154B Discussion 1

January 11th, 2023

Goals

- Logistics.
- Homework Overview.
- Chisel Overview.
- Assignment 1 Notes.

General Information

- Discussions: Wednesdays, 9:00am - 9:50am, Hutchinson Hall 115.
  - Usually 35 mins for discussing homework and 15 mins for discussing quizzes.
- Office Hours: Fridays, 10:30am - 11:50am, Kemper 3106.
- Homework questions -> Office Hours / Piazza.
  - In most cases, should use public questions, which help others.
  - Private questions: questions that exposes parts of your answer.
- Submissions:
  - All submissions are via Gradescope, unless specified otherwise.
  - Late submissions (upto 48 hours) are allowed, with penalty.
  - Solutions are only released after the late due date.

Collaboration Policy

- Unlike in 201A, the homework assignments in 154B must be done *individually*.
- You're encouraged to discuss high-level details with other students.

Homework Assignments

- Homework 1 & 2: single cycle CPU in Chisel.
- Homework 3: five-stage pipelined CPU in Chisel.
- Homework 4: TBD.
- Homework 5: multi-core system in gem5.

Resources for Codespace, Chisel, and RISC-V

- Setting up development environment: either via Codespace or CSIF.
  - See assignment 1: https://jlpteaching.github.io/comparch/modules/dino%20cpu/assignment1/#tools
- RISC-V ISA spec Volume 1: https://riscv.org/technical/specifications/
  - Chapter 2, 5, 7.
  - Quick RISC-V specs: http://riscvbook.com/
  - We are implementing RV64IM.
- Chisel docs: https://www.chisel-lang.org/chisel3/docs/introduction.html
- Chisel API: https://www.chisel-lang.org/api/3.3.3/Chisel/index.html
  - Caution: we are using Chisel 3.3.3.
- Searching for chisel syntax: “chisel 3.3.3 <syntax>” or “chisel lang <syntax>”.

Chisel Basics

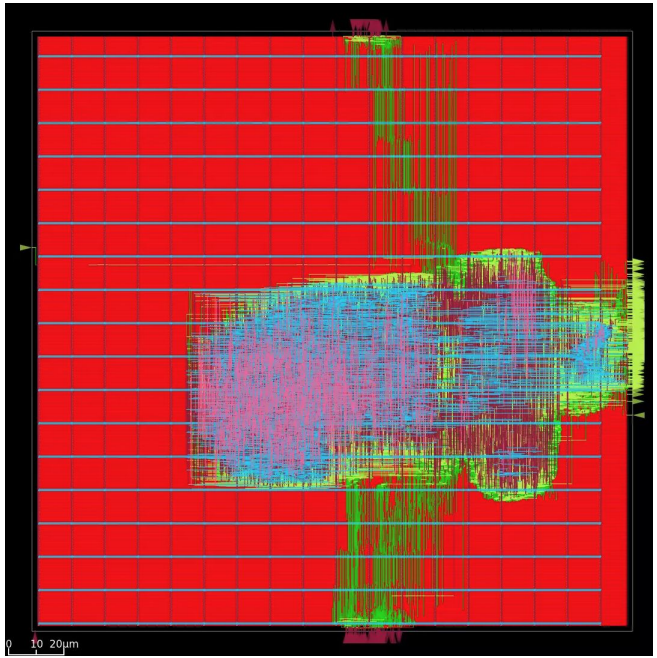
- https://github.com/ECS154B-WQ23/dinocpu-assignment1/blob/main/documentation/chisel-notes/cheat-sheet.md
- https://github.com/freechipsproject/chisel-cheatsheet/releases/latest/download/chisel_cheatsheet.pdf

Chisel

- Hardware description language (HDL).
  - Old HDLs: Verilog, VHDL.
  - The code constructs wires and registers.
  - No physical placement.
- Built on top of Scala, a functional language.
  - Circuits naturally have a tree structure.

What Chisel does not do

- Physical placement of wires and registers.
- On the right, a physical placement for the dual-issue pipelined DINOCPU using 7nm technology.



Why Chisel?

- Has the power of a modern high-level programming language.
  - Encourage software engineering approaches to design digital circuits.
  - Getting compiler optimizations for optimizing circuits, for free.
  - Parameterized circuits/components (e.g., an adder of 5 elements and of 10 elements might share the same codebase).
  - Bug findings, lint tools.
  - Libraries with pre-built hardware interfaces/components.
- Easier (than e.g. Verilog) to comprehend from a software engineer's perspective.
  - Old HDLs require domain knowledge.
  - Chisel is still very involved to learn, but a learning curve is not as steep.
- The industry uses it.
  - Google Edge TPU

What Chisel does?

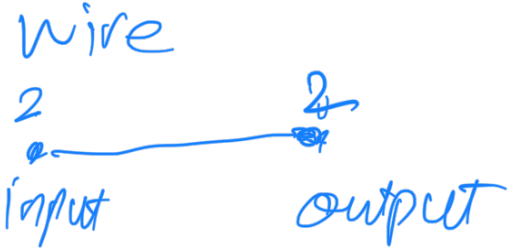
- In short, Chisel -> FIRRTL_IR -> Verilog.
- When you compile Chisel code,
  - The code is compiled to intermediate representations (IRs) in FIRRTL format by the Scala compiler.
  - FIRRTL compiler transform the IRs into a circuit in Verilog. A lot of circuit optimizations take place in this step.
- Make sure you make the circuit work correctly first before doing any optimization.

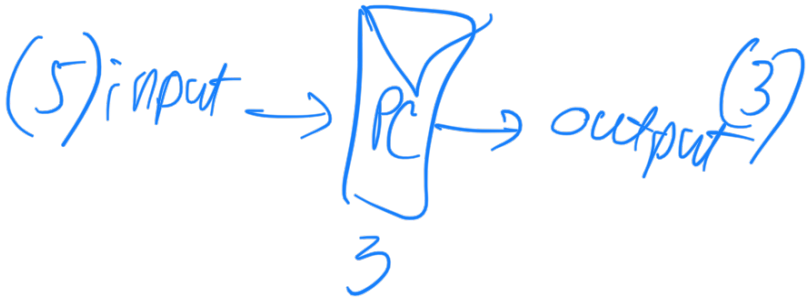
Circuit as a tree structure

- A circuit typically can be represented in a tree form.
- Each component (e.g., wires, registers) must be resolved (or settled in EE language) to exact one possible value at the end of each cycle.
- The code in Chisel won't be evaluated sequentially. The simulator resolved the circuit in a tree form.
- This is important to understand, as this concept appears in a very frequent error: loops in combinational circuits.
    - Root cause: a collection of wires form a cycle, the value of a wire cannot be settled at the end of a cycle.

Wires vs Registers

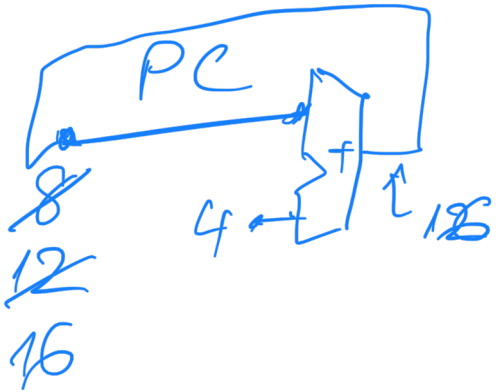
- Both ends of a wire must be the same for each cycle.
  - ALU is essentially a bunch of wires.
  - In other words, changing the value of one end of a wire affects the other end of the wire.
- A register, in contrast, does not have its output affected by its input until the end of a cycle.
  - E.g., if a register holding a value 3, and it is taking an input of 5, then the output is still 3 in the current cycle. The register will have the value of 5 in the next cycle.





Wires vs Registers: Why PC is not a wire?

- In every cycle, we want to update the PC, e.g., PC := PC + 4.
- If PC is a wire,
  - Suppose at the beginning of the cycle, PC = 8.
  - Then PC is updated to 12 in the same cycle.
  - Then PC is updated to 16 in the same cycle.
  - Then PC is updated to 20 in the same cycle.
  - Then PC is updated to 24 in the same cycle.
  - Then PC is updated to 28 in the same cycle.
  - Then PC is updated to 32 in the same cycle.
  - etc.
  - There is not a **single** value that describes the state of the wire PC at the end of the cycle.



Chisel types vs Scala types

- Chisel types represent hardware components.
- It is very unlikely that you will have to use Scala types for the assignments.
- Don’t mix Chisel and Scala types.

Chisel Types

- A Chisel type either represents a collection of bits in hardware, or represents a hardware component.

Chisel Semantics

- Wire: an object of a specific type that can be connected to a port (io) of a hardware component, or another wire of the same width.

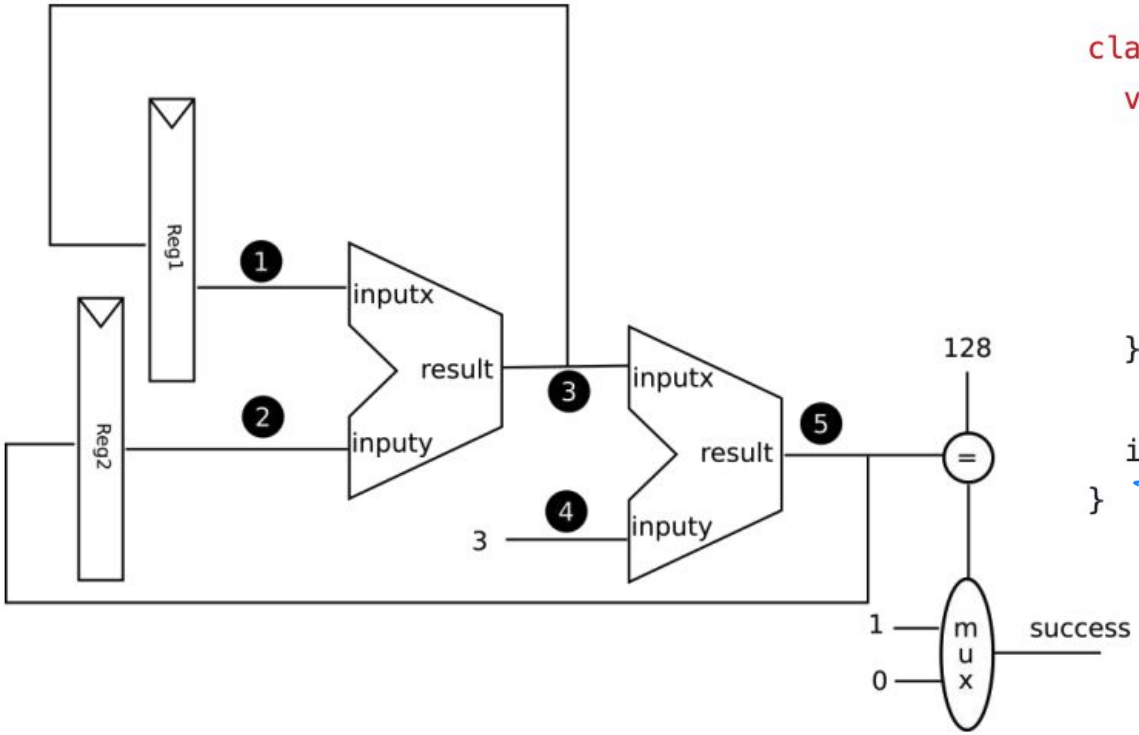
```
val w = Wire(UInt(32.W)) // making a wire of 32 bits wide
```

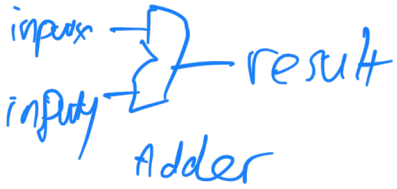
- In most cases, you don't need to declare a variable explicitly for a wire.

```
control.io.opcode := imem.io.instruction(6, 0)
```

- `:=` can be used to connect a wire and a port, or to connect two ports, or to connect to wires.

Example



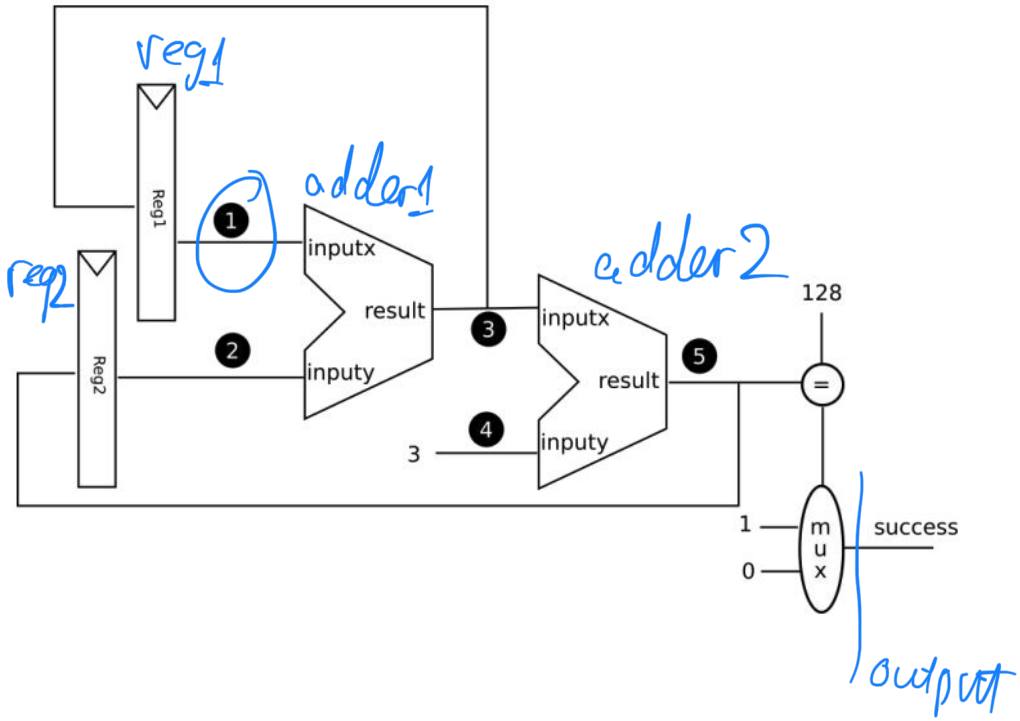



```
class Adder extends Module {
  val io = IO(new Bundle{
    val inputx    = Input(UInt(64.W))
    val inputy    = Input(UInt(64.W))

    val result    = Output(UInt(64.W))
  })

  io.result := io.inputx + io.inputy
}
```

Example




```
class SimpleSystem extends Module {
  val io = IO(new Bundle {
    val success = Output(Bool())
  })

  val adder1 = Module(new SimpleAdder())
  val adder2 = Module(new SimpleAdder())

  val reg1 = RegInit(0.U)
  val reg2 = RegInit(1.U)

  adder1.io.inputx := reg1 // (1)
  adder1.io.inputy := reg2 // (2)

  adder2.io.inputx := adder1.io.result // (3)
  adder2.io.inputy := 3.U // (4)

  reg1 := adder1.io.result // (3)

  reg2 := adder2.io.result // (5)

  when(adder2.io.result === 128.U) {
    io.success := true.B
  } .otherwise {
    io.success := false.B
  }
}
```

Chisel Semantics

77.U (64.W)
77.S
77.U → 64 bits wide

- Integers

```
UInt(32.W)   // Chisel 32-bit unsigned integer.

UInt()       // Chisel unsigned integer where the width is inferred.

SInt()       // Chisel signed integer where the width is inferred.

77.U         // Chisel unsigned integer (width inferred) with the value of 77.

(-7).S(16.W) // Chisel 16-bit signed integer with the value corresponding to -7 in
16-bit signed integer interpretation.

“b010111”.U  // converting a string of binary into the Chisel unsigned integer type.

Wire(UInt(32.W)) // a wire that is 32-bit wide interpreted as an unsigned integers.
```

Chisel Semantics

- Arithmetics, e.g.,

```
val x = Wire(UInt(3.W))

val y = x + 2.U // addition

val z = x - 2.U // subtraction

val t = x >> 2.U // right shift
```

- Comparisons, e.g.,

```
x === y // equality between x and y

x =/= 3.U // inequality between x and an unsigned integer type
```

Chisel Semantics

- Setting/Getting values of registers

```
val x = Reg(UInt(64.W))

val z = Wire(UInt(64.W))

x := z //  “assigning” value of the wire z to the register x

       // i.e.,  z is wired to input of x, value of x changes at the end of cycle

z := x // “assigning” value of the wire z to the register x

       // i.e., . z is wired to output of x, value of z is the value coming out of x
```

Chisel Semantics

cycle 0
x = 3
y = 3

cycle 1
x = 5
y = 5

- Note about registers: the syntax represents hardware constructions, not software.

```
val x = RegInit(3.U(32.W))
val y = Wire(32.W)
x := 5.U // update x to 5 in the next cycle
y := x // assign the value of x in current cycle to y
       // in this cycle, y has the value 3
```

using input of x

using output of x

Chisel Semantics

- Getting a subset of bits from a wire/register.

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

x := 30.U // x = “b11101”.U

val y = x(2,0) // y = “b101”.U, i.e. bit 2 to bit 0 of x
```

x(3) ← getting bit 3 of x

Chisel Semantics

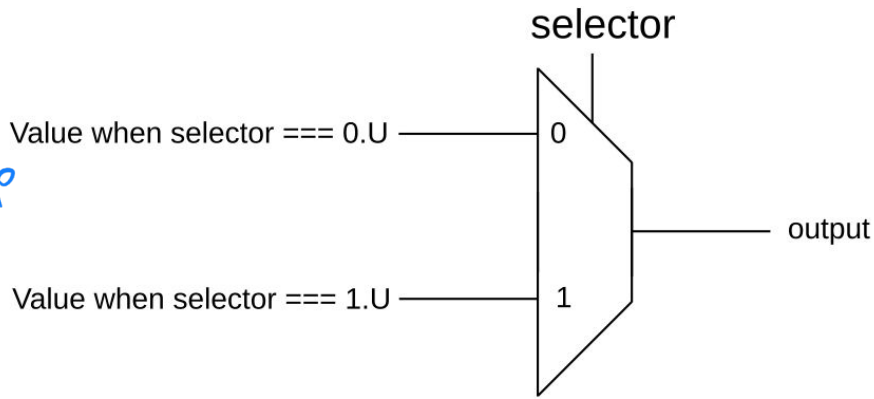
- Branching: using when/elsewhen/otherwise, or Mux, or MuxCases are the same.

```
val x := Mux(selector, true_value, false_value)
```

equals to,

```
when (selector === true.B) { x:= true_value }
.otherwise { x := false_value }
```

when / . else when / . otherwise



Chisel Semantics

- MuxCase is for a more complicated mux cases.
  - Expressive way to express complex conditions.
  - Remember: we want the code to be clear as optimization will take place in other phases.

```
val mux = MuxCase(5.U, Array(
    ((io.funct3 === “b010”.U) & (io.funct7 === “b0000111”.U)) => io.output := “b01010”.U,
    ((io.funct3 === “b011”.U) & (io.funct7 === “b0000111”.U)) => io.output := “b01011”.U,
))
// If none of the conditions matches, the mux outputs the default value 5.U.
```

io.output := mux => Cat(io.input1, io.input2)

Chisel Semantics

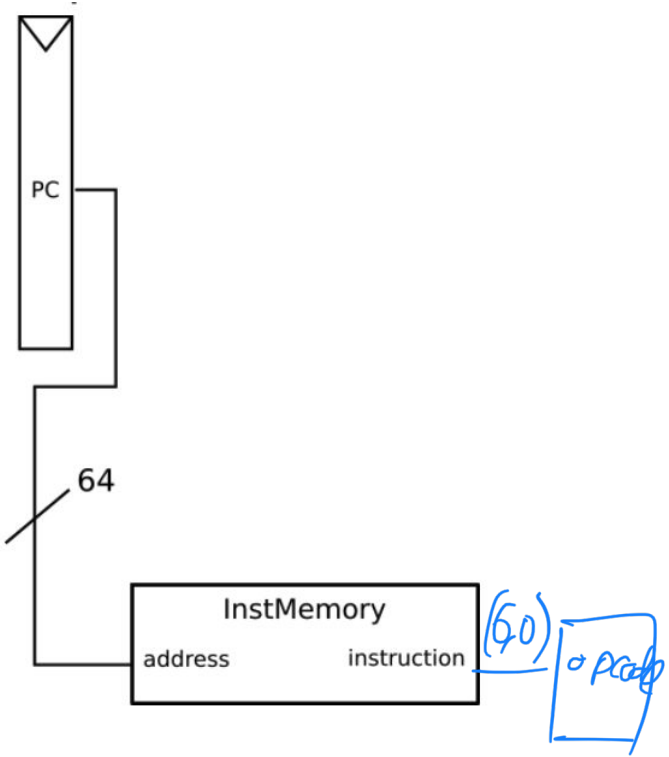
- We are writing hardware, so you don’t need to worry about the performance of a branch as in writing software.

Notes

- For all assignments,
  - RISC-V instructions are 32 bits wide.
  - Registers are 64 bits wide.
  - Addresses are 64 bits wide.

Assignment 1 Notes

- Width of each wire must be specified.
- Indicate that if a wire uses a subset of bits from the input.



Assignment 1 Notes: diagram

- Each component has its inputs and outputs described in its scala file.
- You don’t need to modify/wire from the Control Unit.

Assignment 1 Notes: Testing

- There's a command for testing at the end of each part.
  - E.g., part 1: `Lab1 / testOnly dinocpu.SingleCycleAddTesterLab1`
- The RISC-V binaries and their assembly can be found at,
  - https://github.com/ECS154B-WQ23/dinocpu-assignment1/tree/main/src/test/resources/risc-v
- Inputs and expected outputs,
  - https://github.com/ECS154B-WQ23/dinocpu-assignment1/blob/main/src/main/scala/testing/InstTests.scala

Assignment 1 Notes: Debugging

- Using printf statements,
  - https://github.com/ECS154B-WQ23/dinocpu-assignment1/blob/main/documentation/chisel-notes/printf-debugging.md
- Using the dinocpu.singlestep debugger,
  - https://github.com/ECS154B-WQ23/dinocpu-assignment1/blob/main/documentation/single-stepping.md