RISC-V Processor Developed in the Chisel Programming Language

Ryan Ridley

August 2019

Chapter 1

1.1 Overview

RISC-V is a reduced instruction set architecture that is becoming more mainstream in today's ISA. Chisel, which is a hardware description language embedded in the Scala programming language, is also becoming more popular due to it's object oriented style of hardware programming compared to other HDL languages This document is an overview of how a RISC-V processor was created in Chisel.

1.2 Chisel Introduction

Chisel is a hardware description language embedded in the Scala programming language. This HDL allows the user to describe hardware in a highly parameterized, object oriented way. This is what makes Chisel more attractive than previous HDL languages. Chisel allows the user to output a Verilog description of the hardware designed with Chisel. Designing complicated hardware is made easier with Chisel which can then be converted to Verilog if needed.

1.3 Chisel Template

To begin making a Chisel project, the project template must first be downloaded. This project template deals with the boiler plate code that Chisel needs to create the project. When creating a custom project, the source files containing the hardware description must be in the directory src/main/s-

cala. 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. This 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
    }
  }
```

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) and "peek" (view) 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
    }
  }
```

ALU is the main class that holds all of our logic. Next, a testing class needs to be made. These two classes can live inside the same file. 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 ran. 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. The best way to print internal signals to the console is to use a toPrintable custom message.

```
class MessageRiscv extends Bundle {
   val and = UInt(32.W)

   override def toPrintable: Printable = {
      p"and = 0x${Hexadecimal(and)}\n"
   }
}

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

 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, constructing a single cycle processor is now possible. 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
- messages

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 brings the instruction memory (imem) and main memory (dmem) into the main processing unit (riscv).

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

   val topMessage = Wire(new MessageTop)
   val r = Module(new riscv)
   val im = Module(new imem)
   val dm = Module(new dmem)

   dm.io.memAddress := r.io.memAddress
   dm.io.memWriteData := r.io.memWriteData
   dm.io.memWriteEnable := r.io.memWriteEnable

   r.io.memReadData := dm.io.memReadData

   // print info
   topMessage.instr_pulled := im.io.inst
   topMessage.pc_pulled := r.io.pc / 4.U
   topMessage.memWriteData := r.io.memWriteData
   topMessage.memWriteEnable := r.io.memWriteEnable
   topMessage.memAddress := r.io.memAddress
   topMessage.memReadData := dm.io.memReadData
   printf(p"$topMessage")

   im.io.instAddress := r.io.pc / 4.U

   r.io.instr := im.io.inst

   io.valid := Mux(im.io.inst(6, 0) === "b1110011".U, 0.U, 1.U)
}
```

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` instruc-

tion ends the program. Once an `ECALL` instruction has been run, valid is set to 0. This is necessary for the test class to stop stepping through the program. Next are module declarations for `MessageTop`, `riscv`, `imem`, and `dmem`. These variables `r`, `im`, and `dm` can now be used to deliver values to all the classes. Next, data that needs to be stored in memory is handled. The memory address (`memAddress`), data to be written to that memory location (`memWriteData`), and the write enable (`memWriteEnable`) are given to the dmem module.

```
    dm.io.memAddress := r.io.memAddress
    dm.io.memWriteData := r.io.memWriteData
    dm.io.memWriteEnable := r.io.memWriteEnable
```

The next line pulls data from memory into the processor.

```
  r.io.memReadData := dm.io.memReadData
```

Every cycle, data is pulled from memory. This does not mean that the data pulled will be used. Next is printing information used for debugging. Next, the pc must be calculated so that the instruction memory can pull the correct instruction on the next cycle. This is then given to the riscv module.

```
  im.io.instAddress := r.io.pc / 4.U
  r.io.instr := im.io.inst
```

Lastly, valid is calculated by seeing if an `ECALL` instruction has called.

3.2 RISCV Module

Next is the `riscv` module which moves data between the datapath, decoder, and top modules.

```
class riscv extends Module {
    val io = IO(new Bundle {
    val instr = Input(UInt(32.W))
    val memReadData = Input(SInt(32.W))
    val pc = Output(UInt(32.W))
    val memWriteEnable = Output(UInt(1.W))
    val memAddress = Output(UInt(32.W))
    val memWriteData = Output(SInt(32.W))
```

```
})

    val riscvMessage = Wire(new MessageRiscv)
    val dp = Module(new datapath)
    val d = Module(new decoder)
```

The inputs into the module come from top and are the instruction to be ran as well as the data from memory. The outputs are the pc used to calculate the next instruction and the three signals used to write data to memory. Lastly, the declarations for the modules that signals will be sent to. The following is the debugging information.

```
riscvMessage.instr := io.instr
riscvMessage.memReadData := io.memReadData
riscvMessage.memWriteEnable := io.memWriteEnable
riscvMessage.memWriteData := io.memWriteData
riscvMessage.memAddress := io.memAddress
printf(p"$riscvMessage")
```

Next, the required signals are given to the decoder module.

```
d.io.opcode := io.instr(6,0)
d.io.funct7 := io.instr(31,25)
d.io.funct3 := io.instr(14,12)
d.io.zeroFlag := dp.io.zeroFlag
d.io.lessThanFlag := dp.io.lessThanFlag
d.io.greaterThanFlag := dp.io.greaterThanFlag
```

The opcode, funct7, and funct3 will be used by the decoder to determine which instruction to run. The rest are flags taken from the datapath module to determine if a branch should be taken. Next, signals are sent to the datapath module.

```
dp.io.regSrc := d.io.regSrc
dp.io.regWriteEnable := d.io.regWriteEnable
dp.io.immSrc := d.io.immSrc
dp.io.aluSrc := d.io.aluSrc
dp.io.pcSrc := d.io.pcSrc
dp.io.aluControl := d.io.aluControl
dp.io.memToReg := d.io.memToReg
```

```
dp.io.instr := io.instr
dp.io.memReadData := io.memReadData
dp.io.branchSrc := d.io.branchSrc
```

The datapath uses these signals to control the flow of data. These signals and their purpose will be explained in the datapath section. Lastly, the outputs of the riscv module are set.

```
io.pc := dp.io.pc
io.memWriteEnable := d.io.memWriteEnable
io.memAddress := dp.io.memAddress
io.memWriteData := dp.io.memWriteData
```

The pc and the data memory signals are sent back to the top module.

3.3 Decoder Module

The decoder module sets most of the control signals that the datapath and ALU use for program flow. 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, are all dependent on the control signals set in the decoder. 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 regSrc = Output(UInt(3.W))
      val regWriteEnable = Output(UInt(1.W))
      val immSrc = Output(UInt(2.W))
      val aluSrc = Output(UInt(1.W))
      val pcSrc = Output(UInt(1.W))
      val aluControl = Output(UInt(4.W))
      val memWriteEnable = Output(UInt(1.W))
      val memToReg = Output(UInt(1.W))
      val branchSrc = Output(UInt(2.W))
      val zeroFlag = Input(UInt(1.W))
      val lessThanFlag = Input(UInt(1.W))
```

```
    val greaterThanFlag = Input(UInt(1.W)) })
```

How the decoder determines what control signals to set is a long list of when statements ("if statements" in 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 := 0.U
    io.immSrc := 0.U
    io.aluSrc := 0.U
    io.pcSrc := 0.U
    io.memToReg := 0.U
    io.regWriteEnable := 1.U
    io.memWriteEnable := 0.U
    io.branchSrc := 0.U

   when(io.funct7 === "b0000000".U) {
      when(io.funct3 === "b000".U) {      //ADD
         io.aluControl := 2.U
      }.elsewhen(io.funct3 === "b001".U) { //SLL
         io.aluControl := 3.U
      }.elsewhen(io.funct3 === "b010".U) { //SLT
         io.aluControl := 9.U
      }.elsewhen(io.funct3 === "b011".U) { //SLTU
         io.aluControl := 5.U
      }.elsewhen(io.funct3 === "b100".U) { //XOR
         io.aluControl := 6.U
      }.elsewhen(io.funct3 === "b101".U) { //SRL
         io.aluControl := 7.U
      }.elsewhen(io.funct3 === "b110".U) { //OR
         io.aluControl := 1.U
      }.elsewhen(io.funct3 === "b111".U) { //AND
         io.aluControl := 0.U
      }.otherwise {                       //NONE
         io.aluControl := 15.U
      }
    }.elsewhen(io.funct7 === "b0000001".U){
      when(io.funct3 === "b000".U){       //MUL
```

```
        io.aluControl := 8.U
```

```
      }.elsewhen(io.funct3 === "b100".U){ //DIV
        io.aluControl := 10.U
      }.otherwise {
        io.aluControl := 15.U            //NONE
      }
    }.elsewhen(io.funct7 === "b0100000".U){
      when (io.funct3 === "b101".U) {     //SRA
        io.aluControl := 4.U
      }.elsewhen(io.funct3 === "b000".U){ //SUB
        io.aluControl := 12.U
      }.otherwise {
        io.aluControl := 15.U            //NONE
      }
    }.otherwise {
      io.aluControl := 15.U               //NONE
    }
}.elsewhen(io.opcode === "b0010111".U){    //AUIPC
    io.regSrc := 0.U
    io.immSrc := 2.U
    io.aluSrc := 1.U
    io.pcSrc := 1.U
    io.memToReg := 0.U
    io.regWriteEnable := 1.U
    io.memWriteEnable := 0.U
    io.branchSrc := 0.U
    io.aluControl := 2.U
}.elsewhen(io.opcode === "b0010011".U) {
    io.regSrc := 0.U
    io.immSrc := 0.U
    io.aluSrc := 1.U
    io.pcSrc := 0.U
    io.memToReg := 0.U
    io.regWriteEnable := 1.U
    io.memWriteEnable := 0.U
    io.branchSrc := 0.U

    when (io.funct7(6,1) === "b010000".U){
```

```
      when(io.funct3 === "b101".U){           //SRAI
```

```
        io.aluControl := 4.U
      }.otherwise {
        io.aluControl := 15.U
      }
    }.otherwise{
      when(io.funct3 === "b000".U) {          // ADDI
        io.aluControl := 2.U
      }.elsewhen(io.funct3 === "b001".U) {    // SLLI
        io.aluControl := 3.U
      }.elsewhen(io.funct3 === "b010".U) {    // SLTI
        io.aluControl := 9.U
      }.elsewhen(io.funct3 === "b011".U) {    // SLTIU
        io.aluControl := 5.U
      }.elsewhen(io.funct3 === "b100".U) {    // XORI
        io.aluControl := 6.U
      }.elsewhen(io.funct3 === "b101".U) {    // SRLI
        io.aluControl := 7.U
      }.elsewhen(io.funct3 === "b110".U) {    // ORI
        io.aluControl := 1.U
      }.otherwise {
        io.aluControl := 0.U                 // ANDI
      }
    }
}.elsewhen(io.opcode === "b0000011".U) {  // LOAD
    io.regSrc := 0.U
    io.immSrc := 0.U
    io.aluSrc := 1.U
    io.pcSrc := 0.U
    io.memToReg := 1.U
    io.regWriteEnable := 1.U
    io.memWriteEnable := 0.U
    io.branchSrc := 0.U
    io.aluControl := 2.U
}.elsewhen(io.opcode === "b0100011".U) {  // STORE
    io.regSrc := 0.U
    io.immSrc := 0.U
    io.aluSrc := 1.U
```

```
    io.pcSrc := 0.U
```

```
    io.memToReg := 0.U
    io.regWriteEnable := 0.U
    io.memWriteEnable := 1.U
    io.branchSrc := 0.U
    io.aluControl := 0.U
}.elsewhen(io.opcode === "b1100011".U) {
    io.regSrc := 0.U
    io.immSrc := 1.U
    io.aluSrc := 0.U
    io.pcSrc := 0.U
    io.memToReg := 0.U
    io.regWriteEnable := 0.U
    io.memWriteEnable := 0.U
    io.aluControl := 4.U

    when(io.funct3 === "b000".U & io.zeroFlag === 1.U){   // BEQ
      io.branchSrc := 1.U
    }.elsewhen(io.funct3 === "b001".U & io.zeroFlag === 0.U) { // BNE
      io.branchSrc := 1.U
    }.elsewhen(io.funct3 === "b100".U & io.lessThanFlag === 1.U) {
        // BLT
      io.branchSrc := 1.U
    }.elsewhen(io.funct3 === "b101".U & io.greaterThanFlag === 1.U)
        { // BGE
      io.branchSrc := 1.U
    }.elsewhen(io.funct3 === "b110".U & io.lessThanFlag === 1.U) {
        // BLTU
      io.branchSrc := 1.U
    }.elsewhen(io.funct3 === "b111".U & io.greaterThanFlag === 1.U)
        { // BGEU
      io.branchSrc := 1.U
    }.otherwise {                    // NONE
      io.branchSrc := 0.U
    }

}.elsewhen(io.opcode === "b1101111".U) {        // JAL
    io.regSrc := 4.U
```

```
    io.immSrc := 2.U
```

```
    io.aluSrc := 1.U
    io.pcSrc := 0.U
    io.memToReg := 0.U
    io.regWriteEnable := 1.U
    io.memWriteEnable := 0.U
    io.branchSrc := 1.U
    io.aluControl := 0.U
}.elsewhen(io.opcode === "b1100111".U) {        //JALR
    io.regSrc := 4.U
    io.immSrc := 0.U
    io.aluSrc := 1.U
    io.pcSrc := 0.U
    io.memToReg := 0.U
    io.regWriteEnable := 1.U
    io.memWriteEnable := 0.U
    io.branchSrc := 2.U
    io.aluControl := 2.U
}.elsewhen(io.opcode === "b1110011".U) {        // ECALL
    io.regSrc := 0.U
    io.immSrc := 0.U
    io.aluSrc := 0.U
    io.pcSrc := 0.U
    io.memToReg := 0.U
    io.regWriteEnable := 0.U
    io.memWriteEnable := 0.U
    io.branchSrc := 0.U
    io.aluControl := 0.U
}.otherwise{             // NONE
    io.regSrc := 0.U
    io.immSrc := 0.U
    io.aluSrc := 0.U
    io.pcSrc := 0.U
    io.memToReg := 0.U
    io.regWriteEnable := 0.U
    io.memWriteEnable := 0.U
    io.branchSrc := 0.U
    io.aluControl := 0.U
```

```
}
```

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 regWriteEnable = Input(UInt(1.W))
      val immSrc = Input(UInt(2.W))
      val aluSrc = Input(UInt(1.W))
      val pcSrc = Input(UInt(1.W))
      val aluControl = Input(UInt(4.W))
      val memToReg = Input(UInt(1.W))
      val instr = Input(UInt(32.W))
      val memReadData = Input(SInt(32.W))
      val branchSrc = Input(UInt(2.W))
      val pc = Output(UInt(32.W))
      val memAddress = Output(UInt(32.W))
      val memWriteData = Output(SInt(32.W))
      val zeroFlag = Output(UInt(1.W))
      val lessThanFlag = Output(UInt(1.W))
      val greaterThanFlag = Output(UInt(1.W))
```

First, module declarations and necessary wires are made. Two extends modules are needed because two different values, `branchExtImm` and `jumpImm`, need to be extended. Register write and address data wires are needed for multiplexing.

```
val datapathMessage = Wire(new MessageDatapath)
val rf = Module(new regfile)
val alu = Module(new alu)
val ext1 = Module(new extend)
val ext2 = Module(new extend)
val branchImm = Wire(UInt(12.W))
val jumpImm = Wire(UInt(12.W))
val auiImm = Wire(SInt(32.W))
val memImm = Wire(SInt(32.W))
val branchExtImm = Wire(SInt(32.W))
val pcRegBranch = Wire(UInt(32.W))
```

```
val extImm = Wire(SInt(32.W))
val regWriteData = Wire(SInt(32.W))
val regReadAddress1 = Wire(UInt(5.W))
val regReadAddress2 = Wire(UInt(5.W))
```

Notice that there are four other immediate signals, `auiImm`, `memImm`, `branchExtImm`, `extImm`, that are not using an extend module. These signals do not need to be extended but are still immediate signals that need to be a `Wire()`.

```
//Immediate 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))
auiImm := (Cat(io.instr(31,12), 0.U(12.W))).asSInt
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 := Mux(io.pcSrc.andR, auiImm, ext2.io.extImm)
```

`branchImm` and `jumpImm` are both calculated from the instruction. They are also only 12-bit signals which is why they need to be extended. PC logic is then calculated next. Wires are made to hold calculated values. Remember that when a signal not being used as an input/output of a module or a "placeholder" variable, it needs to be a `Wire()}`.

```
//PC logic
val pcReg = RegInit (0.U(32.W))
val pcNext = Wire(UInt(32.W))
val pcBranch = Wire(SInt(32.W))
val pcPlus8 = Wire(UInt(32.W))
val pcPlus4 = Wire(UInt(32.W))
```

`pcNext`, `pcNext`, `pcNext`, and `pcNext` are

```
pcPlus4 := pcReg + 4.U
pcPlus8 := pcPlus4 + "b100".U
```

```
pcBranch := branchExtImm + pcPlus4.asSInt
pcRegBranch := alu.io.out.asUInt & "hFFFFFFFE".U
pcNext := Mux(io.branchSrc(1).andR, pcRegBranch,
    Mux(io.branchSrc(0).andR, pcBranch.asUInt, pcPlus4))
pcReg := pcNext
io.pc := pcReg
```

Any base number (binary, decimal, hexadecimal) can be added to any other base number by casting it as that base. When calculating pcPlus8, `"b100".U` was used to add the binary number 4 to pcReg. Quotes are required followed by what base the number is. `"h4".U`, `"b100".U`, and `4.U` are all equal. The `.U` is always required when adding, subtracting, multiplying, etc, unsigned numbers to chisel types. Next, the address for data memory is calculated.

```
//Mem logic
memImm := (Cat(io.instr(31,25), io.instr(11,7))).asSInt
io.memAddress := ((Mux(io.memToReg.andR, extImm, memImm)).asSInt
    + rf.io.regReadData1).asUInt
```

`memAddress` is calculated by adding `memImm` to the signal pulled from the register file (`regReadData1`). The register file logic comes next and uses a series of multiplexers to determine the correct address from where the signals will be pulled.

```
//regFile logic
regReadAddress1 := Mux(io.regSrc(0).andR, "b11111".U,
    io.instr(19,15))
regReadAddress2 := Mux(io.regSrc(1).andR, io.instr(11,7),
    io.instr(24,20))
regWriteData := Mux(io.regSrc(2).andR, pcPlus4.asSInt,
    Mux(io.memToReg.andR, io.memReadData, alu.io.out))
```

`regSrc` is a 3-bit control signal assigned in the decoder module. When calculating `regWriteData`, nested `Mux` functions are used to determine what signal should be written to the register file, if any. If `regSrc(2) == 1`, then `pcPlus4` is written to the register file. If `regSrc(2)` does not equal one, the nested `Mux` function is entered. `memToReg` is the control signal used to determine if the signal from memory should be written (`memToReg == 1`) or the output from the ALU should be written (`meToReg == 0`).

```
rf.io.regWriteEnable := io.regWriteEnable
rf.io.regReadAddress1 := regReadAddress1
rf.io.regReadAddress2 := regReadAddress2
rf.io.regWriteAddress := io.instr(11,7)
rf.io.regWriteData := regWriteData
io.memWriteData := rf.io.regReadData2
```

Next, the calculated signals are then set to their corresponding signals in the `regFile` module.

In RISC-V, almost all types of instructions have the same format. The values for the source registers, `regReadAddress1` and `regReadAddress`, occupy the same bits inside the instruction allowing for less work to retrieve/set the values inside those registers. For every instruction type except **U** and **UJ**, `regReadAddress` occupies bits 19-15. For every instruction type except **U**, **UJ**, and **I**, which does not require a second signal from the register file, `regReadAddress2` occupies bits 24-20. For the destination register, every instruction type except **S** and **SB**, which are branch instructions, `regWriteData (rd)` occupies bits 11-7.

Lastly is the ALU logic.

```
//ALU Logic
alu.io.a := Mux(io.pcSrc.andR, pcPlus4.asSInt, rf.io.regReadData1)
alu.io.b := Mux(io.aluSrc.andR, extImm, rf.io.regReadData2)
alu.io.aluControl := io.aluControl
io.zeroFlag := alu.io.zeroFlag
io.lessThanFlag := alu.io.lessThanFlag
io.greaterThanFlag := alu.io.greaterThanFlag
```

The first signal, `alu.io.a`, uses a `Mux` with `pcSrc` as the control signal to determine if the pc or register signal is assigned. The same method is used to define `alu.io.b` by using a `Mux` with `aluSrc` as the control signal, and picks between the immediate value or the second register signal. Finally, `aluControl` is passed into the module and the zero, less than, and greater than flags are grabbed from the ALU module.

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 regWriteEnable = Input(UInt(1.W))
     val regWriteAddress = Input(UInt(5.W))
     val regWriteData = Input(SInt(32.W))
     val regReadAddress1 = Input(UInt(5.W))
     val regReadAddress2 = Input(UInt(5.W))
     val regReadData1 = Output(SInt(32.W))
     val regReadData2 = Output(SInt(32.W))
 })
```

Although the name of each signal is relatively self-explanatory, a list of each signal and the purpose it serves inside the module is shown below:

regfile			
Signal Name	Bit Width	Input or Output	Function of signal
regWriteEnable	1	Input	Write enable
regWriteAddress	5	Input	Address of register to be written to
regWriteData	32	Input	Value to be written into register
regReadAddress1	5	Input	Address of first reg value to be pulled
regReadAddress2	5	Input	Address of second reg value to be pulled
regReadData1	32	Output	Value of first register
regReadData	32	Output	Value of second register

The register file is one of the few modules that does not include any external module declarations since values are only taken from the register file into the datapath.

```
val rf = Mem(32, SInt(32.W))
val regfileMessage = Wire(new MessageRegFile)

when(io.regWriteEnable.andR && !(io.regWriteAddress === 0.U)){
   rf(io.regWriteAddress) := io.regWriteData
}.otherwise {
   rf(0.U) := 0.S
```

```
}
```

Chisel has a built in datatype for memory. Using the `Mem` keyword, a combinational/asynchronous-read, sequential/synchronous-write random-access memory structure can be made. If a sequential/synchronous-read, sequential/synchronous-write memory structure is needed then the `SynceReadMem` keyword can be used. The `Mem` construct is used here with 32 entries, each 32 bits in size. The `when` statement makes sure that regWriteEnable is set to 1 and that the address to write to is not register 0. In RISC-V, register 0 is reserved solely for the value 0. All other registers are general purpose register that can be set to any value. The `otherwise` conditional ensures that register 0 remains at the value 0.

Next, values are pulled from registers and given back to the datapath.

```
io.regReadData1 := rf(io.regReadAddress1)
io.regReadData2 := rf(io.regReadAddress2)
```

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(SInt(32.W))
     val b = Input(SInt(32.W))
     val aluControl = Input(UInt(4.W))
     val out = Output(SInt(32.W))

     val zeroFlag = Output(Bool())
     val lessThanFlag = Output(Bool())
     val greaterThanFlag = Output(Bool())
 })
```

The signals a and b are the two values that are operated on while aluControl is the control signals. `out` is obviously the output of the calculations. `zeroFlag`, `lessThanFlag`, and `greaterThanFlag` are the control signals sent to the decoder to determine if a branch should occur. Like the decoder, the alu

uses when statements to determine what operation should be performed.

```
val aluMessage = Wire(new MessageAlu)

when (io.aluControl === 0.U) { //AND, ANDI
      io.out := io.a & io.b
}.elsewhen (io.aluControl === 1.U) { //OR, ORI
      io.out := io.a | io.b
}.elsewhen (io.aluControl === 2.U) { //ADD, ADDI, AUIPC, JALR
      io.out := io.a + io.b
}.elsewhen (io.aluControl === 3.U) { //SLL, SLLI
      io.out := io.a << io.b(11,0)
}.elsewhen (io.aluControl === 4.U) { //SRA, SRAI
      io.out := io.a >> io.b(11,0)
}.elsewhen(io.aluControl === 5.U){ //SLTU, SLTIU
      when(io.a.asUInt < io.b.asUInt){
         io.out := 1.S
      }.otherwise{
         io.out := 0.S
      }
}.elsewhen (io.aluControl === 6.U) { //XOR, XORI
      io.out := io.a ^ io.b
}.elsewhen (io.aluControl === 7.U) { //SRL, SRLI
      io.out := io.a >> io.b(11,0)
}.elsewhen(io.aluControl === 8.U) { //MUL (not in RV32I)
      io.out := io.a * io.b
}.elsewhen(io.aluControl === 9.U) { //SLT, SLTI
      when(io.a < io.b){
         io.out := 1.S
      }.otherwise{
         io.out := 0.S
      }
}.elsewhen(io.aluControl === 10.U){ //DIV (not in RV32I)
      io.out := io.a / io.b
}.elsewhen(io.aluControl === 12.U){ //SUB
      io.out := io.a - io.b
}.otherwise {
      io.out := 0.S
}
```

Next the control signals are set for branching.

```
io.zeroFlag := Mux(io.a + io.b === 0.S, true.B, false.B)
io.lessThanFlag := (io.a < io.b)
io.greaterThanFlag := (io.a > io.b)
```

3.7 Extend Module

The extend module is used for extending immediate values used in branching and the ALU.

```
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))
 })

 val extendMessage = Wire(new MessageExtend)

 when(io.immSrc === 0.U){
   io.extImm := io.instr12.asSInt
 }.elsewhen(io.immSrc === 1.U){
   io.extImm := (Cat(io.instr12, 0.U(1.W))).asSInt
 }.elsewhen(io.immSrc === 2.U){
   io.extImm := (Cat(io.instr20, 0.U(1.W))).asSInt
 }.otherwise {
   io.extImm := 0.S
 }
```

Some signals that are passed require a 0 to be concatenated on the end. `immSrc` is the control signal that determines this. Notice that by setting `io.extImm` to `io.instr12.asSInt`, Chisel will automatically sign extend `io.instr12.asSInt` to fit the size of `io.extImm`.

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 instAddress = Input(UInt(32.W))
   val inst = Output(UInt(32.W))
})

   val MEM = Mem(1024, UInt(32.W))
   loadMemoryFromFile(MEM, "tests/TESTFILE.X")

   io.inst := MEM(io.instAddress)

}
```

The instruction memory module uses the function `loadMemoryFromFile()} to load hexadecimal` Currently, the path to the test file is `tests/TESTFILE.X`. This is needed for the custom script (`runProject.sh`) that executes the required commands to simulate the processor. `runProject.sh` receives two inputs, the project that is being simulated (`riscvSingle` in this case) and the test to be run. An example shell command would be:`./runProject riscvSingle RV32I_test`. For main memory, the same `Mem` structure is used.

```
   class dmem extends Module {
    val io = IO(new Bundle {
       val memAddress = Input(UInt(32.W))
       val memWriteData = Input(SInt(32.W))
       val memWriteEnable = Input(UInt(1.W))
       val memReadData = Output(SInt(32.W))
    })

    val dmemMessage = Wire(new MessageDmem)
    val mem = Mem(1024, SInt(32.W))

    when(io.memWriteEnable.andR){
      mem.write(io.memAddress, io.memWriteData)
    }
```

```
    io.memReadData := mem(io.memAddress)
  }
```

`memWriteEnable` is the control signal that allows a value to be placed in memory if it is set to 1. When reading from memory there is no enable signal. A value for `memReadData` is always grabbed and sent back to the datapath. Control signals inside the datapath determine if `memReadData` is used, meaning that bogus values are ignored during that instruction

3.9 Top and Test Modules

The test class in this project is very different than other test classes found in beginner projects. How the Chisel template simulated projects is not straightforward and can be complicated to use. Since there is very limited documentation on how the Chisel template simulates it's projects, another method is used that directly uses the Scala build tool (sbt). This method works perfectly for this project and is much more straightforward. The test module is as follows:

```
class riscvSingleTest(t: top) extends PeekPokeTester(t) {
   println("**********STARTING riscvSingleTest*******")
   var cycles = 1
   var validP = peek(t.io.valid)

   println(s"Starting valid = $validP")
   println(s"CYCLE: $cycles")
   while (peek(t.io.valid) == BigInt(1) && cycles < 100) {
      println(s"STARTING NEXT CYCLE: $cycles")
      validP = peek(t.io.valid)
      println(s"valid = $validP")
      step(1)
      cycles += 1
   }
   step(1)
   if (cycles > 98 ) {
      println(s"$cycles cycles were ran and end of program not
          reached. Exiting.")
      System.exit(0)
```

```
  }
  else {
    println(s"test completed in $cycles cycles. Exiting.")
  }
}
```

This class extends `PeekPokeTester` so that signals can be printed to the console for debugging purposes. Two variables are needed for the test module, `cycles` and `validP`. A counter for the cycles is used for debugging and to keep the program from entering into infinite loops. `validP` is used to determine when the processor should stop stepping through the cycles. The while loop checks to see if `validP == 1` and that the cycles have not exceeded 100. This is an arbitrary value which can be changed with increasing test sizes. If an `ECALL` instruction is executed, then `validP` is set to 0. Inside of the loop, `validP` is peeked at again to get the updated value. The test then steps the processor one cycle and adds one to the cycle counter. After the loop, the if statement then prints out if the test completed or exceeded the cycle count. Lastly, a top object is required to initialize the test class and start the simulation.

```
object top extends App {
  iotesters.Driver.execute(args, () => new top) {
    t => new riscvSingleTest(t)
  }
    chisel3.Driver.execute(args, () => new top)
}
```

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
```

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

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 a cache
2. Adding a floating point module
3. Pipelining datapath

3.12 Conclusion

Chisel has many advantages over common HDL's like verilog and VHDL. The main advantage is the ability to use an object oriented style of programming. Since this was a relatively small project, the full potential of Chisel has not been realized. In future iterations of this project, after adding in more functionality, the quality of life factor in designing hardware could increase considerably.