2 Combinators and Placement

Tutorial 1 showed how a circuit can be composed using named wires. We show here how circuits can be composed using combinators. Combinators are functions that allow circuit blocks to be glued together without the need for the connecting wires to be explicitly named. In addition to this, combinators can be used to provide information about where circuit blocks are placed in the resulting implementation. This tutorial provides the steps used in composing and placing a 6-input lookup table (LUT) design using combinators. A schematic for this circuit is shown in Figure 2.1.

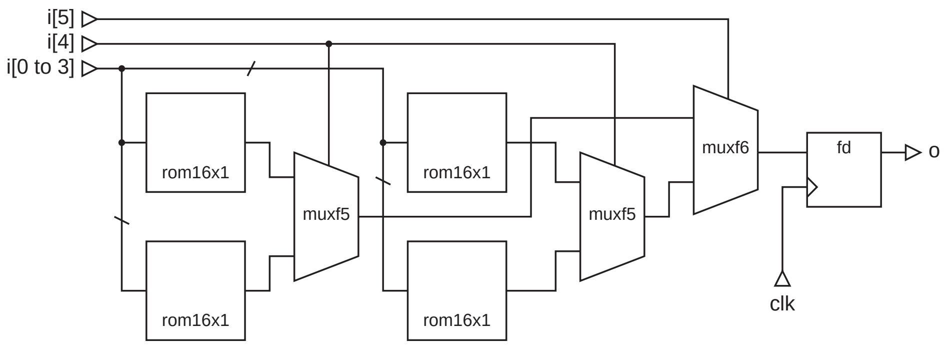

Figure 2.1 Registered Lut6 implementation using multiplexers and 4-input ROMs.

The components in this circuit are particular to the Virtex architecture. The function generators of the CLB are configured as rom16x1 components. They have the same function as LUTs, but are not optimised automatically by the Xilinx implementation tools in terms of their contents. This is advantageous in the context of this tutorial because it allows us to observe the expected layout without any optimisations occuring. The muxf5 and muxf6 are implemented as dedicated logic in the CLB. They provide better performance and resource utilisation when combining the outputs from the function generators.

2.1 Composition Without Placement

The two basic ways to compose circuit blocks in Lava are to connect them in series and to connect them in parallel. Lava provides the `>==>` combinator for series composition and the `par` combinator for parallel composition. Series composition connects the output of one circuit block to the input of another. To compose the final multiplexer (muxf6) in series with the flip-flop (fd), the following is written in Lava:

```
muxAndFd clk = muxf6 >==> fd clk
```

We can see the type of the function `muxAndFd` using `lavai`:

```
*Main> :i muxAndFd
-- muxAndFd is a variable, defined at Lut6.hs:35
muxAndFd :: Bit -> (Bit, (Bit, Bit)) -> Bit
```

The first `Bit` is the clock (`clk`) wire, `(Bit, (Bit, Bit))` are the inputs to the multiplexer and the final `Bit` is the output of the flip-flop. The series composition will only work if the types of the first blocks outputs and the second blocks inputs are the same. In the case of the flip-flop, the clock wire is not required in the series composition and so is provided as an explicitly name wire. In general, the wires that remain after partially applying the arguments of the circuit block function, are the wires used in the series composition.

Parallel composition using `par` combines the input and output types in a pair. To compose the two `rom16x1` blocks in parallel, the following is written in Lava:

```
romAndRom init0 init1 = rom16x1 init0 `par2` rom16x1 init1
```

The type of the composition is:

```
Int -> Int ->
((Bit, Bit, Bit, Bit), (Bit, Bit, Bit, Bit)) -> (Bit, Bit)
```

The two `Int` arguments are the initialisation values of the ROMs. Each rom takes four wires and its type is a four element tuple. Therefore the input type of the parallel composition is a pair of four element tuples. The output of each ROM is a single wire so the output type of the composition is a pair.

The parallel ROMs can now be composed in series with the multiplexer (muxf5). The select input of the multiplexer is not wired up during the series composition. The input type of the multiplexer is a pair with another pair as the second element. We cannot use partial application to combine the correct wires this time. So for this reason, the prelude provides the combinators `fsT` and `snD` to help navigate through pairs when doing series composition. `fsT` combines a block in parallel with a wire, and `snD` combines a wire in parallel with a block. We want to connect the outputs of the ROMs to the second element of the multiplexer input pair. The connecting outputs and inputs are pairs themselves. This composition is written in Lava as follows:

```
romsAndMux init0 init1 = snD(romAndRom init0 init1) >==> muxf5
```

We can now compose the entire circuit by applying the same technique:

```
snD(romsAndMux init0 init1 `par2` romsAndMux init2 init3)
>==> muxAndFd clk
```

This composition is shown in Figure 2.2.

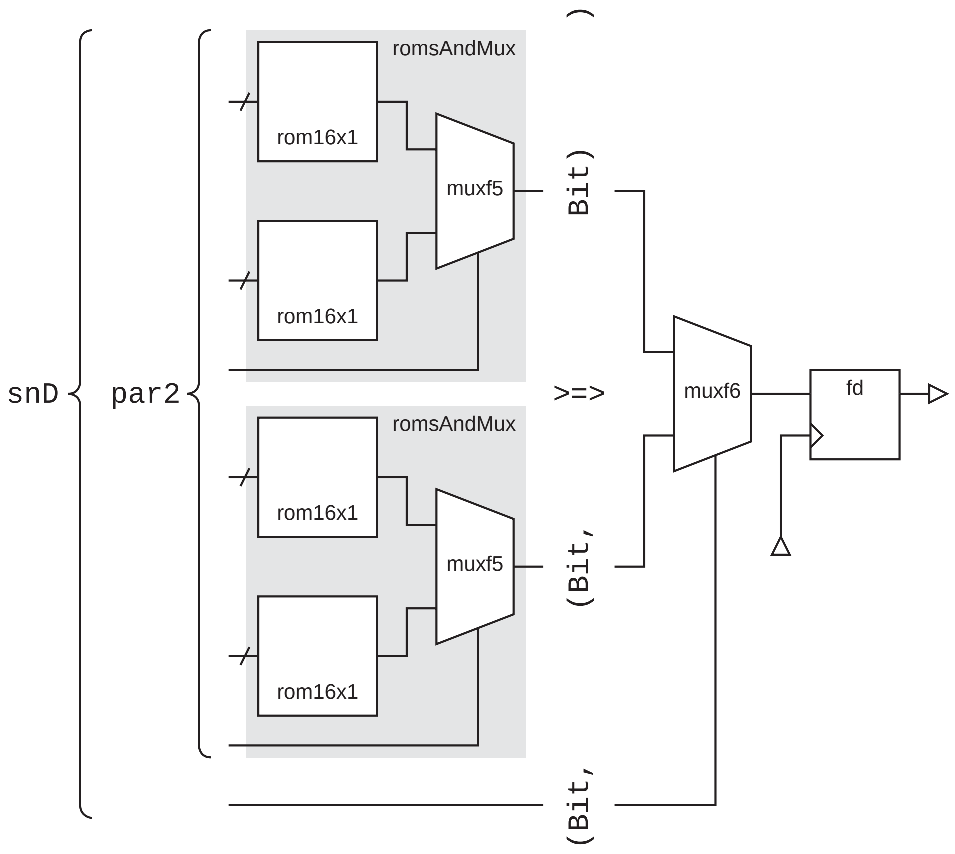

Figure 2.2 Composition of the Lut6 design using series and parallel combinators.

All that remains now is to connect the ROMs together and tidy up the inputs. The above expression has the type:

```
Int -> Int -> Int -> Int -> Bit ->
(Bit, ((Bit, ((Bit, Bit, Bit, Bit), (Bit, Bit, Bit, Bit))),
        (Bit, ((Bit, Bit, Bit, Bit), (Bit, Bit, Bit, Bit)))))
               -> Bit
```

The simplest way to handle all the wires is to creating a custom wiring function and compose it in series with the expression. A suitable wiring function is:

```
wires (a,b,c,d,e,f) =
(f, ((e, ((a,b,c,d), (a,b,c,d))), (e, ((a,b,c,d), (a,b,c,d)))))
```

Although this means reverting back to composition by named wires at the last hurdle, the effort of glueing the blocks together has been reduced. Alternatively, it is possible to define new combinators specific to a problem, or to define and compose wiring blocks. This can help to reduce the amount of wiring and the number of combinators needed to describe a circuit. The complete source text of this circuit is given below.

```
-- function types

muxAndFd :: Bit -> (Bit, (Bit, Bit)) -> Bit
romAndRom :: Int -> Int -> ((Bit, Bit, Bit, Bit), (Bit, Bit, Bit, Bit)) -> (Bit, Bit)
romsAndMux :: Int -> Int -> (Bit, ((Bit, Bit, Bit, Bit), (Bit, Bit, Bit, Bit))) -> Bit
complete :: Int -> Int -> Int -> Int -> Bit
            -> (Bit, ((Bit, ((Bit, Bit, Bit, Bit), (Bit, Bit, Bit, Bit))),
                      (Bit, ((Bit, Bit, Bit, Bit), (Bit, Bit, Bit, Bit))))) -> Bit
wired :: (Int, Int, Int, Int) -> Bit -> (Bit, Bit, Bit, Bit, Bit, Bit) -> Bit

-- functions

muxAndFd clk = muxf6 >==> fd clk
romAndRom init0 init1 = rom16x1 init0 `par2` rom16x1 init1
romsAndMux init0 init1 = snD(romAndRom init0 init1) >==> muxf5

complete init0 init1 init2 init3 clk =
    snD(romsAndMux init0 init1 `par2` romsAndMux init2 init3) >==> muxAndFd clk

wired (init0, init1, init2, init3) clk =
    wires >==> complete init0 init1 init2 init3 clk
    where wires (a,b,c,d,e,f) =
               (f, ((e, ((a,b,c,d), (a,b,c,d))), (e, ((a,b,c,d), (a,b,c,d)))))
```

Note: it might seem like a lot of effort to provide all the types of the functions, in practice this can mostly be done automatically using `lavai`.

2.2 Composition with Placement

Placement is the process of locating the circuit blocks in the final implementation. In the previous example, placement information was not supplied in the Lava source code. In order to implement this design, placement must be provided by an external tool. In the case of Xilinx FPGAs, the tools used to do this would most likely be the Xilinx implementation tools. These provide facilities for both manual and automatic placement of circuit blocks. By manually specifying placement, it is sometimes possible to achieve an implementation that has a higher performance, or more efficient resource utilisation, than an automatically placed implementation.

For this reason, Lava has combinators that can be used to specify the relative location of the circuit blocks being composed. The placement information is tagged onto each circuit block adding an RLOC (relative location) constraint in the output netlist. This gives the relative location of the circuit block in the grid of the FPGA configurable logic blocks (CLBs).

Figure 2.3 shows a possible mapping of the 6-input LUT circuit blocks to a Virtex CLB. The CLB contains two slices, each of which contain a set of programmable logic blocks. These include two LUTs (which the `rom16x1` maps to), a `muxf5`, a `muxf6` and a register. The Virtex RLOC constraint specifies the CLB row, column and slice to place a circuit block. To place the circuit blocks that map to the LUTs, they are RLOCed to the same slice and the Xilinx implementation tools allocate them to the physical resources.

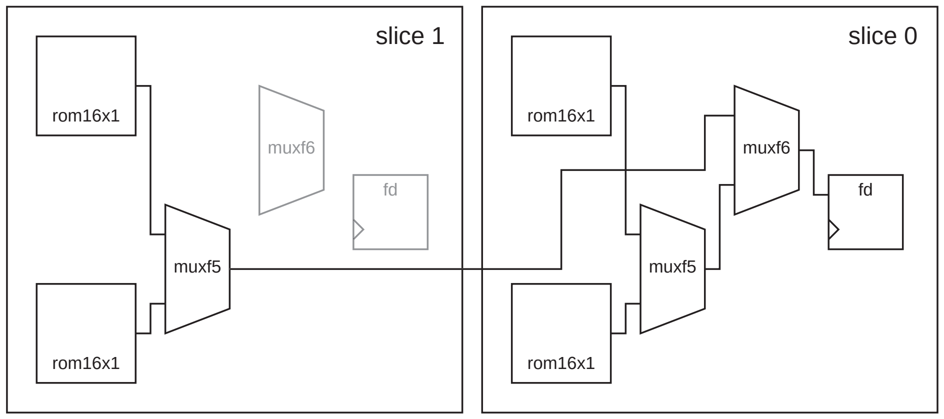

Figure 2.3 Layout of the Lut6 design within a Virtex CLB. The external connections are not shown.

Lava provides combinators that place circuit blocks beside each other, below each other or in the same location. The ability to compose blocks in the same location allows blocks to be mapped to the different resources within a single slice.

To compose the 6-input LUT with the above placement, we start again by composing the final multiplexer (`muxf6`) is series with the flip-flip (`fd`). To specify that they are placed in the same slice, we use the series combinator `>|>`. In a similar way to before, this is written in Lava as:

```
muxAndFdPl clk = muxf6 >|> fd clk
```

The rom16x1 blocks can be composed in each slice using the `vpar2` combinator. The `vpar2` combinator takes two circuit blocks and places the first below the second. Lava treats the two LUTs in a CLB as seperate locations, although the resulting RLOCs for two LUT circuit blocks composed vertically will be the same. The facility to compose LUTs in this way is provided so that circuits that use columns of LUTs, such as arithmetic, can be described easily. The Xilinx implementation tools actually decide how the circuit blocks will be assigned to the LUTs since the RLOC contraints do not provide this information. For this composition, `par2` is replaced with `vpar2`:

```
romAndRomPl init0 init1 = rom16x1 init0 `vpar2` rom16x1 init1
```

It is straightforward now to compose the `muxf5` block with the `rom16x1` blocks in each slice:

```
romsAndMuxPl init0 init1 = snD(romAndRomPl init0 init1) >|> muxf5
```

To compose the `muxAndFdPl` and `romAndRomPl` blocks together is more tricky. This is because the blocks in slice 0 must be composed to share the same location. If they are composed as before with `romAndRomPl` blocks in parallel and the `muxAndFdPl` block composed using >|>, then the muxf6 and register would be placed in slice 1. This is because >|> places blocks in the same location with their origins aligned in the lower left. To do this we need to compose the block in slice 0 in series using >|> and then compose these in series horiztonally with the slice 1 blocks. We can the following rule to break the parallel composition of the `rom16x1` blocks:

```
A hpar2 B = fst A >-> snd B
```

If `A = B = romsAndMuxPl,` and `C = muxAndFdPl`, then the form of the composition we are looking for is (ignoring initial values and the clock):

```
snD(A hpar2 B) >|> C
```

Substituting for the LHS of the above rule:

```
snD(fsT A >-> snD B) >|> C
```

Distributing the `snD` and bracketing the terms we want to place together:

```
snD(fsT A) >-> (snD(snD B) >|> C)
```

Which gives the Lava description for the placed circuit:

```
romsAndMuxesPl init0 init1 init2 init3 clk
= snD(fsT(romsAndMuxPl init0 init1)) >->
  (snD(snD(romsAndMuxPl init2 init3)) >|> muxAndFdPl clk)
```

If the 6-input LUT is described without the placement combinators, the Xilinx implementation tools will come up with a similar placement themselves (probably with the slices reversed, which makes no difference in terms of the Virtex architecture). So why go to the trouble of placing such a small circuit? The advantage is, that this can now be used to build bigger, fully placed circuits. This composition might seem complicated for the size of the circuit, but this is because it is irregular and is complicated by the need for placing circuit blocks on top of each other. Lavas real strength lies in the ease in which regular circuits can be composed and placed. The facilities that Lava provides for doing this are introduced in the next tutuorial.

2.3 Exercise

Provide a Lava description of the 6-input LUT circuit placed using combinators for the Virtex II architecture.