1 Describing Netlists

1.1 Implementing Combinational Logic in LUTs

Structural netlists are described in Lava by composing functions that represent basic library elements or composite circuits. This is very similar to the way the structura netlists are described in VHDL or in Verilog which rely on netlist naming to compose circuit elements. Later we present combinators which provide a much more powerful-way to combine circuits.

Most hardware description languages provide a library of basic gates that correspond to the Xilinx Unified Library components. Designers can build up netlists by creating instances of these abstract gates and writing them together. The implementation software maps these gates to specific resources on the FPGA. Most logic functionality is mapped into the slices of CLBs. The top half of a Virtex-II slice is shown in Figure 1.1 on page 2.

Lava provides a more direct way to specify the mapping into CLB, slices and LUTs. Just like the Xilinx Unified Library the Xilinx Lava library contains specific functions (circuits) that correspond to resources in a Virtex slice e.g. muxcy and xorcy. However, instead of trying to enumerate every possible one, two, three and four input function that can be realized in a LUT Lava instead provides a higher order function for creating a LUT configuration from a higher level specification of the requited function.

As an example consider the task of configuring a LUT to be a two input AND gate. This can be performed by using the lut2 combinator with an argument that is the Haskell function that performs logical conjunction written as &.

```
and2 :: (Bit, Bit) -> Bit
and2 = lut2 (&)
```

The lut2 higher order combinator takes any function of two Boolean parameters and returns a circuit that corresponds to a LUT programmed to perform that function. A circuit in Lava has a type that operates over structures of the Bit type. The and2 circuit is defined to take a pair of bits as input and return a single bit as its output.

Note that lut2 is a higher order combinator because it takes a function as an argument (the & function) and returns a function as a result (the circuit which ANDs its two inputs). Indeed the lut2 combinator can take any function that has the type Bool -> Bool -> Bool and it returns a circuit of type (Bit, Bit) -> Bit. The type of the lut2 function is similar to:

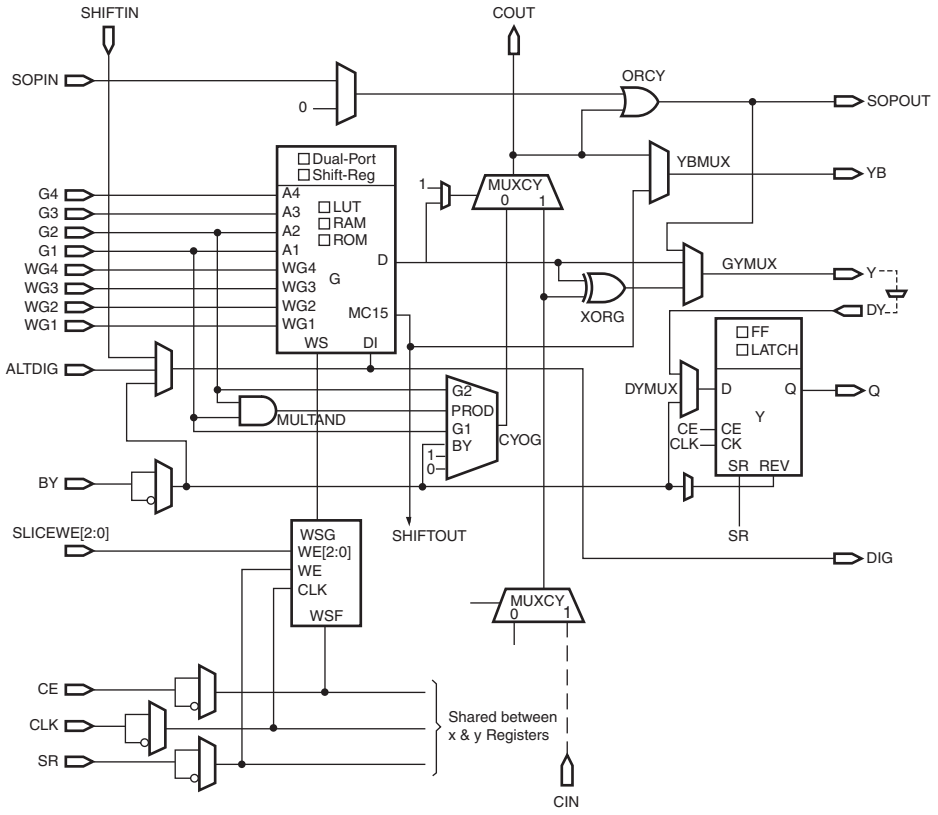
```
lut2 :: (Bool -> Bool -> Bool) -> ((Bit, Bit) -> Bit)
```

When Lava generates a VHDL or EDIF netlist for this component it will instance a LUT with the appropriate programming information. Here is what the generated VHDL for the and2 gate might look like:

```
lut2_1 : lut2 generic map (init => "1000")
              portmap (i0 => lava(5), i1 => lava (6),
                       o => lava(4)) ;
```

When realized in the top part of a slice as shown in Figure 1.1 this gate would be mapped to the upper G function generator which will be configured as a LUT.

Figure 1.1 Typical slice-based resources in a Xilinx FPGA



As another example consider the definition of a three input AND-gate. First, we define a function in Haskell that performs the AND3 operation:

```
and3function :: Bool -> Bool -> Bool -> Bool
and3function a b c = a & b & c
```

The first line gives the type of and function as something that takes three boolean values and returns a boolean value. The next line defines the `and3function` in terms of the

Haksell binary `&` function. Now we are in a position to define a circuit that ANDs three inputs:

```
and3 = lut3 and3function
```

The general idea is that instead of trying to provide a library that tries to enumerate some of the one, two, three and four input logic functions Lava provides four combinators `lut1`, `lut2`, `lut3` and `lut4`. These combinators take functions expressed in the embedding language (Haskell) and return LUTs that realize these functions. This makes it convenient to express exactly what gets packed into one LUT. Later we will see that these higher order combinators are actually overloaded which allows LUT contents to be specified in many different kinds of ways (including an integer or a bit-vector).

Lava does provide a module that contains definitions for commonly used gates mapped into LUTs. These include:

```
inv = lut1 not
and2 = lut2 (&)
or2 = lut2 (||)
xor2 = lut2 exor
muxBit sel (d0, d1)= lut3 muxFn (sel, d0, d1)
```

assuming the following functions are available in addition to the Haskell prelude:

```
exor :: Bool -> Bool -> Bool
exor a b = a /= b

muxFn :: Bool -> Bool -> Bool -> Bool
muxFn sel d0 d1
  = if sel then
      d1
    else
      d0
```

To make a netlist that corresponds to a NAND gate one could simply perform direct instantiation and wire up the signals appropriate just like in VHDL or Verilog:

```
nandGate (a, b) = d
                  where
                  d = inv c
                  c = and2 (a, b)
```

Lava generates a netlist containing two LUT instantiations for this circuit: a LUT2 to implement the AND gate and a LUT1 to implement the inverter. How many LUTs are used to realize this circuit on the FPGA? If no information is given about the location of these two circuits then the mapper is free to merge both the LUT2 for the AND gate and the LUT1 for the inverter into one LUT. However, if the inverter has been laid out in a different location from the AND gate then the two components will not be merged. By default each gate is at logical position (0.0). Since the description above does not translate the two sub-circuits to another location they are understood to occupy the same function generator location and will be merged into one LUT.

1.2 Using Slice Resources

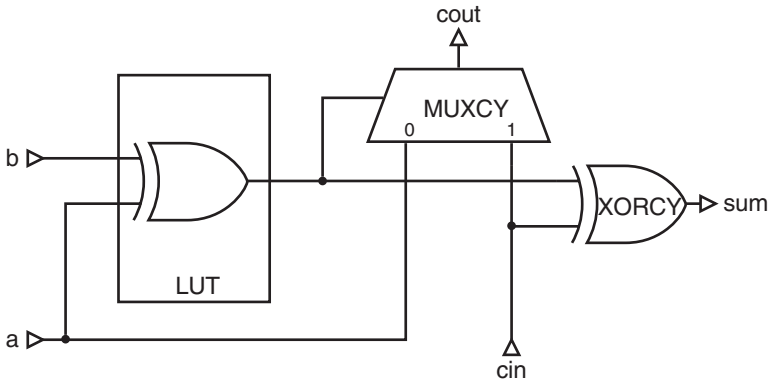
LUTs are not the only resources available in a slice. Lava provides a library of circuits which correspond directly to specific slice resources and these have names similar to the corresponding library unisim library members. A partial list of some of the basic Virtex slice resources supported by Lava is shown below:

```
gnd :: Bit
vcc :: Bit
fd :: Bit -> Bit -> Bit
fde :: Bit -> Bit -> Bit -> Bit
muxcy :: (Bit, (Bit, Bit)) -> Bit
muxcy_l :: (Bit, (Bit, Bit)) -> Bit
xorcy :: (Bit, Bit) -> Bit
xorcy_l :: (Bit, Bit) -> Bit
muxf5 :: (Bit, (Bit, Bit)) -> Bit
muxf6 :: (Bit, (Bit, Bit)) -> Bit
muxf7 :: (Bit, (Bit, Bit)) -> Bit
muxf8 :: (Bit, (Bit, Bit)) -> Bit
```

1.3 A Full Adder Design

A one-bit adder can be accommodated in one half of a slice using just one function generator as a LUT, one MUXCY and one XORCY. For this reason we will describe the one-bit adder in netlist style. A schematic for a one-bit carry chain adder is shown in Figure 1.2.

Figure 1.2 Full Adder Implemention using Fast Carry Chain Logic



This circuit can be implemented in Lava by instancing the required components and composing them using named wires. This is the same composition technique that is available in VHDL and Verilog. An implementation of a one-bit adder is given in the

file tutorials/tutorial1/Tutorial1.hs which can be found in the Lava installation directory. Copy the tutorial files to your own filespace.

The complete source text of this file is shown below.

```
module Main
where
import Lava
import Xilinx

--------------------------------------------------------------------------------

oneBitAdder :: (Bit, (Bit, Bit)) -> (Bit, Bit)
oneBitAdder (cin, (a,b))
  = (sum, cout)
    where
    part_sum = xor2 (a, b)
    sum = xorcy (part_sum, cin)
    cout = muxcy (part_sum, (a, cin))

--------------------------------------------------------------------------------

oneBitAdderCircuit :: (Bit, Bit)
oneBitAdderCircuit
  = (outputBit "sum" sum, outputBit "cout" cout)
    where
    a = inputBit "a"
    b = inputBit "b"
    cin = inputBit "cin"
    (sum, cout) = oneBitAdder (cin, (a,b))

--------------------------------------------------------------------------------

main :: IO ()
main
  = do nl <- netlist "tutorial1" oneBitAdderCircuit
       writeNetlist nl virtex2 [vhdl, edif]

--------------------------------------------------------------------------------
```

This module is intended to produce a binary when can be run to produce the VHDL and EDIF implementations of a one-bit adder. For this reason the module is named “Main” since it is the top-level module (as required by Haskell). When the program runs it evaluates the function “main” in the Main module which in this case elaborates the Lava netlist for the one-bit adder and then writes out the corresponding VHDL and EDIF.

A typical Lava program will always import two libraries. The Lava library brings into scope the basic Lava system which provides the basic types and combinators needed to compose and layout circuits. The Xilinx library brings into scope the Xilinx architecture specific library cells and implementation functions required to describe and implement circuits for Xilinx’s Virtex FPGAs.

To compile this program just type “make”:

```
$ make
lavac -c Tutorial1.hs
lavac -o tutorial1 Tutorial1.o
```

This produces a binary called tutorial1 which you can run the generate the VHDL end EDIF:

```
$ tutorial1
Wed May  1 18:59:35 PDT 2002: Elaborating netlist tutorial1...
Wed May  1 18:59:35 PDT 2002: Done.
Writing tutorial1_package.vhd ...
Writing tutorial1.vhd ...
Wed May  1 18:59:35 PDT 2002
Writing tutorial1.edn...
Wed May  1 18:59:35 PDT 2002
```

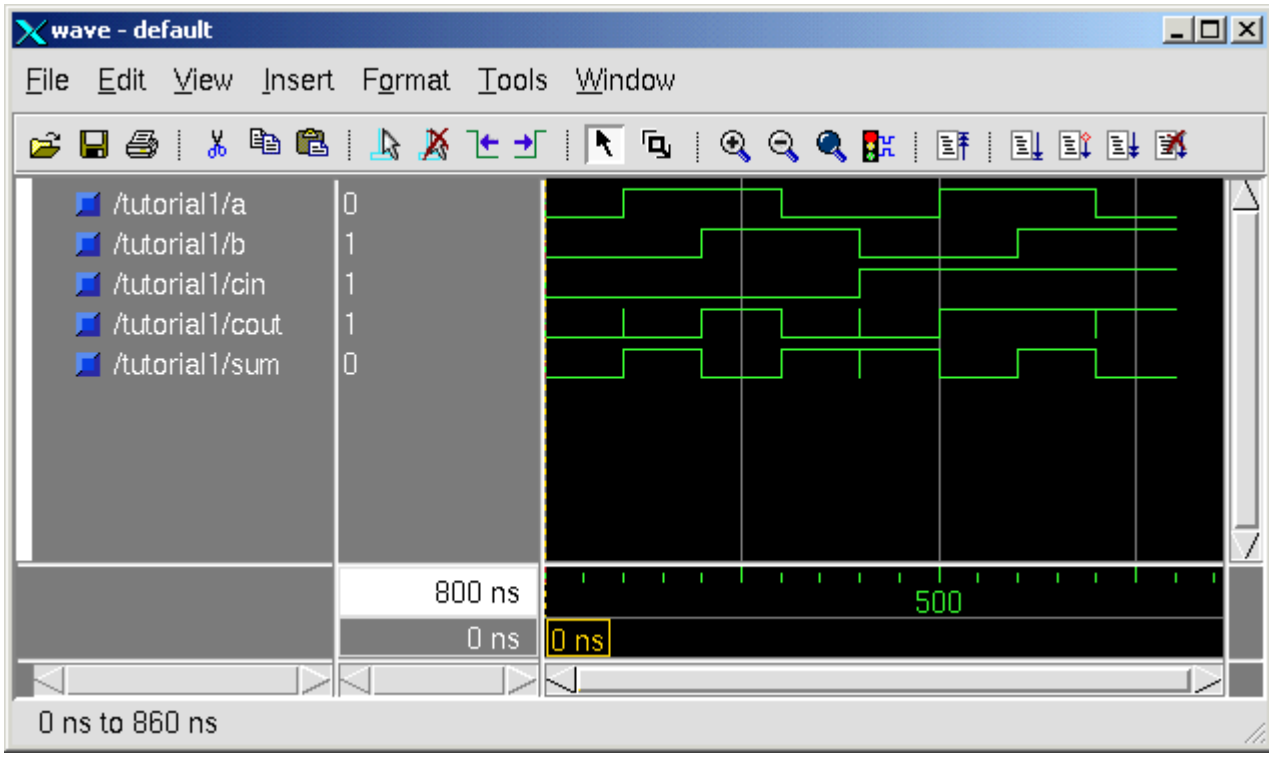
This has produced the VHDL files `tutorial1_package.vhd and tutorial1.vhd` and which can be used to simulate the one-bit adder. The `tutorial1.edn` EDIF file can be used in conjunction with the Xilinx implementation tools to implement this circuit for any Virtex-II architecture FPGA.

The generated VHDL files can be compiled using the Model Technology simulator using the “vcom” rule in the Makefile. First make sure that the modelsim.ini file contains references to properly compiled versions of the unisim library. Then use the vcom rule to perform the compilation:

```
satnam@lagavulin> make vcom
vcom tutorial1_package.vhd
Model Technology ModelSim SE vcom 5.6 Compiler 2002.03 Mar 15 2002
-- Loading package standard
-- Loading package std_logic_1164
-- Compiling package tutorial1_package
vcom tutorial1.vhd
Model Technology ModelSim SE vcom 5.6 Compiler 2002.03 Mar 15 2002
-- Loading package standard
-- Loading package std_logic_1164
-- Loading package tutorial1_package
-- Compiling entity tutorial1
-- Loading package vital_timing
-- Loading package vcomponents
-- Compiling architecture lava of tutorial1
-- Loading entity obuf
-- Loading entity xorcy
-- Loading entity lut2
-- Loading entity ibuf
-- Loading entity muxcy
```

The one-bit adder can now be simulated. An example simulation session is shown in Figure 1.3.

Figure 1.3 Simulation of the one-bit adder using Modelsim



The Makefile in the tutorial directory contains a “xflow” rule which can be used to process the generated EDIF file with the Xilinx build, map, place, route and bitstream generation toos. You can edit the Makefile to change the target device. The result is deposited in the implementation subdirectory and the implementation flow is governed by the files in the etc subdirectory. Here is an example execution of the xflow rule:

```
satnam@lagavulin> make xflow
mkdir implementation
cp tutorial1.edn implementation
cp etc/fast_runtime.opt implementation
cp etc/bitgen.ut implementation
cp etc/bitgen.opt implementation
echo > implementation/tutorial1.ucf
xflow -wd implementation -p xc2v1000ff896-6 -implement fast_runtime.opt -config
bitgen.opt tutorial1.edn
Release 4.2i - Xflow E.38
Copyright (c) 1995-2001 Xilinx, Inc.  All rights reserved.
xflow -wd implementation -p xc2v1000ff896-6 -implement fast_runtime.opt -config
bitgen.opt tutorial1.edn
.... Copying flowfile /build/sjxfndry/E.38/rtf/xilinx/data/fpga.flw into working
directory /home/satnam/lava/tutorials/tutorial1/implementation

Using Flow File: /home/satnam/lava/tutorials/tutorial1/implementation/fpga.flw
Using Option File(s):
 /home/satnam/lava/tutorials/tutorial1/implementation/fast_runtime.opt
 /home/satnam/lava/tutorials/tutorial1/implementation/bitgen.opt

Creating Script File xflow.scr...

#------------------------------------------------#
```

```
# Starting program ngdbuild
# ngdbuild -p xc2v1000ff896-6 -nt timestamp -uc tutorial1.ucf
/home/satnam/lava/tutorials/tutorial1/implementation/tutorial1.edn tutorial1.ngd
#--------------------------------------------#
Release 4.2i - ngdbuild E.38
Copyright (c) 1995-2001 Xilinx, Inc.  All rights reserved.

Command Line: ngdbuild -p xc2v1000ff896-6 -nt timestamp -uc tutorial1.ucf
/home/satnam/lava/tutorials/tutorial1/implementation/tutorial1.edn tutorial1.ngd

Launcher: Executing edif2ngd "tutorial1.edn" "tutorial1.ngo"
INFO:NgdBuild - Release 4.2i - edif2ngd E.38
INFO:NgdBuild - Copyright (c) 1995-2001 Xilinx, Inc.  All rights reserved.
Writing the design to "tutorial1.ngo"...
Reading NGO file
"/home/satnam/lava/tutorials/tutorial1/implementation/tutorial1.ngo" ...
Reading component libraries for design expansion...

Annotating constraints to design from file "tutorial1.ucf" ...

Checking timing specifications ...
Checking expanded design ...

NGDBUILD Design Results Summary:
  Number of errors:     0
  Number of warnings:   0

Writing NGD file "tutorial1.ngd" ...

Writing NGDBUILD log file "tutorial1.bld"...

NGDBUILD done.



#--------------------------------------------#
# Starting program map
# map -o tutorial1_map.ncd tutorial1.ngd tutorial1.pcf
#--------------------------------------------#
Release 4.2i - Map E.38
Copyright (c) 1995-2001 Xilinx, Inc.  All rights reserved.
Using target part "2v1000ff896-6".
Removing unused or disabled logic...
Running cover...
Writing file tutorial1_map.ngm...
Running directed packing...
Running delay-based packing...
Running related packing...
Writing design file "tutorial1_map.ncd"...

Design Summary:
   Number of errors:      0
   Number of warnings:    0
   Number of Slices:                      1 out of   5,120    1%
   Number of Slices containing
      unrelated logic:                    0 out of       1    0%
   Number of 4 input LUTs:                1 out of  10,240    1%
   Number of bonded IOBs:                 5 out of     432    1%
Total equivalent gate count for design:  12
Additional JTAG gate count for IOBs:  240

Mapping completed.
See MAP report file "tutorial1_map.mrp" for details.



#--------------------------------------------#
# Starting program par
# par -w -ol 2 -d 0 tutorial1_map.ncd tutorial1.ncd tutorial1.pcf
#--------------------------------------------#
Release 4.2i - Par E.38
Copyright (c) 1995-2001 Xilinx, Inc.  All rights reserved.
```

```
Constraints file: tutorial1.pcf

Loading design for application par from file tutorial1_map.ncd.
   "tutorial1" is an NCD, version 2.37, device xc2v1000, package ff896, speed -6
Loading device for application par from file '2v1000.nph' in environment
/build/sjxfndry/E.38/rtf.
The STEPPING level for this design is 0.
Device speed data version:  ADVANCED 1.105 2002-05-09.


Resolving physical constraints.
Finished resolving physical constraints.

Device utilization summary:

   Number of External IOBs             5 out of 432     1%
      Number of LOCed External IOBs    0 out of 5       0%

   Number of SLICEs                    1 out of 5120    1%



Overall effort level (-ol):   2 (set by user)
Placer effort level (-pl):    2 (set by user)
Placer cost table entry (-t): 1
Router effort level (-rl):    2 (set by user)
Extra effort level (-xe):     0 (default)

Starting Clock Logic Placement.  REAL time: 9 secs

Finished Clock Logic Placement.  REAL time: 9 secs

Automatic resolution of clock placement was successful.
It was not necessary to constrain the placement of any of the logic driven by
the global clocks with the current clock placement.

########################################################
## Automatic clock placement completed.
########################################################

Starting clustering phase.  REAL time: 10 secs
Finished clustering phase.  REAL time: 10 secs

Dumping design to file tutorial1.ncd.

Starting Directed Placer.  REAL time: 10 secs
Placement pass 1 .

Placer score = 270
Placer score = 270
Finished Directed Placer.  REAL time: 10 secs

Starting Optimizing Placer.  REAL time: 10 secs
Optimizing
Swapped 1 comps.
Xilinx Placer [1]   270   REAL time: 10 secs
Finished Optimizing Placer.  REAL time: 10 secs

Dumping design to file tutorial1.ncd.

Total REAL time to Placer completion: 10 secs
Total CPU time to Placer completion: 10 secs

0 connection(s) routed; 5 unrouted.
Starting router resource preassignment
Completed router resource preassignment. REAL time: 12 secs
Starting iterative routing.
Routing active signals.
.
End of iteration 1
5 successful; 0 unrouted; (0) REAL time: 14 secs
```

```
Constraints are met.
Total REAL time: 14 secs
Total CPU  time: 13 secs
End of route.  5 routed (100.00%); 0 unrouted.
No errors found.
Completely routed.

This design was run without timing constraints.  It is likely that much better
circuit performance can be obtained by trying either or both of the following:

  - Enabling the Delay Based Cleanup router pass, if not already enabled
  - Supplying timing constraints in the input design


Total REAL time to Router completion: 14 secs
Total CPU time to Router completion: 13 secs

Generating PAR statistics.
Dumping design to file tutorial1.ncd.


All signals are completely routed.

Total REAL time to PAR completion: 15 secs
Total CPU time to PAR completion: 14 secs

Placement: Completed - No errors found.
Routing: Completed - No errors found.

PAR done.



#------------------------------------------------#
# Starting program post_par_trce
# trce -e 3 -xml tutorial1.twx tutorial1.ncd tutorial1.pcf
#------------------------------------------------#
Release 4.2i - Trace E.38
Copyright (c) 1995-2001 Xilinx, Inc.  All rights reserved.



Loading design for application trce from file tutorial1.ncd.
   "tutorial1" is an NCD, version 2.37, device xc2v1000, package ff896, speed -6
Loading device for application trce from file '2v1000.nph' in environment
/build/sjxfndry/E.38/rtf.
The STEPPING level for this design is 0.
--------------------------------------------------------------------------------
Release 4.2i - Trace E.38
Copyright (c) 1995-2001 Xilinx, Inc.  All rights reserved.

trce -e 3 -xml tutorial1.twx tutorial1.ncd tutorial1.pcf

Design file:              tutorial1.ncd
Physical constraint file: tutorial1.pcf
Device,speed:             xc2v1000,-6 (ADVANCED 1.105 2002-05-09)
Report level:             error report
--------------------------------------------------------------------------------

WARNING:Timing:2491 - No timing constraints found, doing default enumeration.


Timing summary:
---------------

Timing errors: 0  Score: 0

Constraints cover 6 paths, 5 nets, and 5 connections (100.0% coverage)

Design statistics:
   Maximum combinational path delay:   7.919ns
   Maximum net delay:   0.767ns
```

```
Analysis completed Fri Jun  7 15:39:04 2002
--------------------------------------------------------------------------------

Generating Report ...

Total time: 8 secs



#------------------------------------------#
# Starting program bitgen
# bitgen -w -f bitgen.ut tutorial1.ncd
#------------------------------------------#
Release 4.2i - Bitgen E.38
Copyright (c) 1995-2001 Xilinx, Inc.  All rights reserved.

Loading design for application Bitgen from file tutorial1.ncd.
   "tutorial1" is an NCD, version 2.37, device xc2v1000, package ff896, speed -6
Loading device for application Bitgen from file '2v1000.nph' in environment
/build/sjxfndry/E.38/rtf.
The STEPPING level for this design is 0.
Opened constraints file tutorial1.pcf.

Fri Jun  7 15:39:11 2002

Running DRC.
DRC detected 0 errors and 0 warnings.
Creating bit map...
Saving bit stream in "tutorial1.bit".
Bitstream generation is complete.


xflow done!
85.58s real 68.38s user 4.46s system 85% make xflow
satnam@lagavulin>
```

The Xilinx Floorplanner and the FPGA Editor applications can be used to examine the implemented circiut in the implementation subdirectory. The floorplanner view of the one-bit adder is shown in Figure 1.4. The slice used to implement the one-bit adder is shown in Figure 1.5.

Figure 1.4 Floorplanner view of the one-bit adder

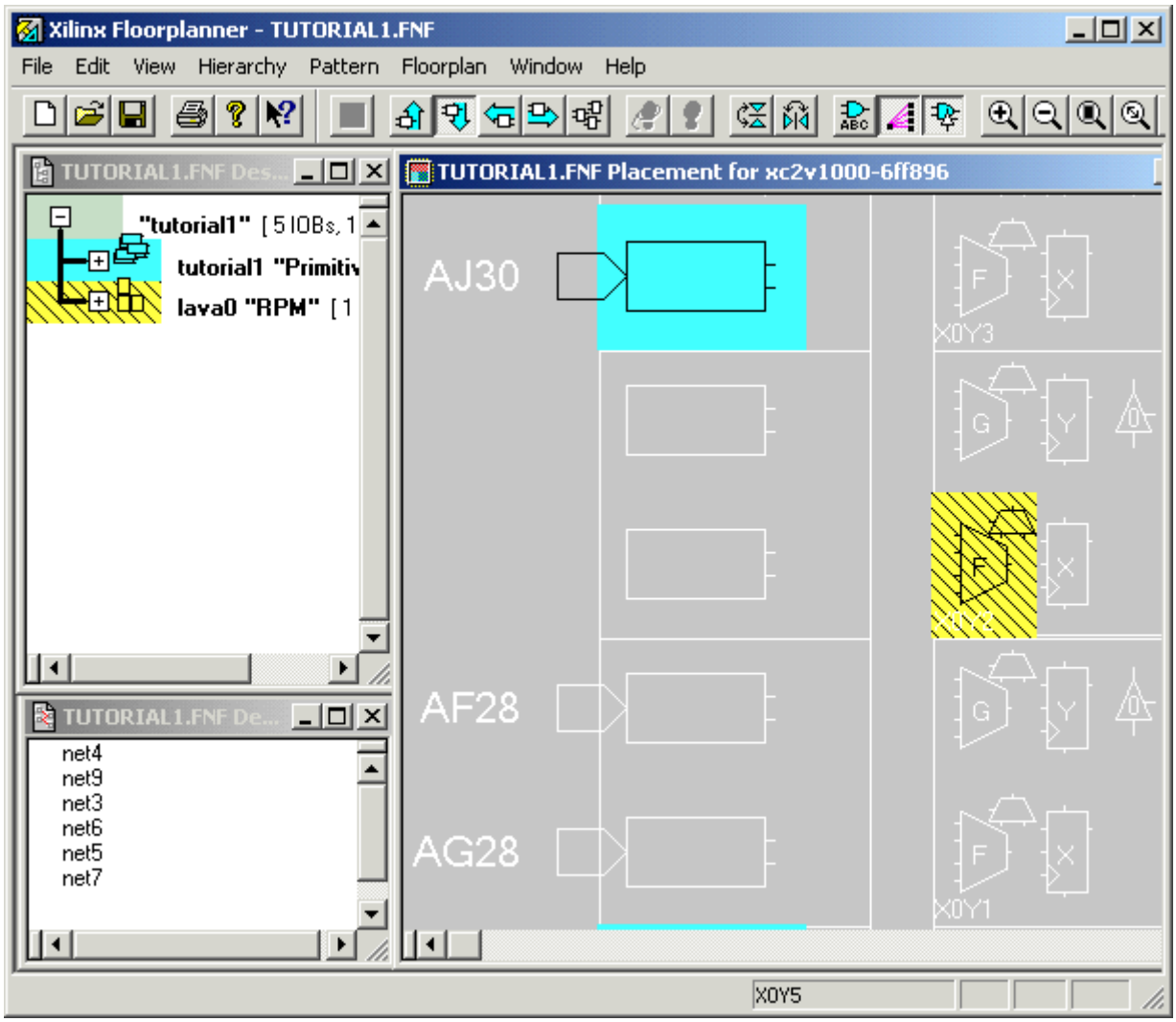


Figure 1.5 FPGA Editor view of the one-bit adder slice