EEE4120F Class Activity - Lecture 18

Consider that you are needing to manually place and route a bitfile for the piece of an FPGA, specifically the single Programable Logic Block (PLB) in the FPGA shown in Appendix B. Each Programmable Logic Element (PLB) in the PLB is a 3-in/1-out LUT, and these are connected via multiplexes. There are 4-1 MUXes at the inputs and 2-1 MUXes deeper in the PLB.

The 'banged' variables (i.e. the ones with an exclamation mark, such as M1.1!) indicates a programmable variable, actually registers (i.e., flipflops), that are assigned when the bitfile is loaded into the PLB.

The bitfile, used to program this PLB of the FPGA, comprises the following bitstream sequence:

{ M1.1, M1.2, M1.3, M1.4, L1.x, L2.x, M2.1, M2.2, M2.3, M2.4, L3.x, L4.x, M3.1, M3.2 }

Note that:

- M1.1 to M1.4 are two-bit values. (e.g. The value of M1.1 is a two-bit setting to select one of the inputs A,B,C or D)
- M2.1 – M3.1 are all single bit values.
- L1.x! is an 8-bit sequence of bits, as is L2.x – L4.x. (e.g. if it is set to 11111111_2 then the output of L1.x will essentially be the constant value 1_2. Not the most efficient way of storing a constant value in an FPGA, but based on this rather limited design that's how it would likely be done, if there are no e.g. memory elements or flipflops in the PLB that can be programmed with a value.
- The inputs to the PLB is A – F. But notice that E and F are not connected up to the PLB in the same way that A-D is. This can be quite typical of an FPGA and PLB design. What we are trying to demonstrate here is that sometimes the input to a PLB is brought directly into the PLB at a deeper point, rather than being connected to multiplexers at the outer layers. Why? Because it can make for more efficient designs, if you need, e.g. an input E that is used in deciding an answer formulated by A and B, you don't necessarily want to waste a chain of multiplexers and LUTs bringing the E input deeper into the PLB where the condition a condition for what to do with the calculated result is made. (You might notice this is illustrated in the IF operation in the example program below).

Hopefully it's pretty clear how the programming is done: the bitstream (illustrated by the thin blue dotted lines) is basically fed sequentially into the FPGA, loaded into the various registers (the 'bangled variables') and when that is done the FPGA is ready to start running.

Now that you hopefully understand what is happening here, let us consider the following simple Verilog HDL program that we want to programme into this PLB of the FPGA. The code program is as follows:

```
// FPGA module to do a simple logic operation
module decider (A,B,C,D,E,F,X,Y);
 // define inputs
 input  A,B,C,D,E,F;
 // define outputs
 output reg X,Y;
 // define intermediate results
 reg j,k;
 // do the following on any change of input
 always @(*)
  begin
   // implement operation
   j <= A|B|~E;
   k <= (A&C)|E;
   // Assign outputs
   if (F) X = j; else X = ~j;
   Y <= F|k;
  end
endmodule
```

Listing 1: code listing for 'decider' module.

The main thing to look at is the code between the begin and end in the always@ block. This is where the operations we want to implement on the FPGA are implemented.

Just in case you've forgotten, the '|' is a bitwise OR, the '&' is a bitwise AND and '~' is bitwise NOT in Verilog.

So we want to decide on the bitfile to send to the PLB to implement:

```
j <= A|B|~E;
k <= (A&C)|E;
if (F) X = j; else X = ~j;
Y <= F|k;
```

You can assume that A,B,C, E and F are all single bit inputs that are fed into the PLB (i.e. in the section marked 'external inputs'. The A,B,C and D on the left of the diagram are all the same, i.e. it is the same A input feeding in to multiplexers M1.1 – M1.4. The LUTs operate, as expected, where an input e.g. of {a=0, b=0, c=0} is mapped to the first bit of the x! 8-bit bit vector x![0]. Similarly {a=0, b=0, c=1} is mapped to x![1] and so on.

Appendix A shows the above HDL program for the decider module, together with a testbench for it and the output of the testbench. This gives a suggestion of what this tiny design does.

Answer the following:

You can print out Appendix B to write on.

(a) Work out the truth table for A|B|~E. The output of this truthtable (i.e. the last column) is the bit sequence that is to be programmed into L1.x.
(b) What bit sequence do you need to programme into M1.1, M1.2? You need to decide the settings for M1.1 and M1.2, such that external inputs A and B are passed through to LUT-1. Note that E is already an input that is feeding into LUT-1, i.e. it is hardcoded that L1.c =E. (e.g. If you set M1.1 to 00 then that is passing external input 'A' through to input 'a' of LUT-1).
(c) Now complete the programming for LUT-2, i.e. the bit vector to assign to L2.x, and the values to assign to M1.3 and M1.4. Assuming that LUT-2 is going to implement the operation (A&C)|E.
(d) Not implement bit sequence for M2.1 to M2.4, and for L3.x and L4.x that will produce the results that will be assigned to the X and Y outputs.
(e) Finally indicate the bit sequences that will be used to assign M3.1 and M3.2 so that X and Y is assigned to the correct outputs.
(f) Last, and no doubt the final expectation, is what is the final bit sequence that you will feed into this PLB in order to configure it?

NOTE: In an test or exam you'd also be asked to hand in the detached Appendix B (or for distance learning asked to scan that page and upload it) so that you answer is clearly recorded in case your text-based answer is unclear.

Appendix A : Code for decider module and its testbench

```
// FPGA module for simple operations
module decider (A,B,C,D,E,F,X,Y);
 // define inputs
 input  A,B,C,D,E,F;
 // define outputs
 output reg X,Y;
 // define intermediate results
 reg j,k;
 // do on any change of input:
 always @(*)
  begin
   // implement operation
   j <= A|B|~E;
   k <= (A&C)|E;
   // Assign outputs
   if (F) X = j; else X = ~j;
   Y <= F|k;
  end
endmodule
```

```
// Testbench for the decider module
module decider_tb ();
  // instantiate regs for test vectors
  reg A,B,C,D,E,F,X,Y;
  // instantiate unit under test
  decider uut (A,B,C,D,E,F,X,Y);

  initial begin
   // define pins to monitor
   $monitor("t=%3d A=%d B=%d C=%d E=%d
             F=%d -> X=%d Y=%d",
            $time,A,B,C,E,F,X,Y);

   // set initial values
   A<=0; B<=0; C<=0; D<=0; E<=0; F<=0;
   // exercise values
   #5
   A <= 1;
   #5
   B <= 1;
   #5
   C <= 1;
   #5
   E <= 1;
   #5
   F <= 1;
  end
 endmodule
```

Output of running the code in iVerilog / EDA Playground

```
# compile the code in iVerilog:
iverilog -g2012 -o decider decider.v decider_tb.v

# run the code in iVerilog:
vvp decider

console output

t=  0 A=0 B=0 C=0 E=0 F=0 -> X=0 Y=0
t=  5 A=1 B=0 C=0 E=0 F=0 -> X=0 Y=0
t= 10 A=1 B=1 C=0 E=0 F=0 -> X=0 Y=0
t= 15 A=1 B=1 C=1 E=0 F=0 -> X=0 Y=1
t= 20 A=1 B=1 C=1 E=1 F=0 -> X=0 Y=1
t= 25 A=1 B=1 C=1 E=1 F=1 -> X=1 Y=1
```

Code available to play with on EDA Playground at: https://www.edaplayground.com/x/3CYd

APPENDIX B : Programmable Logic Block Structure