A Multi-Level FPGA Synthesis Method Supporting HDL Debugging for Emulation-Based Designs

Wen-Jong Fang, Peng-Cheng Kao, and Allen C.-H. Wu
Department of Computer Science
Tsing Hua University
Hsinchu, Taiwan, 300, R.O.C

Abstract

Converting an HDL-based design into an emulation system for design verification is an extremely complex and time-consuming task. One possible solution to improve productivity is an effective emulation-based design methodology that exploits the modularity of designs. This paper develops and explores such a methodology. We present a multi-level synthesis method which is able to establish a direct link between High-level Descriptive Language (HDL) constructs and corresponding circuit designs. This feature greatly facilitates HDL debugging capabilities such that when bugs are detected in sub-netlists, links allow the corresponding HDL source code to be easily recognized. This powerful feature greatly reduces design debugging time. Experimental results on a set of industrial designs are reported in order to demonstrate the effectiveness of the proposed synthesis methodology.

1 Introduction

Because of their reprogrammability, Field Programmable Gate Arrays (FPGAs) have become one of the most popular Application-Specific Integrated Circuits (ASICs) for rapid system prototyping. In addition, the development of reconfigurable systems by integrating FPGAs and Field Programmable Interconnect Chips (FPICs) has become the new trend in rapid prototyping and computation-intensive applications [1, 2, 3].

Logic emulation was the first technique to emerge that uses dynamically reprogrammable systems for prototyping and design verification [1, 4]. A logic emulator is a system consisting of hundreds or thousands of FPGAs and FPICs, which implements a design through a software configuration procedure. A configured logic emulator is equivalent to the chip under design, and can be used for real-time design verification, software development, and prototyping before chip fabrication.

In general, a typical logic-emulation design process consists of two phases: (1) pre-configuration preparation and (2) full-chip configuration [6], as depicted in Figure 1. In the pre-configuration preparation phase, emulation designers create a functionally-equivalent gate-level description for emulation use, which will be replaced later by the actual gate-level design from the hardware design team. In the full-chip configuration phase, designers convert the CLB-based netlist of the design into a set of bit-stream files which can be downloaded into the target emulator. The full-chip configuration phase consists of four steps [6]: (1) partitioning, (2) system mapping, (3) FPGA placement-and-routing, and (4) design downloading (Figure 1).

In this paper, we present a multi-level synthesis method for emulation-based designs and how it can be applied to HDL debugging. Our multi-level synthesis approach tends to establish a direct link between the generated circuit designs and their corresponding HDL constructs. When bugs are detected in sub-netlists during HDL debugging process, links allow the corresponding HDL source code to be easily recognized. Experiments have been conducted and reported to evaluate the effectiveness and efficiency of the proposed synthesis method.

2 Problem Description

Typically, a design under development consists of modules, such as random logic, data path, and memory, which are organized in a hierarchical way. In many industrial design projects, designers reuse some of the ex-
The main objective of the multi-level synthesis method is to build direct links between the constructs of an HDL description and their corresponding circuit designs. We use an HDL structural tree to represent the structural hierarchy of the HDL description of a design. In an HDL structural tree, the root node represents the design, each intermediate node represents higher level construct such as modules, processes, and tasks, and each leaf node represents an independent functional block. An independent functional-block contains a set of statements having the same output signal and it can be synthesized into an independent CLB design without affecting any other statements.

In the Verilog HDL, there are four types of basic statements: (1) assignment, (2) conditional, (3) loop, and (4) event-control statements. In the following sections we first describe the techniques to convert conditional and loop statements into sets of independent functional-blocks. Then, we present the multi-level synthesis method to convert an HDL description into an HDL structural tree.

3.2 Synthesis of conditional and loop statements
3.2.1 Case and if-then-else statements

The case statement is usually used for multi-way branches when all the conditionals are based on the same expression. Figure 4(a) depicts a case description. This description denotes that statements \{S1, S5\}, \{S2, S6\}, \{S3, S7\} or \{S4\} will be executed only when their corresponding conditional C1, C2, C3, or C4 is true. Figure 4(b) shows the top-level circuit block diagram to realize this description, which takes inputs of six signals \texttt{sel}[0 : 1] and \texttt{i1} – \texttt{i4} and outputs two signals \texttt{o1} and \texttt{o2}.

From the hardware point of view, the outcomes of the output signals are dependent on the execution of the statements embedded in the case statement. Hence, we can establish the relationship between each output and the statements which will directly affect it. Consider the output signal \texttt{o1}. By analyzing the description in Figure 4(a), we observe that the outcome of \texttt{o1} is depen-
dent on the result of executing one of the four statements $S_1 - S_4$ under the control of the $sel$ signal. Hence, we can view the structural design of $o_1$ as a Selector which selects the output of one of the four statements controlled by the $sel$ signal, as illustrated in Figure 4(c). As a result, the $o_1$-based circuit can be decomposed into five independent functional-blocks: the circuits of statements $S_1 - S_4$ and the case statement (i.e., a 4-input selector). Similarly, $o_2$ can be implemented as a Selector which selects the output of one of the three statements $S_5 - S_7$ controlled by the $sel$ signal. One exception is that the outcome of $o_2$ will not change when the control signal $sel$ is “11”. Thus, a dummy statement $o_2$ is added to maintain the consistency between the HDL description and its synthesized design. Finally, we can construct the HDL structural tree, as shown in Figure 4(d).

The if-then-else statement and its variations are the other conditional statements used for altering the control flow in a sequential description. Since multi-way branching can be expressed using nested if statements with else clauses, we assume that a nested if-then-else statement is treated as a basic-statement. We first construct a Control Data Flow Graph (CDFG) so that statements which should be executed based on specified conditions can be determined. For example, Figures 5(a) and (b) depict a nested if-then-else Verilog description and its corresponding CDFG. By analyzing the CDFG, we observe that the execution of statements $S_2$, $S_3$, and $S_4$ are controlled by two conditions $C_1$ and $C_2$. $C_1$ contains the statement $S_1 = x_1 \& x_2$ which can be extracted as an independent functional-block. The top-level circuit structural view, the HDL structural view, and the HDL structural tree are depicted in Figures 5(c), (d), and (e), respectively.

### 3.2.2 Synthesis of a process with multiple statements

A process usually consists of multiple conditional, loop, and assignment statements which are executed in sequential order. We use a hierarchical method to synthesize a process with multiple statements. In the first step, we determine the basic-statements in the process. The basic-statements are the statements which can be implemented as an independent hardware blocks, including assignment and conditional statements. For example, the description in Figure 6(a) contains three basic-statements $B_1$, $B_2$, and $B_3$. For each basic-statement, we can determine its input and output signals, as shown in Figure 6(b). We first synthesize each basic-statement into a HDL structural tree using the techniques described in the previous sections.

In the second step, we establish the relationships between output signals of the process and the statements embedded in the process. Since the statements in a process are executed in sequential order, in order to obtain a functionally-correct synthesized circuit, the data and control dependencies between the statements in the same process need to be analyzed. In other words, we have to determine the data and control relations between the generated hardware blocks and the outputs of the process. This can be accomplished by constructing a statement-level control/data flow graph, as shown in Figure 6(c). From the CDFG analysis, we observe that $o_1$ is used as an internal signal (e.g., an output of $B_1$ and an input for both $B_2$ and $B_3$) as well as the output of the process. Furthermore, if both conditions of $x_2$ and $x_3$ are not true then the output signal of $o_1$ should be determined by $B_1$. Hence, the outputs of both $B_2$ and $B_3$ are dependent on the output of $B_1$ and the resulting HDL structural view and structural tree are depicted in Figures 6(b) and (d), respectively.

If a process contains a loop statement such as a for or a while statement, we first unroll the loop body into a number of loop-body blocks. We then form a hardware block for each loop-body block by applying the data/control flow analysis and synthesis methods described.
Table 1: Intermediate nodes generated by the multi-level synthesis method

<table>
<thead>
<tr>
<th>#Lines</th>
<th>#M</th>
<th>#P</th>
<th>#S</th>
<th>#O</th>
<th>#B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>21</td>
<td>1</td>
<td>6</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>86</td>
<td>3</td>
<td>9</td>
<td>16</td>
<td>61</td>
</tr>
<tr>
<td>3</td>
<td>104</td>
<td>9</td>
<td>37</td>
<td>89</td>
<td>180</td>
</tr>
<tr>
<td>4</td>
<td>51</td>
<td>1</td>
<td>12</td>
<td>42</td>
<td>42</td>
</tr>
<tr>
<td>5</td>
<td>16</td>
<td>1</td>
<td>20</td>
<td>42</td>
<td>280</td>
</tr>
<tr>
<td>6</td>
<td>16</td>
<td>1</td>
<td>15</td>
<td>136</td>
<td>217</td>
</tr>
<tr>
<td>7</td>
<td>12</td>
<td>2</td>
<td>11</td>
<td>16</td>
<td>309</td>
</tr>
<tr>
<td>8</td>
<td>40</td>
<td>10</td>
<td>100</td>
<td>358</td>
<td>676</td>
</tr>
<tr>
<td>9</td>
<td>16</td>
<td>1</td>
<td>16</td>
<td>228</td>
<td>496</td>
</tr>
</tbody>
</table>

Table 2: Circuit-size/run-time ( #CLBs /Secs.) comparisons

<table>
<thead>
<tr>
<th>M</th>
<th>P</th>
<th>S</th>
<th>O</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7/2</td>
<td>7/2</td>
<td>10/2</td>
<td>11/3</td>
</tr>
<tr>
<td>2</td>
<td>32/11</td>
<td>52/15</td>
<td>76/14</td>
<td>83/12</td>
</tr>
<tr>
<td>3</td>
<td>29/8</td>
<td>43/9</td>
<td>56/10</td>
<td>83/21</td>
</tr>
<tr>
<td>4</td>
<td>20/4</td>
<td>20/4</td>
<td>21/5</td>
<td>29/9</td>
</tr>
<tr>
<td>5</td>
<td>56/30</td>
<td>56/24</td>
<td>115/18</td>
<td>140/22</td>
</tr>
<tr>
<td>6</td>
<td>32/12</td>
<td>32/13</td>
<td>42/16</td>
<td>113/31</td>
</tr>
<tr>
<td>7</td>
<td>134/57</td>
<td>135/61</td>
<td>174/60</td>
<td>195/97</td>
</tr>
<tr>
<td>8</td>
<td>59/23</td>
<td>61/14</td>
<td>63/15</td>
<td>87/20</td>
</tr>
<tr>
<td>9</td>
<td>115/29</td>
<td>130/47</td>
<td>157/48</td>
<td>244/73</td>
</tr>
<tr>
<td>10</td>
<td>78/38</td>
<td>95/41</td>
<td>148/60</td>
<td>320/82</td>
</tr>
</tbody>
</table>

above. Finally, we apply the same synthesis technique to analyze the data/control dependencies between these loop-body blocks and to convert the loop statement into an HDL structural tree. Currently, we only support for and while statements and we assume that the loop count is known.

The multi-level synthesis method is shown below:

**Procedure Multi-Level_Synthesis(V_{fix})**
begin
Import a Verilog HDL description;
Create a top-level structural tree;
for (each process node) begin
    Determine the I/O ports of the process;
    Determine the basic-statements;
    Form a subtree for each basic-statement;
    Construct CDFGs;
    Perform control and data flow analysis;
    Form a subtree for the process;
    Generate logic equations for leaf nodes;
endfor
Perform FPGA synthesis for each leaf node;
Return the HDL structural tree;
end

4 Implementation and evaluation

We have implemented the multi-level synthesis method in the C programming language within the Quick_ECO [8] emulation-based design system. Quick_ECO is an interactive design system supporting on-line RTL engineering changes. Presently, the system runs on SUN and HP workstations.

We have conducted the following experiments to evaluate the proposed multi-level synthesis method. First, we selected 10 modules from industrial designs which contain a wide variety of designs with different HDL coding styles. The number of lines in our Verilog descriptions ranges from 31 to 839. In our experiments, we targeted the Xilinx 4000 series technology.

Table 1 shows the number of nodes generated at different levels. #Lines, #M, #P, #S, #O, and #B represent the number of lines of Verilog descriptions, the number of module, process, statement, operation, and bitwise nodes, respectively. The results show that the multi-level synthesis method is able to establish direct links between HDL descriptions and their subdatat at five levels, namely the module, process, statement, operation, and bit level. This shows that the multi-level synthesis method provides a finer granularity of designs for HDL debugging.

Table 2 shows the resulting circuit sizes at different levels. The results in Table 2 show that the circuits generated at finer-granularity level used more CLBs than that produced at higher level. For example, the circuit size of design 10 is 78 CLBs when it provides links between generated circuits and HDL description at module level. The circuit size will grow to 148 and 331 CLBs when it provides links at statement and bit levels, respectively. This overhead is due to the inability of optimizing logic across boundaries of finer HDL blocks. Table 2 also shows the synthesis run-time at different levels. The run time was measured on a SPARC20 workstation. The results show that the average run time at finer-granularity level is longer than that at higher level.

5 Conclusions

In this paper, we have presented a multi-level synthesis method supporting HDL debugging for emulation-based designs. Using the multi-level synthesis method, we are able to establish direct links between the constructs in an HDL description and their corresponding circuit designs. This feature permits designers to match an incorrectly functioning subdatat with its HDL code.

In order to provide a direct link between HDL descriptions and their corresponding subdatat lists, we have to pay a circuit overhead price which may be very high in some designs. In such cases, designers may need to exercise tradeoff between circuit overhead and design granularity (e.g., selecting process level instead of statement level) for their HDL debugging.

References


