RISC-V Processor Developed in the Chisel Programming Language

Ryan Ridley

August 2019

Chapter 1

First Chapter

1.1 Overview

RISC-V is a reduced instruction set architecture that is becoming more and more popular in today's processors. Chisel, which is a hardware description language embedded in the Scala programming language, is also becoming more and more popular in toady's time. This document is an overview of how a RISC-V processor was created in Chisel. Chisel basics, Scala basics, debugging, testing, and more will all be covered using the RISC-V processor and (mostly when covering the basics) smaller, easier code snippets as examples.

1.2 Chisel Introduction

Chisel is a hardware description language embedded in the Scala programming language. Chisel allows the user to describe hardware in a highly parameterized, object oriented way. This is what makes Chisel more attractive than Verilog or VHDL. Chisel still allows the user to output a Verilog file of the Chisel code. Using Scala as a basis, object oriented functionality can be used to program logic to aid in the design of hardware.

1.3 Chisel Template

To begin making a Chisel project, the project template must first be downloaded. This project template deal with a lot of the boiler plate code that Chisel needs to create the project. When creating a custom project, the source files must be in the directory src/main/scala. Any test files, files that run tests on the source code, need to be in src/test/scala. These test files will be explained later.

1.4 Scala Basics

1.4.1 Variables

Scala uses many of the same programming paradigms as many other popular programming languages. The first is variables (var) and constant values (val).

```
var numberOfKittens = 6
val kittensPerHouse = 101
val alphabet = "abcdefghijklmnopqrxtuvwxyz"
var done = false
```

The value of the variables **numberOfKittens** and **done** can be changed

```
numberOfKittens += 1
done = false
```

while the others defined with val cannot.

1.4.2 If Statement

One of the most important functions in a programming language is the if statement. Scala supports this in the same fashion that most languages do.

```
if(numberOfKittens > kittensPerHouse) {
   println("Too many kittens!")
}

//braces are not required as long
//as branches are one liners

if (done)
   println("Done!")
else
   numberOfKittens += 1
```

The **if** conditional can also return a value. It is given by the last line of the selected branch.

```
val likelyCharacterSet = if (alphabet.length == 26)
   "english"
else
   "not english"
```

1.4.3 Methods

Methods are defined with the keyword **def**. Parameters are defined in a comma separated list that define the name and type as follows.

```
def methodName(param1: Int, param2:
   String, param3: Boolean) {
   //code
}

//Methods can also be one liners.

def times2(x: Int): Int = 2 * x
```

1.4.4 Lists

Lists are a lot like arrays but support more functionality for appending and extracting.

```
val x = 7
val y = 14
val list1 = List(1, 2, 3)

//Another way to make a list
val list2 = x :: y :: y :: Nil

val list3 = list1 ++ list2
val m = list2.length
val s = list2.size

val headOfList = list1.head //Gets the head of list
val tailOfList = list1.tail //Gets the tail of list

//Gets the third element of the list
val third = list1(2)
```

1.4.5 For Loops

Scala has for loops that work just like in traditional languages.

```
for (i <- 0 to 7) {print(i + " ")}
println()
```

Using **until** instead of **to** to go up to a value, but stop before it.

```
for (i <- 0 until 7) {print(i + " ")}
println()
```

Adding **by** at the end will increment i by a fixed amount.

```
for (i <- 0 to 10 by 2) {print(i + " ")}
println()
```

1.4.6 Classes

Here is an example of a class.

```
// WrapCounter counts up to a
//max value based on a bit size

class WrapCounter(counterBits: Int) {

  val max: Long = (1 << counterBits) - 1
  var counter = 0L

  def inc(): Long = {
    counter = counter + 1
    if (counter > max) {
        counter = 0
    }
    counter
  }
 println(s"counter created with max value \$max")
}
```

The structure of Scala is very similar to languages like Java and C++. Now that a basis of Scala has been set, making hardware will now be much easier and more reusable.

Chapter 2

Chisel Basics

Chisel is a hardware description language that uses Scala as a basis for object oriented programming. All Chisel code requires certain packages to be imported and a specific format that must be followed. Make sure to add these imports at the beginning of every file that includes Chisel code.

```
import chisel3._
import chisel3.util._
import chisel3.iotester.{ChiselFlatSpec,
  Driver, PeekPokeTester}
```

The last import is specifically for testing Chisel code. Here is a very simple Chisel module.

```
// Chisel Code: Declare a new module definition
class Passthrough extends Module {
  val io = IO(new Bundle {
    val in = Input(UInt(4.W))
    val out = Output(UInt(4.W))
  })
  io.out := io.in
}
```

Every Chisel module (class) must extend **Module**. Next are the input and outputs of the module. **io** is needed for every module. All of the vals inside **IO(new Bundle)** are the inputs and outputs of the user. When referencing a variable, **io.variableName** is required. Looking at the **in** variable we can see that it is declared as an input, unsigned integer (UInt), and is a width of four (4.W). The variable **out** is similar except it is declared as an output. This is a simple passthrough module. The output is immediately set to the input. The operator used to connect the values is **:=**. This is basically like taking a wire and connecting the output to the input. This is a directional operator, meaning that the left hand signal drives the right hand signal.

2.1 Types

Unfortunately, Scala types (integers, doubles, etc.) are not compatible with Chisel types. Below is an example.

```
val two = 1 + 1 //Scala integer

val utwo = 1.U + 1.U //Chisel unsigned integer
```

A **.U** must be added to the end of a number to cast into the Chisel unsigned integer type. For booleans, a **.B** is added to the end:

```
val true_value = true.B

val false_value = false.B
```

Casting is another function that Chisel supports. Here are some common castings:

- `asUInt()`
- `asSInt()`
- `asBool()`

2.1.1 Multiplexing and Concatenation

Variables can be set with the **Mux()** or **Cat()** function.

```
class MyOperatorsTwo extends Module {
  val io = IO(new Bundle {
    val in      = Input(UInt(4.W))
    val out_mux = Output(UInt(4.W))
    val out_cat = Output(UInt(4.W))
  })

  val s = true.B
  io.out_mux := Mux(s, 3.U, 0.U)
  io.out_cat := Cat(2.U, 1.U)
}
```

2.1.2 Conditionals

When Statement

When dealing with hardware in Chisel, a different type of conditional statement must be used. Below is an example of a Chisel conditional.

```
class Mux3 extends Module {
  val io = IO(new Bundle {
    val in1 = Input(UInt(16.W))
    val in2 = Input(UInt(16.W))
    val in3 = Input(UInt(16.W))
    val out = Output(UInt(16.W))
  })

  when(io.in1 > io.in2 && io.in1 > io.in3) {
    io.out := io.in1
  }.elsewhen(io.in2 > io.in1 && io.in2 > io.in3) {
    io.out := io.in2
  }.otherwise {
    io.out := io.in3
  }
}
```

Chisel uses **when, elsewhen, and otherwise** for its conditional statements. Works exactly like a regular if statement.

Switch Statement

The switch statement works much the same as a regular switch statement.

```
  switch(x) {
    is(value1) {
      // run if x === value1
    } is(value2) {
      // run if x === value2
    }
  }
Great for control flow.
```

2.1.3 Testing Chisel

There are a few debugging tools that Chisel supports. PeekPokeTester is one class that can be extended that allows the user to "poke" (set) signals and the "peek" (view) the resulting signals. Review the following example:

```
class ALU extends Module {
  val io = IO(new Bundle {
    val a = Input(UInt(32.W))
    val b = Input(UInt(32.W))
    val op = Input(UInt(5.W))
    val out = Output(UInt(32.W))
  })

  when (io.op === 0.U) {
        io.out := io.a + io.b //ADD
    }.elsewhen (io.op === 1.U) {
        io.out := io.a - io.b //SUB
    }.elsewhen (io.op === 2.U) {
        io.out := io.a & io.b //AND
    }.elsewhen (io.op === 3.U) {
        io.out := io.a | io.b //OR
    }.elsewhen (io.op === 4.U) {
        io.out := io.a ^ io.b //XOR
    }.otherwise {
        io.out := io.a
    }
  }
```

Here is the main class that holds all of our logic. Next, a testing class needs to be made. It looks as follows:

```
 class ALUTest(alu: ALU) extends PeekPokeTester(alu) {
    poke(alu.io.a, 17.U)
    poke(alu.io.b, 1.U)
    poke(alu.io.op, 0.U)
    step(1)

    println("The result: " + peek(alu.io.out))
}

object ALU extends App {
  iotesters.Driver.execute(args, () => new ALU) {
    alu => new ALUTest(alu)
  }}
}
```

Chisel finds the object of the main class (ALU) and using **iotesters.Driver.execute** the ALUTest class will be run. Using **poke** values can be inserted into the inputs. The inputs and outputs can also be "peeked" at using **peek**. Using the println function, io.out is printed to the console using **peek(alu.io.out)**. This is useful for debugging code, but the only values that can be peeked at are io inputs and outputs. Any internal signals trying to be peeked at will cause an error to be thrown. If internal signals need to be looked at then a wire must first be made.

```
class debuggingTest extends Module {
 val io = IO(new Bundle{
    val a = Input(UInt(32.W))
    val b = Input(UInt(32.W))
    val out = Output(UInt(32.W))
 })

 val and = Wire(UInt(32.W))
 and := io.a & io.b
 printf(p"a & b = \$and\n")

 io.out := io.a + io.b
}
```

Using this method allows internal signals that are not inputs or outputs to be printed to the terminal. This is used heavily in the RISC-V processor for debugging. If multiple classes are being run and all have print statements in them, then the order in which these classes are called will determine when the print statement gets called. This can cause some confusing debugging because this is different than most hardware description languages when everything is technically ran at once. Chisel is an object oriented programming language at the core so it follows that standard paradigms of most object oriented programming languages.

Chapter 3

RISC-V Processor

Now with a basic understanding of both Scala and Chisel, construction a simple processor is easier than it may seem. The design of this particular processor is split up into 9 classes (or modules) with two supplementary classes needed for simulation.

- top
- riscv
- extend
- decoder
- datapath
- regfile
- alu
- imem
- dmem
- supplementary

An explanation of each class and how it fits into the whole architecture is what will follow next.

3.1 Top Module

Top is exactly what it sounds like. This is the top of the program where everything is brought together. This mainly brings the instruction memory (imem) and main memory (dmem) into the main processing unit (riscv).

```
val io = IO(new Bundle {
val valid = Output(UInt(1.W))
})

val r = Module(new riscv)
val im = Module(new imem)
val dm = Module(new dmem)
val instr_out = Wire(UInt(32.W))
val pc_pulled = Wire(UInt(32.W))
val mem_inW = Wire(UInt(32.W))
val mem_outW = Wire(UInt(32.W))
instr_out := im.io.mem_out
pc_pulled := r.io.pc / 4.U
dm.io.mem_addr := r.io.aluResult
dm.io.mem_in := r.io.writeData
dm.io.enable := r.io.memWrite
r.io.readData := dm.io.mem_out
mem_outW := dm.io.mem_out
mem_inW := r.io.writeData

printf(p"Instruction pulled: ${Hexadecimal(instr_out)}\n")
printf(p"PC pulled: $pc_pulled\n")
printf(p"mem_in($mem_inW) -------- mem_out($mem_outW)\n")
im.io.mem_addr := r.io.pc / 4.U
r.io.instr := im.io.mem_out
io.valid := Mux(instr_out(6, 0) === "b1110011".U, 0.U, 1.U)
printf("\n\n---------NEXT INSTRUCTION---------\n")
}
```

We can first see the class declaration at the top and that has a single output declared which determines if the program is still running. An **ECALL** instruction ends the program. Next are module declarations for riscv, dmem, and imem. These variables **r**, **im**, and **dm** can now be used to deliver values to all the classes. We first pull the PC from the riscv module and give it to imem. Then the new instruction is pulled and given back to the riscv module.

```
    pc_pulled := r.io.pc / 4.U
    r.io.instr := im.io.mem_out
```

Next, values are pulled into dmem so that the correct mem_out value can be calculated. Even though a value is pulled from dmem every cycle, this does not necessarily mean that the value pulled will be used in any calculations. This will become clear when we come to the datapath module. Values are also stored into dmem if the write enable is set. The rest of the module is just debugging information.

```
    dm.io.mem_addr := r.io.aluResult
    dm.io.mem_in := r.io.writeData
    dm.io.enable := r.io.memWrite
    r.io.readData := dm.io.mem_out
```

3.2 RISCV Module

Next is the **riscv** module which moves data between the datapath and decoder. First are the declarations for the decoder module (d) and the datapath module (dp).

```
    val d = Module(new decoder)
    val dp = Module(new datapath)
```

There will be occasional omissions of print statements/debugging information that isn't completely relevant. Data from the instruction is fed into the decoder module. Things like opcode, funct7, funct3, destination register (rd), and some control signals from the alu that come out of the datapath.

```
    d.io.opcode := io.instr(6,0)
    d.io.funct7 := io.instr(31,25)
    d.io.funct3 := io.instr(14,12)
    d.io.rd := io.instr(11,7)
    d.io.zero := dp.io.zero
    d.io.lt := dp.io.lt
    d.io.gt := dp.io.gt
```

These are all the signals that the decoder needs to calculate control signals that will be fed into the datapath.

```
    dp.io.regSrc := d.io.regSrc
    dp.io.regWrite := d.io.regW
    dp.io.immSrc := d.io.immSrc
    dp.io.aluSrc := d.io.aluSrc
    dp.io.aluControl := d.io.aluControl
    dp.io.memToReg := d.io.memW
    dp.io.instr := io.instr
    dp.io.readData := io.readData
    dp.io.branchSrc := d.io.branchSrc
```

The pc is then fetched from the datapath and fed back up to the top module to fetch the next instruction along with data that is going to be written into memory.

```
    io.pc := dp.io.pc
    io.memWrite := d.io.memW
    io.aluResult := dp.io.dataAdd
    io.writeData := dp.io.writeData
    io.memImmP := dp.io.memImmP
```

3.3 Decoder Module

The decoder is the module sets sets most of the control signals that determine what the alu will do, if a branch is necessary, if a value is being written to a register, if a value pulled from memory is begin written to a register, etc. First, the class declaration.

```
  class decoder extends Module {
    val io = IO(new Bundle {
        val opcode = Input(UInt(7.W))
        val funct7 = Input(UInt(7.W))
        val funct3 = Input(UInt(3.W))
        val rd = Input(UInt(5.W))
        val regSrc = Output(UInt(3.W))
        val regW = Output(UInt(1.W))
        val immSrc = Output(UInt(2.W))
        val aluSrc = Output(UInt(1.W))
        val aluControl = Output(UInt(4.W))
        val memW = Output(UInt(1.W))
        val memToReg = Output(UInt(1.W))
        val branchSrc = Output(UInt(1.W))
        val zero = Input(UInt(1.W))
        val lt = Input(UInt(1.W))
        val gt = Input(UInt(1.W))
    })
```

How the decoder determines what control signals to set is a long list of when statements ("if statements" for other programming languages) that uses the opcode, funct7, and funct3 control signals received from the instruction. Since the when statements are relatively self-explanatory and very long there will be no explanation for every block.

```
    when(io.opcode === "b0110011".U) {
      io.regSrc := "b000".U
      io.immSrc := "b00".U
      io.aluSrc := 0.U
      io.memToReg := 0.U
      io.regW := 1.U
      io.memW := 0.U
      io.branchSrc := 0.U

      when(io.funct7 === "b0000000".U) {
          when(io.funct3 === "b000".U) {
              io.aluControl := 2.U
          }.elsewhen(io.funct3 === "b001".U) {
              io.aluControl := 3.U
          }.elsewhen(io.funct3 === "b010".U) {
              io.aluControl := 9.U
          }.elsewhen(io.funct3 === "b011".U) {
              io.aluControl := 5.U
          }.elsewhen(io.funct3 === "b100".U) {
              io.aluControl := 6.U
          }.elsewhen(io.funct3 === "b101".U) {
              io.aluControl := 7.U
          }.elsewhen(io.funct3 === "b110".U) {
              io.aluControl := 1.U
```

```
        }.otherwise {
            io.aluControl := 0.U
        }
    }.otherwise {
        when(io.funct3 === "b000".U) {
            io.aluControl := 8.U
        }.otherwise {
            io.aluControl := 4.U
        }
    }
}.elsewhen (io.opcode === "b0010011".U) {
    io.regSrc := "b000".U
    io.immSrc := "b00".U
    io.aluSrc := 1.U
    io.memToReg := 0.U
    io.regW := 1.U
    io.memW := 0.U
    io.branchSrc := 0.U

    when(io.funct3 === "b000".U) {
        io.aluControl := 2.U
    }.elsewhen(io.funct3 === "b001".U) {
        io.aluControl := 3.U
    }.elsewhen(io.funct3 === "b010".U) {
        io.aluControl := 9.U
    }.elsewhen(io.funct3 === "b011".U) {
        io.aluControl := 5.U
    }.elsewhen(io.funct3 === "b100".U) {
        io.aluControl := 6.U
    }.elsewhen(io.funct3 === "b101".U) {
        io.aluControl := 7.U
    }.elsewhen(io.funct3 === "b110".U) {
        io.aluControl := 1.U
    }.otherwise {
        io.aluControl := 0.U
    }
}.elsewhen (io.opcode === "b0000011".U) {
    io.regSrc := "b000".U
    io.immSrc := "b00".U
    io.aluSrc := 1.U
    io.memToReg := 1.U
    io.regW := 1.U
    io.memW := 1.U
    io.branchSrc := 0.U
    io.aluControl := 0.U
}.elsewhen (io.opcode === "b0100011".U) {
    io.regSrc := "b000".U
    io.immSrc := "b00".U
    io.aluSrc := 1.U
    io.memToReg := 0.U
    io.regW := 0.U
    io.memW := 1.U
    io.branchSrc := 0.U
    io.aluControl := 0.U
}.elsewhen (io.opcode === "b1100011".U) {
    io.regSrc := "b000".U
    io.immSrc := "b01".U
    io.aluSrc := 0.U
    io.memToReg := 0.U
    io.regW := 0.U
    io.memW := 0.U
    io.aluControl := 4.U

    when(io.funct3 === "b000".U & io.zero === 1.U){
        io.branchSrc := 1.U
    }.elsewhen(io.funct3 === "b001".U
       & io.zero === 0.U) {
        io.branchSrc := 1.U
    }.elsewhen(io.funct3 === "b100".U
       & io.lt === 1.U) {
        io.branchSrc := 1.U
    }.elsewhen(io.funct3 === "b101".U
       & io.gt === 1.U) {
        io.branchSrc := 1.U
    }.elsewhen(io.funct3 === "b110".U
       & io.lt === 1.U) {
        io.branchSrc := 1.U
    }.elsewhen(io.funct3 === "b111".U
       & io.gt === 1.U) {
        io.branchSrc := 1.U
    }.otherwise {
        io.branchSrc := 0.U
    }

}.elsewhen (io.opcode === "b1101111".U) {
    io.regSrc := "b100".U
    io.immSrc := "b10".U
    io.aluSrc := 1.U
    io.memToReg := 0.U
    io.regW := 1.U
    io.memW := 0.U
    io.branchSrc := 1.U
    io.aluControl := 0.U
}.otherwise {
    io.regSrc := "b000".U
    io.immSrc := "b00".U
    io.aluSrc := 0.U
    io.memToReg := 0.U
    io.regW := 0.U
    io.memW := 0.U
    io.branchSrc := 0.U
    io.aluControl := 0.U
}
```

The rest of the module is purely debugging information.

3.4 Datapath Module

The datapath connects many modules together including the alu, register file, and extend module. Here is the class declaration.

```
class datapath extends Module {
  val io = IO(new Bundle {
    val regSrc = Input(UInt(3.W))
    val regWrite = Input(UInt(1.W))
    val immSrc = Input(UInt(2.W))
    val aluSrc = Input(UInt(1.W))
    val aluControl = Input(UInt(4.W))
    val memToReg = Input(UInt(1.W))
    val instr = Input(UInt(32.W))
    val readData = Input(UInt(32.W))
    val branchSrc = Input(UInt(1.W))
    val pc = Output(UInt(32.W))
    val dataAdd = Output(UInt(32.W))
    val writeData = Output(UInt(32.W))
    val zero = Output(UInt(1.W))
    val lt = Output(UInt(1.W))
    val gt = Output(UInt(1.W))
    val memImmP = Output(UInt(32.W))
  })
```

First, module declarations and necessary wires are made. Two extends modules are needed because two different values, branchExtImm and extImm, need to be extended.

```
val ext1 = Module(new extend)
val rf = Module(new regfile)
val ext2 = Module(new extend)
val alu = Module(new alu)

val memImmStore = Wire(UInt(32.W))
val memImm = Wire(UInt(32.W))
val branchImm = Wire(UInt(32.W))
val jumpImm = Wire(UInt(32.W))
val branchExtImm = Wire(UInt(32.W))
val extImm = Wire(UInt(32.W))
val memToRegW = Wire(UInt(1.W))
val readDataW = Wire(UInt(32.W))
val aluOutW = Wire(UInt(32.W))
val ra1 = Wire(UInt(5.W))
val ra2 = Wire(UInt(5.W))
val ra4 = Wire(UInt(32.W))
val srcB = Wire(UInt(32.W))
val result = Wire(UInt(32.W))
```

These wires are needed to hold values from muxs or, in the case of the immediates, to hold values that are concatenated together. Next is the branching logic.

```
branchImm := Cat(io.instr(31), io.instr(7), io.instr(30,25),
   io.instr(11,8))
jumpImm := Cat(io.instr(31), io.instr(19,12), io.instr(20),
   io.instr(30,21))
ext1.io.instr12 := branchImm
ext1.io.instr20 := jumpImm
ext1.io.immSrc := io.immSrc
ext2.io.instr12 := io.instr(31,20)
ext2.io.instr20 := jumpImm
ext2.io.immSrc := io.immSrc
branchExtImm := ext1.io.extImm
extImm := ext2.io.extImm
```

BranchImm and jumpImm are both calculated from the 32 bit instruction and then extended using the extend module. PC logic is then calculated next. Wires are made to hold calculated values. Remember that when "placeholder" variables are needed, use **Wire()**.

```
//PC logic
val pcReg = RegInit (0.U(32.W))
val pcNext = Wire(UInt(32.W))
val pcBranch = Wire(UInt(32.W))
val pcPlus8 = Wire(UInt(32.W))
val pcPlus4 = Wire(UInt(32.W))
pcPlus4 := pcReg + "b100".U
pcPlus8 := pcPlus4 + "b100".U
pcBranch := branchExtImm + pcReg
pcNext := Mux(io.branchSrc.andR, pcBranch, pcPlus4)
```

```
  pcReg := pcNext
  io.pc := pcReg
```

One things to note here is that any base number (binary, decimal, hexadecimal) can be added to any other base base number by casting it as that base. When calculating pcPlus4, **"b100".U** was used to add the binary number 4 to pcReg. Quotes are required followed by what base the number is. **"h4".U** and **4.U** (no quotes needed) are all equal. The **.U** is always required when adding unsigned numbers to chisel types. Next, result is calculated using a mux followed by memory logic.

```
    memImmStore := Cat(io.instr(31,25), io.instr(11,7))
    memImm := Mux(io.memToReg.andR, extImm, memImmStore)
    io.memImmP := memImm
    io.dataAdd := rf.io.rd1 + memImm

    result := Mux(io.memToReg.andR, io.readData, alu.io.out)
```

dataAdd is the memory address of either the value to be pulled or to be stored. Since this is a single cycle architecture a value can only be stored or pulled on one cycle allowing for only one variable to be used for storing and pulling. Notice that when calculating memImm, the first parameter in the mux is **io.memToReg.andR**. The Mux function requires a boolean value in the first parameter. Since io.memToReg is a 1 bit value, using .andR at the end will and the bits and return a boolean value. If the 1 bit value is 1 then it will return true. If 0, then it will return false. Next is the register file logic.

```
    //regFile logic
    ra1 := Mux(io.regSrc(0).andR, "b11111".U, io.instr(19,15))
    ra2 := Mux(io.regSrc(1).andR, io.instr(11,7), io.instr(24,20))
    ra4 := Mux(io.regSrc(2).andR, pcPlus4, result)
    rf.io.we3 := io.regWrite
    rf.io.ra1 := ra1
    rf.io.ra2 := ra2
    rf.io.wa3 := io.instr(11,7)
    rf.io.wd3 := ra4
    rf.io.r31 := pcPlus8
    io.writeData := rf.io.rd2
```

In RISC-V, almost all types of instructions have the same format. The values for the destination, rs1, and rs2 registers occupy the same bits inside the instruction allowing for less work to retrieve/set the values inside those registers. For arithmetic and logical instructions rs1 occupies the bits 19-15, ra2 occupies 24-20, and the destination register occupies 11-7. Muxs are used to determine from what register values are to be pulled. For ra1, is io.regSrc(0) is 1, then the value put into ra1 will be the value in the 32nd ("b11111") register. ra2 also uses a mux to switch between the destination register or rs2. ra4 is the value that is going to be written to a register. More of this will become clear when looking at the register file module. Lastly is the ALU logic.

```
//ALU logic
srcB := Mux(io.aluSrc.andR, extImm, rf.io.rd2)

alu.io.a := rf.io.rd1
alu.io.b := srcB
alu.io.aluControl := io.aluControl
alu.io.imm := io.aluSrc
io.zero := alu.io.zero
io.lt := alu.io.lt
io.gt := alu.io.gt
```

The value srcB uses a mux to switch between either rd2 or extImm which is used for branches and jumps. Next is just setting the values inside of the alu module and then retrieving control. We saw before that the alu result was already pulled into the variable **result**.

3.5 Register File Module

We've seen before the values being set inside the decoder module and it may have been slightly confusing. Looking at the class declaration and seeing what signals are inputs and outputs will help clear up confusion.

```
class regfile extends Module {
 val io = IO(new Bundle {
     val we3 = Input(UInt(1.W))
     val ra1 = Input(UInt(5.W))
     val ra2 = Input(UInt(5.W))
     val wa3 = Input(UInt(5.W))
     val wd3 = Input(UInt(32.W))
     val r31 = Input(UInt(32.W))
     val rd1 = Output(UInt(32.W))
     val rd2 = Output(UInt(32.W))
 })
```

Here is a list of each signal and what it does inside the module:

regfile			
Signal Name	Bit Width	Input or Output	Function of signal
we3	1	Input	Write enable
ra1	5	Input	Address of first reg value to be pulled
ra2	5	Input	Address of second reg value to be pulled
wa3	5	Input	Address of register to be written to
wd3	32	Input	Value to be written into register
r31	32	Input	Always "b11111"
rd1	32	Output	Value of first register
rd2	32	Output	Value of second register

Now that an understanding of what each signal does has been made, the rest of the register file module is extremely simple. First, the variable rf is made into a chisel memory type with 32 positions for 32 bit unsigned integers. Next, if the write enable signal is set and the destination register is not register 0, the value wd3 is set in the register at position wa3.

```
val rf = Mem(32, UInt(32.W))

 when(io.we3.andR && !(io.wa3 === 0.U)){
     rf(io.wa3) := io.wd3
 }.otherwise {
     rf(0.U) := 0.U
 }
```

In RISC-V the first register is reserved solely for the value 0. The otherwise statement takes care to set 0 to the first register every time another value tries to be set in register 0. Next, values are pulled from registers and given back to the datapath.

```
io.rd1 := Mux((io.ra1 === 31.U), rf(io.r31), rf(io.ra1))
io.rd2 := Mux((io.ra2 === 31.U), rf(io.r31), rf(io.ra2))
```

3.6 ALU Module

The ALU module deals with all the arithmetic and logical operations. The ALU also sets some control signals used for branching. First, the class declaration.

```
val io = IO(new Bundle {
     val a = Input(UInt(32.W))
     val b = Input(UInt(32.W))
     val aluControl = Input(UInt(4.W))
     val imm = Input(UInt(1.W))
     val out = Output(UInt(32.W))

     val zero = Output(Bool())
     val lt = Output(Bool())
     val gt = Output(Bool())
 })
```

The signals a and b are the two values that are operated on while aluControl and imm are control signals. **out** is obviously the output of the calculations. zero, lt, and gt are the control signals that are sent to the decoder to determine if a branch should occur. Like the decoder, the alu uses when statements to determine what operation it should perform.

```
  val sum = Wire(UInt(32.W))

   when(io.aluControl(3) === 1.U){
       sum := io.a + ~io.b + io.aluControl(3)
   }.otherwise {
       sum := io.a + io.b
   }

   when (io.aluControl === "b0000".U) {
       io.out := io.a & io.b
   }.elsewhen (io.aluControl === "b0001".U) {
       io.out := io.a | io.b
   }.elsewhen (io.aluControl === "b0010".U) {
       io.out := sum
   }.elsewhen (io.aluControl === "b0011".U) {
       when (io.imm.andR) {
           io.out := io.a << io.b(4, 0)
       }.otherwise {
           io.out := io.a << io.b(18,0)
       }
   }.elsewhen (io.aluControl === "b0100".U) {
       when (io.imm.andR) {
           io.out := io.a >> io.b(4, 0)
       }.otherwise {
           io.out := io.a >> io.b(18,0)
       }
   }.elsewhen (io.aluControl === "b0110".U) {
       io.out := io.a ^ io.b
   }.elsewhen (io.aluControl === "b0111".U) {
       when (io.imm.andR) {
           io.out := io.a >> io.b(4, 0)
       }.otherwise {
           io.out := io.a >> io.b(18,0)
       }
   }.elsewhen(io.aluControl === "b1000".U) {
       io.out := sum
   }.elsewhen(io.aluControl === "b1001".U) {
       when (sum(31).andR) {
           io.out := 1.U
       }.otherwise {
           io.out := 0.U
       }
   }.otherwise {
       io.out := 0.U
   }
```

Next the control signals are set for branching.

```
when(io.a - io.b === 0.U)
 {
     io.zero := 1.U
 }.otherwise{
     io.zero := 0.U
 }
 io.lt := (io.a < io.b)
 io.gt := (io.a > io.b)
```

3.7 Extend Module

The extend module is used for extending values into 32 bits. Not much explanation is needed.

```
class extend extends Module {
 val io = IO(new Bundle {
     val instr12 = Input(UInt(12.W))
     val instr20 = Input(UInt(20.W))
     val immSrc = Input(UInt(2.W))
     val extImm = Output(UInt(32.W))
 })

 when(io.immSrc === 0.U){
     when(io.instr12(11) === 1.U){
         io.extImm := Cat("b11111111111111111111".U, io.instr12)
     }.otherwise {
         io.extImm := Cat("b00000000000000000000".U, io.instr12)
     }
 }.elsewhen(io.immSrc === 1.U){
        when(io.instr12(11) === 1.U){
            io.extImm := Cat("b11111111111111111111".U, io.instr12,
                "b0".U)
        }.otherwise{
            io.extImm := Cat("b00000000000000000000".U, io.instr12,
                "b0".U)
        }
    }.elsewhen(io.immSrc === 2.U){
        when(io.instr20(19) === 1.U){
            io.extImm := Cat("b11111111111".U, io.instr20, "b0".U)
        }.otherwise{
            io.extImm := Cat("b00000000000".U, io.instr20, "b0".U)
        }
    }.otherwise {
        io.extImm := 0.U
    }
  }
```

3.8 Memory Modules

Next, we move on to the instruction memory and main memory modules which are both very similar. First, the instruction memory.

```
   class imem extends Module {
    val io = IO(new Bundle {
        val mem_addr = Input(UInt(32.W))
        val mem_out = Output(UInt(32.W))
    })

    val MEM = Mem(1024, UInt(32.W))
    loadMemoryFromFile(MEM, "/PATH/TO/MEM/FILE")

    io.mem_out := MEM(io.mem_addr)

   }
```

The instruction memory module uses the function **loadMemoryFromFile()** to load the hexadecimal instructions from a text file at the given path into the **MEM** variable. The rest of the module is self-explanatory. Main memory works very much the same except that it has a write enable signal.

```
class dmem extends Module {
 val io = IO(new Bundle {
     val mem_addr = Input(UInt(32.W))
     val mem_in = Input(UInt(32.W))
     val enable = Input(UInt(1.W))
     val mem_out = Output(UInt(32.W))
 })
 val mem = SyncReadMem(1024, UInt(32.W))
 mem.write(io.mem_addr, io.mem_in)
 io.mem_out := mem.read(io.mem_addr, io.enable.andR)
}
```

The write enable is necessary due to the fact that it would write random values into memory without it. A value for mem_out is always grabbed and sent back to the datapath. This is fine because the pulled value will only be used if the **memToReg** control signal is set.

3.9 Supplementary

Lastly, two more classes are required to simulate the processor. The first is a testing class that extends **PeekPokeTester** which is a Chisel test harness that allows the user to "poke" (push) values into signals, "peek" at there value, and throw assertions for expected values.

```
class riscvSingleTest(t: top) extends PeekPokeTester(t) {
   var cycles = 0
   var validP = peek(t.io.valid)
   println(s"Starting valid = $validP")
   while (peek(t.io.valid) == BigInt(1) && cycles < 100) {
      validP = peek(t.io.valid)
      println(s"valid = $validP")
      step(1)
      cycles += 1
   }

   if (cycles > 98 ) {
     println(s"$cycles cycles were ran and end of program not
        reached. Exiting.")
     System.exit(0)
   }
   else {
     println(s"Program completed in $cycles cycles. Exiting.")
   }
 }
```

The important part of this class is the while loop that checks if the valid flag is set and also if the number of cycles is above 100. A cycle limit has been given so that an infinite loop does not occur. While these conditions are true, the program will step one cycle and continue to load instructions into memory. The rest is purely debugging information. In addition to the testing class, and object of **top** must be made to attach the testing class to the top module.

```
  object top extends App {
    iotesters.Driver.execute(args, () => new top) {
      t => new riscvSingleTest(t)
    }
  }
```

3.10 Running the Processor

To run this processor, navigate to the **MyChiselProjects** directory in the terminal. The main program (**riscvSingle.scala**) located in MyChiselProject/src/main/scala/ is where changes can be made to the source code. Next, run the script

```
./runProject riscvSingle [test_file_name]
```

This will run all the necessary commands to run the specific project. Test files can be found in MyChiselProject/tests/. The path starting from tests/ should be used. Requirements for tests are

- instructions must be in hexadecimal
- one instruction per line

- last instruction must be **ECALL** to end test
- file must have a .x extension

will work. DO NOT ADD EXTENSIONS on the project file name or the test file name when entering into the script.

3.11 Work in Progress

This processor along with this documentation are still a work in progress and have room to be improved. Due to scripts that were made to make simulating the processor easier and faster, more time can be given to improving the processors architecture. The following are the improvements/additions that will be implemented in the future.

1. Adding more instructions
2. Adding a floating point module

These are the top priority items that will be added when time/obligations allow.