Xilinx Implementation Tutorial

Michael Bales
School of Electrical and Computer Engineering
Georgia Institute of Technology

The purpose of this tutorial is to provide exposure to commonly implemented components in the Xilinx ISE environment. Components we'll look at include global clock buffers, tri-state buffers, and block RAM (BRAM). A few notes on using timing constraints are also provided.

Clock Buffers

Clock buffers are used to increase the fanout capacity of a clock signal, and to minimize skew. They also help the synthesis tools identify which signals are global clocks in complex designs. They are most easily instantiated directly as components.

```
-- From Xilinx's Libraries Guide "lib.pdf"
-- Component Declaration for BUFG should be placed
-- after architecture statement but before begin keyword
component BUFG
       port (O : out STD_ULOGIC;
             I : in STD_ULOGIC);
end component;

-- Component Instantiation for BUFG should be placed
-- in architecture after the begin keyword
       BUFG_INSTANCE_NAME : BUFG
       port map (O => user_O,
                 I => user_I);
```

Tri-state Buffers

Frequently in a design (particularly in designs that manipulate large amounts of data), several computing modules may need access to the same data at different times. For instance, you may have an image processor. Each pixel of the image is stored in RAM. The processor has to fetch each pixel, perform some operation on it, and store the new value in the same RAM location. Then, perhaps you want the data compressed, so a compressor grabs the post-processed data from RAM, encodes it, and outputs the final stream serially. The image processor and the compressor need to access the same RAM, but not at exactly the same time. Each needs independent control of the address and control lines.

Tri-state buffers allow modules to control the same signals at different times without interfering with each other. If each module were connected to the signals directly, those signals would have two (or more) drivers, and that would cause problems.

Here is the VHDL code for a generic tri-state buffer:

```
entity gen_tribuf is
generic(width : positive);
Port ( pass : in  STD_LOGIC;
i : in  STD_LOGIC_VECTOR (width-1 downto 0);
o : out  STD_LOGIC_VECTOR (width-1 downto 0));
end gen_tribuf;

architecture Behavioral of gen_tribuf is
constant hi_imp : STD_LOGIC_VECTOR(width-1 downto 0) := (others => 'Z');
begin
o <= i when pass ='0' else hi_imp;
end Behavioral;
```

By instantiating these between each module and the shared signals, a state machine or other control logic can manage the 'pass' signal so that only the correct module controls the shared signals at any time.

Block RAM (BRAM)

Many modern FPGAs have two types of memory elements available. Distributed RAM takes the form of flip-flops that are created within the general-purpose logic fabric of the chip. This is good for storing small amounts of data, making registers, shift registers, etc. Block RAM is dedicated, configurable memory with address, data, and control ports. If your design stores and manipulates a lot of data, Block RAM (BRAM) is the way to go. Otherwise, the synthesizer may try to generate enough storage out of the FPGA's logic slices, which could cause you to run out of room for the logic portion of your design.

The BRAM in Xilinx's FPGAs is fairly versatile. For the Virtex-4, each block consists of 18 kbits, and can be configured in any aspect ratio from 16k single bits, 8k 2-bit words, up to 512 36-bit words. Also, blocks can be automatically chained together to form larger memories. For example, by specifying a RAM with 1024 words that are 36 bits each, the synthesis tools would connect two blocks together.

BRAM also supports dual-ports. Each port has its own data in, data out, address bus, and clock, as well as EN and WE control lines. The two ports can even vary in data word width.

To get the synthesizer to recognize that BRAM should be used instead of distributed RAM, you must use a particular entity declaration and architecture. A simple example is shown here:

```
entity my_ram is
        port (          clka : in std_logic;
                        ena : in std_logic;
                        wea : in std_logic;
                        addra : in std_logic_vector(7 downto 0);
                        dia : in std_logic;
                        doa : out std_logic);
end my_ram;

architecture Behavioral of my_ram is
type ram_type1 is array (255 downto 0) of std_logic;
signal RAM1 : ram_type1;
begin
        process (clka)
        begin
                if (clka'event and clka = '1') then
                        if (ena = '1') then
                                if (wea = '1') then
                                        RAM1(conv_integer(addra)) <= dia;
                                end if;
                                doa <= RAM1(conv_integer(addra));
                        end if;
                end if;
        end process;

end Behavioral;
```

The BRAM described above uses a single port, and consists of 256 single-bit entries. The number of addresses is controlled simply by the address bus width, and by the array size of *ram_type1*. Likewise, the word width is controlled by the data bus widths, and again in the element size of the type *ram_type1*. Also note that this BRAM writes first, then places data on the output bus. Read-then-write and no-change synchronizations are also supported. Here is an example of a dual-port BRAM.

```
entity vector_ram is
        port(clka : in std_logic;
        clkb : in std_logic;
        ena : in std_logic;
        enb : in std_logic;
        wea : in std_logic;
        web : in std_logic;
        addra : in std_logic_vector(8 downto 0);
        addrb : in std_logic_vector(8 downto 0);
        dia : in std_logic_vector(35 downto 0);
        dib : in std_logic_vector(35 downto 0);
        doa : out std_logic_vector(35 downto 0);
        dob : out std_logic_vector(35 downto 0));
end vector_ram;

architecture syn of vector_ram is

type ram_type3 is array (0 to 511) of std_logic_vector(35 downto 0);
shared variable RAM3 : ram_type3;
begin

        process (CLKA)
        begin
                if CLKA'event and CLKA = '1' then
                        if ENA = '1' then
                                if WEA = '1' then
                                        RAM3(conv_integer(ADDRA)) := DIA;
                                end if;
                                DOA <= RAM3(conv_integer(ADDRA));
                        end if;
                end if;
        end process;

        process (CLKB)
        begin
                if CLKB'event and CLKB = '1' then
                        if ENB = '1' then
                                if WEB = '1' then
                                        RAM3(conv_integer(ADDRB)) := DIB;
                                end if;
                                DOB <= RAM3(conv_integer(ADDRB));
                        end if;
                end if;
        end process;
end syn;
```

Other BRAM configurations, as well as code samples for many other useful synthesis constructs, can be found in ISE under Edit-> Language Template menu, in the VHDL->Synthesis Constructs -> Coding Examples folder. More information about Virtex-4 BRAM resources can be found in the Virtex-4 User's Guide.

Timing Constraints and Synthesis Effort

We often want to maximize the operating frequency of a design, or at least reach a minimum target speed. While pipelining and parallel processing are two tools that help us with those tasks, maximum operating frequency is heavily dependent on how the place and route tools map a design onto the FPGA logic fabric. There are two primary means for telling the tools what areas to focus on: timing constraints and synthesis effort.

Synthesis effort is a global effect, and is good to use if you have extra logic space available on the FPGA and want to squeeze more speed out of the design. Alternatively, you can reduce the footprint of the design at the expense of speed. By right-clicking on the Synthesize menu in the Processes window and selecting Properties, you can specify what optimization you're after. There are two properties to adjust: Optimization Goal and Optimization Effort. Optimization Goal lets you specify whether you're more worried about speed or area. Optimization Effort tells the computer how much computing power you're willing to throw at the problem. Setting this to 'High' tells the computer that you're willing to wait a while (sometimes a long while) for even more speed or less area. Sometimes the effect is very small, and may not be worth the wait. However, changing the Optimization Goal can have a drastic impact on size versus speed.

Another synthesis option to consider is Resource Sharing (found by right-clicking on Synthesize, selecting Properties, and clicking on the HDL Options category, 2nd from bottom). If resource sharing is selected, the synthesizer will allow logic functions to share common logic paths (for example, two separate adders may be allowed to share some adder circuitry). Resources sharing usually results in slower performance, but saves area. If speed is what you're after, turn off Resource Sharing.

Specifying a timing constraint tells Place & Route what you're timing requirements are for particular signal or path. P&R will then spend extra effort looking for ways to place logic blocks and route signals to meet your goal. This tool should be used carefully; if your constraints are too stringent, or if you specify too many constraints, P&R may not find a solution. Be selective. However, placing one or two constraints on global clock signals or long, vital data paths very often improves speed considerably.

Timing constraints are most easily added using the Constraints Editor, under User Constraints in the Processes window. Here, you can enter the desired period for clock signals (under the Global tab), and desired setup and offset times for signals (under the Ports tab). Simply enter your values in the editor, save, and close. The next time you run Place & Route, your criteria will be taken into account.