# 实验八：多层级模块设计

## 1. 实验目的

1.1. 学习模块化的硬件设计方法和 chisel 模块的例化和连线；

1.2. 学习常用的基于 valid 和 ready 信号的握手通信协议。

## 2. 实验内容

2.1. 学习模块例化和连线的方法；

2.2. 使用 valid-ready 通信协议进行数据传输。

## 3. 实验相关 Chisel 语法介绍

### 3.1. 模块(Module)例化

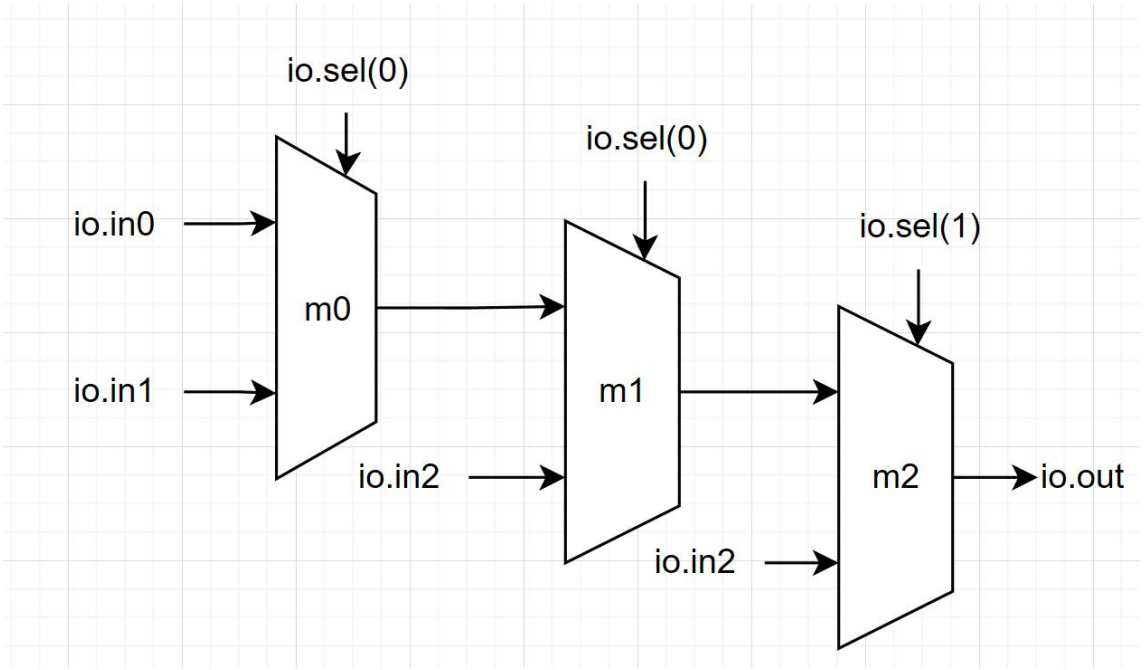
在 Chisel 里面是用一个自定义的类来定义模块的，这个类有以下三个特点：（1）继承自 Module 类。(2) 有一个抽象字段“io”需要实现，该字段必须引用前面所说的端口对象。(3)在类的主构造器里进行内部电路连线。

因为非字段、非方法的内容都属于主构造方法,所以用操作符“:=”进行的赋值、用“<>”进行的连线或一些控制结构等，都属于主构造方法。从 Scala 的层面来讲，这些代码在实例化时表示如何构造一个对象；从 Chisel 的层面来讲，它们就是在声明如何进行模块内部子电路的连接、信号的传递，类似于 Verilog 的 assign 和 always 语句。实际上这些用赋值表示的电路连接在转换成 Verilog 时，在组合逻辑中就是大量的 assign 语句，在时序逻辑中就是 always 语句。

还有一点需要注意，这样定义的模块会继承一个字段“clock”，类型是 Clock，它表示全局时钟，在整个模块内可见。对于组合逻辑，是用不上它的，而时序逻辑虽然需要这个时钟但也不用显式声明。除 clock 外还会继承一个字段“reset”,类型是 Reset，表示全局复位信号，在整个模块内可见。对于需要复位的时序元件，也可以不用显式声明该字段。如果确实需要用到全局时钟和复位，则可以通过它们的字段名称来使用，但需要注意类型是否匹配，经常需要“reset.asBool”这样的语句把 Reset 类型转换成 Bool 类型用于控制。隐式的全局时钟和复位端口只有在生成 Verilog 代码时才能看到。

要例化一个模块，并不是直接用 new 生成一个实例对象就完成了，还需要再把实例对象传递

给单例对象 Module 的 apply 方法。这种形式是由 Scala 的语法限制造成的，就像端口需要写成“IO(new Bundle {.})”、无符号数要写成“UInt(n.W)”等一样。例如，下面的代码演示了通过例化双输入多路选择器构建四输入多路选择器，电路图和代码如下所示：




```
1.  import chisel3._
2.
3.  class Mux2 extends Module {
4.    val io = IO(new Bundle{
5.      val sel = Input(UInt(1.W))
6.      val in0 = Input(UInt(1.W))
7.      val in1 = Input(UInt(1.W))
8.      val out = Output(UInt(1.W))
9.    })
10.
11.   io.out := (io.sel & io.in1) | (~io.sel & io.in0)
12. }
13.
14. class Mux4 extends Module {
15.   val io = IO(new Bundle {
16.     val in0 = Input(UInt(1.W))
17.     val in1 = Input(UInt(1.W))
18.     val in2 = Input(UInt(1.W))
19.     val in3 = Input(UInt(1.W))
20.     val sel = Input(UInt(2.W))
21.     val out = Output(UInt(1.W))
22.   })
23.   val m0 = Module(new Mux2)
24.   m0.io.sel := io.sel(0)
```

```
  m0.io.in0 := io.in0
  m0.io.in1 := io.in1
  val m1 = Module(new Mux2)
  m1.io.sel := io.sel(0)
  m1.io.in0 := io.in2
  m1.io.in1 := io.in3
  val m2 = Module(new Mux2)
  m2.io.sel := io.sel(1)
  m2.io.in0 := m0.io.out
  m2.io.in1 := m1.io.out
  io.out := m2.io.out
}
```

上述代码中的 Mux2 模块是我们前面实验中写过的模块，我们要用 2 选 1 选择器 Mux2 来构建一个 4 选 1 选择器 Mux4。选择器 Mux4 接收 4 个输入，in0、in1、in2 和 in3，和选择信号 sel。我们通过把 new Mux2 作为参数传递给 Module，例化模块 m0、m1 和 m2。然后再把 m0 模块的输出作为 m1 模块的其中一个输入，用:=的方法连接起来即可。

在上面的例子中，模块 Mux2 例化了 3 次，实际只需要一次性例化 3 个模块就可以了。对于要多次例化的重复模块，可以利用向量的工厂方法 VecInit[T <: Data]。因为该方法接收的参数类型是 Data 的子类，而模块的字段 io 正好是 Bundle 类型，并且实际的电路连线仅仅只需针对模块的端口，所以可以把待例化模块的 io 字段组成一个序列，或者按重复参数的方式作为参数传递。通常使用序列作为参数，这样更节省代码。生成序列的一种方法是调用单例对象 Seq 里的方法 fill，该方法的一个重载版本有两个单参数列表，第一个接收 Int 类型的对象，表示序列的元素个数，第二个是传名参数，接收序列的元素。

因为 Vec 是一种可索引的序列，所以这种方式例化的多个模块类似于“模块数组”，用下标索引第 n 个模块。另外，因为 Vec 的元素已经是模块的端口字段 io，所以要引用例化模块的某个具体端口时，路径里不用再出现“io”。例如：

```
import chisel3._
class Mux4_2 extends Module {
  val io = IO(new Bundle {
    val in0 = Input(UInt(1.W))
    val in1 = Input(UInt(1.W))
    val in2 = Input(UInt(1.W))
    val in3 = Input(UInt(1.W))
    val sel = Input(UInt(2.W))
    val out = Output(UInt(1.W))
  })
  // 例化了三个 Mux2，并且参数是端口字段 io
  val m = VecInit(Seq.fill(3)(Module(new Mux2).io))
  m(0).sel := io.sel(0)  // 模块的端口通过下标索引，并且路径里没有“io”
```

```
14.   m(0).in0 := io.in0
15.   m(0).in1 := io.in1
16.   m(1).sel := io.sel(0)
17.   m(1).in0 := io.in2
18.   m(1).in1 := io.in3
19.   m(2).sel := io.sel(1)
20.   m(2).in0 := m(0).out
21.   m(2).in1 := m(1).out
22.   io.out := m(2).out
23. }
```

## 3.2. valid-ready 握手协议解析

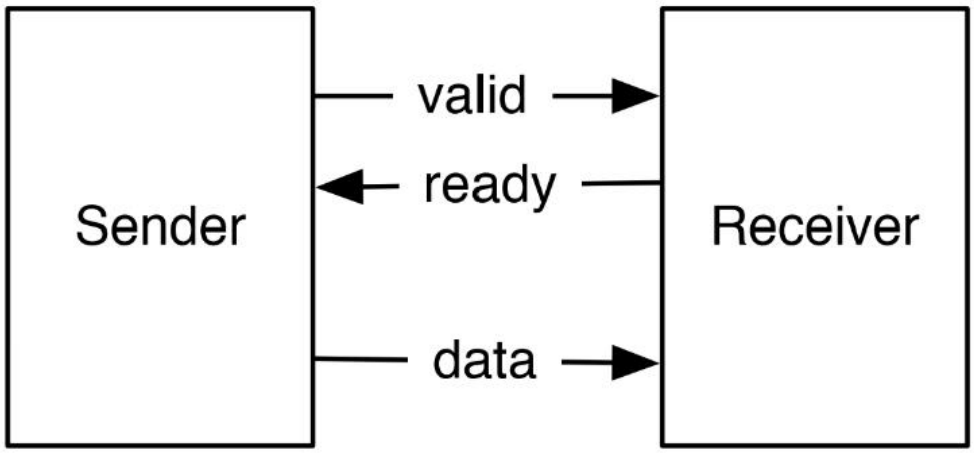
数字电路中经常需要用到 valid ready 握手协议，很多总线都是基于该协议。数据产生端——生产者数据准备好后将标志信号 valid 置位。数据接受端——消费者准备好接收数据则将标志信号 ready 置位。在时钟沿同时出现 valid 和 ready 置位，则完成数据传输。

Ready-Valid 接口是一种简单的控制流接口，包含：

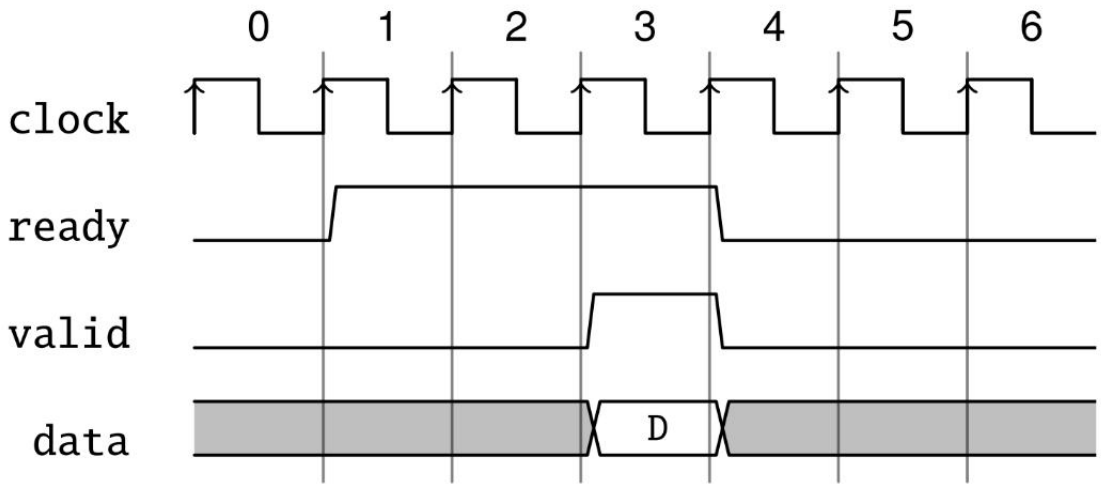
data：发送端向接收端发送的数据；

valid：发送端到接收端的信号，用于指示发送的数据是否有效；

ready：接收端到发送端的信号，用于指示是否可以接收数据；

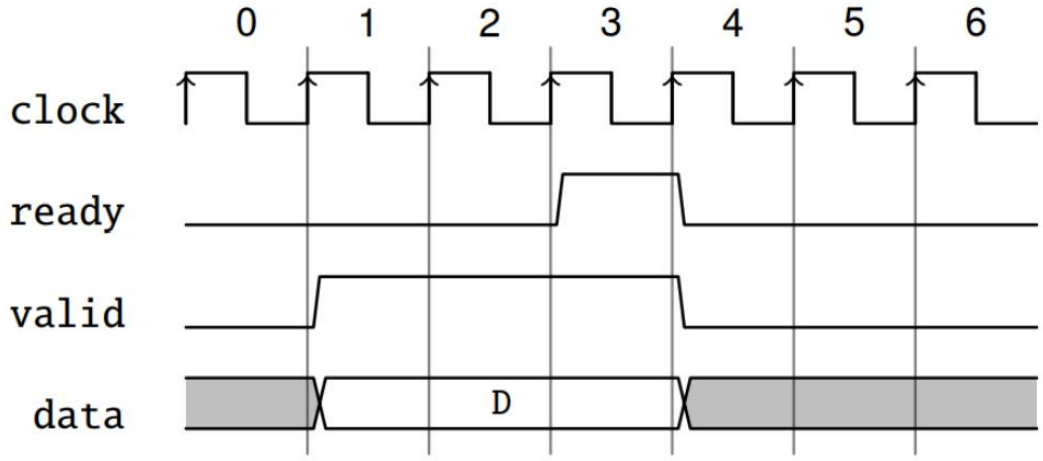



下图是当接收端在发送端准备好数据之前，就将 ready 置有效时（从第一个时钟周期开始）的时序图：



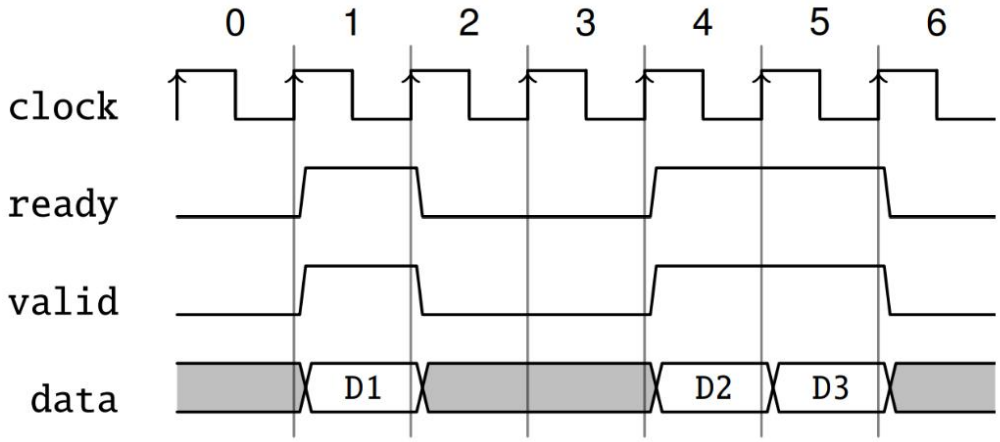

上图中，数据在第三个时钟周期准备好，同时 valid 置有效，此时 ready 信号和 valid 信号都被设置，数据传输进行。而在第四个时钟周期，发送端没有要发送的数据，接收端也没有准备好接受数据。如果接收端每个时钟周期都可以接受数据，ready 信号此时可以硬编码为 true。

下图是当发送端在接收端准备好接收数据之前，就将 valid 信号设置时（从第一个时钟周期开始）的时序图：




数据传输发生在第三个时钟周期，同样从第四个时钟周期开始，发送端没有要发送的数据，接收端也没有准备好接受数据。类比 always ready 接口，我们可以想到是不是有一个 always valid 接口。确实，不过这种情况下数据可能在给出 ready 信号的时候不改变，我们只能简单地丢弃这个握手信号。

下图是 Ready-Valid 接口的另一种变化的时序图：




上图中，在第一个时钟周期，ready 和 valid 同时被设置一个时钟周期，数据 D1 的传输也在这个周期进行。数据可以连续传输（每个时钟周期内），比如第四、第五个时钟周期内 D2 和 D3 的传输。

## 3.3. DecoupleIO 使用

Ready-Valid 握手协议是一种常用的通信方式，在 Chisel 中可以使用 DecoupleIO，定义如下：

```
1.  class DecoupledIO[T <: Data](gen: T) extends Bundle {
2.      val ready = Input(Bool())
3.      val valid = Output(Bool())
4.      val bits = Output(gen)
5.  }
```

使用它需要导入 util 包。这个 DecoupledIO 根据数据 data 的类型来参数化，用 DecoupledIO 封装后，可以通过接口的 bits 字段来访问 data。

这里包含了三个位域，ready、valid、和 bits，其中 ready 表示发送端到接收端的信号，用于指示发送的数据是否有效。ready 表示接收端到发送端的信号，用于指示是否可以接受数据，bits 是自定义的数据格式，表示发送端到接收端的数据内容。

发送端在 data 准备好之后就会设置 valid 信号，接收端在准备好接收一个字的数据的时候就会设置 ready 信号。数据的传输会在两个信号，valid 信号和 ready 信号，都被设置时才会进行。如果两个信号有任何一个没被设置，那就不会进行数据传输。

显然这个 DecoupledIO 接口是发送端的，而接收端的接口是完全反过来的，那么我们就需要使用 Chisel 中的另一个函数 Flipped 了。Flipped 函数可以将一个 Bundle 内的输入输出全都颠倒过来，所以如果一个发送端的 Ready-Valid 数据接口定义如下：

```
1.  val out = DecoupledIO(UInt(8.W))
```

那么接收端就需要翻转端口，使用前面提到的 Flipped 方法：

```
val in = Flipped(DecoupledIO(UInt(8.W)))
```

此时我们便可以直接通过<>连接两个端口

```
in <> out
```

这里无需区分左右和方向，Chisel 会自动匹配两个端口并将对应的 valid、ready 和 bits 位域进行连接，如果发生了类型不匹配，则 Chisel 将会直接报错。

## 3.4. Counter 的使用

计数器也是一个常用的硬件电路。在 Chisel.util 包里定义了一个自增计数器原语 Counter，它的 apply 方法是：apply(cond: Bool, n: Int): (UInt, Bool)，接收两个参数：

第一个参数是 Bool 类型的使能信号，为 true.B 时计数器从 0 开始每个时钟上升沿加 1 自增，为 false.B 时则计数器保持不变；

第二个参数需要一个 Int 类型的具体正数，当计数到 n 时归零。

该方法返回一个二元组：第一个元素是计数器的计数值，第二个元素是判断计数值是否等于 n 的结果。一般来说我们只用到第一个。

例如，如果我们想从 0 计数到 100，只在输入信号有效的时候才开始计数，并且在计数到 100 的时候才输出有效值，那么 Chisel 代码可以这样写：

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

class MyCounter extends Module {
  val io = IO(new Bundle {
    val en = Input(Bool())
    val out = Output(UInt(7.W))
    val valid = Output(Bool())
  })

  val (a, b) = Counter(io.en, 100)
  io.out := a
  io.valid := b
}
```

Counter 这个方法返回两个变量，分别赋值给 a 和 b，a 表示计数器的值，而 b 表示“计数器等于 100”这一逻辑判断的真假性。在 Counter 的第一个参数表示当 io.en 时，计数器才会向上累加，否则将保持原有的值不变。

Counter 的另外一个 apply 方法是 apply(n:Int)，n 表示计数的最大值，而计数器的增长则需要通过主动调用方法 inc()，实例代码如下：

```
class MyCounter extends Module {
  val io = IO(new Bundle {
    val en = Input(Bool())
    val out = Output(UInt(8.W))
    val valid = Output(Bool())
  })

  val cnt = Counter(100)
  when(io.en){
    cnt.inc()
  }
  io.out := cnt.value
  io.valid := cnt.value === 100.U
}
```

这段代码的实际效果与上述 Counter 的 apply 方法：apply(cond: Bool, n: Int): (UInt, Bool) 相同，此时如果想要让计时器增长需要主动调用 cnt.inc()，需要获取计数器的值需要调用 cnt.value

### 3.5. PopCount 使用

在硬件电路设计中常常需要计算某个信号中 1 的个数，Chisel 中有封装好的原语 PopCount，能够接受 Bits、UInt 或者 Seq 信号，最终返回输入信号中 1 的个数。

```
PopCount(Seq(true.B, false.B, true.B, true.B)) // 结果为3.U
PopCount(Seq(false.B, false.B, true.B, false.B)) // 结果为1.U
PopCount("b1011".U) // 结果为3.U
PopCount(myUIntWire) // 动态计数器
```

## 4.实验步骤

本次实验我们利用 valid 和 ready 的协议编写两个模块，一个用于发送请求（Sender），一个用于接受请求（PopCounter），其中 PopCounter 模块用于计算 Sender 发来的数据中 1 的个数。同时 PopCounter 模块并不能时刻工作，其内部有一个计数器，当计数器达到某一个值时将会拉低 ready 请求，这时候输出的结果也将无效。接下来我们分步实现：

首先实现 Sender：

```
class Sender extends Module{
  val io = IO(new Bundle(){
```

```
    val out = DecoupledIO(UInt(7.W))
  })

  val (cnt, _) = Counter(io.out.fire, 7)
  io.out.valid := true.B
  io.out.bits := cnt

}
```

对于 Sender，我们只有一个输出端口，用 DecoupleIO 定义，用于向其他模块发送请求。我们定义了一个计数器，cnt 用于接受计数器的返回值，而对于"计数器的值为 7"这一事件，我们并不关心，因此可以直接使用_作为一个匿名变量来接受，后续并不需要使用到。

这里注意计数器的递增条件为 io.out.fire，io.out.fire 的含义是 out 这个端口成功握手，也就是 io.out.valid 和 io.out.ready 同时为高。而在这个模块中，io.out.valid 始终为 true，则实际 fire 的值也就取决于外部模块的 ready 信号是否拉高。

接下来实现 PopCounter：

```
class PopCounter extends Module{
  val io = IO(new Bundle(){
    val in = Flipped(DecoupledIO(UInt(7.W)))
    val res = ValidIO(UInt(3.W))
  })

  val readyCnt = Counter(15)
  readyCnt.inc()

  io.in.ready := (readyCnt.value <= 10.U)
  io.res.valid := io.in.fire
  io.res.bits := PopCount(io.in.bits)
}
```

在这个模块中，我们用 io.in 接收外部的请求，由于是输入端口，因此需要使用 Flipped 来翻转 DecoupleIO 的端口（此时 io.in.ready 则变为了输出信号）。

我们首先定义一个计数器 readyCnt，这个计数器用来控制 ready 信号的高低，计数器每个周期都会增长，因此直接调用 inc()方法。

我们对于 io.in.ready 拉高的条件是，计数器的值小于 10，当计数器计满之后会溢出，则又会从 0 开始向上增长。

这个模块会计算输入信号中 1 的个数，因此使用了 PopCount 这一方法。io.res.valid 仅当输入端口与 Sender 握手时才有效，因此使用了 io.in.fire 方法。

最后编写一个顶层模块用于例化并连接两个子模块：

```
class PopCounterModule extends Module{
  val io = IO(new Bundle(){
    val res = ValidIO(UInt(3.W))
  })

  val sender = Module(new Sender)
  val receiver = Module(new PopCounter)

  receiver.io.in <> sender.io.out
  io.res := receiver.io.res
}
```

这个模块只有输出端口，我们分别例化两个模块 sender 和 receiver，将 receiver 的输入端口和 sender 的输出端口连接起来，并将最终的计算结果输出到 PopCounterModule 的顶层端口 io.res 中。

**任务一：**

- 使用 ready-valid 的握手协议将一个模块的数据传输到另一个模块
- 模块 DataSender 存储了 32 个位宽为 8bit 的数据；模块 DataReceiver 有相同大小的 32 个位宽为 8bit 的寄存器。模块 DataSender 和模块 DataReceiver 之间是用 ready 和 valid 信号进行通讯。
- 模块 DataReceiver 的 ready 信号会有规律拉低（在代码中由一个 readyCnt 计数器控制，当 readyCnt 被 3 除余 1 时，ready 信号才会拉高），需要将模块 DataSender 中的所有数据传输到模块 DataReceiver 中。
- 由于每 3 个周期 ready 会拉高一次，共 32 个数据，因此 100 个周期后数据将会检查数据是否完整。
- 通过运行以下命令，检测代码的正确性。

  ```
  mill MyChiselProject.test.testOnly exp8.todo.TestDataTrans
  ```

```
lc3@lc3-virtual-machine:~/chisel_exp$ mill MyChiselProject.test.testOnly exp8.todo.TestDataTrans
[50/83] MyChiselProject.compile
[info] compiling 1 Scala source to /home/lc3/chisel_exp/out/MyChiselProject/compile.dest/classes ...
[info] done compiling
[83/83] MyChiselProject.test.testOnly
TestDataTrans:
i = 0 sendData = 7 receiverData = 7
i = 1 sendData = 5 receiverData = 5
i = 2 sendData = 147 receiverData = 147
i = 3 sendData = 72 receiverData = 72
i = 4 sendData = 89 receiverData = 89
i = 5 sendData = 181 receiverData = 181
i = 6 sendData = 88 receiverData = 88
i = 7 sendData = 211 receiverData = 211
i = 8 sendData = 31 receiverData = 31
i = 9 sendData = 141 receiverData = 141
i = 10 sendData = 107 receiverData = 107
i = 11 sendData = 130 receiverData = 130
i = 12 sendData = 66 receiverData = 66
i = 13 sendData = 224 receiverData = 224
i = 14 sendData = 70 receiverData = 70
i = 15 sendData = 124 receiverData = 124
i = 16 sendData = 125 receiverData = 125
i = 17 sendData = 29 receiverData = 29
i = 18 sendData = 20 receiverData = 20
i = 19 sendData = 83 receiverData = 83
i = 20 sendData = 168 receiverData = 168
i = 21 sendData = 233 receiverData = 233
i = 22 sendData = 5 receiverData = 5
i = 23 sendData = 211 receiverData = 211
i = 24 sendData = 222 receiverData = 222
i = 25 sendData = 7 receiverData = 7
i = 26 sendData = 251 receiverData = 251
i = 27 sendData = 42 receiverData = 42
i = 28 sendData = 172 receiverData = 172
i = 29 sendData = 30 receiverData = 30
i = 30 sendData = 237 receiverData = 237
i = 31 sendData = 198 receiverData = 198
- dataSender should pass these test
```