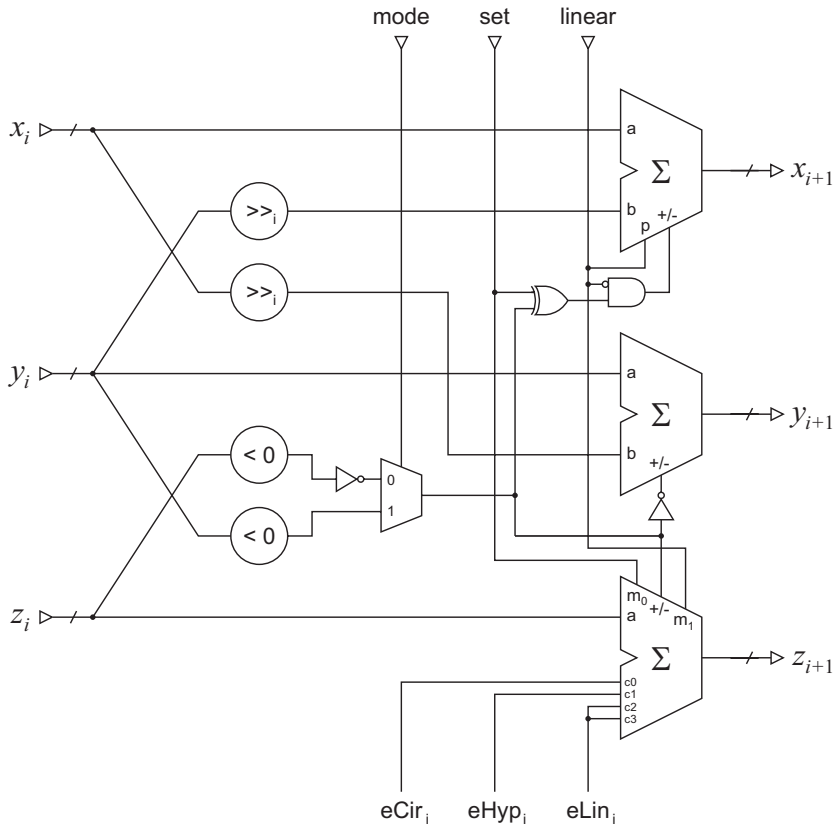
6 Parametrization

This tutorial illustrates how the features of Lava can be used to design parametrized circuits that can be used as building blocks in larger designs. Parametrized circuits, or parametrized ‘cores’, feature a number of functional and implementation alternatives that are selected by parameters provided at the design interface. Parametrization can be used to make a library of common functions available to users whilst providing flexibility and hiding the low-level details of the designs. The steps in describing a CORDIC processor are given here to demonstrate how the features of Lava can be used to parametrize the function, resource usage and performance of designs. Figure 6.1 shows the implementation of the CORDIC processor iteration described in this tutorial.

Figure 6.1 CORDIC processor iteration implementation.



The CORDIC processor implements the COordinate Rotational Digital Computer (CORDIC) Algorithm. This was developed by Volder[1] to implement trigonometric equations and was later extended by Walther[2] to implement a larger range of equations, including hyperbolic and square root equations. The CORDIC algorithm is an iterative method that computes these functions using only shifts and adds. It is designed to compute a range of equation sets and modes. Each iteration computes three equations, the result of which depends on the desired set and mode. The general function of the iteration can be expressed with the following equations:

$$x_{i+1} = x_i - 2^{-i} m d_i y_i$$

$$y_{i+1} = y_i - 2^{-i} d_i x_i$$

$$z_{i+1} = z_i - d_i e_i$$

where:

$$d_i = \begin{cases} \text{if } z_i < 0 \text{ then } -1 \text{ else } 1 & \text{for rotation mode} \\ \text{if } y_i < 0 \text{ then } 1 \text{ else -1} & \text{for vector mode} \end{cases}$$

$$e_i = \begin{cases} \text{atan } (2^{-i}) & \text{for circular equation set} \\ \text{atanh } (2^{-i}) & \text{for hyperbolic equation set} \\ 2^{-i} & \text{for linear equation set} \end{cases}$$

$$m = \begin{cases} 1 & \text{for circular equation set} \\ -1 & \text{for hyperbolic equation set} \\ 0 & \text{for linear equation set} \end{cases}$$

The iteration design in Figure 6.1 is a direct implementation of the above equations. This consists of 3 adder/subtractors, two shifters and control logic for selecting the adder/subtractor mode. The complete design can be implemented by completely unrolling the iterations, by composing them in series, or by partially unrolling the iterations and including some additional control logic to perform resource sharing. All arithmetic is 2's complement fixed-point arithmetic.

The circuit is controlled by the input signals: mode, set and linear. The control logic takes these signals as inputs and outputs the adder/subtractor control signals. All adder/subtractors have a add/sub input that when low, selects add mode and when high, selects subtract mode. The top adder/substractor includes an input p that when high passes the input a directly to the output. The lower adder/subtractor has two inputs m0 and m1 that select between 4 constants (as described in the previous tutorial). The variables d_i and m take the value of 1, -1, 0, and so their effect on the result can be computed by selecting the add/sub mode. The constants e_i, are provided in the constant adder/subtractor. The shifters are constant shifters that implement the 2^i term by a left shift of i places.

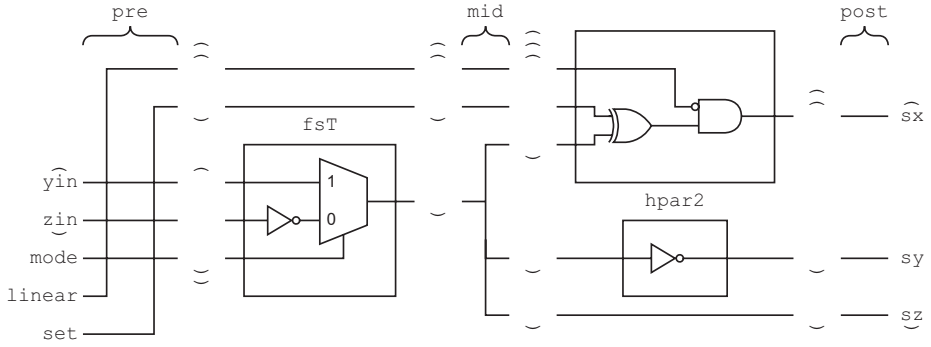
In this tutorial we shall how illustrate how to describe the CORDIC design such that the number of iterations, arithmetic precision and degree of pipelining are all provided as design parameters.

6.1 Composition

The design can be divided into three types of circuit: the adder/subtractors, the constant shifters and the control logic. The constant shifters can be implemented as wires and hence do not require layout. Since we assume 2's complement arithmetic, the < 0 operator can also be implemented as a wire by selecting the most-significant bit (MSB) from the y_i and z_i inputs. The adder/subtractors are variations on the design in the previous tutorial and have the same layout. In this section we shall first show how the control logic can be composed, second demonstrate how parametrized wiring circuits can be described and third show how all the blocks can be composed together.

From the previous tutorial, and the fact that there are no more than 4-inputs, we know that the adder/subtractor designs will occupy a single slice column. One possible layout is to place the adders beside each other horizontally. The adder/subtractors each have a unique add/sub input signal originating from the control logic. The control logic can be mapped to three LUTs, each of which place efficiently under their corresponding adder/subtractor. To composition involves composing the 3 LUTs in series, and then composing the result in series below the second inputs of the adder/subtractors (the add/sub inputs). The series composition is shown in Figure 6.2.

Figure 6.2 Series composition of the control logic. The brackets indicate the position of the brackets in the input and output types of each composition and wiring function.



This composition can be described in Lava as follows:

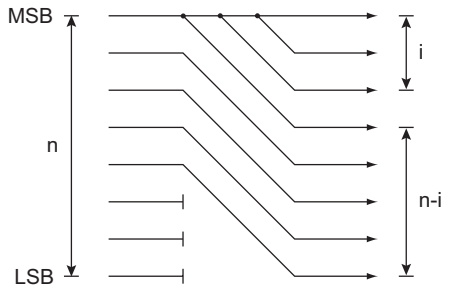
```
invMuxFunc :: Bool -> Bool -> Bool -> Bool
invMuxFunc mode zmsb ymsb = muxFn mode (not zmsb) ymsb

xorNotAnd :: Bool -> Bool -> Bool -> Bool
xorNotAnd a s l = (a /= s) && (not l)

modeLogic :: Bit -> Bit -> Bit -> (Bit, Bit) -> (Bit, Bit, Bit)
modeLogic s l m
  = pre s l m >=> fsT(lut3 invMuxFunc) >=>
      mid >-> snD(inv `hpar2` lut3 xorNotAnd) >=> post
    where
    pre s l m (z, y) = ((m, z, y), (s, l))
    mid (a, (s, l)) = (a, (a, (a, s, l)))
    post (a, (b, c)) = (a, b, c)
```

The < 0 operator can be implemented by a function that takes a bus and returns the wire that represents the MSB. We use the type `[Bit]` to represent a bus, where the head of the list is the least-significant bit (LSB) and the last element of the list is the most-significant bit. Therefore the < 0 operator can be implemented as a recursive function that returns the last element of a list. This is written in Lava as follows:

```
takeTail [l] = l
takeTail (l:ls) = takeTail ls
```

The right shift operator shifts the wires in a bus by i places towards the LSB and sign extends the result. Figure 6.3 shows the wire selection according to the bus width parameter n, and the shift parameter i. The result is formed by taking the n-i most-significant bits and combining them with the MSB, or sign bit, repeated i times.

Figure 6.3 Right shift operator bus selection according to the bus width parameter n and the shift parameter i.



The right shift operation can be implemented by the following parametrized function:

```
rightShift n i x
  = takeTailN (n-i) x ++ fillList msb i
  where
  msb = takeTail x
```

The `rightShift` function takes the last (n-i) elements of the input list x (using the function `takeTailN`) and concatinates these with the last element of x repeated i times (using the function `fillList`). The functions `takeTailN` and `fillList` are defined below.

```
takeTailNAux c [] = []
takeTailNAux 0 l = l
takeTailNAux c (l:ls) = takeTailNAux (c-1) ls

takeTailN c l = takeTailNAux (length l-c) l

fillList e 0 = []
fillList e n = e:(fillList e (n-1))
```

The bottom adder/subtractor of Figure 6.1 was described in the previous tutorial. The other two adders/subtractors can be implemented by the same circuit, with the middle adder/subtractors p input tied to ground. These are described in Lava as follows:

```
addSubFunc :: Bool -> Bool -> Bool -> Bool -> Bool
addSubFunc a b s p = muxFn p ((a == b) == s) a

oneBitAddSub :: Bit -> Bit -> (Bit, (Bit, Bit)) -> (Bit, Bit)
oneBitAddSub s p (cin, (a,b))
  = (sum, cout)
    where
    part_sum = lut4 addSubFunc (a, b, s, p)
    sum = xorcy (part_sum, cin)
    cout = muxcy (part_sum, (a, cin))

addSub :: Int -> Bit -> (Bit, ([Bit], [Bit])) -> ([Bit], Bit)
addSub n p (s, (a, b))
  = col n (oneBitAddSub s p) (s, zip a b)
```

It is now straightforward to compose the adder/subtractors in parallel with each other and place them horizontally. Using `hpar2` operator for this composition forms the input and output wires to have the type of two nested pairs. This makes it possible to compose the other blocks to the correct inputs in series using the `fsT` and `snD` operators. Some additional wiring is required to seperate out the add/sub inputs for composition with the control logic and to tidy up the outputs. The adder/subtractors are composed as follows:

```
addSubIter n s l c0 c1 c2 c3
  = pre >=> addSubConst4 n s l c0 c1 c2 c3 `hpar2`
    (addSub n gnd `hpar2` addSub n l) >=> post
  where
  pre ((sz, sy, sx), (za, ((ya, yb), (xa, xb))))
    = ((sz, za), ((sy, (ya, yb)), (sx, (xa, xb))))
  post ((zout, cz), ((yout, cy), (xout, cx)))
    = (zout, yout, xout)
```

All the circuit blocks can now be composed together to form a single iteration of the CORDIC processor:

```
cordicIter i n s l m c0 c1 c2 c3
  = pre >=> snD(snD(snD(rightShift n i) `par2`
            snD(rightShift n i))) >=>
    fsT((takeTail `par2` takeTail) >=> modeLogic s l m) /\
    addSubIter n s l c0 c1 c2 c3
  where
  pre (zin, yin, xin)
    = ((zin, yin), (zin, ((yin, xin), (xin, yin))))
```

The CORDIC processor composition is completed by composing a number of iterations together as a horizontal 2-sided composition. The Lava description for this is as follows:

```
cordic numIters bitWidth point set linear mode clk
  = hser cir
  where
  numItersInt = round numIters
  is = [0..numItersInt-1]
  c0s = [floatToBool bitWidth point e | e <- eCir]
  c1s = [floatToBool bitWidth point e | e <- eHyp]
  c2s = [floatToBool bitWidth point e | e <- eLin]
  c3s = [floatToBool bitWidth point e | e <- eLin]
  eCir = [atan(2**(-i)) | i <- [0..((numIters)-1)]]
  eHyp = [atanh(2**(-i)) | i <- [0..((numIters)-1)]]
  eLin = [(2.0**(-i)) | i <- [0..((numIters)-1)]]
  cir = [cordicIter i bitWidth set linear mode c0 c1 c2 c3 |
         (i, c0, c1, c2, c3) <- zip5 is c0s c1s c2s c3s]
```

Note that the constants are computed as floating point numbers and provided to the design as it is created. The function `floatToBool` (given in the Appendix) converts the floating point values to 2's complement fixed point boolean constants. Partial application is used to initialise the constant adder circuit with these boolean constants. The parameter `numIters` determines the number of iterations that are implemented and the parameter `bitWidth` determines the precision of the arithmetic. A designer can use these parameters to trade-off the resource usage against the numerical accuracy of the processor. An implementation giving the optimal resource usage for a given accuracy, or the optimal accuracy for a given resource usage, can be selected by simply changing the parameters.

6.2 Pipelining

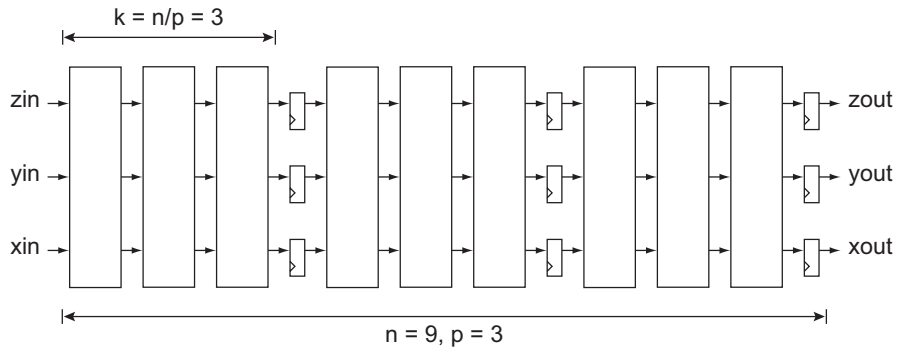
The CORDIC design described so far, is an entirely combinational circuit. The performance of the circuit is limited by the critical path, which passes through all of the iterations. Therefore, the maximum processing rate decreases as the number of iterations, and hence numerical accuracy, increases. The critical path can be broken by pipelining the iterations, which increases throughput at the expense of increasing latency. In this

section we show a higher-order function can be used to parametrically pipeline the CORDIC design, and indeed, any other design featuring a regular 2-sided composition.

Figure 6.4 shows the parameters n and p are used to determine the location of pipeline registers. Registers are inserted on all connecting wires between the iterations. The number of iterations is given by the parameter n and the number of pipeline stages by the parameter p. Pipeline registers are inserted every k=n/p iterations.

Figure 6.4

Parametrized pipelining of the CORDIC processor. The number of iterations is given by the parameter n and the number of pipeline stages by the parameter p. Pipeline registers are inserted every k=n/p iterations.



The iterations of the CORDIC processor are represented in Lava as a list of circuits. The processor can be pipelined by transforming this list into a list of circuits, where every kth circuit is replaced with the original circuit composed with a pipeline register. A higher-order function (functions where the arguments are themselves functions) can be written in Lava to take the list of circuits, the pipeline circuit and perform the transformation. This function can be written as a simple recursion in Lava:

```
addPipe i k [] pipe = []
addPipe 1 k (c:cs) pipe = (c >-> pipe) : addPipe k k cs pipe
addPipe i k (c:cs) pipe = c : addPipe (i-1) k cs pipe

pipeline n p cirs pipe =
  addPipe k k cirs pipe
  where
  k = round (n/p)
```

The series composition `c >-> pipe` composes element c from the list with the pipeline registers `pipe`. Since the pipeline registers are themselves a parameter, this approach can be used to pipeline general 2-sided regular compositions because the correct pipeline registers to match the wiring pattern can be provided. The pipeline registers suitable for the CORDIC processor are defined as follows:

```
pipeStage clk = pre >=> vmaP (fd clk) `hpar2` (vmaP (fd clk)
`hpar2` vmaP (fd clk)) >=> post
  where
  pre (d0, d1, d2)
    = (d0, (d1, d2))
  post (q0, (q1, q2))
    = (q0, q1, q2)
```

To use this pipelining function a few simple changes need to be made to the top-level CORDIC processor function:

```
cordic numIters numPipes bitWidth point set linear mode clk
  = hser pipeCir
  where
  numItersInt = round numIters
  is = [0..numItersInt-1]
  c0s = [floatToBool bitWidth point e | e <- eCir]
  c1s = [floatToBool bitWidth point e | e <- eHyp]
  c2s = [floatToBool bitWidth point e | e <- eLin]
  c3s = [floatToBool bitWidth point e | e <- eLin]
  eCir = [atan(2**(-i)) | i <- [0..((numIters)-1)]]
  eHyp = [atanh(2**(-i)) | i <- [0..((numIters)-1)]]
  eLin = [(2.0**(-i)) | i <- [0..((numIters)-1)]]
  cir = [cordicIter i bitWidth set linear mode c0 c1 c2 c3 |
         (i, c0, c1, c2, c3) <- zip5 is c0s c1s c2s c3s]
  pipeCir = pipeline numIters numPipes cir (pipeStage clk)
```

Now, the parameter `numPipes` is included, which determines the number of pipeline stages in the design.

6.3 Exercises

1. Provide a specialised version of the CORDIC processor that takes mode, set and linear as parameters.

2. Use the `pipeline` function to pipeline the N-bit pattern matcher design.

6.4 References

[1] J. Volder, “The CORDIC Trigonometric Computing Technique”, IRE Trans. Electronic Computing, Vol. EC-8, pp330-334, 1959.
[2] J. Walther, “A Unified Algorithm for Elementary Function”, Spring Joint Computer Conference, pp379-385, 1971.
[3] R. Andraka, “A survey of CORDIC algorithms for FPGA based computers”, FPGA Conference, 1998.
[4] E. Keller, Dynamic Circuit Specialization of a CORDIC Processor, SPIE Conference, 2000.
[5] “LogiCORE CORDIC Product Specification”, http://www.xilinx.com/ipcenter/catalog/search/logicore/xilinx_cordic.htm, Xilinx Inc., 2002.

6.5 Appendix

Function `floatToBool`: converts a floating point number into a list of boolean constants that represent a 2’s complement fixed point number. Parameter n is the precision, p is the point position and v is the floating point number.

```
convertBit c w v =
  if c == 0 then
    if d >= 1 then
      [True]
    else
      [False]
  else
    if d >= 1 then
      convertBit (c-1) (w-1) (v-(2**w)) ++ [True]
    else
      convertBit (c-1) (w-1) v ++ [False]
  where
  d = v/(2**w)

floatToBool n p v =
  if v < 0 then convertBit (n-2) (p-2) (v+msb) ++ [True]
           else convertBit (n-2) (p-2) v ++ [False]
  where
  msb = 2**(p-1)
```

Function `computeCordic`: software implementation of the CORDIC algorithm.

```
data Set =   Circular
           | Hyperbolic
           | Linear

data Mode =   Rotation
            | Vector

computeX i s x y d =
  x - y*m*d*2**(-i)
  where
```

```
  m = case s of Circular   -> 1
                Hyperbolic -> -1
                Linear     -> 0

computeY i x y d =
  y + x*d*2**(-i)

computeZ i s z d =
  z - d*e
  where
  e = case s of Circular   -> atan(2**(-i))
                Hyperbolic -> atanh(2**(-i))
                Linear     -> 2**(-i)

computeCordic i ni s m (x,y,z) =
  if i==ni then
             (nx,ny,nz)
           else
             computeCordic (i+1) ni s m (nx,ny,nz)
  where
  nx = computeX i s x y d
  ny = computeY i x y d
  nz = computeZ i s z d
  d = case m of Rotation -> if z<0 then -1
                                   else 1
                Vector   -> if y<0 then 1
                                   else -1
```