A VHDL Design Methodology for FPGAs

Michael Gschwind, Valentina Salapura

Institut für Technische Informatik
Technische Universität Wien
Treitlstraße 1, A–1040 Wien, AUSTRIA

Abstract

In order to generate efficient FPGA designs, the HDL description style has to be adapted to the requirements of FPGA architecture. Unlike ASIC targets, FPGAs offer a fixed set of resources which can be used to generate an efficient design. This requires that the HDL source code of a design be adapted to exploit the available resources.

Since the structure of synthesized logic is inferred from the description style, the source-level coding has to be adapted to yield optimal designs. Generating optimized designs requires a good understanding of the FPGA target. In addition, the code has to consider the description styles supported by a particular synthesis tool, while the specific hardware description language used is only of minor importance.

1 Introduction

FPGAs are an efficient hardware target when only small series are needed, or for rapid prototyping. The FPGAs are complex enough to implement more than glue logic, including complex designs up to several thousands gates. As the logic capacity of FPGAs increases, synthesis for FPGAs is becoming more important.

To efficiently exploit increased logic capacity of FPGAs, synthesis tools and efficient synthesis methods for FPGAs targeting become necessary. One solution to designing large designs efficiently is to use VHDL [3] synthesis. Several synthesis tools exist for mapping these descriptions to various FPGA families.

Using a synthesis-based approach, retargeting a design to other technologies becomes possible at little extra cost. When using FPGAs for rapid prototyping, synthesis can be targeted at FPGAs to exercise it for verification purposes, and later an ASIC implementation can be derived.

While ideally, the synthesizable VHDL model should be the same for all target technologies, the efficiency of the resulting design is very much dependent on the description and technology used. This paper discusses optimization issues and methodology for VHDL designs targeted at FPGAs. While this issue has been raised for ASIC designs [5], many issues remain for FPGA targets.

Due to their architecture, optimization problems found in ASIC designs may be amalgamated, heightened or outright reversed for FPGA designs. In this respect, especially LUT-based architectures (such as the Xilinx devices used as example in this paper) are different due to their coarse-grained architecture, while finer-grained architectures behave more like ASICs.

Designing with FPGAs, one of the major differences is that logic functions of the same size cannot be traded: there is a given number of every resource, and whether it is used or not will not change chip size. On the other hand, trading a 'cheaper' (less complex) cell for a more 'expensive' (more complex) one can actually improve the device budget, if there is an ample amount of the more expensive resource available.

We discuss design strategies for generating efficient VHDL models for FPGA synthesis. The results presented here were obtained empirically by generating various descriptions for the same semantic operation, compiling them and comparing their timing and area characteristics.

This paper is organized as follows: in section 2, we describe the environment used to collect the data. Section 3 shows the usage of fast-carry logic, and section 4 gives an overview of finite state machine optimization for FPGAs. Optimization of multiplexing structures is covered in section 5, and section 6 discusses storage structures. Section 7 describes the interaction between synthesis tools and target specific fitters for placement and routing, and we draw our conclusions in section 8.

2 Environment

The experiments described here were made using the Xilinx XC4000 FPGA series [12]. We have chosen this architecture mainly for tool and support availability, but also because they are a very versatile and advanced FPGA technology.

The data presented here were collected using the Synopsys VHDL design analyzer/FPGA compiler (versions 3.1a–3.3a) [9] [6], the XSI Xilinx/Synopsys interface [14] and X-BLOX as cell generator (XACT 5.1). Synopsys synthesis and XACT were targeted at a Xilinx XC4013mq240-5 FPGA. For low-level operations, we use the XACT and Viewlogic/Powerview tools for analysis and simulation [13], [11].

The code for various test circuits was written in VHDL, using the IEEE Std_Logic_1164 package [4]. This package is used in most new VHDL synthesis tools and ensures code portability between tools from different vendors. The synthesis syntax for a given function block may also depend on the tool. The syntax given in this paper was tested using the Synopsys Design Analyzer.

The underlying optimizations described here are not restricted to a particular source code format. Thus, they are not restricted to VHDL, but apply equally well to other hardware description languages such as Verilog. In fact, many synthesis tools are independent of the source language and have front-ends for both VHDL and Verilog, as well as other special-purpose formats for lookup tables, state machines, etc.

The initial structure of synthesized logic is directly inferred from the structure of the hardware description. Thus, the quality of the final hardware very much depends on the description style used at a higher level. To account for this , the high-level description has to be adapted to guide the synthesis tool to choose the appropriate implementation. This is especially important to exploit special-purpose features such as fast carry logic available in many architectures.

3 Efficient adder implementation

The Xilinx XC4000 series contains special purpose hardware to efficiently implement fast carry logic as found in adders, subtracters, counters and other related function blocks. When special purpose circuitry such as this is available, an optimal solution is based on the usage of these facilities.

Normally, no algorithmic advantages can be gained by substituting a superior description for such a function block, as the special purpose hardware is implemented at the hardware level. Thus, a brute force approach is given a significant advantage, as it maps directly to the hardware.

It is, however, difficult for VHDL compilers to use special purpose features which are available in FPGAs under certain conditions, such as the fast-carry logic or the builtin RAM.

Xilinx provides a partial solution to this problem by supplying a DesignWare library for adders, subtracters, counters, and comparators. In Synopsys, DesignWare libraries are used to implement common, complex functional units which can be used by the design analyzer. In the X-BLOX DesignWare library, these functional units are not actually implemented using the Synopsys FPGA compiler. Instead, references to X-BLOX modules are inserted in the net list. When the design is post-processed for final layout using XACT, X-BLOX is invoked as module generator to synthesize appropriate functional units. X-BLOX has intimate knowledge of Xilinx circuits, so it can generate logic geared to special features such as fast-carry logic.

To compare different modeling styles and their implementation in hardware, we have described an adder module with several different algorithms:

rip a ripple-carry adder

srip a structured ripple-carry adder built from 1-bit full adder modules

cla a carry look-ahead adder

“+” using the VHDL “+”operator

These adders have been described in different styles, and compiled with and without X-BLOX. The description style played little role in the final hardware efficiency, and only the VHDL “+” operator could be mapped to a Xilinx DesignWare block using fast-carry logic. When compiling the circuits without the X-BLOX library, the only difference was the usage of a Synopsys DesignWare fast-carry adder. The Synopsys DesignWare library also contains a ripple-carry adder, which may be used instead of the carry look-ahead adder. Synopsys FPGA Compiler automatically selects the implementation used for HDL operators depending on the optimization constraints.

For function blocks with a width of 4 bits or less, Synopsys does not introduce a level of hierarchy for instantiated HDL operators. Since Synopsys cannot map a sea-of-gates description generated for these function blocks to a DesignWare module, structures with 4 bits are not mapped to X-BLOX modules. Thus, the timing and resource usage of a single, narrow module is actually worse than that of larger function blocks.

The advantage of using an ungrouped sea-of-gates representation of small modules is that they can be integrated and optimized with surrounding logic. This is not possible when using the X-BLOX DesignWare libraries which present a black box to the Synopsys FPGA compiler. This integration effect of HDL operators with surrounding

model	4	6	8	12
rip	9	17	25	33
srip	9	18	28	36
cla	7	15	23	33
+, no X-BLOX	10	18	25	33
+, X-BLOX	10	6	8	10

Table 1: Size in CLBs of adders (widths from 8 to 12 bits) using different description methods. The test circuits were synthesized using FPGA compiler 3.3a, and routed using `ppr` (XACT 5.1). Area results as reported by `ppr`.

model	4	6	8	12
rip	60.70	117.90	173.50	233.30
srip	60.80	129.40	159.20	220.80
cla	58.80	89.20	112.80	150.40
+, no X-BLOX	80.60	94.40	110.80	169.60
+, X-BLOX	80.60	51.10	54.40	56.40

Table 2: Timing (in ns) of adders (widths from 8 to 12 bits) using different description methods. The test circuits were synthesized using FPGA compiler 3.3a, and routed using `ppr` (XACT 5.1). Timing results are pad-to-pad delays as reported by `xdelay` and include the propagation delay of input and output pads. Thus, relative speed differences are more pronounced than they may appear here.

logic explains the results reported by Fields [2], where the same design has been compiled with and without X-BLOX.

As expected, the version using the X-BLOX DesignWare library had better performance. Resource usage was however higher for the X-BLOX based version, which contradicts the results reported in table 1. This discrepancy arises because table 1 reports the resource usage of stand-alone modules, whereas [2] reports resource usage for a specific design where adders may share LUTs and CLBs with surrounding logic.

4 State machines

The generation of state machines is another area where conventional ASIC synthesis and FPGA synthesis differ. When ASICs are the target technology, fully encoded representations such as binary or gray code encoding of states lead to space efficient designs, whereas the faster one-hot encoding scheme consumes more resources [8].

This is different in FPGA designs, where the state decoding logic for decoding a binary encoding would consume many CLBs, while many flip-flops on the same die go unused! Thus one-hot encoding is not only much faster, but also the more compact representation [1].

Table 3 gives the synthesis results of a simple finite state machine for the Xilinx XC4000 series and the LSI 10k ASIC library [7]. The table compares four encoding techniques available in Synopsys: one-hot encoding, a solution adapted to the particular FSM (auto), gray code encoding, and binary encoding of states.

The one-hot encoding scheme uses 6 flip-flops for state encoding, while all other implementations use 3. Although this leads to significantly larger ASIC implementations, the FPGA FSM implementation is comparable to the smallest solution. While the optimal encoding always depends on the particular state machine being used, for most state machines one-hot encoding is superior for FPGA implementations. One-hot encoding not only is the

	XC4000		*LSI 10K*	
FSM Encoding	time (ns)	space (CLBs)	time (ns)	space (units)
one-hot, space	18.2	8	17.0	73
one-hot, time	18.2	8	11.4	95
auto, space	56.8	7	13.9	46
auto, time	33.7	7	11.1	68
gray, space	26.6	7	18.1	58
gray, time	32.7	8	9.7	28
binary, space	52.3	10	17.2	54
binary, time	43.1	13	12.6	92

Table 3: Resource usage for FSM compilation using different encoding schemes and optimization constraints. These results were reported by FPGA compiler and design compiler, respectively (version 3.1a).

```
ARCHITECTURE mux OF select IS
BEGIN
  value_out <= source (conv_integer(unsigned(sel)));
END mux;
```

Figure 1: Signal selection using a multiplexer.

```
ARCHITECTURE tristate OF select IS
BEGIN
  tri_state_bus:
    FOR i IN 0 TO depth -1 GENERATE
      value_out <= source (i) WHEN (i = conv_integer(unsigned(sel)))
                   ELSE (Others => 'Z');
    END generate;
END tristate;
```

Figure 2: Signal selection using a tri-state bus.

fastest encoding, but also one of the smallest representations because it exploits the availability of many flip-flops on an FPGA.

In some tools, VHDL source level encoding of the state vector may be necessary to achieve this. Synopsys supports the extraction of state machines from a design and to define an encoding to be used for the state vector. This approach is advantageous, since several different encodings can be tested and compared without having to modify the source code.

5 Signal selection

Selecting between two input signals is a common operation. Many conditional VHDL statement will generate a multiplexer to choose between different input sources:

```
IF (sel = '0') THEN
  out <= signal_0;
ELSE
  out <= signal_1;
END IF;
```

But multiplexers can also be introduced with other constructs, where it is less obvious. For example, choosing a particular input source with an index will normally generate a multiplexer (figure 1). These multiplexers grow with the number of input signals and signal width.

Multiplexers are expensive to implement in FPGAs, as their implementation requires many CLBs and routing resources. An alternative method of selecting an input signal from several options is to use a tri-state bus. This method is advantageous on Xilinx FPGAs, as tri-state buffers and tri-state buses (in the form of longlines) are already integrated on the chip.

Tristate devices can be generated using the following assignment:

```
bus <= value WHEN enable ELSE (Others => 'Z');
```

Using tristate drivers, a similar signal selection can be implemented with a tristate bus (see figure 2). Depending on the FPGA part, Xilinx supports between 16 and 64 three-state busses ("longlines") per chip and between 10 and 34 tri-state buffers per longline. If these longlines are not used by any other circuitry, using them for signal selection allows to pack more functionality in a single FPGA.

Depending one the number of input sources and input width, different implementations may optimize either resource usage or timing. In the XC4000 series, a 4:1 multiplexer can be implemented using a single CLB, leading to minimal timing. Larger multiplexers exceed the capacity of a single FPGA, and as the number of inputs increases, tri-state solutions offer competitive timing.

Especially for wide signals or a large number of input sources, a tri-state based selection method reduces CLB usage drastically. No combinatorial logic is necessary to select the input, instead only the select signal is decoded to drive tri-state buffers which flank each CLB. For narrow signals (1 or 2 bits), CLB usage is comparable.

For wider signals, a selection mechanism based on tri-state functionality is preferable: the tristate implementation uses a fixed number of CLBs for generating tri-state buffer control signals, and a tri-state buffer for each

signal bit, i.e., $n * w$ tri-state devices (n being the number of signals, w the width of the signal in bits). Often, these tri-state resources are unused, so this implementation increases overall FPGA resource utilization. Tables 4 and 5 compare the FPGA resource usage for signal selection using multiplexers and tristate buffers, respectively.

	bus width						
		1	2	4	8	16	32
no. of signals	2	1	1	2	4	8	32
	4	1	2	4	8	16	32
	8	3	5	14	24	44	84
	16	5	10	28	48	88	168
	32	11	26	61	103	187	355

Table 4: This table shows the number of CLBs required for selecting an output signal with multiplexing logic, as a function of the number of input signals and signal width. These results were reported by FPGA compiler 3.1a.

	bus width											
		1		2		4		8		16		32	
		CLBs	tri	CLBs	tri	CLBs	tri	CLBs	tri	CLBs	tri	CLBs	tri
no. of signals	2	1	2	1	4	1	8	1	16	1	32	1	64
	4	2	4	2	8	2	16	2	32	2	64	2	128
	8	4	8	4	16	4	32	4	62	4	128	4	256
	16	8	16	8	32	8	64	8	128	8	256	8	512
	32	19	32	19	64	19	128	19	256	19	512	19	1024

Table 5: This table shows the number of CLBs and tristate buffers required for selecting an output signal using a tristate bus, as a function of the number of input signals and signal width. These results were reported by FPGA compiler 3.1a.

One caveat is the number of tri-state resources (buffers and longlines) which are available and their connectivity. Since the connectivity of tri-state buffer elements (TBUF) is fixed, and there is a limited number of longlines, there are upper bounds as to the size of the tri-state select mechanism. For example, the XC4010 has 40 longlines and 22 TBUFs per longline, restricting the select mechanism to a width of 40 bits (if all longlines are dedicated to a single select) and the output can be selected from a maximum of 22 signals. (In the XC4000 series, horizontal longlines can be split, so the XC4010 can also be configured as having up to 80 longlines with 11 TBUFs per longline.)

The choice of output selection mechanism has a significant influence on the size of all blocks where signals have to be selected, e.g. register files.

6 Storage structures

6.1 Sequential elements

In Xilinx FPGAs, each CLB contains 2 flip-flops which often go unused. Using such a flip-flop does not use any extra resources in most cases, as it will be located in the block which computes the result.

Latches have to be built using CLB function generators and require 1 CLB per bit (whereas a flip-flop normally comes for free).

Sometimes, of course, the exact functionality of either a flip-flop or a latch is required. In these cases, the appropriate type of storage element has to be used. But when the exact nature of the storage element is of minor importance, flip-flops are obviously beneficial.

6.2 Memory elements

While using the X-BLOX library helps optimize some circuits, others cannot be optimized to use these features. A solution is to include Xilinx library elements as 'components' in VHDL, and use either already available circuits in the Xilinx libraries, or to generate one's own circuits with XACT (The full range of XACT tools can be used to generate these circuits: X-BLOX, `memgen`, XDE,…) or a schematic entry tool to be included in the design. These parts can then be optimized to use all features available on a particular target technology. While this approach requires more radical VHDL source code modification, the huge gains possible make this worthwhile for some resources. Isolating these features in a distinct `ENTITY` will enhance portability and restrict code changes to only a few lines of code. Note that these resources (RAMs, etc.) will probably also need to be adapted to the specific ASIC process to yield optimal results.

storage element	selection scheme	area (CLBs)	time (ns)
latches	multiplexers	570	88.2
latches	tri-state bus	402	99.8
flip flops	multiplexers	283	58.7
flip flops	tri-state bus	156	51.5
XC4000 RAM capability		8	29.2

Table 6: Occupied CLBs using five different VHDL coding strategies for a 16x16 scratch pad RAM. The test circuits were synthesized using FPGA compiler 3.2a, and routed using `ppr` (XACT 5.1). Area results as reported by `ppr`. Timing results are pad-to-pad delays as reported by `xdelay` and include the propagation delay of input and output pads. Thus, relative speed differences are more pronounced than they may appear here.

```
ARCHITECTURE xilinx_ram_capability OF scratch_pad IS
  ...
  COMPONENT RAM16X1
  PORT ( D, A3, A2, A1, A0, WE : IN std_logic;
         O : OUT std_logic);
  END COMPONENT;
  ...
BEGIN
  ...
    FOR i IN 0 TO width - 1 GENERATE
      cell_ram16x1: RAM16x1 PORT MAP (
                                      D  => value_in(i),
                                      A3 => addr(3),
                                      A2 => addr(2),
                                      A1 => addr(1),
                                      A0 => addr(0),
                                      WE => write,
                                      O  => value_out(i));
    END generate;
  ...
END xilinx_ram_capability;
```

Figure 3: VHDL code for generating a scratch pad RAM based on the Xilinx XC4000 series RAM capability. Large RAMs can be assembled using the basic RAM capability available as macros (`RAM16x1`, `RAM32x1`) in the XACT library.

Table 6 shows how a design can be optimized by using Xilinx XC4000 features. We compare 5 different designs for a 16x16 RAM. Depending on the VHDL description, Synopsys FPGA compiler generates either flip-flops or latches as storage elements, and uses either a MUX-based signal selection scheme or tri-state buses. These implementation specifics are orthogonal, giving four possible implementations. The synthesis results for these four different architectures show how coding style in VHDL can affect resource consumption, in the example of our 16x16 scratch pad RAM yielding a 4-fold improvement.

A fifth, alternative design is based on the usage of the Xilinx XC4000 capability, where each FG function generator can be reprogrammed to act as 16x1 RAM cell. By using this customization of the RAM cell, the final design uses only 1.4% of the original design.

To generate this design, we use macros from the Xilinx XACT library, which are instantiated as VHDL components in the VHDL description (see figure 3). An alternative way to create RAMs based on the Xilinx RAM capability is to use the `memgen` tool [13] from the XACT distribution and include the resulting RAM as `COMPONENT`.

7 Placement and routing

Due to the fixed layout and limited routing resources of FPGAs, placement and routing is a very important issue in FPGAs. In general, synthesis tools do not consider placement and routing. Low-level operations such as place and route are normally the problem domain of fitters, which understand the architecture of the specific target device. Only so can one expect reasonable operations, even though few fitters today feature sophisticated place and route strategies.[1]

Fitters also perform low-level optimizations to take full advantage of the target architecture. In many cases, these fitters are also responsible for mapping a generic net list onto LUT-based CLB structures. Depending on the

[1] One solution to improve the performance of placement and routing for large designs is the use of floor planning for FPGAs [2] [15].

synthesis tool used, partitioning into CLBs may occur either during the synthesis or routing step. A good example is Synopsys, which offers two different synthesis tools, Design Compiler and FPGA Compiler. While FPGA compiler understands the LUT-based structure of CLBs, Design Compiler views design as sea-of-gates structures, with each gate contributing a given delay.

Obviously, the FPGA compiler reports more accurate timing information and can perform better target-specific optimization. However, not all FPGA families are supported with FPGA Compiler. Thus, while the Xilinx 4k series is supported by FPGA Compiler (a library for Design Compiler also exists), the Xilinx 3k and 7k and the Altera Flex8000 families are only supported by Design Compiler. The level and quality of timing information given by Design Compiler varies, depending on the family. While Design Compiler does not provide any timing information at all for Xilinx 7k EPLD series, the Altera Flex8000 family does contain timing information.

The timing information provided by the Altera Flex8k library seems to contain worst-case timing information for single gates. Since multiple gates can normally be allocated to a single LUT, delays are grossly over-estimated. Realistic timing reports can only be generated after the placement and routing using Altera's fitter `maxplus2`.

As timing characteristics are influenced heavily by placement and routing in FPGA technology, meaningful gate-level simulation can only be performed with backannotated timing information generated by the appropriate fitter .

8 Conclusion and future work

We have shown that VHDL models are highly dependent on the target technology. A slight modification in the description can cause considerable change in the implementation efficiency, especially when dealing with fixed-resource FPGAs.

By using target-device specific optimizations and description styles, overall circuit performance can be enhanced greatly. We have demonstrated how VHDL circuits can be optimized for FPGA targets by adapting descriptions styles to the available resources, such as flip-flops, three-state buffers and others. This affects coding styles for many basic design blocks, such as storage elements, multiplexers and finite state machines.

We have also shown how to exploit FPGA-specific special-purpose circuitry, such as fast carry logic or RAM blocks. Using these special purpose logic can yield tremendous improvements in space efficiency and timing. In the future, we will also more closely explore the interaction between high-level synthesis tools and vendor-specific FPGA fitting tools, and also design portability and device retargetability.

References

[1] Peter Alfke and Bernie New. Implementing state machines in LCA devices. In *The Programmable Logic Data Book*, pages 8–169 – 8–172. Xilinx, Inc., San Jose, CA, 2nd edition, 1994. XAPP 027.001.

[2] Carol A. Fields. Proper use of hierarchy in HDL-based high density FPGA design. In Will Moore and Wayne Luk, editors, *Field-Programmable Logic and Applications: 5th International Workshop, FPL '95*, volume 975 of *Lecture Notes in Computer Science*, pages 168–177, Berlin, Germany, August 1995. Springer Verlag.

[3] IEEE. *IEEE Standard VHDL Language Reference Manual*. IEEE, New York, NY, 1988. IEEE Standard 1076-1987.

[4] IEEE. *IEEE Standard Multivalue Logic System for VHDL Model Interoperability (std_logic_1164)*. IEEE, New York, NY, 1993. IEEE Standard 1164-1993.

[5] Manfred Selz. *Untersuchungen zur synthesegerechten Verhaltensbeschreibung mit VHDL*. PhD thesis, Universität Erlangen-Nürnberg, Erlangen, Germany, March 1994.

[6] Synopsys. *Design Compiler Family Reference*. Synopsys, Inc., Mountain View, CA, April 1995. (Version 3.3a).

[7] Synopsys. Finite state machine tutorial source code. `${SYNOPSYS}/doc/syn/examples/fsm/proc2.vhd`, April 1995. (Version 3.3a).

[8] Synopsys. *Finite State Machines–Application Note.* Synopsys, Inc., Mountain View, CA, April 1995.

[9] Synopsys. *VHDL Compiler Reference*. Synopsys, Inc., Mountain View, CA, April 1995. (Version 3.3a).

[10] Synopsys. *VSS Reference*. Synopsys, Inc., Mountain View, CA, April 1995. (Version 3.3a).

[11] Viewlogic Systems. *Using Powerview*. Viewlogic Systems, Inc., Marlboro, MA, 1994.

[12] Xilinx. *The Programmable Logic Data Book.* Xilinx, Inc., San Jose, CA, 2nd edition, 1994.

[13] Xilinx. *XACT Reference Guide*. Xilinx, Inc., San Jose, CA, April 1994.

[14] Xilinx. *XACT Xilinx Synopsys Interface FPGA User Guide*. Xilinx, Inc., San Jose, CA, December 1994.

[15] Xilinx. *Floorplanner User Guide*. Xilinx, Inc., San Jose, CA, February 1995. (preliminary version).