Getting Started with Chisel

Jonathan Bachrach, Vincent Lee
EECS Department, UC Berkeley
{jrb}@eecs.berkeley.edu

May 22, 2015
Chapter 1

Chisel Installation

1.1 Introduction

This chapter is an installation guide for Chisel (Constructing Hardware In a Scala Embedded Language) and is intended to prepare your system for subsequent tutorials. Chisel is a hardware construction language embedded in the high-level programming language Scala.

1.1.1 Development Tool Installation

If you are running Mac or a variant of Linux, you will need to install the appropriate tools for your OS, which are described in the following sections:

MacOSX

1. Install XCODE, including console tools.

Linux

Install the following packages:

1. g++-4.8
2. openjdk-7-jre

using

```bash
sudo apt-get install
```

1.2 Setting Up the Tutorial

In subsequent tutorials, you will be using the files distributed in the chisel-tutorial repository. To obtain these tutorials files, cd to the directory = $DIR where you want to place the Chisel tutorial and type:

```bash
cd $DIR
git clone https://github.com/ucb-bar/chisel-tutorial.git
```

Your copy of the Chisel Tutorial repository will then be in $DIR/chisel-tutorial. Define this as a variable in your bash environment named $TUT_DIR.

This is the Chisel tutorial directory structure you should see, which is explained more in the next tutorial:

<table>
<thead>
<tr>
<th>chisel-tutorial/</th>
</tr>
</thead>
<tbody>
<tr>
<td>Makefile</td>
</tr>
<tr>
<td>examples/</td>
</tr>
<tr>
<td>Makefile</td>
</tr>
<tr>
<td>build.sbt</td>
</tr>
<tr>
<td>Accumulator.scala</td>
</tr>
<tr>
<td>problems/</td>
</tr>
<tr>
<td>Makefile</td>
</tr>
<tr>
<td>build.sbt</td>
</tr>
<tr>
<td>Counter.scala</td>
</tr>
<tr>
<td>solutions/</td>
</tr>
<tr>
<td>Makefile</td>
</tr>
<tr>
<td>build.sbt</td>
</tr>
<tr>
<td>Counter.scala</td>
</tr>
</tbody>
</table>

The following tutorials will explain features of Chisel by presenting source code examples. The repository is split into examples, problems, and solutions, where the problems have some piece of the design for you to fill out and where the examples and solutions are meant to be complete designs that should pass the given tests. In order to run either, you simply need to change directory into the appropriate subdirectory and type make of the particular lesson name. We will use the repository to first test out if your machine is set up to use Chisel.

To test your Chisel distribution and verify that your system contains all the correct tools, run the following commands:

```bash
cd $TUT_DIR/examples
make Parity.out
```

This will run a test build and will take a minute before it completes. If your system is set up correctly, you should see a message [success] followed by the total time of the run, and date and time of completion. If you see a success than your system has
been set up correctly and you can continue to the next tutorial where we will explain more about the basics of Chisel.

1.3 The Tutorials

For these tutorials, we assume basic knowledge of digital circuits and blocks. Tutorial 1 will guide you through a quick compilation of the emulator and Verilog generation, and explain some basic constructs such as register and combinational logic. Tutorial 2 will explain the basics of Chisel. Tutorial 3 will explain how to use basic primitive types and logical operations that are used in Chisel and how to use them in context of several examples. Tutorial 4 will explain how to instantiate components and apply parametrization. Tutorial 5 will explain how to use the Chisel test harness. Tutorial 6 will explain how to set up your own Chisel project and how to build it. Tutorial 7 will revisit conditional register updates and explain how to construct memories. Finally, tutorial 8 will introduce how to use Scala constructs such as if...else and for loops.

Along the way there are assignments highlighted with red titles. These assignments are built around files in the tutorial problems directory. In order to check successful completion of the entire set of getting started assignments run:

```
cd $TUT_DIR/problems
make getting-started
```

until no error appears.

The following set of tutorials were written using the build settings Scala version 2.11 and Chisel version 2.2.
Chapter 2

The Basics

2.0.1 The Chisel Directory Structure

Once you have acquired the tutorial files you should see the following Chisel tutorial directory structure under $TUT_DIR:

```
chisel-tutorial/
  Makefile
  examples/  # chisel examples
    Makefile  # for running examples
    build.sbt # project description
    Accumulator.scala ...
  problems/  # skeletal files for tutorial problems
    Makefile  # for running / testing problems
    build.sbt # project description
    Counter.scala ...
  solutions/ # solutions to problems
    Makefile  # for running solutions
    build.sbt # project description
    Counter.scala ...
```

Chisel source files are distributed between examples, problems, and solutions directories. The tutorial contains the files that you will be modifying under problems/ while the solutions/ folder contains the reference implementations for each of the problems. Finally, examples/ contains source to the complete examples given in this tutorial.

Finally, the build.sbt files contain the build configuration information used to specify what version of Chisel to make your project with.

2.1 Running Your First Chisel Build

In this section, we explain how to run your first build to explore what Chisel has to offer. We will go through a simple example for a GCD module and familiarize ourselves with the source files, simulation, and Verilog generation. More comprehensive details will follow in subsequent sections of the tutorial.

2.1.1 The Chisel Source Code

Now that you are more familiar with what your Chisel directory structure contains, let’s start by exploring one of the Chisel files. Change directory into the examples/ directory and open up the GCD.scala file with your favorite text editor.

You will notice that file is already filled out for you to perform the well known GCD algorithm and should look like:

```
package TutorialExamples

import Chisel...

class GCD extends Module {
  val io = new Bundle {
    val a = UInt(INPUT, 16)
    val b = UInt(INPUT, 16)
    val e = Bool(INPUT)
    val z = UInt(OUTPUT, 16)
    val v = Bool(OUTPUT)
  } ...

  val x = Reg(UInt())
  val y = Reg(UInt())
  when (x > y) { x := x - y }
  unless (x > y) { y := y - x }
  when (io.e) { x := io.a; y := io.b }
  io.z := x
  io.v := y === UInt(0)
}
```

The first thing you will notice is the import Chisel._ declaration; this imports the Chisel library files that allow us to leverage Scala as a hardware construction language. After the import declarations you will see the Scala class definition for the Chisel component you are implementing. You can think of this as almost the same thing as a module declaration in Verilog.

Next we see the I/O specification for this component in the val io = new Bundle(...) definition. You will notice that the bundle takes several arguments as part of its construction, each with a specified type (UInt, Bool, etc.), a direction (either INPUT or OUTPUT), and a bit width. If a bit width is not
specified, Chisel will infer the appropriate bit width for you (in this case default to 1). The io Bundle is essentially a constructor for the component that we are constructing.

The next section of code performs the actual GCD computation for the module. The register declarations for x and y tell Chisel to treat x and y as a register of type UInt().

```scala
val x = Reg(UInt()) // declares x as UInt register
val y = Reg(UInt()) // declares y as UInt register
```

The when statement tells Chisel to perform the operation on a positive clock edge if the condition is true, treating the left hand assignments as synchronous. This is similar to how Verilog uses `always @ (posedge clk)` to specify synchronous logic.

Finally we see the output assignments for the computation for io.z and io.v. One particular thing to notice is that, we do not have to specify the width of x and y in this example. This is because Chisel does the bit width inference for you and sets these values to their appropriate widths based on the computation they are storing.

### 2.1.2 Running the Chisel Simulation

Now that we are familiar with the Chisel code for the GCD.scala file, let’s try to simulate it by generating the C++ models. Change directory into the $DIR/examples/ directory. Here you will see one lonely Makefile which we will call with:

```
make GCD.out
```

This will fire off the Chisel emulator that will run the simulation for the component defined in GCD.scala. If the simulation succeeds, you should see some debug output followed by:

```
PASSED
[success] Total time: 2 s, completed Feb 28, 2013 \ 8:14:37 PM
```

The debug output is generated by the test harness which composes the second half of the GCD.scala file. We will talk about this more later. In addition to the debug output, the build also creates C++ models which can be used to simulate and debug more complicated designs.

### 2.1.3 Generating the Verilog

One of the most powerful features of Chisel is its ability to generate FPGA and ASIC Verilog from the Scala sources that you construct. To do this, change directory into the $DIR/examples/verilog/ directory and again run:

```
make GCD.v
```

This will start the Verilog generation for the GCD Chisel file. When the Verilog generation finishes, you should see a [success] message similar to the one you saw in the emulator and a new GCD.v file. If you open up GCD.v, you will find that Chisel has compiled GCD.scala into its equivalent Verilog source.

You will find that the Chisel compiler has generated an equivalent Verilog module that performs the GCD computation.

The Verilog source is roughly divided into three parts:

1. Module declaration with input and outputs
2. Temporary wire and register declaration used for holding intermediate values
3. Register assignments in `always @ (posedge clk)`

### 2.2 Combinational Logic

#### 2.2.1 The Scala Node: Declaring Wires

Constructing combinational logic blocks in Chisel is fairly straightforward; when you declare a val in Scala, it creates a node that represents the data that it is assigned to. As long as the value is not assigned to be a register type (explained later), this tells the Chisel compiler to treat the value as wire. Thus any number of these values can be connected and manipulated to produce the value that we want.

Suppose we want to construct a single full adder. A full adder takes two inputs a and b, and a carry in cin and produces a sum and carry out cout. The Chisel source code for our full adder will look something like:

```scala
class FullAdder extends Module {
  val io = new Bundle {
    val a = UInt(INPUT, 1)
    val b = UInt(INPUT, 1)
    val cin = UInt(INPUT, 1)
    val sum = UInt(OUTPUT, 1)
    val cout = UInt(OUTPUT, 1)
  }
  // Generate the sum
  val xor_b = a ^ b
  io.sum := xor_b ^ cin
  // Generate the carry
  val and_b = a & b
  val b_and_cin = b & cin
```

### 2.2 Combinational Logic
2.2. COMBINATIONAL LOGIC

\[ \text{val } \text{a.and.cin} = \text{io.a} \& \text{io.cin} \]
\[ \text{io.cout} := \text{a.and.b} | \text{b.and.cin} | \text{a.and.cin} \]

where \text{cout} is defined as a combinational function of inputs \text{a}, \text{b}, and \text{cin}.

You will notice that in order to access the input values from the \text{io} bundle, you need to first reference \text{io} since the input and output values belong to the \text{io} bundle. The \( | \), \&, and \(^\hat{\ }\) operators correspond to bitwise OR, AND, and XOR operations respectively.

The corresponding wires for each of these values is shown below in Figure 2.1. You will notice that each \text{val} corresponds to exactly one of the wires.

2.2.2 Bit Width Inference

If you don’t explicitly specify the width of a value in Chisel, the Chisel compiler will infer the bit width for you based on the inputs that define the value. Notice in the \text{FullAdder} definition, the widths for \text{a_xor_b}, \text{a_and_b}, \text{b_and_cin}, and \text{a_and_cin} are never specified anywhere. However, based on how the input is computed, Chisel will correctly infer each of these values are one bit wide since each of their inputs are the results of bitwise operations applied to one bit operands.

A quick inspection of the generated Verilog shows these values are indeed one bit wide:

\[
\text{module FullAdder(}
\text{input} [1:0] \text{io.a,}
\text{input} [1:0] \text{io.b,}
\text{input} [1:0] \text{io.cin,}
\text{output} \text{io.sum,}
\text{output} \text{io.cout);}
\]
\[
\text{wire} \text{T0;}
\text{wire} \text{a.and.cin;}
\]

Suppose we change the widths of the \text{FullAdder} to be 2 bits wide each instead such that the Chisel source now looks like:

\[
\text{class FullAdder extends Module {}
\text{val io = new Bundle {}
\text{val a = \text{UInt(INPUT, 2)}}
\text{val b = \text{UInt(INPUT, 2)}}
\text{val cin = \text{UInt(INPUT, 2)}}
\text{val sum = \text{UInt(OUTPUT, 2)}}
\text{val cout = \text{UInt(OUTPUT, 2)}}
\}}
\]
\[
\text{// Generate the sum}
\text{val a_xor_b = \text{io.a} ^ \text{io.b}}
\text{io.sum := \text{a_xor_b} ^ \text{io.cin}}
\]
\[
\text{// Generate the carry}
\text{val a_and_b = \text{io.a} \& \text{io.b}}
\text{val b_and_cin = \text{io.b} \& \text{io.cin}}
\text{val a_and_cin = \text{io.a} \& \text{io.cin}}
\text{io.cout := \text{a_and.b} | \text{b_and.cin} | \text{a_and.cin}}
\}
\]

As a result, the Chisel compiler should infer each of the intermediate values \text{a_xor_b}, \text{a_and_b}, \text{b_and_cin}, and \text{a_and_cin} are two bits wide. An inspection of the Verilog code correctly shows that Chisel inferred each of the intermediate wires in the calculation to be 2 bits wide.

\[
\text{module FullAdder{}
\text{input [1:0] \text{io.a,}
\text{input [1:0] \text{io.b,}
\text{input [1:0] \text{io.cin,}
\text{output [1:0] \text{io.sum,}
\text{output [1:0] \text{io.cout);}
\]
\[
\text{wire[1:0] T0;}
\text{wire[1:0] a.and.cin;}
\text{wire[1:0] T1;}
\text{wire[1:0] b.and.cin;}
\text{wire[1:0] a.and.b;}
\text{wire[1:0] T2;}
\text{wire[1:0] a.xor.b;}
\]
\[
\text{assign \text{io.cout} = \text{T0;}}
\text{assign \text{T0} = \text{T1} | \text{a.and.cin;}}
\text{assign \text{a.and.cin} = \text{io.a} & \text{io.cin;}}
\text{assign \text{a.and.b} = \text{io.a} & \text{io.b;}}
\text{assign \text{io.sum} = \text{T2;}}
\text{assign \text{T2} = \text{a.xor.b} ^ \text{io.cin;}}
\text{assign \text{a.xor.b} = \text{io.a} ^ \text{io.b;}}
\endmodule
2.3 Using Registers

Unlike Verilog, specifying a register in Chisel tells the compiler to actually generate a positive edge triggered register. In this section we explore how to instantiate registers in Chisel by constructing a shift register.

In Chisel, when you instantiate a register there are several ways to specify the connection of the input to a register. As shown in the GCD example, you can "declare" the register and assign what it's input is connected to in a `when...` block or you can simply pass the value that the register is clocking as a parameter to the register.

If you choose to pass a next value to the register on construction using the `next` named parameter, it will clock the new value every cycle unconditionally:

```chisel
// Clock the new register value on every cycle
val y = io.x
val z = Reg(next = y)
```

If we only want to update if certain conditions are met we use a `when` block to indicate that the registers are only updated when the condition is satisfied:

```chisel
// Clock the new register value when the condition a > b
val x = Reg(UInt())
when (a > b) { x := y }
    .elsewhen (b > a) { x := z }
    .otherwise { x := w }
```

It is important to note that when using the conditional method, the values getting assigned to the input of the register match the type and bitwidth of the register you declared. In the unconditional register assignment, you do not need to do this as Chisel will infer the type and width from the type and width of the input value.

The following sections show how these can be used to construct a shift register.

2.3.1 Unconditional Register Update

Suppose we want to construct a basic 4 bit shift register that takes a serial input and generates a serial output. For this first example we won’t worry about a parallel load signal and will assume the shift register is always enabled. We also will forget about the register reset signal.

If we instantiate and connect each of these 4 registers explicitly, our Chisel code will look something like:

```chisel
class ShiftRegister extends Module {
    val io = new Bundle {
        val in = UInt(INPUT, 1)
        val out = UInt(OUTPUT, 1)
    }
    val r0 = Reg(next = io.in)
    val r1 = Reg(next = r0)
    val r2 = Reg(next = r1)
    val r3 = Reg(next = r2)
    io.out := r3
}
```

If we take a look at the generated Verilog, we will see that Chisel did indeed map our design to a shift register. One thing to notice is that the clock signal and reset signals are implicitly attached to our design.

```verilog
module ShiftRegister(input clk, input reset, input io_in, output io_out);
    reg[0:0] r3;
    reg[0:0] r2;
    reg[0:0] r1;
    reg[0:0] r0;
    assign io_out = r3;
    always @posedge clk begin
        r3 <= r2;
        r2 <= r1;
        r1 <= r0;
        r0 <= io.in;
    end
endmodule
```

2.3.2 Conditional Register Update

As mentioned earlier, Chisel allows you to conditionally update a register (use an enable signal) using the `when`, `.elsewhen`, `.otherwise` block. Suppose we add an enable signal to our shift register, that allows us to control whether data is shift in and out on a given cycle depending on an enable input signal. The new shift register now looks like:

```chisel
class ShiftRegister extends Module {
    val io = new Bundle {
        val in = UInt(INPUT, 1)
        val enable = Bool(INPUT)
        val out = UInt(OUTPUT, 1)
    }
    val r0 = Reg(UInt())
    val r1 = Reg(UInt())
}
```

```verilog
module ShiftRegister(input clk, input reset, input io_in, output io_out);
    reg[0:0] r3;
    reg[0:0] r2;
    reg[0:0] r1;
    reg[0:0] r0;
    assign io_out = r3;
    always @posedge clk begin
        r3 <= io.enable ? r2 : r3;
        r2 <= r1;
        r1 <= r0;
        r0 <= io.in;
    end
endmodule
```
2.3. USING REGISTERS

val r2 = Reg(UInt())
val r3 = Reg(UInt())

when (io.enable) {
  r0 := io.in
  r1 := r0
  r2 := r1
  r3 := r2
}

io.out := r3

Notice that it is not necessary to specify an otherwise condition as Chisel will correctly infer that the old register value should be preserved otherwise.

2.3.3 Register Reset

Chisel allows you to specify a synchronous reset to a certain value by specifying an additional parameter when you first declare them. In our shift register, let’s add a reset capability that resets all the register values to zero synchronously. To do this we need to provide our register declarations a little more information using the init parameter with what value we want on a synchronous reset:

```scala
class ShiftRegister extends Module {
  val io = new Bundle {
    val in = UInt(INPUT, 1)
    val enable = Bool(INPUT)
    val out = UInt(OUTPUT, 1)
  }

  // Register reset to zero
  val r0 = Reg(init = UInt(0, width = 1))
  val r1 = Reg(init = UInt(0, width = 1))
  val r2 = Reg(init = UInt(0, width = 1))
  val r3 = Reg(init = UInt(0, width = 1))

  when (io.enable) {
    r0 := io.in
    r1 := r0
    r2 := r1
    r3 := r2
  }
  io.out := r3
}
```

This will generate slightly different looking Verilog source code but will still function the same as the previous implementation of the shift register with reset.

2.3.4 Sequential Circuit Problem

The following exercises can be found in your $TUT_DIR/problems/ folder. You will find that some parts of the tutorial files have been completed for you and the section that you need to will need to complete is indicated in the file. The solutions to each of these exercises can be found in the $TUT_DIR/solutions/ folder.

The first tutorial problem is to write a sequential circuit that sums in values. You can find the template in $TUT_DIR/problems/Accumulator.scala including a stubbed out version of the circuit:

```scala
class Accumulator extends Module {
  val io = new Bundle {
    val in = UInt(INPUT, 1)
    val out = UInt(OUTPUT, 8)
  }

  // flush this out ...
  io.out := UInt(0)
}
```

and a complete tester that confirms that you have successfully designed the circuit. Run

```
make Accumulator.out
```

until your circuit passes the tests.
Chapter 3

Basic Types and Operations

3.1 Chisel Assignments and Re-assignments

When you first define a value in Chisel, we use the = operator in order to tell Chisel to allocate the value for the first time. On every subsequent reassignment to the value, we must use a := when reassigning the value.

Since we are constructing a digital circuit, the notion of reassignment does not make much sense since connections between circuit nodes only need to be specified once. However, there are some cases when we will need to perform reassignment to a value in Chisel since it is compiled sequentially unlike Verilog. Thus it may be necessary to perform re-assignment when a value or connection is not known until later in the Chisel source.

A simple example of when reassignment is necessary is in the construction of the top level I/O for your module; the values of the output are not immediately known at the time of declaration.

Consider the simple FullAdder circuit from previous tutorial that determines the sum sum and carry out cout given two values a and b, and a carry in cin.

```scala
class FullAdder extends Module {
  val io = new Bundle {
    // first definition of io values so use =
    val a = UInt(INPUT, 1)
    val b = UInt(INPUT, 1)
    val cin = UInt(INPUT, 1)
    val sum = UInt(OUTPUT, 1)
    val cout = UInt(OUTPUT, 1)
  }
  // Generate the sum
  val a_xor_b = io.a ^ io.b
  // Reassignment to io.sum so use :=
  io.sum := a_xor_b ^ io.cin
  // Generate the carry
  val a_and_b = io.a & io.b
  val b_and_cin = io.b & io.cin
  val a_and_cin = io.a & io.cin
  // reassignment to io.cout so use :=
  io.cout := a_and_b | b_and_cin | a_and_cin
}
```

In this example we make sure to use the := reassignment for the io.sum and io.cout output values because we only know what they're values are later in the code and not at the time of construction of the io Bundle. All other values in this example use the = assignment operator since they need to be created.

In general, the rule of thumb is to use the reassignment operator := if the value already has been assigned by the = operator, otherwise the = operator should be used. Note that if you do not use the = or := operators correctly you will get an error when you try and compile your design.

3.2 The Chisel UInt Class

In the previous examples we have been using the UInt type which is an unsigned integer as the type for all of our values. For many of the basic computations in Chisel the UInt class is sufficient. The following example shows some of the commonly used UInt operations in the context of a simple ALU.

```scala
class BasicALU extends Module {
  val io = new Bundle {
    val a = UInt(INPUT, 4)
    val b = UInt(INPUT, 4)
    val opcode = UInt(INPUT, 4)
    val output = UInt(OUTPUT, 4)
  }
  io.output := UInt(0)
  when (io.opcode === UInt(0)) {
    io.output := io.a // pass A
  } .elsewhen (io.opcode === UInt(1)) {
    io.output := io.b // pass B
  } .elsewhen (io.opcode === UInt(2)) {
  }
```

1The UInt class definition for Chisel can be found in the /chisel/src/main folder in the compiler source repository, not the chisel-tutorial. You can obtain the Chisel source by cloning https://github.com/ucb-bar/chisel.git

2We ignore overflow and underflow in this example.
CHAPTER 3. BASIC TYPES AND OPERATIONS

You will notice that there are multiple reassignments to `io.output` inside a `when` block which indicates that the value of `io.output` can take many different values depending on the `io.opcode` in this example. Also notice that in order to specify constants to add to our operands, we must also specify them as a `UInt` type as `UInt` operations on different type operands is not allowed.

```
// Specify that 1 is a UInt type
io.output := io.a + UInt(1)
```

A list of commonly used `UInt` operations is given in the table below:

<table>
<thead>
<tr>
<th>Operand</th>
<th>Operation</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Add</td>
<td><code>UInt</code></td>
</tr>
<tr>
<td>-</td>
<td>Subtract</td>
<td><code>UInt</code></td>
</tr>
<tr>
<td>*</td>
<td>Multiply</td>
<td><code>UInt</code></td>
</tr>
<tr>
<td>/</td>
<td><code>UInt</code> Divide</td>
<td><code>UInt</code></td>
</tr>
<tr>
<td>-6</td>
<td>Bitwise Negation</td>
<td><code>UInt</code></td>
</tr>
<tr>
<td>^</td>
<td>Bitwise XOR</td>
<td><code>UInt</code></td>
</tr>
<tr>
<td>&amp;</td>
<td>Bitwise AND</td>
<td><code>UInt</code></td>
</tr>
<tr>
<td>!</td>
<td>Bitwise OR</td>
<td><code>Bool</code></td>
</tr>
<tr>
<td>==</td>
<td>Equal</td>
<td><code>Bool</code></td>
</tr>
<tr>
<td>!=</td>
<td>Not Equal</td>
<td><code>Bool</code></td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater</td>
<td><code>Bool</code></td>
</tr>
<tr>
<td>&lt;</td>
<td>Less</td>
<td><code>Bool</code></td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater or Equal</td>
<td><code>Bool</code></td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less or Equal</td>
<td><code>Bool</code></td>
</tr>
</tbody>
</table>

3.2.2 Bit Concatenation

Chisel also allows you to easily concatenate bits together using `Cat`. Suppose you have a data bus that you would like to drive with two separate words `A` and `B`. In order to concatenate these two values together we simply say:

```
val A = UInt(width = 32)
val B = UInt(width = 32)
val bus = Cat(A, B) // concatenate A and B
```

Again, the first argument to `Cat` will be placed in the high part while the second argument gets the low part of `bus`. Thus for this example bits 0 to 31 of `bus` correspond to `B`, while bits 32 to 63 correspond to `A`.

3.2.3 LFSR16 Problem

In this assignment, write the LFSR16 circuit as shown below:

```scala
class ByteSelector extends Module {
  val io = new Bundle {
    val in = UInt(INPUT, 32)
    val offset = UInt(INPUT, 2)
    val out = UInt(OUTPUT, 8)
  }
  io.out := UInt(0, width = 8)
  when (io.offset === UInt(0)) {
    io.out := io.in(7,0) // pull out lowest byte
  }
  when (io.offset === UInt(1)) {
    io.out := io.in(15,8) // pull out second byte
  }
  when (io.offset === UInt(2)) {
    io.out := io.in(23,16) // pull out third byte
  }
  .otherwise {
    io.out := io.in(31,24) // pull out highest byte
  }
}
```
by filling in the following module:

```scala
class LFSR16 extends Module {
  val io = new Bundle {
    val inc = Bool(INPUT)
    val out = UInt(OUTPUT, 16)
  }
  // ...
  io.out := UInt(0)
}
```

found in `$TUT_DIR/problems/LFSR16.scala`. Make sure to define and initialize an internal register to one and update it when `inc` is asserted. Use bit concatenation and bit extraction in conjunction with the xor operator `^`. Run

```
make LFSR16.out
```

until your circuit passes the tests.

### 3.2.4 UInt Operation Bit Inference

Note that for some operations such as addition and multiplication, that number of resulting bits of the computation can be greater than the number of bits for the operands.

Consider the following example where we multiply two 16 bit numbers `A` and `B` together. Note that the product of two 16 bit numbers is at worst 32 bits wide.

```scala
class HiLoMultiplier() extends Module {
  val io = new Bundle {
    val A = UInt(INPUT, 16)
    val B = UInt(INPUT, 16)
    val Hi = UInt(OUTPUT, 16)
    val Lo = UInt(OUTPUT, 16)
  }
  io.Lo := mult(15, 0)
  io.Hi := mult(31, 16)
}
```

Notice that we never specify the width of the value `mult` anywhere in the Chisel source. Normally if we performed this in Verilog we would have had to specify the width beforehand. But a look at the generated Verilog for this example shows that Chisel correctly inferred the `mult` value to be 32 bits wide:

```verilog
module HiLoMultiplier(  
  input [15:0] io_A,  
  input [15:0] io_B,  
  output[15:0] io.Hi,  
  output[15:0] io.Lo);  

wire[15:0] T0;  
wire[31:0] mult; // Chisel infers this to be 32 bits  
wire[15:0] T1;  
assign io.Lo = T0;  
assign T0 = mult[4'hf:1'h0];  
assign io.Hi = T1;  
assign T1 = mult[5'h1f:5'h10];  
endmodule
```

As we get to more complicate designs, it will become more clear that bit inference in Chisel is a very powerful feature that makes constructing hardware more efficient. A list of common bit inferences is shown below for commonly used operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result Bit Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z = X + Y</td>
<td>max(Width(X), Width(Y))</td>
</tr>
<tr>
<td>Z = X - Y</td>
<td>max(Width(X), Width(Y))</td>
</tr>
<tr>
<td>Z = X &amp; Y</td>
<td>max(Width(X), Width(Y))</td>
</tr>
<tr>
<td>Z = X</td>
<td>Y</td>
</tr>
<tr>
<td>Z = X ^ Y</td>
<td>max(Width(X), Width(Y))</td>
</tr>
<tr>
<td>Z = ~X</td>
<td>Width(X)</td>
</tr>
<tr>
<td>Z = Mux(C, X, Y)</td>
<td>max(Width(X), Width(Y))</td>
</tr>
<tr>
<td>Z = X &lt;&lt; n</td>
<td>Width(X) + n</td>
</tr>
<tr>
<td>Z = X &gt;&gt; n</td>
<td>Width(X) - n</td>
</tr>
<tr>
<td>Z = Cat(X, Y)</td>
<td>Width(X) + Width(Y)</td>
</tr>
<tr>
<td>Z = Fill(n, x)</td>
<td>Width(X) + n</td>
</tr>
</tbody>
</table>

### 3.3 The Chisel Bool Class

The Bool class in Chisel is used to represent the result of logical expressions and takes either the values `true` or `false`. These can be used in conditional statements such as when blocks.

```scala
val change = io.a === io.b // change gets Bool type  
when (change) {  
  // exec if change is true  
  ....  
  }.otherwise {  
    ....  
  }
```

You can instantiate a Bool value like this:

```scala
val true_value = Bool(true)  
val false_value = Bool(false)
```
3.4 Casting Between Types

When assigning values, it is required that you assign a value of the same type. For instance, if you try to assign a Bool type to an output value that is expecting a UInt type, you will get an error.

```chisel
...  
val io = new Bundle {
  val in = UInt(INPUT, 2)
  val out = UInt(OUTPUT, 1)
}
// attempted Bool assignment to UInt
io.out := (in === UInt(0)) // UInt cast
...  
```

The correct way to perform the intended operation is to cast the resulting Bool type to a UInt using the `toUInt()` cast. The correct Chisel code will look like:

```chisel
...  
val io = new Bundle {
  val in = UInt(INPUT, 2)
  val out = UInt(OUTPUT, 1)
}
io.out := (in === UInt(0)).toUInt() // UInt cast
...  
```

Some of the common casts that you may use are:

- `toUInt()`
- `toSInt()`
- `toBool()`
Chapter 4

Instantiating Modules

4.1 Module Instantiation

Like other hardware description languages, Chisel allows fairly straightforward module instantiation to enable modularity and hierarchy. In Chisel, instantiating a Module class is the equivalent to instantiating a module in Verilog. To do this, we simply use a call to Module with module created with the Scala new keyword in order to indicate that we are instantiating a new module. We want to make sure we assign this to a value so that we can reference its input and outputs which we also need to connect.

For example, suppose we would like to construct a 4-bit adder using multiple copies of the FullAdder module. as shown in the Figure 1. The Chisel source code is shown below.

```
// A 4-bit adder with carry in and carry out
class Adder4 extends Module {
    val io = new Bundle {
        val A = UInt(INPUT, 4)
        val B = UInt(INPUT, 4)
        val Cin = UInt(INPUT, 1)
        val Sum = UInt(OUTPUT, 4)
        val Cout = UInt(OUTPUT, 1)
    }
    // Adder for bit 0
    val Adder0 = Module(new FullAdder())
    Adder0.io.a := io.A(0)
    Adder0.io.b := io.B(0)
    Adder0.io.cin := io.Cin
    val s0 = Adder0.io.sum
    // Adder for bit 1
    val Adder1 = Module(new FullAdder())
    Adder1.io.a := io.A(1)
    Adder1.io.b := io.B(1)
    Adder1.io.cin := Adder0.io.cout
    val s1 = Cat(Adder1.io.sum, s0)
    // Adder for bit 2
    val Adder2 = Module(new FullAdder())
    Adder2.io.a := io.A(2)
    Adder2.io.b := io.B(2)
    Adder2.io.cin := Adder1.io.cout
    val s2 = Cat(Adder2.io.sum, s1)
    // Adder for bit 3
    val Adder3 = Module(new FullAdder())
    Adder3.io.a := io.A(3)
    Adder3.io.b := io.B(3)
    Adder3.io.cin := Adder2.io.cout
    io.Sum := Cat(Adder3.io.sum, s2).toUInt()
    io.Cout := Adder3.io.cout
}
```

In this example, notice how when referencing each module I/O we must first reference the io that contains the ports for the I/Os. Again, note how all assignments to the module I/Os use a reassignment operator :=. When instantiating modules, it is important to make sure that you connect all the input and output ports. If a port is not connected, the Chisel compiler may optimize away portions of your design that it find unnecessary due to the unconnected ports and throw errors or warnings.

4.2 The Vec Class

The Vec class allows you to create an indexable vector in Chisel which can be filled with any expression that returns a chisel data type. The general syntax for a Vec declaration is given by:

```
val myVec = Vec.fill( <number of elements> ) { <data type> }
```
CHAPTER 4. INSTANTIATING MODULES

Where <number of elements> corresponds to how long the vector is and <data type> corresponds to what type of class the vector contains.

For instance, if we wanted to instantiate a 10 entry vector of 5 bit UInt values, we would use:

```scala
val ufix5_vec10 := Vec.fill(10) { UInt(width = 5) }
```

If we want to define a vector of registers...

```scala
val reg_vec32 := Vec.fill(32){ Reg() }
```

In order to assign to a particular value of the Vec, we simply assign the target value to the vector at a specified index. For instance, if we wanted to assign a UInt value of zero to the first register in the above example, the assignment would look like:

```scala
reg_vec32(1) := UInt(0)
```

To access a particular element in the vector at some index, we specify the index of the vector. For example, to extract the 5th element of the register vector in the above example and assign it to some value `reg5`, the assignment would look like:

```scala
val reg5 = reg_vec(5)
```

The syntax for the Vec class is slightly different when instantiating a vector of modules. When instantiating a vector of modules the data type that is specified in the braces is slightly different than the usual primitive types. To specify a vector of modules, we use the `io` bundle when specifying the type of the vector. For example, in order to specify a Vec with 16 modules, say FullAdders in this case, we would use the following declaration:

```scala
val FullAdders = Vec.fill(16){ Module(new FullAdder()).io }
```

Note that in this example, we keep track of the sum output in a Vec of Bools. This is because Chisel does not support bit assignment directly. Thus in order to get the n-bit wide sum in the above example, we use an n-bit wide Vec of Bools and then cast it to a UInt(). Note that it must first be casted to the Bits() type before casting it to UInt().

4.3 Parametrization

In the previous Adder example, we explicitly instantiated four different copies of a FullAdder and wired up the ports. But suppose we want to generalize this structure to an n-bit adder. Like Verilog, Chisel allows you to pass parameters to specify certain aspects of your design. In order to do this, we add a parameter in the Module declaration to our Chisel definition. For a carry ripple adder, we would like to parametrize the width to some integer value n as shown in the following example:

```scala
// A n-bit adder with carry in and carry out
class Adder(n: Int) extends Module {
  val io = new Bundle {
    val A = UInt(INPUT, n)
    val B = UInt(INPUT, n)
    val Cin = UInt(INPUT, 1)
    val Sum = UInt(OUTPUT, n)
    val Cout = UInt(OUTPUT, 1)
  }
  // create a vector of FullAdders
  val FAs = Vec.fill(n){ Module(new FullAdder()).io }
  // define carry and sum wires
  val carry = Vec.fill(n+1){ UInt(width = 1) }
  val sum = Vec.fill(n){ Bool() }
  // first carry is the top level carry in
  carry(0) := io.Cin
  // wire up the ports of the full adders
  for(i <- 0 until n) {
    FAs(i).a := io.A(i)
    FAs(i).b := io.B(i)
    FAs(i).cin := carry(i)
    carry(i+1) := FAs(i).cout
    sum(i) := FAs(i).sum.toBool()
  }
  io.Sum := sum.toBits().toUInt()
  io.Cout := carry(n)
}
```

Notice we use the keyword `new` in the vector definition before the module name FullAdder. For how to actually access the `io` on the vector modules, refer to the next section.

4.2.1 Vec Shift Reg Problem

The next assignment is to construct a simple bit shift register. The following is a template from `$TUT_DIR/problems/VecShiftRegisterSimple.scala`:

```scala
class VecShiftRegisterSimple extends Module {
  val io = new Bundle {
    val in = UInt(INPUT, 8)
    val out = UInt(OUTPUT, 8)
  }
  val delays = Vec.fill(4){ Reg(UInt(width = 8)) }
  ...
  io.out := UInt(0)
}
```

where out is a four cycle delayed copy of values on in.
You will notice that modules are instantiated in a Vec class which allows us to iterate through each module when assigning the ports connections to each FullAdder. This is similar to the generate statement in Verilog. However, you will see in more advanced tutorials that Chisel can offer more powerful variations.

Instantiating a parametrized module is very similar to instantiating an unparametrized module except that we must provide arguments for the parameter values. For instance, if we wanted to instantiate a 4-bit version of the Adder module we defined above, it would look like:

```scala
val adder4 = Module(new Adder(4))
```

We can also instantiate the Adder by explicitly specifying the value of its parameter n like this:

```scala
val adder4 = Module(new Adder(n = 4))
```

Explicitly specifying the parameter is useful when you have a module with multiple parameters. Suppose you have a parametrized FIFO module with the following module definition:

```scala
class FIFO(width: Int, depth: Int) extends Module {...}
```

You can explicitly specify the parameter values in any order:

```scala
val fifo1 = Module(new FIFO(16, 32))
val fifo2 = Module(new FIFO(width = 16, depth = 32))
val fifo3 = Module(new FIFO(depth = 32, width = 16))
```

All of the above definitions pass the same parameters to the FIFO module. Notice that when you explicitly assign the parameter values, they can occur in any order you want such as the definition for fifo3.

### 4.4 Advanced Parametrization

Although parameters can be passed explicitly through a Module’s constructor, this technique does not scale when parameterizing large designs with many generic components. For a more detailed explanation of why a better parameterization method is needed, please see the Advanced Parameterization Manual. In addition, this manual explains heuristics for how to organize and parameterize large designs, which we highly recommend one reads prior to using this functionality in a design. The following, however, is a basic introduction.

Every Module has its own params object, which acts as a dictionary. Querying this object is shown below.

```scala
val width = params[Int]('width')
```

If params is queried and no parameter matches the query, Chisel throws a ParameterUndefinedException. Notice the query return type must be provided.

Suppose a parent Module wants to override or add parameters to its child’s params object. This case requires adding a partial function (a Scala way of defining key-value mappings) to the Child Module constructor:

```scala
class Parent extends Module {
  val io = new Bundle {...}
  val width = params[Int]('width') // returns 10
  // Partial function is added to Module constructor
  val child = Module(new Child, {'n' => 40})
}
class Child extends Module {
  val io = new Bundle {...}
  val width = params[Int]('width') // returns 10
}
```

An example which is impossible to do with simple parameterization, but simple with the advanced parameterization, is when using a generic Mesh generator with a custom MyRouter Module. The existing source code might look like:

```scala
class Mesh(routerConstructor: () => Router, n:Int) extends Module {
  val io = new Bundle {...}
  val width = params[Int]('width') // returns 10
  val n = params[Int]('n') // returns 20
  // Function is added to Module constructor
  val child = Module(new Child, ('n' => 40))
}
class Child extends Module {
  val io = new Bundle {...}
  val width = params[Int]('width') // returns 10
  val n = params[Int]('n') // returns 40
}
```

```scala
class Parent extends Module {
  val io = new Bundle {...}
  val width = params[Int]('width') // returns 10
  val n = params[Int]('n') // returns 20
  // Function is added to Module constructor
  val child = Module(new Child, ('n' => 40))
}
class Child extends Module {
  val io = new Bundle {...}
  val width = params[Int]('width') // returns 10
  val n = params[Int]('n') // returns 40
}
```
a parameter, RoutingFunction that we want to sweep for a design space evaluation. Using the simple parameterization method would require a change to the Mesh Module’s constructor to include RoutingFunction.

Alternatively, one can use the params object to implicitly pass the RoutingFunction:

```scala
class MyRouter extends Module {
  val io = new Bundle { ... }
  val myRoutingFunction = params[RoutingFunction]('r')
  ...
}
class Top extends Module {
  val io = new Bundle { ... }
  val mesh = Module(new Mesh(() => new MyRouter), {'r' => new RoutingFunction})
}
```

Thus if A=10, B=14, and select was true, the value of out would be assigned 10. Notice how using the Mux primitive type abstracts away the logic structures required if we had wanted to implement the multiplexer explicitly.

### 4.5.2 Parameterized Width Adder Problem

The next assignment is to construct an adder with a parameterized width and using the built-in addition operator +. The following is a template from $TUT_DIR/problems/Adder.scala:

```scala
class Adder(val w: Int) extends Module {
  val io = new Bundle {
    val in0 = UInt(INPUT, 1)
    val in1 = UInt(INPUT, 1)
    val out = UInt(OUTPUT, 1)
  }
  io.out := UInt(0)
}
```

where out is sum of w width unsigned inputs in0 and in1. Notice how val is added to the width parameter value to allow the width to be accessible from the tester as a field of the adder module object. Run

```bash
make Adder.out
```

until your circuit passes the tests.

---

### 4.5 Built In Primitives

Like other HDL, Chisel provides some very basic primitives. These are constructs that are built into the Chisel compiler and come for free. The Reg, UInt, and Bundle classes are such primitives that has already been covered. Unlike Module instantiations, primitive do not require explicit connections of their io ports to use. Other useful primitive types include the Mem and Vec classes which will be discussed in a more advanced tutorial. In this tutorial we explore the use of the Mux primitive.

#### 4.5.1 The Mux Class

The Mux primitive is a two input multiplexer. In order to use the Mux we first need to define the expected syntax of the Mux class. As with any two input multiplexer, it takes three inputs and one output. Two of the inputs correspond to the data values A and B that we would like to select which can be any width and data type as long as they are the same. The third input select which is a Bool type determines which one to output. A select value of true will output the first value A, while a select value of false will pass B.

```scala
val out = Mux(select, A, B)
```
Chapter 5

Writing Scala Testbenches

5.1 The Scala Testbench Simulation

Chisel’s Scala based testbench is the first line of defense against simple bugs in your design. The Scala testbench uses several unique Chisel constructs to perform this. To see how this works, let’s first explore a simple example.

5.1.1 Scala Testbench Example

Below is the ByteSelector.scala module definition from the previous tutorial and the corresponding Chisel test harness.

```scala
package TutorialExamples

import Chisel._

class ByteSelector extends Module {
  val io = new Bundle {
    val in = UInt(INPUT, 32)
    val offset = UInt(INPUT, 2)
    val out = UInt(OUTPUT, 8)
  }

  io.out := UInt(0, width = 8)
  when (io.offset === UInt(0)) {
    io.out := io.in(7,0)
  } .elsewhen (io.offset === UInt(1)) {
    io.out := io.in(15,8)
  } .elsewhen (io.offset === UInt(2)) {
    io.out := io.in(23,16)
  } .otherwise {
    io.out := io.in(31,24)
  }
}

class ByteSelectorTests(c: ByteSelector) extends Tester(c) {
  val test_in = 12345678
  for (t <- 0 until 4) {
    poke(c.io.in, test_in)
    poke(c.io.offset, t)
    step(1)
    expect(c.io.out, (test_in >> (t * 8)) & 0xFF)
  }
}
```

In the test harness ByteSelectorTests we see that the test portion is written in Scala with some Chisel constructs inside a Tester class definition. The device under test is passed to us as a parameter.

In the for loop, the assignments for each input of the ByteSelector is set to the appropriate values using poke. For this particular example, we are testing the ByteSelector by hardcoding the input to some known value and checking if each of the 4 offsets returns the appropriate byte. To do this, on each iteration we generate appropriate inputs to the module and tell the simulation to assign this value to the input of the device we are testing:

```scala
val test_in = 12345678
for (t <- 0 until 4) {
  // set in of the DUT to be some known word
  poke(c.io.in, test_in)
  // set the offset of the DUT
  poke(c.io.offset, t)
  ...
}
```

Next we step the circuit. We next advance the simulation by calling the step function. This effectively advances the simulation one clock cycle in the presence of sequential logic.

```scala
step(1)
```

Finally, we check for expected outputs. In this case, we check the expected output of ByteSelector as follows:

```scala
expect(c.io.out, (test_in >> (t * 8)) & 0xFF)
```

This defines the reference output expected for this particular cycle of the simulation. Since the circuit we are testing is purely combinational, we expected that the output we define appears on any advancement of the simulation. The expect function will record either true or false after checking if the output generates the expected reference output. The
results of successive `expect`'s are `anded` into a `Tester` field called `ok` which starts out as `true`. The value of the `ok` field determines the success or failure of the tester execution.

Actually `expect` is defined in terms of `peek` roughly as follows:

```scala
def expect(data: Bits, expected: BigInt) = 
  ok = peek(data) == expected && ok
```

where `peek` gets the value of a signal from the DUT.

### 5.1.2 Simulation Debug Output

Now suppose we run the testbench for the `ByteSelector` defined previously. To do this, `cd` into the `$DIR/problems` directory and run `make ByteSelector`.

When we run the testbench, we will notice that the simulation produces debug output every time the `step` function is called. Each of these calls gives the state of the inputs and outputs to the `ByteSelector` and whether the check between the reference output and expected output matched as shown below:

```
STARTING ../emulator/problems/ByteSelector
---
POKE ByteSelector__io_in <- 12345678
POKE ByteSelector__io_offset <- 0
STEP 1 <- 0
PEEK ByteSelector__io_out -> 0x4e
EXPECT ByteSelector__io_out <- 78 == 78 PASS
POKE ByteSelector__io_in <- 12345678
POKE ByteSelector__io_offset <- 1
STEP 1 <- 0
PEEK ByteSelector__io_out -> 0x61
EXPECT ByteSelector__io_out <- 97 == 97 PASS ...
POKE ByteSelector__io_in <- 12345678
POKE ByteSelector__io_offset <- 3
STEP 1 <- 0
PEEK ByteSelector__io_out -> 0x00
EXPECT ByteSelector__io_out <- 0 == 0 PASS
PASSED // Final pass assertion
```

Also notice that there is a final pass assertion "PASSED" at the end which corresponds to the `allGood` at the very end of the testbench. In this case, we know that the test passed since the `allGood` assertion resulted in a "PASSED" output message here.

### 5.1.3 General Testbench

In general, the scala testbench should have the following rough structure:

- Set inputs using `poke`
- Advance simulation using `step`
- Check expected values using `expect` (and/or `peek`)
- Repeat until all appropriate test cases verified

For sequential modules we may want to delay the output definition to the appropriate time as the `step` function implicitly advances the clock one period in the simulation. Unlike Verilog, you do not need to explicitly specify the timing advances of the simulation; Chisel will take care of these details for you.

### 5.2 Max2 Testbench Problem

In this assignment, write a tester for the Max2 circuit:

```scala
class Max2 extends Module {
  val io = new Bundle {
    val in0 = UInt(INPUT, 8)
    val in1 = UInt(INPUT, 8)
    val out = UInt(OUTPUT, 8)
  }
  io.out := Mux(io.in0 > io.in1, io.in0, io.in1)
}
```

found in `$TUT_DIR/problems/Max2.scala` by filling in the following tester:

```scala
class Max2Tests(c: Max2) extends Tester(c) {
  for (i <- 0 until 10) {
    poke(c.io.in0, 0)
    poke(c.io.in1, 0)
    step(1)
    expect(c.io.out, 1)
  }
}
```

using random integers generated as follows:

```scala
// returns random int in 0..lim-1
val in0 = rnd.nextInt(lim)
```

Run `make Max2.out` until the circuit passes your tests.
5.3 Limitations of the Testbench

The Chisel testbench works well for simple tests and small numbers of simulation iterations. However, for larger test cases, the Chisel testbench quickly becomes more complicated and slower simply due to the inefficiency of the infrastructure. For these larger and more complex test cases, we recommend using the C++ emulator or Verilog test harnesses which run faster and can handle more rigorous test cases.
Chapter 6

Creating Your Own Project

6.1 Creating Your Own Projects

In order to create your own projects from scratch, you will need to create a directory, a Chisel source file, and a build.sbt configuration file. In the first part of this tutorial we cover the basic calls to SBT in order generate appropriate files. At the end of the tutorial, we will explain how the Makefile infrastructure can make the process more streamlined.

6.1.1 Directory Structure

The simplest project file organization is using a single directory containing your Scala project file and your Chisel source file. The project directory structure would look like:

```
Hello/
| build.sbt           # scala configuration file
| Hello.scala         # your source file
```

We will refer to the path to the Hello directory as `$BASEDIR` from here on. More sophisticated directory structures can be useful in the future. Consult the SBT documentation for more information.

6.1.2 The Source Directory and Chisel Main

The top directory `$BASEDIR/` contains Scala source files containing all of the Chisel module definitions for your circuit and a main method. In this simple example, we have one Scala source file as shown below:

```
package Hello
import Chisel._

class HelloModule extends Module {
  val io = new Bundle {
    val out = UInt(OUTPUT, 8)
  }
  io.out := UInt(42)
}

class HelloModuleTests(c: HelloModule)
  extends Tester(c) {
    step(1)
    expect(c.io.out, 42)
  }

object hello {
  def main(args: Array[String]): Unit = {
    val margs = Array("--backend", "c", "--genHarness",
                      "--compile", "--test")
    chiselMainTest(margs, () => Module(new HelloModule())) {
      c => new HelloModuleTests(c)
    }
  }
}
```

In the above example, we have a module definition in package `Hello` for a `Hello` module. The main method calls `chiselMainTest` for a new `Hello` module. In addition to creating the module, the call to `chiselMainTest` also includes a call to execute the scala testbench defined in the routine `HelloModuleTests`.

6.1.3 The build.sbt Template

The `build.sbt` configuration file is located in the top folder and contains a number of settings used by `sbt` when building and compiling the Chisel sources. The following shows the recommended `build.sbt` template that should be used:

```
scalaVersion := "2.10.2"
resolvers ++= Seq(
  "scct-github-repository" at
  "http://mtkopone.github.com/scct/maven-repo"
)
```

Note that when you have multiple Scala files, in order for main to recognize your module definition, your module definition must be in the same package as the main function.
libraryDependencies +=
  "edu.berkeley.cs" % "chisel" % "latest.release"

The SBT project file contains a reference to Scala version greater or equal to 2.10.2 and a dependency on the latest release of the Chisel library.

6.2 Compiling the Chisel Source

6.2.1 Compiling the Emulation Files

In order to launch SBT to compile the Chisel code we must first be in the directory $BASEDIR/. The following call is then made to compile and run the Hello module:

sbt run

6.2.2 Running the Chisel Tests

To actually run the tests referenced in the main method of $BASEDIR/Hello.scala, we need to tell SBT to also generate the harness and run the tests. For instance, for our Hello module introduced earlier, the Chisel main method references a test routine HelloTests. In order to both compile the Hello component and run the tests defined in Hello, we make the following call to sbt:

sbt "run --backend c --compile --test --genHarness"

Note the addition of the 5 arguments at the end of the call to run. This will both compile the .cpp and .h files for the emulator and run the Chisel tests defined.

6.2.3 Compiling Verilog

Similarly to compile the Chisel code and generate the Verilog HDL, a similar call to SBT is made with slightly different arguments. The call looks like:

sbt "run --backend v --genHarness"

Note the call is very similar to when generating C++; the key difference is the parameter to the --backend attribute which is now v which specifies to sbt that we would like to compile our Chisel component to Verilog.

6.3 Putting It All Together

In summary, the bare minimum project components that are necessary for your project to get off the ground are the following files:

1. $BASEDIR/build.sbt
2. $BASEDIR/<Chisel source files>.scala

Together, these files compose a Chisel project and can be used to generate the Verilog and C++ files. It is strongly recommended that you supplement the file structure with appropriate Makefiles but is not strictly necessary (examples can be found in the Chisel tutorial project).
Chapter 7

Conditional Assignments and Memories

7.1 Conditional Register Updates

As shown earlier in the tutorial, conditional register updates are performed with the `when` block which takes a `Bool` value or some boolean expression to evaluate. In this section we more fully explore how to use this `when` conditional update structure.

If a `when` block is used by itself, Chisel will assume that if the condition for the `when` block doesn’t evaluate to true, there is no update to the register value. However, most of the time we don’t want to limit ourselves to a single conditional. Thus in Chisel we use `.elsewhen` and `.otherwise` statements to select between multiple possible register updates as shown in the following sections.

7.1.1 The `.elsewhen` Clause

When specifying a conditional update, we may want to check several conditions which we want to check in some order. To do this for register updates, we use a `when` `.elsewhen` structure. This is analogous to an `if...else if` control structure in sequential programming. As with `else if` clauses, as many `.elsewhen` statements can be chained together in a single `when` block.

The general structure thus looks like:

```
when (<condition 1>) {<register update 1>}
.elsewhen (<condition 2>) {<register update 2>}
...
.elsewhen (<condition N>) {<register update N>}
```

Where `<condition 1>` through `<condition N>` represent the trigger conditions of their respective `<register update>` segments.

An example of this statement in action is shown in the following implementation of a simple stack pointer. Suppose, we need to maintain a pointer that keeps track of the address of the top of a stack. Given a signal `pop` that decrements the stack pointer address by 1 entry and a signal `push` that increments the stack pointer address by 1 entry, the implementation of just the pointer would look like the following:

```scala
class StackPointer(depth:Int) extends Module {
  val io = new Bundle {
    val push = Bool(INPUT)
    val en = Bool(INPUT)
    val pop = Bool(INPUT)
  }
  val sp = Reg(init = UInt(0, width = log2Up(depth)))
  when (io.en && io.push && (sp != UInt(depth-1))) {
    sp := sp + UInt(1)
  } .elsewhen(io.en && io.pop && (sp > UInt(0))) {
    sp := sp - UInt(1)
  }
}
```

Notice that in this implementation, the push signal has higher priority over the `pop` signal as it appears earlier in the `when` block.

7.1.2 The `.otherwise` Clause

In order to specify a default register update value if all the conditions in the `when` block fail to trigger, we use an `.otherwise` clause. The `.otherwise` clause is analogous to the `else` case that completes an `if ... else` block. The `.otherwise` statement must occur last in the `when` block.

The general structure for the complete `when` block now looks like:

```
when (<condition 1>) {<register update 1>}
.elsewhen (<condition 2>) {<register update 2>}
...
.elsewhen (<condition N>) {<register update N>}
.otherwise {<default register update>}
```

In the previous example, we could add a default statement which just assigns `sp` to the current value
CHAPTER 7. CONDITIONAL ASSIGNMENTS AND MEMORIES

of sp. The block would then look like:

```java
when(io.en && io.push && (sp != UInt(depth-1))) {
    sp := sp + UInt(1)
} .elsewhen(io.en && io.pop && (sp > UInt(0))) {
    sp := sp - UInt(1)
}.otherwise {
    sp := sp
}
```

The explicit assignment to preserve the value of sp is redundant in this case but it captures the point of the .otherwise statement.

7.1.3 The unless Clause

To complement the when statement, Chisel also supports an unless statement. The unless statement is a conditional assignment that triggers only if the condition is false. The general structure for the unless statement is:

```java
unless ( <condition> ) { <assignments> }
```

For example, suppose we want to do a simple search of the contents of memory and determine the address that contains some number. Since we don’t know how long the search will take, the module will output a done signal when it is finished and until then, we want to continue to search memory. The Chisel code for the module would look like:

```java
class MemorySearch extends Module {
    val io = new Bundle {
        val target = UInt(INPUT, 4)
        val address = UInt(OUTPUT, 3)
        val en = Bool(INPUT)
        val done = Bool(INPUT)
    }
    val index = Reg(init = UInt(0, width = 3))
    val list = Vec(UInt(0), UInt(4), UInt(15), UInt(14), UInt(2), UInt(5), UInt(13))
    val memVal = list(index)
    val done = (memVal === io.target) || (index === UInt(7))
    unless (done) {
        index := index + UInt(1)
    }
    io.done := done
    io.address := index
}
```

In this example, we limit the size of the memory to 8 entries and use a vector of literals to create a read only memory. Notice that the unless statement is used to terminate the iteration if it see that the done signal is asserted. Otherwise, it will continue to increment the index in memory until it finds the value in target or reaches the last index in the memory (7).

7.2 Combinational Conditional Assignment

You can also use the when .elsewhen .otherwise block to define combinational values that may take many values. For example, the following Chisel code show how to implement a basic arithmetic unit with 4 operations: add, subtract, and pass. In this example, we check the opcode to determine which operation to perform and conditionally assign the output.

```java
class BasicALU extends Module {
    val io = new Bundle {
        val a = UInt(INPUT, 4)
        val b = UInt(INPUT, 4)
        val opcode = UInt(INPUT, 2)
        val output = UInt(OUTPUT, 4)
    }
    io.output := UInt(0)
    when (io.opcode === UInt(0)) {
        io.output := io.a + io.b // ADD
    } .elsewhen (io.opcode === UInt(1)) {
        io.output := io.b - io.b // SUB
    } .elsewhen (io.opcode === UInt(2)) {
        io.output := io.a // PASS A
    } .otherwise {
        io.output := io.b // PASS B
    }
}
```

Notice that this can easily be easily expanded to check many different conditions for more complicated arithmetic units or combinational blocks.

7.3 Read Only Memories

To instantiate read only memories in Chisel, we use a vector of constant literals and specify a literal type. For example, in order to instantiate an 4 entry read only memory with the values 0 to 3, the definition would look like the following:

```java
val numbers = Vec(UInt(0), UInt(1), UInt(2), UInt(3)) { UInt(width = 2) }
```

Notice that we need to specify the type of literal in the ... braces following the literals. Accessing the values in the read only memory is the same as accessing an entry in a Vec. For example, to access the 2nd entry of the memory we would use:

```java
val entry2 = numbers(2)
```
7.4 Read-Write Memories

Chisel contains a primitive for memories called Mem. Using the Mem class it is possible to construct multiplexed memories that can be synchronous or combinational.

7.4.1 Basic Instantiation

The Mem construction takes a memory depth and a data type which it is composed of. The general declaration structure looks like:

```chisel
val myMem = Mem(<type>, <depth>)
```

Where <depth> corresponds to the number of entries of <type> are in the memory.

For instance, if you wanted to create a 128 deep memory of 32 bit UInt types, you would use the following instantiation:

```chisel
val myMem = Mem(UInt(width = 32), depth = 128)
```

Note that when constructing a memory in Chisel, the initial value of memory contents cannot be specified. Therefore, you should never assume anything about the initial contents of your Mem class.

7.4.2 Synchronous vs. Combinational Read

It is possible to specify either combinational or synchronous read behavior during instantiation by setting the seqRead parameter when defining the Mem. The seqRead parameter is a Bool that tells Chisel if you want synchronous read behavior memory or not.

For instance, if we wanted a combinational read 128 entry memory of 32 bit UInt types, we would use the following definition:

```chisel
val syncMem = Mem(UInt(width = 32), depth = 128, seqRead = true)
```

By default, Chisel will assume that the read behavior is combinational.

7.4.3 Adding Write Ports

To add write ports to the Mem, we use a when block to allow Chisel to infer a write port. Inside the when block, we specify the location and data for the write transaction. In general, adding a write port requires the following definition:

```chisel
when (<write condition>) {
  <memory name>(<write address>) := <write data>
}
```

Where <write address> refers to the entry number in the memory to write to. Also notice that we use the reassignment operator := when writing to the memory.

For example, suppose we have a 128 deep memory of 32 bit UInt types. If we wanted to write a 32 bit value `dataIn` to the memory at location `writeAddr` if as write enable signal `we` is true, our Chisel code would look like:

```chisel
... val myMem = Mem(UInt(width = 32), depth = 128) when (we) {
  myMem(writeAddr) := dataIn
}
...```

7.4.4 Adding Read Ports

Depending on the type of read behaviour specified, the syntax for adding read ports to Mem in Chisel is slightly different for combinational read and synchronous read memories.

**Combinational Read Ports**

For combinational read memories, adding read ports to the memory simply amounts to placing an assignment inside a when block with some trigger condition. If you want Chisel to infer multiple read ports, simply add more assignments in the when definition. The general definition for read ports is thus:

```chisel
when (<read condition>) {
  <read data 1> := <memory name>(<read address 1>)
  ...
  <read data N> := <memory name>(<read address N>)
}
```

For instance, if you wanted a 128 entry memory of 32 bit UInt values with two combinational read ports, with some read enable re and reads from addresses `raddr1` and `raddr2`, we would use the following when block definition:
... val myMem =  
  Mem(UInt(width = 32), 128, seqRead = false)  
val read_port1 = UInt(width = 32)  
val read_port2 = UInt(width = 32)  
when (re) {  
read_port1 := myMem(raddr1)  
read_port2 := myMem(raddr2)  }
...

Note that the type and width of the read_port1 and read_port2 should match the type and width of the entries in the Mem.

### Synchronous Read Ports

In order to add synchronous read ports to the Chisel Mem class, Chisel requires that the output from the memory be assigned to a Reg type. Like the combinational read port, a synchronous read assignment must occur in a when block. The general structure for the definition of a synchronous read port is as follows:

... val myMem =  
  Mem(UInt(width = 32), depth = 128, seqRead = true)  
val read_port = Reg(UInt(width = 32))  
when (re) {  
read_port := myMem(raddr)  }
...

Since the module is parametrized to be depth entries deep, in order to correctly extract the minimum width of the stack pointer sp we take the log2Up(depth). This takes the base 2 logarithm of depth and rounds up.

### 7.4.5 Example of Mem in Action

Here we provide a small example of using a memory by implementing a stack.

Suppose we would like to implement a stack that takes two signals push and pop where push tells the stack to push an input dataIn to the top of the stack, and pop tells the stack to pop off the top value from the stack. Furthermore, an enable signal en disables pushing or popping if not asserted. Finally, the stack should always output the top value of the stack.

```scala
class Stack(depth: Int) extends Module {  
val io = new Bundle {  
  val dataIn = UInt(INPUT, 32)  
  val dataOut = UInt(OUTPUT, 32)  
  val push = Bool(INPUT)  
  val pop = Bool(INPUT)  
  val en = Bool(INPUT)  
}  
// declare the memory for the stack  
val stack_mem =  
  Mem(UInt(width = 32), depth, seqRead = false)  
val sp = Reg(init = UInt(0, width = log2Up(depth)))  
val dataOut = Reg(init = UInt(0, width = 32))  
// Push condition - make sure stack isn't full  
when(io.en && io.push && (sp != UInt(depth-1))) {  
  stack_mem(sp + UInt(1)) := io.dataIn  
  sp := sp + UInt(1)  }
// Pop condition - make sure the stack isn't empty  .elsewhen(io.en && io.pop && (sp > UInt(0))) {  
  sp := sp - UInt(1)  }
when(io.en) {  
  dataOut := stack_mem(sp)  }
io.dataOut := dataOut  
}
```

### 7.4.6 Load/Search Mem Problem

In this assignment, write a memory module that supports loading elements and searching based on the following template:

```scala
class DynamicMemorySearch(val n: Int, val w: Int) extends Module {  
val io = new Bundle {  
  val isWr = Bool(INPUT)  
  val wrAddr = UInt(INPUT, log2Up(n))  
  val data = UInt(INPUT, w)  
  val en = Bool(INPUT)  
  val target = UInt(OUTPUT, log2Up(n))  
  val done = Bool(OUTPUT)  
}  
val index = Reg(init = UInt(0, width = log2Up(n)))  
val memVal = UInt(0)  
/// fill in here  
io.done := Bool(false)  
io.target := index  
}
```

and found in $TUT_DIR/problems/DynamicMemorySearch.scala. Notice how it supports depth and width parameters n and w and how the address width is computed from the depth. Run

```
make DynamicMemorySearch.out
```
until your circuit passes the tests.
Chapter 8

Scripting Hardware Generation

8.1 Using the For loop

Often times parametrization requires instantiating multiple components which are connected in a very regular structure. A revisit to the parametrized Adder component definition shows the for loop construct in action:

```scala
// A n-bit adder with carry in and carry out
class Adder(n: Int) extends Module {
    val io = new Bundle {
        val A = UInt(INPUT, n)
        val B = UInt(INPUT, n)
        val Cin = UInt(INPUT, 1)
        val Sum = UInt(OUTPUT, n)
        val Cout = UInt(OUTPUT, 1)
    }

    // create a vector of FullAdders
    val FAs = Vec.fill(n){ Module(new FullAdder()).io }
    val carry = Vec.fill(n+1){ UInt(width = 1) }
    val sum = Vec.fill(n){ Bool() }

    // first carry is the top level carry in
    carry(0) := io.Cin

    // wire up the ports of the full adders
    for(i <- 0 until n) {
        FAs(i).a := io.A(i)
        FAs(i).b := io.B(i)
        FAs(i).cin := carry(i)
        carry(i+1) := FAs(i).cout
        sum(i) := FAs(i).sum.toBool()
    }

    io.Sum := sum.toBits().toUInt()
    io.Cout := carry(n)
}
```

Notice that a Scala integer i value is used in the for loop definition as the index variable. This indexing variable is specified to take values from 0 until n, which means it takes values 0, 1, 2..., n-1. If we wanted it to take values from 0 to n inclusive, we would use for (i <- 0 to n).

It is also important to note, that the indexing variable i does not actually manifest itself in the generated hardware. It exclusively belongs to Scala and is only used in declaring how the connections are specified in the Chisel component definition.

The for loop construct is also very useful for assigning to arbitrarily long Vecs.

8.2 Using If, Else If, Else

As previously mentioned, the if, elseif, and else keywords are reserved for Scala control structures. What this means for Chisel is that these constructs allow you to selectively generate different structures depending on parameters that are supplied. This is particularly useful when you want to turn certain features of your implementation "on" or "off", or if you want to use a different variant of some component.

For instance, suppose we have several simple counters that we would like to package up into a general purpose counter module: UpCounter, DownCounter, and OneHotCounter. From the definitions below, we notice that for these simple counters, the I/O interfaces and parameters are identical:

```scala
// Simple up counter that increments from 0 and wraps around
class UpCounter(CounterWidth:Int) extends Module {
    val io = new Bundle {
        val output = UInt(OUTPUT, CounterWidth)
        val ce = Bool(INPUT)
    }

    // Simple down counter that decrements from 2^CounterWidth-1 then wraps around
    class DownCounter(CounterWidth:Int) extends Module {
        val io = new Bundle {
            val output = UInt(OUTPUT, CounterWidth)
            val ce = Bool(INPUT)
        }

        // Simple one hot counter that increments from one hot 0 to CounterWidth-1 then wraps around
        class OneHotCounter(CounterWidth:Int) extends Module {
            val io = new Bundle {
                val output = UInt(OUTPUT, CounterWidth)
            }
```

Notice that a Scala integer i value is used in the for loop definition as the index variable. This indexing variable is specified to take values from 0 until n, which means it takes values 0, 1, 2..., n-1. If we wanted it to take values from 0 to n inclusive, we would use for (i <- 0 to n).

It is also important to note, that the indexing variable i does not actually manifest itself in the generated hardware. It exclusively belongs to Scala and is only used in declaring how the connections are specified in the Chisel component definition.

The for loop construct is also very useful for assigning to arbitrarily long Vecs.

8.2 Using If, Else If, Else

As previously mentioned, the if, elseif, and else keywords are reserved for Scala control structures. What this means for Chisel is that these constructs allow you to selectively generate different structures depending on parameters that are supplied. This is particularly useful when you want to turn certain features of your implementation "on" or "off", or if you want to use a different variant of some component.

For instance, suppose we have several simple counters that we would like to package up into a general purpose counter module: UpCounter, DownCounter, and OneHotCounter. From the definitions below, we notice that for these simple counters, the I/O interfaces and parameters are identical:

```scala
// Simple up counter that increments from 0 and wraps around
class UpCounter(CounterWidth:Int) extends Module {
    val io = new Bundle {
        val output = UInt(OUTPUT, CounterWidth)
        val ce = Bool(INPUT)
    }

    // Simple down counter that decrements from 2^CounterWidth-1 then wraps around
    class DownCounter(CounterWidth:Int) extends Module {
        val io = new Bundle {
            val output = UInt(OUTPUT, CounterWidth)
            val ce = Bool(INPUT)
        }

        // Simple one hot counter that increments from one hot 0 to CounterWidth-1 then wraps around
        class OneHotCounter(CounterWidth:Int) extends Module {
            val io = new Bundle {
                val output = UInt(OUTPUT, CounterWidth)
            }
```

Notice that a Scala integer i value is used in the for loop definition as the index variable. This indexing variable is specified to take values from 0 until n, which means it takes values 0, 1, 2..., n-1. If we wanted it to take values from 0 to n inclusive, we would use for (i <- 0 to n).

It is also important to note, that the indexing variable i does not actually manifest itself in the generated hardware. It exclusively belongs to Scala and is only used in declaring how the connections are specified in the Chisel component definition.

The for loop construct is also very useful for assigning to arbitrarily long Vecs.

8.2 Using If, Else If, Else

As previously mentioned, the if, elseif, and else keywords are reserved for Scala control structures. What this means for Chisel is that these constructs allow you to selectively generate different structures depending on parameters that are supplied. This is particularly useful when you want to turn certain features of your implementation "on" or "off", or if you want to use a different variant of some component.

For instance, suppose we have several simple counters that we would like to package up into a general purpose counter module: UpCounter, DownCounter, and OneHotCounter. From the definitions below, we notice that for these simple counters, the I/O interfaces and parameters are identical:

```scala
// Simple up counter that increments from 0 and wraps around
class UpCounter(CounterWidth:Int) extends Module {
    val io = new Bundle {
        val output = UInt(OUTPUT, CounterWidth)
        val ce = Bool(INPUT)
    }

    // Simple down counter that decrements from 2^CounterWidth-1 then wraps around
    class DownCounter(CounterWidth:Int) extends Module {
        val io = new Bundle {
            val output = UInt(OUTPUT, CounterWidth)
            val ce = Bool(INPUT)
        }

        // Simple one hot counter that increments from one hot 0 to CounterWidth-1 then wraps around
        class OneHotCounter(CounterWidth:Int) extends Module {
            val io = new Bundle {
                val output = UInt(OUTPUT, CounterWidth)
            }
```
CHAPTER 8. SCRIPTING HARDWARE GENERATION

We could just instantiate all three of these counters and multiplex between them but if we needed one at any given time this would be a waste of hardware. In order to choose between which of these three counters we want to instantiate, we can use Scala’s if, else if, else statements to tell Chisel how to pick which component to instantiate based on a CounterType parameter:

```scala
class Counter(CounterWidth: Int, CounterType: String) extends Module {
  val io = new Bundle {
    val output = UInt(OUTPUT, CounterWidth)
    val ce = Bool(INPUT)
  }
  if (CounterType == "UpCounter") {
    val upcounter = new UpCounter(CounterWidth)
    upcounter.io.ce := io.ce
    io.output := upcounter.io.output
  } else if (CounterType == "DownCounter") {
    val downcounter = new DownCounter(CounterWidth)
    downcounter.io.ce := io.ce
    io.output := downcounter.io.output
  } else if (CounterType == "OneHotCounter") {
    val onehotcounter = new OneHotCounter(CounterWidth)
    onehotcounter.io.ce := io.ce
    io.output := onehotcounter.io.output
  } else {
    // default output 1
    io.output := UInt(1)
  }
}
```

By consolidating these three counter components into a single Counter module, we can instantiate a different counter by simply changing the parameter CounterType. For instance:

```scala
// instantiate a down counter of width 16
val downcounter = Module(new Counter(16, "DownCounter"))
```

This allows seamless alternation between them.

8.3 Using def

Chisel also allows the usage of the Scala def statement to define Chisel code that may be used frequently. These def statements can be packaged into a Scala Object and then called inside a Module. The following Chisel code shows an alternate implementation of a counter using def that increments by amt if the inc signal is asserted.

```scala
object Counter {
  def wrapAround(n: UInt, max: UInt) =
    Mux(n > max, UInt(0), n)
  def counter(max: UInt, en: Bool, amt: UInt) = {
    val x = Reg(init = UInt(8, max.getWidth))
    x := wrapAround(x + amt, max)
  }
}
```

In this example, we use calls to subroutines defined in the Counter object in order to perform the appropriate logic.

8.4 Parameterized Vec Shift Reg Problem

The next assignment is to construct a bit shift register with delay parameter. The following is a the template from $TUT_DIR/problems/VecShiftRegisterParam.scala:

```scala
class VecShiftRegisterParam(val n: Int, val w: Int)
  extends Module {
  val io = new Bundle {
    val in = UInt(INPUT, w)
    val out = UInt(OUTPUT, w)
  }
  ...
  io.out := UInt(0)
}
```

where `out` is a n cycle delayed copy of values on `in`. Also notice how `val` is added to each parameter value to allow those values to be accessible from the tester. Run

```bash
make VecShiftRegisterParam.out
```

until your circuit passes the tests.
8.5 Mul Lookup Table Problem

The next assignment is to write a 16x16 multiplication table using Vec. The following is a the template from TUT_DIR/problems/Mul.scala:

```scala
class Mul extends Module {
  val io = new Bundle {
    val x = UInt(INPUT, 4)
    val y = UInt(INPUT, 4)
    val z = UInt(OUTPUT, 8)
  }
  val muls = new ArrayBuffer[UInt]()
  // flush this out ...
  io.z := UInt(0)
}
```

As a hint build the lookup table using a rom constructed from the tab lookup table represented as a Scala ArrayBuffer with incrementally added elements (using +=):

```scala
val tab = Vec(muls)
```

and lookup the result using an address formed from the x and y inputs as follows:

```scala
io.z := tab(Cat(io.x, io.y))
```

Run

```
make Mul.out
```

until your circuit passes the tests.