Implementation of a Highly Configurable Wallace Tree Multiplier with Chisel

Lin Jiawei

Institute of Computing Technology, Chinese Academy of Sciences

2021/6/26

source code

Implementation of a Highly Configurable Wallace Tree Multiplier with Chisel

- Use recursion to achieve Wallace tree compression
- Only 120 lines of chisel code (booth-4 encoding + tree compression + final adder)
- Fully configurable operation width
- You can control how many pipeline stages will be generated
- You can control where to insert a pipeline stage during compression
- High performance
  - An example: 3-stage 64x64 bit signed multiplier ⇔ < 350ps latency

Chisel implementation

- Booth encoding: very easy to implement with chisel
- Tree compression:
    - Step1: for each column in the tree, write a recursive function to compress the column
    - Step2: organize the compressed columns as a new tree
    - Step3: check parameters to decide if we need to insert registers after the new tree
    - Step4: repeat Step1 on the new tree, until the depth of the tree are reduced to 2, go to Step5
    - Step5: use a adder to calculate final result

Algorithm

- Booth-4 encoding
  - generate partial products
  - n*n mul ⇔ n/2 partial products
- Tree compression
  - optimize the tree(reduce sign-ext bits for signed multiplier)
  - add all products by column compression

Algorithm-Booth4 encoding

- n*n mul ⇔ n/2 partial products

$$X * Y = (x_{n-3} + x_{n-2} - 2 * x_{n-1}) * 2^{n-2} * Y$$
$$+(x_{n-5} + x_{n-4} - 2 * x_{n-3}) * 2^{n-4} * Y + \cdots$$
$$+(x_{-1} + x_0 - 2 * x_1) * 2^0 * Y, \qquad x_{-1} = 0$$

```
for(i <- Range(0, len, 2)){
  val x = if(i==0) Cat(a(1,0), 0.U(1.W)) else if(i+1==len) SignExt(a(i, i-1), 3) else a(i+1, i-1)
  val pp_temp = MuxLookup(x, 0.U, Seq(
    1.U -> b_sext,
    2.U -> b_sext,
    3.U -> bx2,
    4.U -> neg_bx2,
    5.U -> neg_b,
    6.U -> neg_b
  ))
```

```
val b_sext, bx2, neg_b, neg_bx2 = Wire(UInt((len+1).W))
b_sext := SignExt(b, len+1)
bx2 := b_sext << 1
neg_b := (~b_sext).asUInt()
neg_bx2 := neg_b << 1
```

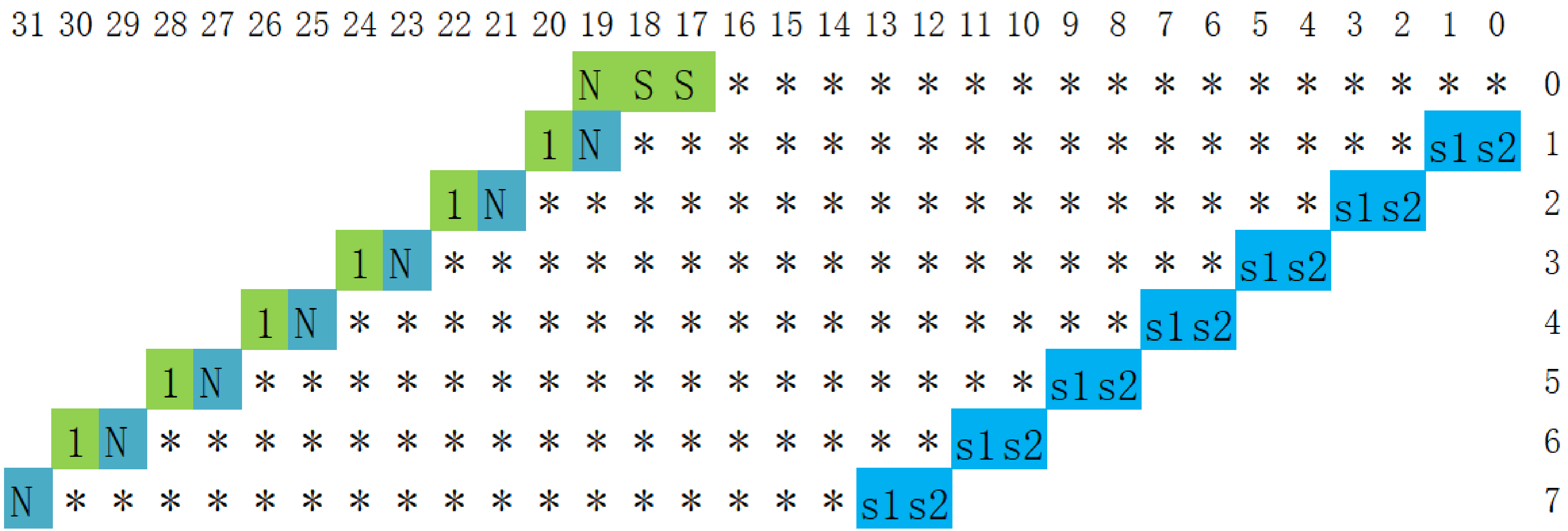
Algorithm-Booth4 encoding

- Our target: calculate the sum of the partial products

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	0
S	S	S	S	S	S	S	S	S	S	S	S	S	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*			1
S	S	S	S	S	S	S	S	S	S	S	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*					2
S	S	S	S	S	S	S	S	S	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*							3
S	S	S	S	S	S	S	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*									4
S	S	S	S	S	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*											5
S	S	S	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*													6
S	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*															7

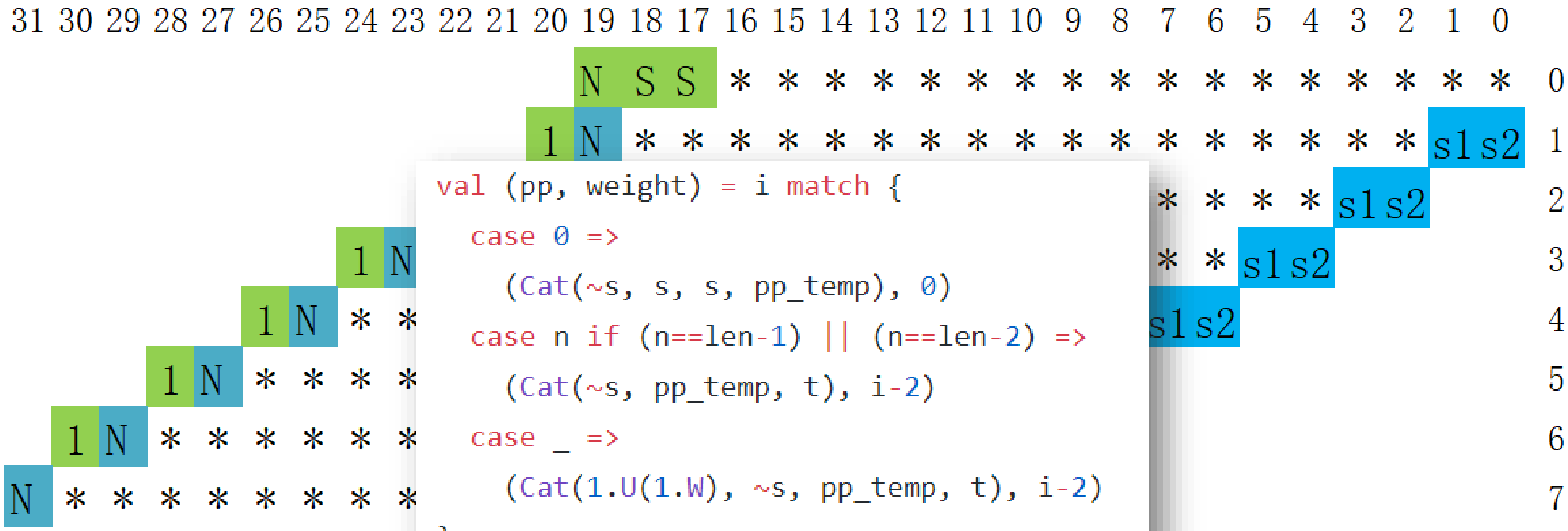
Algorithm-Sign-ext optimization

- The optimization algorithm is beyond our scope
- But it’s very easy to implement with Chisel



Algorithm-Sign-ext optimization

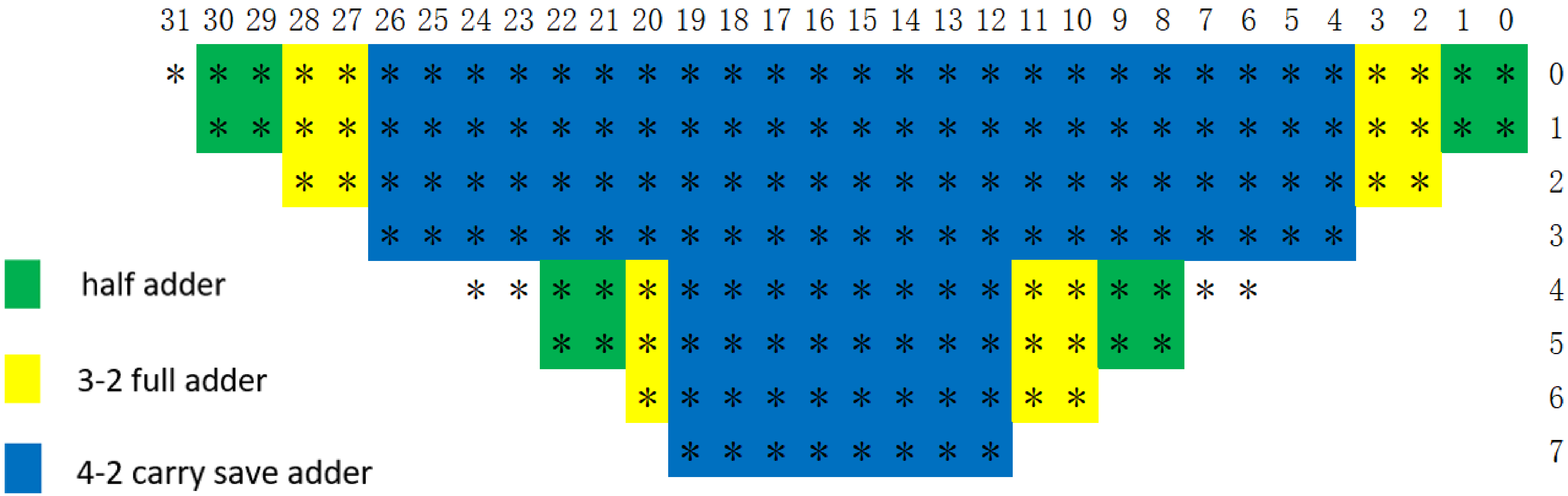
- The optimization algorithm is beyond our scope
- But it’s very easy to implement with Chisel




```scala
val (pp, weight) = i match {
  case 0 =>
    (Cat(~s, s, s, pp_temp), 0)
  case n if (n==len-1) || (n==len-2) =>
    (Cat(~s, pp_temp, t), i-2)
  case _ =>
    (Cat(1.U(1.W), ~s, pp_temp, t), i-2)
}
```

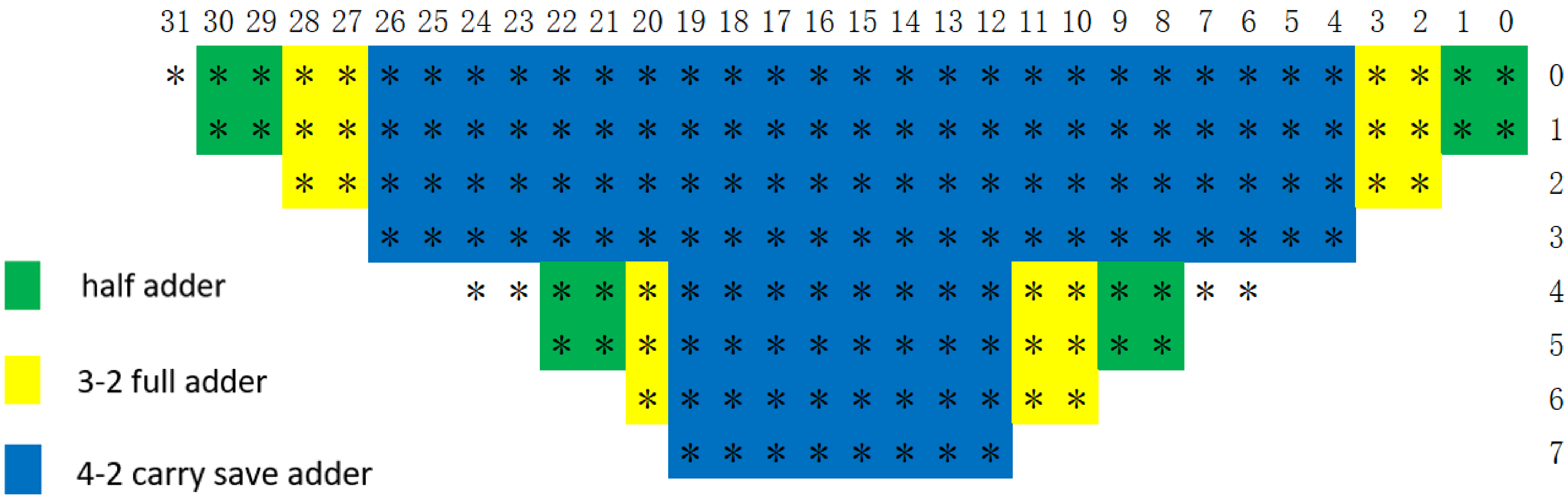
Algorithm-Tree compression

- Tree representation `val columns: Array[Seq[Bool]]`



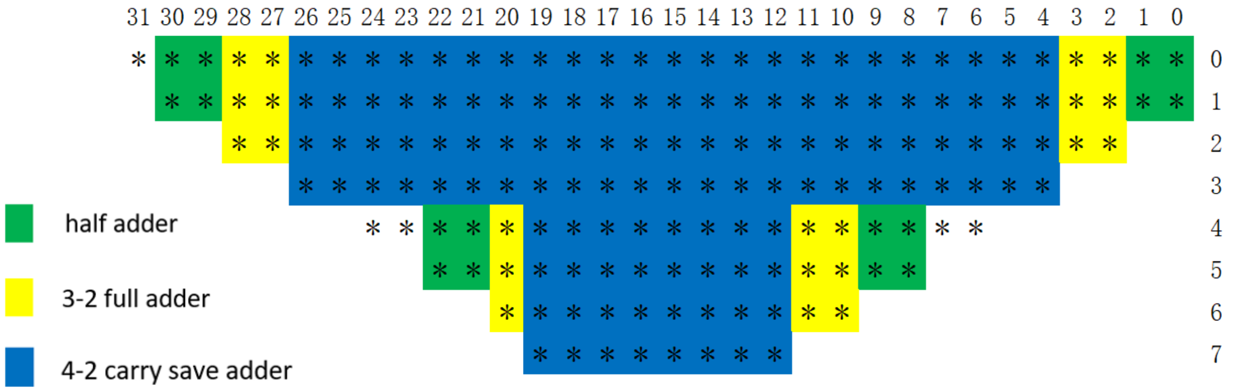
Algorithm-Tree compression

- An 16 * 16 example `val columns: Array[Seq[Bool]]`
- Length of columns == 32
- 0 < Size of each column < 8



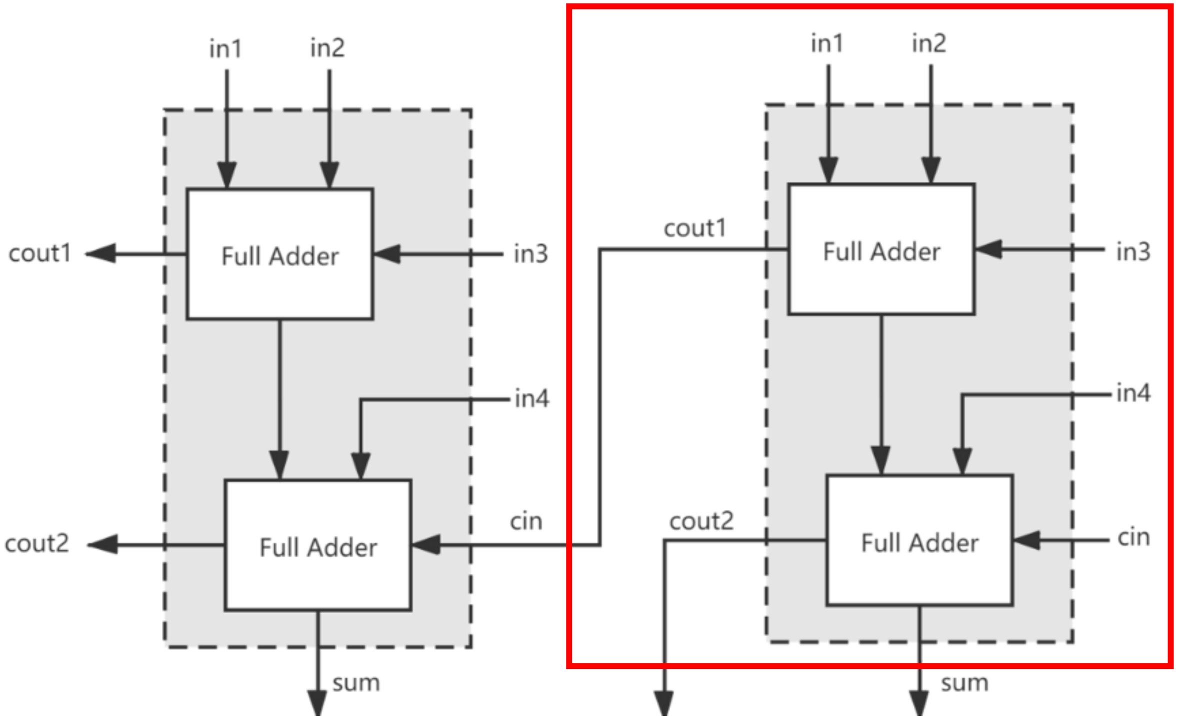
Compress a column

- Match column size
  - 1-> do nothing
  - 2-> half adder
  - 3-> full adder
  - 4-> 4-2 adder
  - n-> x + (n-x), x∈[1, 4]




```
def addOneColumn(col: Seq[Bool], cin: Seq[Bool]): (Seq[Bool], Seq[Bool], Seq[Bool]) = {
  var sum = Seq[Bool]()
  var cout1 = Seq[Bool]()
  var cout2 = Seq[Bool]()
  col.size match {
    case 1 =>  // do nothing
      sum = col ++ cin
    case 2 =>
      val c22 = Module(new C22)
      c22.io.in := col
      sum = c22.io.out(0).asBool() +: cin
      cout2 = Seq(c22.io.out(1).asBool())
    case 3 =>
      val c32 = Module(new C32)
      c32.io.in := col
      sum = c32.io.out(0).asBool() +: cin
      cout2 = Seq(c32.io.out(1).asBool())
    case 4 =>
      val c53 = Module(new C53)
      for((x, y) <- c53.io.in.take(4) zip col){
        x := y
      }
      c53.io.in.last := (if(cin.nonEmpty) cin.head else 0.U)
      sum = Seq(c53.io.out(0).asBool()) ++ (if(cin.nonEmpty) cin.drop(1) else Nil)
      cout1 = Seq(c53.io.out(1).asBool())
      cout2 = Seq(c53.io.out(2).asBool())
    case n =>
      val cin_1 = if(cin.nonEmpty) Seq(cin.head) else Nil
      val cin_2 = if(cin.nonEmpty) cin.drop(1) else Nil
      val (s_1, c_1_1, c_1_2) = addOneColumn(col take 4, cin_1)
      val (s_2, c_2_1, c_2_2) = addOneColumn(col drop 4, cin_2)
      sum = s_1 ++ s_2
      cout1 = c_1_1 ++ c_2_1
      cout2 = c_1_2 ++ c_2_2
  }
  (sum, cout1, cout2)
}
```

Compress a column




```scala
def addOneColumn(col: Seq[Bool], cin: Seq[Bool]): (Seq[Bool], Seq[Bool], Seq[Bool]) = {
  var sum = Seq[Bool]()
  var cout1 = Seq[Bool]()
  var cout2 = Seq[Bool]()
  col.size match {
    case 1 =>  // do nothing
      sum = col ++ cin
    case 2 =>
      val c22 = Module(new C22)
      c22.io.in := col
      sum = c22.io.out(0).asBool() +: cin
      cout2 = Seq(c22.io.out(1).asBool())
    case 3 =>
      val c32 = Module(new C32)
      c32.io.in := col
      sum = c32.io.out(0).asBool() +: cin
      cout2 = Seq(c32.io.out(1).asBool())
    case 4 =>
      val c53 = Module(new C53)
      for((x, y) <- c53.io.in.take(4) zip col){
        x := y
      }
      c53.io.in.last := (if(cin.nonEmpty) cin.head else 0.U)
      sum = Seq(c53.io.out(0).asBool()) ++ (if(cin.nonEmpty) cin.drop(1) else Nil)
      cout1 = Seq(c53.io.out(1).asBool())
      cout2 = Seq(c53.io.out(2).asBool())
    case n =>
      val cin_1 = if(cin.nonEmpty) Seq(cin.head) else Nil
      val cin_2 = if(cin.nonEmpty) cin.drop(1) else Nil
      val (s_1, c_1_1, c_1_2) = addOneColumn(col take 4, cin_1)
      val (s_2, c_2_1, c_2_2) = addOneColumn(col drop 4, cin_2)
      sum = s_1 ++ s_2
      cout1 = c_1_1 ++ c_2_1
      cout2 = c_1_2 ++ c_2_2
  }
  (sum, cout1, cout2)
}
```

Compress a column

- After compression, size of each column becomes smaller
- Organize the compressed columns as a new tree

```
val columns_next = Array.fill(2*len)(Seq[Bool]())
var cout1, cout2 = Seq[Bool]()
for( i <- cols.indices){
  val (s, c1, c2) = addOneColumn(cols(i), cout1)
  columns_next(i) = s ++ cout2
  cout1 = c1
  cout2 = c2
}
```

```
def addOneColumn(col: Seq[Bool], cin: Seq[Bool]): (Seq[Bool], Seq[Bool], Seq[Bool]) = {
  var sum = Seq[Bool]()
  var cout1 = Seq[Bool]()
  var cout2 = Seq[Bool]()
  col.size match {
    case 1 =>  // do nothing
      sum = col ++ cin
    case 2 =>
      val c22 = Module(new C22)
      c22.io.in := col
      sum = c22.io.out(0).asBool() +: cin
      cout2 = Seq(c22.io.out(1).asBool())
    case 3 =>
      val c32 = Module(new C32)
      c32.io.in := col
      sum = c32.io.out(0).asBool() +: cin
      cout2 = Seq(c32.io.out(1).asBool())
    case 4 =>
      val c53 = Module(new C53)
      for((x, y) <- c53.io.in.take(4) zip col){
        x := y
      }
      c53.io.in.last := (if(cin.nonEmpty) cin.head else 0.U)
      sum = Seq(c53.io.out(0).asBool()) ++ (if(cin.nonEmpty) cin.drop(1) else Nil)
      cout1 = Seq(c53.io.out(1).asBool())
      cout2 = Seq(c53.io.out(2).asBool())
    case n =>
      val cin_1 = if(cin.nonEmpty) Seq(cin.head) else Nil
      val cin_2 = if(cin.nonEmpty) cin.drop(1) else Nil
      val (s_1, c_1_1, c_1_2) = addOneColumn(col take 4, cin_1)
      val (s_2, c_2_1, c_2_2) = addOneColumn(col drop 4, cin_2)
      sum = s_1 ++ s_2
      cout1 = c_1_1 ++ c_2_1
      cout2 = c_1_2 ++ c_2_2
  }
  (sum, cout1, cout2)
}
```

Compress the whole tree

- Organize the compressed result as a new tree
- [Optional] insert registers after the new tree
- Repeat until max(cols.size) == 2

```
def addAll(cols: Array[Seq[Bool]], depth: Int): (UInt, UInt) = {
  if(max(cols.map(_.size)) <= 2){
    val sum = Cat(cols.map(_(0)).reverse)
    var k = 0
    while(cols(k).size == 1) k = k+1
    val carry = Cat(cols.drop(k).map(_(1)).reverse)
    (sum, Cat(carry, 0.U(k.W)))
  } else {
    val columns_next = Array.fill(2*len)(Seq[Bool]())
    var cout1, cout2 = Seq[Bool]()
    for( i <- cols.indices){
      val (s, c1, c2) = addOneColumn(cols(i), cout1)
      columns_next(i) = s ++ cout2
      cout1 = c1
      cout2 = c2
    }

    val needReg = stages.contains(depth)
    val toNextLayer = if(needReg)
      columns_next.map(_.map(x => RegNext(x))) // TODO: use 'RegEnable' instead
    else
      columns_next

    addAll(toNextLayer, depth+1)
  }
```

Compress the whole tree

- 'stages' is a parameter of the multiplier
- You can control where to insert a pipeline stage by this parameter

```
def addAll(cols: Array[Seq[Bool]], depth: Int): (UInt, UInt) = {
  if(max(cols.map(_.size)) <= 2){
    val sum = Cat(cols.map(_(0)).reverse)
    var k = 0
    while(cols(k).size == 1) k = k+1
    val carry = Cat(cols.drop(k).map(_(1)).reverse)
    (sum, Cat(carry, 0.U(k.W)))
  } else {
    val columns_next = Array.fill(2*len)(Seq[Bool]())
    var cout1, cout2 = Seq[Bool]()
    for( i <- cols.indices){
      val (s, c1, c2) = addOneColumn(cols(i), cout1)
      columns_next(i) = s ++ cout2
      cout1 = c1
      cout2 = c2
    }

    val needReg = stages.contains(depth)
    val toNextLayer = if(needReg)
      columns_next.map(_.map(x => RegNext(x))) // TODO: use 'RegEnable' instead
    else
      columns_next

    addAll(toNextLayer, depth+1)
  }
```

Final Step-adder

- When the max(cols.size) is reduced to 2, 'addAll' returns 'sum' and 'carry'
- Calculate the sum of 'sum' and 'carry', then we get final result

```
if(max(cols.map(_.size)) <= 2){
  val sum = Cat(cols.map(_(0)).reverse)
  var k = 0
  while(cols(k).size == 1) k = k+1
  val carry = Cat(cols.drop(k).map(_(1)).reverse)
  (sum, Cat(carry, 0.U(k.W)))
} else {

val (sum, carry) = addAll(cols = columns, depth = 0)
io.result := sum + carry
```

Summary

- We don’t write code to connect adders and wires
- We write code to describe the rules on how to connect adders and wires
- The multiplier is highly configurable
  - Input width
  - Pipeline stages
- Only about 120 lines of Chisel code
  - Implemented the same function in thousands of lines of code in Verilog
  - More configurable
  - Better scalability
  - Easier to read

Thank You!