Chisel を用いたエッジ検出フィルタの実装と Vivado HLS との比較

佐々木　隆汰

December 2020

Contents

Contents

Chisel の導入

- JDK と sbt を導入
  (https://www.scala-sbt.org/)
- chisel-template を clone
  (https:
  //github.com/freechipsproject/chisel-template)
- プロジェクト名をリネーム

Chisel の文法

- 各モジュールは Module クラスを拡張して作成。

Listing 1: L チカモジュール

```
1  class Hello extends Module {
2       val io = IO(new Bundle {
3          val led = Output (UInt (1.W))
4       })
5       val CNT_MAX = (50000000 / 2 - 1).U;
6
7       val cntReg = RegInit(0.U(32.W))
8       val blkReg = RegInit(0.U(1.W))
9
10      cntReg := cntReg + 1.U
11      when (cntReg === CNT_MAX) {
12         cntReg := 0.U
13         blkReg := ~blkReg
14      }
15      io.led := blkReg
16 }
```

信号型と定数

- ビットのベクトルを表す 3 つのデータ型。

```
1 Bits(8.W)
2 UInt(8.W)
3 SIng(8.W)
```

- データ幅は Width 型で表す。
- 定数データを記述するときはメソッド呼び出しを用いる。

```
1 8.U(4.W)     // ビット幅は引数で指定
2 0.U
3 -3.S         // ビット幅は指定しなくても推論してくれる
```

- 論理型（UInt(1.W) の拡張）

```
1 Bool()
2 true.B
3 false.B
```

組み合わせ回路

- 論理演算子

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

- 算術演算子

```
1  val add = a + b
2  val sub = a - b
3  val neg = - a
4  val mul = a * b
5  val div = a / b
6  val mod = a % b
```

組み合わせ回路

- 信号タイプを先に定義してから、更新演算子を使用して信号に値を割り当てることもできる。

```
1 val w = Wire (UInt())
2 w := a & b
```

- 1 ビットの抽出

```
1 val sign = x(31)
```

- 範囲指定で抜き出し

```
1 val lowByte = largeWord(7, 0)
```

- 信号の結合

```
1 val word = Cat(highByte, lowByte)
```

組み合わせ回路

- マルチプレクサ

```
1 val y = Mux (sel, a, b)
```

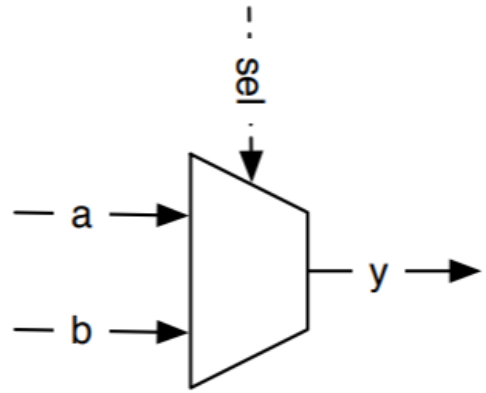


Figure 2.2: A basic 2:1 multiplexer.

レジスタ

- リセット時に 0 で初期化される 8 ビットレジスタ

```
1 val reg = RegInit(0.U(8.W))
2 reg := d                          // 更新演算子で更新
3 val q = reg                       // レジスタの値を取り出す
```

- 初期値が不定 (X) のレジスタ

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

- カウンター

```
1 val cntReg = RegInit(0.U(8.W))
2 cntReg := Mux (cntReg === 9.U, 0.U, cntReg + 1.U)
```

Bundle と Vec

- Bundle: 異なるタイプの複数の信号をまとめる。

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

- フィールドにはドット表記でアクセス。

```
val ch = Wire (new Channel())
ch.data := 123.U                    // 更新演算子で更新できる
ch.valid := true.B                  // 更新演算子で更新できる
val b = ch.valid                    // フィールドだけを参照
val channel = ch                    // 全体を参照
```

- Vec: 同じタイプの複数の信号をまとめる。

```
val v = Wire (Vec (3, UInt(4.W)))
v(0) := 1.U                         // アクセスは配列と同じ
v(1) := 3.U                         // アクセスは配列と同じ
v(2) := 5.U                         // アクセスは配列と同じ
```

テスト

PeekPokeTester で IO ポートからデータを入出力できる

```
1  class DeviceUnderTest extends Module {
2      val io = IO (new Bundle {
3          val a = Input (UInt(32.W))
4          val b = Input (UInt(32.W))
5          val out = Output (UInt(2.W))
6      })
7      io.out := io.a & io.b
8  }
```

```
1  class TesterSimple (dut: DeviceUnderTest)
2          extends PeekPokeTester(dut) {
3      poke (dut.io.a, 0.U)
4      poke (dut.io.b, 1.U)
5      step (1)
6      println ("Result is: " + peek(dut.io.out).toString)
7      poke (dut.io.a, 4.U)
8      poke (dut.io.b, 2.U)
9      step (1)
10     expect (dut.io.out, 0)
11 }
```

Contents

- ✓ Chisel の導入
- ▷ Pros. & Cons.
- ▶ ChiselImProc の説明
- ▶ Vivado HLS との比較

Pros. & Cons.

Pros.

- OOP の強みは大体使える。
- 型パラメータを利用できる。
- 再利用性は高い。
- 出力された v ファイルと元の Scala のコードの対応が見やすい。

Cons.

- クロックは常に意識しないと厳しい。
- AXI は chisel3 だと結局ない（？）
- v ファイルがやたら重い。
- XRESET には対応していない。

AXI Stream

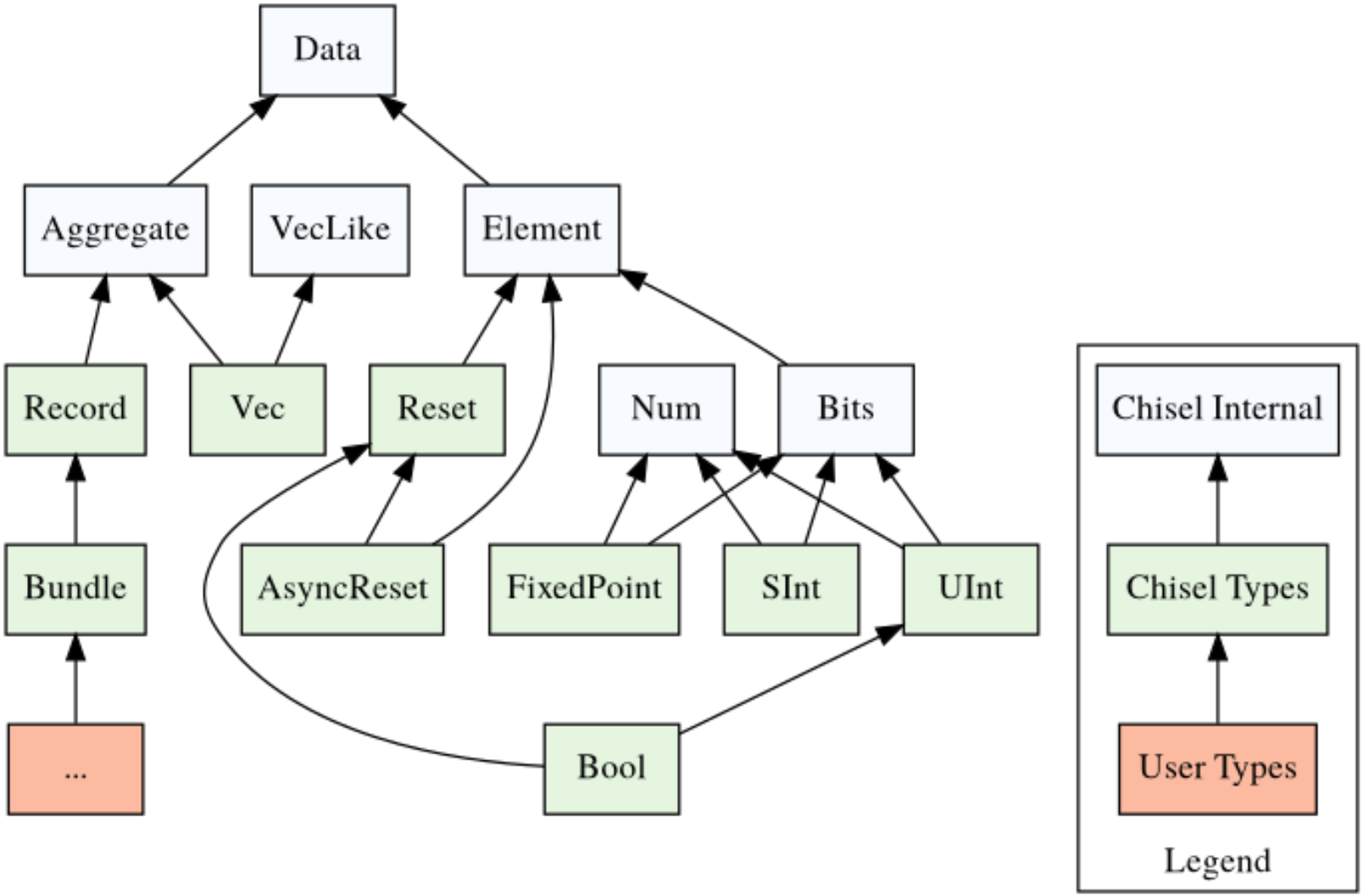
- 型パラメータを利用できる。
- 上限境界 (<:) なども利用できる。

Listing 2: "AXI Stream Interface"

```
1  class AXIStreamIF[T <: Data](gen: T)
2          extends ReadyValidIO[T](gen) {
3      val user = Output (Bool())
4      val last = Output (Bool())
5
6      override def cloneType: this.type
7          = new AXIStreamIF(gen).asInstanceOf[this.type]
8  }
9
10 object AXIStreamMasterIF {
11     def apply [T <: Data] (gen: T): AXIStreamIF[T]
12         = new AXIStreamIF[T] (gen)
13 }
14
15 object AXIStreamSlaveIF {
16     def apply [T <: Data] (gen: T): AXIStreamIF[T]
17         = Flipped (new AXIStreamIF[T] (gen))
18 }
```

- Data 型は Chisel のすべてのデータの supertype

Data Types Overview



AXI Stream を用いたモジュール

```
1  class FifoAXIStreamDIO[T <: Data, U <: Data] (
2      private val genEnq: T,
3      private val genDeq: U
4  ) extends Bundle {
5      val enq = AXIStreamSlaveIF(genEnq)
6      val deq = AXIStreamMasterIF(genDeq)
7  }
8
9  class FifoAXIStreamIO[T <: Data] (private val gen: T)
10         extends FifoAXIStreamDIO(gen, gen) {
11 }
12
13 abstract class FifoAXIS[T <: Data] (gen: T, depth: Int)
14         extends Module {
15     val io = IO (new FifoAXIStreamIO[T] (gen))
16     assert (depth > 0,
17         "Number of buffer elements needs to be larger than 0")
18 }
```

※ enq と deq は Chisel と僕のコードで逆。

積和モジュール

- 8bit で 3x3 は遅延しない（1 クロックで動作できる）
- 8bit で 5x5 は遅延する（1 クロックで動作できない）
- 16bit で 3x3 は遅延？

```
1  class MulAdd (dataWidth: Int, num: Int) extends Module {
2      val io = IO(new Bundle {
3          val a = Input (Vec(num, UInt(dataWidth.W)))
4          val b = Input (Vec(num, UInt(dataWidth.W)))
5          val output = Output (UInt ((2*dataWidth).W))
6      })
7
8      var i = 0
9      var tmp = 0.U((2*dataWidth).W)
10     val step = 3                    // 8bit で3x3は遅延しない
11     for (i <- 0 until num by step) {
12         var intmp = 0.U((2*dataWidth).W)
13         for (j <- 0 until step) {
14             intmp += io.a(i+j) * io.b (i+j)
15         }
16         tmp += intmp
17     }
18     io.output := tmp
19 }
```

Verilog ファイル生成

```
1  module MulAdd(
2    input  [7:0]  io_b_0,
3    input  [7:0]  io_b_1,
4    input  [7:0]  io_b_2,
5    input  [7:0]  io_b_3,
6    input  [7:0]  io_b_4,
7    input  [7:0]  io_b_5,
8    input  [7:0]  io_b_6,
9    input  [7:0]  io_b_7,
10   input  [7:0]  io_b_8,
11   output [15:0] io_output
12 );
13   wire [15:0] _T = 8'h1 * io_b_0; // @[ChiselImProc.scala 32:32]
14   wire [16:0] _T_1 = {{1'd0}, _T}; // @[ChiselImProc.scala 32:19]
15   wire [15:0] _T_3 = 8'h2 * io_b_1; // @[ChiselImProc.scala 32:32]
16   wire [15:0] _T_5 = _T_1[15:0] + _T_3; // @[ChiselImProc.scala 32:19]
17   wire [15:0] _T_6 = 8'h1 * io_b_2; // @[ChiselImProc.scala 32:32]
18   wire [15:0] _T_8 = _T_5 + _T_6; // @[ChiselImProc.scala 32:19]
19   wire [16:0] _T_9 = {{1'd0}, _T_8}; // @[ChiselImProc.scala 34:13]
20   wire [15:0] _T_11 = 8'h2 * io_b_3; // @[ChiselImProc.scala 32:32]
21   wire [16:0] _T_12 = {{1'd0}, _T_11}; // @[ChiselImProc.scala 32:19]
22   wire [15:0] _T_14 = 8'h4 * io_b_4; // @[ChiselImProc.scala 32:32]
23   wire [15:0] _T_16 = _T_12[15:0] + _T_14; // @[ChiselImProc.scala 32:19]
24   wire [15:0] _T_17 = 8'h2 * io_b_5; // @[ChiselImProc.scala 32:32]
25   wire [15:0] _T_19 = _T_16 + _T_17; // @[ChiselImProc.scala 32:19]
26   wire [15:0] _T_21 = _T_9[15:0] + _T_19; // @[ChiselImProc.scala 34:13]
27   wire [15:0] _T_22 = 8'h1 * io_b_6; // @[ChiselImProc.scala 32:32]
28   wire [16:0] _T_23 = {{1'd0}, _T_22}; // @[ChiselImProc.scala 32:19]
29   wire [15:0] _T_25 = 8'h2 * io_b_7; // @[ChiselImProc.scala 32:32]
30   wire [15:0] _T_27 = _T_23[15:0] + _T_25; // @[ChiselImProc.scala 32:19]
31   wire [15:0] _T_28 = 8'h1 * io_b_8; // @[ChiselImProc.scala 32:32]
32   wire [15:0] _T_30 = _T_27 + _T_28; // @[ChiselImProc.scala 32:19]
33   assign io_output = _T_21 + _T_30; // @[ChiselImProc.scala 36:15]
34 endmodule
```

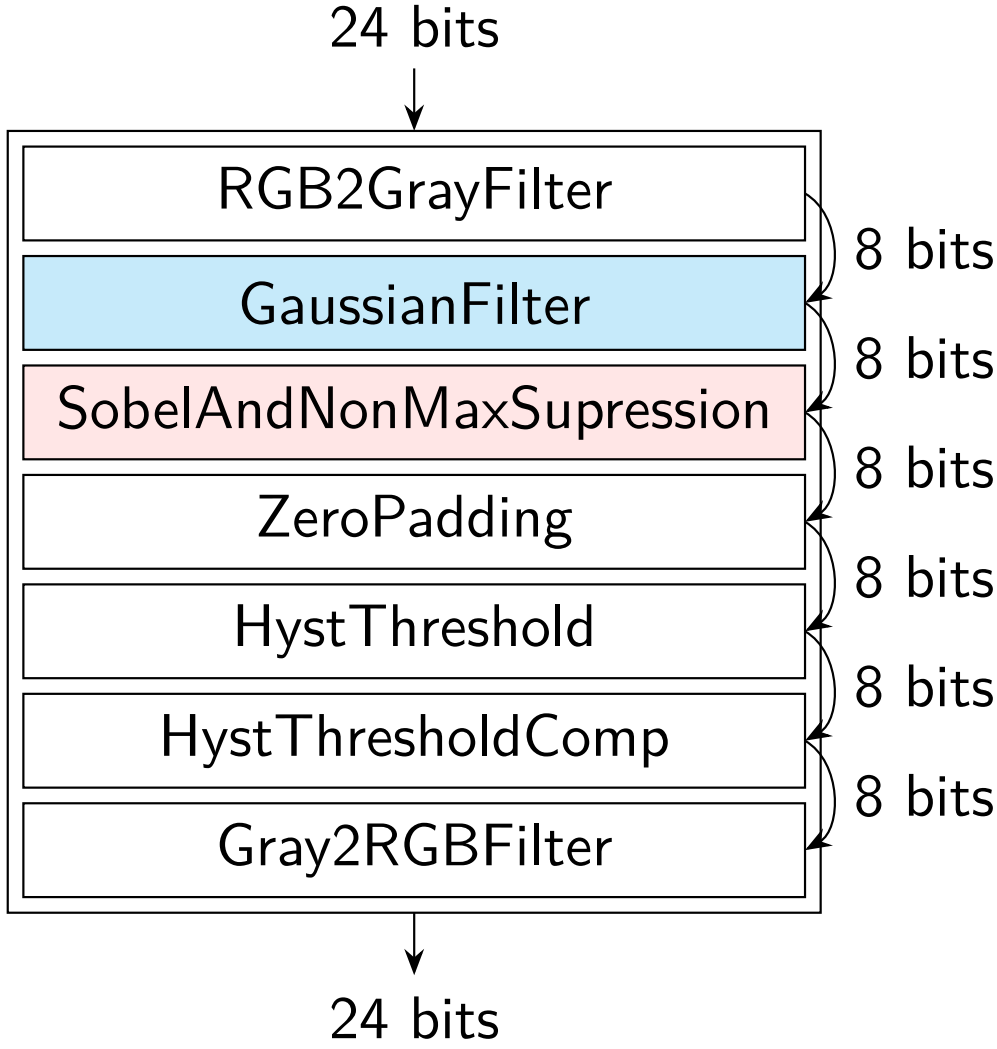
Verilog ファイル生成

- IO, Register, Wire は変数名がそのまま使われる。
- フィールド名は_で分けられる。
- 計算途中のデータなどは_T_番号で名付けられる。
- コメントで元の Scala のどこの部分と対応しているか明記される。
- 使われていない Wire や Register などは消される。
- 出力の v ファイルを分割できない。
  -fsm, –split-modules オプションをつければ分けれそうだが分けれない。

Contents

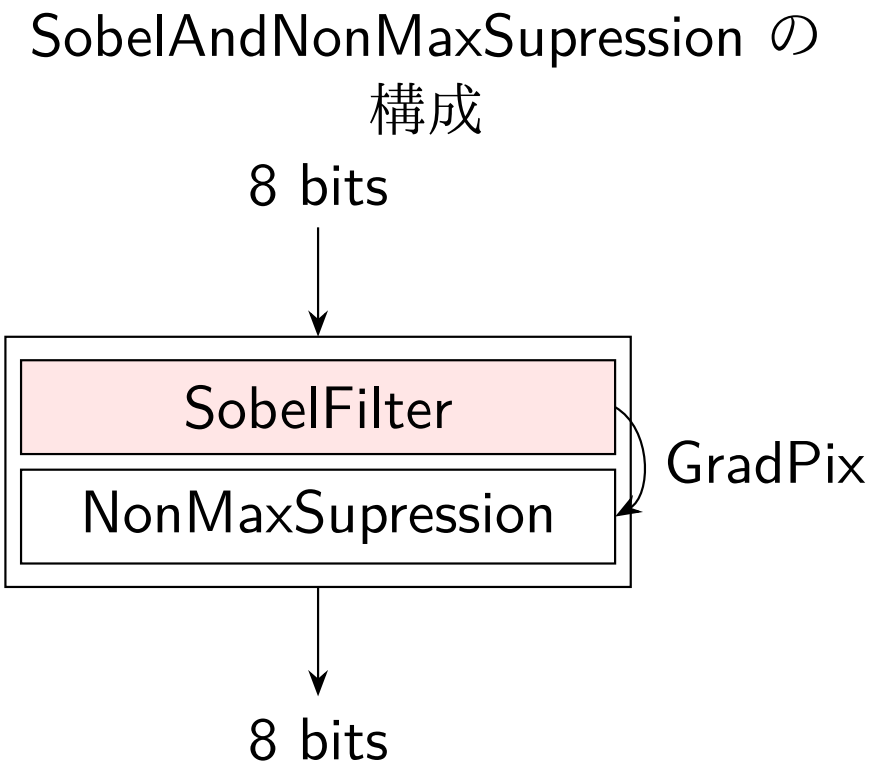
- ✓ Chisel の導入
- ✓ Pros. & Cons.
- ▷ ChiselImProc の説明
- ▶ Vivado HLS との比較

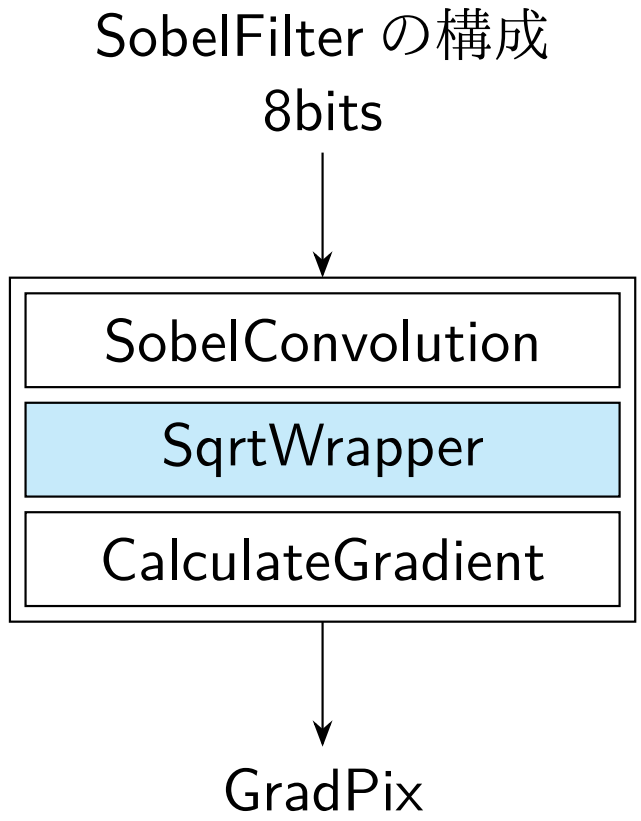
ChiselImProc の全体構成



- Gaussian Filter は 8bit の 5x5 サイズの畳み込みで遅延。
- 3x3 サイズに落として計算。

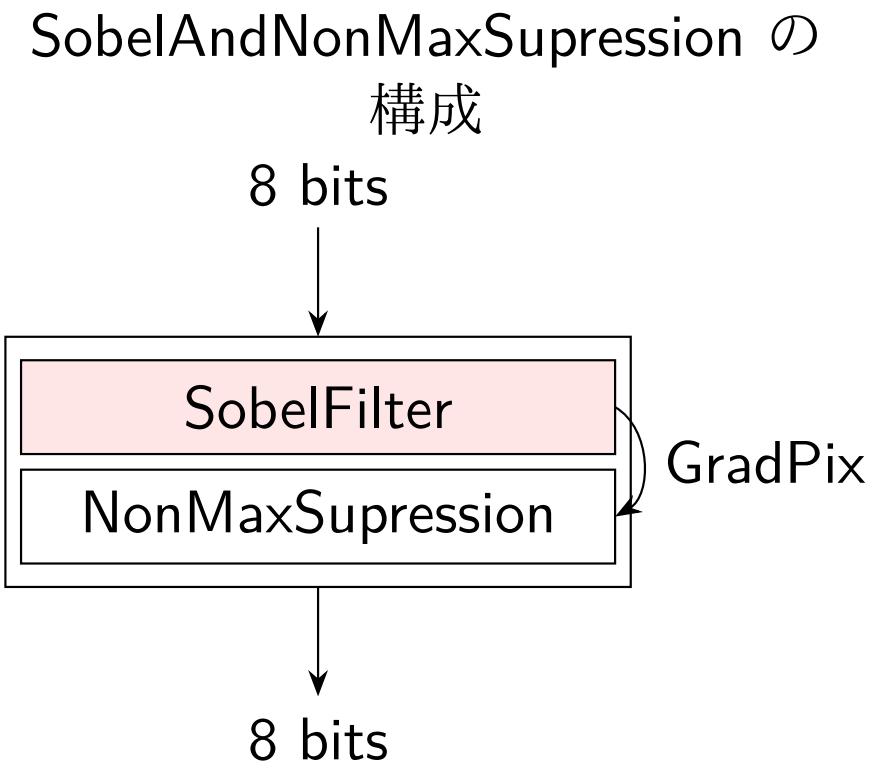
SobelAndNonMaxSupression の構成

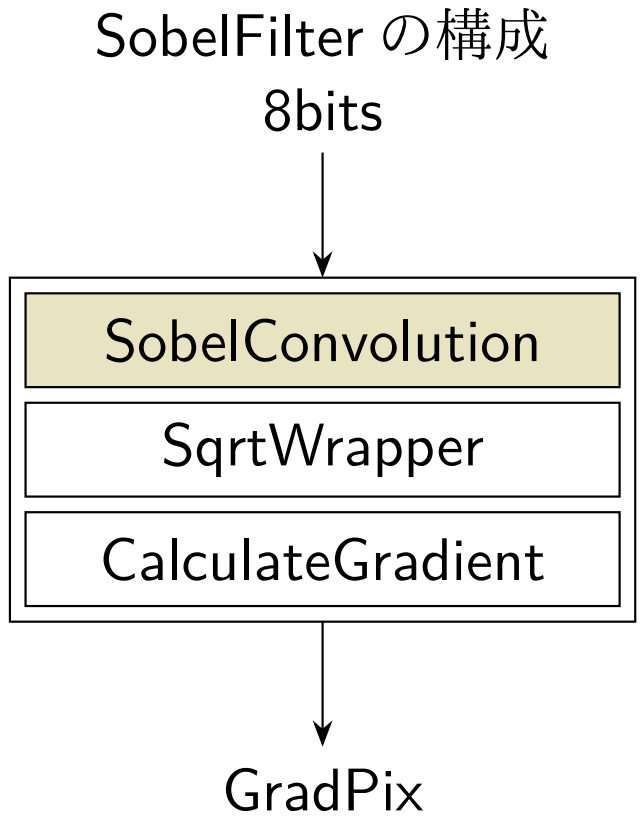




32 bit の開平法は遅延するので、レジスタを導入。

SobelAndNonMaxSupression の構成





16 bit の 3x3 サイズの畳み込みが遅い可能性。

Contents

- ✓ Chisel の導入
- ✓ Pros. & Cons.
- ✓ ChiselImProc の説明
- ▷ Vivado HLS との比較

Lenna の画像によるテストベンチ

Vivado HLS

Chisel

- HLS のほうだとなぜか 2 値になっていない
- last 信号の処理が甘いため 20 ピクセルほどずれる

Lenna の画像によるテストベンチ

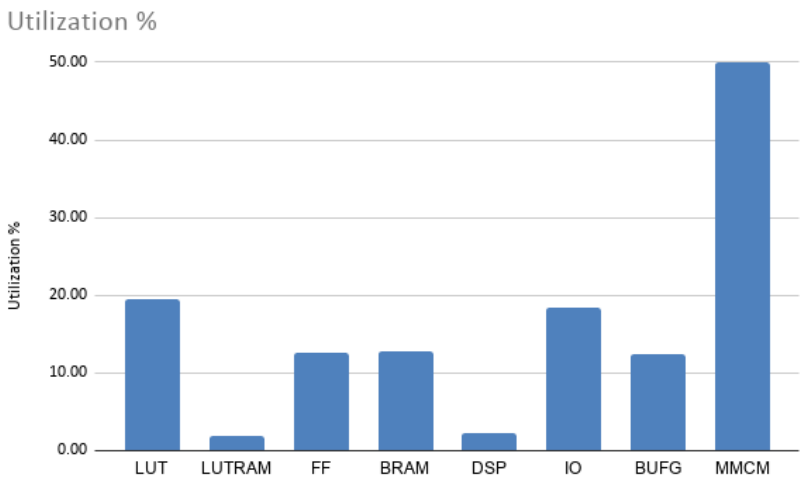
Vivado HLS

Chisel

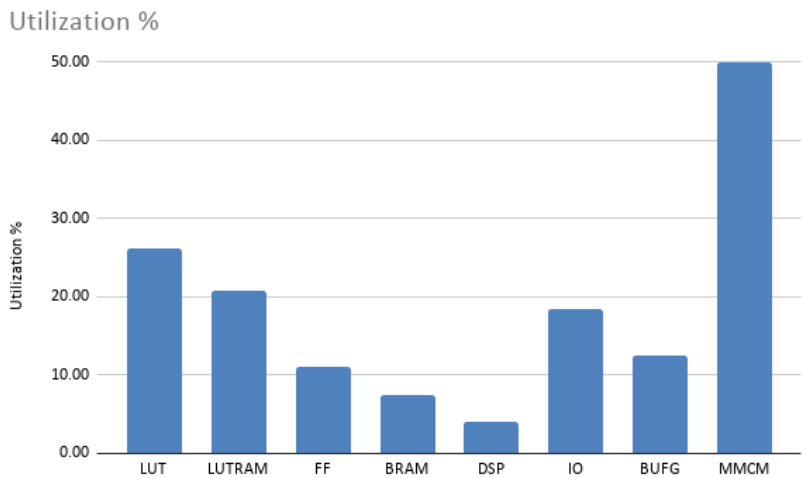
- HLS のほうだとなぜか 2 値になっていない
- last 信号の処理が甘いため 20 ピクセルほどずれる

Implementation 後の Utilization

Vivado HLS



Chisel



Chisel の BRAM は System ila $(+15 \sim 20\%)$ が含まれる。

参考文献

- Digital Design with Chisel
  https://raw.githubusercontent.com/wiki/schoeberl/chisel-book/chisel-book.pdf https://github.com/schoeberl/chisel-book
- Chisel 入門書「Digital Design with Chisel」1 章の勉強記録
  https://qiita.com/Kosuke_Matsui/items/80dde1219e3ce02c4b66
- Chisel 3 Wiki
  https://github.com/chipsalliance/chisel3/wiki
- Scala Doc
  https://www.chisel-lang.org/api/latest/chisel3/index.html
- Cheet Sheet
  https://github.com/freechipsproject/chisel-cheatsheet/releases/latest/download/chisel_cheatsheet.pdf