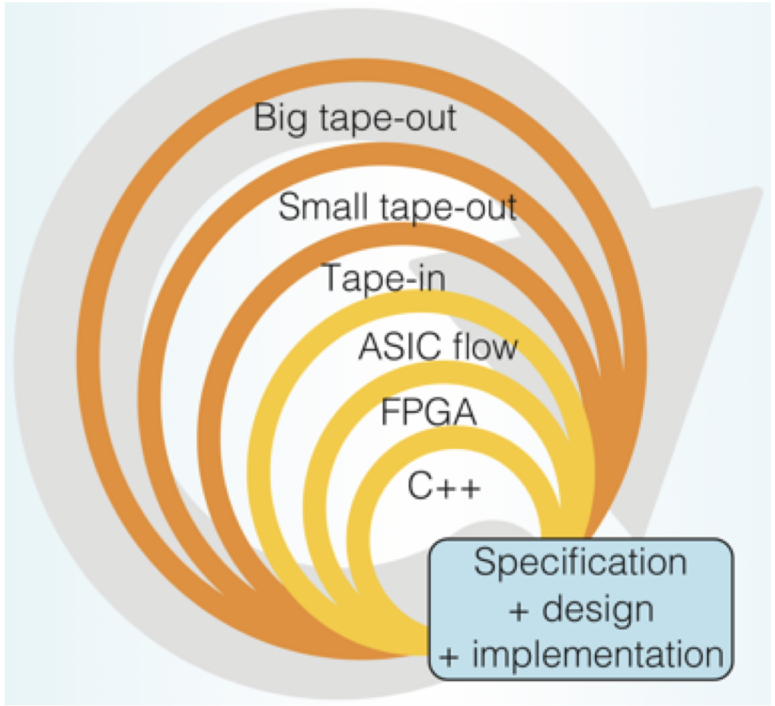
Higher-Order Hardware Design with Chisel 3

Jack Koenig
SiFive

Agile Hardware Design

- Hardware design is traditionally waterfall
- Developing new hardware is hard
  - Architectural Design Space Exploration
  - RTL Development
  - Verification - “did we build the thing right?”
  - Validation - “did we build the right thing?”
- Attacking NRE Costs
- Requires faster design cycle
- Designers need to be productive
  - Do more with less code
  - Leverage libraries
  - **Write reusable code**



* from Lee et. al. “An Agile Approach to Building RISC-V Microprocessors,” in IEEE Micro 2016.

Chisel

- **Constructing** Hardware In a Scala Embedded Language
- Domain Specific Language where the domain is digital design
- NOT high-level synthesis (HLS) nor behavioral synthesis
- Write Scala program to construct and connect hardware objects
  - Parameterized types
  - Object-Oriented Programming
  - Functional Programming
  - Static Typing
- Embedded in Scala, meta-programming language and the actual hardware construction language are the same
- **Intended as a platform upon which to build higher-level abstractions**

Basic Chisel

```
import chisel3._

class GCD extends Module {
  val io = IO(new Bundle {
    val a  = Input(UInt(32.W))
    val b  = Input(UInt(32.W))
    val e  = Input(Bool())
    val z  = Output(UInt(32.W))
    val v  = Output(Bool())
  })
  val x = Reg(UInt(32.W))
  val y = Reg(UInt(32.W))
  when (x > y)   { x := x -% y }
  .otherwise     { y := y -% x }
  when (io.e) { x := io.a; y := io.b }
  io.z := x
  io.v := y === 0.U
}
```

That’s Basically It

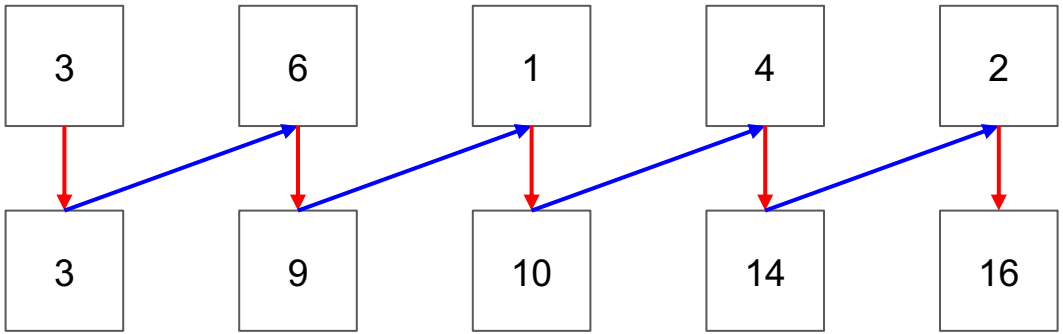
- Now you all know Chisel
- Any questions?

- Just kidding
- **Intended as a platform upon which to build higher-level abstractions**
- Rocket-chip is chock full of abstractions
  - Diplomacy
  - RegisterRouter
  - BusBlocker
  - FIFOFixer
  - CacheCork
  - OrderOgler
- Let’s pick a relatively simple abstraction and see where it takes us

Prefix Sum

The sum of the running totals (prefixes) of a sequence of numbers

$$y_0 = x_0$$
$$y_1 = x_0 + x_1$$
$$y_2 = x_0 + x_1 + x_2$$
$$\ldots$$

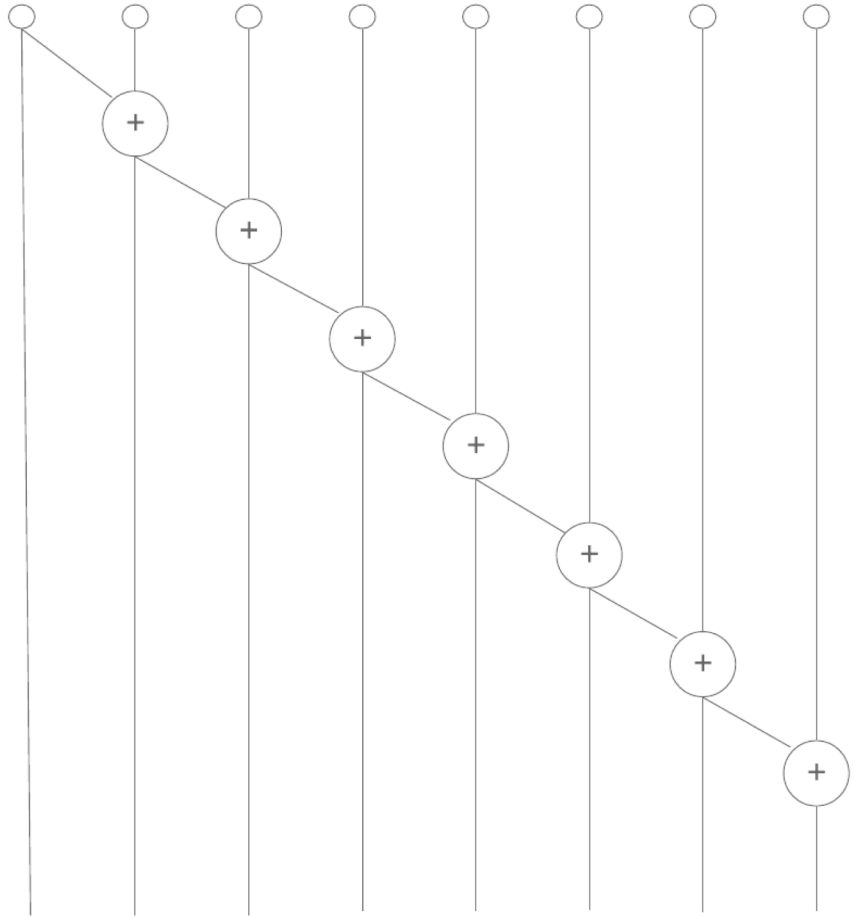


Ripple Prefix Sum

- Naive implementation of the basic algorithm
- Looks kind of like hardware
- Quacks kind of like hardware

Surprise! It’s a ripple-carry adder

- To be more precise, it is a generalization of a ripple-carry adder
- Prefix sum can apply to any associative operator



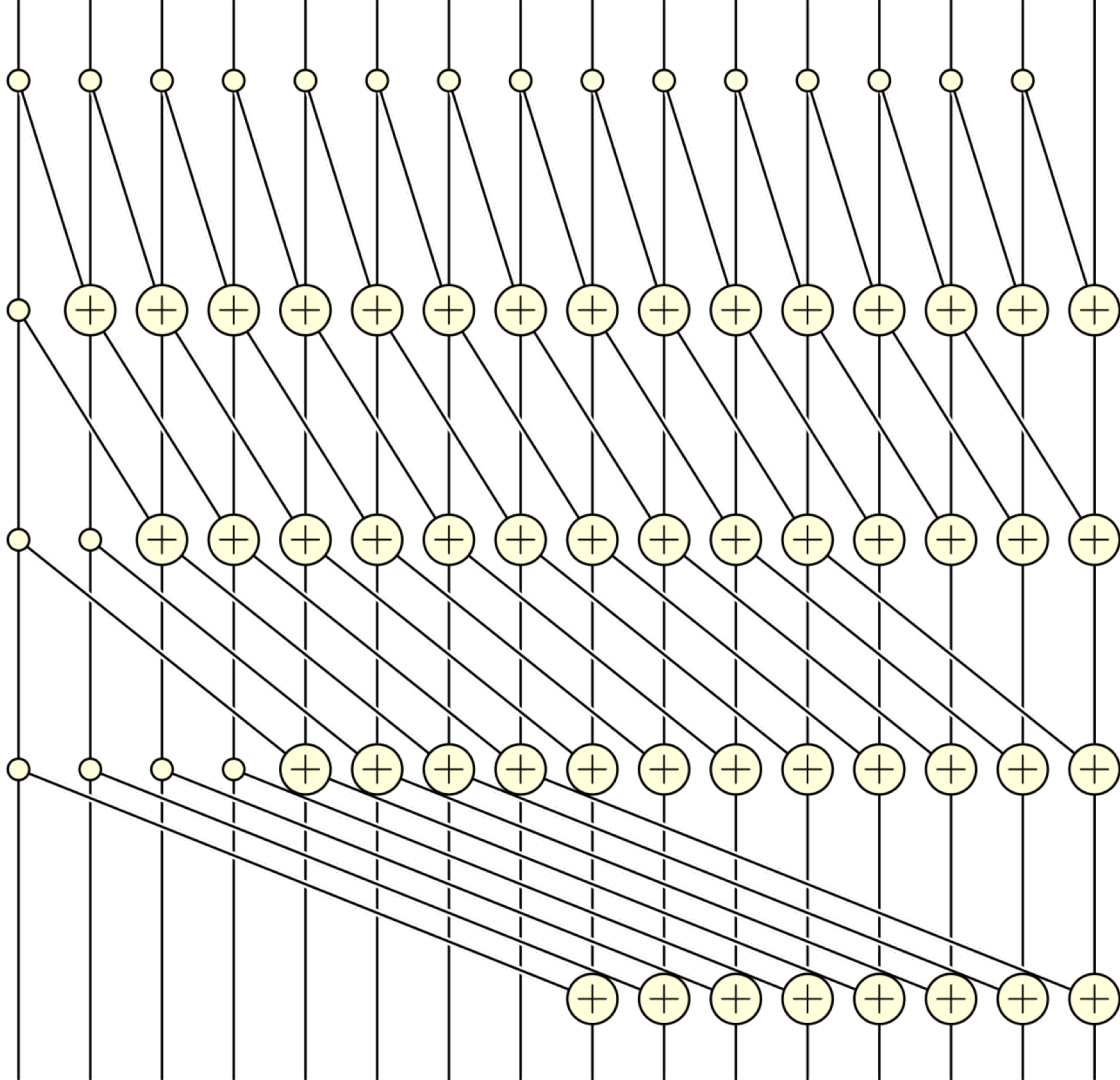
Let’s Define an Interface!

```
trait PrefixSum {
  // out[0] = summands[0]
  // out[1] = summands[0] + summands[1]
  // out[2] = summands[0] + summands[1] + summands[2]
  // ...
  // where + is your associative operator (reflexivity not required)
  def apply[T](summands: Seq[T])(associativeOp: (T, T) => T): Vector[T]
}
```

```scala
// N-1 area, N-1 depth
object RipplePrefixSum extends PrefixSum {
  def apply[T](summands: Seq[T])(associativeOp: (T, T) => T): Vector[T] = {
    def helper(offset: Int, x: Vector[T]): Vector[T] = {
      if (offset >= x.size) {
        x
      } else {
        val layer = Vector.tabulate(x.size) { i =>
          if (i != offset) {
            x(i)
          } else {
            associativeOp(x(i-1), x(i))
          }
        }
        helper(offset+1, layer)
      }
    }
    helper(1, summands.toVector)
  }
}
```

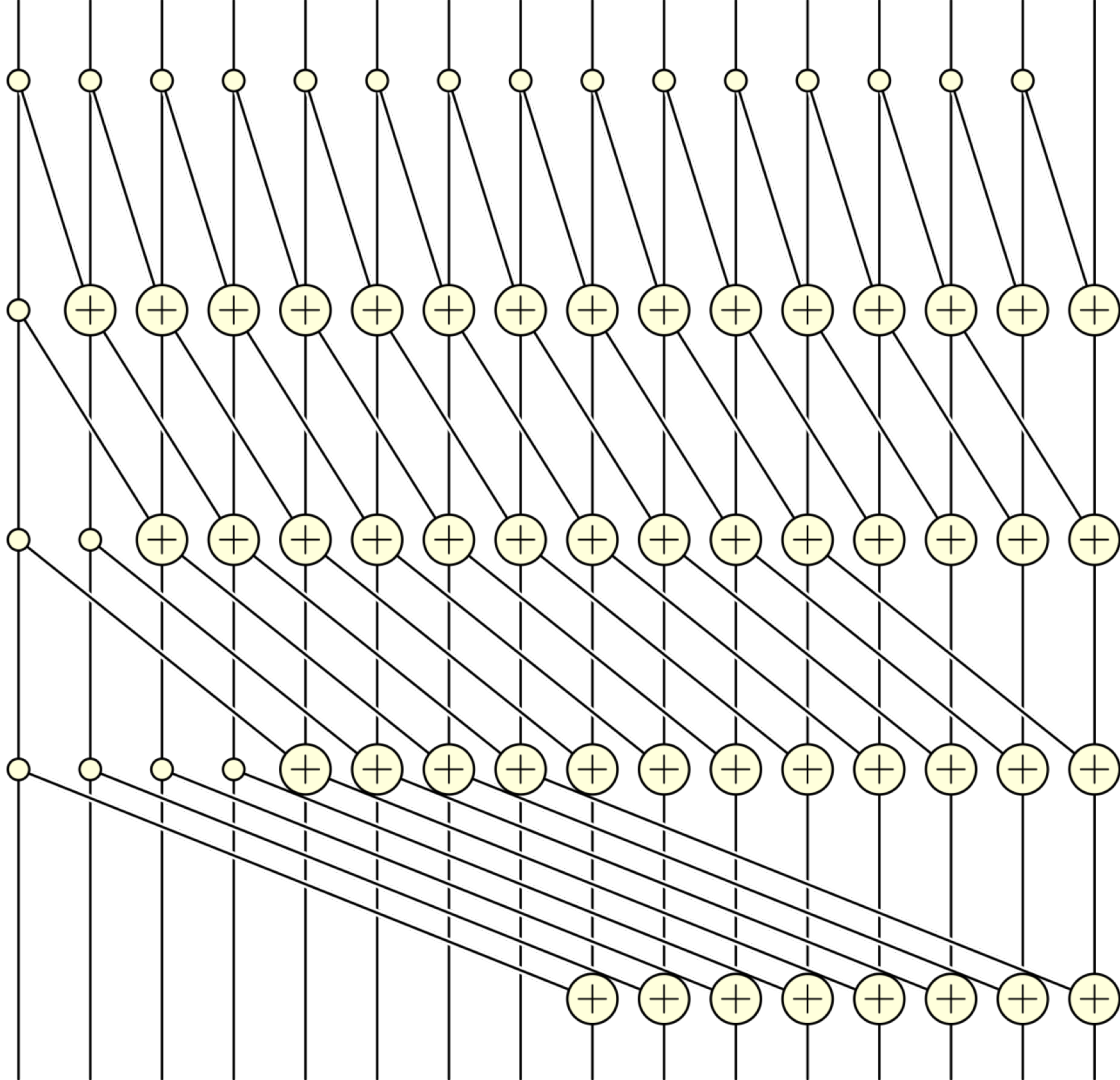
Can we do better?

- Obviously we can, since I asked

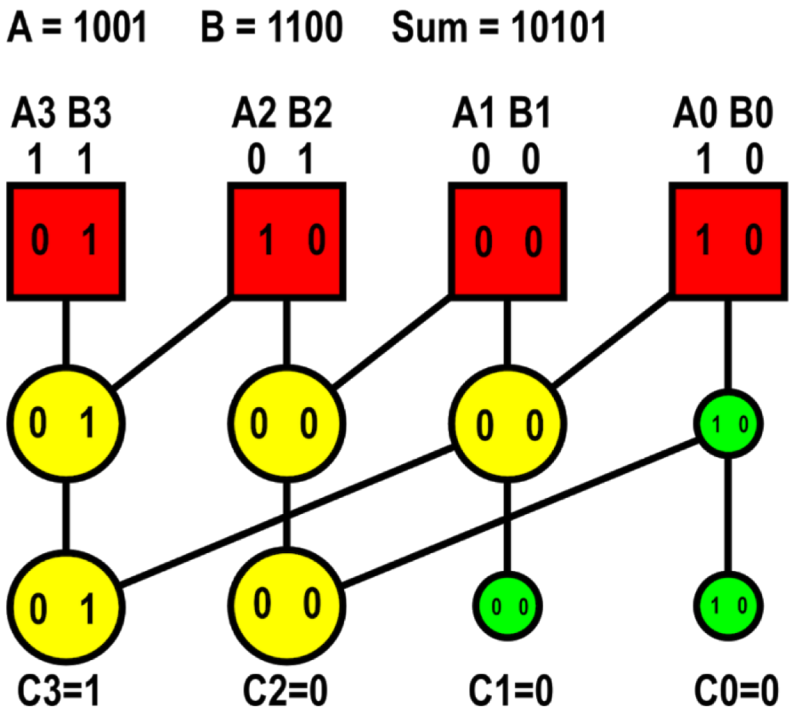


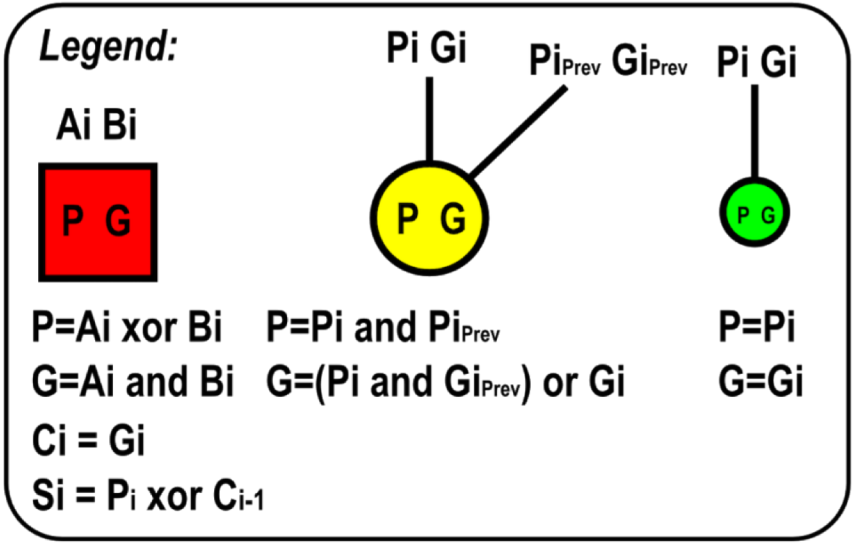
Dense Prefix Sum

- Obviously we can, since I asked
- Far fewer levels of logic at the cost of area
- O(n*log(n)) area
- log(n) depth
- Does this look familiar?



4-bit Kogge-Stone Adder





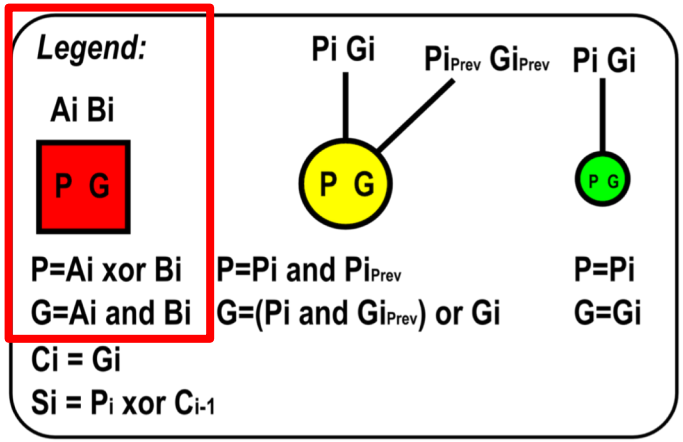
https://en.wikipedia.org/wiki/Kogge%E2%80%93Stone_adder#/media/File:4_bit_Kogge_Stone_Adder_Example_new.pngr

```scala
// O(NlogN) area, logN depth
object DensePrefixSum extends PrefixSum {
  def apply[T](summands: Seq[T])(associativeOp: (T, T) => T): Vector[T] = {
    def helper(offset: Int, x: Vector[T]): Vector[T] = {
      if (offset >= x.size) {
        x
      } else {
        val layer = Vector.tabulate(x.size) { i =>
          if (i < offset) {
            x(i)
          } else {
            associativeOp(x(i-offset), x(i))
          }
        }
        helper(offset << 1, layer)
      }
    }
    helper(1, summands.toVector)
  }
}
```

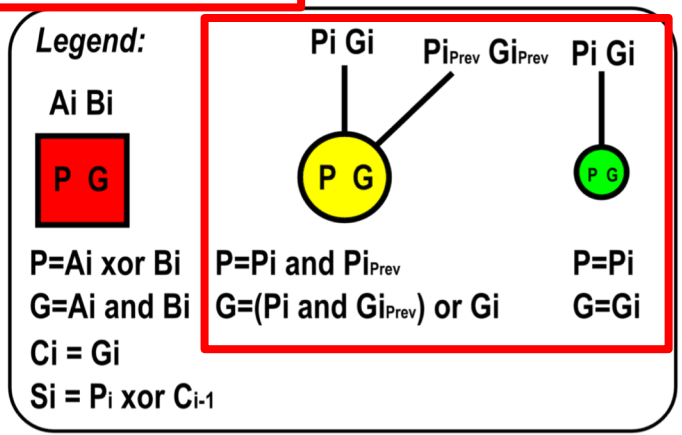
```scala
// N-1 area, N-1 depth
object RipplePrefixSum extends PrefixSum {
  def apply[T](summands: Seq[T])(associativeOp: (T, T) => T): Vector[T] = {
    def helper(offset: Int, x: Vector[T]): Vector[T] = {
      if (offset >= x.size) {
        x
      } else {
        val layer = Vector.tabulate(x.size) { i =>
          if (i != offset) {
            x(i)
          } else {
            associativeOp(x(i-1), x(i))
          }
        }
        helper(offset+1, layer)
      }
    }
    helper(1, summands.toVector)
  }
}
```

```scala
// O(NlogN) area, logN depth
object DensePrefixSum extends PrefixSum {
  def apply[T](summands: Seq[T])(associativeOp: (T, T) => T): Vector[T] = {
    def helper(offset: Int, x: Vector[T]): Vector[T] = {
      if (offset >= x.size) {
        x
      } else {
        val layer = Vector.tabulate(x.size) { i =>
          if (i < offset) {                    // if (i != offset)
            x(i)
          } else {
            associativeOp(x(i-offset), x(i)) // associativeOp(x(i-1), x(i)
          }
        }
        helper(offset << 1, layer)             // helper(offset + 1, layer)
      }
    }
    helper(1, summands.toVector)
  }
}
```

```
import chisel3._

class KoggeStoneAdder(val width: Int) extends Module {
  require(width > 0)
  val io = IO(new Bundle {
    val a = Input(UInt(width.W))
    val b = Input(UInt(width.W))
    val z = Output(UInt((width+1).W))
  })

  // Split up bit vectors into individual bits
  val as = io.a.asBools
  val bs = io.b.asBools

  // P = Ai xor Bi
  // G = Ai and Bi
  // Pairs = (P, G)
  val pairs = as.zip(bs).map { case (a, b) => (a ^ b, a && b) }

  ...
```

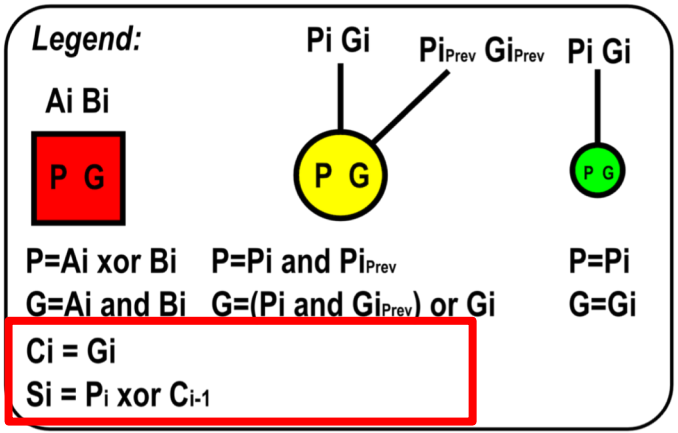


```
// Recall: apply[T](summands: Seq[T])(associativeOp: (T, T) => T): Vector[T]

// At each stage of prefix sum tree
// P = Pi and Pi_prev
// G = (Pi and Gi_prev) or Gi
val pgs = DensePrefixSum(pairs) {
  case ((pp, gp), (pi, gi)) => (pi && pp, (pi && gp) || gi)
}
```



```
// Carries are Generates from end of prefix sum
val cs = false.B +: pgs.map(_._2) // Include carry-in of 0
// Sum requires propogate bits from first step
val ps = pairs.map(_._1) :+ false.B // Include P for overflow

// Si = Pi xor Ci-1
val sum = ps.zip(cs).map { case (p, c) => p ^ c }
```

```
  // Recombine bits into bitvector
  io.z := VecInit(sum).asUInt
}
```



Note the Legend is a little misleading:

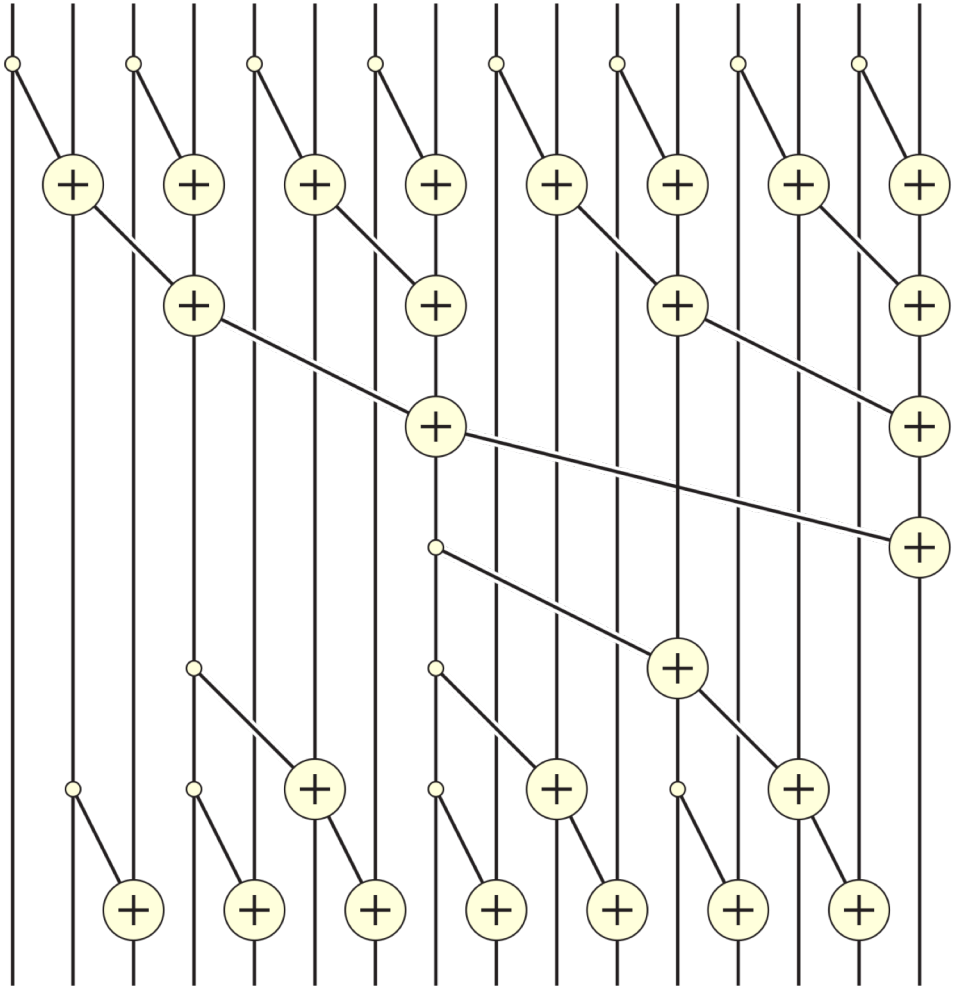
- Gi refers to the Gi output of the prefix sum
- Pi refers to the P from the red box

Abstraction!

- That Kogge-Stone adder was pretty cool
- Can we reuse more code?
- Obviously we can, since I asked

Sparse Prefix Sum

- Deeper logic than Dense, but much less logic
- 2n area
- 2log(n) depth
- We can implement this in Chisel as well
- (it’s a little more complicated than the previous examples, but not that bad)



```
class AbstractAdder(val width: Int, prefixSum: PrefixSum) extends Module {
  ...

  // At each stage of prefix sum tree
  // P = Pi and Pi_prev
  // G = (Pi and Gi_prev) or Gi
  val pgs = prefixSum(pairs) {
    case ((pp, gp), (pi, gi)) => (pi && pp, (pi && gp) || gi)
  }

  ...
```

```
class RippleCarryAdder(width: Int) extends AbstractAdder(width, RipplePrefixSum)

class KoggeStoneAdder(width: Int) extends AbstractAdder(width, DensePrefixSum)

class BrentKungAdder(width: Int) extends AbstractAdder(width, SparsePrefixSum)
```

It’s not about the adders

- Prefix Sum generalizes beyond adders
- From *Prefix Sums and Their Applications* by Guy Blelloch (‘93)
  - Lexically compare strings of characters
  - Add multi precision numbers
  - Evaluate polynomials
  - Solve recurrences
  - Implement radix sort
  - Implement quicksort
  - Solve tridiagonal linear systems
  - Delete marked elements from an array
  - Dynamically allocate processors
  - Perform lexical analysis
  - Search for regular expressions
  - Implement some tree operations
  - Label components in two dimensional images

Questions?

- Find us on Github!
  - https://github.com/freechipsproject/chisel3
  - https://github.com/freechipsproject/firrtl
  - https://github.com/chipsalliance/rocket-chip
- Chat with us on Gitter!
  - https://gitter.im/freechipsproject/chisel3
- Learn Chisel
  - https://github.com/freechipsproject/chisel-bootcamp
    - Now with no installation required!
  - https://stackoverflow.com/questions/tagged/chisel
  - https://groups.google.com/forum/#!forum/chisel-users

Extra Slides

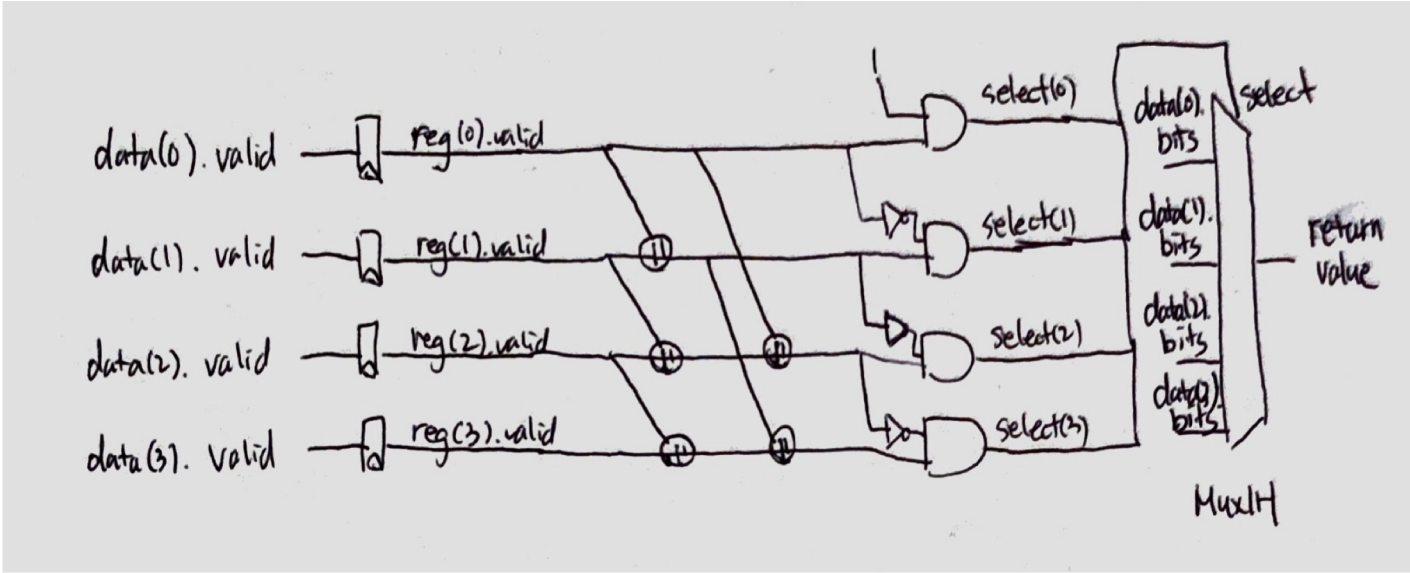
Add a layer operator

```
trait PrefixSum {
  // out[0] = summands[0]
  // out[1] = summands[0] + summands[1]
  // out[2] = summands[0] + summands[1] + summands[2]
  // ...
  // where + is your associative operator (reflexivity not required)
  // layerOp is called on each level of the circuit
  def apply[T](summands: Seq[T])(associativeOp: (T, T) => T,
      layerOp: (Int, Vector[T]) => Vector[T] = idLayer[T] _): Vector[T]
  def layers(size: Int): Int
  def idLayer[T](x: Int, y: Vector[T]) = y
}
```

```
// O(NlogN) area, logN depth
object DensePrefixSum extends PrefixSum {
  def layers(size: Int) = if (size == 0) 1 else 1+log2Ceil(size)
  def apply[T](summands: Seq[T])(associativeOp: (T, T) => T,
      layerOp: (Int, Vector[T]) => Vector[T]): Vector[T] = {
    def helper(layer: Int, offset: Int, x: Vector[T]): Vector[T] = {
      if (offset >= x.size) {
        x
      } else {
        helper(layer+1, offset << 1, layerOp(layer, Vector.tabulate(x.size) { i =>
          if (i < offset) {
            x(i)
          } else {
            associativeOp(x(i-offset), x(i))
          }
        }))
      }
    }
    helper(1, 1, layerOp(0, summands.toVector))
  }
}
```

```scala
// N-1 area, N-1 depth
object RipplePrefixSum extends PrefixSum {
  def apply[T](summands: Seq[T])(associativeOp: (T, T) => T): Vector[T] = {
    def helper(offset: Int, x: Vector[T]): Vector[T] = {
      if (offset >= x.size) {
        x
      } else {
        val layer = mutable.ListBuffer[T]()
                  for (i <- 0 until x.size) {
            if (i != offset) {
              layer += x(i)
            } else {
              layer += associativeOp(x(i-1), x(i))
            }
          }
          layer.toVector
        }
        helper(offset+1, layer)
      }
    }
    helper(1, summands.toVector)
  }
}
```

```scala
object Example {
  // Select the first data valid data element
  def apply[T <: Data](data: Seq[ValidIO[T]], prefixSum: PrefixSum = DensePrefixSum): T = {
    val reg = RegNext(data)
    val fill = Cat(prefixSum(reg.map(_.valid))(_ || _).reverse)
    val select = fill & ~Cat(0.U, fill)
    Mux1H(select, reg.map(_.bits))
  }
}
```



Borrow good ideas from software, build a compiler!