Experiment 2

Combinational Circuits

Objective

Familiarize with basic building blocks to design and implement combinational hardware, specifically with Mux and its variants.

2.1 Chisel: Hardware Operations

First, let us take a look at a list of operators that can be used to modify any piece of hardware. Table 2.1 categorizes such operators in different groups and also mentions the type of operands that can be operated on.

Table 2.1: Different groups of hardware operators.

Operator Symbol	Description	Operand Type
&& || !	AND, OR, NOT (logical)	Bool
& | ~ ^	AND, OR, NOT, XOR (bitwise)	UInt, SInt, Bool
<< >>	shift left, shift right (sign extend for SInt)	UInt, SInt
+ −	addition, subtraction	UInt, SInt
* / %	multiplication, division, modulus	UInt, SInt
=== =/=	equal, not equal (returns Bool)	UInt, SInt
> >= < <=	different comparisons (returns Bool)	UInt, SInt

2.1.1 Width Inference

Furthermore, if the output width of a module is not specified, it can be inferred by the tools, while generating the hardware (verilog output). Width inference is a useful feature and does offer certain level of optimization with it. Table 2.2[1] lists the width inference for some of the hardware operations.

Table 2.2: Bit width inference.

Operation	Bit Width
out = in1 + in2	W(out) = max{W(in1), W(in2) }
out = in1 +& in2	W(out) = max{W(in1), W(in2) } + 1
out = in1 & in2	W(out) = max{W(in1), W(in2) }
out = in1 * in2	W(out) = W(in1) + W(in2)
out = in1 ≪ shift	W(out) = W(in1) + max(shift)
out = in1 ≫ shift	W(out) = W(in1) - min(shift)
out = Cat(in1 , in2)	W(out) = W(in1) + W(in2)

A few examples of arithmetic operations with only addition and subtraction are shown in Listing 2.1.

[1] W(x) represents the bit width of signal or wire x.

Note the use of suffix symbols (% and &) on the operators in order to manipulate specific hardware implementation attributes.

```
// Arithmetic operations

// Addition without width expansion
val sum = x + y // OR
val sum = x +%y

// Addition with width expansion
val sum = x +& y

// Subtraction without width expansion
val sum = x - y // OR
val sum = x -%y

// Subtraction with width expansion
val sum = x -& y
```

Listing 2.1: Arithmetic operations.

2.1.2 Bitfield Manipulation

Listing 2.2 illustrates implementations for bitfield manipulations. Chisel library functions or constructs are utilized to implement different bitfield operations. For instance, `Cat` is used to concatenate multiple bitfields, with its first operand taking the leftmost place in the resulting output. Similarly, AND reduction of bits can be performed using `bits.andR` as illustrated in Listing 2.2.

```
// Bitfield manipulations
val xMSB = x(31)         // when x is 32-bit
val yLowByte = y(7, 0)   // y is atleat 8-bit

// concatenates bitfields with first operand on left
val address = Cat(highByte, lowByte)

// replicate a string multiple times
val duplicate = Fill(2, "b1010".U) // "b10101010".U

// Bitfield reductions
val data = "b00111010".U
val allOnes = data.andR             // performs AND reduction
val anyOne = data.orR              // performs OR reduction
val parityCheck = data.xorR        // performs XOR reduction
```

Listing 2.2: Bitfield manipulations

Another useful Chisel utility is `BitPat` (bit pattern), which provides literals with masks. `BitPat` is used for generating bit patterns involving don't care bits. An equality comparison of a bit pattern generated using `BitPat` will ignore don't care bits as illustrated in Listing 2.3.

```
// BitPat example
// define partial opcodes for RISC V instructions
def opCode_BEQ = BitPat("b000?????1100011")
def opCode_BLT = BitPat("b100?????1100011")

// opcode matching with don't care bits
when(opCode_BEQ === "b000110001100011".U){
   // above comparison evaluates to true.B
   // user code
}
```

Listing 2.3: BitPat illustration.

2.2 Mux: A Simple Combinational Block

The Mux is an essential combinational block and the Chisel library provides quiet a few constructs for this. Listing 2.4 shows the implementation of one bit 2-to-1 multiplexer using binary operations. On the other hand Listing 2.5 implements the same 2-to-1 multiplexer with 32-bit wide inputs using the Mux construct from the Chisel library.

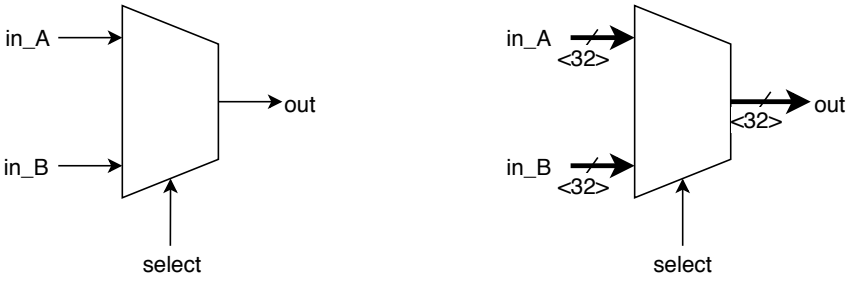

Figure 2.1: A simple 2 to 1 Mux.

```
import chisel3._

// Mux IO interface class
class Mux_2to1_IO extends Bundle {
   val in_A   = Input(Bool())
   val in_B   = Input(Bool())
   val select = Input(Bool())
   val out    = Output(Bool())
}
// 2 to 1 Mux implementation
class Mux_2to1 extends Module {
   val io = IO(new Mux_2to1_IO)

   // update the output
   io.out := io.in_A & io.select | io.in_B & (~io.select)
}
println((new chisel3.stage.ChiselStage).emitVerilog(new Mux_2to1()))
```

Listing 2.4: Mux 2-to-1 with scalar inputs.

```
import chisel3._

// Mux IO interface class
class Mux_2to1_IO extends Bundle {
    val in_A   = Input(UInt(32.W))
    val in_B   = Input(UInt(32.W))
    val select = Input(Bool())
    val out    = Output(UInt())
}
// 2 to 1 Mux implementation
class Mux_2to1 extends Module {
    val io = IO(new Mux2to1_IO)

    // update the output
    io.out := Mux(io.select, io.in_A, io.in_B)
}
println((new chisel3.stage.ChiselStage).emitVerilog(new Mux_2to1()))
```

Listing 2.5: Mux 2-to-1 with vector inputs.

2.2.1 Mux Tree

This is not a new construct, but illustrates different possible ways to wire multiple 2-to-1 multiplexers, to construct higher order multiplexers. You may think of it as nested multiplexers where the output of one multiplexer is the input of another multiplexer. Figure 2.2 shows two different implementations of a 4-to-1 multiplexer. It is important to notice that there are two key differences between these two implementations. First difference is in the number of selection lines. The second difference, which is more important, is the inherent priority of `in_4` over `in_3`, `in_2` and `in_1` for the 4-to-1 multiplexer in Figure 2.2(b). Similarly for this very same multiplexer, the `in_3` has priority over `in_2` and `in_1`. In contrast, all the inputs to the multiplexer in Figure 2.2(a) are of equal priority.

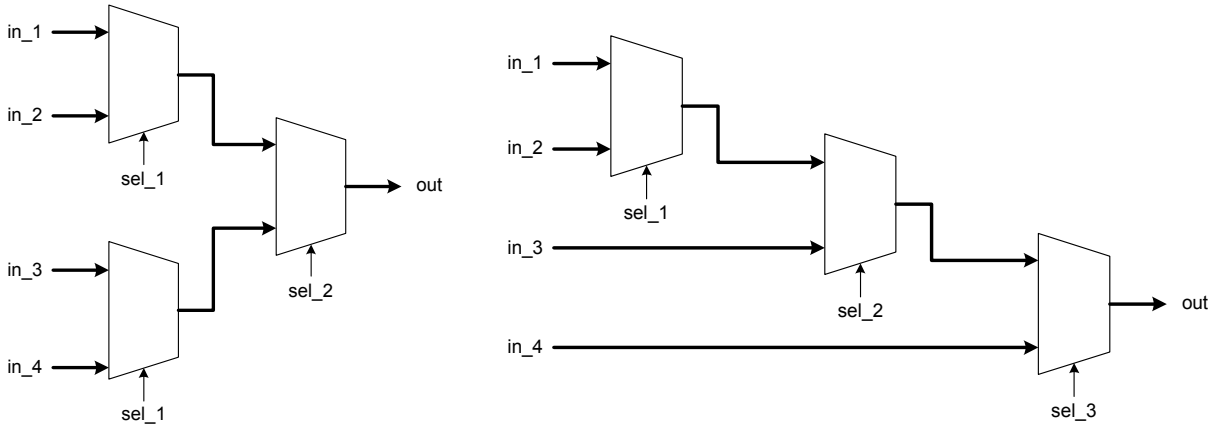

Figure 2.2: Two different 4-to-1 Mux implementations. (a) Equal priority inputs, (b) inputs with priority.

Listing 2.6 shows how an 8-to-1 mux, with equal priority inputs can be implemented. A 4-to-1 multiplexer with input priority is implemented in Listing 2.7. A multiplexer construct with input

priority, named PriorityMux, is available in the Chisel library as well.

```
// An 8-to-1 Mux example
import chisel3._

class LM_IO_Interface extends Bundle {
    val in  = Input(UInt(8.W))
    val s0  = Input(Bool())
    val s1  = Input(Bool())
    val s2  = Input(Bool())
    val out = Output(Bool())          // UInt(1.W))
}

class Mux_8to1 extends Module {
    val io = IO(new LM_IO_Interface)

    val Mux4_to_1_a = Mux(io.s1, Mux(io.s0, io.in(3), io.in(2)),
    Mux(io.s0, io.in(1), io.in(0)))
    val Mux4_to_1_b = Mux(io.s1, Mux(io.s0, io.in(7), io.in(6)),
    Mux(io.s0, io.in(5), io.in(4)))

    val Mux2_to_1 = Mux(io.s2, Mux4_to_1_b, Mux4_to_1_a)

    // Connecting output of 2_to_1 Mux with the output port.
    io.out := Mux2_to_1
}
println((new chisel3.stage.ChiselStage).emitVerilog(new Mux_8to1()))
```

Listing 2.6: Mux 8-to-1 with equal priority inputs.

```
// Mux with input priority
import chisel3._

class IO_Interface extends Bundle {
    val in  = Input(UInt(4.W))
    val s1  = Input(Bool())
    val s2  = Input(Bool())
    val s3  = Input(Bool())
    val out = Output(Bool())    // UInt(1.W))
}

class Mux_Tree extends Module {
    val io = IO(new IO_Interface)

    io.out := Mux(io.s3, io.in(3), Mux(io.s2, io.in(2),
    Mux(io.s1, io.in(1), io.in(0))))
}
println((new chisel3.stage.ChiselStage).emitVerilog(new Mux_Tree()))
```

Listing 2.7: Mux 4-to-1 with input priority.

2.2.2 MuxCase

If there are more than two input lines, we can use `MuxCase` as an alternate of instantiating multiple 2-to-1 multiplexers. This construct implements the same functionality as that of a multiplexer with equal priority inputs and each selection dependency is represented as a tuple in a Scala array. The basic syntax is shown below.

```
MuxCase(default, Array(c1 -> a, c2 -> b, ...))
```

Listing 2.8 implements an 8-to-1 Mux using `MuxCase`. The conditions can be written using Scala collection `Array` or `Seq`. We will discuss more about Scala collections in Exp 9.

Another important thing to note here is importing of another library, `chisel3.util._`, to use `MuxCase` or other variants of multiplexers available in the utilities package of Chisel library. We need to import this library in order to use these constructs.

```
// 8 to 1 mux using MuxCase
import chisel3._
import chisel3.util._

class MuxCase_ex extends Module {
    val io = IO(new Bundle{
        val in0 = Input(Bool())
        val in1 = Input(Bool())
        val in2 = Input(Bool())
        val in3 = Input(Bool())
        val in4 = Input(Bool())
        val in5 = Input(Bool())
        val in6 = Input(Bool())
        val in7 = Input(Bool())
        val sel = Input(UInt(3.W))
        val out = Output(Bool())
    })

    io.out := MuxCase(false.B, Array(
    (io.sel===0.U) ->     io.in0,
    (io.sel===1.U) ->     io.in1,
    (io.sel===2.U) ->     io.in2,
    (io.sel===3.U) ->     io.in3,
    (io.sel===4.U) ->     io.in4,
    (io.sel===5.U) ->     io.in5,
    (io.sel===6.U) ->     io.in6,
    (io.sel===7.U) ->     io.in7)
    )
}

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

Listing 2.8: MuxCase illustration for eight inputs.

2.2.3 MuxLookup

We observed that the input selection in `MuxCase` required boolean expressions to test the select signal. Use of these expressions can be avoided with the `MuxLookup` construct. Similar to `MuxCase`, its outputs and conditions are given as a Scala collection. The basic syntax is shown below.

```
MuxLookup(io.select, default, Array(c1 -> a, c2 -> b, ...))
```

Listing 2.9 implements an 8-to-1 Mux using `MuxLookup`.

```
// 8 to 1 mux using MuxLookup
import chisel3._
import chisel3.util._

class MuxLookup extends Module {
    val io = IO(new Bundle{
        val in0 = Input(Bool())
        val in1 = Input(Bool())
        val in2 = Input(Bool())
        val in3 = Input(Bool())
        val in4 = Input(Bool())
        val in5 = Input(Bool())
        val in6 = Input(Bool())
        val in7 = Input(Bool())
        val sel = Input(UInt(3.W))
        val out = Output(Bool())
    })

    io.out := MuxLookup(io.sel , false.B, Array(
    (0.U) ->    io.in0,
    (1.U) ->    io.in1,
    (2.U) ->    io.in2,
    (3.U) ->    io.in3,
    (4.U) ->    io.in4,
    (5.U) ->    io.in5,
    (6.U) ->    io.in6,
    (7.U) ->    io.in7)
    )
}

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

Listing 2.9: An 8-to-1 mux implementation using MuxLookup.

2.2.4 Mux1H

Another interesting multiplexer construct is the `Mux1H` (termed as Mux 1 hot). In Mux1H, the number of select lines is same as the number of inputs as shown in Figure 2.3. At any particular time instance only one select line should be high and each select line corresponds to one input. If more than one

select lines are high, the output is undetermined. Listing 2.10, shows the usage of Mux1H. For the `sel` signal values of 1, 2, 4 or 8, the corresponding output will be in0, in1, in2 or in3 respectively.

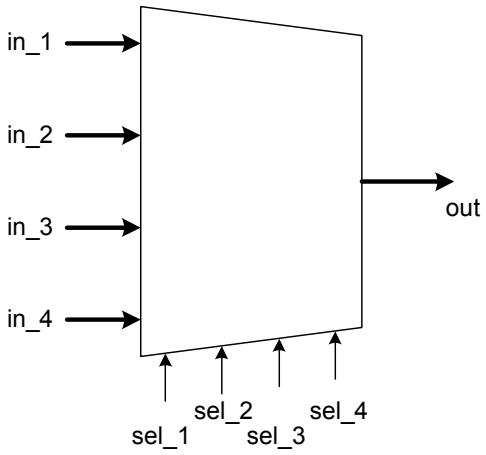

Figure 2.3: Block diagram for Mux1H.

```
// Mux-Onehot example
import chisel3._
import chisel3.util._

class mux_onehot_4to1 extends Module {
    val io = IO(new Bundle {
        val in0 = Input(UInt(32.W))
        val in1 = Input(UInt(32.W))
        val in2 = Input(UInt(32.W))
        val in3 = Input(UInt(32.W))
        val sel = Input(UInt(4.W))
        val out = Output(UInt(32.W))
    })

    io.out := Mux1H(io.sel, Seq(io.in0, io.in1, io.in2, io.in3))
}

println((new chisel3.stage.ChiselStage).emitVerilog(new mux_onehot_4to1()))
```

Listing 2.10: Mux1H illustration for four inputs.

2.3 Bundle and Vectors

If we have multiple input-output (IO) signals that need to be used by a module, we can conveniently put them together using `Bundle`. IO bundles can be constructed using a class that extends from `Bundle` with the signal directions defined explicitly.

Vectors are constructed using `Vec` and can be used to make an array of hardware or signals of same type. For instance, while defining an IO class, we can make an input to be a vector of length n. This will make the input to be an array of n elements. Listing 2.11 illustrates the use of bundles and vectors.

```
import chisel3._

class LM_IO_Interface extends Bundle{
    // Make an input from a Vector of 4 values
    val data_in = Input(Vec(4,(UInt(32.W))))

    // Signal to control which vector is selected
    val data_selector = Input(UInt(2.W))

    val data_out = Output(UInt(32.W))
    val addr = Input(UInt(5.W))

    // The signal is high for write
    val wr_en = Input(Bool())
}

class Mem_bundle_intf extends Module {
    val io = IO(new LM_IO_Interface)

    io.data_out := 0.U

    // Make a memory of 32X32
    val memory = Mem(32, UInt(32.W))

    when(io.wr_en){
        // Write for wr_en = 1
        // Write at memory location addr, with selected data from data_in (
         Vector)
        memory.write(io.addr, io.data_in(io.data_selector))
    } .otherwise{
        // Asyncronous read from addr location
        io.data_out := memory.read(io.addr)
    }
}

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

Listing 2.11: An example illustrating the use of Bundle and Vec.

2.4 Flipped and Bulk constructs

If we want to change the direction of any signal or a bundle of signals, we simply flip the bundle by applying the `Flipped` construct and the direction of all the signals is reversed i.e. from inputs are changed to outputs and vice versa. This is quite useful in scenarios where we are using bundles that are input to one module and output to another module.

We can also connect IO signals with same names across different modules by using the bulk connector `<>`. One scenario, where this would be useful, is a master/slave configuration as shown in Figure 2.4.

In master, we have a bundle which is input from a slave and the slave has the same connections as output to the master. We can connect the same bundle to the master and to the slave using the bulk connector. Listing 2.12 provides an illustration of using flipped and bulk connection.

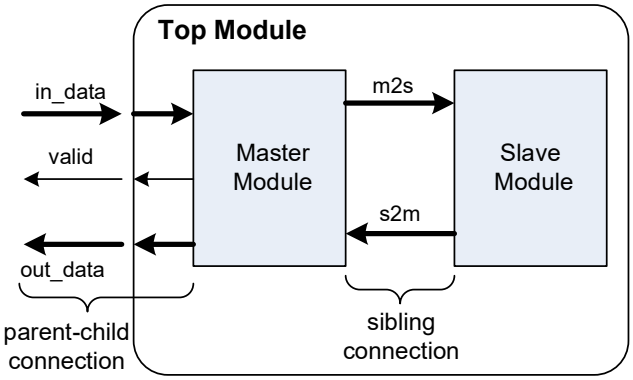

Figure 2.4: Bulk Connection Diagram

```
import chisel3._
import chisel3.util._

class Interface extends Bundle {
    val in_data  = Input(UInt(6.W))
    val valid    = Output(Bool())
    val out_data = Output(UInt(6.W))
}

class MS_interface extends Bundle {
    val s2m  = Input(UInt(6.W))
    val m2s  = Output(UInt(6.W))
}

class Top_module extends Module {
    val io = IO(new Interface)

    val master = Module(new Master)
    val slave  = Module(new Slave)
    //connecting top with master => same direction, same name connects
    io <> master.io.top_int
    //connecting master with slave => opposite direction, same name connects
    master.io.MS <> slave.io
}

class Master extends Module {
    val io = IO(new Bundle {
        val top_int = new Interface
        val MS = new MS_interface
    })
```

```
    io.MS.m2s := io.top_int.in_data
    io.top_int.valid := true.B
    io.top_int.out_data := io.MS.s2m
}

class Slave extends Module {
    val io = IO(Flipped(new MS_interface))

    io.s2m := io.m2s + 16.U
}

println(chisel3.Driver.emitVerilog(new Top_module))
```

Listing 2.12: Flipped and bulk connection example.

2.5 Exercises

Exercise 1: In Listing 2.5, find a way to use combinational hardware instead of the pre-defined Mux module as done in Listing 2.4. The IO bundle must remain the same. (*Hint*: See Listing 2.2 for how to manipulate bits.)

Exercise 2: In Listing 2.6, an 8-to-1 mux is created using mux tree or nested muxes while Listing 2.9 does the same but with MuxLookup. Try to alter Listing 2.9 by using nested MuxLookups. The first MuxLookup will contain only two branches each of which will contain another MuxLookup. These next MuxLookups will contain four branches each.

Exercise 3: Refer to Listing 2.10 to create a 4-to-2 encoder using Mux1H. (*Hint*: You may have to use pre-determined inputs instead of ports.)

2.6 Assignments

Task 1: Write Chisel code for a 5-to-1 multiplexer with specifications given in Table 2.3. A skeleton code is also given in Listing 2.13; use it as a starting point and remember: only write your code is the space specified.

Table 2.3: 5-to-1 mux specifications

{s2,s1,s0}	Output
000	0.U
001	8.U
010	16.U
011	24.U
1xx	32.U

```
package Lab2

import chisel3._

class LM_IO_Interface extends Bundle {
```

```
    val s0  = Input(Bool())
    val s1  = Input(Bool())
    val s2  = Input(Bool())
    val out = Output(UInt(32.W))
}

class Mux_5to1 extends Module {
    val io = IO(new LM_IO_Interface)

    // Start coding here


    // End your code here
}
println((new chisel3.stage.ChiselStage).emitVerilog(new Mux_5to1))
```

Listing 2.13: 5-to-1 Mux skeleton code

Task 2: Write Chisel code of 4-bit Barrel shifter shown in Figure 2.5.

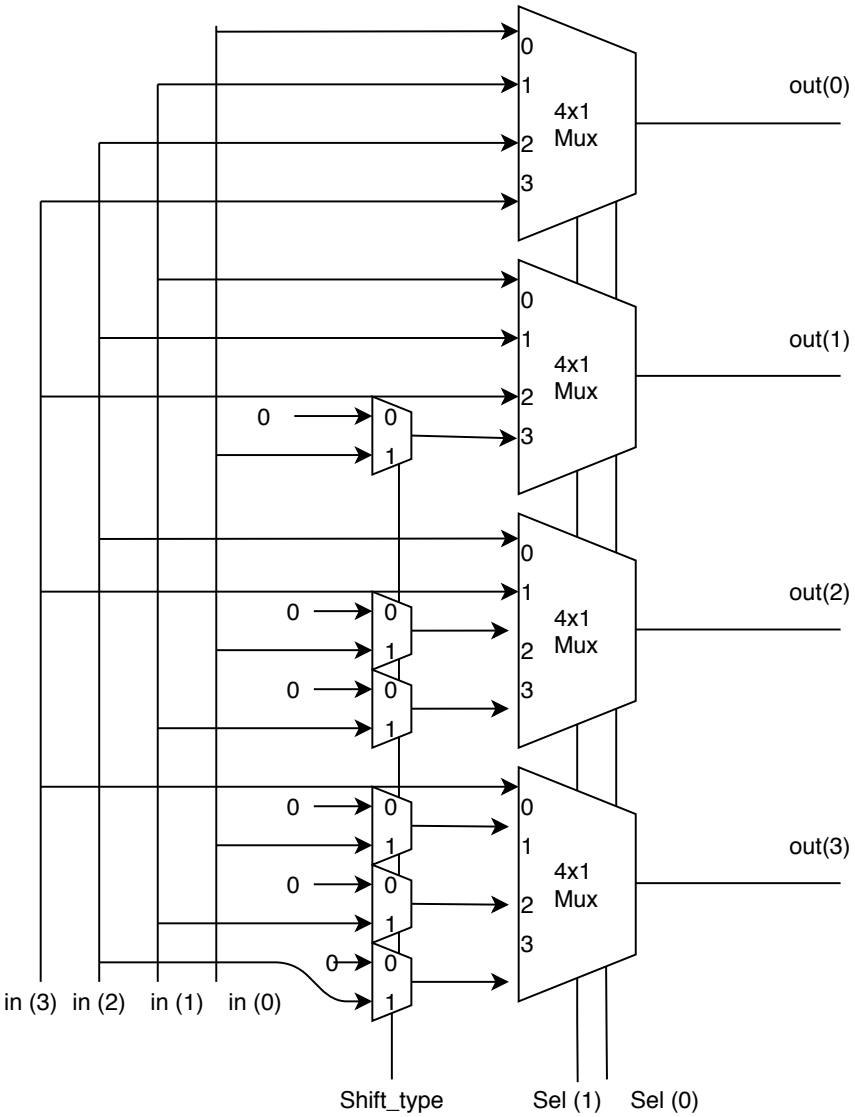

Figure 2.5: Four bit Barrel shifter.

```
package Lab2

import chisel3._
import chisel3.util._

class barrel_shift extends Module{
    val io = IO(new Bundle{
        val in = Vec(4, Input(Bool()))
        val sel = Vec(2, Input(Bool()))
        val shift_type = Input(Bool())
        val out = Vec(4, Output(Bool()))
    })

    // Start you code here


    // End your code here
}
println((new chisel3.stage.ChiselStage).emitVerilog(new barrel_shift))
```

Listing 2.14: 4 bit Barrel shifter skeleton code

Task 3: What are the hardware differences between `MuxCase` and `MuxLookup`.