Experiment 3

Control Flow and Combinational Circuits

Objective

To be able to perform] analysis and design of complex combinational circuits while using different control flow constructs.

3.1 Control Flow

The control flow is the order in which individual modules are enabled or selected to perform their respective operations. Control flow blocks allow us to enable a hardware block conditionally. Hardware control flow can be realized through different conditional constructs, which perform similar job. Chisel provides two conditional constructs, namely `when` block and `switch` block. Nesting of these conditional blocks is possible.

3.1.1 When, Elsewhen, Otherwise Construct

Similar to `if`, `else` in verilog, we can control the data flow by using `when`, `.elsewhen`, `.otherwise`. In Chisel, the `if`/`else` block is treated as a Scala conditional construct, which includes or excludes a hardware block in the generated hardware (i.e. in the emitted verilog). On the other hand, the working of `when` construct in Chisel is similar to `if`/`else` in verilog. The argument to `when` is a conditional expression that returns a `Bool`. An incompletely specified combinational output results in an error. One such possible scenario is when an unconditional update is not provided for a combinational output.

The `when` block is translated to Mux circuit, where the selection lines for Mux are dependent on the conditions used by `when` or `.elsewhen`.

```
when(/* condition for when */){
    /* do this if the condition for when is true */
}
.elsewhen(/* condition for elsewhen */){
    /* do this if the when is false and
    condition for .elsewhen is true. */
}
.otherwise{
    /* do this as a default condition, .otherwise
    do not require a condition as it is the default
    when no previous condition is met. */
```

```
}
```

The advantage of using .otherwise block is that we can omit the initialization, which other wise is required to avoid initialization error. For example, if we want to implement a decoder as shown in Figure 3.1 using when conditional construct, a possible implementation is illustrated in Listing 3.1.

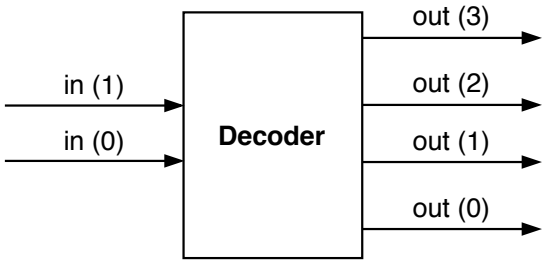

Figure 3.1: A simple 2 to 4 decoder.

```
//Example 2 to 4 decoder
import chisel3._
class LM_IO_Interface extends Bundle {
    val in  = Input(UInt(2.W))
    val out = Output(UInt(4.W))
}
class Decoder_2to4 extends Module {
    val io = IO(new LM_IO_Interface)

    when(io.in === "b00".U) {
        io.out := "b0001".U
    } .elsewhen(io.in === "b01".U) {
        io.out := "b0010".U
    } .elsewhen(io.in === "b10".U) {
        io.out := "b0100".U
    } .otherwise {
        io.out := "b1000".U
    }

}

println((new chisel3.stage.ChiselStage).emitVerilog(new Decoder_2to4()))
```

Listing 3.1: Decoder using when construct.

Similarly, it is also possible to implement an encoder, like the one shown in Figure 3.2, using when construct. Listing 3.2 illustrates the Chisel implementation for the encoder.

```
class EncoderIO extends Bundle {
    val in  = Input(UInt(4.W))
    val out = Output(UInt(2.W))
}
```

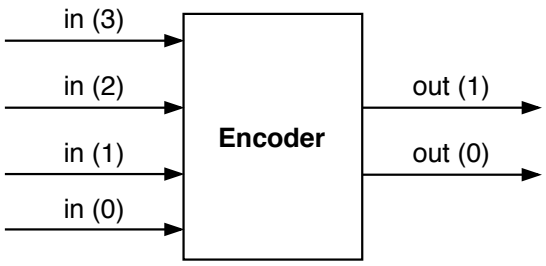

Figure 3.2: A simple 4 to 2 encoder.

```
class Encoder4to2 extends Module {
    val io = IO(new EncoderIO)

    when (io.in === "b0001".U) {
        io.out := "b00".U
    } .elsewhen(io.in === "b0010".U) {
        io.out := "b01".U
    } .elsewhen(io.in === "b0100".U) {
        io.out := "b10".U
    } .otherwise {
        io.out := "b11".U
    }
}
```

Listing 3.2: Encoder implementation using `when` construct.

3.1.2 Switch

The `switch` construct in Chisel is similar to a `case` statement in Verilog. However, there is one key difference between `switch` and `case` syntax. The `switch` construct does not have the `default` case. Due to the absence of default case, we might run into an initialization errors. In addition, `is` keyword is used to mark different cases for the `switch` construct. Chisel implementation for 2 to 4 decoder using `switch` construct is illustrated in Listing 3.3.

```
class DecoderIO extends Bundle {
    val in  = Input(UInt(2.W))
    val out = Output(UInt(4.W))
}

class Decoder2to4 extends Module {
    val io = IO(new DecoderIO)
    io.out := 0.U
    switch (io.in) {
        is ("b00".U) {
            io.out := "b0001".U
        }
        is ("b01".U) {
            io.out := "b0010".U
```

```
        }
        is ("b10".U) {
            io.out := "b0100".U
        }
        is ("b11".U) {
            io.out := "b1000".U
        }
    }
}
```

Listing 3.3: Decoder using `switch` construct.

3.2 The ALU Module

The Arithmetic Logic Unit (ALU) is one of the core building blocks of a microprocessor. We will develop an ALU that will perform all the operations required by the base instruction set architecture, RV32I [**?**]. An ALU requires two operands and produces the result based on the operation selected. The block diagram, illustrating the ALU interfaces is shown in Figure 3.3.

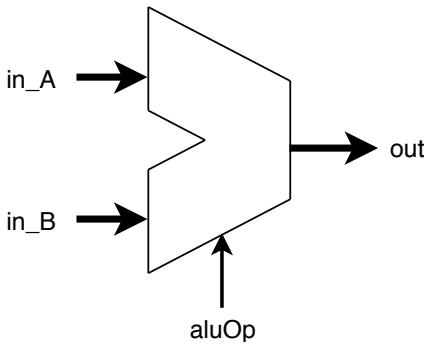

Figure 3.3: ALU block diagram representation.

3.2.1 ALU Operations

Different operations that will be performed by the ALU are grouped in the form of an `object`. A Scala `object` is used to define all the operations supported by the ALU. Listing 3.4 defines the ALUOP object for this purpose.

```
object ALUOP {
    // ALU Operations, may expand/modify in future
    val ALU_ADD     = 0.U(4.W)
    val ALU_SUB     = 1.U(4.W)
    val ALU_AND     = 2.U(4.W)
    val ALU_OR      = 3.U(4.W)
    val ALU_XOR     = 4.U(4.W)
    val ALU_SLT     = 5.U(4.W)
    val ALU_SLL     = 6.U(4.W)
    val ALU_SLTU    = 7.U(4.W)
    val ALU_SRL     = 8.U(4.W)
    val ALU_SRA     = 9.U(4.W)
```

```
    val ALU_COPY_A = 10.U(4.W)
    val ALU_COPY_B = 11.U(4.W)
    val ALU_XXX    = 15.U(4.W)
}
```

Listing 3.4: Alu operations defined as an object.

3.2.2 ALU Parameterization

Parameterization is one of the highly acclaimed features of Chisel based hardware design, providing flexibility to the design. There are many different ways to parameterize a design as we will learn in Exp 5. To parameterize the ALU design, we have chosen trait, which is a fundamental code reuse block in Scala. We will learn more about traits in Exp 13

We define trait Config, as provided by Listing 3.5, for ALU parameterization. This is a rather simple trait, which only specifies the word length and the control signal width. We use this trait by extending ALUIO bundle as well as ALU module with Config (see Listing 3.6).

```
trait Config {
    // word length configuration parameter
    val WLEN           = 32

    // ALU operation control signal width
    val ALUOP_SIG_LEN  = 4
}
```

Listing 3.5: Config trait.

It is important to notice that Listing 3.6 starts with `import` ALUOP._ despite the fact that object ALUOP is defined in the same file. Another aspect is the use of Scala wildcard "_". In the current scope, it simply means that include all the objects and classes implemented in the ALUOP package (which is object ALUOP only in this case). Scala wildcard will be discussed in detail in Exp 12.

```
import ALUOP._

class ALUIO extends Bundle with Config {
    val in_A       = Input(UInt(WLEN.W))
    val in_B       = Input(UInt(WLEN.W))
    val alu_Op     = Input(UInt(ALUOP_SIG_LEN.W))
    val out        = Output(UInt(WLEN.W))
    val sum        = Output(UInt(WLEN.W))
}

class ALU extends Module with Config {
    val io = IO(new ALUIO)

    val sum    = io.in_A + Mux(io.alu_Op(0), -io.in_B, io.in_B)
    val cmp    = Mux(io.in_A (XLEN-1) === io.in_B(XLEN-1), sum(XLEN-1),
    Mux(io.alu_Op(1), io.in_B(XLEN-1), io.in_A(XLEN-1)))
```

```
    val shamt  = io.in_B(4,0).asUInt
    val shin   = Mux(io.alu_Op(3), io.in_A, Reverse(io.in_A))
    val shiftr = (Cat(io.alu_Op(0) && shin(XLEN-1), shin).asSInt >> shamt)(
      XLEN-1, 0)
    val shiftl = Reverse(shiftr)

    val out =
    Mux(io.alu_Op === ALU_ADD.U || io.alu_Op === ALU_SUB.U, sum,
    Mux(io.alu_Op === ALU_SLT.U || io.alu_Op === ALU_SLTU.U, cmp,
    Mux(io.alu_Op === ALU_SRA.U || io.alu_Op === ALU_SRL.U, shiftr,
    Mux(io.alu_Op === ALU_SLL.U, shiftl,
    Mux(io.alu_Op === ALU_AND.U, (io.in_A & io.in_B),
    Mux(io.alu_Op === ALU_OR.U,  (io.in_A | io.in_B),
    Mux(io.alu_Op === ALU_XOR.U, (io.in_A ^ io.in_B),
    Mux(io.alu_Op === ALU_COPY_A.U, io.in_A,
    Mux(io.alu_Op === ALU_COPY_A.U, io.in_B, 0.U)))))))))

    io.out := out
    io.sum := sum
}
```

Listing 3.6: ALU implementation.

3.3 Interfaces

Chisel provides us a standard library of interfaces (facilitating interoperability of RTL) and generators for commonly used hardware blocks.

3.3.1 Valid Interface

Valid is a standard interface provided by the Chisel library. Specifically Valid is a bundle that adds valid bit to the data as shown in Figure 3.4. When valid data is put on data lines (bits) by the producer, it asserts valid bit. However, producer does not wait for the consumer in valid interface. This helps other devices to be synchronized.

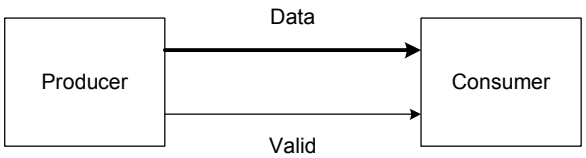

Figure 3.4: Valid signal for producer and consumer model.

Listing 3.7 shows the process of adding valid signal to the data bits, while an example illustrating the use of Valid interface is provided in Listing 3.8.

```
// data bits without valid signal
class DataWithoutValid extends Bundle {
  val data_bits = Output(UInt(8.W))
}
```

```
// data bits with valid signal
val DataWithValid = Valid(new DataWithoutValid)
```

Listing 3.7: Adding Valid interface to data.

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

class Valid_Interf extends Module{
   val io  = IO(new Bundle {
      val in  = Flipped(Valid(UInt(8.W))) //valid = Input,   bits = Input
      val out = Valid(UInt(8.W))          //valid = Output,  bits = Output
   })
   io.out := RegNext(io.in)
}
println(chisel3.Driver.emitVerilog(new Valid_Interf))
```

Listing 3.8: Valid interface illustration.

3.3.2 Decoupled Interface

Chisel provides some built-in standard interfaces that should be used whenever possible for interoperability. Decoupled is one of those standard interfaces of Bundle type, which augments ready-valid signaling to the data bits for handshaking. The ready valid signaling used by the Decoupled interface is illustrated pictorially in Figure 3.5. For decoupled IO, the producer uses the interface as is, while the consumer uses flipped interface.

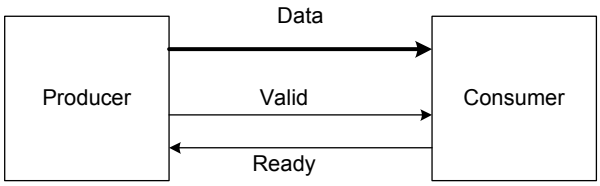

Figure 3.5: Ready valid signaling for producer consumer model used by Decoupled.

Any data type that is a subclass of the class 'Data' can be passed to the Decoupled constructor. The `object` Decoupled adds a ready-valid handshaking protocol to the data bundle. Signal 'ready' is used to show that this device is ready for communication if its high, while 'valid' is used to indicate that data 'bits' are valid. It is important to mention that no signaling requirements are imposed on ready and valid. The `apply` method of Decoupled adds the handshaking protocol using the DecoupledIO, while DecoupledIO is a subclass of ReadyValidIO signaling. Listing 3.9 illustrates the ReadyValidIO signaling. The use of type parameters (`[T]` and its variants) will be explained in Experiment 5.

```
// DecoupledIO class definition
class DecoupledIO[+T <: Data] extends ReadyValidIO[T]
```

```
class ReadyValidIO[T <: Data](gen: T) extends Bundle {
    val ready = Input(Bool())
    val valid = Output(Bool())
    val bits = Output(gen)
}
```

Listing 3.9: Ready valid interface illustration.

3.4 Exercises

Exercise 1: Refer to Listing 3.3, implement 4 to 2 encoder using switch-is statements.

Exercise 2: Write Chisel code for a standard RISCV ALU using switch, is statements.

3.5 Assignments

Task 1: In a standard RISC-V ALU whenever we have a branch instruction, the ALU computes it and tells the PC whether or not to jump to the immediate value. Implement the conditional branch module of a standard RV32I using combinational circuits and control constructs. A skeleton code is given in Listing 3.10, start coding in it. Block level diagram of conditional branch module is shown in Figure 3.6.

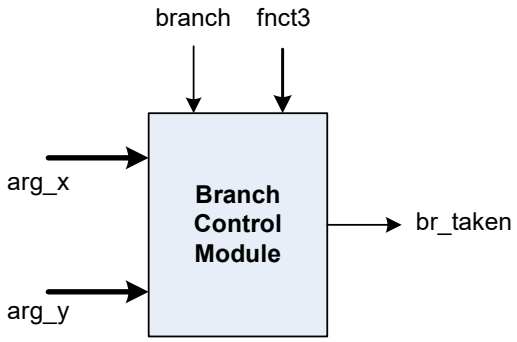

Figure 3.6: Branch control of RV32I.

```
// Branch control (Assignment)
package lab3
import chisel3._
import chisel3.util._

class LM_IO_Interface_BranchControl extends Bundle {
    val fnct3     = Input(UInt(3.W))
    val branch    = Input(Bool())
    val arg_x     = Input(UInt(32.W))
    val arg_y     = Input(UInt(32.W))
    val br_taken  = Output(Bool())
}
```

```
class BranchControl extends Module {
    val io = IO(new LM_IO_Interface_BranchControl)
    // Start Coding here


    // End your code here
    // Well, you can actually write classes too. So, technically you have no
    limit ; )
}

println((new chisel3.stage.ChiselStage).emitVerilog(new Branch_Control))
```

Listing 3.10: Skeleton code for Branch control of RV32I.

Task 2: Immediate extension is a key part of decode stage and needs to provide sign extended immediate to the execute stage. Implement an RV32I standard immediate extension module using skeleton code available in Listing 3.11. Block level diagram of immediate extension is shown in Figure 3.7.

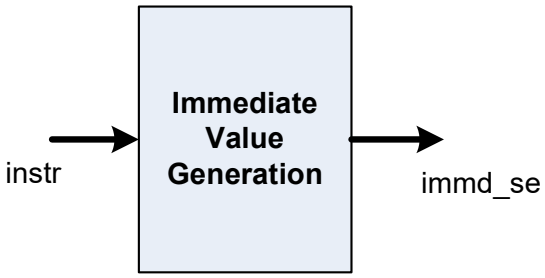

Figure 3.7: Immediate extension module of RV32I.

```
// Immediate (Assignment)
package lab3
import chisel3._
import chisel3.util._

class LM_IO_Interface_ImmdValGen extends Bundle {
    val instr = Input(UInt(32.W))
    val immd_se = Output(UInt(32.W))
}

class ImmdValGen extends Module {
    val io = IO(new LM_IO_Interface_ImmdValGen)

    // Start coding here

    // End your code here
    // Well, you can actually write classes too. So, technically you have no
    limit ; )
}
```

Listing 3.11: Skeleton code for Immediate extension module of RV32I.

Task 3: To understand the concept of Valid interface, write a code of 2 to 4 Decoder and wrap the output in Valid construct. Skeleton code for this task is available in Listing 3.12. Block level diagram of immediate extension is shown in Figure 3.8.

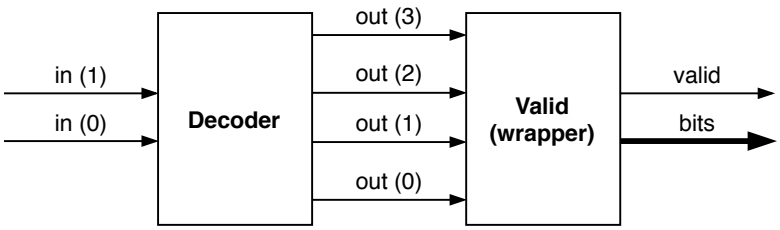

Figure 3.8: Decoder with output wrapped in Valid interface.

```
package lab3
import chisel3._
import chisel3.util._

class LM_IO_Interface_decoder_with_valid extends Bundle {
    val in  = Input(UInt(2.W))
    val out = Valid(Output(UInt(4.W)))
}

class decoder_with_valid extends Module {
    val io = IO(new LM_IO_Interface_decoder_with_valid)

    // Start coding here

    // End coding here
}
```

Listing 3.12: Skeleton code for Decoder with output wrapped in Valid interface.

Task 4: In Listing 3.6, we see that there are two outputs of ALU, io.out and io.sum. When opcode is for add, we see that both io.out and io.sum are equal. What might be the use case of these two different outputs?