

Chapter 3

Chisel设计基础

罗秋明

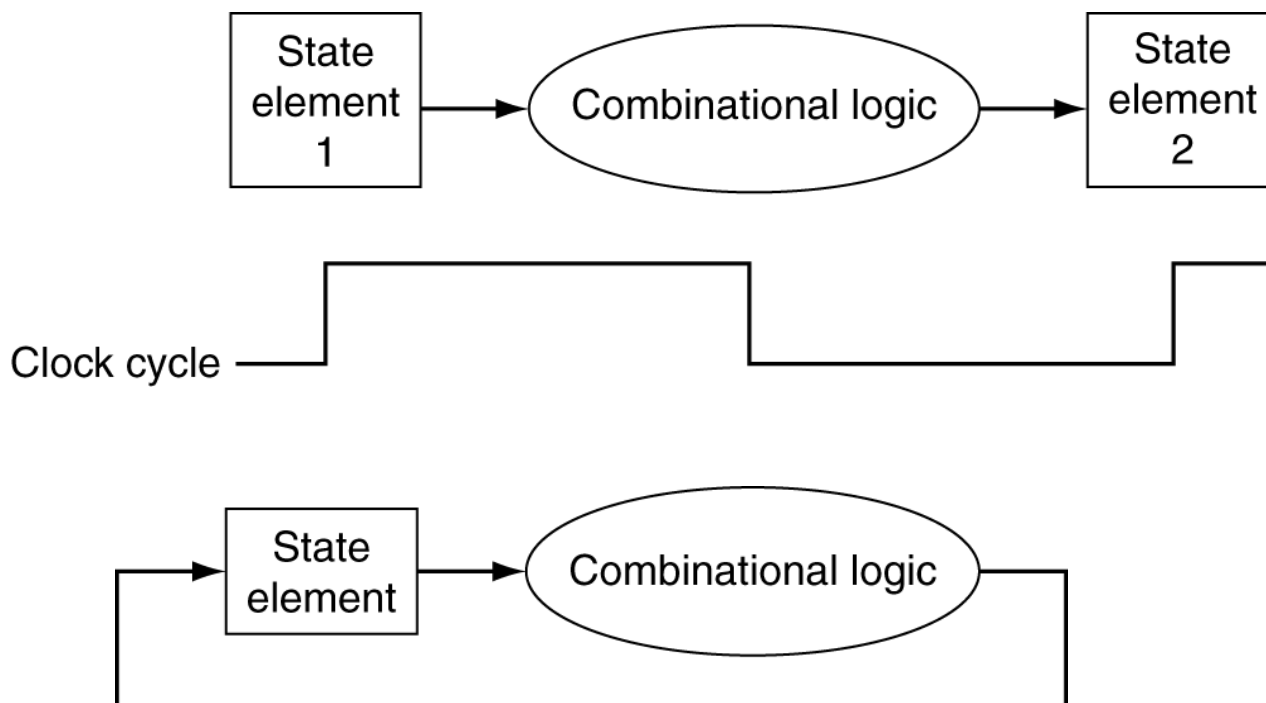
2023-08-15



- 3.1 数字系统的分层设计
- 3.2 Chisel设计基础
 - 组合逻辑设计
 - 时序逻辑设计

3.1 层次性设计

- 数字系统基本形态
 - 组合逻辑和时序逻辑共同构成
 - 有反馈和无反馈形态





■ 层次性设计

■ 扁平/单层设计

- 直接描述硬件细节（行为描述）
- 适合小规模系统

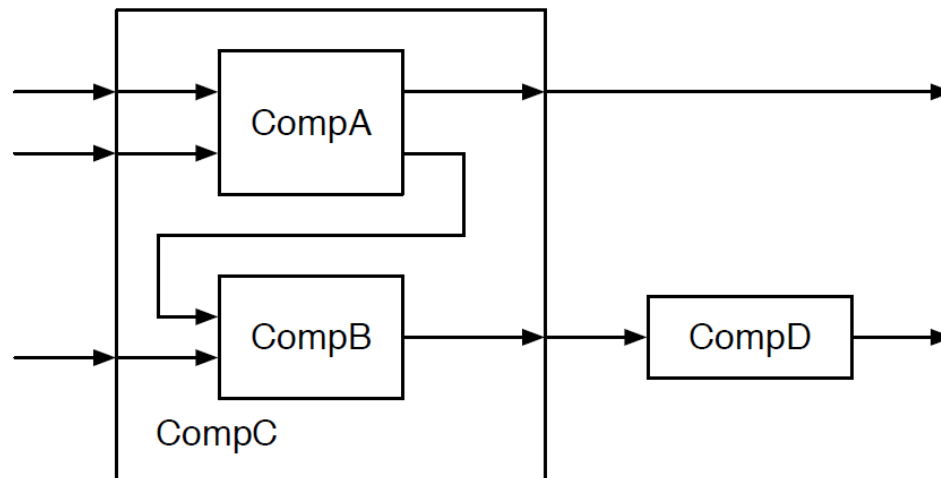
■ 层次性设计

- 采用分治策略（上层系统有底层多个部件构成）
- 将不同部件/器件组合而成（拓扑互连结构）
- 适合大规模系统

■ 独立功能部件component

- 用接口描述其外部关系（输入 和 输出）
- 含有内部逻辑或者内部结构

■ 分层示例



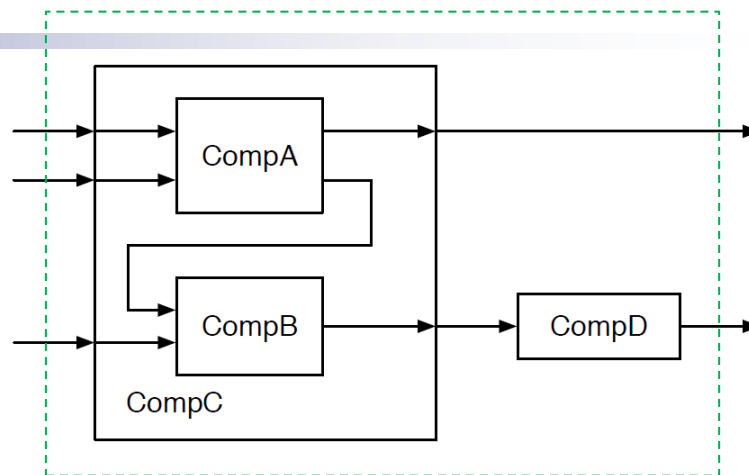
■ 顶层

- 由器件C、D构成
- 3输入、2输出
- 器件C
 - 进一步划分为器件A、B
 - 器件A：3输入2输出
 - 器件B：2输入1输出

顶层器件描述

- 器件 module
- 接口 IO

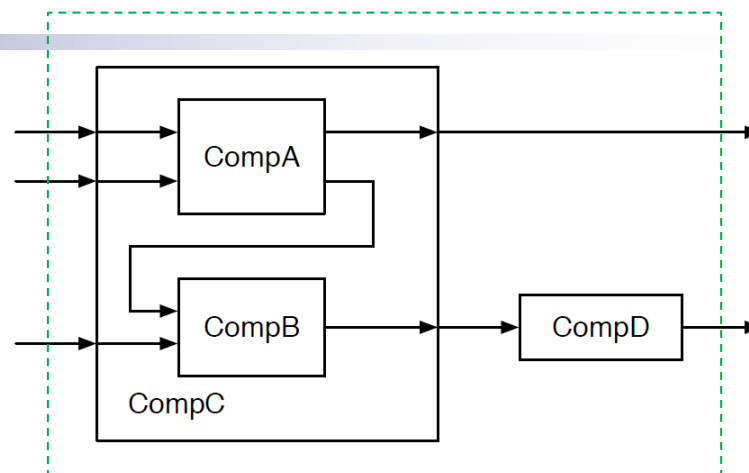
```
class TopLevel extends Module {
  val io = IO(new Bundle {
    val in_a = Input(UInt(8.W))
    val in_b = Input(UInt(8.W))
    val in_c = Input(UInt(8.W))
    val out_m = Output(UInt(8.W))
    val out_n = Output(UInt(8.W))
  })
  // create C and D
  val c = Module(new CompC())
  val d = Module(new CompD())
```



```
// connect C
c.io.in_a := io.in_a
c.io.in_b := io.in_b
c.io.in_c := io.in_c
io.out_m := c.io.out_x
// connect D
d.io.in := c.io.out_y
io.out_n := d.io.out
}
```

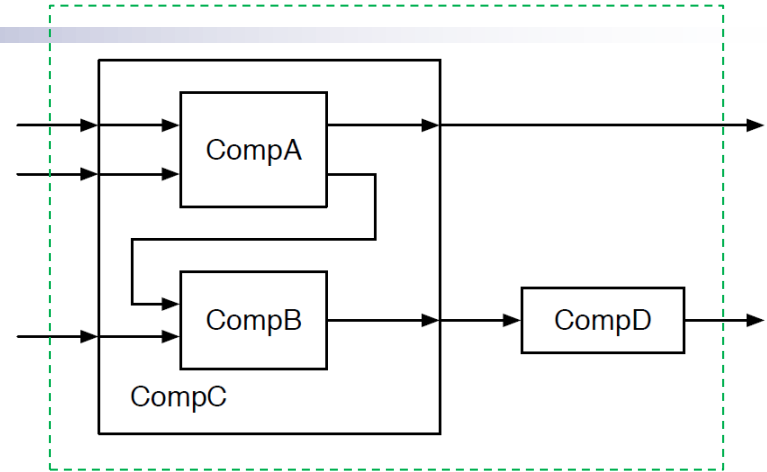
器件C、D描述

```
class CompC extends Module {
  val io = IO(new Bundle {
    val in_a = Input(UInt(8.W))
    val in_b = Input(UInt(8.W))
    val in_c = Input(UInt(8.W))
    val out_x = Output(UInt(8.W))
    val out_y = Output(UInt(8.W))
  })
  // create components A and B
  val compA = Module(new CompA())
  val compB = Module(new CompB())
```



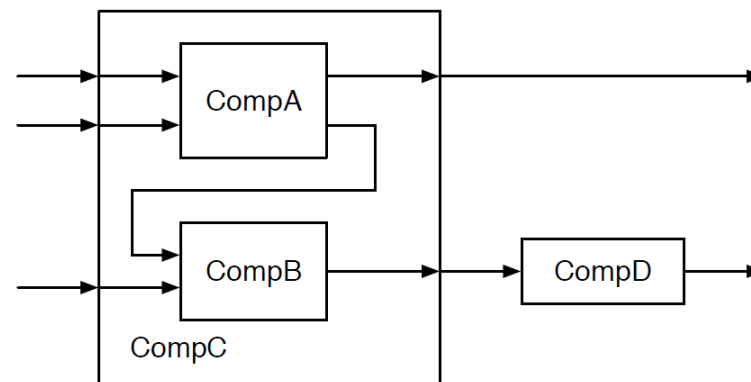
```
// connect A
compA.io.a := io.in_a
compA.io.b := io.in_b
io.out_x := compA.io.x
// connect B
compB.io.in1 := compA.io.y
compB.io.in2 := io.in_c
io.out_y := compB.io.out
}
```

```
class CompD extends Module {  
  val io = IO(new Bundle {  
    val in = Input(UInt(8.W))  
    val out = Output(UInt(8.W))  
  })  
  // function of D  
}
```



■ 器件A、B描述

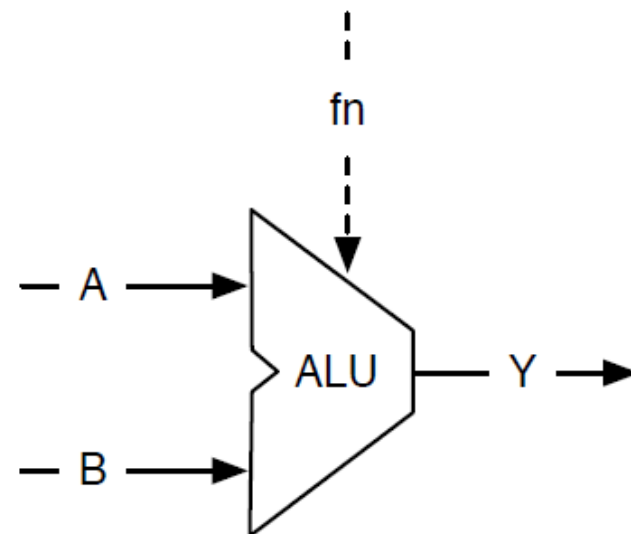
```
class CompA extends Module {
    val io = IO(new Bundle {
        val a = Input(UInt(8.W))
        val b = Input(UInt(8.W))
        val x = Output(UInt(8.W))
        val y = Output(UInt(8.W))
    })
    // function of A
}
```



```
class CompB extends Module {
    val io = IO(new Bundle {
        val in1 = Input(UInt(8.W))
        val in2 = Input(UInt(8.W))
        val out = Output(UInt(8.W))
    })
    // function of B
}
```

■ ALU器件

```
import chisel3.util._
class Alu extends Module {
  val io = IO(new Bundle {
    val a = Input(UInt(16.W))
    val b = Input(UInt(16.W))
    val fn = Input(UInt(2.W))
    val y = Output(UInt(16.W))
  })
  // some default value is needed
  io.y := 0.U
  // The ALU selection
  switch(io.fn) {
    is(0.U) { io.y := io.a + io.b }
    is(1.U) { io.y := io.a - io.b }
    is(2.U) { io.y := io.a | io.b }
    is(3.U) { io.y := io.a & io.b }
  }
}
```



//chisel3的switch里面没有default分支



■ 批量端口连接

■ 运算操作符

- `<>`

■ 连接功能

- 按bundle内的信号名，同名连接
- 不匹配的信号直接忽略（不连接）

■ <>连接示例

■ 支流流水级部件

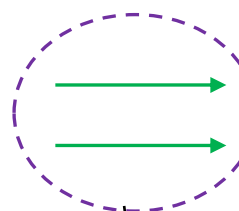
■ 取指fetch、译码decode、执行execute

```
class Fetch extends Module {  
  val io = IO(new Bundle {  
    val instr = Output(UInt(32.W))  
    val pc = Output(UInt(32.W))  
  })  
  // ... Implementation of fetch  
}
```

```
val fetch = Module(new Fetch())  
val decode = Module(new Decode())  
val execute = Module(new Execute())
```

```
fetch.io <> decode.io  
decode.io <> execute.io  
io <> execute.io
```

```
class Decode extends Module {  
  val io = IO(new Bundle {  
    val instr = Input(UInt(32.W))  
    val pc = Input(UInt(32.W))  
    val aluOp = Output(UInt(5.W))  
    val regA = Output(UInt(32.W))  
    val regB = Output(UInt(32.W))  
  })  
  // ... Implementation of decode  
}
```



```
class Decode extends Module {  
  val io = IO(new Bundle {  
    val instr = Input(UInt(32.W))  
    val pc = Input(UInt(32.W))  
    val aluOp = Output(UInt(5.W))  
    val regA = Output(UInt(32.W))  
    val regB = Output(UInt(32.W))  
  })  
  // ... Implementation of decode  
}
```

```
class Execute extends Module {  
  val io = IO(new Bundle {  
    val aluOp = Input(UInt(5.W))  
    val regA = Input(UInt(32.W))  
    val regB = Input(UInt(32.W))  
    val result = Output(UInt(32.W))  
  })  
  // ... Implementation of execute  
}
```

```
val fetch = Module(new Fetch())  
val decode = Module(new Decode())  
val execute = Module(new Execute())
```

```
fetch.io <> decode.io  
decode.io <> execute.io  
io <> execute.io
```



■ 轻量级器件——函数

■ 器件问题

- 需要严格的端口格式
- 需要连线细节

■ 简化

- 如果输入输出数量较少
- 输出单一信号

```
def adder (x: UInt , y: UInt) = {  
    x + y  
}
```

```
val x = adder(a, b)  
val y = adder(c, d)
```

Functions, as lightweight hardware **generators**
Chisel has already an adder generation function,
like +



```
def delay(x: UInt) = RegNext(x) }
```

```
val delOut = delay(delay(delIn))
```




3.2 Chisel设计基础

- 3.2.1 Chisel组合逻辑设计基础
- 3.2.2 Chisel时序逻辑设计基础

Chisel库（各种版本号）链接

edu.berkeley.cs

或

<https://javadoc.io/doc/edu.berkeley.cs>




3.2.1 Chisel组合逻辑设计基础

- 布尔表达式是一切电路的基础

`val e = (a & b) | c`

`val f = ~e`

`e := c & b` 

- Chisel允许更高抽象级的描述

■ 条件更新操作

缺省值也可以在信号定义时给出

```
val w = Wire(UInt())
```

```
val w = WireDefault(0.U)
```

```
w := 0.U
```

//缺省值

```
when (cond) {
```

```
    w := 3.U
```

//条件成立时的值

```
}
```

```
val w = Wire(UInt())
```

```
when (cond) {
```

```
    w := 1.U
```

```
} .otherwise {
```

```
    w := 2.U
```

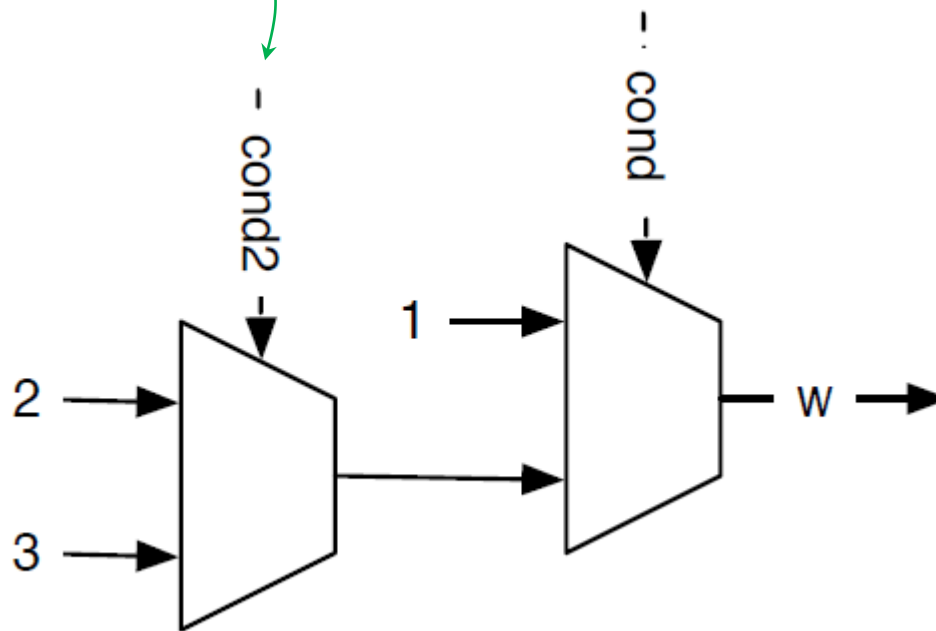
```
}
```

本质上是复选器实现的

■ 嵌套的条件更新

```
val w = Wire(UInt())
when (cond) {
    w := 1.U
} .elsewhen (cond2) {
    w := 2.U
} .otherwise {
    w := 3.U
}
```

如果判定条件采用
同一信号，则使用
Switch更有效



■ 区分

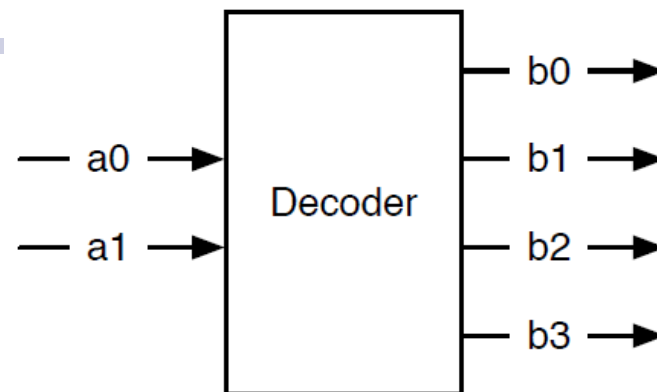
- Scala的 `if, else if, and else`
 - Scala代码, 指令流程控制
 - 可以用于硬件生成器的参数计算
 - Chisel的 `when, .elsewhen, and .otherwise`
 - Chisel硬件生成语句
- 从此也可以更好理解Chisel和Scala的关系

■ 解码器decoder设计

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

```
result := 0.U    //缺省值
```

```
Switch (sel) {  
  is (0.U) { result := 1.U}  
  is (1.U) { result := 2.U}  
  is (2.U) { result := 4.U}  
  is (3.U) { result := 8.U}  
}
```



A 2-bit to 4-bit decoder

a	b
00	0001
01	0010
10	0100
11	1000

条件覆盖不完整的赋值,
会在VHDL/Verilog中导致Latch的出现
Chisel不允许条件覆盖不完整的赋值

- 如果不使用Chisel的UInt类型

```
switch (sel) {  
  is ("b00".U) { result := "b0001".U}  
  is ("b01".U) { result := "b0010".U}  
  is ("b10".U) { result := "b0100".U}  
  is ("b11".U) { result := "b1000".U}  
}
```

a	b
00	0001
01	0010
10	0100
11	1000

可化简为

$\text{result} := 1.U \ll \text{sel}$

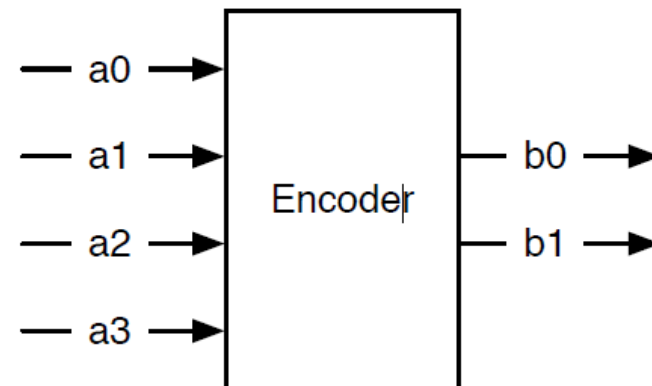
■ 编码器encoder设计

- **注意**: One-hot编码 (只有一位为1)

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

```
b := "b00".U //缺省值
```

```
switch (a) {
  is ("b0001".U) { b := "b00".U }
  is ("b0010".U) { b := "b01".U }
  is ("b0100".U) { b := "b10".U }
  is ("b1000".U) { b := "b11".U }
}
```



A 4-bit to 2-bit encoder.

a	b
0001	00
0010	01
0100	10
1000	11
????	??


■ Chisel硬件生成器角色示例

- 输入16bit, 编码输出4bit
- 每一位都检测
 - 例如若xxx001000, 则在v(3)里面填写上3, 否则v(3) 填0
 - v (0~15) 中最多只有一个非0值
 - v(15)是v(0)~v(15)的OR规约结果

```
val v = Wire(Vec(16, UInt(4.W)))
```

```
v(0) := 0.U
```

```
for (i <- 1 until 16) { //scala代码, 不生成硬件  
    v(i) := Mux(hotIn(i-1), i-1.U, 0.U) | v(i - 1) //Chisel代码, 生成硬件  
}  
val encOut = v(15)
```



■ 仲裁器Arbiter设计

■ 仲裁方法:

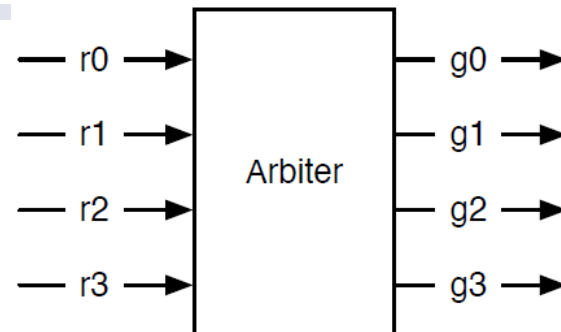
- 编号小的优先级高
- $(r0-r3)=0101$ 时, $(g0-g3)=0001$

```
val grant = VecInit(false.B, false.B, false.B)
val notGranted = VecInit(false.B, false.B)
```

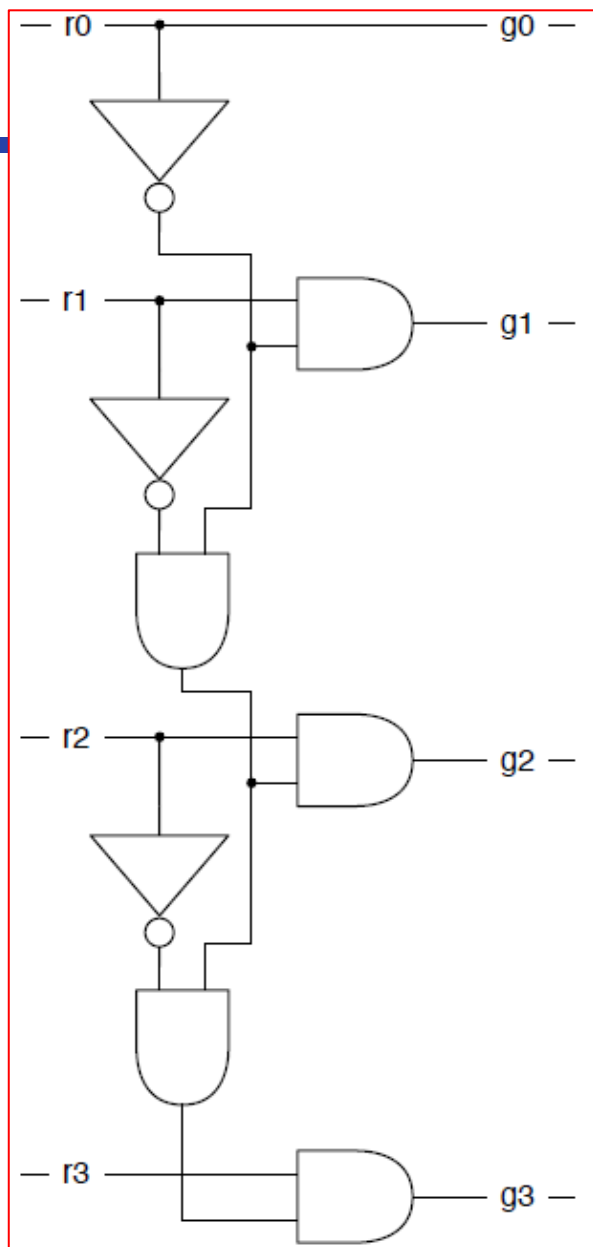
```
grant(0) := request(0)
notGranted(0) := !grant(0)
```

```
grant(1) := request(1) && notGranted(0)
notGranted(1) := !grant(1) && notGranted(0)
```

```
grant(2) := request(2) && notGranted(1)
```



A symbol for a 4-bit arbiter



小规模时，等效描述

```
val grant = WireDefault("b0000".U(3.W))
switch (request) {
  is ("b000".U) { grant := "b000".U }
  is ("b001".U) { grant := "b001".U }
  is ("b010".U) { grant := "b010".U }
  is ("b011".U) { grant := "b001".U }
  is ("b100".U) { grant := "b100".U }
  is ("b101".U) { grant := "b001".U }
  is ("b110".U) { grant := "b010".U }
  is ("b111".U) { grant := "b001".U }
}
```

大规模时，使用硬件生成器风格

```
val grant = VecInit.fill(n)(false.B)
```

```
val notGranted = VecInit.fill(n)(false.B)
```

```
grant(0) := request(0)
```

```
notGranted(0) := !grant(0)
```

```
for (i <- 1 until n) {
```

```
    grant(i) := request(i) && notGranted(i-1)
```

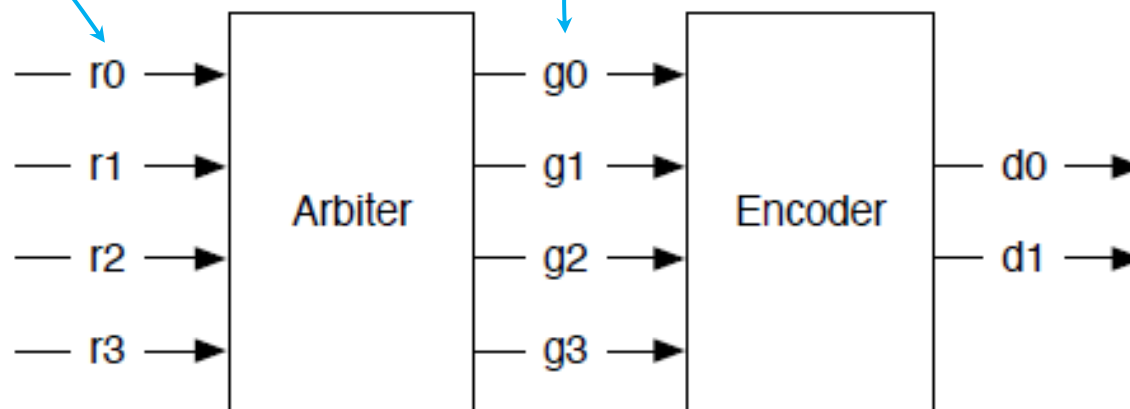
```
    notGranted(i) := !grant(i) && notGranted(i-1)
```

```
}
```

■ 优先编码器Priority Encoder设计

■ 编码器+仲裁器

- 允许多位为1的输入
- 只有低位的1进入编码器



■ 比较器Comparator设计

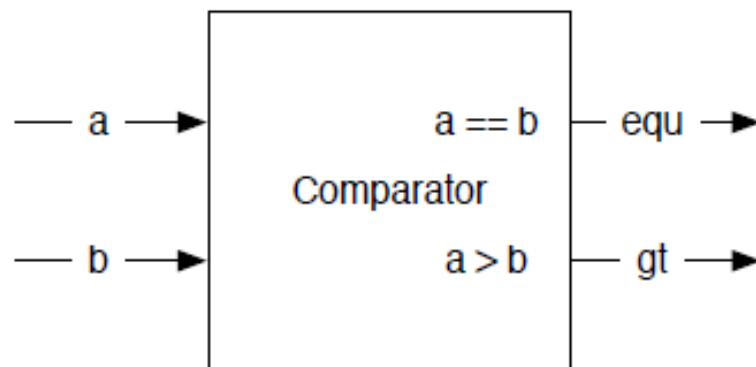
■ 基本功能

- 相等 $a===b$, 大于 $a>b$

■ 其他功能

- 可以同过上述基本功能的组合实现
- $\text{equ or gt} \rightarrow a \geq b$ is true
- $\text{not gt} \rightarrow a \leq b$

```
val equ = a === b  
val gt = a > b
```

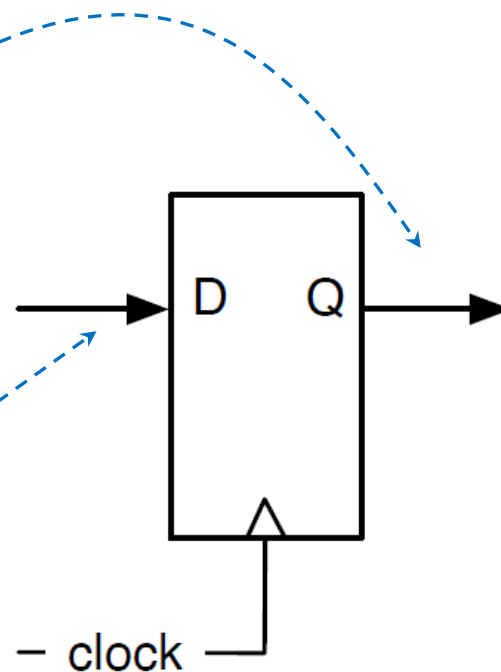


3.2.2 Chisel时序逻辑设计基础

- 时序状态
 - D触发器 D flip-flop
 - 多位D触发器 → 寄存器register

■ 寄存器端口

- 数据输入端
- 状态输出端
- 时钟、复位、使能



val q = **RegNext**(d)

val delayReg = **Reg**(UInt(4.W))//两步指定输入信号
delayReg := delayIn

■ 生成寄存器

■ `Chisel.Reg(t,next,init)`

- `val reg = Chisel.Reg`
`(UInt(8.W), next = io.in * 2.U, init = 0.U)`

- 参数next和init可以省略

- `val delayReg = Chisel.Reg(UInt(4.W))` // 声明、类型

- `delayReg := delayIn` // 赋初值

- `val D = Chisel.Reg(init=false.B)` // 声明、赋初值
// 类型推理

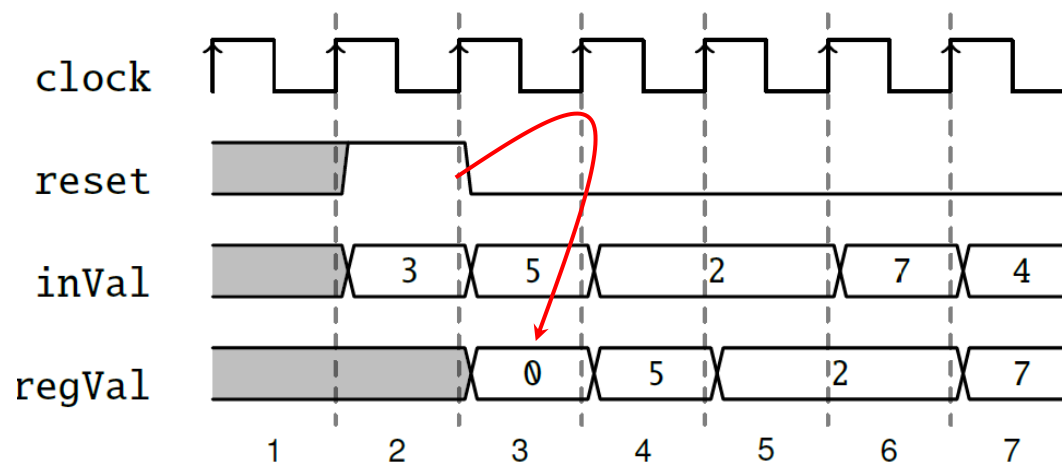


- 生成寄存器
 - $\text{Reg}(t)$
 - $\text{val reg} = \text{Reg}(\text{UInt}(8.W))$
 - $\text{reg} := \text{regIn}$

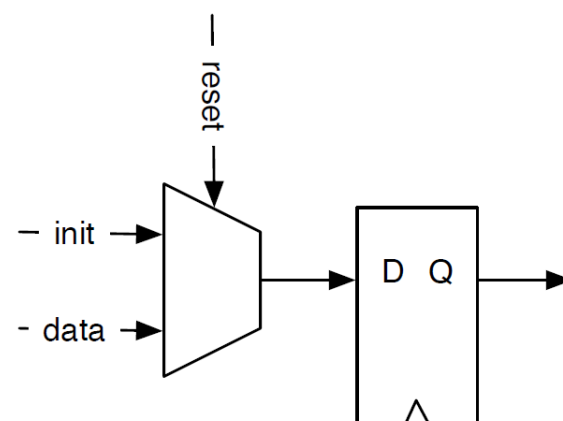
■ RegInit(resetData)

- `val valReg = RegInit(0.U(4.W))`
- `valReg := inVal`

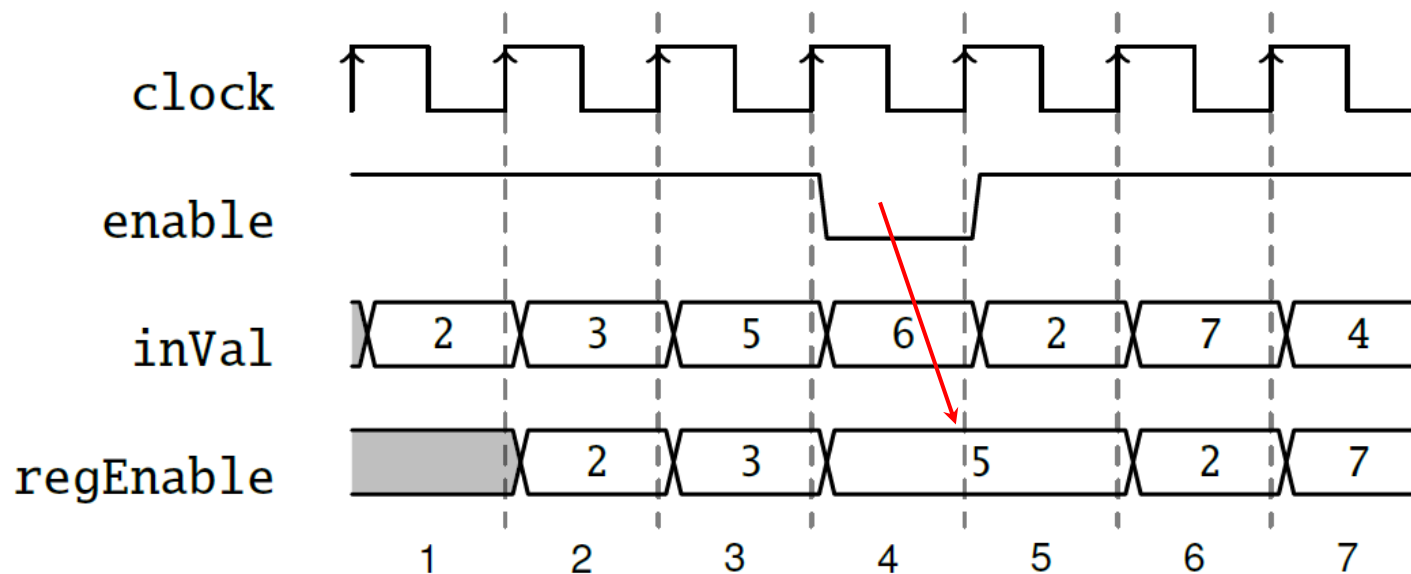
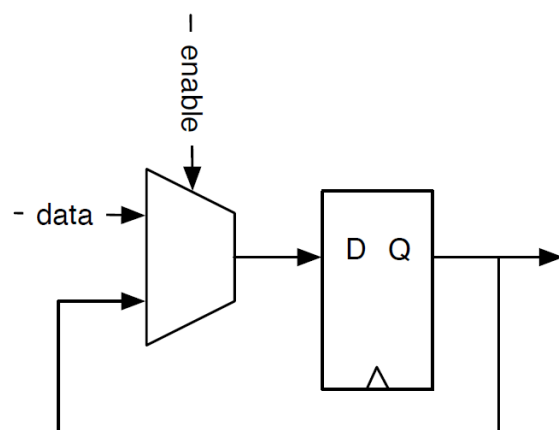
- 复位信号是系统信号
- Chisel使用同步复位机制



//复位时的初值



- En使能:
 - `val enableReg = Reg(UInt(4.W))`
 - `when (enable) {`
 `enableReg := inVal`
■ `}`
- `val enableReg2 = RegEnable(inVal , enable)`

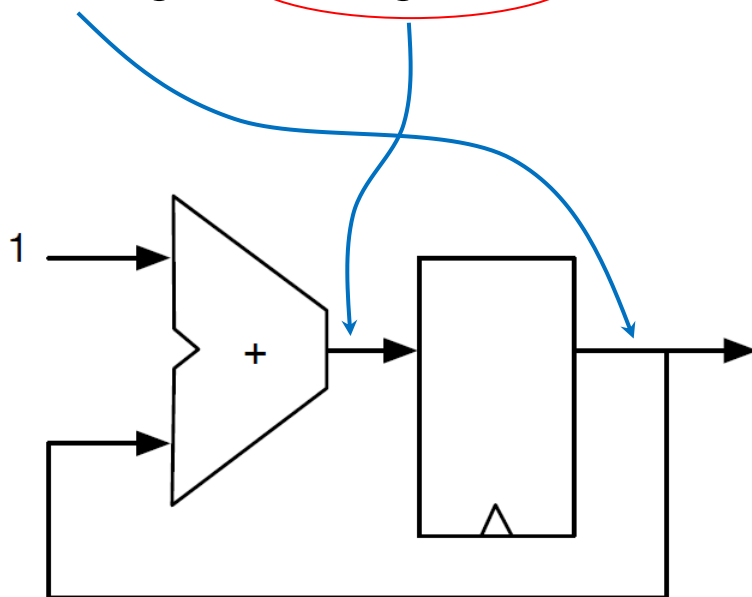


- Enable+Reset
 - `val resetEnableReg = RegInit(0.U(4.W))`
 - `when (enable) {`
 `resetEnableReg := inVal`
■ `}`
- Enable+Reset+Init
 - `val resetEnableReg2 =`
 `RegEnable(inVal , 0.U(4.W), enable)`

计数器counter设计

寄存器+加法器:

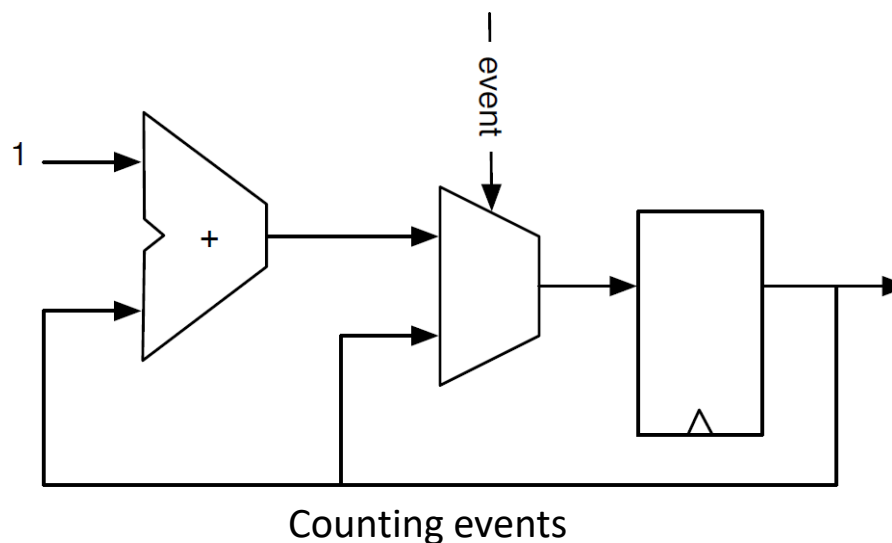
- `val cntReg = RegInit(0.U(4.W))`
- `cntReg := cntReg + 1.U`



An adder and a register result in counter

■ 事件计数（计数使能）：

- `val cntEventsReg = RegInit(0.U(4.W))`
- `when(event) {`
`cntEventsReg := cntEventsReg + 1.U`
- `}`



■ 循环计数:

- `val cntReg = RegInit(0.U(8.W))`
- `cntReg := cntReg + 1.U`
- `when(cntReg === N) {`
 `cntReg := 0.U`
■ `}`

■ 用mux的等效描述

- `val cntReg = RegInit(0.U(8.W))`
- `cntReg := Mux(cntReg === N, 0.U, cntReg + 1.U)`



- 反向循环计数：
 - `val cntReg = RegInit(N)`
 - `cntReg := cntReg - 1.U`
 - `when(cntReg === 0.U) {`
 `cntReg := N`
`}`

■ 用生成器函数产生正向计数器:

- // This function returns a counter
- def genCounter(n: Int) = {
 val cntReg = RegInit(0.U(8.W))
 cntReg := Mux(cntReg == n.U, 0.U, cntReg + 1.U)
 cntReg
}

函数返回值 (硬件/寄存器)

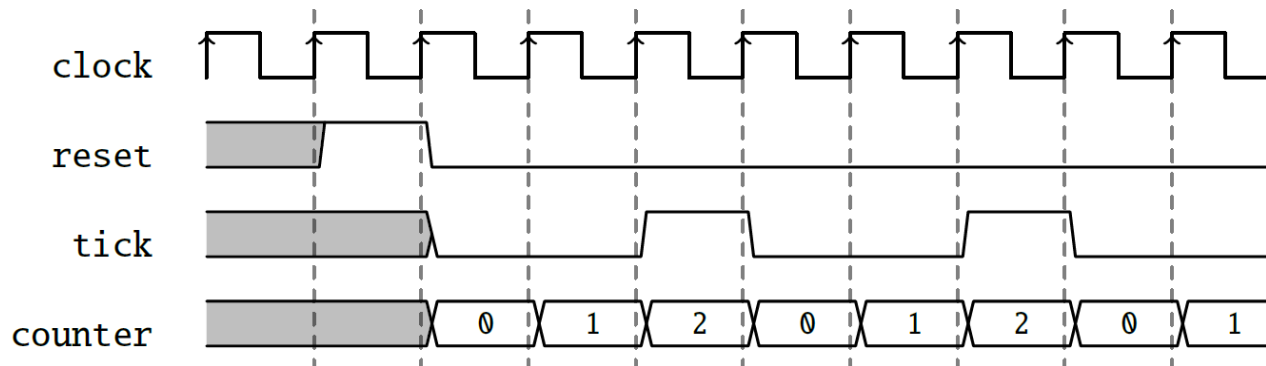
- // now we can easily create many counters
- val count10 = genCounter(10)
- val count99 = genCounter(99)

Chisel的“生成器”特性的体现

降频 和 降频计数:

降频信号产生

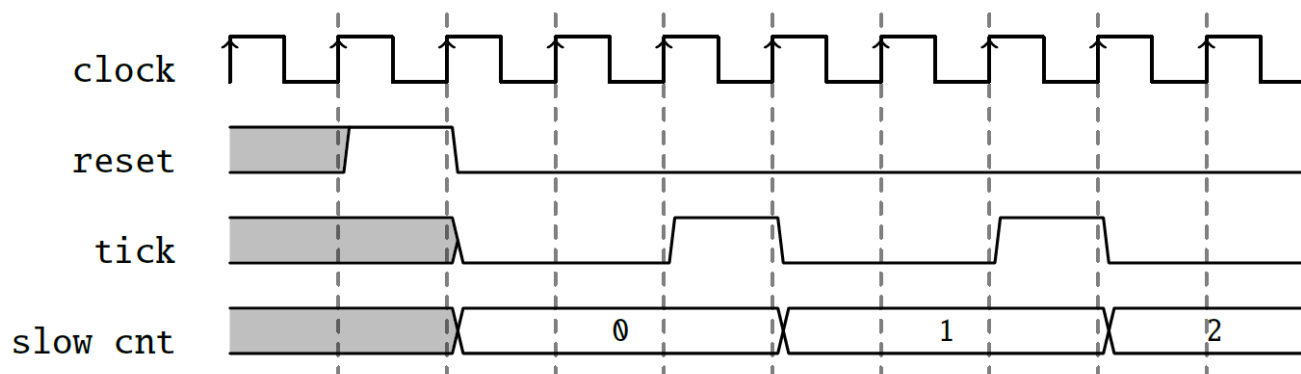
- `val tickCounterReg = RegInit(0.U(32.W))`
- `val tick = tickCounterReg === (N-1).U`
- `tickCounterReg := tickCounterReg + 1.U`
- `when (tick) {`
 `tickCounterReg := 0.U`
■ `}`



A waveform diagram for the generation of a slow frequency tick

降频计数:

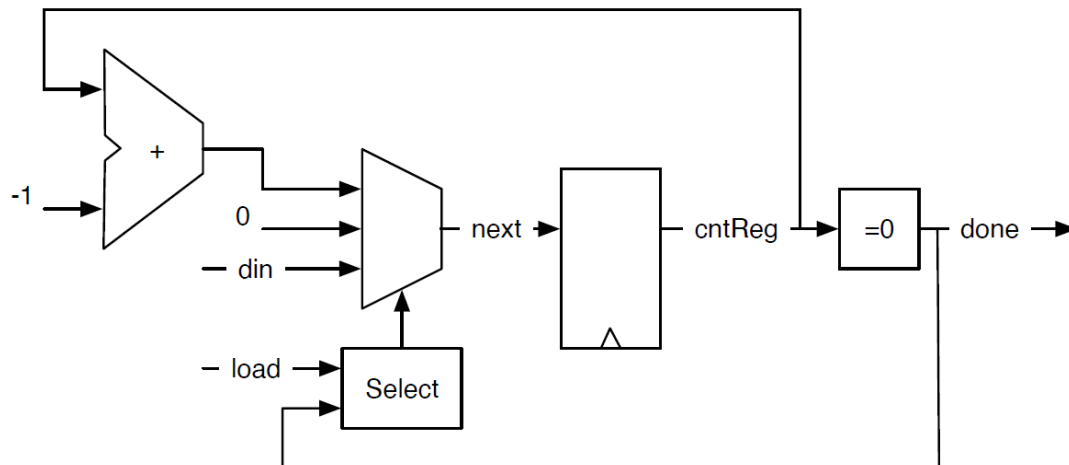
- 作为使能信号 (仍以clock为系统工作时钟)
- `val lowFrequCntReg = RegInit(0.U(4.W))`
- `when (tick) {`
 $\text{lowFrequCntReg} := \text{lowFrequCntReg} + 1.U$
`}`



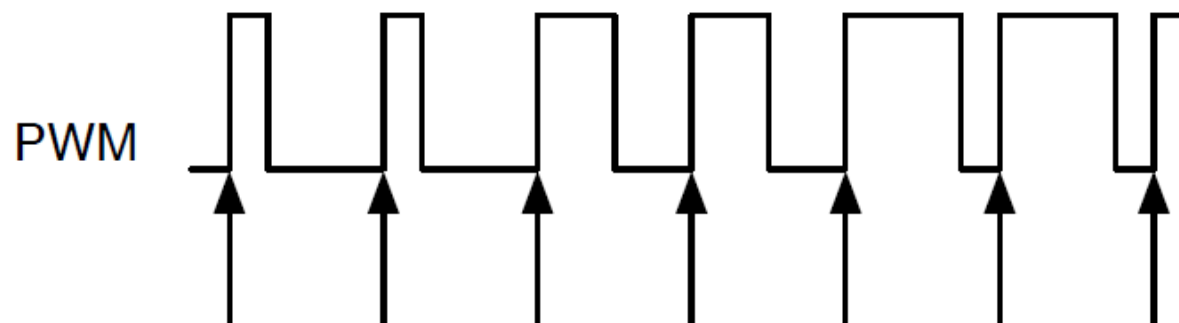
Using the slow frequency tick.

■ 定时器（单发）：

- `val cntReg = RegInit(0.U(8.W))`
- `val done = cntReg === 0.U`
- `val next = WireDefault(0.U)`
- `when (load) {`
`next := din`
`}.elsewhen (!done) {`
`next := cntReg - 1.U`
`}`
- `cntReg := next`



■ 脉宽调制PWM:



Pulse-width modulation

- 10个周期, 前3三个周期为高
- `def pwm(nrCycles: Int, din: UInt) = {`
- `val cntReg =`
`RegInit(0.U(unsignedBitLength(nrCycles - 1).W))`
- `cntReg := Mux(cntReg === (nrCycles - 1).U, 0.U, cntReg +`
`1.U)`
- `din > cntReg`
- `}`
- `val din = 3.U`
- `val dout = pwm(10, din)`



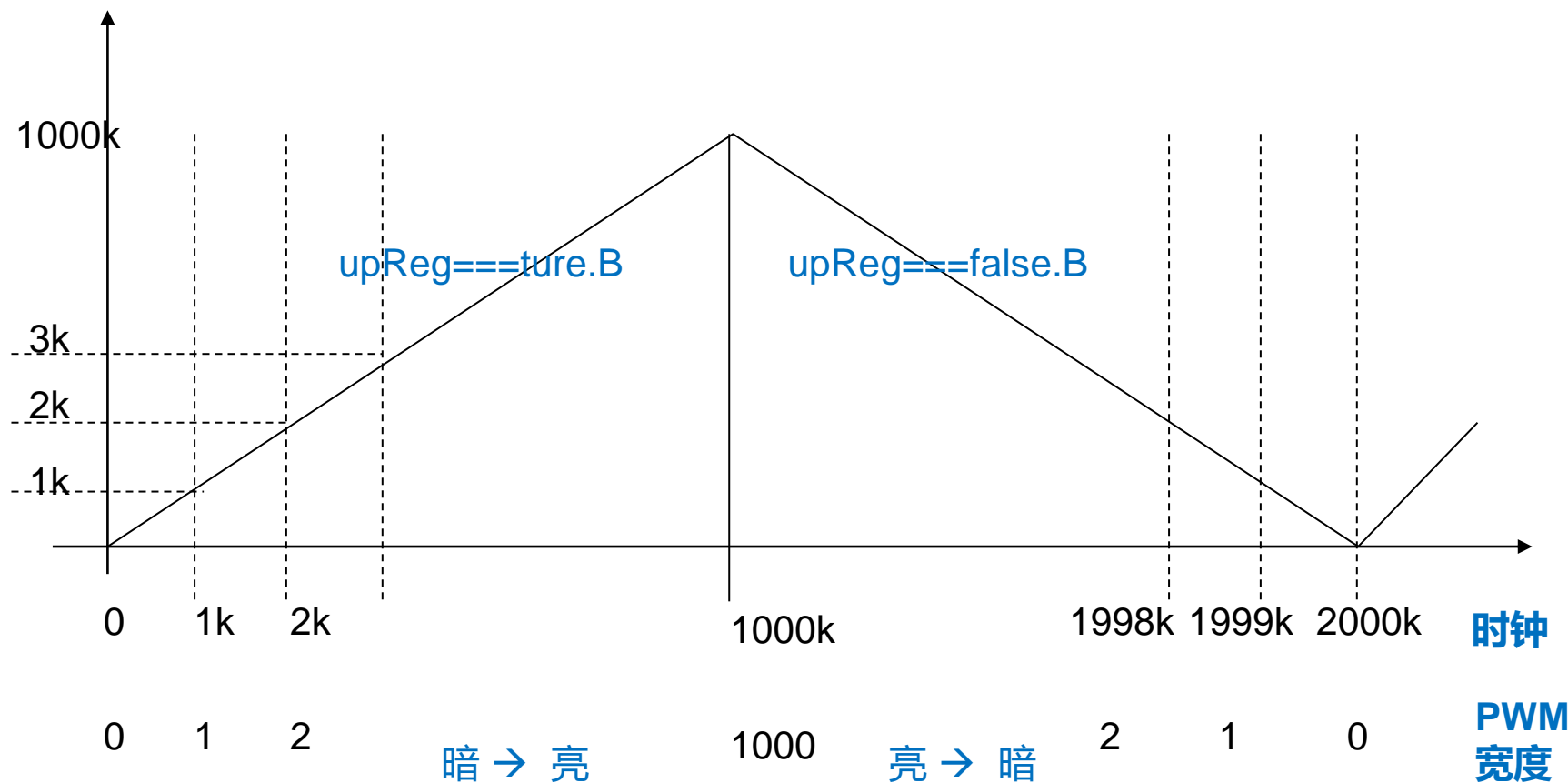
- `val FREQ = 100000000 // a 100 MHz clock input`
- `val MAX = FREQ/1000 // 1 kHz`

- `val modulationReg = RegInit(0.U(32.W))`
- `val upReg = RegInit(true.B)`
- `when (modulationReg < FREQ.U && upReg) {` //向上计数
- `modulationReg := modulationReg + 1.U`
- `} .elsewhen (modulationReg === FREQ.U && upReg) {` //反转向下计数
- `upReg := false.B`
- `} .elsewhen (modulationReg > 0.U && !upReg) {` //向下计数
- `modulationReg := modulationReg - 1.U`
- `} .otherwise { // 0` //反转向上计数
- `upReg := true.B`
- `}`

- `// divide modReg by 1024 (about the 1 kHz)`
- `val sig = pwm(MAX, modulationReg >> 10)` //根据计数值输出PWM



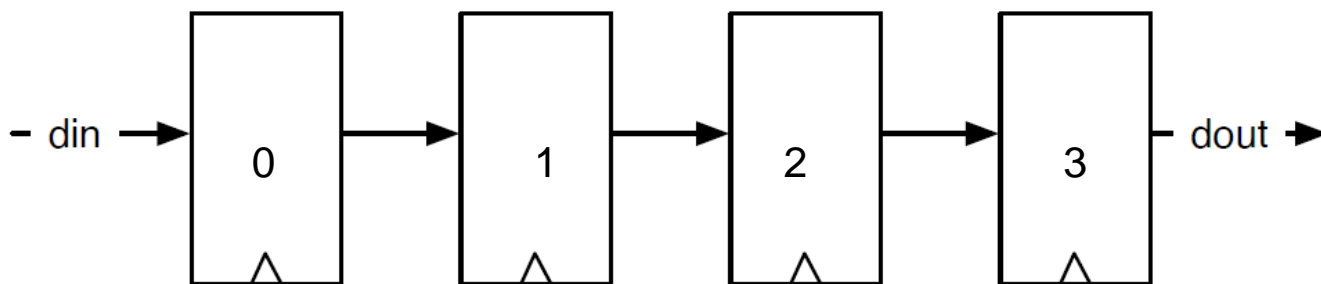
modulationReg



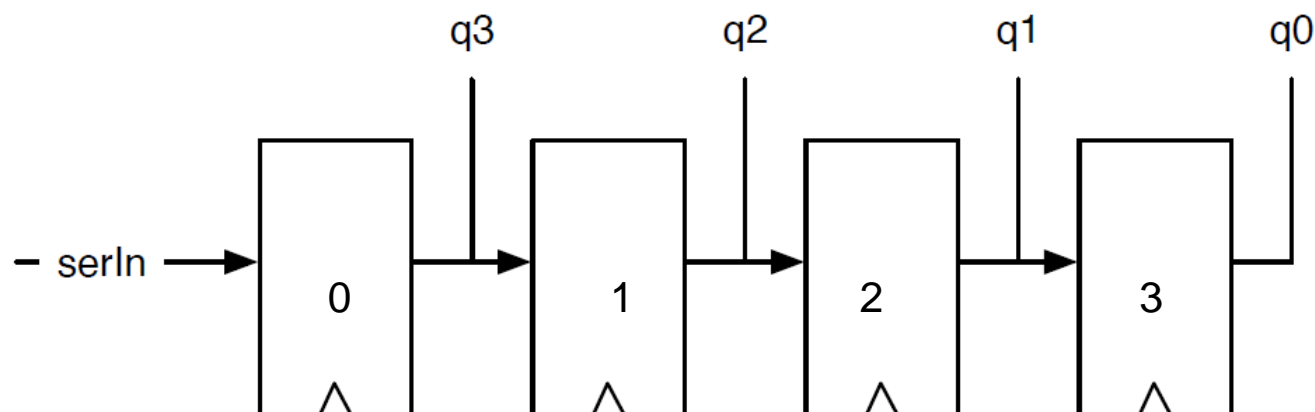
■ 移位寄存器Shift Registers设计

■ 移位寄存器

- 多位D触发器串接 → 移位寄存器 shift register
- 仍有并行输出信号
- `val shiftReg = Reg(UInt(4.W))`
- `shiftReg := shiftReg(2, 0) ## din`
- `val dout = shiftReg(3)` //只输出1bit信号驱动dout

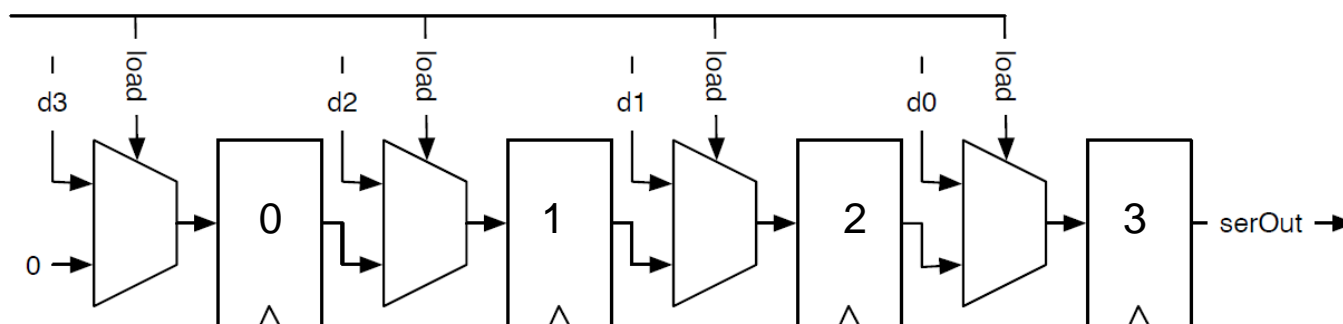


- 移位寄存器 (带并行输出)
 - `val outReg = RegInit(0.U(4.W))`
 - `outReg := serIn ## outReg(3, 1)`
 - `val q = outReg` // 4bit信号并行输出驱动dout



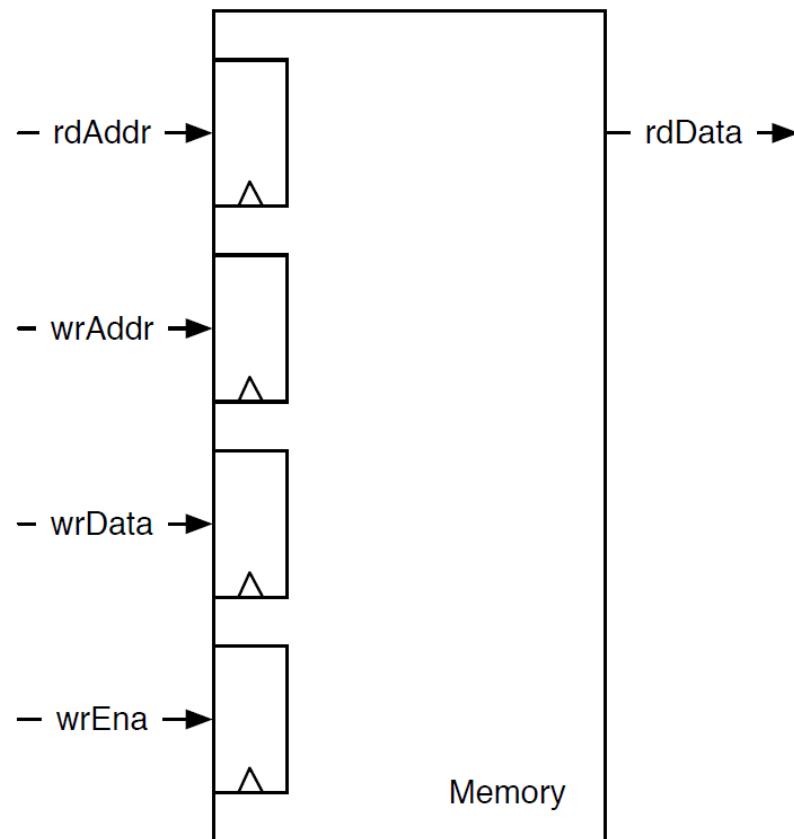
■ 移位寄存器 (带并行输入)

- `val loadReg = RegInit(0.U(4.W))`
- `when (load) {`
`loadReg := d`
- `} otherwise {`
`loadReg := 0.U ## loadReg(3, 1)`
- `}`
- `val serOut = loadReg(0)` // 4bit信号并行输出驱动dout



■ 内存Memory设计

- 状态单元，类似于Register
- 一组可寻址的Reg Vec
- 同步内存
- 双端口
 - 读端口：输入rdAddr
输出rdData
 - 写端口：输入wrAddr
输入wrData
输入wrEna





■ 1 KB Memory

- class Memory() extends Module {
- val io = IO(new Bundle {
- val rdAddr = Input(UInt(10.W))
- val rdData = Output(UInt(8.W))
- val wrAddr = Input(UInt(10.W))
- val wrData = Input(UInt(8.W))
- val wrEna = Input(Bool())
- })
- val mem = SyncReadMem(1024, UInt(8.W))
- io.rdData := mem.read(io.rdAddr)
- when(io.wrEna) {
- mem.write(io.wrAddr , io.wrData)
- }
- }



■ 同一地址同时读写问题

- ...前面端口描述相同...
- `val mem = SyncReadMem(1024, UInt(8.W))`
- `val wrDataReg = RegNext(io.wrData)`
- `val doForwardReg = RegNext(io.wrAddr === io.rdAddr && io.wrEna)`
- `val memData = mem.read(io.rdAddr)`
- `when(io.wrEna) {`
- `mem.write(io.wrAddr , io.wrData)`
- `}`
- `io.rdData := Mux(doForwardReg , wrDataReg , memData)`
- `}`

