Experiment 6

Sequential Circuits

Objective

Learn the implementation of new sequential circuits as well as usage of existing modules (Chisel utilities).

6.1 Register Modules

Sequential circuits are heart of any digital design. They are used to implement states and state elements. In addition, state machines and memories, as discussed in subsequent experiments, can be constructed from sequential circuits. There are three different constructs in Chisel to define registers as will be discussed next.

6.1.1 Reg

A simple D type flip flop is used to define `Reg` construct in *Chisel*. Figure 6.1 shows block diagram for single- and multi- bit register using internal clock.

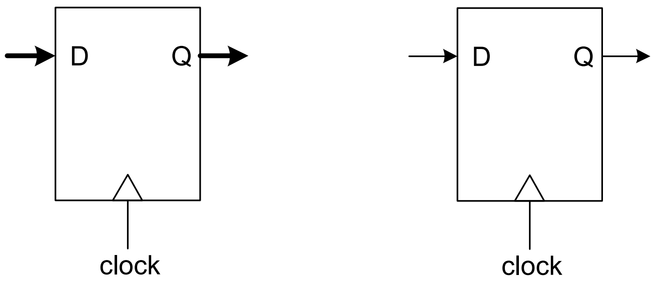

Figure 6.1: One or multi-bit register

The syntax for realizing an n bit register using `Reg` is given below.

```
val reg_nBit = Reg(UInt(n.W))
```

Using the above syntax we have defined a register with signal type `UInt` and having n-bits width. The output of `Reg` is updated with the input value on positive edge of the clock and effectively is delayed by at most one clock cycle. One key attribute of `Reg` is that we cannot assign an initial value to it. Rather the compiler will initialize it with a random value.

6.1.2 RegInit

To overcome the limitation of Reg, which does not allow to assign an initial value, we can use `RegInit` construct to assign an initial value on reset. The block diagram of `RegInit` is shown below in Figure 6.2. From Figure 6.2, we can notice that the only difference between Reg and RegInit is the presence of a Mux, on input signal path, in case of RegInit. When the *reset* is asserted, init_value is applied to the register input.

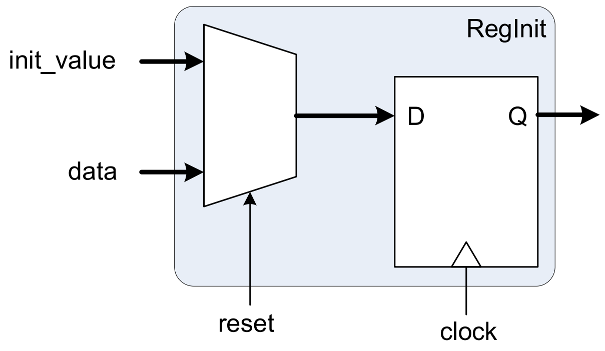

Figure 6.2: Block diagram of `RegInit` construct.

The syntax to instantiate a register using `RegInit` is given below, while Listing 6.1 provides few illustrations for the usage of `RegInit`.

```
val reg_withInit = RegInit(initial_value(n.W))
```

```
// following uses of RegInit are valid
val reg1 = RegInit(24.U(8.W))
val reg2 = Reg(UInt(8.W))
val reg3 = RegInit(reg2)

// following uses are invalid
val reg1 = RegInit(0.U(UInt(8.W)))
val reg2 = RegInit(UInt(8.W))
```

Listing 6.1: Illustration of `RegInit`.

6.1.3 RegNext

`RegNext` is another construct from *Chisel* library, which allows to assign an initial value to the register and one can also connect both the input and the output in one go. It can be used to get one cycle delayed version of the input signal. Connecting an input as well as an output can be done using `RegNext` construct as follows.

```
io.out := RegNext(io.in)
```

6.1.4 RegEnable

Another useful construct is `RegEnable`, which allows to control the updating of the register output as shown in Figure 6.3. The following syntax illustrates its usage.

```
val regWithEnable = RegEnable(nextVal , ena)
```

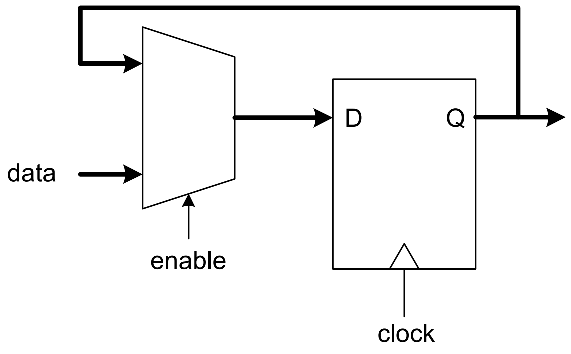

Figure 6.3: Register with enable input.

6.2 Shift Register

Now with the information we have acquired lets make a simple shift register, with serial in and parallel out connectivity as shown in Figure 6.4. Listing 6.2 implements a simple shift-register with 4-bit parallel output.

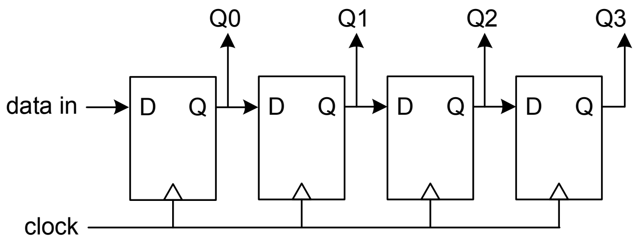

Figure 6.4: Shift register with 4-bit parallel output.

```
// shift register example
import chisel3._

class shift_register(val init: Int = 1) extends Module {
   val io = IO(new Bundle{
      val in = Input(Bool())
      val out = Output(UInt(4.W))
   })
   val state = RegInit(init.U(4.W))   // register initialization

   // serial data in at LSB
   val nextState = (state << 1) | io.in
   state := nextState
```

```
    io.out := state
}
println((new chisel3.stage.ChiselStage).emitVerilog(new shift_register))
```

Listing 6.2: Shift register

6.3 Counter

To this point we have been through many variants of simple counter implementation. Now we are going to implement rather a bit sophisticated variant of counter that can count using predefined minimum and maximum values, while optimizing (minimizing) for the hardware resources. The implementation is shown below in Listing 6.3.

```
// Optimized counter example
import chisel3._
import chisel3.util._

class counter(val max: Int, val min: Int = 0) extends Module {
    val io = IO(new Bundle{
        val out = Output(UInt(log2Ceil(max).W))
    })
    val counter = RegInit(min.U(log2Ceil(max).W))

    // If the max count is of power 2 and the min value = 0,
    // then we can skip the comparator and the Mux
    val count_buffer = if (isPow2(max) && (min == 0)) counter + 1.U
    else Mux(counter === max.U, min.U, counter + 1.U)
    counter := count_buffer
    io.out  := counter
}
println((new chisel3.stage.ChiselStage).emitVerilog(new counter(32)))
```

Listing 6.3: Counter implementation using minimum hardware resources.

In the above example, if the maximum count value is equal to some power of 2 while the minimum value is 0 then we can omit the comparator and Mux. For this special case, we can use a simple incrementing register that overflows when it reaches its maximum value. The conditional code in Listing 6.3 can also be written as follows.

```
val count_buffer = Mux(isPow2(max)&&(min==0).B, counter+1.U, Mux(counter ===
     max.U, min.U, counter+1.U) )
```

Listing 6.4: Modified condition code for counter.

Here the condition for the first Mux will be evaluated as a binary value, which will be responsible for the selection of corresponding path.

6.3.1 One-shot Timer

One shot timer is a simple clock counter that gives a high signal for one cycle when it counts to zero starting with a value from reload register. Listing 6.5 provides one possible implementation.

```
// one shot timer implementation
val timer_count = RegInit(0.U(8.W))
val done = timer_count === 0.U
val next = WireInit(0.U)

when (reload){
   next := din                   // load the data from input
}
.elsewhen (!done){
   next := timer_count - 1.U   // decrement the timer
}
timer_count := next            // update the timer
```

Listing 6.5: One-shot timer implementation.

6.3.2 PWM Generation

PWM generator requires two parameters for proper functioning. One parameter, the *max-cycle* count, determines the frequency of PWM signal and the second parameter, the duty-cycle, determines the pulse width of PWM signal. PWM generator can be implemented using the block diagram shown in Figure 6.5. The implementation of PWM generator is provided in Listing 6.6.

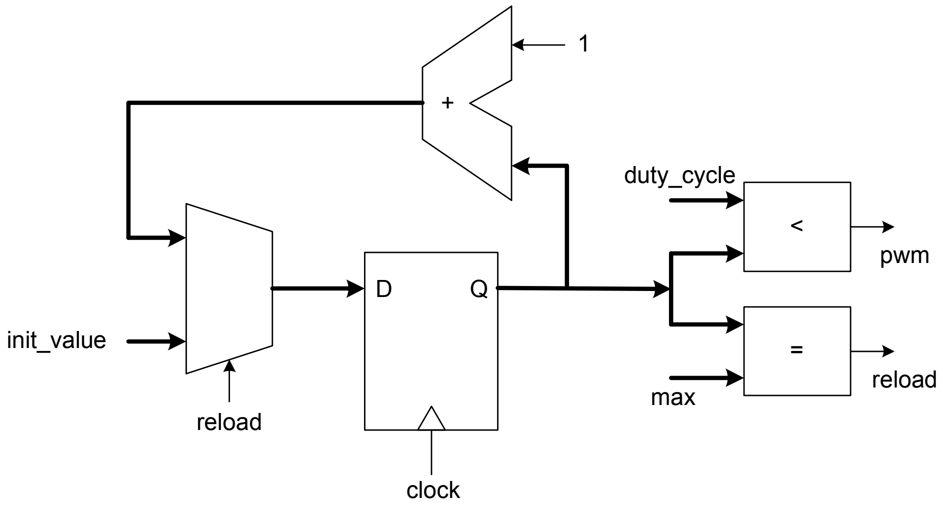

Figure 6.5: Block diagram for PWM generation.

```
// PWM example
import chisel3._
import chisel3.util._

class PWM(val max: Int = 2, val duty_cycle: Int = 1) extends Module {
   val io = IO(new Bundle{
        val out = Output(Bool())
    })
    val counter = RegInit(0.U(log2Ceil(max).W))
    counter := Mux(counter === max.U, 0.U, counter+1.U)
    io.out := duty_cycle.U > counter
}

println((new chisel3.stage.ChiselStage).emitVerilog(new PWM(15)))
```

Listing 6.6: PWM generator.

6.4 Register File

Register file is an essential building block of a microprocessor. The IO interface of a register file mainly depends on the instruction set architecture. For simple assembly instructions, employing register-register operand architecture, the processor might need two source operands from the register file and can also store the result of an operation to the register file. As a result, the register file requires two read ports and one write port. Figure 6.6 shows the block diagram of the register file with read and write ports.

A register file can be implemented using either a register vector or a memory block. Here we will implement a register file using register vector as provided in Listing 6.7.

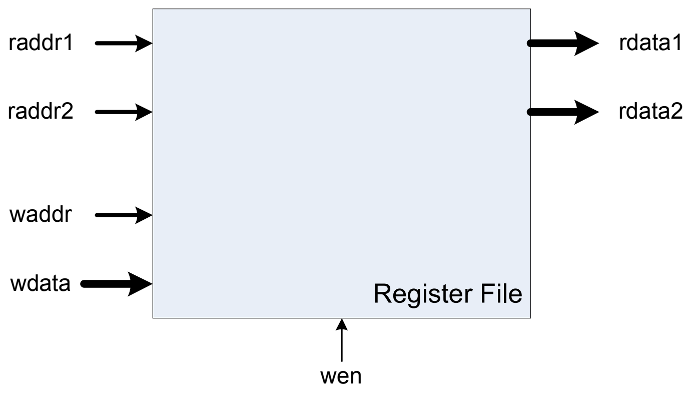

Figure 6.6: The register file with two read and one write port.

```
import chisel3._

class RegFileIO  extends Bundle with Config {
    val raddr1 = Input(UInt(5.W))
    val raddr2 = Input(UInt(5.W))
    val rdata1 = Output(UInt(XLEN.W))
    val rdata2 = Output(UInt(XLEN.W))
    val wen    = Input(Bool())
    val waddr  = Input(UInt(5.W))
    val wdata  = Input(UInt(XLEN.W))
}

class RegFile extends Module with Config {
    val io = IO(new RegFileIO)
    val regs = Reg(Vec(REGFILE_LEN , UInt(XLEN.W)))

    io.rdata1 := Mux((io.raddr1.orR), regs(io.raddr1), 0.U)
    io.rdata2 := Mux((io.raddr2.orR), regs(io.raddr2), 0.U)

    when(io.wen & io.waddr.orR) {
        regs(io.waddr) := io.wdata
    }
}
```

Listing 6.7: Register file implementation using register vector.

6.5 Pipes and Queues

6.5.1 Pipe

Pipe is a Chisel construct that can delay the inputs by a specified amount of time. It requires two input arguments, a Valid interface for data input and an integer by which latency of pipe will be set. Output data will be delayed by the specified number of clock cycles. Listing 6.8 illustrates an example implementation using Pipe construct.

```

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

class Pipe 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 := Pipe(io.in,5)
}
println(chisel3.Driver.emitVerilog(new Pipe))
```

Listing 6.8: Pipe construct.

6.5.2 Queues

Queue creates a FIFO (first-in, first-out) with Decoupled interfaces for both input and output connectivity as depicted in Figure 6.7. Both the data type and the depth of the queue i.e the number of data elements, are configurable. The syntax of the Queue construct in Chisel is illustrated in Listing 6.9.

```
class My_Queue extends Module{
```

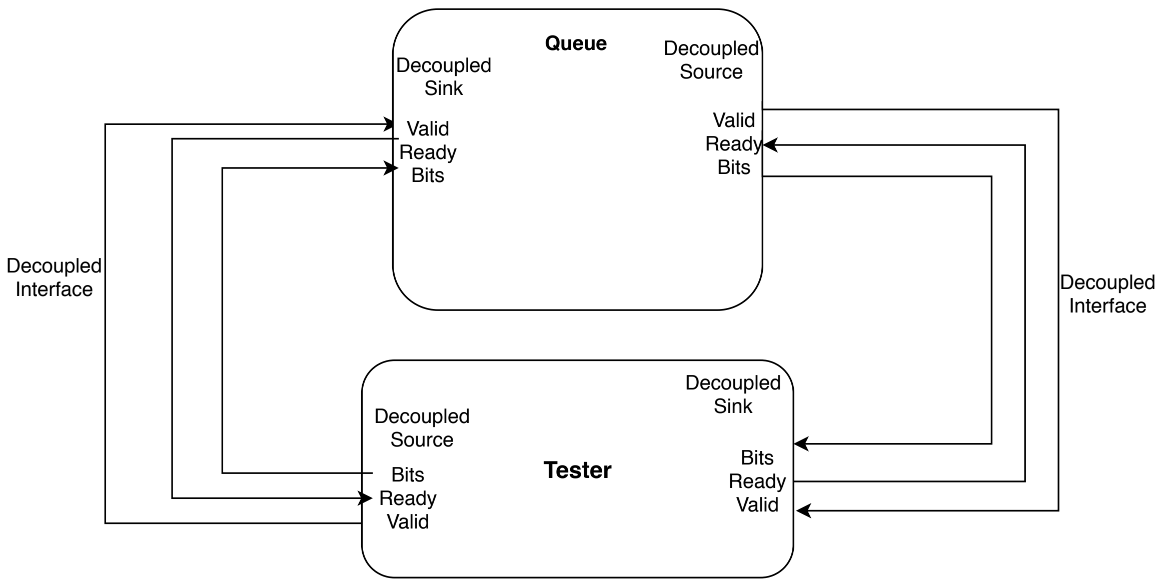

Figure 6.7: Queue with decoupled interface.

```
  val io  = IO(new Bundle {
    val in  = Flipped(Decoupled(UInt(8.W))) //valid = Input,  ready =
      Output, bits = Input
    val out = Decoupled(UInt(8.W))          //valid = Output, ready =
      Input , bits = Output
  })
  val queue = Queue(io.in, 5)              // 5-element queue
  io.out <> queue
}
```

Listing 6.9: Queue with decoupled interface.

6.6 Black Boxes

Integrating existing Verilog IP is an essential part of many chip designs. Chisel provides support to integrate Verilog IP using BlackBox. The module defined as BlackBox will be instantiated in the generated Verilog, but no Verilog code will be generated to define the behavior of the module. In addition, there is no implicit clock or reset in BlackBox, which is similar to RawModule. So clock and reset needs to be explicitly declared and connected to BlackBox.

Let us use BlackBox for the implementation of a 32-bit Adder. Listing 6.10 shows the use of BlackBox to define the Adder module in Verilog. The Verilog source file is placed at the path src/main/resouces/Adder.v. Alternatively, we can also include Verilog code inline in the BlackBox as illustrated in Listing 6.11.

```
class BlackBoxAdder extends BlackBox with HasBlackBoxResource {
  val io = IO(new Bundle() {
    val in1 = Input(UInt(32.W))
    val in2 = Input(UInt(32.W))
    val out = Output(UInt(33.W))
  })
```

```
    setResource("/Adder.v")
}
```

Listing 6.10: Adder module using BlackBox.

```
class BlackBoxAdder extends BlackBox with HasBlackBoxInline {
    val io = IO(new Bundle() {
        val in1 = Input(UInt(32.W))
        val in2 = Input(UInt(32.W))
        val out = Output(UInt(33.W))
    })
    setInline("BlackBoxAdder.v",
    s"""
    |module BlackBoxAdder(
    |  input [32:0] in1,
    |  input [32:0] in2,
    |  output [33:0] out
    |);
    |always @* begin
    |  out <= ((in1) + (in2));
    |end
    |endmodule
    """.stripMargin)
}
```

Listing 6.11: Adder module using BlackBox with inline Verilog.

Since Chisel does not accept BlackBoxes as a top Module, to use BlackBox based Adder we need to instantiate in another module as given below.

```
val BBAdder = Module(new BlackBoxAdder)
```

6.7 Exercises

Exercise 1: Add parallel load capability to shift register implemented in Listing 6.2.

Exercise 2: Modify the counter implementation in the Listing 6.3 using the recommended modification from Listing 6.4.

Exercise 3: Modify the one shot timer in Listing 6.5 to work as a two shot timer.

Exercise 4: Figure 6.8 contains three decoupled interfaces. Each has a ready,valid and data whenever both ready and valid are high it means its a valid transaction otherwise no transaction will be occurred. There are two queues, each has depth of ‘5’ and width of UInt 8-bits. It can be seen from the figure decoupled interface between the queues is controlled by the queues automatically and perform enqueue and dequeue operations but you can control other two decoupled interface from testbench. Write Chisel code and simulate behavior using the skeleton code provided in Listing 6.12. Use decoupled interfaces and Queue construct and explain the behavior of circuit. Use io.in(decoupled interface) and io.out (decoupled interface) in your code.

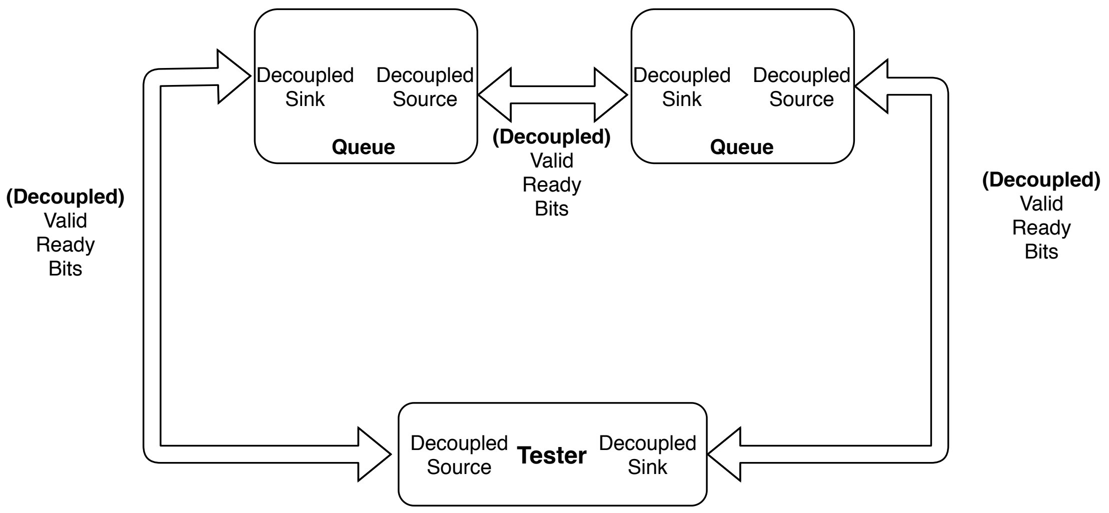

Figure 6.8: Decoupled and Queues.

```
package Lab6
import chisel3._
import chisel3.util._
import chisel3.iotesters.{ChiselFlatSpec,Driver,PeekPokeTester}

class My_Queue extends Module{
// your code begin

// your code end here
}
```

Listing 6.12: Skeleton code for problem 1.

6.8 Assignments

Task 1: What hardware will be created in the counter example above in Listing 6.3, if we pass 4 to it or if we passed 13 to it as max count.

Task 2: Make a counter by using 1-bit XOR. Meaning that the MSB of the register would be checked if its one it should go back to 0 else it should keep on counting. This counter should be parameterized for different values max-count, the counter is shown in Figure 6.9 there is also a limitation to this approach what is it? The skeleton code is provided in Listing 6.13.

```
// Counter with XOR example
import chisel3._
import chisel3.util._

class counter_with_xor(val max: Int = 1) extends Module {
    val io = IO(new Bundle{
        val out = Output(UInt((log2Ceil(max).W)))
    })
    // Start Coding here
```

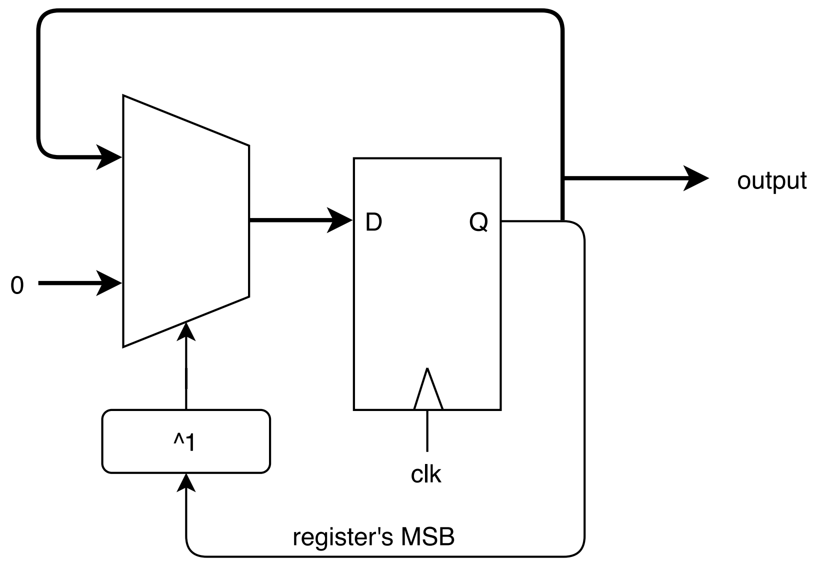

Figure 6.9: Counter using 1-bit XOR.

```

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

Listing 6.13: Counter using 1-bit XOR.

Task 3: Make a generator for shift register with parallel load inputs capability. Block diagram of the assignment is given below in Figure 6.10. The skeleton code is provided in Listing 6.14.

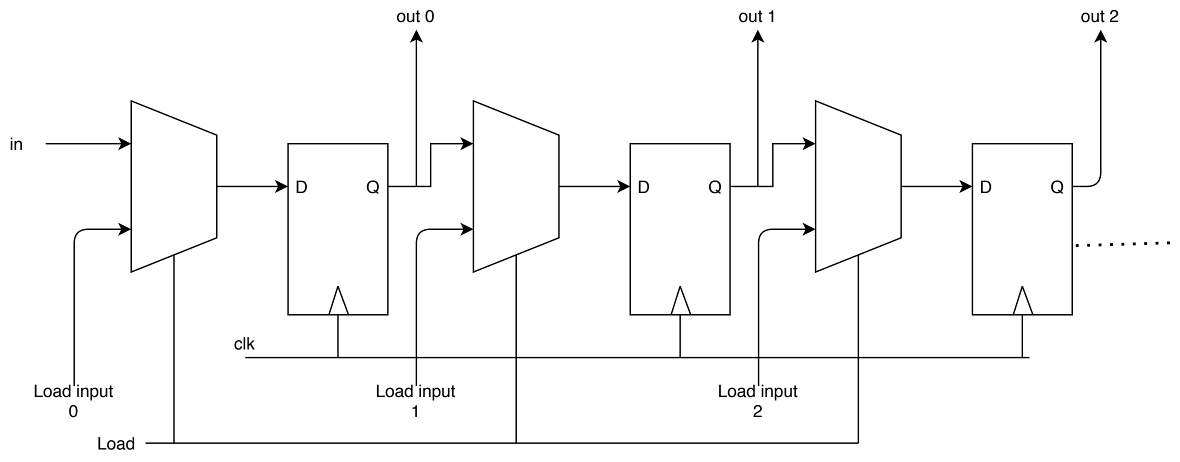

Figure 6.10: Shift register with parallel load.

```
// Shift register with parallel load
import chisel3._
import chisel3.util._

class shift_reg_with_parallel_load(val len : Int=1) extends Module {
    val io = IO(new Bundle{
        val out = Vec(len, Output(Bool()))
        val load_in = Vec(len, Input(Bool()))
        val in = Input(Bool())
        val load = Input(Bool())
    })
    // 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
    shift_reg_with_parallel_load(n)))
```

Listing 6.14: Skeleton code from shift register with parallel input load generator.

Task 4: Consider the up-down counter that has 1 bit control input, which controls the count direction as up or down counter. If up-down bit is 1 the counter will count upward and count down otherwise. Write a Chisel program that implements this counter.

```
// Up-down counter example
import chisel3._
import chisel3.util._

class up_down_counter(val max: Int = 10) extends Module {
    val io = IO(new Bundle{
        val out = Output(UInt(log2Ceil(max).W))
        val up_down = Input(Bool())
    })
    // Start code here


    // End your code here
}

println((new chisel3.stage.ChiselStage).emitVerilog(new up_down_counter(4)))
```

Listing 6.15: Skeleton code for up down counter.