Contents

1 Hardware description languages

Hardware description languages, HDLs for short, are used to model circuits, typically digital. HDLs are **declarative** languages, they describe how the circuit should be constructed.

This is analogous to how HTML works. As an example, consider creating a list:

```
<ul>
  <li>Name: Siv Jensen, Affiliation: FrP</li>
  <li>Name: Jonas Gahr Støre, Affiliation: AP</li>
  <li>Name: Bjørnar Moxnes, Affiliation: Rødt</li>
  <li>Name: Malcolm Tucker, Affiliation: DOSAC</li>
</ul>
```

Which can then be rendered by an HTML engine to look something like this:

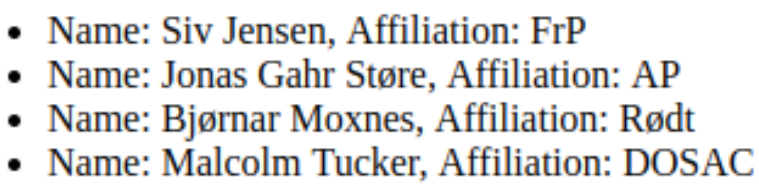

Figure 1: The HTML after being rendered

Instead of describing how text should be rendered HDLs describe how wires and components in a circuit should be connected. Although we have yet to introduce chisel, let's look at some code for a chisel circuit: It is not

necessary to understand what is going on in this code to continue. The following code generates the circuit shown in figure 2.

```
class SimpleCounter() extends Module {
  val io = IO(
    new Bundle {
      val reg_a = Output(UInt(32.W))
    }
  )

  val reg_a = RegInit(0.U(8.W))
  reg_a := reg_a + 1.U

  io.reg_a := reg_a
}
```

Just like the HTML describes a document the chisel code describes a simple circuit shown below:

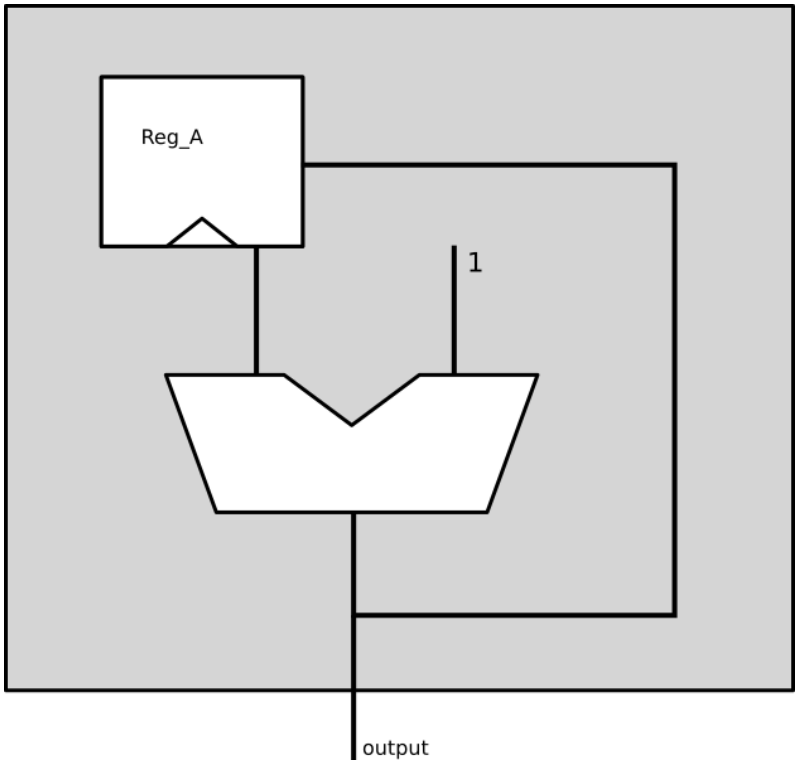

Figure 2: Simple adder circuit generated using the chisel code on the previous page.

In both examples the common theme is describing a **structure** for some component which must be rendered. In the HTML example the structure is **rendered** by your for instance your web-browser, and the same goes for the hardware description.

While the path from HTML -> Browser is fairly short it's a lot more involved for hardware description! This shouldn't come as a suprise, displaying text is less complex than creating digital circuits. A very simplified version of this is shown in figure 3.

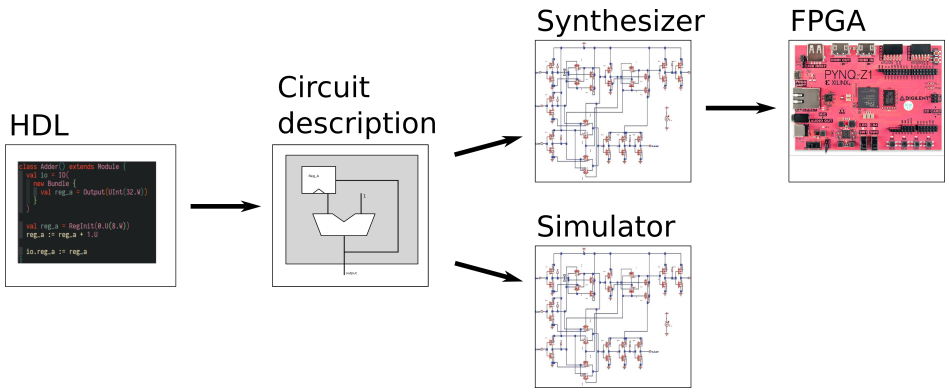

Figure 3: Flow using HDL.

2 Scala and Chisel

It is rather uncommon to write raw HTML nowadays, instead we write code that **generates** HTML. For the example above a generator might be written like this in scala:

```
def generateList(politicians: List[String], affiliations:
↪  Map[String, String]): String = {
  val inner = new ArrayBuffer[String]()
  for(ii <- 0 until politicians.size){
    val nameString = politicians(ii)
    val affiliationString = affiliations(nameString)
    inner.add(s"<li>Name: $nameString, Affiliation:
    ↪  $affiliationString</li>")
  }
  "<ul>\n" + inner.mkString("\n") + "</ul>"
}

// Or if you prefer brevity
def generateList2(politicians: List[String], affiliations:
↪  Map[String, String]): String = {
  val inner = politicians.map(p => s"<li>Name: $p, Affiliation
  ↪  ${affiliations(p)}</li>")
  "<ul>\n" + inner.mkString("\n") + "</ul>"
}
```

In this example a scala program manipulates HTML, and these builders can then be composed together:

```
def generateDistricts(districts: List[(String, List[String])],
↪  affiliations: Map[String, String]): String = {
  val inner = districts.map{ case(district, politicians) =>
    s"<li>$district\n" + generateList(politicians,
    ↪  affiliations) + "\n</li>"
  }.mkString("\n")

  s"<div>\n$inner\n</div>"
}
```

Just like the HTML, a chisel program is just a scala program that builds a chisel hardware description, thus it can be argued that "chisel program" is about as meaningless as calling the above program a "html program", but nonetheless we will refer to a scala program building chisel as chisel programs.

We expand upon our first toolchain description:

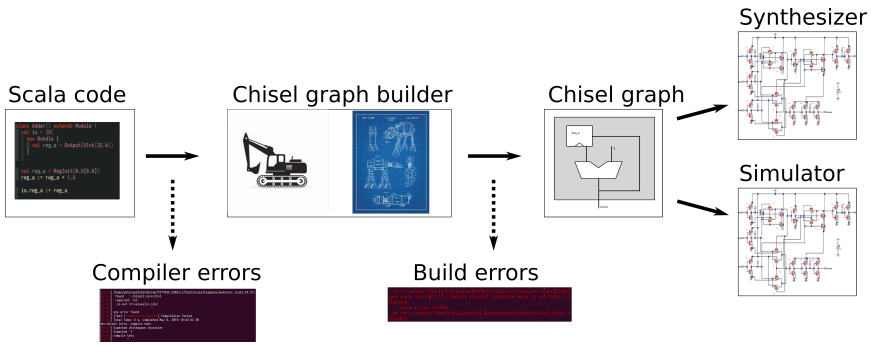

Figure 4: Toolchain flow using chisel.

2.1 Scala -> Chisel Graph Builder

The starting point, a scala program describing how to build a chisel graph. This program is not constrained in any way, it's able to do anything any other scala program does, it does not face any restrictions in order to use chisel.

In order to go from scala to a chisel graph builder the program must first be compiled, which exposes invalid programs (for instance typos, usage before declaration and similar)

2.2 Chisel Graph Builder -> Chisel Graph

After compiling the program can now be run. There are three common outcomes from the builder:

- The builder discovers an invalid circuit. This is analogous to a HTML tag missing.
- A wire is unconnected. During building the builder discovered that a wire was not connected. This is impossible to determine during compile time (unless you solve the halting problem). Thus, it is only detected during building of the circuit.
- The circuit is well-formed and can be instantiated.

2.3 Chisel Graph -> Simulator

After the circuit is verified it can now be used by a simulator. Several simulators are available and have different advantages and disadvantages. If nothing else is specified, the backend used is FIRRTL.

The simulator allows us to test how our circuit will react as its inputs are changed, allowing us to verify the correctness of our designs.

This is **HUGE** in HDL land as this guarantee does not hold for many HDLs (including VHDL and Verilog)! In the days of yore it was very common to see circuits being well-behaved in the simulator and misbehaving on an FPGA, but this is not the case with Chisel. The reason for this is that chisel adopts a fully synchronous model, but it is sufficient that you know that chisel won't lie to you like VHDL would.

2.4 Chisel Graph -> Hardware

This is not part of the course, but for the interested the approach here is to generate verilog from the chisel graph which is then used in a vendor specific toolchain for FPGAs or even ASICs.

These toolchains are generally not very fun to use, not only because they are made by very very evil people, but because hardware is a difficult, complex and complicated domain.

You can now take a look at the chisel introduction.