Chisel Design on Fundamental IC

Xiaokun Yang

9/24/2024

Constant and Wire

1. Constant

3.U(4.W) // 4-bit constant of 3

“hff”.U(8.W)

“o377”.U(8.W)

“b1111_1111”.U(8.W)

true.B false.B

2. Wire

val w=Wire(UInt(8.W))

val w=Wire(Vec(n, UInt(8.W))

w(0) := 3.U(8.W)

w(1) := 5.U(8.W)

```
class wire_connection(bw:Int) extends Module{
  val io=IO(new Bundle {
    val in_a = Input(UInt(bw.W))
    val out_b = Output(UInt(bw.W))
  })
  val w1 = Wire(UInt(bw.W))
  val w2 = Wire(Vec(2, UInt(bw.W)))
    w1 := 7.U(bw.W) & io.in_a
    w2(0) := 3.U(bw.W) | io.in_a
    w2(1) := 5.U(bw.W) & w1
    io.out_b := w2(0) & w2(1)
}
```

Combinational and Sequential Circuit

1. Mux

val c = Mux(sel, a, b)

2. Counter

val cnt = RegInit(0.U(8.W))

val nxt_cnt = Wire(UInt(8.W))

nxt_cnt = Mux(cnt===9.U, 0.U, cnt+1.U)

```
class counter(bw:Int) extends Module{
  val io=IO(new Bundle {
    val in_en = Input(UInt(1.W))
    val out_cnt = Output(UInt(bw.W))
  })
  val cnt = RegInit(0.U(bw.W))
  val nxt_cnt = Wire(UInt(bw.W))
  nxt_cnt := Mux(io.in_en===1.U : UInt, Mux(cnt===9.U, 0.U, cnt+1.U), cnt)
  cnt := nxt_cnt
  io.out_cnt := cnt
}
```

when, if-else

1. when-elsewhen-otherwise
2. if-else: cannot be used to construct hardware when working with UInt, Bool, or other hardware types. Instead, Chisel's when-elsewhen-otherwise should be used as they are designed to describe hardware logic.

```
class mux_ifelse(bw:Int) extends Module {
  val io = IO(new Bundle {
    val in_sel = Input(UInt(1.W))
    val in_a   = Input(UInt(bw.W))
    val in_b   = Input(UInt(bw.W))
    val out_c   = Output(UInt(bw.W))
  })
  if (io.in_sel==1.U) {
    io.out_c := io.in_a
  } else {
    io.out_c := io.in_b
  }
}
```

```
class mux(bw:Int) extends Module {
  val io = IO(new Bundle {
    val in_sel = Input(UInt(1.W))
    val in_a   = Input(UInt(bw.W))
    val in_b   = Input(UInt(bw.W))
    val out_c   = Output(UInt(bw.W))
  })
  when (io.in_sel === 1.U) {
    io.out_c := io.in_a
  }.otherwise {
    io.out_c := io.in_b
  }
}
```

```
// Generated by CIRCT firtool-1.62.0
module mux_ifelse(
  input         clock,
                reset,
                io_in_sel,
  input  [3:0] io_in_a,
                io_in_b,
  output [3:0] io_out_c
);

  assign io_out_c = io_in_b;
endmodule
```

Register/Register File

1. Register
2. Register file

```
class register(bw: Int) extends Module {
  val io = IO(new Bundle {
    val in_d = Input(UInt(bw.W))
    val out_q = Output(UInt(bw.W))
  })
  val register = RegInit(0.U(bw.W))
    register := io.in_d
    io.out_q := register
}
```

```
class register_file(n: Int, bw: Int) extends Module {
  val io = IO(new Bundle {
    val in_d = Input(Vec(n, UInt(bw.W)))
    val out_q = Output(Vec(n, UInt(bw.W)))
  })
  val register_file = RegInit(VecInit.fill(n)(0.U(bw.W)))
  val register_layer = for (i <-0 ≤ until < n) yield {
    register_file(i) := io.in_d(i)
    io.out_q(i) := register_file(i)
  }
}
```

Shift Register

1. Shift Register

```
class shift_register(depth: Int,bw: Int) extends Module{
  val io = IO(new Bundle() {
    val in = Input(UInt(bw.W))
    val out = Output(UInt(bw.W))
  })
  val reg = RegInit(VecInit.fill(depth)(0.U(bw.W)))
    reg(0) := io.in
    for(i <- 1 ≤ until < depth){
      reg(i) := reg(i-1)
    }
  io.out := reg(depth - 1)
}
```

Register files/Sync SRAM

1. Synchronous SRAM/Register File

LBL-ICS

```
class syn_ram(bw: Int, depth: Int) extends Module {
  val io = IO {
    new Bundle() {
      val ena = Input(Bool())
      val enb = Input(Bool())
      val wea = Input(UInt((bw/8).W))
      val addra = Input(UInt((log2Ceil(depth).W)))
      val addrb = Input(UInt((log2Ceil(depth).W)))
      val dina = Input(UInt(bw.W))
      val doutb = Output(UInt(bw.W))
    }
  }
  val mem = RegInit(VecInit.fill(depth)(VecInit.fill(bw / 8)(0.U(8.W))))
    when(io.ena) {
      for (i <- 0 ≤ until < bw / 8) {
        when(io.wea(i)) {
          mem(io.addra)(i) := io.dina(8 * (i + 1) - 1, 8 * i)
        }
      }
    }

  val data_out = RegInit(0.U(bw.W))
    when(io.enb) {
      data_out := Cat(mem(io.addrb).reverse)
    }
  io.doutb := data_out
}
```

Register files/Async SRAM

1. Asynchronous SRAM/Register File

```
LBL-ICS
class asyn_ram(bw: Int, depth: Int) extends Module {
  val io = IO(new Bundle {
    val clka = Input(Clock()) // Write clock
    val clkb = Input(Clock()) // Read clock
    val ena = Input(Bool()) // Enable write
    val enb = Input(Bool()) // Enable read
    val wea = Input(Vec(bw / 8, Bool())) // Write enable mask (one bit per byte)
    val addra = Input(UInt(log2Ceil(depth).W)) // Write address
    val addrb = Input(UInt(log2Ceil(depth).W)) // Read address
    val dina = Input(UInt(bw.W)) // Write data
    val doutb = Output(UInt(bw.W)) // Read data
  })

  // Memory Array (depth x bw bits wide)
  val mem = SyncReadMem(depth, UInt(bw.W)) // Memory for storing `bw`-bit wide words
```

```
// Write logic on clka with byte-level write enable mask
withClock(io.clka) {
  val data_in = WireDefault(0.U(bw.W))
  when(io.ena) {
    // Byte-wise write masking
    for (i <- 0 ≤ until < bw / 8) {
      when(io.wea(i)) { // Only write the byte if the corresponding bit in wea is set
        data_in := io.dina(8 * (i + 1) - 1, 8 * i)
      }
    }
    mem.write(io.addra, data_in) // Write back the modified word
  }
}
```

```
// Read logic on clkb
withClock(io.clkb) {
  val data_out = Wire(UInt(bw.W))
  when(io.enb) {
    data_out := mem.read(io.addrb) // Read from memory on clkb
  }.otherwise {
    data_out := 0.U // Output zero when not enabled
  }
  io.doutb := data_out
}
}
```

Instantiation

1. Use Vector.fill(n) to build the vector for modules instantiation, and then connection with zipWithIndex.map

2. Alternative: for loop

```
class parallel_multiplier(n: Int, bw: Int) extends Module {
  val io = IO(new Bundle {
    val in_a = Input(Vec(n, UInt(bw.W)))
    val in_b = Input(Vec(n, UInt(bw.W)))
    val out_s = Output(Vec(n, UInt(bw.W)))
  })
  val multipliers = Vector.fill(n)(Module(new FP_multiplier_10ccs(bw)).io)
  multipliers.zipWithIndex.map(x=>x._1.in_a := io.in_a(x._2))
  multipliers.zipWithIndex.map(x=>x._1.in_b := io.in_b(x._2))
  multipliers.zipWithIndex.map(x=>io.out_s(x._2) := x._1.out_s)
  multipliers.map(_.in_en := true.B)
  //val multiplier_layer = for (i <-0 ≤ until < n) yield {
  //  val multiplier = Module(new FP_multiplier_10ccs(bw)).io
  //  multiplier.in_en := true.B
  //  multiplier.in_a := io.in_a(i)
  //  multiplier.in_b := io.in_b(i)
  //  io.out_s(i) := multiplier.out_s
  //}
}
```

Cheat Sheet

Operators:

Chisel	Explanation	Width
`!x`	Logical NOT	`1`
`x && y`	Logical AND	`1`
`x || y`	Logical OR	`1`
`x(n)`	Extract bit, `0` is LSB	`1`
`x(n, m)`	Extract bitfield	`n - m + 1`
`x << y`	Dynamic left shift	`w(x) + maxVal(y)`
`x >> y`	Dynamic right shift	`w(x) - minVal(y)`
`x << n`	Static left shift	`w(x) + n`
`x >> n`	Static right shift	`w(x) - n`
`Fill(n, x)`	Replicate `x`, `n` times	`n * w(x)`
`Cat(x, y)`	Concatenate bits	`w(x) + w(y)`
`Mux(c, x, y)`	If `c`, then `x`; else `y`	`max(w(x), w(y))`
`~x`	Bitwise NOT	`w(x)`
`x & y`	Bitwise AND	`max(w(x), w(y))`
`x | y`	Bitwise OR	`max(w(x), w(y))`
`x ^ y`	Bitwise XOR	`max(w(x), w(y))`
`x === y`	Equality(triple equals)	`1`
`x != y`	Inequality	`1`
`x =/= y`	Inequality	`1`
`x + y`	Addition	`max(w(x),w(y))`
`x +% y`	Addition	`max(w(x),w(y))`
`x +& y`	Addition	`max(w(x),w(y))+1`
`x - y`	Subtraction	`max(w(x),w(y))`
`x -% y`	Subtraction	`max(w(x),w(y))`
`x -& y`	Subtraction	`max(w(x),w(y))+1`
`x * y`	Multiplication	`w(x)+w(y)`
`x / y`	Division	`w(x)`
`x % y`	Modulus	`bits(maxVal(y)-1)`
`x > y`	Greater than	`1`
`x >= y`	Greater than or equal	`1`
`x < y`	Less than	`1`
`x <= y`	Less than or equal	`1`
`x >> y`	Arithmetic right shift	`w(x) - minVal(y)`
`x >> n`	Arithmetic right shift	`w(x) - n`

Math Helpers:

`log2Ceil(in:Int): Int` log_2(`in`) rounded up

`log2Floor(in:Int): Int` log_2(`in`) rounded down

`isPow2(in:Int): Boolean` `True` if `in` is a power of 2

References

1. Chisel Env Setup – XY. pdf
2. Chisel and IC Projects Structuring – XY. pdf
3. Chisel Bootcamp
4. Chisel3 Cheat sheet
5. Chisel textbook, version 5
6. Chisel Designs for Fundamental IC, Git repository: https://github.com/LBL-ICS/Chisel-for-Fundamental-IC.git
7. Xiaokun Yang, “Integrated Circuit Design: IC Design Flow and Project-Based Learning”, CRC Press -Taylor & Francis Group, ISBN: 978-1-032-03079-1 (ebook ISBN: 978-1-003-18708-0), First edition
8. Project Structuring for IC Design and Simulation, Git repository: https://github.com/LBL-ICS/IC-Design.git