Chapter 2

Chisel数字信号和运算

罗秋明

2023-08-14

2.1 数字信号

- 数字信号
  - Val
  - 区别于Var

- 信号数值类型
- 组合逻辑信号
- 时序逻辑信号
- 总线/线束/信号数组（向量）

信号的数值类型和位宽

- 信号的数值类型
  - Bits、UInt、SInt
  - 位向量（1位 或 多位)
  - 适用于组合逻辑和时序逻辑
- 信号的位宽　　(整数 n -> Width n.W)
  - Bits(8.W)
  - UInt(8.W)
  - SInt(10.W)

常量信号

- 常量类型：整数、布尔
- 整数常量
  - 类型
    - 无符号 0.U
    - 有符号 -3.S
  - 信号位宽
    - 0.U(4.W)
    - 8.S(n.W)
  - 注意区分
    - 3.U(32) 中的32不是位宽，而是第32bit，对数值3而言第32位是0

信号类型和位宽 的 自动推断

- 优势
  - 无需明确的类型和位宽说明
  - 源码简洁、易于阅读
  - VHDL/Verilog无此特性
- 示例
  - "hff".U // hexadecimal representation of 255
  - "o377".U // octal representation of 255
  - "b1111_1111".U // binary representation of 255
    //数字之间的下划线自动被忽略

以上三者相当于用（8.W）指定位宽

逻辑值

- 布尔类型
  - Bool()
- 布尔常量取值
  - true.B
  - false.B

信号集的操作

- 信号中的1 bit
  - val sign = x (31)
- 信号中的连续bit　　(子集)
  - val lowByte = largeWord (7, 0)
- 信号拼接　　(超集)
  - val word = Cat (highByte , lowByte)

Chisel defined hardware functions, invoked on v.

Function	Description	Data types
`v.andR v.orR v.xorR`	AND, OR, XOR reduction	UInt, SInt, returns Bool
`v(n)`	extraction of a single bit	UInt, SInt
`v(end, start)`	bitfield extraction	UInt, SInt
`Fill(n, v)`	bitstring replication, n times	UInt, SInt
`Cat(a, b, ...)`	bitfield concatenation	UInt, SInt

信号集与硬件

- 信号不等于硬件
- 硬件信号
  - 绑定于线网Wire、寄存器Reg和接口IO

2.2 组合逻辑信号

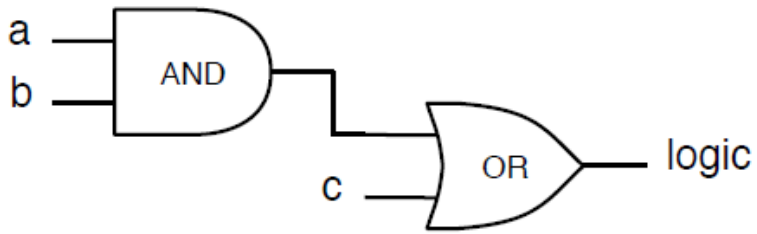
- 线网Wire （组合逻辑信号）
  - val w = Wire(UInt())
  - val number = WireDefault (10.U(4.W))
- 用 := 更新信号的值
  - w := a & b
  - 区分 = ← → :=

- 信号变换
  - 使用类似C、Java的逻辑操作符

- 示例

基于a、b、c信号来进行位宽推理
当a、b、c位宽变化时无需修改代码

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



逻辑运算 和 算术运算

- +/-操作结果的位宽取决于操作数最大位宽
- *操作结构位宽为乘数位宽之和

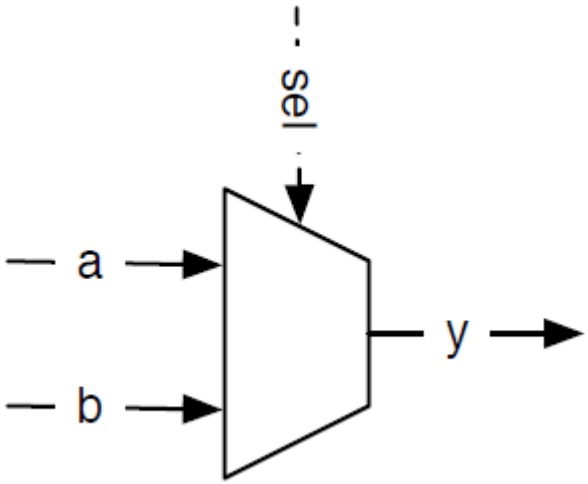
```
val and = a & b          // bitwise and
val or = a | b    // bitwise or
val xor = a ^ b  // bitwise xor
val not = ~a      // bitwise negation

val add = a + b          // addition
val sub = a - b  // subtraction
val neg = -a      // negate
val mul = a * b          // multiplication
val div = a / b   // division
val mod = a % b          // modulo operation
```

Chisel defined hardware operators

Operator	Description	Data types
* / %	multiplication, division, modulus	UInt, SInt
+ -	addition, subtraction	UInt, SInt
=== =/=	equal, not equal	UInt, SInt, returns Bool
> >= < <=	comparison	UInt, SInt, returns Bool
<< >>	shift left, shift right (sign extend on SInt)	UInt, SInt
~	NOT	UInt, SInt, Bool
& | ^	AND, OR, XOR	UInt, SInt, Bool
!	logical NOT	Bool
&& ||	logical AND, OR	Bool

信号的路由——复选器

- N →1 多选一路由
  - val result = Mux(sel, a, b)

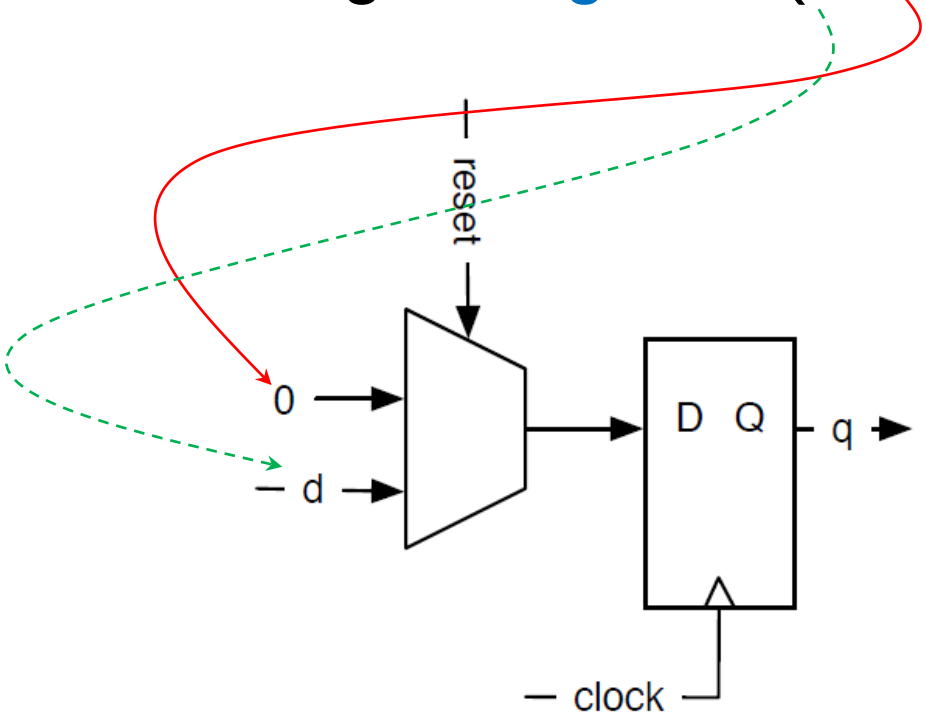


  - sel取值：true.B 或 false.B

2.3 时序逻辑信号

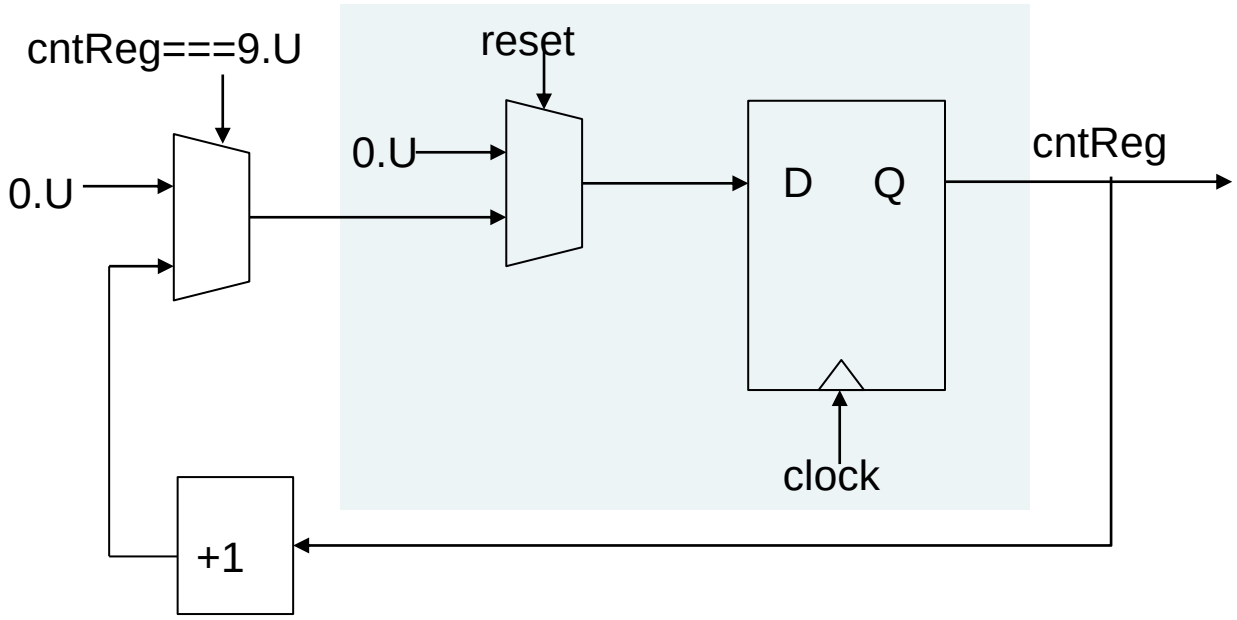
- 寄存器Regster （时序逻辑信号）
  - val reg = Reg(SInt ())
  - val reg = RegInit (0.U(8.W))
- 用 := 更新信号的值
  - reg := d
  - val q = reg
- 定义时同时完成输入连接
  - val nextReg = RegNext (d)

- 定义时给出复位值、同时完成输入连接
  - val bothReg = RegNext (d, 0.U)



计数器示例

```
val cntReg = RegInit (0.U(8.W))

cntReg := Mux(cntReg === 9.U, 0.U, cntReg + 1.U)
```



2.4 线束/信号向量

- 总线
  - 一组相关信号 （组合逻辑信号 或 时序逻辑信号）
- 线束bundle
  - 不同类型的信号集合 (类似C语言的结构体)
- 信号向量vec (类似C语言的数组)
  - 相同类型的信号集合
- 嵌套
  - Bundle和vec可以互相嵌套

Bundle示例

- 定义Bundle

```
class Channel() extends Bundle {
val data = UInt(32.W)
val valid = Bool()
}
```

- 使用Bundle

```
val ch = Wire(new Channel())
ch.data := 123.U
ch.valid := true.B
val b = ch.valid

val channel = ch
```

Vec示例(Wire)

- 定义线网Wire的Vec

```
val v = Wire(Vec(3, UInt(4.W)))
```

- 使用Vec（按索引访问）

```
v(0) := 1.U
v(1) := 3.U
v(2) := 5.U

val idx = 1.U(2.W)
val a = v(idx)
```

对于线网Wire的Vec
本质上是一组信号和复选器
- 根据索引来选则信号

Vec示例(Register)

- 定义寄存器Register的Vec

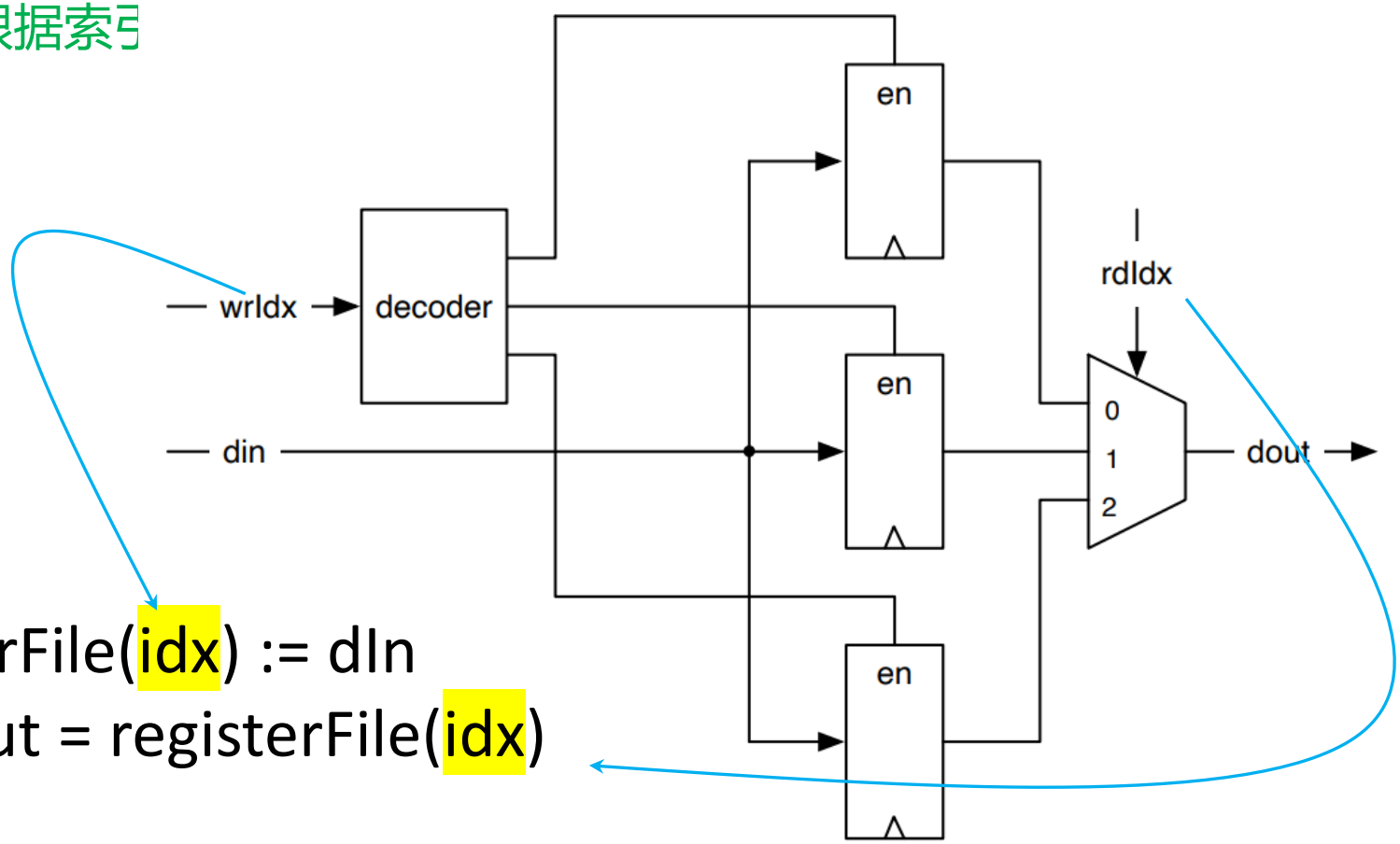
```
val registerFile = Reg(Vec(32, UInt(32.W)))
```

- 使用Vec（按索引访问）

```
registerFile(idx) := dIn
val dOut = registerFile(idx)
```

对于寄存器Register的Vec
本质上是一组寄存器和复选器

- 根据索引



registerFile(idx) := dIn

val dOut = registerFile(idx)

- 示例1：带初值的寄存器Register的Vec

```
val initReg =
        RegInit(VecInit (0.U(3.W), 1.U, 2.U))

val resetVal = initReg(sel)
initReg (0) := d
initReg (1) := e
initReg (2) := f

registerFile(idx) := dIn
val dOut = registerFile(idx)
```

示例：带初值的Register的Vec

```
val initReg =
        RegInit(VecInit (0.U(3.W), 1.U, 2.U))
                                    //初值不等，为1/2/3
val resetVal = initReg(sel)
initReg (0) := d
initReg (1) := e
initReg (2) := f
```

```
val resetRegFile =          //初值相等
        RegInit(VecInit(Seq.fill (32) (0.U(32.W))))
val rdRegFile = resetRegFile (sel)
```

Bundel/Vec混合嵌套示例

- 定义Bundle组成的Vec

```
val vecBundle = Wire(Vec(8, new Channel()))
```

- 定义Vec构成的Bundle

```
class BundleVec extends Bundle {
val field = UInt(8.W)
val vector = Vec(4,UInt(8.W))
}
```

Bundle寄存器的复位初值设置

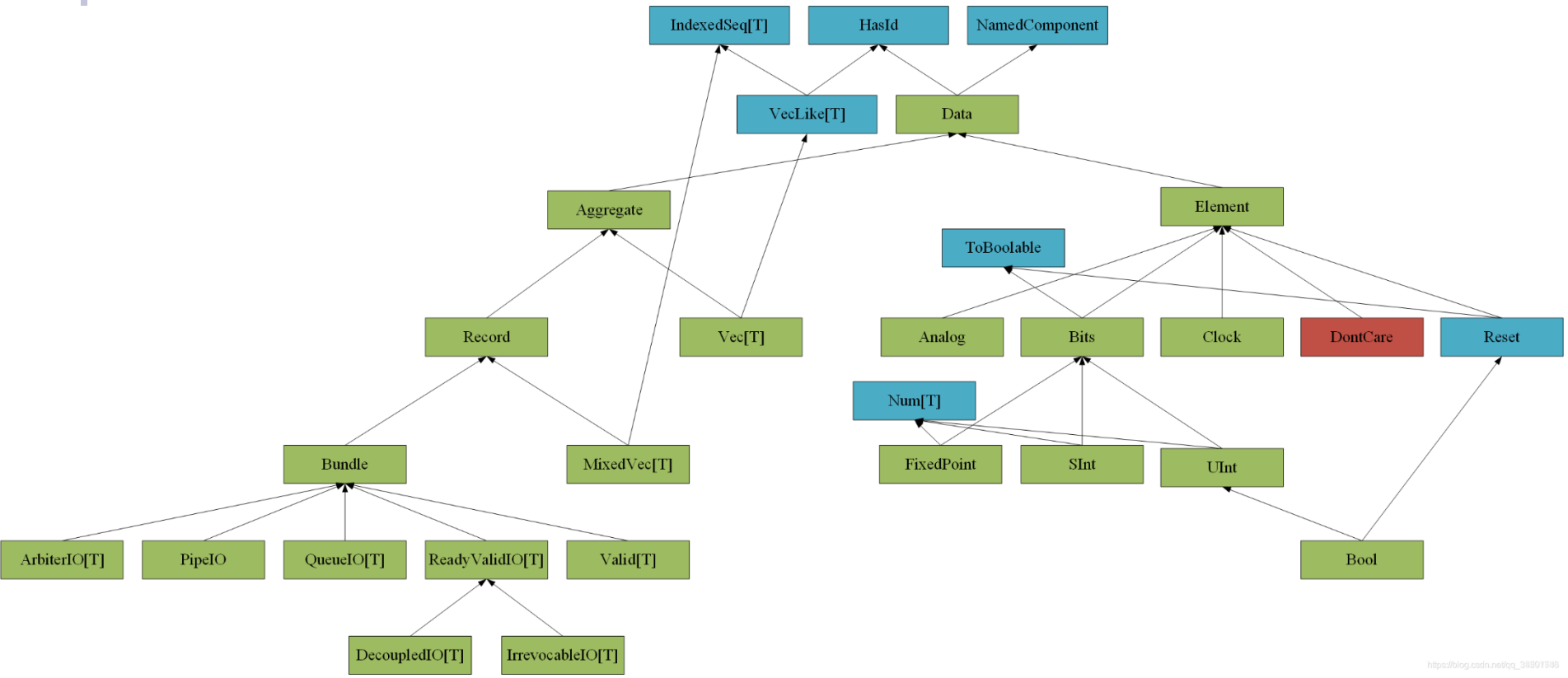
```
val initVal = Wire(new Channel())
initVal.data := 0.U
initVal.valid := false.B

val channelReg = RegInit(initVal)
```

2.5 Chisel的并发执行

- 所有硬件生成语句并发执行
  - 虽然在源代码中按先后次序编写
  - Chisel描述的对象是硬件而非“程序”
  - Chisel中“非硬件生成”的语句仍是“软件”
- Chisel语句执行的结果是生成硬件描述
  - FIRRTL/Verilog

附1：Chisel的数据类型

- 区别于Scala数据类型
- 绿色方块是class，红色是object，蓝色是trait



- 有字面量（如1.U(32.W)）的数据类型用于赋值、初始化寄存器等操作
- 无字面量（如UInt(32.W)）的数据类型则用于声明端口、构造向量等。