Chiselを始めたい人に読んでほしい本（七夕雅俊)_01章

はじめに

ChielはRISC-V(Rocket ChipやBOOM)、GoogleのEdge TPUの実装に用いられたプログラミング言語。
あんまり日本語情報がないらしく(当時、今はWebサイトは少ないが本はちょいちょい引っ掛かる)のでこの本をまとめたとのこと。
最終的にVerilog-HDLのコードを生成する。System-Cと似たような感じか？Scalaの知識も多少必要らしい。（関数型言語。やったことない。。）
Chisel 3.3を対象。

スタイル

Scala（camelCase）

`val someDataBits = 100`

Chisel（snake_case）

`val w_some_data = 0.U`

Chisel（その他）

```
// wire
val w_some_flag = false.B
// reg
val r_somee_data = 0.U(8.W)
// module instance
val m_some_module = Module(new SomeModule)
// Bundle
class SomeBundle extends Bundle {
  some_input = Bool()
}
```

Chiselでは以下の点がVerilogと異なる

- ネットとレジスタの信号の記述が〃

- モジュールのインスタンスと同時に接続が行われない。Scala変数にモジュールのインスタンスが格納される。

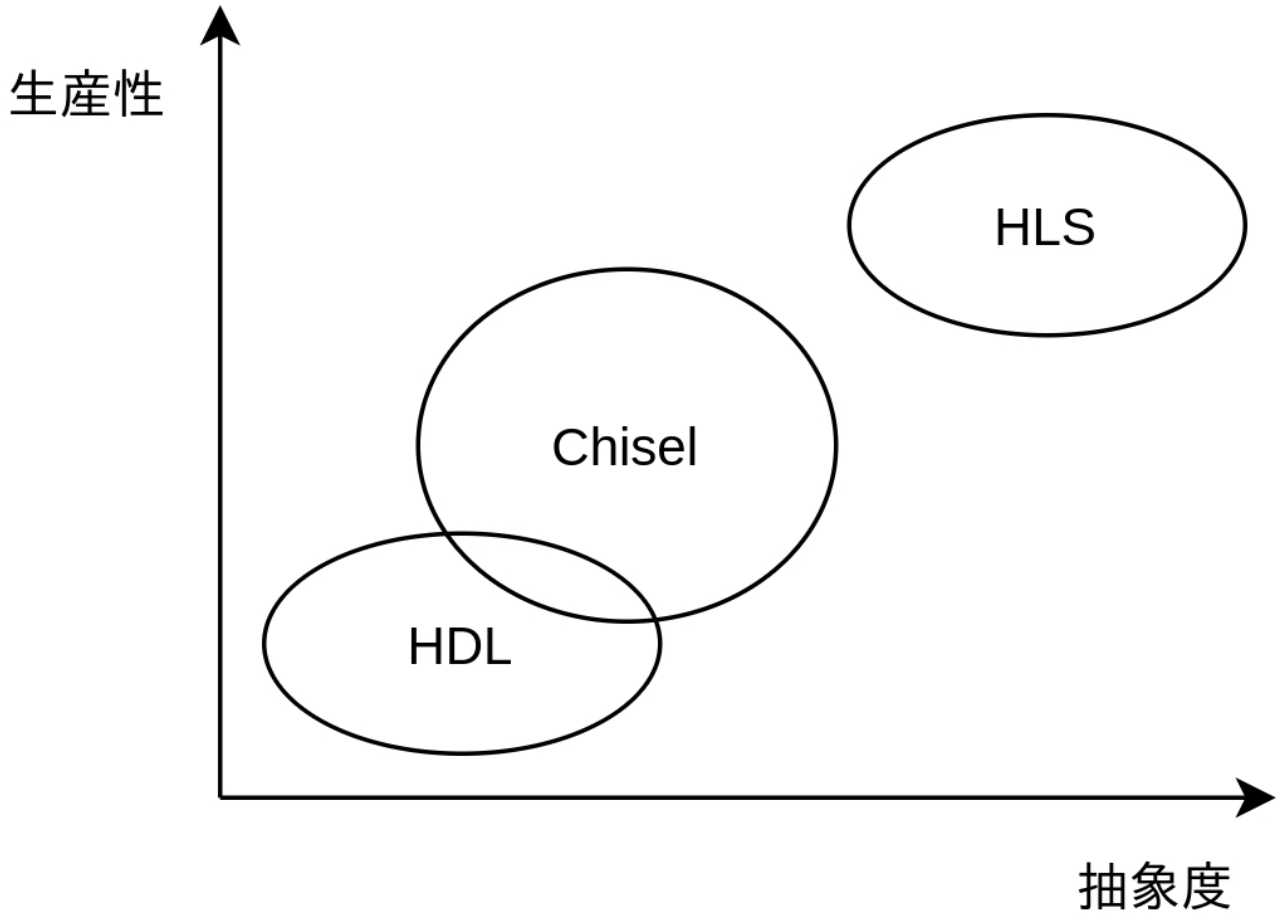
経緯

HDLはそもそもハードウェアの趣味レーション用言語で、サブセットとして論理合成可能な記述が推測できるような形式だった(FF推定記述)
これらの言語は抽象化などの概念を持っていないため、複雑・大規模な設計に対して問題が顕在化し、その解としてSystem Verilogが登場した。ただし、依然ソフトウェアのような機能のサポートはされていない状態。
Chisel(Construction Hardware In a Scala Embedded Language)はUC berkleyがScalaの内部DSL(Domain Specific Language/ドメイン固有言語：特定の用途向けに設計された言語、ここではScalaの文法を用いてHWを設計するための言語拡張エクステンションと思えばいい、DSLの利点としては汎用言語に特定領域への特化を行わせると結果として言語思想との齟齬が発生し汎用性が失われるが、DSLであれば目的に応じて取捨選択できるので言語本体の思想への影響をなくせる。)として開発され、Scalaの関数型言語、オブジェクト指向のメリットを享受できる言語となっている。DAC2021で論文が発表されている。当時のChiselはChisel 2.xだが、現状では非推奨(deprecated)となっている。※今もChisel 3.x系が主流みたい。3.6が最新っぽい。

立ち位置

抽象度的にはHDLと同じレイヤー。扱うのはネットやレジスタといった論理回路のプリミティブなd－多で、HDLと同様の扱いをすることも可能。Scalaに備わったモダンな機能を使うことで、作成するモジュールをパラメタライズし、一つのChiselモジュールから複数の異なったRTLを生成することが可能となっている。このため、HCL（Hardware Construction Language)と呼称されている。
ただし、ChiselにはHLS(High Level Synthesis)のようなアルゴリズムをハードウェアに変更するような機能は備わっていない。(そのためHLSとは異なるというのが著者の見解)
その代わりにパラメタライズして抽象度を上げ、生産性を向上させることができる。

図1.1著者の考えるChiselのポジションより抜粋。



Fifoの例

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

/**
    * FIFO リード側 I/O
    */
class FIFORdIO(bits: Int) extends Bundle {
    val enable = Input(Bool())
    val empty = Output(Bool())
    val data = Output(UInt(bits.W))
}

/**
    * FIFO ライト側 I/O
    */
class FIFOWrIO(bits: Int) extends Bundle {
    val enable = Input(Bool())
    val full = Output(Bool())
    val data = Input(UInt(bits.W))
```

```
}

/**
    * FIFO I/O
    * @param bits データのビット幅
    * @param depth FIFOの段数
    * @param debug trueでデバッグモード
    */
class FIFOIO(bits: Int, depth: Int = 16, debug: Boolean = false)
    extends Bundle {
val depthBits = log2Ceil(depth)

    val wr = new FIFOWrIO(bits)
    val rd = new FIFORdIO(bits)

    val dbg = if (debug) { Some(Output(new Bundle {
        val r_wrptr = Output(UInt(depthBits.W))
        val r_rdptr = Output(UInt(depthBits.W))
        val r_data_ctr = Output(UInt((depthBits + 1).W))
    })) } else {
        None
    }

    override def cloneType: this.type =
        new FIFOIO(bits, depth, debug).asInstanceOf[this.type]
}

/**
    * 単純なFIFO
    * @param dataBits データのビット幅
    * @param depth FIFOの段数
    * @param debug trueでデバッグモード
    */
class FIFO(dataBits: Int = 8, depth: Int = 16, debug: Boolean = false)
    extends Module {

    // parameter
    val depthBits = log2Ceil(depth)

    def ptrWrap(ptr: UInt): Bool = ptr === (depth - 1).U
```

```
val io = IO(new FIFOIO(dataBits, depth, debug))

val r_fifo = RegInit(VecInit(Seq.fill(depth)(0.U(dataBits.W))))
val r_rdptr = RegInit(0.U(depthBits.W))
val r_wrptr = RegInit(0.U(depthBits.W))
val r_data_ctr = RegInit(0.U((depthBits + 1).W))

// リードポインタ
when(io.rd.enable) {
    r_rdptr := Mux(ptrWrap(r_rdptr), 0.U, r_rdptr + 1.U)
}

// ライトポインタ
when(io.wr.enable) {
    r_fifo(r_wrptr) := io.wr.data
    r_wrptr := Mux(ptrWrap(r_wrptr), 0.U, r_wrptr + 1.U)
}

// データカウント
when (io.wr.enable && io.rd.enable) {
    r_data_ctr := r_data_ctr
} .otherwise {
    when (io.wr.enable) {
        r_data_ctr := r_data_ctr + 1.U
    }
    when (io.rd.enable) {
        r_data_ctr := r_data_ctr - 1.U
    }
}

// IOとの接続
io.wr.full := r_data_ctr === depth.U
io.rd.empty := r_data_ctr === 0.U
io.rd.data := r_fifo(r_rdptr)

// テスト用のデバッグ端子の接続
if (debug) {
    io.dbg.get.r_wrptr := r_wrptr
    io.dbg.get.r_rdptr := r_rdptr
```

```
        io.dbg.get.r_data_ctr := r_data_ctr
    }
}

/**
    * FIFOのRTL生成処理
    */
object ElaborateFIFO extends App {
    chisel3.Driver.execute(Array(""), () => new chapter1.FIFO(16))
}
```

Chiselには標準のテスト機構がある。こいつをモジュールに仕込んでおくとテストが楽になる。

```
/**
    * FIFOの単体テストクラス
    * @param c FIFOモジュールのインスタンス
    */
class FIFOUnitTester(c: FIFO) extends PeekPokeTester(c) {

    /**
        * アイドル
        */
    def idle(): Unit = {
        poke(c.io.rd.enable, false)
        poke(c.io.wr.enable, false)
        step(1)
    }

    /**
        * FIFOにデータを書き込む
        * @param data データ
        */
    def push(data: BigInt): Unit = {
        poke(c.io.wr.enable, true)
        poke(c.io.wr.data, data)
        step(1)
    }

    /**
```

```
        * FIFOのデータを読みだし、期待値と比較
        * @param exp 期待値
        */
    def pop(exp: BigInt): Unit = {
        expect(c.io.rd.data, exp)
        poke(c.io.rd.enable, true)
        step(1)
    }

    /**
        * プッシュとポップを同時に行う
        * @param data 設定するデータ
        * @param exp 期待値
        */
    def pushAndPop(data: BigInt, exp: BigInt): Unit = {
        expect(c.io.rd.data, exp)
        poke(c.io.rd.enable, true)
        poke(c.io.wr.enable, true)
        poke(c.io.wr.data, data)
        step(1)
    }
}
```

```
/**
    * FIFOのテストクラス
    */
class FIFOTester extends ChiselFlatSpec {
    val dutName = "chapter1.FIFO"
    val dataBits = 8
    val depth = 16

    it should "ホストがpushを実行すると、FIFOにデータが書き込まれる [FIFO-000]" in {
        val outDir = dutName + "-fifo-push"
        val args = Array(
            "--top-name", dutName,
            "--target-dir", s"test_run_dir/$outDir",
            "-tgvo=on"
        )
```

```
        Driver.execute(args, () => new FIFO(dataBits, depth, true)) {
            c => new FIFOUnitTester(c) {
                val setData = Range(0, 16).map(_ => floor(random * 256).toInt)

                expect(c.io.rd.empty, true)
                for ((data, idx) <- setData.zipWithIndex) {
                    push(data)
                    expect(c.io.rd.empty, false)
                    expect(c.io.rd.data, setData(0))
                }
                idle()
            }
        } should be (true)
    }
}
```

この辺のことはverilogでもできる。