Organización del Computador
Descripción de Hardware con Lava
– VERSIÓN PRELIMINAR –

B. Gonzalez Kriegel - N. Wolovick

17 de abril de 2001

1 Introducción

Lava es un sistema experimental de descripción y verificación de hardware utilizado como plataforma de prueba en la verificación formal de hardware en la universidad de Chalmers. Lava se desarrolla paralelamente en Xilinix Inc., uno de los mayores poveedores de *FPGA*[1] del mundo.

Lava consiste en un lenguaje de descripción de hardware *HDL* inmerso en Haskell. Se compone de una serie de módulos que brindan al usuario medios para construir, simular y verificar sus circuitos.

Existen muchos *HDL*s establecidos en el mercado (VHDL, VeriLog) que podrían ser utilizados, sin embargo Lava gana en sencillez pues no tiene en cuenta detalles de bajo nivel como tiempos y estructura interna de compuertas y se concentra en brindar funciones constructoras de alto orden para lograr descripciones económicas, elegantes, y llenas de información, a través de *patrones de conexión*.

Lava se basa en dos ideas fundamentales:

- La *descripción* del circuito captura los aspectos sobresalientes de estructura y comportamiento.
- Esta descripción puede ser analizada de varias maneras, generando diversas *interpretaciones*.

La primera idea permite al diseñador concentrarse en un sólo lenguaje para describir los dos aspectos fundamentales del diseño. Mientras que la segunda posibilita que el diseño sea *verificable* y *construible*.

Las interpretaciones posibles de una descripción van desde la simple simulación, a la generación de descripciones simbólicas compatibles con probadores de teoremas, pasando por generadores de código VHDL que permiten conectarlo con las herramientas *CAD* del mercado.

La idea de descripción de hardware mediante equaciones recursivas no es nueva pues se viene trabajando desde hace casi dos décadas en el tema [3] [6]. Existen otros acercamientos alternativos a la descripción de hardware mediante funciones, como por ejemplo el lenguaje relacional Ruby [7] que comparte muchas características con Lava.

[1] Field Programmable Gate Arrays.

1.1 Un Ejemplo Concreto

Mediante un ejemplo presentaremos los aspectos operativos generales de Lava y su entorno. Nuestro objetivo será describir un semisumador, para luego simularlo y construirlo. Comenzamos con una especificación de la función *semiSum* : $\mathcal{B}^2 \rightarrow \mathcal{B}^2$ a través de una tabla.

a	b	sum	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

La función resulta tan inmediata que podemos llegar de manera directa a una implementación utilizando compuertas lógicas discretas En el archivo de texto plano `primero.hs` escribimos la

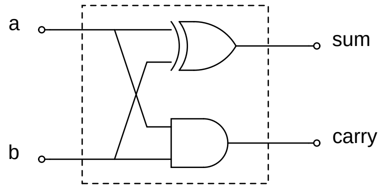

Figura 1: SemiSumador

descripción Lava del semisumador.

```
primero.hs
1 import Lava
2
3 semiSum (a,b) = (sum,carry)
4   where
5         sum   = xor2 (a,b)
6         carry = and2 (a,b)
```

La primera línea incluye los módulos básicos que componen Lava, en particular el que define las compuertas lógicas `xor2` y `and2`. Luego, a partir de la línea 3 se define el circuito que posee una entrada de dos señales y una salida, también de dos señales. En general nuestros circuitos adherirán a esta convención, es decir una entrada y una salida donde cualquier de las dos puede ser compuesta.

Llamamos al sistema

```
% /usr/local/Lava2000/Bin/lava
-- Lava 2000 -------------------------------------------------------------
.
Prelude>
```

y cargamos nuestra primera descripción

```
Prelude> :l primero.hs
.
primero.hs
Main>
```

Con la descripción en memoria ejecutamos la *simulación* (la más sencilla de las interpretaciones) para valores puntuales del dominio.

```
Main> simulate semiSum (low,low)
(low,low)
Main> simulate semiSum (high,high)
(low,high)
```

Suele ser conveniente agregar definiciones que sean los casos de prueba y para escribir un poco menos. Modificamos `primero.hs` para incluir

```
primero.hs
.
prueba1 = simulate semiSum (low,high)
prueba2 = simulate semiSum (high,low)
```

Recargamos las definiciones para reflejar los cambios y utilizamos las definiciones como abreviaciones de los casos de prueba.

```
:r
.
primero.hs
Main> prueba1
(high,low)
Main> prueba2
(high,low)
```

También es posible simular sobre una secuencia de casos

```
Main> simulateSeq semiSum [(low,low),(low,high),(high,low)]
[(low,low),(high,low),(high,low)]
```

o bien sobre la secuencia que cubre todo el dominio[2].

```
Main> simulateSeq semiSum domain
[(low,low),(high,low),(high,low),(low,high)]
```

A partir de la descripción Lava es posible realizar otras interpretaciones como la generación de código VHDL, un estandard de la industria para la descripción de hardware.

```
Main> writeVhdl "semiSum" semiSum
Writing to file "semiSum.vhd" ... Done.
```

El archivo generado `semiSum.vhd`, que supone las definiciones básicas dadas en `Lava2000/Auxiliary/lava.vhd`, es el siguiente:

[2]dado que es finito.

```
-- Generated by Lava 2000

use work.all;

entity
  semiSum
is
port
  -- clock
  ( clk : in bit

  -- inputs
  ; inp_1 : in bit
  ; inp_2 : in bit

  -- outputs
  ; outp_0 : out bit
  ; outp_1 : out bit
  );
end entity semiSum;

architecture
  structural
of
  semiSum
is
  signal w1 : bit;
  signal w2 : bit;
  signal w3 : bit;
  signal w4 : bit;
begin
  c_w2      : entity id    port map (clk, inp_1, w2);
  c_w3      : entity id    port map (clk, inp_2, w3);
  c_w1      : entity xor2  port map (clk, w2, w3, w1);
  c_w4      : entity and2  port map (clk, w2, w3, w4);

  -- naming outputs
  c_outp_0  : entity id    port map (clk, w1, outp_0);
  c_outp_1  : entity id    port map (clk, w4, outp_1);
end structural;
```

Finalmente podemos salir del entorno

```
Main> :q
%
```

2 Circuitos Combinacionales

2.1 Circuitos son Funciones

Un circuito puede ser visto como una caja negra con salidas que dependen funcionalmente de las entradas, por lo que resulta natural elegir funciones para representar circuitos. Una compuerta AND_2, por ejemplo, es una función que toma un par (a, b) como entrada y devuelve un x en su salida que cumple con la funcionalidad de la compuerta

```
x = and2 (a,b)
```

Por convención las señales o cables se agrupan para que todas las cajas negras tengan una única entrada y salida. Estas pueden ser *simples* o *compuestas*. Para el semisumador definido en el módulo `Arithmetic`, las entradas y salidas son señales compuestas denotadas por uplas.

```
(s,cout) = halfAdd (cin,a)
```

Cuando es necesario más de dos cables en la entrada o salida, estas se distribuyen en dos grupos internos. Los lineamientos dados son sólo convenciones que nos permiten utilizar los combinadores de patrones de circuitos que se verán más adelante.

```
(s,cout) = fullAdd (cin,(a,b))
```

Existen también *señales compuestas genéricas*, las cuales no definen a priori cuantas señales contendrán. Por ejemplo la compuerta genérica `andl` utiliza este tipo de señal que se denota con una lista.

```
f1 = andl [low,low,high,low]
f2 = andl [high,high]
```

Se puede definir *señales anónimas* mediante la composición directa de circuitos,

```
x = inv (and2 (a,b))
```

o bien es posible nombrar las señales para reutilizarlas y describir circuitos con *fanout* mayor a 1, o para estructurar nuestra descripción

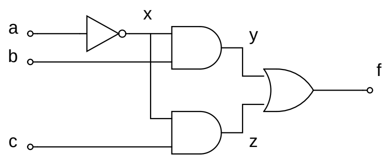

Figura 2: Circuito Combinacional con fanout>1

```
f (a,b,c) = or2 (y,z)
  where
        x = inv a
        y = and2 (x,b)
        z = and2 (x,c)
```

La *abstracción* en la estructura de los circuitos está dada por la definición de circuitos. Este concepto es crucial en el diseño del hardware donde la complejidad crece tan rápidamente que si no somos capaces de definir "cajas negras" que serán construidas a partir de otras más sencillas, cualquier diseño no elemental se torna inmanejable.

En la sección 1.1 describíamos un semisumador y comprobamos que funcionaba como tal. Este circuito puede ser utilizado para construir circuitos más complejos como por ejemplo un sumador completo, el cual no tendrá ninguna forma de ver la estructura interna abstraída por la definición del circuito, por lo que un cambio de estructura manteniendo el comportamiento en el semisumador resulta transparente al sumador completo.

2.1.1 Ejercicios

1. A partir del semisumador y compuertas `xor2` obtenga la descripción de un sumador completo. Simulelo en todo su dominio para comprobar su correcto funcionamiento. Obtenga el código VHDL.

2. Defina y simule un circuito `multiplexor`. Este tiene como entrada un par formado por una señal simple y un par `(x,y)`. La salida es `x` si la señal simple es 0 y `y` en caso contrario.

2.2 Circuitos Combinacionales Básicos

Un circuito combinacional no contiene memoria interna, y esto se traduce en la ausencia retroalimentación, flip-flops o relojes. El circuito más simple que podemos hacer es utilizando compuertas 0-arias, o sea las constantes 1 y 0, o en notación `Lava` las constantes `low` y `high`.

```
nada () = low
```

```
Main> simulate nada ()
low
```

donde una entrada que consiste en ningún cable se denota con `()`.

Siguiendo en la escala de complejidad, tenemos las dos compuertas 1-arias (`id`, `inv`) y ocho de las dieciseis binarias (`and2`, `nand2`, `or2`, `nor2`, `xor2`, `xnor2`, `equiv`, `impl`). También existen compuertas n-arias (`andl`, `nandl`, `orl`, `norl`, `xorl`) para los operadores binarios asociativos y conmutativos, donde la entrada es una señal compuesta genérica.

```
and3 (a,b,c) = andl [a,b,c]
```

```
Main> simulateSeq and3 domain
[low,low,low,low,low,low,low,high]
```

Es interesante, dado que las compuertas son funciones, conocer el tipo de estas.

```
Main> :t low
low :: Signal Bool
Main> :t and2
and2 :: (Signal Bool,Signal Bool) -> Signal Bool
```

`Lava` además de manejar señales booleanas, es capaz de manejar *cables que acarrean señales enteras*, por lo que existen compuertas para manejar este tipo de señales (`plus`, `times`, etc.).

Entre los circuitos combinatorios básicos también existe un multiplexor genérico `mux` que funciona para señales booleanas y enteras, y en el módulo `Arithmetic.hs` tenemos semisumadores y sumadores completos `halfAdd`, `fullAdd`.

En el apéndice A de `Lava2000/Doc/tutorial.ps` [2] econtramos una referencia rápida de todos los módulos de `Lava` que proveen primitivas para la descripción de circuitos así como las interpretaciones implementadas.

2.3 Ejercicios

1. Defina un *ordenador de 2 bits* que toma como entrada un par de bits y devuelve los mismos bits pero el menor a la derecha y el mayor a la izquierda.

2. Use 3 sumadores completos para hacer un sumador de 3 bits. Simule su diseño y compruebe que es correcto.

3 Circuitos más Grandes

3.1 Circuitos que Cablean

Usualmente en un diagrama esquemático de un circuito, no nos preocupamos por el orden o ubicación de las las conexiones, generando bifurcaciones, conjunciones y movimientos de todo tipo, sin respetar ningúna regla. Todas estas formas pueden ser capturadas por circuitos que no operan sobre los valores de las entradas, pero si sobre su orden, a fin de hacer explícita esta información y poder explotarla.

Los patrones de conexión más comunes son el intercambio y la copia.

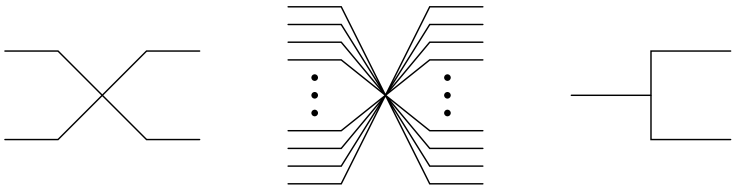

Figura 3: swap, swapl, copy

```
:l Patterns
.
Patterns> simulate swap (low,high)
(high,low)
Patterns> simulate copy low
(low,low)
```

El circuito `zipp` desarma par de señales compuestas en una señal compuesta de pares. El circuito `unzipp` hace lo inverso.

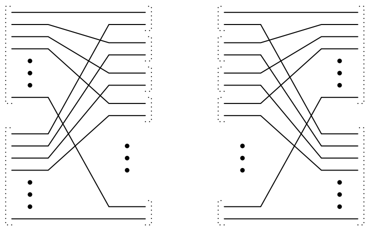

Figura 4: zipp, unzipp

```
Patterns> simulate zipp ([low,low,high,high],[low,high,low,high])
[(low,low),(low,high),(high,low),(high,high)]
```

Para estos circuitos genéricos donde las señales compuestas tienen una cantidad variable de señales, la representación gráfica suele no ser suficientemente expresiva y puede ser conveniente revisar las definiciones dadas en `Lava2000/Modules/Patterns.hs`

Apareamiento y separación también están entre los circuitos que cablean.

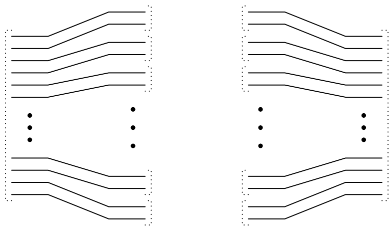

Figura 5: pair,unpair

```
Patterns> simulate pair [low, high, high, low, high, high]
[(low,high),(high,low),(high,high)]
```

Finalmente mostramos las primitivas que parten en dos y rearman un conjunto de señales.

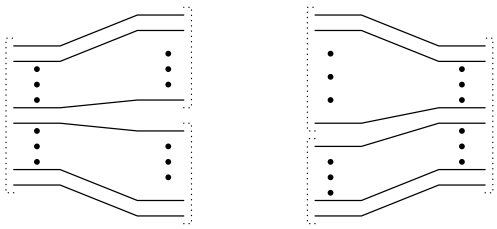

Figura 6: halveList, append

```
Patterns> simulate halveList [low,high,low,high,high,low]
([low,high,low],[high,high,low])
```

3.1.1 Ejercicios

1. Defina los siguientes circuitos de cableo.

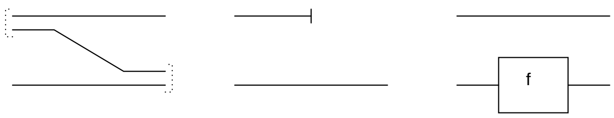

Figura 7: rsh, p1, first

3.2 Circuitos que Convierten Tipos de Señales

Además de las señales booleanas, vimos que `Lava` maneja señales enteras, con lo cual para realizar circuitos mixtos necesitamos de *circuitos de conversión de tipos de señales* que están en el módulo `Arithmetic`. Las dos funciones son `int2bin` y `bin2int`, la primera tiene el siguiente tipo,

```
Patterns> :l Arithmetic
.
Arithmetic> :t bin2int
bin2int :: [Signal Bool] -> Signal Int
```

donde el *LSB* es la primera componente de la señal compuesta genérica. Su circuito inverso es

```
Arithmetic> :t int2bin
int2bin :: Num a => a -> Signal Int -> [Signal Bool]
```

que antes de la señal con el entero toma la cantidad de dígitos binarios que se usarán para representarlo.

```
Arithmetic> simulate (int2bin 3) 4
[low,low,high]
```

Debe quedar claro que `(int2bin 3)` es un circuito, el cual toma como entrada una señal entera simple y devuelve una señal booleana compuesta genérica de tres simples.

Aunque no vayamos a construir ningún circuito con señales enteras, estas funciones serán útiles a la hora de testear circuitos aritméticos como se verá más adelante.

3.3 Patrones de Conexión

Si se quiere diseñar un sumador de 3,4,5 o 6 bits de manera ordenada, se descubre rápidamente que es posible generar cualquier sumador de n bits repitiendo esa misma estructura.

Lava captura estas estructuras comunes a través de *patrones de conexión*. Estos son circuitos de alto orden, es decir circuitos que construyen circuitos a partir de otros circuitos (comparar con las funciones de alto orden de Haskell como `map`, `fold`, `scan`, etc.).

El patrón de conexión más común es la *composición serial* donde dos circuitos se conectan uno a continuación del otro. Su forma prefija es `serial` y su infija es `->-`. Un pequeño ejemplo sería

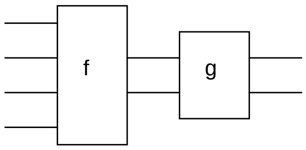

Figura 8: f ->- g

```
Patterns> simulateSeq (and2 ->- inv) domain
[high,high,high,low]
```

Se puede ver que esta operación es asociativa (`f->-(g->-h)=(f->-g)->-h`). Su generalización se denomina `compose`

```
Patterns> :t compose
compose :: [a -> a] -> a -> a
```

y realiza una composición serial de una lista de circuitos donde `serial c1 c2=compose [c1,c2]`

Otro patrón de conexión básico es la *composición paralela* que transforma dos circuitos, cada uno con una entrada y una salida en un circuito que toma un par de entradas y genera un par de salidas. La generalización de este patrón es simplemente el combinador de alto orden de Haskell

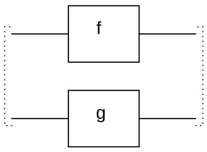

Figura 9: f-|-g

`map`.

```
Patterns> simulateSeq (map inv) (domainList 3)
[[high,high,high],[high,high,low],[high,low,high],[high,low,low],[low,high,high],
[low,high,low],[low,low,high],[low,low,low]]
```

donde vemos que `(domainList 3)` genera el dominio $\mathcal{B}^3$ en forma de señal compuesta genérica.

Otro patrón de paralelización es `parl`, el cual toma dos circuitos y aplica en paralelo el primer circuito a la primera mitad de la señal compuesta genérica y el segundo circuito a la segunda mitad de la señal compuesta genérica.

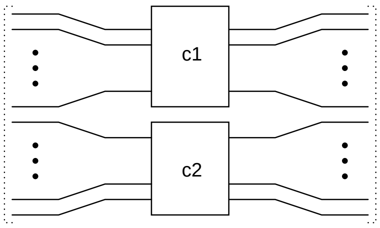

Figura 10: parl c1 c2

```
parl c1 c2 = halveList ->- (c1 -|- c2) ->- append
```

y esta última es la definición del módulo de patrones, que resulta mucho más corta y precisa que el diagrama esquemático. Esta situación suele darse en general, con lo que es aconsejable darle un vistazo a las definiciones que brinda el módulo.

Veamos un ejemplo donde utilizamos estos patrones de conexión y un circuito de cableado para obtener una descripción compacta del circuito de dos inversores.

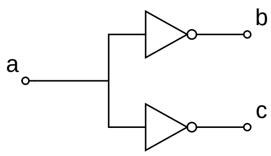

Figura 11: copy ->- (inv -|- inv)

```
dobleinv = copy ->- (inv -|- inv)
```

Alternativamente podríamos haber escrito sin utilizar combinadores ni circuitos de cableado una descripción similar,

```
dobleinv' a = (b,c)
  where
        b = inv a
        c = inv a
```

sólo que no se explicitan en Lava el *sharing*, el paralelismo y la secuenciación, que puede resultar relevante a la hora de dar interpretaciones que evalúen el paralelismo o generen el *layout* físico de los circuitos.

Claramente los circuitos resultan equivalentes, pero es mejor si lo comprobamos utilizando la interpretación de la simulación.

```
Main> simulateSeq dobleinv domain
[(high,high),(low,low)]
Main> simulateSeq dobleinv' domain
[(high,high),(low,low)]
```

Si a un circuito lo podemos expresar de la siguiente forma

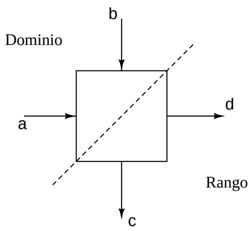

Figura 12: circ :: (a,b) → (c,d)

```
circ :: (a,b) -> (c,d)
```

entonces podemos generar patrones que repliquen el circuito en fila, columna o grilla.

```
Main> :t row
row :: ((a,b) -> (c,a)) -> (a,[b]) -> ([c],a)
```

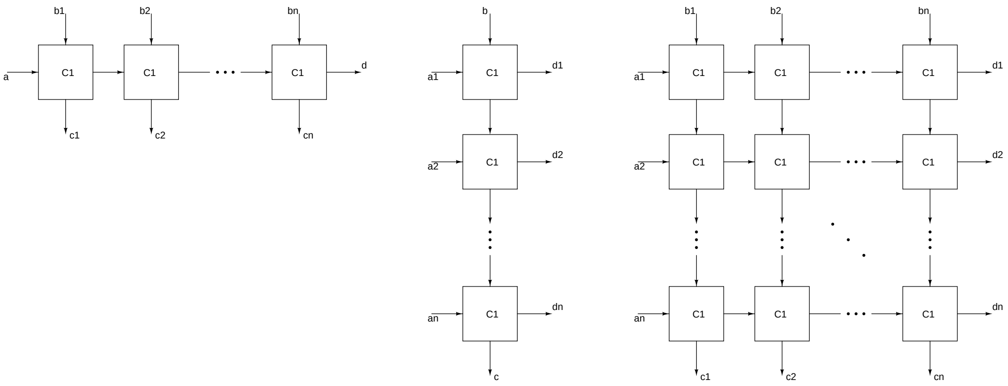

Figura 13: row, column, grid

```
Main> :t column
column :: ((a,b) -> (b,c)) -> ([a],b) -> (b,[c])
Main> :t grid
grid :: ((a,b) -> (b,a)) -> ([a],[b]) -> ([b],[a])
```

Vemos que el patrón de fila conecta en serie los laterales verticales, generando un circuito que a partir de una señal simple que va al lateral vertical de más a la izquierda y de una compuesta genérica que va a los laterales superiores, forma una salida compuesta genérica por los laterales inferiores y una señal simple por el lateral vertical más a la derecha.

La utilidad de estos patrones se ve en el sumador de n bits. Si tenemos un sumador completo con el siguiente tipo,

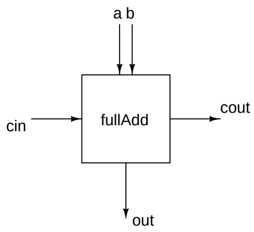

Figura 14: fullAdd

```
Main> :l Arithmetic
.
Arithmetic> :t fullAdd
fullAdd :: (Signal Bool,(Signal Bool,Signal Bool)) -> (Signal Bool,Signal Bool)
```

un *sumador de ondulación* de n bits puede ser descripto como una fila de sumadores completos

```
sumador (cin,inps) = row fullAdd (cin,inps)
```

donde `inps` es una señal compuesta genérica de pares de señales. Más sintéticamente podemos escribir

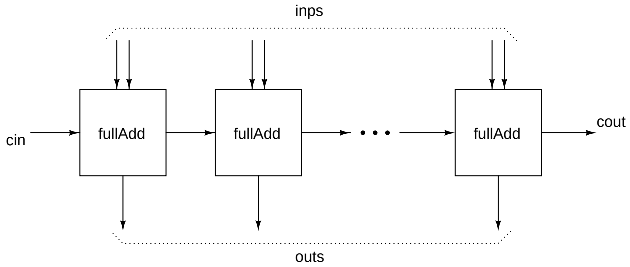

Figura 15: Sumador de n bits

```
sumador = row fullAdd
```

Utilizando patrones de conexión y circuitos conversores, simulamos el sumador

```
Main> simulate sumador (low, zipp ((int2bin 3) 3, (int2bin 3) 2))
([high,low,high],low)
```

Notamos que el circuito de cableo `zipp` se utiliza para reorganizar las entradas de modo que sean compatibles con lo que espera el sumador. Se muestra también un diagrama esquemático de lo que hacemos en la simulación.

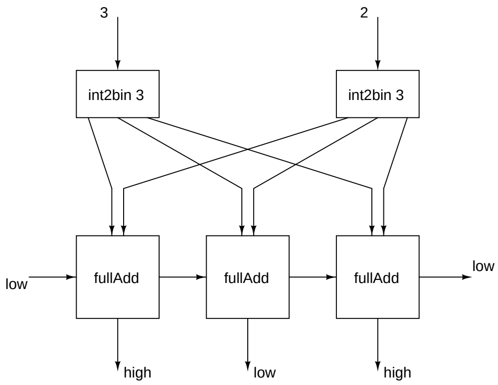

Figura 16: Sumador con (int2bin 3)

3.3.1 Ejercicios

1. Muestre como `first` puede ser definido en términos de primitivas y patrones de conexión.
2. Describa un `bitAdder`, que toma una señal simple y una compuesta genérica que representa un número y devuelve el número original sumando el bit de entrada junto a un acarreo.
3. El circuito de cableo `mirror` se define como sigue en el módulo `Patterns.hs`

```
mirror circ (a,b) = (c,d)
  where
    (d,c) = circ (b,a)
```

Interprete el efecto que tiene respecto a la figura 12 y defina `col` en términos de `row` y `mirror`.

4. Describa el patrón `grid` en términos de `row` y `col`.

5. ¿Es asociativa la composición paralela `(-|-)`?

6. ¿El circuito `(a->-b) -|- (c->-d)` es equivalente a `(a-|-c) ->- (b-|-d)` ?

3.4 Verificación

Lava es capaz de generar una interpretación del circuito compatible con un *probador de teoremas*, a fin de poder *verificar propiedades de seguridad*. Las propiedades de seguridad (*safety properties*) son del tipo "nada malo va a suceder". Ejemplos de propiedades para verificar podrían ser:

- Nunca un semisumador tiene ambas entradas en verdadero.
- El sumador genérico de de n bits siempre es conmutativo.

Además de las propiedades de seguridad existen las *propiedades de estar vivo* (*liveness properties*) que indican "eventualmente algo bueno sucederá", sin embargo este tipo de propiedades no resultan relevantes para los circuitos combinacionales que salvo mínimos retardos presentan un comportamiento atemporal.

Lava se puede conectar a diversos probadores de teoremas, pero sólo utilizaremos VIS [8].

Podemos pensar que una propiedad es un circuito con una entrada, posiblemente compuesta y una salida simple. La tarea del verificador es demostrar, mediante manipulación simbólica, que el circuito devuelve `high` o verdadero para todo valor del dominio, es decir comprueba que la proposición sea una tautología. Los circuitos constantes `low` y `high` dan los resultados esperados.

```
Prelude> :l Lava
.
Lava> vis high
Vis: ... (t=0.05system) \c
Valid.

Lava> vis low
Vis: ... (t=0.06system) \c
Falsifiable.
```

Veamos un ejemplo donde verificamos la conmutatividad del sumador completo. Agregamos la siguente definición al archivo `primero.hs`,

primero.hs
```
.
prop_fullAddConmutativo (c,(a,b)) = ok
  where
        out1 = fullAdd (c,(a,b))
        out2 = fullAdd (c,(b,a))
        ok   = out1 <==> out2
```

para luego cargarlo y obtener a la interpretación que llama al probador de teoremas.

```
Lava> :l primero.hs
.
Main> vis prop_fullAddConmutativo
Vis: ... (t=0.05system) \c
Valid.
```

Efectivamente VIS logró demostrar que el sumador completo es conmutativo. Debemos hacer el uso de la *compuerta genérica de equivalencia* que en notación prefija se denomina `equal` y en infija `<==>`. Esta sirve para verificar la igualdad señal por señal de cualquier tipo de señal compuesta[3].

```
Main> simulate equal ([low,low],[low,low])
high
Main> simulate equal ((low,low),(low,low))
high
```

También podriamos haber verificado esta propiedad por simulación comprobando que la salida es verdadera para todo valor de entrada.

```
Main> simulateSeq prop_fullAddConmutativo domain
[high,high,high,high,high,high,high,high]
```

Podemos definir *propiedades paramétricas respecto a circuitos*. Veamos dos de ellas.

primero.hs

```
.
prop_Equivalente circ1 circ2 a = ok
  where
        out1 = circ1 a
        out2 = circ2 a
        ok   = out1 <==> out2

prop_Conmutativo circ (as,bs) = ok
  where
        out1 = circ (as,bs)
        out2 = circ (bs,as)
        ok   = out1 <==> out2
```

Utilizamos las propiedades en dos ejemplos, en uno verificamos que nuestro semisumador es equivalente al dado por el módulo `Arithmetic` y el otro comprueba la conmutatividad de la compuerta 2-aria `impl`.

```
Main> :r
.
primero.hs
```

[3]No confundir `equal` con `equiv` pues sus dominios son bien distintos: `equiv :: (Signal Bool,Signal Bool) -> Signal Bool` y `equal :: Generic a => (a,a) -> Signal Bool`.

```
Main> vis (prop_Equivalente semiSum halfAdd)
Vis: ... (t=0.07system) \c
Valid.

Main> vis (prop_Conmutativo impl)
Vis: ... (t=0.05system) \c
Falsifiable.
```

Debe quedar claro que los circuitos que son parámetros de la propiedad deberán tener la forma (tipo) adecuado.

Si definimos un comparador genérico de n bits,

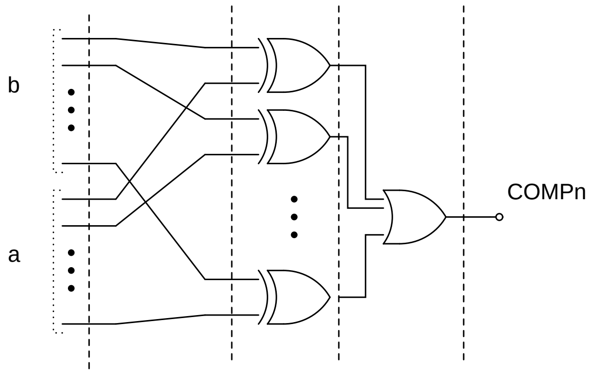

Figura 17: Comparador de n bits

primero.hs

```
.
comp = zipp ->- (map xor2) ->- orl
```

es lógico intentar demostrar que es conmutativo, pero nos encontramos con un problema.

```
Main> :r
.
primero.hs
Main> vis (prop_Conmutativo comp)
ERROR: Unresolved overloading
*** Type       : Fresh [Signal Bool] => IO ProofResult
*** Expression : vis (prop_Conmutativo comp)
```

Sabemos que la propiedad es válida para cualquier tamaño del circuito, sin embargo resulta complicado verificar de manera automática propiedades que valgan para todo n. Debemos elegir un n específico y verificar que la propiedad sobre el circuito genérico se cumple para dicho tamaño.

```
primero.hs
.
prop_Conmutativo_ParaTamanio n circ =
  forAll (list n) $ \as ->
    forAll (list n) $ \bs ->
      prop_Conmutativo circ (as,bs)
```

Ahora si podemos verificar, para distintos tamaños, que nuestro comparador es conmutativo.

```
Main> :r
.
primero.hs
Main> vis (prop_Conmutativo_ParaTamanio 8 comp)
Vis: ... (t=0.06system) \c
Valid.

Main> vis (prop_Conmutativo_ParaTamanio 16 comp)
Vis: ... (t=0.05system) \c
Valid.

Main> vis (prop_Conmutativo_ParaTamanio 32 comp)
Vis: ... (t=0.08system) \c
Valid.
```

Para conocer los resultados intermedios del proceso de verificación, podemos consultar los archivos `./Verify/circuit.mv` y `./Verify/circuit.mv.out` que contienen respectivamente la entrada y la salida de la llamada a VIS[4].

3.4.1 Ejercicios

1. Defina la propiedad `prop_semiSumNuncaAmbasSalidasHigh` y verifique que el semisumador `semiSum` cumple con la propiedad de que sus salidas nunca son altas al mismo tiempo. * ¿Porqué debería suceder esto?

2. Defina la propiedad `prop_Equivalente_ParaTamanio` que verifica la equivalencia de dos circuitos dado un tamaño n.

3. La propiedad de asociatividad sobre un circuito `circ` de dos entradas y una salida es válida cuando se cumple $\forall x, y, z \; circ\ (a, circ\ (b, c)) = circ\ (circ\ (a, b), c))$

   (a) Defina `prop_Asociativo` con parámetro `circ`.

   (b) Encuentre dos compuertas 2-arias, una asociativa y la otra no.

4. (pregunta de Haskell) ¿Cómo escribiría un programa que verificara por ejemplo que el circuito `comp` es conmutativo para tamaños de 1 a n?

5. Defina un sumador completo `fullAdd'` utilizando dos semisumadores y una compuerta `or2`. Verifique la equivalencia entre `fullAdd` y `fullAdd'`.

[4] Existe un pequeño error en Lava que en vez de mostrar el tiempo ocupado por el usuario o el total muestra el tiempo de sistema. Basta con chequear `./Verify/circuit.mv.out` luego de cada demostración para tener el *profiling* completo.

6. *Dado el siguiente circuito

```
alwaysLow () = low        -- toma cualquier cosa y siempre devuelve low
sumador' = copy ->- (alwaysLow -|- zipp) ->- sumador
```

(a) Dibuje el diagrama esquemático.

(b) ¿Qué es lo que hace? ¿Para qué se utiliza `copy` y `alwaysLow`? ¿Se le ocurre otra solución?

(c) Verifique para tamaños $n = \{2, 4, 8, 16, 32, 64, 128\}$ que `sumador'` es conmutativo.

(d) ¿Cómo son los tiempos de verificación en función de n? ¿Resulta más rápido verificar o simular y comparar?[5].

4 Circuitos Secuenciales

En Lava también es posible describir circuitos secuenciales de tipo *síncrono*, es decir circuitos con realimentación y *elementos de retardo* comandados por un único reloj. Al no modelarse detalles de bajo nivel, un circuito secuencial se comporta como una *FSM*, es decir su comportamiento en el ciclo de reloj $(t + 1)$ está totalmente determinado por el estado de los flip-flops (elementos de retardo) y las entradas de estos.

El bloque constructivo básico para este tipo de circuitos es el `delay`, que toma un parámetro (estado inicial del flip-flop) y una *señal temporal* que será retrasada un ciclo de clock por la memorización en el flip flop. Claramente un `delay low` se corresponde con el *FFD* cuyo estado inicial es 0. Tenemos que hacer notar que `delay` es otra de las compuertas genéricas, que igual que `mux` y `equal` pueden tomar señales de cualquier tipo, entre simples y compuestas y dentro de las simples pueden ser booleanas o enteras.

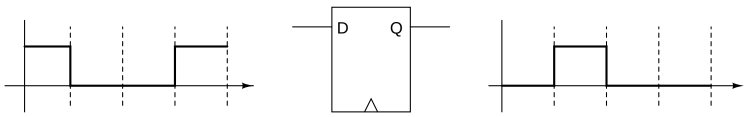

Figura 18: *FFD*

Utilizando la interpretación `simulateSeq` veamos el comportamiento del componente.

```
Prelude> :l Sequential
.
Sequential> simulateSeq (delay low) [high,low,low,high]
[low,high,low,low]
```

Vemos que la señal temporal se denota con paréntesis cuadrados, y que en la salida no se muestra el estado final del elemento de retardo.

Describamos en el archivo `segundo.hs` un *FFT* a partir del elemento de retardo.

[5]Los tiempos los deberá ver en `./Verify/circuit.mv.out` luego de cada verificación.

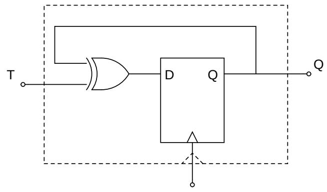

Figura 19: *FFT* a partir de un *FFD*

segundo.hs
```
ffd = delay low

fft t = out
  where
        out = ffd d
        d   = xor2 (t,out)
```

Vemos como `out` y `d` al ser mutuamente recursivos, describen la realimentación del circuito que pasa a través del retardo. Simulamos el circuito para conocer su comportamiento.

```
Prelude> :l segundo.hs
.
Main> simulateSeq fft [low,low,high,low,high,low,high,low,high,low,high,low]
[low,low,low,high,high,low,low,high,high,low,low,high]
```

Describir cualquier tipo de circuito secuencial a partir de un esquemático, no debería ser más complicado que una traducción de un circuito secuencial, como el primer ejercicio de esta sección.

4.1 Ejercicios

1. Describa el siguiente circuito secuencial y obtenga el diagrama de transición de estados a partir de la interpretación de simulación.

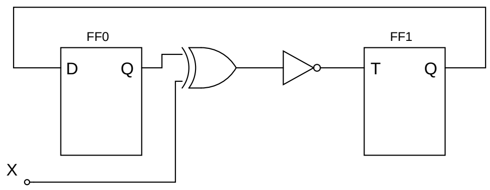

Figura 20: Un circuito secuencial

2. Esquematice el circuito.

   `vis (copy ->- (xor2 -|- (swap ->- xor2)) ->- equiv)`

   ¿Qué propiedad verifica sobre `xor2`?.

3. Implemente el *FFSR* y el *FFJK* a partir de `delay`. Simule y compruebe su correcto funcionamiento.

4. Defina el circuito `siempre` que tiene una entrada y una salida. La salida es alta mientras la entrada esté en `high`. Si la entrada cae a `low`, entonces la salida cae por siempre. Simúlelo a fin de mostrar su funcionamiento.

5. Construya un circuito sincronizador que toma dos entradas go_1 y go_2 y genera una salida go. Esta salida será 1 si ambas entradas son 1 o cuando go fue 1 y alguna de las dos entradas es 1.

5 Circuitos Genéricos Mediante Ecuaciones Recursivas

(vacio)

Referencias

[1] Per Bjesse and Koen Claessen and Mary Sheeran and Satnam Singh, *Lava: Hardware Design in Haskell*, International Conference on Functional Programming, 174–184, 1998

[2] Koen Claessen and Mary Sheeran, *A Tutorial on Lava: A Hardware Description and Verification System*, `http://www.cs.chalmers.se/~ koen/Lava/tutorial.ps`, August 2000

[3] John O'Donnell, *Hardware description with recursion equations*, Proceedings of the IFIP 8th International Symposium on Computer Hardware Description Languages and their Applications, North-Holland, 363–382, April 1987

[4] John O'Donnell, *From Transistors to Computer Architecture: Teaching Functional Circuit Specification in Hydra*, Functional Programming Languages in Education, 195–214, 1995

[5] John O'Donnell, *Introduction to Digital Circuit Specification with Hydra Lite*, Unversity of Glasgow Technical Report, May 1999

[6] Steven Johnson, *Synthesis of Digital Designs from Recursion Equations*, The ACM Distinguished Dissertation Series, The MIT Press, 1984

[7] Geraint Jones and Mary Sheeran, *Circuit Design in Ruby*, In Staunstrup, editor, Formal Methods for VLSI Design, Amsterdam, Elsevier Science Publications, 1990

[8] *VIS: A system for Verification and Synthesis*, The VIS Group, `http://www-cad.eecs.berkeley.edu/~ vis/`