# ScalaHDL

by Yao Li

# Outline

- Comparing with MyHDL
- Issues to be discussed
- Future Plan

# Comparing with MyHDL

```
def compare(a1, a2, z1, z2, dir):

    @always_comb
    def logic():
        z1.next = a1
        z2.next = a2
        if dir == (a1 > a2):
            z1.next = a2
            z2.next = a1

    return logic
```

```
def compare(a: HDLType, b: HDLType,
  x: HDLType, y: HDLType, dir: Int) {
  if (dir == ASC) {
    when (a > b) {
      x := b
      y := a
    } .otherwise {
      x := a
      y := b
    }
  } else {
    when (a > b) {
      x := a
      y := b
    } .otherwise {
      x := b
      y := a
    }
  }
}
```

## Comparing with MyHDL

```
def compare(a1, a2, z1, z2, dir):
```

don’t need
two levels of
function definitions

```
        z1.next = a1

        z2.next = a2

        if dir == (a1 > a2):

            z1.next = a2

            z2.next = a1
```

don’t need to return
the inner function
in the end of the outter function

```
def compare(a: HDLType, b: HDLType,
  x: HDLType, y: HDLType, dir: Int) {
  if (dir == ASC) {
    when (a > b) {
      x := b
      y := a
    } .otherwise {
      x := a
      y := b
    }
  } else {
    when (a > b) {
      x := a
      y := b
    } .otherwise {
      x := b
      y := a
    }
  }
}
```

cannot just write
if (dir == (a > b))
now

# Comparing with MyHDL

```
def bitonicMerge(a, z, dir):
    …
    if n > 1:
        t = [Signal(intbv(0)[w:]) for i in range(n)]
        comp = [compare(a[i], a[i+k], t[i], t[i+k], dir) for i in range(k)]
        loMerge = bitonicMerge(t[:k], z[:k], dir)
        hiMerge = bitonicMerge(t[k:], z[k:], dir)
        return comp, loMerge, hiMerge
    else:
        feed = feedthru(a[0], z[0])
        return feed
```

---

```
def bitonicMerge(a: Seq[HDLType],
    z: Seq[HDLType], dir: Int) {
    …
    if (n > 1) {
        val t = (for (i <- 0 until n) yield bool(0)).map(toHDLType)
        for (i <- 0 until k) {
            compare(a(i), a(i + k), t(i), t(i + k), dir)
        }
        bitonicMerge(t.take(k), z.take(k), dir)
        bitonicMerge(t.drop(k), z.drop(k), dir)
    } else {
        z.head := a.head
    }
}
```

# Comparing with MyHDL

```
def bitonicMerge(a, z, dir):
    …
    if n > 1:
        t = [Signal(intbv(0)[w:]) for i in range(n)]
        comp = [compare(a[i], a[i+k], t[i], t[i+k], dir) for i in range(k)]
        loMerge = bitonicMerge(t[:k], z[:k], dir)
        hiMerge = bitonicMerge(t[k:], z[k:], dir)
        return comp, loMerge, hiMerge
    else:
        feed = feedthru(a[0], z[0])
        return feed
```

---

```
def bitonicMerge(a: Seq[HDLType],
    z: Seq[HDLType], dir: Int) {
    …
    if (n > 1) {
        val t = (for (i <- 0 until n) yield bool(0)).map(toHDLType)
        for (i <- 0 until k) {
            compare(a(i), a(i + k), t(i), t(i + k), dir)
        }
        bitonicMerge(t.take(k), z.take(k), dir)
        bitonicMerge(t.drop(k), z.drop(k), dir)
    } else {
        z.head := a.head
    }
}
```

more naturally to write applications of function compare in this way

don't need a feedthru function here

## Issues to be discussed: Hierarchy

- For example, we now want to make compare block in bitonic sort to be a independent module.
- It seems MyHDL would flatten the hierarchy instead of keeping it.
- In ScalaHDL, we want have multiple modules.

# Issues to be discussed: Hierarchy

- A simple approach:
    - just convert bitonic-sort module to Verilog
    - leave the usage of compare module as it is
    - a user can convert the compare module only with one more function call
    - pros: simple, flexible; cons: no checking (but Verilog tools can do that)
- A more complicated approach:
    - convert the expected module and all modules it requires
    - pros: can check the correctness of the usage before conversion; cons: complicated.

# Issues to be discussed: Type Inference

```python
def compare(a1, a2, z1, z2, dir):


    @always_comb

    def logic():

        z1.next = a1

        z2.next = a2

        if dir == (a1 > a2):

            z1.next = a2

            z2.next = a1


    return logic
```

```scala
def compare(a: HDLType, b: HDLType,
  x: HDLType, y: HDLType, dir: Int) {
  if (dir == ASC) {
    when (a > b) {
      x := b
      y := a
    } .otherwise {
      x := a
      y := b
    }
  } else {
    when (a > b) {
      x := a
      y := b
    } .otherwise {
      x := b
      y := a
    }
  }
}
```

# Issues to be discussed: Type Inference

tmp_14 and tmp_15 are not supposed to be wire here!

```
Part of expected Verilog:

reg tmp_14;
reg tmp_15;

always @(tmp12, tmp13) begin
  if (tmp_12 > tmp_13) begin
    tmp_14 <= tmp_13;
    tmp_15 <= tmp_12;
  end
  else begin
    tmp_14 <= tmp_13;
    tmp_15 <= tmp_12;
  end
end
```

```
def compare(a: HDLType, b: HDLType,
  x: HDLType, y: HDLType, dir: Int) {
  if (dir == ASC) {
    when (a > b) {
      x := b
      y := a
    } .otherwise {
      x := a
      y := b
    }
  } else {
    when (a > b) {
      x := a
      y := b
    } .otherwise {
      x := b
      y := a
    }
  }
}
```

# Issues to be discussed: Type Inference

- In ScalaHDL, we can’t know what code is inside which function definition.
- We will infer if a signal is a register according to whether it is inside a sync block, or a when statement block.
- The when statement will be an “always” combinational logic block.
- For nested when statement, the most upper level one will be the combinational logic block.

# Future Plan

- Complete the type inference part.
- Make ScalaHDL able to generate valid Verilog code for both new FlipFlop and BitonicSort.
- Make Simulator runnable for both new FlipFlop and BitonicSort.
- The if statement.

## Any Questions?

Thanks!