Prefix Adder

Yunqian Luo

2021.9.27

What is a (Full) Adder

Input:

$$
\begin{aligned}
x &= x_{n-1} x_{n-2} \cdots x_0 \\
y &= y_{n-1} y_{n-2} \cdots y_0 \\
c_{\text{in}} &\in \{0, 1\}
\end{aligned}
$$

Output:

$$
\begin{aligned}
s &= (x + y + c_{\text{in}}) \mod 2^n \\
c_{\text{out}} &= \left\lfloor \frac{x + y + c_{\text{in}}}{2^n} \right\rfloor \in \{0, 1\}
\end{aligned}
$$

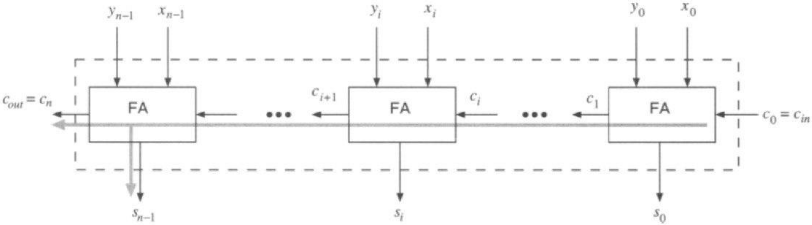
Carry Bit Propagation

We can calculate c_{out} sequentially:

$$
\begin{aligned}
c_0 &= c_{\text{in}} \\
c_1 &= \left\lfloor \frac{x_0 + y_0 + c_0}{2} \right\rfloor \\
c_2 &= \left\lfloor \frac{x_1 + y_1 + c_0}{2} \right\rfloor \\
\ldots \\
c_{\text{out}} = c_n &= \left\lfloor \frac{x_{n-1} + y_{n-1} + c_{n-1}}{2} \right\rfloor
\end{aligned}
$$

Naïve Implementation: Carry-Ripple Adder

A thread of full adders concatenated together.



Wires: $O(n)$. Depth: $O(n)$.

[0] Figure from Ercegovac, Miloš D., and Tomás Lang. "CHAPTER 2 - Two-Operand Addition." In Digital Arithmetic, edited by Miloš D. Ercegovac and Tomás Lang

Observation on Carry Bit Propagation

Can we go faster?

Given x and y, the function $f_{i,j} : c_i \to c_j \ (i < j)$ is determined. Notice that the function is necessarily non-decreasing, there are three such functions:

- **Kill** $c_j = 0$ no matter what c_i is.
- **Propagate** $c_j = c_i$.
- **Generate** $c_j = 1$ no matter what c_i is.

Observation on Carry Bit Propagation

Can we go faster?

Given x and y, the function $f_{i,j} : c_i \to c_j \ (i < j)$ is determined. Notice that the function is necessarily non-decreasing, there are three such functions:

- **Kill** $c_j = 0$ no matter what c_i is.
- **Propagate** $c_j = c_i$.
- **Generate** $c_j = 1$ no matter what c_i is.

Encode such a function by two bits $(g_{i,j}, p_{i,j})$:

- **Kill** $g_{i,j} = p_{i,j} = 0$.
- **Propagate** $g_{i,j} = 0, p_{i,j} = 1$.
- **Generate** $g_{i,j} = 1, p_{i,j} = 0$.

How to Calculate a and p

Notice

$$f_{i,j} = f_{j-1,j} \circ f_{j-2,j-1} \circ \cdots \circ f_{i,i+1} \tag{1}$$

Function composition corresponds to encoded calculation:

$$
\begin{aligned}
f_{i,j} &= f_{k,j} \circ f_{i,k} && (2)\\
(g_{i,j}, p_{i,j}) &= (g_{k,j}, p_{k,j}) \circ (g_{i,k}, p_{i,k}) && (3)\\
&= (g_{k,j} + g_{i,k} p_{k,j}, p_{i,j} p_{j,k}) && (4)
\end{aligned}
$$

Function composition is associative
$\Longrightarrow$ Encoded calculation is also associative.

How to Calculate a and p

Notation: $f_i = f_{0,i},\ (g_i, p_i) = (g_{i,i+1}, p_{i,i+1})$.

To obtain s_i, we only needs c_i

$$s_i = (x_i + y_i + c_i) \mod 2$$

To obtain c_i, we only needs $g_{0,i}, p_{0,i}$

$$c_i = g_{0,i} + c_0 p_{0,i}$$

To obtain $g_{0,i}, p_{0,i}$,

$$(g_{0,i}, p_{0,i}) = (g_{i-1}, p_{i-1}) \circ (g_{i-1}, p_{i-1}) \circ \cdots \circ (g_0, p_0)$$

The problem is reduced to calculating all **prefix sum** of an associative operator.

Naïve Method Once Again

Denote the summands by $x_0, x_2, \ldots, x_{n-1}$,
$s_{i,j} = s_i \circ s_{i+1} \circ \cdots \circ s_j$, $s_i = s_{0,i}$.
Calculate one by one:

$$s_0 = x_0$$
$$s_i = x_i \circ s_{i-1}$$

Latency: $O(n)$.

Slow as before.

Constant Size Parallelization

Calculate two in one step:

$$\begin{aligned} s_0 &= x_0 \\ (s_i, s_{i+1}) &= (s_{i-1} \circ x_i, s_{i-1} \circ x_i \circ x_{i+1}) \end{aligned}$$

Latency: $O(n)$.

Maybe a little bit faster.

Brent Kung Adder

Stage 1 Calculate a tree of partial sum $S_i = s_{z(i),i}$ for all i, where $z(i)$ is i in binary representation with all trailing ones set to zero.

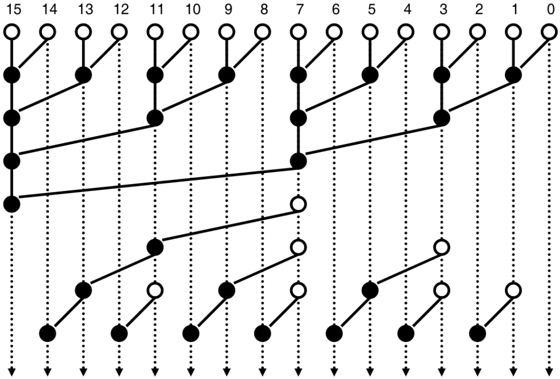
E.g. $z(1101011_{(2)}) = 1101000_{(2)}$

Stage 2 Propagate the partial sum to obtain the prefix sum

$$\begin{aligned} s_{1101011_{(2)}} =& S_{1101011_{(2)}} \\ &+ S_{1101000_{(2)}-1} \\ &+ S_{1100000_{(2)}-1} \\ &+ S_{1000000_{(2)}-1} \end{aligned}$$

Brent Kung Adder

Wires: $O(n)$. Depth: $2 \log n$. Max fan-out: 2.



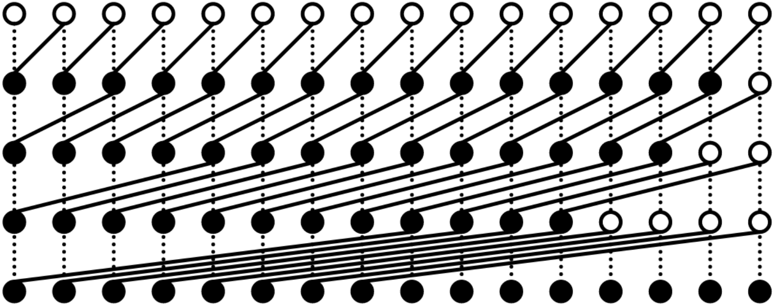
[0] Figure from https://web.stanford.edu/class/archive/ee/ee371/ee371.1066/lectures/lect_04.2up.pdf

Kogges Stone Adder

In the k-th layer, we obtain the partial sum $s_{i,i-2^k+1}$ for each i.

$$s_{i-2^k+1,i} = s_{i-2^{k-1}+1,i} + s_{i-2^k-1,i-2^{k-1}}$$

Wires: $O(n \log n)$. Latency: $\log n$. Max fan-out: 2.



[0] Figure from https://web.stanford.edu/class/archive/ee/ee371/ee371.1066/lectures/lect_04.2up.pdf

Implementation

https://github.com/sequencer/arithemic

```scala
trait FullAdder extends Module {
    val width: Int
    val a: UInt = IO(Input(UInt(width.W)))
    val b: UInt = IO(Input(UInt(width.W)))
    val z: UInt = IO(Output(UInt((width + 1).W)))
}

class PrefixAdder(val width: Int, prefixSum: PrefixSum) extends FullAdder {
    val as: Seq[Bool] = a.asBools
    val bs: Seq[Bool] = b.asBools

    val pairs: Seq[(Bool, Bool)] = prefixSum.zeroLayer(as, bs)

    val pgs: Vector[(Bool, Bool)] = prefixSum(pairs)

    val cs: Vector[Bool] = false.B +: pgs.map(_._2) // Include carry-in of 0
    val ps: Seq[Bool] = pairs.map(_._1) :+ false.B // Include P for overflow
    val sum: Seq[Bool] = ps.zip(cs).map { case (p, c) => p ^ c }

    z := VecInit(sum).asUInt
}
```

Implementation

```
trait PrefixSum {
  def apply(aummands: Seq[(Bool, Bool)]): Vector[(Bool, Bool)]
  def associativeOp(leaf: Seq[(Bool, Bool)]): (Bool, Bool)
  def zeroLayer(a: Seq[Bool], b: Seq[Bool]): Seq[(Bool, bool)]
}

trait CommonPrefixSum extends PrefixSum {
  def associativeOp(leaf: Seq[(Bool, Bool)]): (Bool, Bool) = leaf match {
    case Seq((p0, g0), (p1, g1)) => (p0 && p1, (g0 && p1) || g1)
    case Seq((p0, g0), (p1, g1), (p2, g2)) => ...
    ...
  }
}
```

Implementation

```scala
// simply translate the graph to the code, apply transformation layer by layer
object RippleCarrySum extends CommonPrefixSum {
  def apply(summands: Seq[(Bool, Bool)]): Vector[(Bool, Bool)] = {
    def helper(offset: Int, x: Vector[(Bool, Bool)]): Vector[(Bool, Bool)] = {
      if (offset ≥ x.size) { x } else {
        val layer: Vector[(Bool, Bool)] = Vector.tabulate(x.size) { i ⇒
          if (i ≠ offset) x(i) else associativeOp(Seq(x(i - 1), x(i)))
        }
        helper(offset + 1, layer)
      }
    }
    helper(1, summands.toVector)
  }
}
```

$$
\begin{aligned}
&\mathrm{helper}(1, [s_0, s_1, s_2, \ldots, s_{n-1}]) \\
=\ &\mathrm{helper}(2, [s_0, s_0 \circ s_1, s_2, \ldots, s_{n-1}]) \\
\ldots& \\
=\ &\mathrm{helper}(n, [s_0, s_0 \circ s_1, s_0 \circ s_1 \circ s_2, \ldots, s_0 \circ s_1 \circ \cdots s_{n-1}]) \\
=\ &[s_0, s_0 \circ s_1, s_0 \circ s_1 \circ s_2, \ldots, s_0 \circ s_1 \circ \cdots s_{n-1}]
\end{aligned}
$$

Implementation

Following the same strategy we implement other prefix adders:

```
object RippleCarry3Sum extends CommonPrefixSum { ... }
object BrentKungSum extends CommonPrefixSum { ... }
object KoggeStoneSum extends CommonPrefixSum { ... }
```

Just translate from corresponding graphs.

Implementation

```
class PrefixGraph { ... }

trait HasPrefiSumWithGraphImp extends PrefixSum {
  val PrefixGraph: PrefixGraph
  override def apply(summands: Seq[(Bool, Bool)]): Vector[(Bool, Bool)] = {
    ...
  }
}

// play it like this
val d = new CommonPrefixSum with HasPrefixSumWithGraphImp {
    val prefixGraph: PrefixGraph = PrefixGraph(os.resource / "graph.json")
}
```

```
\Thanks
\end{presentation}
```