Optimizing Xilinx designs through primitive instantiation

Andreas Ehliar
Department of Electrical Engineering
Computer Engineering
SE-58183 Linköping
ehliar@isy.liu.se

ABSTRACT
This paper is intended as a guideline for people who are interested in manual instantiation of FPGA primitives as a way of improving the performance of an FPGA design. The focus of the paper is on designs where slice primitives like flip-flops and lookup tables are instantiated. Guidelines on how to develop a design with manual instantiation are presented together with a case study of a high performance bitserial two's complement divider where a majority of the area is manually instantiated. This divider is capable of reaching a maximum frequency of 345 MHz in the fastest Virtex-4 while utilizing less than 150 LUTs thanks to the high amount of manual optimizations. An open source library containing modules intended to promote the structured development of modules with manually instantiated components is also presented.

Categories and Subject Descriptors
B.6 [Logic Design]: Miscellaneous.

Keywords
FPGA, Primitive instantiation.

1. INTRODUCTION
Todays RTL synthesis tools are powerful tools that vastly increase the productivity of designers. In most cases the tools are more than adequate, but unfortunately there are situations where it is necessary to tweak the design in order to achieve the desired results. This can be done in various ways. One way is simply to rewrite the RTL code slightly until the synthesis tool infers the desired components. A more powerful way is to use synthesis attributes, such as KEEP, to force the synthesis tool into certain synthesis patterns.

In some cases though, it may be more productive for a designer to simply instantiate FPGA primitives by hand, instead of trying to coax the synthesis tool into producing the desired result. This has the added advantage of making the design less dependent on the synthesis tool, as the same primitives will be instantiated regardless of synthesis options, tool versions, etc. The disadvantage is of course that the design will become less portable and harder to maintain. Nevertheless, manual instantiation of a few critical parts of a design may be the only way to reach the desired operating frequency. Another valid reason to use manual instantiation is to reduce the area, for example by making sure that a BlockRAM is used instead of distributed RAM. Manual instantiation can also be used to save power in some cases (e.g., instantiating a BlockRAM manually so that the enable signal to the BlockRAM can be controlled). Finally, manual instantiation can be used to work around synthesis bugs where the tool infers the wrong logic.

This paper is intended as an introduction on how to handle primitive instantiation in an efficient manner through a case study of a bit-serial two's complement divider optimized for Xilinx FPGAs. More specifically, it is intended to discuss the instantiation of slice related primitives such as LUTs, flip-flops, and carry-chains, although parts of the discussion is also relevant to primitives like BlockRAMs and DSP blocks.

2. RELATED WORK
The author is not aware of any published guidelines for effective use of instantiated slice primitives, hence this paper. However, there are other resources of interest to someone who is manually instantiating slice primitives.

First of course is the datasheet for the FPGA, particularly those parts that discuss the organization of the CLBs and slices. For Xilinx users, the appropriate library guide is also required reading, such as for example the library guide for Virtex-6 [1]. For users of Altera FPGAs, there is a document describing how to use low level primitives as well, although this document also contains little information about how to do this effectively [2].

Another interesting study for any reader interested in manual optimization of FPGA design is [3], where the advantages and drawbacks of manual floorplanning using RLOC directives are discussed. (Although the advantages are probably reduced today due to improvement in placement and routing algorithms.)

3. EFFICIENT SLICE USAGE
The key to creating an efficient FPGA design is to ensure that it maps efficiently to the slices in the FPGA fabric. This is especially important for structures with carry-chains such as adders, subtracters, and comparators.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
FPGA World 2010, Sept. 6, Copenhagen, Denmark
Copyright © 2010 ACM ISBN 978-1-4503-0481-8/10/09... $10.00.
Copyright © 2011 by ASME
3.1 How adders are implemented in Xilinx FPGAs

In order to understand how adders and subtracters work, it is necessary to review how the carry-chain works in Xilinx devices. Normally a full adder is implemented as shown in equation 1 and equation 2, where \( a_i \) and \( b_i \) are inputs, \( r_i \) is the result, and \( c_{out} \) and \( c_{in} \) represents the carry out and carry in.

\[
r_i = a_i \oplus b_i \oplus c_{in} \quad (1)
\]
\[
c_{out} = a_i b_i + C_{in} (a + b) \quad (2)
\]

However, when implementing a full adder in Xilinx devices, equation 2 is rewritten as shown in equation 3.

\[
c_{out} = (a_i \oplus b_i) C_{in} + (a_i \oplus b_i) a \quad (3)
\]

This rewrite allows the term \((a_i \oplus b_i)\) to be reused for both the calculation of \( r_i \) and \( c_{out} \). How this is actually implemented in Xilinx devices with 4-input LUTs is shown in Figure 1. (The \( g \) component is described in Section 3.2.)

3.2 Efficient adders in 4-input LUT devices

When creating high speed logic it is always desirable to limit the logic depth. This is especially important when creating adders in FPGAs due to the relatively long critical path of adders. However, as shown in Figure 1, a plain adder only uses two of the four LUT inputs. This means that it is possible to combine an adder with other logic without any increase in the critical path, as long as no more than four LUT inputs are required. In general, this addition can be described as in equation 4:

\[
result = f(x_0 , x_1 , x_2 , x_3 ) + g(x_0 , x_1 ) + c_{in} \quad (4)
\]

where \( c_{in} \) is the first carry in, and \( f(x_0 , x_1 , x_2 , x_3 ) \) is any bitwise logical operation of \( x_0 , x_1 , x_2 , \) and \( x_3 \). Finally, \( g \) is limited to either a constant value, \( x_0 \), or the bitwise and of \( x_0 \) and \( x_1 \). (The last option uses the so called MULT_AND primitive.) See also Figure 1 for a view of where the \( g \) function is located in a slice.

While the number of different functions that can be implemented using equation 4 is very large, a designer will usually do quite well when considering only the two functions shown in Listing 1.

```
Listing 1: The most useful combinations of adders and other logic in 1 LUT / bit

always @* begin // Adder/subtracter
  if (x_2) result_addsub = x0 - x1;
  else result_addsub = x0 + x1;
  // 2-to-1 multiplexer and adder
  if (x3) result_addmulx = x0 + x1;
  else result_addmulx = x0 + x2;
end
```

However, in some situations a designer may be better off using another structure allowed by equation 4. For example, if an ALU is designed, the MULT_AND primitive can be used to enable the carry-chain only when needed, as shown in Listing 2. Another sometimes useful function is shown in Listing 2 where one input is added to the bit-wise logical or of \( x_1 \), \( x_2 \), and \( x_3 \). While seemingly an odd operation to do, this can be very useful if these signals are all direct outputs from registers. In that case the reset input of the flip-flops can be used to disable all but one signal, essentially creating the equivalent of a 3-to-1 multiplexer. This is very useful in datapaths, and is heavily used in for example Nios II [4].

```
Listing 2: Examples of other useful combinations of adders and other logic in 1 LUT / bit

always @* begin // Using MULT_AND to disable // the carry-chain
  if (x0) result_multand = x1 + x2;
  else result_multand = x1 | x2;
  // Adder combined with bit-wise or // of 3 inputs.
  result_or = x0 + (x1 | x2 | x3);
end
```

3.3 Efficient adders in 6-input LUT devices

In 6-input LUT devices such as the Virtex-5, Virtex-6, and Spartan-6, the carry-chain structure is more flexible as the input to the carry-chain is set through the \( O5 \) output of the LUT. This is a big improvement compared to previous devices as the input to the carry-chain is now under total user control. In essence, an adder in a 6-input LUT device can be described by equation 5:

\[
result = f(x_0 , x_1 , x_2 , x_3 , x_4 , x_5 ) + g(x_0 , x_1 , x_2 , x_3 , x_5 ) + c_{in} \quad (5)
\]

where both \( f \) and \( g \) are general bitwise logical functions. Contrast this to a 4-input device (equation 4), where the input to the carry-chain is much more limited. This allows a wide variety of useful functions to be implemented using only one LUT / bit, such as an adder with a 2-to-1 multiplexer in front of each input to the adder or an adder/subtractor where either the first or the second operand can be negated.
3.4 Using the carry-chain for comparisons

Another good use of the carry-chain are as a part of an efficient comparator. An example of this is shown in Figure 2, where the two incoming vectors are compared for equality two bits at a time. This circuit functions by propagating the carry as long as both inputs are equal and forcing the carry to 0 once at least one pair of bits are unequal.

It is possible to create an even more area efficient comparator if one of the inputs is a constant. In this case the constant can be encoded in the LUT content, leading to a comparator that uses only one LUT for every 4 bits. Another intriguing possibility is to configure the LUT as either a distributed RAM or a shift register (SRL16), which will allow the constant comparison value to be changed dynamically, paving the way for a very area efficient dynamically configurable comparator.

It could also be noted that the carry-chain is also perfectly suited for wide and or or operations.

3.5 Efficient multiplexers

Another important optimization opportunity is wide multiplexer-like structures. In 4-input LUT based FPGAs from Xilinx, a 2-to-1 multiplexer can easily be implemented using only one LUT. Larger multiplexers (up to 16-to-1) can use the MUXF5-MUXF8 slice components that allow larger multiplexers to be created in a relatively area efficient way. An example of how the MUXF5 slice component can be used to combine the output of two LUTs is shown in Figure 3.

Sadly, the use of MUXFx primitives inhibits the use of the XORCY primitive, which makes it impossible to combine an adder with a large multiplexer without resorting to multiple levels of logic.

\[1\] While the MUXF5 primitive significantly eases the task of creating a 4-to-1 mux, it is actually possible to create a 4-to-1 mux using only two 4-input LUTs without any additional primitives by using an elegant circuit described in [4].
5. ALTERNATIVES TO MANUAL INSTANTIATION: SYNTHESIS ATTRIBUTES

While primitive instantiation is a powerful tool, there are often other ways to achieve the same result. One of the more powerful options is to use one of the many synthesis attributes. These can often be used to guide the synthesis tool into making the correct implementation decisions. Some of the more useful synthesis attributes for XST are:

- **KEEP**: This directive forces a signal to be excluded from optimization so that it is guaranteed to be left in the final design.
- **KEEP_HIERARCHY**: This directive can allow or prohibit the synthesis tool from making cross module optimizations.
- **SIGNAL_ENCODING**: This allows the designer to control the encoding of the state in a state machine (e.g., one-hot coding, gray coding, etc). Setting this to "user" makes sure that the synthesis tool keeps the encoding specified by the user which is very useful in those cases where the synthesis tool makes a bad encoding decision.
- **EQUIVALENT_REGISTER_REMOVAL**: This allows the designer to control whether identical flip-flops should be combined or not. A typical use case for disabling this optimization is when the designer is splitting a high-fanout signal into several different nets.

6. GUIDELINES FOR PRIMITIVE INSTANTIATION

This section contains a small set of common sense guidelines that the author has found useful when dealing with FPGA designs that are heavily optimized for a specific FPGA family. While the author has limited experience with FPGAs from vendors beside Xilinx, the guidelines are general enough to make sense for other FPGAs as well.

**Guideline 1**: The most obvious guideline is to avoid this kind of optimization if not absolutely necessary. Or, as Donald Knuth eloquently puts it (albeit in the context of software): "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil."[5] Consider whether there is some other way to achieve the desired synthesis result, for example by using synthesis attributes as discussed in Section 5.

**Guideline 2**: As a corollary to the first guideline, identify the locations that can benefit the most from manual instantiation by studying the map report (to find area inefficiencies) and the timing reports. Even in a large design there may be opportunities for significant savings through a limited use of manual primitive instantiation.

**Guideline 3**: When manual instantiation is used, always document why it had to be used. This makes sure that the manually instantiated parts can be removed at a later point if the reasons for its existence are no longer true. (Such as when a new and improved synthesis tool is able to synthesize the original RTL code in a more optimal fashion.)

Listing 5: Example of FPGA specific and generic code with a checker

```verilog
module example1(input wire clk, rst, a, b, output wire result);
reg result_gen;
wire result_inst;

// Generic code
always @(posedge clk) begin
  result_gen <= a & b;
  if(rst) result_gen <= 0;
end

// Rationale for Xilinx specific code:
// The reset input of the flip-flop is quite slow compared to the D input. Unfortunately the synthesis tool will sometimes use this input suboptimally. The code below makes sure that the reset input is only used by the reset signal.
FDR ff (.C(clk), .D(a & b), .R(rst), .Q(result_inst));

// Sanity checker:
// Even though both the generic version and the Xilinx specific version are both included at the same time in the source code, there should be no extra area cost in the synthesized output as the synthesis tool will remove unused logic.
always @(posedge clk) begin
  if (result_inst !== result_gen) begin
    $display("%m: Sanity checker failed.");
    $stop;
  end
end
```

**Guideline 4**: If manual instantiation is used, it is a very good idea to also include a portable version of the same hardware and use either generics (VHDL), parameters (Verilog), or defines (Verilog) to determine whether the output of the portable version or the instantiated version is used. This will simplify debugging and module maintenance. A careful designer could also use some sort of assertions to discover differences in the portable and instantiated version as shown in Listing 5. An exception for this guideline is when it is not feasible to write synthesizable code in a generic manner for the component’s functionality, such as when instantiating a DCM or another complex component.

**Guideline 5**: When manually instantiating more than one
The components named "Shift" implement the following concatenation: \( \text{out} = \text{result} \ll 1 \mid \text{nextbit}; \)

![Diagram](image)

**Figure 4:** Restoring divider before and after retiming. Note that the marked components will occupy two LUTs / bit in a) whereas the marked components only occupy one LUT / bit in b). All registers also have a clock enable input, but this is not shown in the figure.

7. **CASE STUDY: A 32-BIT TWO’S COMPLEMENT DIVIDER**

This section will discuss how to optimize a bit-serial divider for Xilinx FPGAs by manual instantiation of FPGA primitives. This case study shows how the performance of a design can be improved by merging adders with related logic as described in Section 3. The general architecture of a restoring divider is shown in Figure 4a. While the divisor and dividend are 32 bits wide, the divider actually uses 33-bit wide numbers internally to avoid corner-cases for large values.

While this example is targeted at a Virtex-4, the techniques will apply equally well to other contemporary Xilinx devices with 4-input LUTs as well. The divider has not been designed for 6-input LUTs (although the divider should still be quite efficient on such architectures).

Note that the shift register that holds the dividend can also be used to hold the quotient (one dividend bit is consumed for every generated quotient bit). Note also that the box called "Shift" does not consume any actual hardware as this is merely a concatenation of different signals.

The architecture in Figure 4a is suboptimal in Xilinx FPGAs however, as the marked components in Figure 4a (the adder and the multiplexer) will occupy two LUTs per bit, leading to a critical path which is longer than necessary. It would be better if the multiplexer was situated before the adder instead of after it. Fortunately this can be achieved by retiming the result register as shown in Figure 4b. However, when retiming the register like this it is also necessary to change the subtracter into an adder. The divisor is then inverted using an additional layer of LUTs in front of the “inverted divisor” register, and the carry in is set to 1, hence creating a subtracter.

7.1 **Persuading the synthesis tool to actually merge the multiplexer and adder**

While Figure 4b is a good architecture, actually implementing this architecture using plain Verilog without any instantiation was harder than expected. It turns out that the synthesis tool realizes that both multiplexers in Figure 4b are identical and merges both multiplexers into one. This negates all benefits of the retiming performed when going from Figure 4a to Figure 4b.

Although it is possible that some combination of synthesis attributes and rewriting of the RTL code could have solved this situation, after a few hours of searching for the correct combination it was deemed easier to manually instantiate this part of the design.

7.2 **Dealing with negative numbers**

The architecture shown in Figure 4 is only capable of dealing with unsigned numbers. While there are a wide variety of algorithms available for binary division capable of two’s complement division, the author suspects that many of these would be difficult to map efficiently to the Virtex-4. Fortunately it turns out that a simple, almost naïve, solution can be implemented very efficiently in a Virtex-4.

By taking the absolute value of both the dividend and divisor as a pre-processing step, the core of the divider will only need to care about positive numbers. After a division is finished, the quotient and remainder can be negated (if necessary) as a post processing step.

\(^{2}\) Readers unfamiliar with the workings of a bit-serial restoring divider, are referred to a textbook on computer architecture or computer arithmetic for more information.

\(^{3}\) As shown in equation 4, one of the inputs to an adder using 1 LUT / bit has to be unmodified (or a bit-wise and of two inputs), and this means that it is not possible to invert the divisor inside the slice if a 2-to-1 multiplexer should be present in the same LUT.
At first glance, this would seem to require 128 LUTs (32 LUTs for each of the dividend, divisor, quotient, and remainder respectively). However, it turns out that this functionality can be merged into the LUTs already present in the divider. This is easiest to see for the divisor where the 32 LUTs used to invert the input can easily be used to take the absolute value of divisor before inverting it.

In Figure 4, the shift register has a total of 3 inputs; load enable, the dividend input, and the current value of the shift register (the shift enable signal is handled by the clock enable input of the shift register). This means that the final output could be used as a negate enable signal and could thus be used to negate both the dividend and the quotient without any added area cost. Behavioral Verilog code for the final version of the shift register is shown in

Finally, if the remainder is used it may also necessary to be able to correct the sign of this output. This can be done by combining a negate-circuit with the left multiplexer shown in Figure 4b. Thus, a two’s complement bit-serial divider can be implemented using basically the same amount of hardware as an unsigned divider without any loss of performance.

7.3 Performance and area

A comparison of the performance of this divider with a few other implementations is shown in Table 1. The divider presented in this paper is compared with four other dividers from a recent publication [8]. This publication does not contain any information about the number of LUTs and flip-flops, but fortunately the authors have additional data about their work available online [6] which includes these numbers.

As can be seen, the maximum frequency of the divider presented in this paper is significantly higher thanks to the extensive FPGA optimizations while at the same time being area efficient. Also note that the divider described in [8] is only capable of unsigned division. However, in fairness to the authors of [8], the latency of a division is significantly shorter in the R4_a2 and R8_a2 dividers even when compensated for clock frequency.

8. A LIBRARY FOR SLICE PRIMITIVE INSTANTIATION

When developing a design with a relatively high percentage of manually instantiated components, such as the divider one described in the previous section, it is very important to use a structured development method (something the author has become painfully aware of after years of non-structured and ad-hoc usage of FPGA primitives). The author has therefore developed a library to support the development of such designs based on the fact that the majority of primitive instantiations seems to fall into one of four categories in the author’s experience:

- Individual components such as a single flip-flop or a single LUT
- Registers of various widths
- Carry-chain based structures
- Multiplexer (MUXF5, MUXF6, . . . ) based structures

The first case has seldom been a big issue problem in the author’s experience. Handling a single instantiated LUT or flip-flop specially is relatively straight forward for a person tasked with module maintenance. Even for a person unfamiliar with Xilinx primitives, it should be easy to read the documentation for a component such as LUT4 or FDRE and understand how they work.

For the remaining cases, a library has been written that will be available for download at http://www.da.isy.liu.se/~ehliar/ae_instlib/ under the MIT open-source license [9], which is an open source license which allows both open and closed-source usage scenarios. The divider source code will also be included as an example of how to use the library.

8.1 Handling register instantiation

Registers of various bit widths are fairly easy to handle and understand, but it makes sense to create a parameterized module to allow registers to be created using different bit widths without having to write a generate loop at each location where a register is needed. A typical use case for this

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Frequency</th>
<th>LUTs</th>
<th>FFs</th>
<th>#cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>This work</td>
<td>357 MHz</td>
<td>148</td>
<td>146</td>
<td>35</td>
</tr>
<tr>
<td>non_rest_1 [6]</td>
<td>263 MHz</td>
<td>389</td>
<td>236</td>
<td>33</td>
</tr>
<tr>
<td>non_rest_2 [6]</td>
<td>156 MHz</td>
<td>402</td>
<td>234</td>
<td>17</td>
</tr>
<tr>
<td>R4_a2 [6]</td>
<td>263 MHz</td>
<td>239</td>
<td>232</td>
<td>16</td>
</tr>
<tr>
<td>R8_a2 [6]</td>
<td>189 MHz</td>
<td>470</td>
<td>298</td>
<td>16</td>
</tr>
<tr>
<td>OpenCores [7]</td>
<td>197 MHz</td>
<td>237</td>
<td>165</td>
<td>32</td>
</tr>
</tbody>
</table>

Listing 6: Example of register instantiation

```
reg [4:0] next_state;
wire [4:0] state;
always @* begin
    next_state = state;
    case(state)
        5’d0: begin
            next_state <= 1;
            // lots of other code
        endcase
        default: begin
            if(a == 5’d0) next_state <= 5’d0;
            else next_state <= 5’d1;
        endcase
end
```
Listing 7: Instantiating the merged multiplexer and adder

```
// Inferring the multiplexer and adder always @ (i_in) begin
if (msb) begin
    next_remainder = inv_divisor + remainder, next_bit;
end else begin
    next_remainder = inv_divisor + new_remainder, next_bit;
end
end
```

// Fully manual method of implementing a carry- // chain by using ae_instlib carrylogic Comb Jul #(.WIDTH(33),

```
VALIDATE (16'b00110100111001010),
```

merged_adder

```
(. carryin(1), . carryout(),
 . result (next_result [31:0]),
 . carrychain_in (inverted_divisor [32:0]),
 . divisor [32:0]),
 . divisor [32:0])
```

Listing 8: Using the library to instantiate a large

```
// Semi-manual method of creating a carry− // chain by using ae_instlib where the // LUT content is inferred while the // carry−chain is instantiated
wire [32:0] tmp;
assign tmp[0] = next_bit ’inv_divisor [0];
assign tmp[31:1] = inverted_divisor [32:1] ’
```

```
msb ? remainder [31:0] :
```

```
new_remainder [31:0];
```

```
carrylogic_comb_nolut #(.WIDTH(33))
```

```
merged_adder_and_2_to_1_mux
```

```
(. carryin(1), . carryout(),
 . result (next_remainder [31:0]),
 . carrychain_in (inverted_divisor [32:0]),
 . lut_to_carry (tmp));
```

A carry-chain based function can basically be written in three different ways. The easiest for the designer is simply to let the synthesis tool infer the components. In contrast to this, the most cumbersome way is to manually instantiate all components manually and specify the content of the LUTs by hand. Finally, a semi-manual method can sometimes be used to good effect when the user manually instantiates the MUXCY and XORCY primitives, while the synthesis tool infers the content of the LUT.

Listing 8: Using the library to instantiate a large multiplexer like structure

```
// Semi−manual approach
```

```
mux (IO (a [31:0] | b [31:0] | c [31:0]),
 .11 (ctrl ? 32'b0:
 (S1 [d [31:0]:e [31:0]),
 .SO(SO),
 .result (result));
```

```
// Fully automatic approach
```

```
mux (IO (a [31:0]),
 .11 (b [31:0]),
 .12 (c [31:0]),
 .13 (32'b0),
 .14 (d [31:0]),
 .15 (e [31:0]),
 .16 (32's1)),
 .17 (32's1),
 .SO(SO),
 .result (result));
```

8.3 Multiplexer based structures

Similarly to the carry-chain based structures described in the previous section, multiplexer based structures can be created in different ways. A semi-manual approach where only MUXFx components are instantiated and the remaining parts are inferred by the synthesizer is shown in Listing 8 together with a fully manual approach where the LUT content is specified by using the INIT0 and INIT1 attributes.

```
```

scenario in Xilinx devices is when the synthesis tool makes a suboptimal choice as to which signal should be connected to the (relatively slow) reset input of the flip-flop. In this case it may be necessary to manually instantiate the flip-flops in order to select which signal to use as the reset input (if any). An example of this usage is shown in Listing 6.

8.2 Carry-chain based structures

Manual instantiation of components related to carry-chains such as adders and subtracters is one of the most common optimizations the author has done over the years. The usual reason for doing this is that the synthesis tool is unable to map the RTL code into the desired hardware configuration. Either too much hardware is used (e.g. two LUTs per bit instead of one LUT per bit) or the critical path is poorly managed.

In the divider case study, all three methods were used. The first method is used for the relatively simple case of calculating the inverted absolute value of the divisor. The fully manual method was used to ensure that the (optional) negation of the remainder could be merged into the left multiplexer of Figure 4b.

Finally, the semi-manual approach was used for both the merged adder/multiplexer and the merged shift register/negator in the divider.) An example of both the fully manual and semi-manual method is shown in Listing 7 together with code which infers the same behavior for reference.
8.4 Discussion
The ability to manually instantiate primitives is certainly a powerful tool, and the author has used it to great effect in a number of designs. An example of such a design is a high speed 32-bit processor capable of operating at 357 MHz in a Virtex-4 of speedgrade 12 [10]. This operating frequency could never have been reached without a high degree of manual optimizations. (Around 11% of the flip-flops and 18% of the LUTs are manually instantiated in this design.) This shows that it is certainly feasible to use the techniques described in this paper in a relatively large design to good effect. However, this doesn't change the fact that manual instantiation is certainly a pretty time consuming endeavor, even though the instantiation library described in Section 8 has helped somewhat in this regard, especially in terms of maintainability.

Another issue which has not been discussed in detail in this paper is portability, especially in regards to ASICs. If guideline 4 is violated, as it is possible to port a design with manually instantiated FPGA primitives like carry-chains and multiplexers through the use of a small compatibility library containing synthesizable code for FPGA primitives like LUT1…LUT4, MUXCY, and XORCY. In fact, a good synthesis tool is able to synthesize an adder consisting of instantiated LUTs and carry-chain primitives almost as efficiently as if the adder was inferred from the + operator in Verilog [11].

8.5 Future work
The library has been very useful for the author, but even so there are a few obvious improvements that are possible. An interesting possibility is to investigate whether it makes sense to use a similar approach on non-Xilinx FPGAs such as FPGAs from Altera.

Another important feature which would be quite easy to implement is a VHDL version. (Although the Verilog version should be fully usable from VHDL, assuming a synthesis and simulation tool capable of handling Verilog/VHDL co-simulation is available.)

9. CONCLUSIONS
Manual instantiation of FPGA primitives is a powerful tool that can improve the performance of an FPGA design significantly if used correctly. An example of this is the divider described in Section 7 which, thanks to the manual instantiation of FPGA primitives, is able to reach a throughput / area figure which is significantly higher than other FPGA based dividers to the best of the author’s knowledge.

While manual instantiation is a relatively costly development paradigm, a primitive library such as the one presented in Section 8 will reduce this cost by standardizing the interface to manually instantiated carry-chains and multiplexers.

It is the hope of the author that the guidelines and techniques described in this paper can serve as an inspiration both to readers familiar and readers unfamiliar with primitive instantiation.

Acknowledgments
Many thanks to Daniel Källmning for doing a lot of implementation work on a bit-serial divider based on the architecture described in Section 7.

10. REFERENCES
   http://www.arithmetic-circuits.org/FixedPointDivision/FixedPointDivision.html.
   http://opencores.org/project,serial_div_uu downloaded on 2010-08-10.
   http://www.opensource.org/licenses/mit-license.php.