
实验四：时序逻辑电路

1. 实验目的

- 1.1. 掌握时序逻辑的基础概念，了解时序图的相关概念；
- 1.2. 学习使用 Chisel 实现各种常见的寄存器；
- 1.3. 学习计数器电路的实现。

2. 实验内容

- 2.1. 时序逻辑概念与时序图介绍；
- 2.2. Chisel 中实现 D 触发器，带复位的 D 触发器，带使能的 D 触发器；
- 2.3. 使用 Chisel 的 Reg 语法搭建计数器电路。

3. 实验相关 Chisel 语法介绍

在本小节，我们将介绍接下来实验过程中使用到的 Chisel 语法。

3.1. Reg/RegInit

在 Chisel 中，Reg 和 RegInit 用于定义和初始化寄存器。寄存器是一个存储单元，可以在时钟边沿存储和输出数据。

Reg 用于定义一个寄存器，可以存储一个指定类型的数据，但在定义时不会初始化寄存器的值，需要在电路运行过程中进行赋值。例如：

```
1. val myReg = Reg(UInt(8.W))
```

在这个例子中，myReg 是一个 8 位宽的无符号整数寄存器。

RegInit 用于定义并初始化寄存器，它在定义寄存器的同时为寄存器赋予一个初始值。例如：

```
1. val myRegInit = RegInit(0.U(8.W))
```

在这个例子中，myRegInit 是一个 8 位宽的无符号整数寄存器，并且初始值为 0。

Reg 和 RegInit 的区别主要体现在两个地方：

1) 初始化：Reg 定义的寄存器在初始化时没有指定值，而 RegInit 定义的寄存器在初始化时有一个指定的初始值。

2) 使用场景：Reg 适用于需要在电路运行过程中动态赋值的场景，而 RegInit 适用于需要在电路初始化时指定初始值的场景。

3.2. RegEnable / RegNext

RegEnable 用于定义一个带使能信号的寄存器。使能信号用于控制寄存器的更新，只有当使能信号为真时，寄存器才会在时钟边沿更新其值（注意：Chisel 中默认的有效时钟边沿是上升沿）。

```
1. val enable = true.B
2. val myRegEnable = RegEnable(next = 3.U, init = 0.U, enable = enable)
```

在这个例子中，myRegEnable 是一个带使能信号的寄存器，初始值为 0，当 enable 为真时，寄存器更新值为 3。

RegNext 用于创建一个时钟周期延迟的寄存器，它将当前周期的输入值在下一个时钟周期输出。

```
1. val myRegNext = RegNext(next = 3.U, init = 0.U)
```

在这个例子中，myRegNext 是一个寄存器，初始值为 0，每个时钟周期将输入值 3 存储并在下一个时钟周期输出。

RegEnable 和 RegNext 的区别主要体现在：RegEnable 需要一个使能信号来控制寄存器是否更新，适用于需要在特定条件下更新寄存器的场景。RegNext 则是实现时钟周期延迟，不需要使能信号，每个时钟周期都会更新寄存器的值。

3.3. log2Ceil

log2Ceil 是 Chisel 中的一个实用函数，用于计算一个给定整数的以 2 为底的对数，并向上取整。这个函数在设计硬件时尤其有用，因为它可以帮助确定存储某个最大值所需的最小位宽。它位于 chisel3.util 这个 package 中，需要 import chisel3.util._ 来导入。

例如，如果我们需要计数到 15，那么计数器的位宽至少要是 4 位，因为 $2^4 = 16$ 是大于或等于 15 的最小 2 的幂。log2Ceil 可以帮助我们自动计算出这个位宽，而不需要手动计算。对应到下面的代码中，如果 value=15，那么 width=4。

```
1. val width = log2Ceil(value)
```

4. 实验步骤

4.1. 时序逻辑电路介绍

在本节实验中，我们将介绍时序（Sequential）逻辑电路。时序逻辑电路与组合逻辑电路不同，时序逻辑电路的输出不仅取决于当前的输入值，还依赖于先前的输入值。因此，时序逻辑具有记忆功能，这意味着它们能存储状态信息。这种记忆功能使得时序逻辑电路在实现复杂功能时非常有用，例如计数器、寄存器和状态机等。

时序逻辑电路的基本构建单元是触发器（flip-flop）和锁存器（latch）。触发器是一种双

稳态器件，它具有两个稳定状态，可以存储一位二进制信息。常见的触发器类型包括 D 触发器（DFF, D Flip-Flop）、T 触发器、JK 触发器和 SR 触发器。触发器通常在时钟信号的控制下工作，时钟信号决定了触发器状态变化的时间点。

在时序逻辑电路中，时钟信号（clock signal）起着至关重要的作用。时钟信号是一种周期性的方波信号，它为整个电路提供了同步基准。在每个时钟周期中，触发器根据输入信号和当前状态更新其状态，从而实现时序逻辑的功能。

我们首先来看下一个最常见的 D 触发器的例子，图 1 展示的是一个 D 触发器的电路图，其内部由若干逻辑门和反馈连接构成。D 触发器有两个主要输入端：数据输入端（D, Data）和时钟输入端（CLK），以及两个输出端：正常输出（Q）和反相输出（Q'）。当时钟信号的上升沿（或下降沿，具体取决于触发器的类型）到来时，D 触发器将数据输入端 D 的值锁存到输出端 Q，并保持该值直到下一次时钟沿到来。具体来说，D 触发器内部通常包含两个锁存器，分别称为主锁存器和从锁存器。主锁存器在时钟信号为低电平时对数据进行采样，而从锁存器在时钟信号为高电平时将主锁存器的值传递到输出端。这种主从结构确保了 D 触发器在时钟信号的一个完整周期内只进行一次状态变化，从而避免了输入信号的毛刺（glitch）对电路的影响。

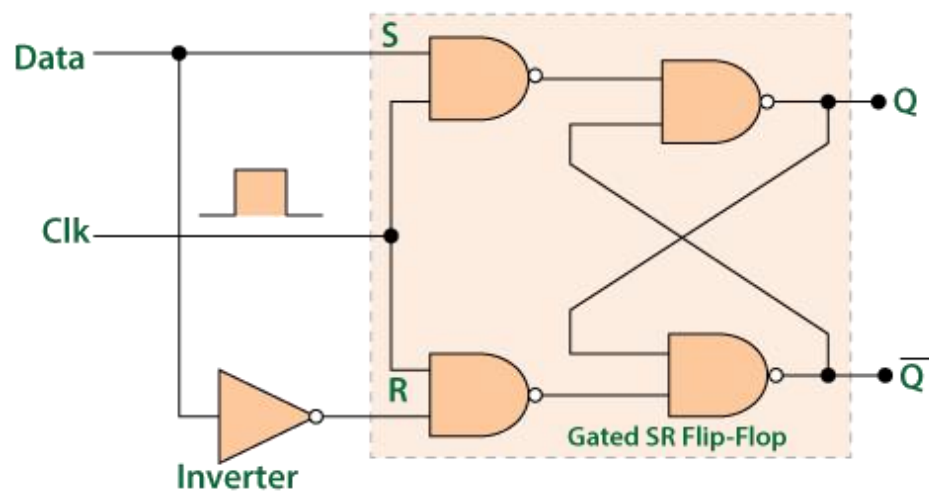


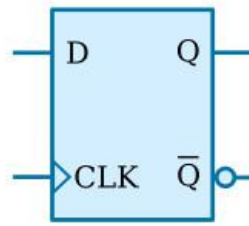
图 1 D 触发器的电路图

D 触发器的行为可以通过图 2（B）的真值表（truth table）和图 2（C）的时序图（timing diagram）来描述。在真值表中，我们列出了 D 触发器在不同输入条件下的输出状态，例如，当 D 为 1 且时钟信号出现上升沿时，输出 Q 将变为 1；当 D 为 0 且时钟信号出现上升沿时，输出 Q 将变为 0。时序图则展示了 D 触发器在连续时钟周期内的输入输出关系，通过时序图，我们可以直观地理解 D 触发器的工作过程。

接下来我们仔细讲解下图 2（C）的 D 触发器的时序图，在此我们假设此 D 触发器在接收

到时钟上升沿的时候才会变化。首先在 t_0 时刻，此时 CLK 是一个稳定的低电平，根据 (B)，此时 Q 保持不变； t_1 时刻，此时 CLK 是一个上升沿， $D=1$ ，根据 (B)，在 CLK 之后 Q 就变为了 1，此时 D 触发器也就成功保存了输入的 $D=1$ 这个值； t_2 时刻是一个下降沿，Q 不会有任何变化，Q 还是保持 1； t_3 时刻，是第二个上升沿，此时 D 的值为 0，那么 Q 的值会在这个边沿之后变为了 0； t_4 时刻也是一个下降沿，Q 不会有任何变化； t_5 时刻是第三个上升沿，D 的值是 0，因此 Q 还是为 0；一直到 t_7 这第四个上升沿，此时 $D=1$ ，因此 Q 在上升沿之后也就变为了 1。

通过对这一时序图的分析，我们可以发现 D 触发器的输出 Q 在每一个时钟上升沿时都会更新为输入 D 的值，而在时钟的其他部分，如下降沿和稳定的高或低电平期间，输出 Q 则保持不变。这种行为展示了 D 触发器在时钟信号的控制下如何准确地锁存输入数据，并在预定的时刻更新输出。进一步分析 D 触发器的时序图，我们还可以理解到建立时间和保持时间的重要性。建立时间是指在时钟信号的有效边沿到来之前，数据输入必须保持稳定的最小时间；保持时间是指在时钟信号的有效边沿之后，数据输入必须继续保持稳定的最小时间。这些时间参数确保 D 触发器能够正确地采样和锁存数据，避免由于数据变化过快而导致的错误。



(A)

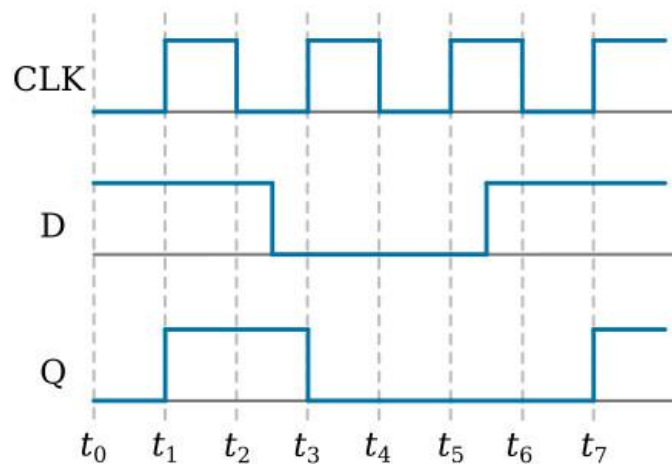
CLK	D	Q_{next}	Comment
Rising edge	0	0	Store 0
Rising edge	1	1	Store 1
Non-rising	X	Q	No change

Q_{next} - "after the clock transition" output

Q - the current output

X - the signal is irrelevant

(B)



(C)

图 2 D 触发器的符号图 (A)、真值表 (B)、时序图 (C)

4.2. Chisel 中的时序逻辑电路

带复位的 D 触发器

在上一小节中，我们介绍了时序逻辑电路的基本概念，以及时序逻辑电路的一个常见组成单元 D 触发器，分析了其时序图，从而对时序电路有了基本的认识，接下来就可以开始回归到如何在 Chisel 中实现时序逻辑电路这一话题上。

首先我们需要介绍一下寄存器 Reg 的概念，在 Chisel 中，Reg 用于表示表示一个具有记忆功

能的时序逻辑单元，这和我们前面介绍的 D 触发器非常相似，不同的地方在于 Chisel 中的 Reg 是带 Reset 的，这意味着在系统复位时，寄存器可以被初始化到一个特定的值，从而确保电路在启动时处于已知状态，而我们前面介绍的 D 触发器是不具备初始值的，另外在这里有必要介绍一下 Reset 信号，Reset 信号在时序逻辑电路中起到了至关重要的作用，它用于将电路复位到一个已知的初始状态，否则此时电路的行为是无法确定的，我们无法确定下一个时钟上升沿来的时候具体的时序电路的逻辑是怎么样的。所以我们可以将 Reg 理解为是一种带了 Reset 的 D 触发器，如图 3 所示，Reset 也分为了同步 Reset (synchronous reset) 和异步复位 (asynchronous reset)，同步复位和异步复位的区别在于同步复位和异步复位的区别在于同步复位是在时钟信号的控制下进行的，也就是说复位信号只有在时钟的上升沿或下降沿到来时才会起作用，而异步复位则不依赖于时钟信号，可以在任何时候对电路进行复位。

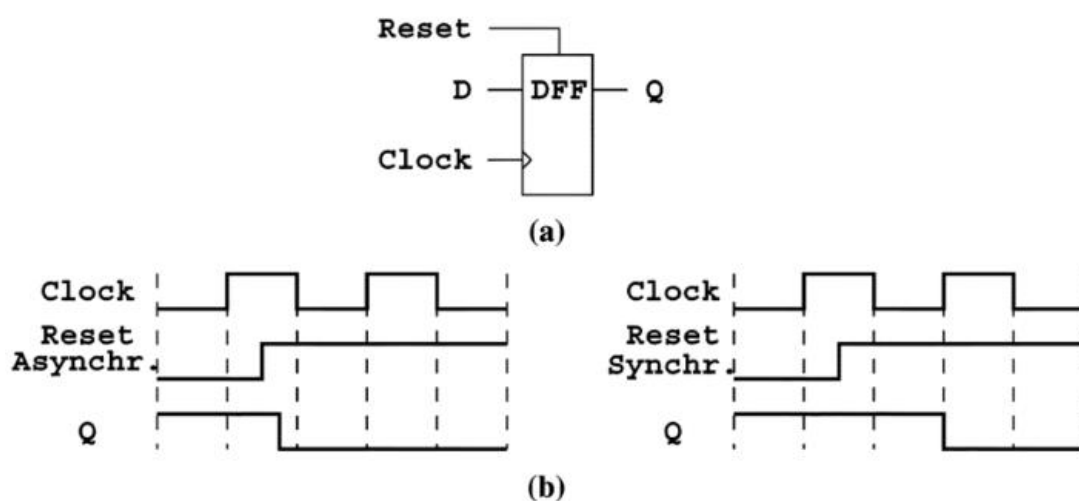


图 3 带复位的 D 触发器

在 Chisel 中，我们可以通过直接使用 Reg，例如下面的 Chisel 代码中我们创建了一个简单的模块来实现带同步复位的 D 触发器的功能：

src/main/scala/exp4/SyncResetDFF.scala

```
1. package exp4
2.
3. import chisel3._
4.
5. class SyncResetDFF extends Module {
6.   val d = IO(Input(Bool()))
7.   val q = IO(Output(Bool()))
8.
9.   val qReg = Reg(Bool())
10.
11.   when(reset.asBool) {
12.     qReg := false.B
```

```

13.   }.otherwise {
14.     qReg := d
15.   }
16.
17.   q := qReg
18. }

```

上述代码中，在低 11 行我们在复位的时候将 qReg 的值赋值为 false，也就是复位的时候 qReg 的值为 0，在没有复位的情况，也就是第 14 行的位置，我们将 qReg 的值赋值为输入信号 d，这样我们在每一个 clock 上升沿到来的时候都会捕捉输入信号 d 的值，同时我们在第 17 行的地方将 qReg 的值连接到了端口 q 中，观察上述代码后还可以发现，我并没有显式定义 reset 和 clock 信号。实际上，在 Chisel 中，所有 extends (Scala 的继承) 了 Module 的模块都会自动包含 clock 和 reset 信号，这些信号由顶层模块隐式地传递。因此，我们可以在 when 语句中直接使用 reset 信号，而无需显式声明。为了证明这一点，我们可以查看生成的 Verilog 代码：

```

1.  module SyncResetDFF(
2.    input clock,
3.    input reset,
4.    input d,
5.    output q
6.  );
7.
8.    reg qReg;
9.    always @(posedge clock) begin
10.     if (reset)
11.       qReg <= 1'h0;
12.     else
13.       qReg <= d;
14.    end // always @(posedge)
15.    assign q = qReg;
16. endmodule

```

从上面的 Verilog 代码可以看到，reset 和 clock 信号实际上是存在的，我们在 Chisel 中是可以直接使用的 reset 或者 clock 信号而不用显示定义（再次注意这里有一个前提是你的模块 extends 了 Module，如果 extends 的是 RawModule，那么 reset 和 clock 是需要自行定义的）。

可以切换到实验配套项目根目录，在 src/main/scala/exp4/SyncResetDFF.scala 中看到这个代码的内容，我们可以通过下面的命令来执行测试来验证这个模块的功能正确性：

```
mill MyChiselProject.test.testOnly exp4.TestSyncResetDFF
```

```

lc3@lc3-virtual-machine:~/chisel_exp$ mill MyChiselProject.test.testOnly exp4.TestSyncResetDFF
[76/83] MyChiselProject.test.compile
[info] compiling 1 Scala source to /home/lc3/chisel_exp/out/MyChiselProject/test/compile.dest/classes ...
[info] done compiling
[83/83] MyChiselProject.test.testOnly
TestSyncResetDFF:
Initial state: d = 0, reset = 1, q = 0
After reset: q = 0
After releasing reset: d = 0, reset = 0, q = 0
After applying d = true: q = 1
After applying d = false: q = 0
After applying reset again: q = 0
- SyncResetDFF should correctly reset and latch data

```

在上面的 Chisel 代码中，我们在 reset 的时候给 dReg 赋值了一个初始值，实际上在 Chisel 中有更简单清晰的方式来实现 reset 复位的时候赋值寄存器的值，我们可以使用 RegInit 来实现，RegInit 的语法定义如下：

```

1.  /** Construct a [[Reg]] from a type template initialized to the specified value on reset
2.  * @param t The type template used to construct this [[Reg]]
3.  * @param init The value the [[Reg]] is initialized to on reset
4.  */
5.  def apply[T <: Data](t: T, init: T)(implicit sourceInfo: SourceInfo): T
6.
7.  /** Construct a [[Reg]] initialized on reset to the specified value.
8.  * @param init Initial value that serves as a type template and reset value
9.  */
10. def apply[T <: Data](init: T)(implicit sourceInfo: SourceInfo): T

```

有两种使用方式，第一种方式为第 5 行的这一种方式，RegInit 接受两个参数，一个是 t 也就是 Reg 的数据类型，另一个是 init 初始值，这里的初始值就是在 reset 信号到来的时候进行赋值的一个值，也就是 SyncResetDFF 这个例子中 when(reset)里我们进行赋值的内容，根据这个定义，我们可以改造上面的 SyncResetDFF，得到如下的 Chisel 代码：

src/main/scala/exp4/SyncResetDFF_1.scala

```

1.  package exp4
2.
3.  import chisel3._
4.
5.  class SyncResetDFF_1 extends Module {
6.    val d = IO(Input(Bool()))
7.    val q = IO(Output(Bool()))
8.
9.    val qReg = RegInit(Bool(), false.B)
10.
11.    qReg := d
12.
13.    q := qReg
14.  }
15.

```


这里需要能够理解 `Bool()` 是一个类型，而 `false.B` 是这个类型的具体值，同样的例子还有 `UInt(3.W)` 类型可以有 `3.U(3.W)` 这个具体的值。接下来我们看到 `RegInit` 语法定义中第 10 行的使用方式，这里更进一步，只需要提供一个 `init` 初始值就可以创建一个具有 `reset` 赋值的寄存器了，这是因为 Chisel 可以通过 `init` 的这个具体的值来推断出类型，就比如可以从 `false.B`、`true.B` 中推断出是 `Bool` 类型，从 `3.U` 中推断出 `UInt` 类型。我们同样可以改造 `SyncResetDFF`，得到如下的代码：

src/main/scala/exp4/SyncResetDFF_2.scala

```
1. package exp4
2.
3. import chisel3._
4.
5. class SyncResetDFF_2 extends Module {
6.   val d = IO(Input(Bool()))
7.   val q = IO(Output(Bool()))
8.
9.   val qReg = RegInit(false.B)
10.
11.   qReg := d
12.
13.   q := qReg
14. }
```

值得注意的是，在我们实际写 Chisel 代码中，最常用到的就是上述这种方式的 `RegInit`，因为这个用法通常更方便简洁。

上述两个代码我们可以将其生成 Verilog，如下：

```
1. module SyncResetDFF_2(
2.   input clock,
3.   input reset,
4.   input d,
5.   output q
6. );
7.
8.   reg qReg;
9.   always @(posedge clock) begin
10.     if (reset)
11.       qReg <= 1'h0;
12.     else
13.       qReg <= d;
14.   end // always @(posedge)
15.   assign q = qReg;
```

16. endmodule

对比 SyncResetDFF 生成的 Verilog，可以发现是一致的，这说明这三种用法是等价的。

带使能的 D 触发器

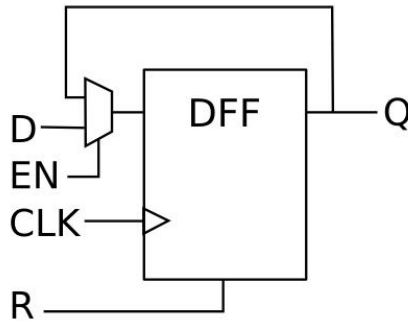


图 4 带使能端的 D 触发器

在上一小节中，我们的 D 触发器在每一个 clock 上升沿都会锁存输入值 d 到寄存器中，但是有时候我们不希望这样子，而是希望在满足某一逻辑条件的时候（例如一个 EN 信号为 true 的时候）才锁存输入值 d 到寄存器中，那么这时候就可以使用带使能的 D 触发器来实现这个功能。带使能的 D 触发器的逻辑框图如图 4 所示，相比于带复位的 D 触发器，我们增加了一个 EN 使能信号，根据这个图我们可以得知当 EN 信号为高电平（true）时，D 触发器会在时钟的上升沿将输入值 d 锁存到寄存器中。而当 EN 信号为低电平（false）时，寄存器的值将保持不变，不会更新为输入值 d。接下来我们首先使用最基本的 RegInit 和 when 语句来实现一个简单的带使能的 D 触发器，以下是实现代码：

src/main/scala/exp4/EnableDFF.scala

```
1. package exp4
2.
3. import chisel3._
4. import chisel3.util._
5.
6. class EnableDFF extends Module {
7.   val io = IO(new Bundle {
8.     val d = Input(Bool())
9.     val en = Input(Bool())
10.    val q = Output(Bool())
11.  })
12.
13.   val qReg = RegInit(false.B)
14.
15.   when(io.en) {
```

```

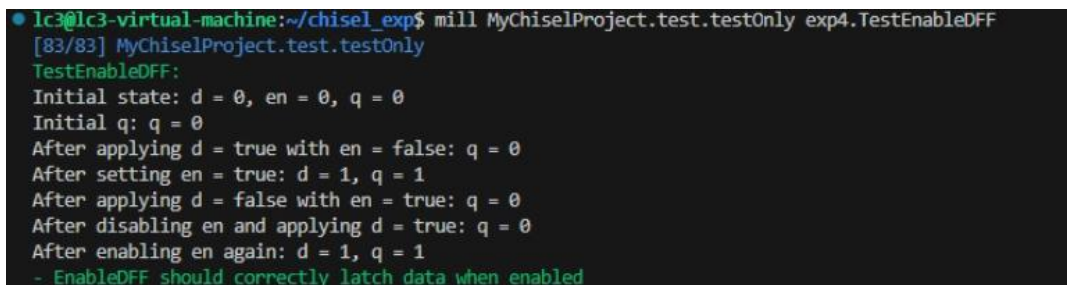
16.   qReg := io.d
17. }
18.
19. io.q := qReg
20. }

```

在这个实现中，我们定义了一个带使能的 D 触发器 EnableDFF，它有三个输入端口和一个输出端口。输入端口包括数据输入 d、使能信号 en 以及输出端口 q。我们使用 RegInit 初始化寄存器 qReg 为 false.B，然后在 when 语句中检查 en 信号。如果 en 为高电平，则将输入 d 的值锁存到 qReg 中，否则寄存器的值保持不变。最后，将寄存器的值赋给输出 q。

可以切换到实验配套项目根目录，在 src/main/scala/exp4/EnableDFF.scala 中看到这个代码的内容，我们可以通过下面的命令来执行测试来验证这个模块的功能正确性：

```
mill MyChiselProject.test.testOnly exp4.TestEnableDFF
```



```

● lc3@lc3-virtual-machine:~/chisel_exp$ mill MyChiselProject.test.testOnly exp4.TestEnableDFF
[83/83] MyChiselProject.test.testOnly
TestEnableDFF:
Initial state: d = 0, en = 0, q = 0
Initial q: q = 0
After applying d = true with en = false: q = 0
After setting en = true: d = 1, q = 1
After applying d = false with en = true: q = 0
After disabling en and applying d = true: q = 0
After enabling en again: d = 1, q = 1
- EnableDFF should correctly latch data when enabled

```

对比上一小节中的 SyncResetDFF_2 这个例子，可以发现 SyncResetDFF_2 其实就是 EnableDFF 在使能信号 io.en 一直为 true 时候的一个特例，这时候这个 SyncResetDFF_2 的功能类似每一个时钟周期都会锁存输入信号 d 到寄存器中，而不会有条件地阻止更新。在 Chisel 中，我们可以使用一个更简洁的方式来实现这种行为，即使用 RegNext。RegNext 会在每个时钟周期自动将输入信号的当前值锁存到寄存器中，而无需额外的使能信号。下面是一个使用 RegNext 实现的示例：

src/main/scala/exp4/SyncResetDFF_3.scala

```

1. package exp4
2.
3. import chisel3._
4. import chisel3.util._
5.
6. class SyncResetDFF_3 extends Module {
7.   val io = IO(new Bundle {
8.     val d = Input(Bool())
9.     val q = Output(Bool())
10.  })
11.
12. // 使用 RegNext 将 d 的值锁存到寄存器中
13. val qReg = RegNext(io.d, init = false.B)

```

```

14.
15. io.q := qReg
16. }

```

上面这个例子中，RegNext 可以在第二个参数指定 init 的值，但是实际上这个值也可以不指定，也就是说 init 是可选的，上述代码实现的功能和 SyncResetDFF_2 是完全一样的。

回到 EnableDFF 这个代码中，EnableDFF 这个数字逻辑组件是很常见的，就像上面提到的其他 Reg 语法一样，Chisel 也给带使能端的 DFF 提供了一个用于提高代码简洁性和可读性的硬件原语：RegEnable，它的语法定义如下：

```

1. /** Returns a register with the specified next, update enable gate, and no reset initialization.
2.  *
3.  * @example {{{
4.  * val regWithEnable = RegEnable(nextVal, ena)
5.  * }}}
6.  */
7. def apply[T <: Data](next: T, enable: Bool): T
8.
9. /** Returns a register with the specified next, update enable gate, and reset initialization.
10.  *
11.  * @example {{{
12.  * val regWithEnableAndReset = RegEnable(nextVal, 0.U, ena)
13.  * }}}
14.  */
15. def apply[T <: Data](next: T, init: T, enable: Bool): T

```

第 7 行的用法中，RegEnable 可以接收两个参数，一个是 next，也就是在使能信号为高时要锁存的数据值，另一个是 enable 用来控制是否锁存 next 数据。而在第 15 行的用法中，RegEnable 可以接收三个参数，第一个是 next，在使能信号为高时要锁存的数据值，第二个是 init，用于初始化寄存器的值，第三个还是 enable 用来控制是否锁存 next 数据。通过这种方式，我们可以在寄存器初始状态和使能信号控制下，实现带有初始值的 D 触发器。

通过 RegEnable，我们可以简化 EnableDFF 从而得到如下的代码：

src/main/scala/exp4/EnableDFF_1.scala

```

1. package exp4
2.
3. import chisel3._
4. import chisel3.util._
5.
6. class EnableDFF_1 extends Module {
7.   val io = IO(new Bundle {

```

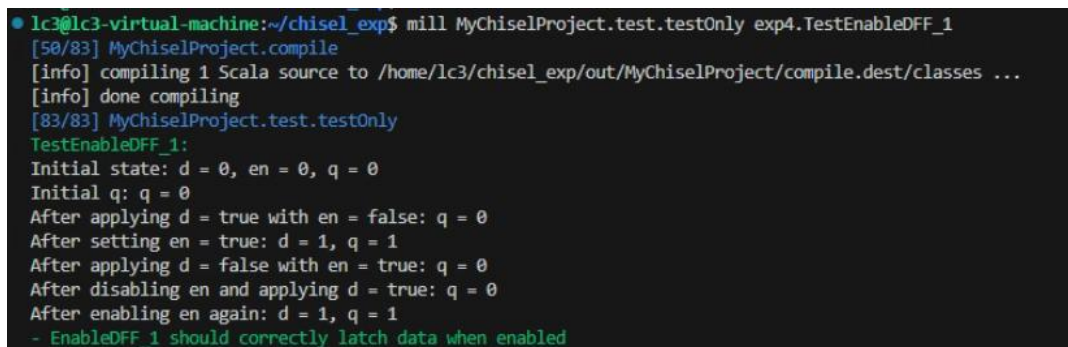
```

8.   val d = Input(Bool())
9.   val en = Input(Bool())
10.  val q = Output(Bool())
11. })
12.
13.  val qReg = RegEnable(io.d, false.B, io.en)
14.
15.  io.q := qReg
16. }

```

可以切换到实验配套项目根目录，在 `src/main/scala/exp4/EnableDFF_1.scala` 中看到这个代码的内容，我们可以通过下面的命令来执行测试来验证这个模块的功能正确性：

```
mill MyChiselProject.test.testOnly exp4.TestEnableDFF_1
```



```

● lc3@lc3-virtual-machine:~/chisel_exp$ mill MyChiselProject.test.testOnly exp4.TestEnableDFF_1
[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
TestEnableDFF_1:
Initial state: d = 0, en = 0, q = 0
Initial q: q = 0
After applying d = true with en = false: q = 0
After setting en = true: d = 1, q = 1
After applying d = false with en = true: q = 0
After disabling en and applying d = true: q = 0
After enabling en again: d = 1, q = 1
- EnableDFF_1 should correctly latch data when enabled

```

至此，我们就学会了 `Reg`、`RegInit`、`RegNext`、`RegEnable` 等常用寄存器语法的使用，我们将来的 Chisel 设计中将会经常使用到。

4.3.计数器实现

在学习了 Chisel 的 `Reg` 相关语法后，我们就能开始设计一些简单的时序逻辑模块，例如计数器，计数器顾名思义就是能够对某个数值进行递增或递减操作的电路。计数器在数字电路中应用广泛，比如用于分频、时间间隔测量、事件计数等。

我们首先来设计一个最基本的递增计数器，每个时钟周期计数值加一。我们可以使用 `RegInit` 来定义一个带有初始值的寄存器，并在每个时钟周期更新它的值。根据我们的设想，我们可以写出如下的 Chisel 代码：

`src/main/scala/exp4/SimpleCounter.scala`

```

1.  package exp4
2.
3.  import chisel3._
4.  import chisel3.util._
5.

```

```

6. class SimpleCounter(val max: Int) extends Module {
7.   val io = IO(new Bundle {
8.     val out = Output(UInt(log2Ceil(max).W))
9.   })
10.
11.   // 定义一个寄存器，并初始化为 0
12.   val countReg = RegInit(0.U(log2Ceil(max).W))
13.
14.   // 每个时钟周期计数值加一
15.   countReg := countReg + 1.U
16.
17.   // 当计数值达到最大值时，重置为 0
18.   when(countReg === (max - 1).U) {
19.     countReg := 0.U
20.   }
21.
22.   // 将计数值输出
23.   io.out := countReg
24. }

```

在这个 SimpleCounter 模块中，我们定义了一个名为 countReg 的寄存器，并将其初始化为 0。每个时钟周期，寄存器的值都会加一。当计数值达到最大值时，我们将其重置为 0。计数器的当前值通过 io.out 输出。

需要注意的是，log2Ceil 是 Chisel 中的一个实用函数，用于计算一个给定整数的以 2 为底的对数，并向上取整。这个函数在设计硬件时尤其有用，因为它可以帮助确定存储某个最大值所需的最小位宽。如果我们将这个计数器的计数值画出来，我们就能得到这样一个折线图（类似一个锯齿线段图）：

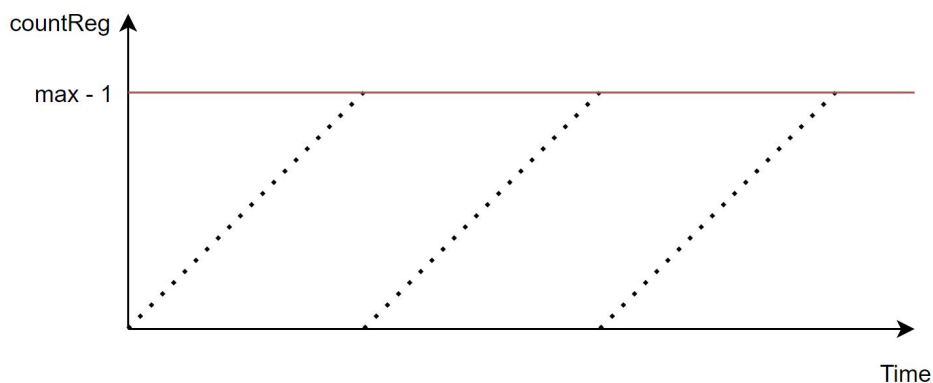


图 5 countReg 的数值变化图

可以切换到实验配套项目根目录，在 src/main/scala/exp4/SimpleCounter.scala 中看到这个代码的内容，我们可以通过下面的命令来执行测试来验证这个模块的功能正确性：

```
mill MyChiselProject.test.testOnly exp4.TestSimpleCounter
```

```
● lc3@lc3-virtual-machine:~/chisel_exp$ mill MyChiselProject.test.testOnly exp4.TestSimpleCounter
[83/83] MyChiselProject.test.testOnly
TestSimpleCounter:
maxCount is 4
Initial count: 0
Count at step 1: 1
Count at step 2: 2
Count at step 3: 3
Count after maxCount steps: 0
- SimpleCounter should correctly count up to max and reset
```

级联的寄存器

到目前为止我们都是在一个模块中只定义并使用到了一个寄存器，实际上复杂的时序逻辑设计往往包括了多个寄存器的组合使用，比如一个寄存器的输出可以作为另一个寄存器的输入，例如我们可以实现一个这样的小模块：当计数器计数到 10 和 20 的时候，让一个标志寄存器翻转一次数值（或者说取反），并将这个标志寄存器的值输出出来，下面是一个实现这个功能的小模块的 Chisel 代码：

src/main/scala/exp4/CounterWithFlag.scala

```
1. package exp4
2.
3. import chisel3._
4. import chisel3.util._
5.
6. class CounterWithFlag(val counterMax: Int) extends Module {
7.   val io = IO(new Bundle {
8.     val flagOut = Output(Bool())
9.   })
10.
11.   // 计数器寄存器
12.   val counter = RegInit(0.U(log2Ceil(counterMax).W))
13.
14.   // 标志寄存器
15.   val flag = RegInit(false.B)
16.
17.   // 计数器每个时钟周期加一
18.   counter := counter + 1.U
19.
20.   // 当计数器达到 10 或 20 时，标志寄存器翻转
21.   when(counter === 10.U || counter === 20.U) {
22.     flag := ~flag
23.   }
24.
25.   // 当计数器达到最大值时，回绕到 0
```

```

26. when(counter === (counterMax - 1).U) {
27.   counter := 0.U
28. }
29.
30. // 将计数器值和标志寄存器值输出
31. io.flagOut := flag
32. }

```

在这个 CounterWithFlag 模块中，我们定义了一个计数器和一个标志寄存器。当计数器达到 10 或 20 时，标志寄存器翻转其值，并将其值输出。我们同样可以画出这个 flagOut 信号的折线图，这样可以更直观的展示效果：

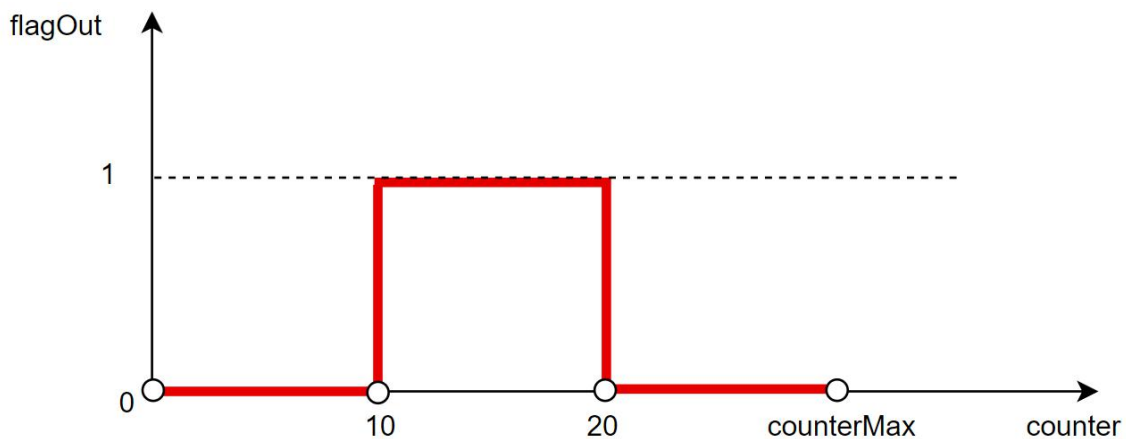


图 6 flagOut 的数值变化图

在上面这个图中，我们可以看到 flagOut 的输出信号值类似一个方波，我们可以看到中间 flagOut 拉高（处于高电平）的时间是 10 个 Counter 的计数值，我们可以用占空比来衡量方波的高电平占整个计数周期的比例，占空比的计算公式为：占空比 = 高电平时间 / 总时间，对于图 6，则占空比为 $10 / \text{counterMax}$ ，如果我们的 counterMax 为 50，那么占空比也就是 $10/50 = 20\%$ 。进一步地，可以发现 flagOut 在 counter 的计数值为 10 才开始拉高，并且在计数值为 20 时再次拉低。这种现象引出了一个重要的概念：相位。相位表示信号的时间偏移量。对于一个周期性的方波信号，相位可以描述信号在一个周期内相对于参考点的时间位置。用数学术语来说，相位可以表示为信号的时间偏移量占整个周期的比例。相位的计算公式为：相位 = 时间偏移 / 周期时间，代入 flagOut 这个例子，那么相位的计算方式就是：相位 = 高电平开始时间 / counterMax，假设此时 counterMax 是 40，那么相位 = $10 / 40 = 25\%$ ，我们可以用百分数来表示相位，也可以用度数来表示，一个完整的周期就是 2π 。

同样地，可以切换到实验配套项目根目录，在 src/main/scala/exp4/CounterWithFlag.scala 中看到这个代码的内容，我们可以通过下面的命令来执行测试来验证这个模块的功能正确性：

```
mill MyChiselProject.test.testOnly exp4.TestCounterWithFlag
```

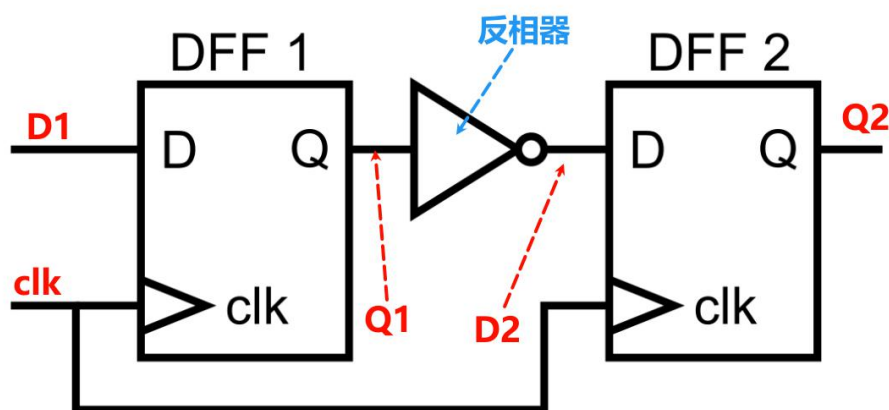


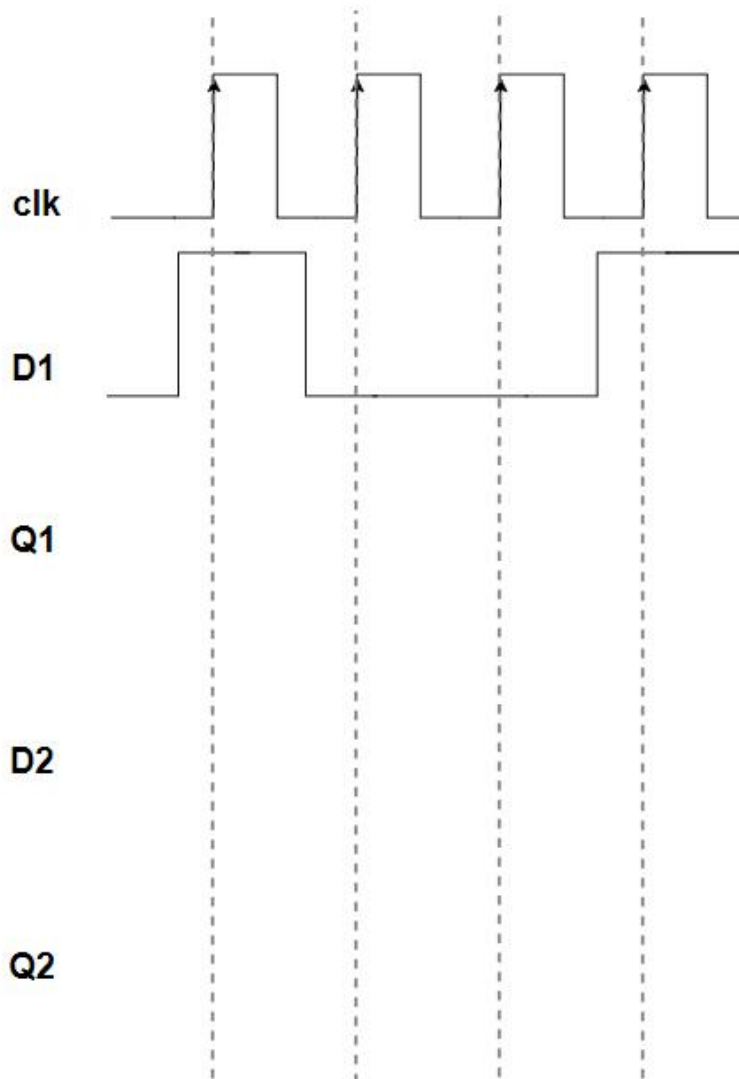
```

● lc3@lc3-virtual-machine:~/chisel_exp$ mill MyChiselProject.test.testOnly exp4.TestCounterWithFlag
[83/83] MyChiselProject.test.testOnly
TestCounterWithFlag:
maxCount is 30
Initial flag: 0
Counter at step 0: flagOut = 0
Counter at step 1: flagOut = 0
Counter at step 2: flagOut = 0
Counter at step 3: flagOut = 0
Counter at step 4: flagOut = 0
Counter at step 5: flagOut = 0
Counter at step 6: flagOut = 0
Counter at step 7: flagOut = 0
Counter at step 8: flagOut = 0
Counter at step 9: flagOut = 0
Counter at step 10: flagOut = 1
Counter at step 11: flagOut = 1
Counter at step 12: flagOut = 1
Counter at step 13: flagOut = 1
Counter at step 14: flagOut = 1
Counter at step 15: flagOut = 1
Counter at step 16: flagOut = 1
Counter at step 17: flagOut = 1
Counter at step 18: flagOut = 1
Counter at step 19: flagOut = 1
Counter at step 20: flagOut = 0
Counter at step 21: flagOut = 0
Counter at step 22: flagOut = 0
Counter at step 23: flagOut = 0
Counter at step 24: flagOut = 0
Counter at step 25: flagOut = 0
Counter at step 26: flagOut = 0
Counter at step 27: flagOut = 0
Counter at step 28: flagOut = 0
Counter at step 29: flagOut = 0
Counter after maxCount steps: 0
- CounterWithFlag should correctly toggle flag at 10 and 20 and reset counter

```

任务一：完成下面时序图的绘制，你需要补充 Q1，D2，Q2 三个信号的波形（寄存器的初始值都为 0）





任务二：使用计数器实现一个方波信号输出器，使得能够产生 10% 占空比的方波，相位偏移为 30%，计数器计数最大值限制为 100。

提示：请仔细查看 CounterWithFlag 这个例子的实验讲义内容，该任务所有所有涉及的知识点都有所提及。

根据 `src/main/scala/exp4/CounterWithFlag.scala` 的代码，如果我们想把计数器的最大值限制为 1000，那么可以将 12 行的 `counterMax` 直接赋值为 1000，在此基础上如果我们想要生成相位偏移为 30%，占空比为 10% 且计数器最大值为 1000 的方波输出器，则可以把第 21 行的 `counter === 10.U || counter === 20.U` 改成 `counter === 300.U || counter === 400.U` 即可。

代码框架如下：

实验框架代码位于 `src/main/scala/exp4/todo/SquareWave.scala` 中（注意端口代码不可修改！）：

```
1. package exp4.todo
2.
3. import chisel3._
4. import chisel3.util._
5.
6. class SquareWave extends Module {
7.   val io = IO(new Bundle {
8.     val waveOut = Output(Bool())
9.   })
10.
11. // TODO: fill your code...
12. }
```

如果写完了代码，可以执行下面的命令来执行测试：

```
mill MyChiselProject.test.testOnly exp4.todo.TestSquareWave
```

```
● lc3@lc3-virtual-machine:~/chisel_exp$ mill MyChiselProject.test.testOnly exp4.todo.TestSquareWave
[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
TestSquareWave:
Detected High Time: 10
Expected Duty Cycle: 10%, Actual Duty Cycle: 10.0%
Expected Phase Offset: 30, Detected Phase Offset: 30
- TestSquareWave should work well
```

出现上面的内容则说明测试通过。