bypassed by conCATenation or by transmuting to `Vec` or `Bundle`

```
val data = 0.U(8.W)
val ones = chisel3.util.Fill(4, 1.U) // 0b1111
// {4{1}}
val data2 = chisel3.util.Cat(data, ones).U(8.W) // 0b0000_1111
// {data,ones}
val data22 = chisel3.util.Cat(data(7,4), ones) // 0b0000_1111
```

The reasoning is that if you need subword assignment you probably need better abstraction with an aggregate/structured types.

`Vec`

`Vec`tors are like arrays. They have fixed size of same type/size elements. This restriction is relaxed in `MixedVec`

```
val a = Vec(4, Bool())
// with default options this becomes:
/*
logic a_0;
logic a_1;
logic a_2;
logic a_3;
*/
```

`Bundle`

Are like `struct` in C, they group together several named fields. This is mostly used to group ports (module inputs/outputs) into Bundles.

```
val b = new Bundle({
  val a = Bool()
  val b = Bool()
  val c = Bool()
})
/*
logic b_a
logic b_b
logic b_c
*/
```

`Module` definition

```
module M
#(
    parameter ARG = 32'hc000_0000
)
(
    input logic clock,
    input logic reset,
    input logic inn,
    input logic [31:0] inn_,
    input logic [63:0][31:0] inn_arr, // Packed array
    // Chisel does not support unpacked arrays: [31:0] inn_arr [63:0]

    output logic out,
    output logic [31:0] out_,
    output logic [63:0][31:0] out_arr,
);
//...
endmodule
```

```
// whole class definition is actually class constructor
class M(ARG: Long = 0xc000_0000L) extends Module {
    require(ARG > 0) // assert assumptions
    /*
    // Module has implicit clock&reset signals
    val clock = Input(Clock())
    val reset = Input(Reset())
    */

    // all IO connections are typically joined in IO Bundle
    val io = IO(new Bundle {
        val inn = Input(Bool())
        val inn_ = Input(UInt(32.W/*idth*/)) // common pitfall: [31:0] has width c
        val inn_arr = Input(Vec(64, UInt(32.W)))
        // To consider vectors as packed arrays instead of scalarized inputs
        // the following options are needed to be passed to firtool:
        // -preserve-aggregate=all -scalarize-public-modules=false -scalarize-ext-

        val out = Output(Bool())
        val out_ = Output(UInt(32.W))
        val out_arr = Output(Vec(64, UInt(32.W)))
    })
    // ...
}
```

Module instantiation

```
M m (
    .inn     (_inn),
    .inn_    (_inn_),
    .inn_arr (_inn_arr),
    .out     (_out),
    .out_    (_out_),
    .out_arr (_out_arr),
);
```

```
val m = Module(new M)
/*
// implicit clock&reset signals are automatically connected
m.clock <> clock
m.reset <> reset
*/
```

```
m.io.inn <> _inn
m.io.inn_ <> _inn_
m.io.inn_arr <> _inn_arr
m.io.out <> _out
m.io.out_ <> _out_
m.io.out_arr <> _out_arr
// or if we use :=
m.io.inn := _inn
m.io.inn_ := _inn_
m.io.inn_arr := _inn_arr
_out := m.io.out
_out_ := m.io.out_
_out_arr := m.io.out_arr
// if any connections are missing we get elaboration error
```

`=` , `:=` , `<>`

- `=` is Scala assignment (evaluated at elaboration time)
- `sink := source` is Chisel (mono-directional) assignment (`assign =` , `<=`)
- `<>` is Chisel bi-directional connection operator (`assign =`)

"Variables"

There is no concept of `logic` because everything needs to be typed, so you need to use reg or wire explicitly:

Verilog	Chisel
`logic a;`	/
`wire a;`	`val a = Wire(/*type*/)`
`reg a;`	`val a = Reg(/*type*/)`
	`val a = RegInit(reset_value)`
	`val a = RegNext(next_value, reset_value)`

`RegInit`

```
val r = RegInit(0.U(32.W)) // NOTE: implicit clock&reset
// Chisel will automatically put reg assignments into always_ff block
// that is by default run on positive edge
r := 1.U
r := 2.U // last assignments wins
```

manually in Chisel:

```
val r = Reg(UInt(32.W)) // NOTE: implicit clock
r := 1.U
// when cannot use Scala's if as those are evaluated at elaboration
//(are equal to Verilog's `generate if`)
when(reset) { // will add special HW AST that will emit `if` in Verilog
  r := 0.U
}.elsewhen(false.B) {
  // never
}.otherwise {
  r := 2.U
}
```

in verilog:

```
reg [31:0] r;
always @(posedge clock) begin
  if (reset)
    r <= 0;
  else
    r <= 2;
end
```

Switch

```
case (digit)
  4'b0000: segs = 7'b1000000; // 0
  4'b0001: segs = 7'b1111001; // 1
  4'b0010: segs = 7'b0100100; // 2
  4'b0011: segs = 7'b0110000; // 3
  4'b0100: segs = 7'b0011001; // 4
  4'b0101: segs = 7'b0010010; // 5
  4'b0110: segs = 7'b0000010; // 6
  4'b0111: segs = 7'b1111000; // 7
  4'b1000: segs = 7'b0000000; // 8
  4'b1001: segs = 7'b0010000; // 9
  4'b1010: segs = 7'b0001000; // A
  4'b1011: segs = 7'b0000011; // b
  4'b1100: segs = 7'b1000110; // C
  4'b1101: segs = 7'b0100001; // d
  4'b1110: segs = 7'b0000110; // E
  4'b1111: segs = 7'b0001110; // F
  default: segs = 7'b1111111; // off
endcase
```

```
io.segs := "b11111111".U // default
switch(digit) {
  is("b0000".U) { io.segs := "b1000000".U } // 0
  is("b0001".U) { io.segs := "b1111001".U } // 1
  is("b0010".U) { io.segs := "b0100100".U } // 2
  is("b0011".U) { io.segs := "b0110000".U } // 3
  is("b0100".U) { io.segs := "b0011001".U } // 4
  is("b0101".U) { io.segs := "b0010010".U } // 5
```

```
    is("b0110".U) { io.segs := "b0000010".U } // 6
    is("b0111".U) { io.segs := "b1111000".U } // 7
    is("b1000".U) { io.segs := "b0000000".U } // 8
    is("b1001".U) { io.segs := "b0010000".U } // 9
    is("b1010".U) { io.segs := "b0001000".U } // A
    is("b1011".U) { io.segs := "b0000011".U } // b
    is("b1100".U) { io.segs := "b1000110".U } // C
    is("b1101".U) { io.segs := "b0100001".U } // d
    is("b1110".U) { io.segs := "b0000110".U } // E
    is("b1111".U) { io.segs := "b0001110".U } // F
  }
```

Testing

```
module tb_GPI;
    // Declare testbench signals
    logic clock;
    logic reset;
    logic [4:0] address;
    logic [31:0] rd_data;
    // ...

    // Instantiate the GPI module
    GPI dut (
        .clock(clock),
        .reset(reset),
        .address(address),
        .rd_data(rd_data),
        // ...
    );

    // Clock generation
    initial begin
        clock = 0;
        forever #5 clock = ~clock; // 100 MHz clock
    end

    // Test sequence
    initial begin
        // Initialize signals
        reset = 1;
        address = 5'b0;

        // Apply reset
        #10;
        reset = 0;

        #20;
        assert (rd_data == 0);

        $finish;
    end
endmodule
```

```
class GPISpec extends AnyFreeSpec with Matchers {
  "GPI should work" in {
    simulate(new GPI()) { dut =>
      // Initialize signals
      dut.reset.poke(true.B)
      dut.slot_io.address.poke(0.U)

      // Apply reset
      dut.clock.step(1)
      dut.reset.poke(false.B)

      dut.clock.step(2)
      dut.slot_io.rd_data.expect(0.U)
    }
  }
}
```

Idiomatic Chisel

Verilog can be mapped to Chisel, but such Chisel code is not idiomatic/optimal

More high level constructs

- `val (count: UInt, wrap: Bool) = Counter(0 until 40000, enabled, reset)`
- `DecoupledIO` bundle with ready-valid interface
- `UIntToOH` & `OHToUInt` for one-hot encoding/decoding
- `Mux1H(sel: Seq[Bool], in: Seq[Data])`
- `ShiftRegister(in: Data, n: Int[, en: Bool]): Data`
- and much more in Chisel cheatsheet

Directly connecting instances

In Chisel you can connect instances directly without intermediate wires:

```
-   val w = Wire(Bool())
val m1 = Module(M1())
-   m1.w <> w
val m2 = Module(M2())
-   m2.w <> w

+   m1.w <> m2.w
```

Using `Bundle` for common ports

```
// FPro bus (consumer viewpoint)
class FPRO extends Bundle {
  val video_cs = Input(Bool())
  val mmio_cs = Input(Bool())
  val write = Input(Bool())
  val read = Input(Bool())
  val addr = Input(UInt(21.W))
  val wr_data = Input(UInt(32.W))
  val rd_data = Output(UInt(32.W))
}
```

Reuse `Bundle`

```
// this module produces FPro signals
class mcs_bridge extends Module {
  val fp = Flipped(FPRO()) // flips Input <-> Output
  // ...
}

// this module passes down FPro signals
class mmio_subsystem extends Module {
  val fp = IO(FPRO())
  // ...
  val mmioController = Module(new mmio_controller())
  mmioController.fp <> fp // will connect all wires automatically
  // ...
}

// this module consumes FPro signals
class mmio_controller extends Module {
  val fp = IO(FPRO())
  // ...
}
```

Using `trait` for interfaces

```
// define common interface for MMIO cores
trait MMIO_core {
  val slot_io = IO(new Slot())
}

class GPI extends Module with MMIO_core {
  // implicit slot_io inherited from trait
  // val slot_io = IO(new Slot())

  // ...
}

class GPO extends Module with MMIO_core {
  // val slot_io = IO(new Slot())

  // ...
}
```

this allows easier core additions

```
var slots: Array[MMIO_core] = Array()

// Instantiate the GPO (General Purpose Output)
val gpo = Module(new GPO())
gpo.io.data_out <> io.data_out
slots :+= gpo // append to array

// Instantiate the GPI (General Purpose Input)
val gpi = Module(new GPI())
gpi.io.data_in <> io.data_in
slots :+= gpi

// ...

require(slots.length < 64)
for ((core, i) <- slots.zipWithIndex) {
  core.slot_io.address <> reg_addr(i)
  core.slot_io.rd_data <> read_data(i)
  core.slot_io.wr_data <> write_data(i)
  core.slot_io.read <> read(i)
  core.slot_io.write <> write(i)
  core.slot_io.cs <> cs(i)
}
```

Compilation overheat

~1min for compilation to Verilog/running all tests

Discovered Vivaldo problems

- associate ELF does not work correctly if MCS is not in top module
- vivaldo does not check folder for new files so folder needs to be reimported
- sometimes changes are not detected (generate bitstream uses old files)

Sources

- https://www.researchgate.net/publication/383664706_Hardware_Generators_with_Chisel
- https://circt.llvm.org/
- https://stackoverflow.com/questions/44548198/chisel-code-transformation
- https://chipyard.readthedocs.io/en/stable/Tools/index.html
- https://chipyard.readthedocs.io/en/stable/Customization/Firrtl-Transforms.html
- Schoeberl, M. (2019). Digital Design with Chisel. Kindle Direct Publishing
- https://github.com/freechipsproject/chisel-bootcamp
- https://en.wikipedia.org/wiki/SystemVerilog