Booth Radix-4 Multiplier for Low Density PLD Applications (VHDL)

Features

The following topics are covered via the Lattice Diamond ver.2.0.1 Design Software.

• Overview of the Booth Radix-4 Sequential Multiplier
• State Machine Structure and Application of Booth Algorithm
• Booth Radix-4 Word-Width Scalability
• Testing the Multiplier with a Test Bench

Introduction

This VHDL module uses a simple 2-state finite state machine (FSM) to evaluate groupings of 3 bits held in a product register and chose one of five possible operations based on those groupings. The state diagram for this 2-state FSM is found below in Figure 1. This 3-bit recoded shift and add process is known as the Booth algorithm. The version used in this module is known as the Booth Radix-4 multiplication algorithm.

![Figure 1 – Booth Radix-4 FSM State Diagram](image)

Background

The Booth Radix-4 algorithm reduces the number of partial products by half while keeping the circuit’s complexity down to a minimum. This results in lower power operation in an FPGA or CPLD and provides for multiplication when no hard multipliers are otherwise available such as in a Lattice MachXO2 PLD which was used in this example. Booth Recoding makes these advantages possible by skipping clock cycles that add nothing new in the way of product terms. The Radix-4 Booth Recoding is simply a multiplexor that selects the correct shift-and-add operation based on the groupings of bits found in the product register. The product register holds the multiplier. The multiplicand and the two’s complement of the multiplicand are added based on the recoding value. The recoding is found in Table 1 below. An example multiplication can be found in Figure 2 below.
### Table 1: Radix-4 Booth Encoding

<table>
<thead>
<tr>
<th>Select Line (encoding)</th>
<th>Partial Products (operation)</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>add 0</td>
</tr>
<tr>
<td>001</td>
<td>add multiplicand</td>
</tr>
<tr>
<td>010</td>
<td>add multiplicand</td>
</tr>
<tr>
<td>011</td>
<td>add 2(^{st}) multiplicand</td>
</tr>
<tr>
<td>100</td>
<td>subtract 2(^{st}) multiplicand</td>
</tr>
<tr>
<td>101</td>
<td>subtract multiplicand</td>
</tr>
<tr>
<td>110</td>
<td>subtract multiplicand</td>
</tr>
<tr>
<td>111</td>
<td>subtract 0</td>
</tr>
</tbody>
</table>

### Problem

- **Multiplicand (mc)**: 00001111<sub>2</sub> → 15<sub>10</sub>
- **Multiplier (mp)**: 00001001<sub>2</sub> → 9<sub>10</sub>

\[
\frac{15_{10} \times 9_{10}}{} \rightarrow 135_{10}
\]

---

**Booth Encoding**

<table>
<thead>
<tr>
<th>Encoding</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
</tr>
<tr>
<td>001</td>
<td>+mc</td>
</tr>
<tr>
<td>010</td>
<td>+mc</td>
</tr>
<tr>
<td>011</td>
<td>+2mc</td>
</tr>
<tr>
<td>100</td>
<td>+2mc</td>
</tr>
<tr>
<td>101</td>
<td>-mc</td>
</tr>
<tr>
<td>110</td>
<td>-mc</td>
</tr>
<tr>
<td>111</td>
<td>0</td>
</tr>
</tbody>
</table>

**Operations determined by groupings of 3-bit encodings**

- Bit-width divided by 2 (6-bit/2 = 4 cycles)

- **shr**: Shift register
- **shr and sign extend**: Shift and sign extend

---

**Note:** Encodings are determined by results after each line on the right and not simply by initial product register content. Above example is to illustrate how the groupings work.
The Software required/used for this design:  

Application

Building the Circuit

The Booth Radix-4 multiplier can be scaled from 4 bits up in even values such as 6, 8, 10... The user is limited by the logic density and speed of the PLD. Larger word widths require larger circuits with longer propagation delays. This being said larger circuits will require a slower clocking. A 6-bit multiplier was benchmarked at 135 MHz in a MachXO2, while an 18-bit multiplier was able to run at 125 MHz.

The design has five input ports ("clk", "n_reset", "start", "mcand", and "mplier") and two output ports ("done" and "product"). The multiplier requires a start pulse to initialize the FSM with values from the "mcand" and "mplier" inputs and put the FSM in the “BUSY” state. Math steps can be sequenced by using the “start” and “done” signals between instantiations.

The RTL diagram for an 18-bit implementation can be found in Figure 3 below.

The included test bench was created from the “generate test bench template” command in the “HDL Diagram” window. Inspect the “booth_mult_tb.vhd” file by reading the VHDL comments for understanding. Aldec Active-HDL waveforms for 8-bit and 64-bit implementations can be found below in Figures 4 and 5.
Figure 4 – Active-HDL Test Bench Output for 8-bit Implementation

Figure 5 – Active-HDL Test Bench Output for 64-bit Implementation

Design Software

Lattice Diamond Design Software version 2.0.1 was used to develop the “booth_mult.vhd” with supporting software from Synopsis (Synplify Pro for Lattice) and Aldec (Active-HDL Lattice Edition). Diamond can be used as a stand alone development environment with alternative synthesis and simulation software.

Conclusion

Additional Information

Further design support, product tutorials, application notes, users guides and other documentation can be found on the Lattice website

For questions relating to this reference design, the author can be contacted at:

Appendix

The complete Lattice Diamond project can be downloaded from the Digi-Key, eewiki.net website under the Programmable Logic section. The VHD files are located below.

booth_mult.vhd
booth_mult_tb.vhd

Feedback for Our Sponsor

Please take a few seconds to help us justify the continued development and expansion of the eewiki.

Click on one of our Digi-Key links on your way to search for or purchase electronic components.

Is the eewiki helpful? Comments, feedback, and questions can be sent to eewiki@digikey.com.