Muhammad Shahzaib

Assignment 3

Exercise1:

Code:

```
package assignment3

object Exercise1and2 extends App{
  println("-------EXERCISE1---------")
  type R = Int

  val ulst = List(-3,-2,-1,0,1,2,3)

  def compose(g : R => R ,  h: R => R , k: R=>R) = (x:R) => g(h(k(x)))

  // implement ax^2 + bx + c (a = 3 , b= 5 , c =7)
  def y1 = compose(x => (x*x) * 3  , _*5 ,  _+7 )

  val umap1 = ulst.map(y1(_))

  println(s"linealy mapped 1 = $umap1")

}
```

Output:

```
-------EXERCISE1---------
linealy mapped 1 = List(1200, 1875, 2700, 3675, 4800, 6075, 7500)
```

Muhammad Shahzaib

Exercise 2:

Code:

```
val ulst_zipped = ulst.zip(umap1)
println(s"Zipped ulst with umap1 = $ulst_zipped")

val ulst_zip_index = ulst_zipped.zipWithIndex
println(s"Resulting list zip with index = $ulst_zip_index")

val result = ulst_zipped.reduce((a,b) => (a._1 + b._1 , a._2 + b._2))
println(s"result $result")
val mean = result._2.toFloat / ulst_zip_index.length
println(s"mean of f(x) = $mean")
```

Output:

```
-------EXERCISE2---------
Zipped ulst with umap1 = List((-3,1200), (-2,1875), (-1,2700), (0,3675), (1,4800), (2,6075), (3,7500))
Resulting list zip with index = List(((-3,1200),0), ((-2,1875),1), ((-1,2700),2), ((0,3675),3),
((1,4800),4), ((2,6075),5), ((3,7500),6))
result (0,27825)
mean of f(x) = 3975.0
```

Muhammad Shahzaib

Exercise 3:

Code:

```
package assignment3

object Exercise3 extends App{

  def magnitude(vec: Vector[Int]): Unit = {
    val ulst_square = vec.map(x => x*x)
    val ulst_add = ulst_square.foldLeft(0)(_+_)  // for addition of list
    val ulst_sqrt = math.sqrt(ulst_add)
    println(s"Square of every element in vector = $ulst_square")
    println(s"Addition of vector = $ulst_add")
    println(s"Final answer, norm of vector = $ulst_sqrt")
  }

  val ulst = Vector(1,2,3,4,5)
  magnitude(ulst)
}
```

Output:

```
Square of every element in vector = Vector(1, 4, 9, 16, 25)
Addition of vector = 55
Final answer, norm of vector = 7.416198487095663
```

Muhammad Shahzaib

Exercise 4:

Code:

```
package assignment3

object Exercise4 extends App{
  println("-------listing 11.2 using wildcard------------")
  val  ulst = List(1,2,3,4,5)

  val ulst_twice = ulst.map(_*2)
  println(s"list elements doubled = $ulst_twice")

  val f = (g:Int) => if (g > 2) g*g else None

  val ulst_squared  =ulst.map(f(_))
  println(s"list elements squared selectively = $ulst_squared")


  println("-------listing 11.3 using wildcard------------")
  //def g(v:Int) = List(v-1 , v , v+1)
  val g = (v:Int) => List(v-1 , v , v+1)
  val ulst_extended = ulst.map(g(_))
  println(s"Extended list using map = $ulst_extended")

  val ulst_extended_flatmap = ulst.flatMap(g(_))
  println(s"Extended list using flatmap = $ulst_extended_flatmap")



  println("-------listing 11.4 using wildcard------------")
  //def h(x:Int) = if (x>2) Some(x) else None
  val h = (x:Int) => if (x>2) Some(x) else None
  val ulst_selective = ulst.map(h(_))
  println(s"Selective element of list with map = $ulst_selective")

  val ulst_selective_flatmap = ulst.flatMap(h(_))
  println(s"Selective element of list with flatmap = $ulst_selective_flatmap")


}
```

Muhammad Shahzaib

Output:

```
-------listing 11.2 using wildcard------------
list elements doubled = List(2, 4, 6, 8, 10)
list elements squared selectively = List(None, None, 9, 16, 25)
-------listing 11.3 using wildcard------------
Extended list using map = List(List(0, 1, 2), List(1, 2, 3), List(2, 3, 4), List(3, 4, 5), List(4, 5,
6))
Extended list using flatmap = List(0, 1, 2, 1, 2, 3, 2, 3, 4, 3, 4, 5, 4, 5, 6)
-------listing 11.4 using wildcard------------
Selective element of list with map = List(None, None, Some(3), Some(4), Some(5))
Selective element of list with flatmap = List(3, 4, 5)
```

Muhammad Shahzaib

Exercise 1 (FSM):

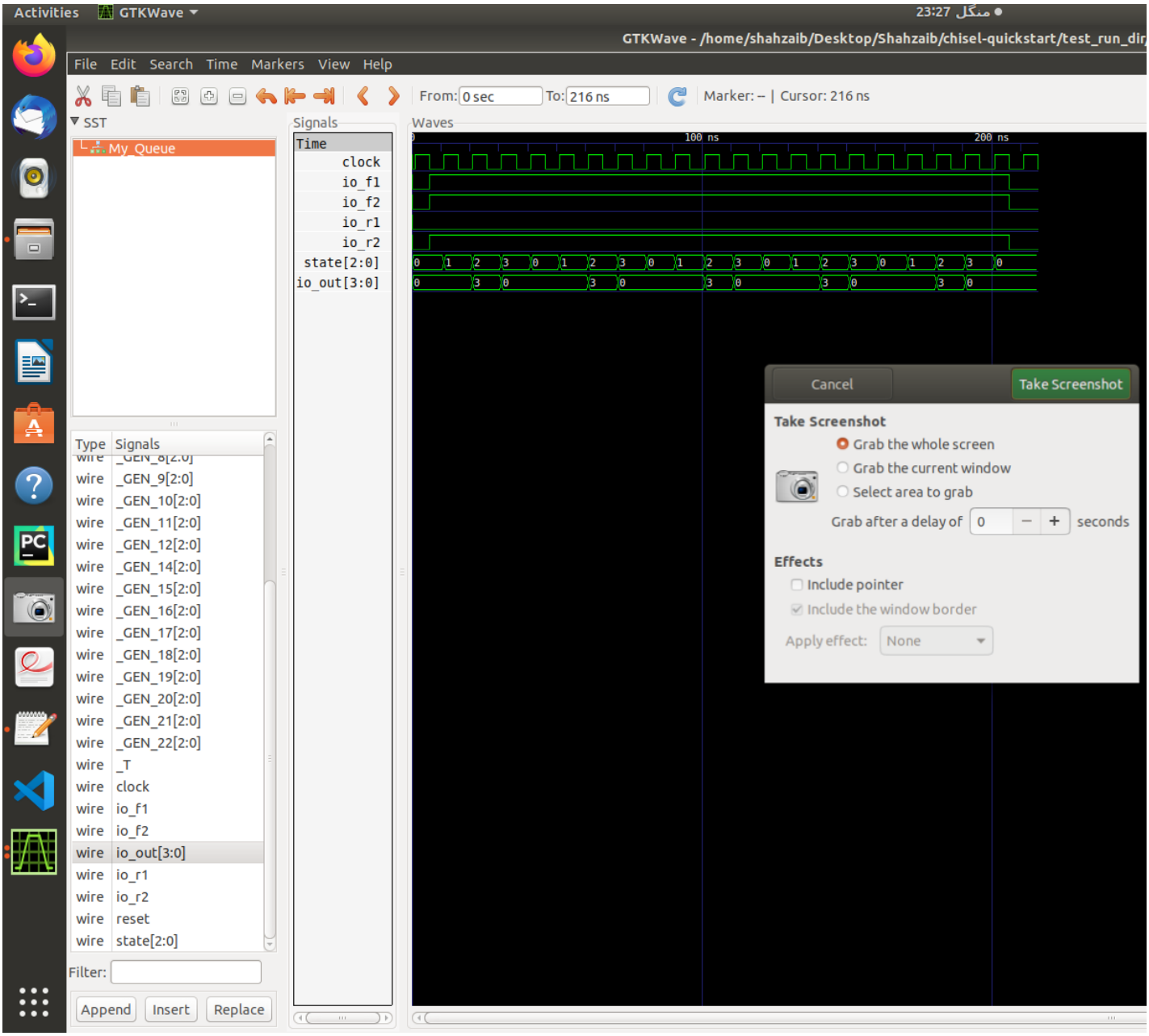
Counter till 3:

Code:

```scala
package lab7
import chisel3._
import chisel3.util._


class My_Queue extends Module{
    val io = IO(new Bundle {
        val f1 = Input(Bool())
        val f2 = Input(Bool())
        val r1 = Input(Bool())
        val r2 = Input(Bool())
        val out = Output(UInt(4.W))

    })

    val s0 :: s1 :: s2 :: s3 :: s4 :: s5 :: Nil = Enum(6)
    val state = RegInit(s0)

    switch(state){
        is(s0){
            when(io.f1 === false.B && io.f2 === false.B){
                state := s0
            }.elsewhen(io.f1 === true.B && io.f2 === false.B){
                state := s1
            }.elsewhen(io.f1 === false.B && io.f2 === true.B){
                state := s5
            }.elsewhen(io.f1 === true.B && io.f2 === true.B){
                state := s1
            }
        }

        is(s1){
            when(io.f1 === false.B && io.r1 === false.B){
                state := s1
            }.elsewhen(io.f1 === true.B){
                state := s2
            }.elsewhen(io.f1 === false.B && io.r1 === true.B){
                state := s0
            }
        }

        is(s2){
            when(io.f1 === false.B && io.r1 === false.B){
                state := s2
            }.elsewhen(io.f1 === true.B){
                state := s3
            }.elsewhen(io.f1 === false.B && io.r1 === true.B){
                state := s1
            }
        }

        is(s3){
            state := s0
        }

        is(s4){
            when(io.f2 === true.B){
                state := s3
            }.elsewhen(io.f2 === false.B && io.r2 === false.B){
                state := s4
            }.elsewhen(io.f1 === false.B && io.r1 === true.B){
                state := s5
            }
        }

        is(s5){
            when(io.f2 === true.B){
                state := s4
            }.elsewhen(io.f2 === false.B && io.r2 === false.B){
                state := s5
            }.elsewhen(io.f1 === false.B && io.r1 === true.B){
                state := s0
            }
        }
    }

    when ((state === s2 && (io.f1 === false.B && io.r1 === false.B)) || (state === s2 && (io.f1 ===
true.B)) || (state === s2 && (io.f1 === false.B) && (io.r1 === true.B))){
            io.out:=3.U
        }.elsewhen((state === s4 && (io.f2 === true.B)) || (state === s4 && (io.f2 === false.B && io.r2
=== false.B)) || (state === s4 && (io.f2 === false.B ) && (io.r2 === true.B))){
            io.out:=7.U
        }.otherwise{
             io.out:=0.U
    }

}

object My_Queue{
    def main(args: Array[String])
    {
        var obj = new My_Queue();
    }
}
```

Muhammad Shahzaib

Output:



Exercise 2 (Deep copy):

Code:

```
package assignment3

object Exercise2_deep_copy extends App{

  case class car(brand: String , model: String , price: Int)

  val c1 = car("Toyota" , "Corolla" , 200000)
  println("Brand is " , c1.brand)
  println("Model is " , c1.model)
  println("Price is " , c1.price)

  //lets create a copy of object c1
  println("After create a copy of c1")
  println("Its create a deep copy because it copy every thing from the object")
  val c2 = c1.copy()
  println("Brand is " , c2.brand)
  println("Model is " , c2.model)
  println("Price is " , c2.price)

  //lets create another copy of c1 and change model and price only
  println("After create a copy of c1 and change model and price")
  val c3 = c1.copy(model = "Aqua" , price = 280000)
  println("Brand is " , c3.brand)
  println("Model is " , c3.model)
  println("Price is " , c3.price)

}
```

Output:

```
(Brand is ,Toyota)
(Model is ,Corolla)
(Price is ,200000)
After create a copy of c1
Its create a deep copy because it copy every thing from the object
(Brand is ,Toyota)
(Model is ,Corolla)
(Price is ,200000)
After create a copy of c1 and change model and price
(Brand is ,Toyota)
(Model is ,Aqua)
(Price is ,280000)
```

Muhammad Shahzaib

Exercise 3 (Map and Flatmap):

Code:

```
package assignment3

object Exercise3_map_flatmap extends App{

  println("--------map on LIST--------------")
  val ulst = List(1,5,7,8 , 2)
  val ulst_mod = ulst.map(x => x*2)
  println(ulst_mod)

  println("--------map on SET--------------")
  val uset = Set(1,5,7,8 , 2)
  val uset_mod = uset.map(x => x*2)
  println(uset_mod)


  println("--------map on SEQUENCE--------------")
  val useq = Seq(1,5,7,8 , 2)
  val useq_mod = useq.map(x => x*2)
  println(useq_mod)

  println("--------map on Array--------------")
  val uarr = Array(1,5,7,8 , 2)
  val uarr_mod = uarr.map(x => x*2)
  println(uarr_mod.toList)

  println("--------map on Vector--------------")
  val uvec = Vector(1,5,7,8 , 2)
  val uvec_mod = uvec.map(x => x*2)
  println(uvec_mod.toList)

  println("--------map on Map--------------")
  val uMap = Map('a' -> 2 , 'b' -> 4 , 'c' -> 6)
  val l = (k:Int , v:Int) => Some(k -> v*2)

  val uMap_map = uMap.map {
    case (k, v) => l(k, v)
  }
  println(s"map values doubled using map = $uMap_map")

  val uMap_flatmap = uMap.flatMap{
    case (k,v) => l(k,v)
  }
  println(s"map values doubled using map = $uMap_flatmap")

}
```

Output:

```
--------map on LIST--------------
List(2, 10, 14, 16, 4)
--------map on SET--------------
HashSet(10, 14, 2, 16, 4)
--------map on SEQUENCE--------------
List(2, 10, 14, 16, 4)
--------map on Array--------------
List(2, 10, 14, 16, 4)
--------map on Vector--------------
List(2, 10, 14, 16, 4)
--------map on Map--------------
map values doubled using map = List(Some((97,4)), Some((98,8)), Some((99,12)))
map values doubled using map = Map(97 -> 4, 98 -> 8, 99 -> 12)
```

Exercise 1 (apply function):

Code:

```
package assignment3

object Exercise1_apply extends App{

  def apply(lst : List[Int]): Unit = {
    val modified_list = List(lst.foldLeft(0)(_+_))
    println (s" Apply method for the List with .apply = ${modified_list.apply (0)}")
  }

  val ulst = List(1,2,3,4,5)
  apply(ulst)

}
```

Output:

```
Apply method for the List with .apply = 15
```

Exercise 1 (implicit):

Code:

```
package assignment3
import scala.language.implicitConversions

object Exercise1_implicit extends App{

  class Implicit_Function (i : Int) {
    // Implicit conversion to List and then increment
    val i_implicit_seq_inc = i.map(_ + 3)

    // Implicit function for conversion
    implicit def any_name(i: Int): List[Int] = List(i)
  }

  val convert = new Implicit_Function(1)
  println("Implicit conversion and increment: " +convert.i_implicit_seq_inc)

}
```

Output:

```
Implicit conversion and increment: List(4)
```

Exercise 2(Implicit):

Code:

```
package assignment3
import scala.language.implicitConversions

object Exercise2_implicit extends App {

  def implct(i : Any , j: Any): Unit ={

    val i_implicit_seq_inc = i + j.toString

    implicit def any_name(i: Any , j: Any) = {
      i.toString
      j.toString

    }


    println(i_implicit_seq_inc)
  }

  implct( "Hello " , "World")

}
```

Output:

Muhammad Shahzaib