Brent-Kung fast adder description, simulation and formal verification using Lava.

Leandro Marsó
Facultad de Ciencias Exactas, Físicas y Naturales
Universidad Nacional de Córdoba
Email: elleandro@gmail.com

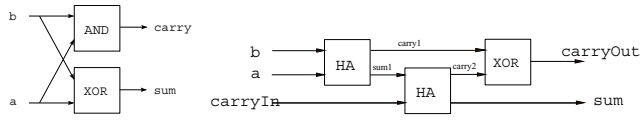
***Abstract*— Integrated Circuits Design can be made by using ideas that come from Computer Science, and particulary from Functional Programing, that can give us more abstract representations and verication techniques in order to keep up with the ever-increasing complexity of modern hardware designs. Using Lava [1], a HDL embedded in Haskell, we explain how to design, simulate, and formal verify a carry chain binary adder and a fast adder parameterized in the size of its inputs.**

I. Introduction

We shall present a Lava description of a carry chain adder. With this simple circuit we can verify all additions properties in less computer time than with a more complex one, like the Brent-Kung adder. We shall then prove that the latter is correctly designed, by means of an equivalence ckecking between them.

A. Simple circuits

Following figure 1a we will describe a half adder:



(a) Half adder (b) Full adder

Fig. 1: Bit adders

```
import Lava

halfAdd (a, b) = (sum, carry)
   where
      sum   = xor2 (a, b)
      carry = and2 (a, b)
```

The `import Lava` statement is needed to import a module which defines a number of operations that we can use to build circuits, like definitions of gates `xor2` and `and2`, among others.

Figure 1b, can be used to write a circuit description out of it, by giving names to every internal signal of the circuit and writing down the subcomponent definitions, as follows:

```
fullAdd (carryIn, (a, b)) = (sum, carryOut)
    where
       (sum1,carry1) = halfAdd (a, b)
       (sum, carry2) = halfAdd (carryIn, sum1)
       carryOut      = xor2 (carry2, carry1)
```

Note that we made use of the previous circuit description `halfAdd`, as well as the `xor2` function to build the new circuit called `fullAdd`.

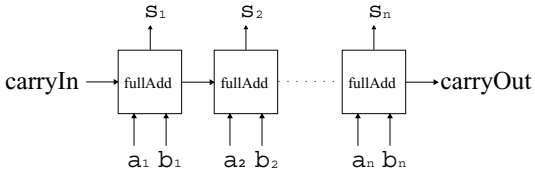


Fig. 2: Binary Adder

Lastly, the binary adder shown in figure 2 can be described by:

```
adder (carryIn, ([], []))     = ([], carryIn)
adder (carryIn, (a:as, b:bs)) = (sum:sums, carryOut)
   where
      (sum, carry)     = fullAdd (carryIn,(a, b))
      (sums, carryOut) = adder (carry, (as, bs))
```

B. Connection Patterns

Standard Conections Patterns

Conection patterns are high-order functions[1] that when used to build circuits, we call them high-order circuits, or circuit generators.

Looking into the `adder` definition and its topology, we want to capture its structure and generalize the conection pattern it uses. We can do this by replacing the `fullAdder` circuit by a not yet defined parameter, that in the function definition, it becomes another input to the circuit, that we call it `circ`:

```
row circ (carryIn, ([])) = ([], carryIn)
row circ (carryIn, a:as) = (b:bs, carryOut)
   where
      (b, carry)     = circ (carryIn, a)
      (bs, carryOut) = row circ (carry, as)
```

The `row` function takes a `circ` circuit, a set of inputs, and connects them as shown in figure 3a. Now, using the `row` circuit generator, the binary adder can be described simply by:

```
adder' (carry, inps) = row fullAdd (carry, inps)
```

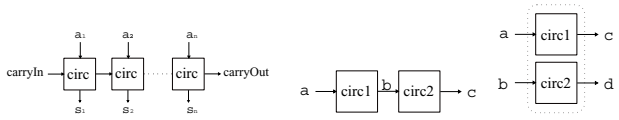
Or we can go further in simplicity by currying the definition:

```
adder'' = row fullAdd
```

Defining `adder'` and `adder''` that way is quite convenient because we gain expressivness by thinking in terms of "circuit generators" instead of recursion over lists.

Seen the advantage of defining connection patterns, we present two more circuit generators that will be used later:

[1]Functions are higher-order when they can take other functions as arguments, and return them as results.



(a) Row (b) Serial (c) Parallel

Fig. 3: Different Circuit Conection Patterns

```
par cir1 cir2 (a, b) = (c, d)
   where
      c = cir1 a
      d = cir2 b
```

It is very usefull to define a more graphical version of function `par`, by defining the infix operator `-|-`:

```
cir1 -|- cir2 = par cir1 cir2
```

And lastly serial circuits conection and its infix operator version:

```
serial cir1 cir2 a = c
   where
      b = cir1 a
      c = cir2 b

cir1 ->- cir2 = serial cir1 cir2
```

II. Interpretations

Though our goal is to introduce Lava as a tool for hardware designers, until now, every line of code that we wrote was in plain Haskell. Actually, designers only need to know which is the set of gates (e.g. `xor2`, `or2`, `inv`, etc) that are available in the Lava module, when describing circuits in terms of those definitions, for the circuit to be used by any of the following interpretations.

A. Simulation

We can simulate a circuit using the `simulate` operation, the circuit and the state of the inputs to the circuit, for instance:

```
simulate adder (low,([high,low],[low,high])
```

yields: `([high,high],low)`. We can also simulate sequences of inputs with the `simulateSeq` operation:

```
simulateSeq halfAdd [(low,low),(high,low),(low,high)]
```

that will return `[(low,low),(high,low),(high,low)]`

B. Formal Verification

Verification in Lava can be done by means of feeding the circuit description with symbolic inputs, and using the output of the circuit as the input to a SAT solver [2], the one we used is called Minisat. We shall verify all properties of the sum, to be sure that our adder is correctly designed. But beforehand, we need to change a little the definition of the `adder` circuit, just to make verification examples easier. We define an adder that does not take in a carry bit, and throws away the resulting carry.

```
adder2 ([],[]) = []
adder2 (a:as, b:bs) = sum:sums
  where
    (sum, carry)     = halfAdd (a, b)
    (sums, carryOut) = adder (carry, (as, bs))
```

Now we state the commutative property of the sum as:

```
prop_AdderCommutative (as, bs) = ok
   where
      out1 = adder2 (as, bs)
      out2 = adder2 (bs, as)
      ok   = out1 <==> out2
```

Note that the `<==>` operator is the infix version of a function that maps between two lists the gate `xnor2`, which is the logical equality operation. Because it is very hard to automatically verify properties for any size, we will define a new property which includes the size of the circuit to be verified:

```
prop_AdderCommutative_ForSize n =
   forAll (list n) $ \as ->
     forAll (list n) $ \bs ->
       prop_AdderCommutative (as, bs)
```

So now we do verification by calling Minisat from Lava giving the size number:

```
minisat (prop_AdderCommutative_ForSize 32)
```

In case our circuit is correctly designed we get:

```
Minisat: ... (t=0.00system) Valid.
```

Otherwise we can get any of the next results:

```
Minisat: ... (t=0.00system) Falsifiable.
Minisat: ... (t=0.00system) Inderterminate.
```

Associative property can be stated as follows:

```
prop_AdderAssociative (as, bs, cs) = ok
   where
      out1 = adder2 (adder2 (as, bs), cs)
      out2 = adder2 (as, adder2 (bs, cs))
      ok   = out1 <==> out2
```

To verify that zero is the identity element of the addition, we will add extra logic to the circuit to transform one of the two numbers to zero:

```
alwaysLow :: [Signal Bool] -> [Signal Bool]
alwaysLow (as) = [low | n <- [1..n]]
   where
      n = length as

addZero = (id -|- alwaysLow) ->- adder2
-- id is the identity function
```

Now we do the verification on the new circuit:

```
prop_AdderZero (as,bs) = ok
   where
      out = addZero (as, bs)
      ok   = out <==> as
```

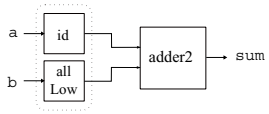


Fig. 4: addZero circuit

C. Generating RTL Description

As well as we use circuit descriptions and symbolic data to prove some properties, we can also use it to generate VHDL code. After describing the Brent-Kung fast adder, we will show how to generate a VHDL netlist from the Lava description.

III. Fast Adder Description and Verification

We are going to design a fast adder based on the Brent and Kung paper [3], and also based in a recursive pattern proposed by M.Sheeran [4] to generate a parallel prefix network used to compute the carries.

A. Brent-Kung Adder

1) Brent-Kung Operator: The $\circ$ operator is defined as:

$$(g, p) \circ (\hat{g}, \hat{p}) = (g \vee (p \wedge \hat{g}), p \wedge \hat{g})$$

In Lava, we can write it following the figure 5:

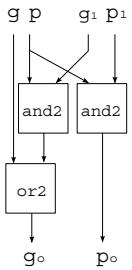


Fig. 5: Brent-Kung Dot Operator

```
dotOp ((g1, p1) ,(g, p)) = (go, po)
   where
      go = or2 (g, and2 (p, g1))
      po = and2 (p, p1)
```

If `dotOp` describes correctly the operator, it must fulfill the associative property, as Brent and Kung demostrated. The property can be stated as:

```
checkAssociativeDotOp (a, b, c) = ok
   where
      (d, e) = dotOp (a, dotOp (b, c))
      (f, g) = dotOp (dotOp (a, b), c)
      ok1    = d <=> f
      ok2    = e <=> g
      ok     = ok1 <=> ok2
```

2) Generate and Propagate Circuit: Now we have to describe a circuit that computes the generate and propagate signal, defined as:

$$g_i = a_i \wedge b_i, p_i = a_i \oplus b_i$$

Those signals are generated in parellel given two binary numbers a[n] and b[n] of length n.

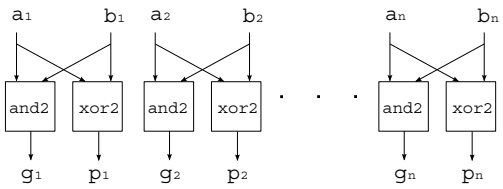


Fig. 6: Generate and Propagate Circuit

The Lava code is:

```
gAndPs ([],[]) = []
gAndPs (a:as, b:bs) = (g,p):gps
   where
      (g, p) = (and2 (a, b),xor2 (a, b))
      gps    = gAndPs (as, bs)
```

3) Overall Circuit: Asuming that we already have the carry chain computation block designed, denoted as the parallel prefix network, the overall circuit proposed in Brent and Kung paper can be conviniently seen as in figure 7.

Looking at the right of figure 7 it can be found the Lava description of the overall pattern, which is done by conecting

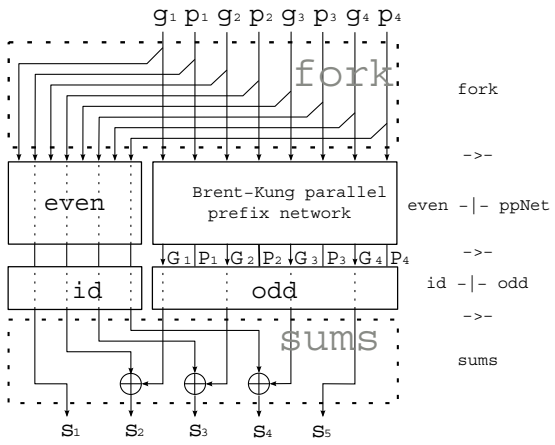


Fig. 7: Brent-Kung Adder

other blocks using the `->-` and the `-|-` circuit generators. The Lava code will be exactly the same as the picture description, but all in the same line. Again, looking into figure 7 we shall define the `fork`, `evens`, `odds` and `sums` circuits:

```
fork as = (as, as)
--
even as = cs
   where
      (bs,cs) = unzip as
--
odd as = bs
   where
      (bs,cs) = unzip as
-- some shorter definitions:
dropP = id -|- odds
dropG = evens -|- ppNet
--
sums (a:as,bs) = (a:lastXor (as,init bs),cOut)
   where
      cOut = last bs
--
lastXor (as, bs) = map xor2 cs
   where
      cs = zipp (as, bs)
```

And the overall circuit will be:

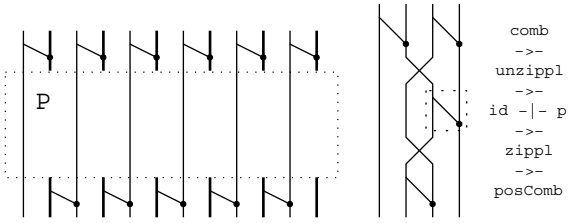
```
fastAdd = gAndPs ->- fork ->- dropG ->- dropP ->-
          sums
```

B. Parallel Prefix Network

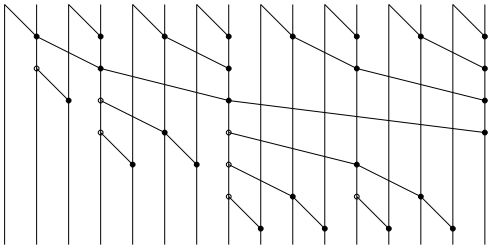
We shall now describe a parallel prefix network with maximum fan-out of two, i.e. the Brent-Kung carry chain computation block. Sheeran [4] proposed a recurrence pattern (figure 8a) called "wrap" where if in every step of iteration, we take the result of the previous iteration (the P circuit) and we apply the dot operation as shown, it can lead us to build parallel prefix networks like figure 8c.

Figure 8b are the first two iterations of circuit `ppNet`, where the doted box is the base case of the description, and black dots are the `dotOp`. What we are producing is networks like figure 8c. So let's describe `ppNet` circuit:

```
dop [a, b] = [a, dotOp(a, b)]
--
unzipl []         = ([],[])
unzipl [a]        = ([a], [])
```



(a) Recursive Parallel Construction (b) ppNet First iterations



(c) Brent-Kung Parallel Prefix Network

Fig. 8: Parallel Prefix Network Construction

```
unzipl (a:b:abs) = (a:as, b:bs)
   where
      (as, bs) = unzipl abs
--
zipl ([], [])     = []
zipl ([a], [])    = []
zipl (a:as, b:bs) = a:b:zipl(as, bs)
-- zipl and unzipl are the key to
-- have a binary adder that accepts
-- any input sizes
--
comb []     = []
comb [a]    = []
comb (a:as) = dop [a, head as] ++ comb (tail as)
--
posComb (a:as)  = a: (comb (init as))++ [last as]
--
miti p = unzipl ->- (id -|- p) ->- zipl
--
wrap p = comb ->- miti p ->- posComb
```

So finally, `ppNet` from figure 7 can be described as:

```
ppNet [a]    = []
ppNet [a, b] = dop [a, b]
ppNet as     = wrap ppNet as
```

IV. Formal Verification and VHDL netlist generation of the fast adder

A. Verification

We could easily modify the `fastAdd` to make it drop the carry out, and we will have a new circuit called `fastAdd2` that should do the same as the `adder2`. To state that in Lava we can write:

```
prop_Equivalent adder2 fastadd2 a = ok
   where
      out1 = adder2   a
      out2 = fastadd2 a
      ok   = out1 <==> out2
```

B. Generating VHDL Description

It can be done as simple as defining the next function:

```
fastAdder n = writeVhdlInputOutputNoClk
              "BrentKungFastAdder" fastAdd
              (varList n "a", varList n "b")
              (varList n "sum", var "cout")
```

To generate the VDHL netlist, we must specify the actual size of the adder, so we can use it as `fastAdder 16` and we will get the BrentKungFastAdder.vhl file which is 376 lines long. If this netlist is to be used by a synthesis tool, it should not touch the wires so as to preserve the network structure.

V. Conclusion

Using functional programing we managed to describe, verify and generate VHDL description of both adders in less than 150 lines of code (comments included), which is less than halve of the VHDL 16 bits adder generated code.

We want to point out some advantages of this aproach:

- We can generate adders of any size
- Verification code is fully reusable
- We work with an unified language for description, simulation and verification resulting in a simpler design flow

Functional Programing give us the ability to design correct circuits in a simple and compact way, easily integrated into a standard design flow. We also want to stress again that given the fact that Lava circuits are plain Haskell programs, System Level simulations can be easily done, when needed. It is worth to mention the use of Functional Programing expresiveness to describe circuits that adapt to their context, for example to the delay profile of the inputs, as can be found in Sheeran's [4] paper. Functional Programing techniques are being also [5] used to take into account non functional properties as area, power consumption and timing, even when working at a high level of abstraction on early stages of the design.

Acknowledgment

The author would like to thank to M. Sheeran and K. Claessen for making Lava and its documentation be freely available.

References

[1] K. Claessen and M. Sheeran, "A tutorial on Lava: A hardware descritption and verification system," Website, 2000, http://www.cs.chalmers.se/~koen/Lava.

[2] K. Claessen, N. Een, M. Sheeran, and N. Sorensson, "SAT-Solving in practice," in *9th International Workshop on Discrete Event Systems (WODES'08), Göteborg, Sweden*, May 2008.

[3] R. P. Brent and H. T. Kung, "A Regular Layout for Parallel Adders," *IEEE Transaction on Computers*, vol. C-31, Issue: 3, pp. 260–264, 1982.

[4] M. Sheeran, "Parallel prefix network generation: an application of functional programming In Hardware Design and Functional Languages," in *Hardware design and Functional Languages (HFL07), Braga, Portugal*, March 2007.

[5] E. Axelsson, K. Claessen, and M. Sheeran, "Wired: Wire-aware circuit design," in *Proc. of Conference on Correct Hardware Design and Verification Methods (CHARME)*, ser. Lecture Notes in Computer Science, vol. 3725. Springer Verlag, October 2005.