Chisel: Control Flow & Combinational Circuits

Muhammad Tahir*

Lecture 3

*Professor, Electrical Engineering Department, University of Engineering and Technology Lahore

Contents

Conditional Constructs

- Used to enable a hardware block conditionally
- Multiple conditional constructs do the similar job
- Chisel provides `when` block and `switch` block
- Nesting of conditional blocks is possible

Encoder Decoder Block Diagram

Illustration of `when` and `switch` using encoder and decoder modules

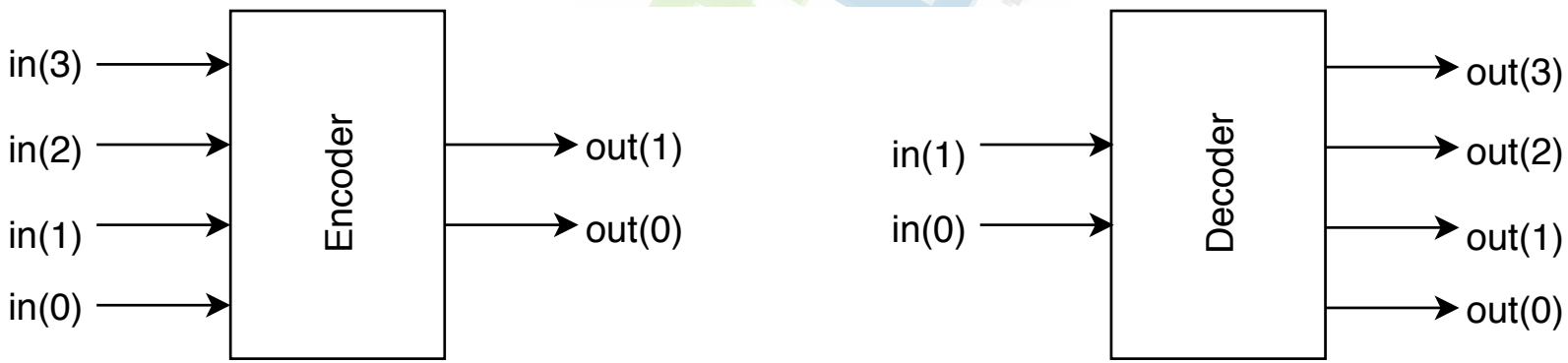

Figure: Encoder (4 to 2) and decoder (2 to 4).

Chisel: when Syntax

```
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. */
}
```

Chisel: when

Implementing 4 to 2 Encoder with when

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

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
    }
}
```

Chisel: `switch` Syntax

```
switch (state) {
    is(state1){
        // logic implementation which runs when state ===
        state1
    }
    is(state2){
        // logic implementation which runs when state ===
        state1
    }
    // what about the default state
}
```

Chisel: switch

```
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
          }
     }
}
```

ALU Module

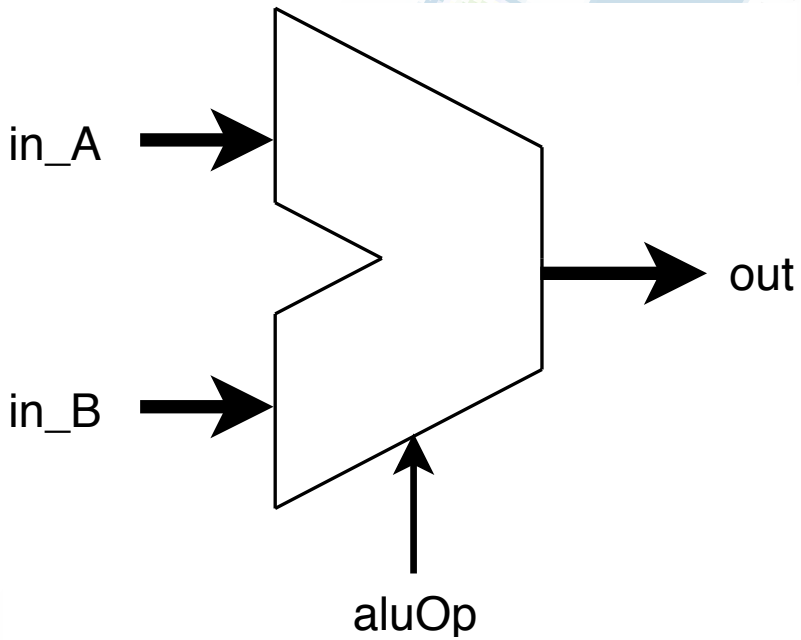

Figure: ALU block diagram representation.

ALU Operations

Define ALU operations using Scala `object`

```
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 1: ALU operations.

ALU Implementation: MuxLookup

```
import ALUOP._

trait Config {
    // global config parameters
    val WLEN                = 32
    val XLEN                = 32
    val ALUOP_SIG_LEN       = 4   // control signal's width
}

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))
}
```

ALU Implementation: MuxLookup Cont'd

```
class Alu_Lookup extends Module with Config {
    val io = IO(new AluIO)

    val shamt = io.in_B(4,0).asUInt

    io.out := MuxLookup(io.alu_Op, io.in_B, Seq(
    ALU_ADD  -> (io.in_A + io.in_B),
    ALU_SUB  -> (io.in_A - io.in_B),
    ALU_SRA  -> (io.in_A .asSInt >> shamt).asUInt,
    ALU_SRL  -> (io.in_A >> shamt),
    ALU_SLL  -> (io.in_A << shamt),
    ALU_SLT  -> (io.in_A .asSInt < io.in_B.asSInt),
    ALU_SLTU -> (io.in_A < io.in_B),
    ALU_AND  -> (io.in_A & io.in_B),
    ALU_OR   -> (io.in_A | io.in_B),
    ALU_XOR  -> (io.in_A ^ io.in_B),
    ALU_COPY_A -> io.in_A ))

}
```

ALU Implementation: Optimized

```
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
}
```

Comparing ALU Implementations

Table: ALU resource usage comparison for different implementations.

Parameter	MuxLookUp Based	Optimized
Slice LUTs as Logic	440	230
F7 Muxes	28	0
Bonded IOB Pads	132	132
Primitives, LUT2, LUT3, LUT4	52, 76, 75	2, 79, 3
Primitives, LUT5, LUT6	45, 252	53, 126
Primitives, CARRY4	32	8

Valid Interface

- A Bundle which adds valid bit to the data
- When valid data is put on data lines (bits) by the producer, it asserts valid bit
- Producer does not wait for the consumer

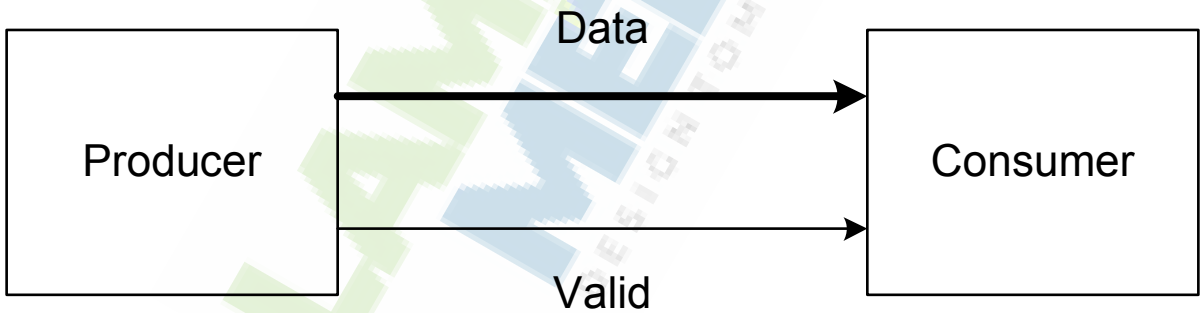

Figure: Valid signal for producer and consumer model.

Valid Interface Cont'd

- `object Valid` acts as factory to generate `Valid` interfaces
- Wraps the data by calling the companion object of `class Valid`

```
// 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)
```

$\Longleftrightarrow$

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

class DataWithValid extends Bundle {
  val valid   = Output(Bool())
  val data    = Output(new
        DataWithoutValid)
}
```

Ready Valid Interface

- A bundle providing *valid* and *ready* signals for handshaking
- Producer uses the interface as is
- The consumer uses flipped interface

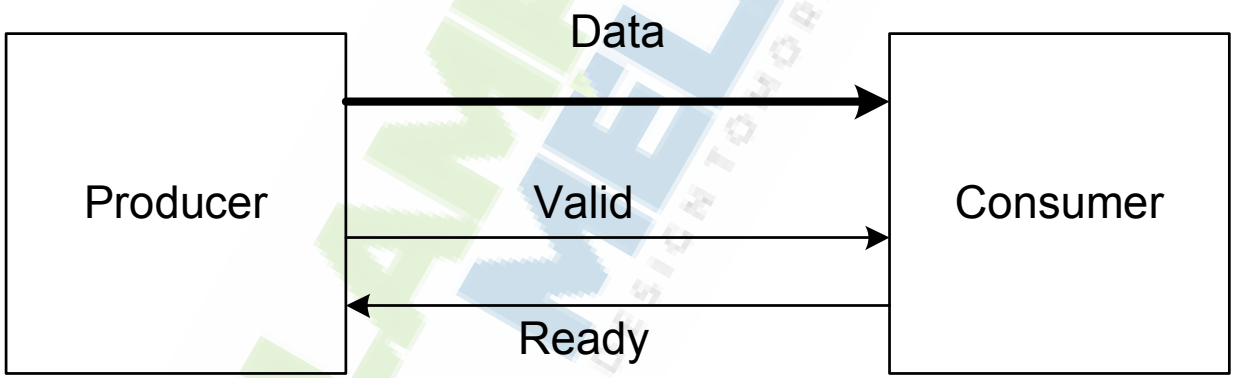

Figure: Ready valid signaling for producer and consumer model.

Ready Valid Interface

- *valid* indicates that the producer has put valid data on data lines/bits
- *ready* indicates that the consumer is ready to accept the data this cycle
- No signaling requirements are imposed on ready and valid

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

DecoupledIO and Decoupled

- DecoupledIO is a subclass of ReadyValidIO signaling

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

- The `object` Decoupled wraps/adds a ready-valid handshaking protocol to the data bundle
- The `apply` method of Decoupled adds the handshaking protocol using the DecoupledIO

Reading List

- Read the relevant sections from Chapters 2 and 4 of [Schoeberl, 2019]
- Also consult the [chisel3, 2020] for further details
- For type parameters and associated variance read Sections 19.3 to 19.6 of [Odersky et al., 2016]

References

chisel3 (2020).
Chisel3 library reference.
https://www.chisel-lang.org.

Odersky, M., Spoon, L., and Venners, B. (2016).
Programming in Scala.
Artima Incorporation.

Schoeberl, M. (2019).
Digital Design with Chisel.
Kindle Direct Publishing.