

The following commands will test the blanking circuitry:

```
C:\XCPROJ-V\BLANK_40> XSPORT 00110  
C:\XCPROJ-V\BLANK_40> XSPORT 10110
```

The first command should display a numeral 6 on the 7-segment LED. The second command will disable all the drivers and extinguish the LED segments.

Measuring Gate Delay

Figure 5.12 shows a simple circuit, DELAY, that demonstrates gate delays. When INP transitions to a logic 0 value, the outputs should all go to logic 0 after 1 gate delay. Then when INP rises back to logic 1, OUT1 will go high after 1 gate delay. This places a logic 1 on both inputs to the second AND gate so OUT2 will go high 1 gate delay after OUT1 did. And OUT3 will go to logic 1 after another gate delay, followed by OUT4 1 gate delay later.

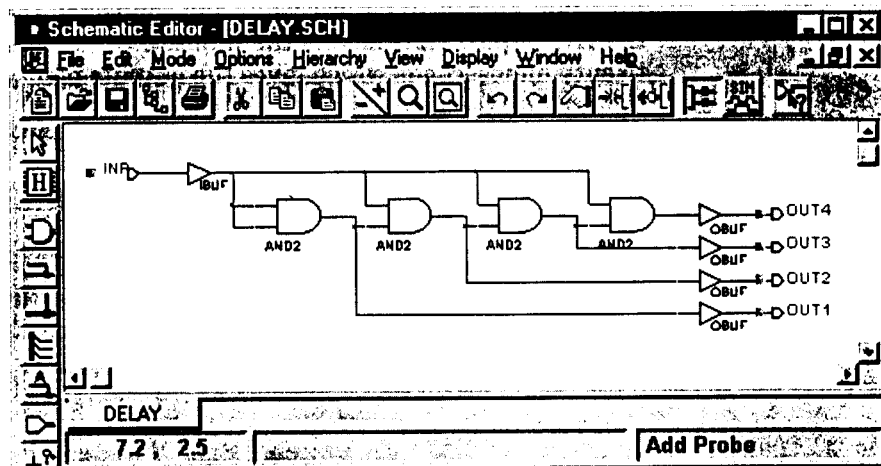


Figure 5.12 A simple circuit for demonstrating propagation delay.

Note the small boxes attached to each input and output terminal. These are probes which monitor the net or terminal they are attached to and send the logic values to the **Waveform Viewer** window in the simulator. They were added as follows:

1. Select the **Mode** → **Test Points** menu item in the **Schematic Capture** window.
2. Click on each net or terminal which you want to appear in the **Waveform Viewer** window of the simulator.

Once the probes are added to the schematic, the signals will appear in the **Waveform Viewer** window as soon as we start the simulator.

A functional simulation of the DELAY circuit is not very interesting - the outputs OUT4..OUT1 move in exact synchrony with the INP input! In order to see the effect of gate delays, click on the drop-down menu for the simulation mode in the toolbar of the **Logic Simulator** window and select **Unit** instead of **Functional**. This puts the simulator in the unit-delay mode where the output of a gate changes 1 time unit after the inputs change.

But how long is a gate delay? That is determined by the simulation precision used by the simulator algorithm. The simulation precision is the smallest increment of time between two non-simultaneous signal transitions. You can set the precision by selecting the **Options** → **Preferences...** menu item in the **Logic Simulator** window. In the **Preferences** window that appears (Figure 5.13), type the desired precision in the **Simulation Precision** box of the **Simulation** tab. I chose 1 nanosecond in this example. I also set the period of the B0 stimulator to 10 nanoseconds and connected the B2 bit of the stimulator to the INP input. That means the input to the DELAY circuit will change every twenty nanoseconds which gives the outputs plenty of time to settle before the input changes again.

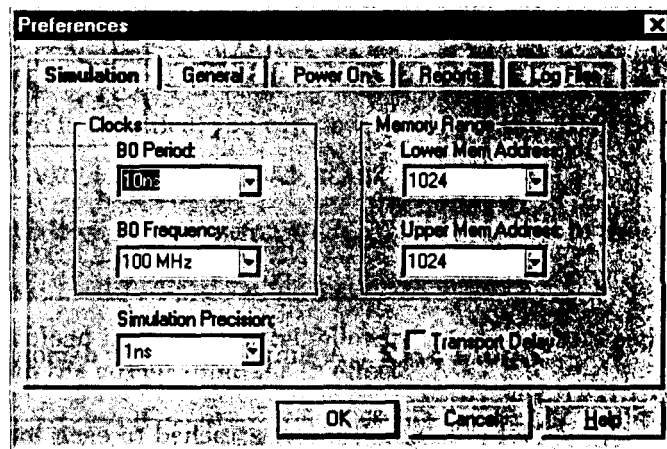


Figure 5.13 Setting the simulation precision.

Once the simulation precision and the input stimulator are set, we can simulate the design and view the effects of unit gate delays. The results are shown in Figure 5.14. As expected, the outputs are each offset by 1 gate delay on a rising transition, and they all fall at the same time on a negative-going transition. But note that OUT1 changes 3 ns (i.e., 3 gate delays) after a signal tran-

sition on the INP input. The 2 additional gate delays arise from the delays incurred by going through the input and output buffers.

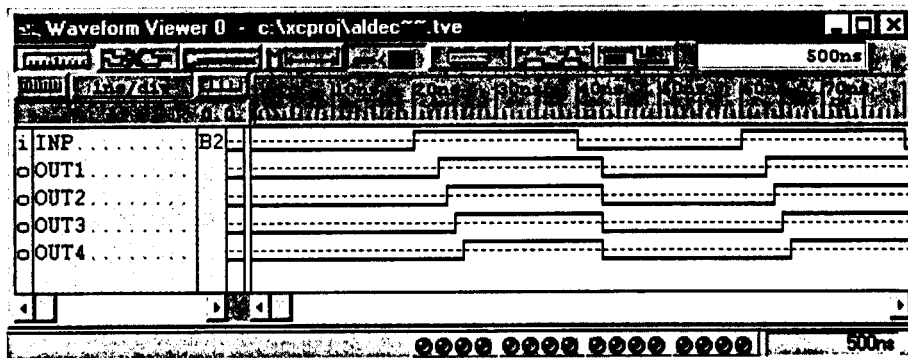


Figure 5.14 Simulated waveforms for the DELAY circuit with unit gate delays.

The actual circuit delays in the DELAY circuit cannot be tested from your PC keyboard or observed using the LED digit. Instead, you have to drive the input with the oscillator on the XS40 or XS95 Boards and observe the outputs with a dual-trace oscilloscope. Use the following pin assignments for the XS40 Board:

NET INP	LOC=P13;	# the oscillator signal enters on pin 13 of the XS40
NET OUT1	LOC=P25;	
NET OUT2	LOC=P26;	
NET OUT3	LOC=P24;	
NET OUT4	LOC=P20;	

And use these pin assignments for the XS95 Board:

NET INP	LOC=P9;	# the oscillator signal enters on pin 9 of the XS95
NET OUT1	LOC=P21;	
NET OUT2	LOC=P23;	
NET OUT3	LOC=P19;	
NET OUT4	LOC=P17;	

After compiling and downloading the DELAY design, attach one channel of the oscilloscope to the pin driven by the on-board oscillator of the XS40 or XS95 Board. Attach the other channel to one of the pins driven by the outputs of the DELAY circuit. After some adjustment of the scope, you should see waveforms similar to those of Figure 5.14.

You can use the simulator with unit-gate delays to observe the delay through the 8-bit adder we built in the last chapter. The maximum delay through the

carry adder chain occurs when one of the operands is 11111111 (0xFF in hexadecimal), the other operand is 00000000 and the carry input flips from 0 to 1. Figure 5.15 shows the waveforms and delays for this case. The carry input goes to a logic 1 at 40 nanoseconds, and the carry output follows at 58 nanoseconds. That is 18 gate delays since the simulation precision was set to 1 nanosecond. From our previous discussions, we calculated this delay to be 16 gate delays, not 18. Once again, the 2 extra gate delays come from the IBUF input buffer for the CIN signal and the OBUF output buffer for the COUT signal.

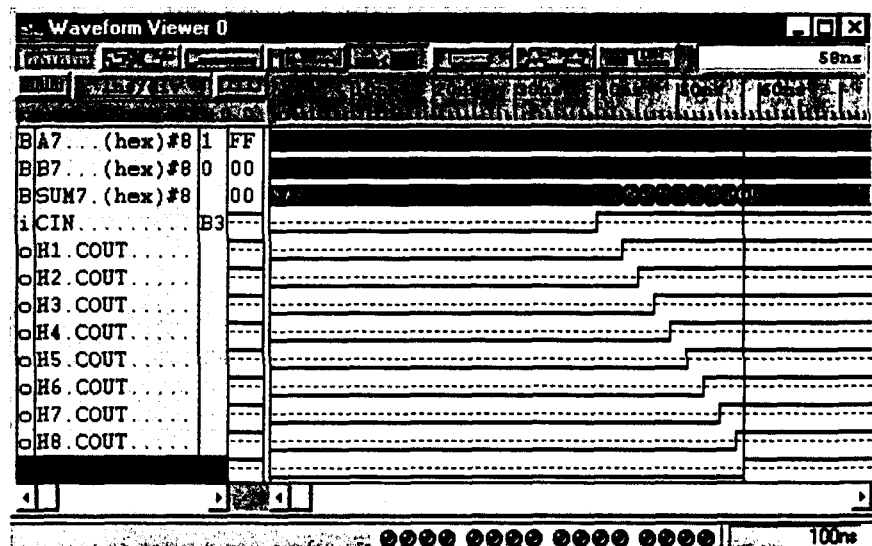


Figure 5.15 Waveforms for a simulation of the maximum ripple-carry chain delay of an 8-bit adder.

You can even examine the delays as the intermediate carry bits flip. This requires you to add signals from lower levels of the design hierarchy to the **Waveform Viewer**. Just select **Signal → Add Signals...** to bring up the **Component Selection for Waveform Viewer** window shown in Figure 5.16. Click on one of the MY_ADD1 adder bits in the **Chip Selection** panel. Then click on the COUT pin in the right-most panel and **Add** it to your **Waveform Viewer** window. Once all the intermediate carry signals are displayed, you can see the cascade effect as they flip from 0 to 1.

This is nice and simple. But in the real-world not all gate delays are equal. And the routing of signals between macrocells or CLBs in an FPLD also contributes to the total delay. In order to simulate these delays, you need to use the **Timing** mode of the simulator. But the Timing mode cannot do an accurate simulation unless it has the gate and routing delay information that comes

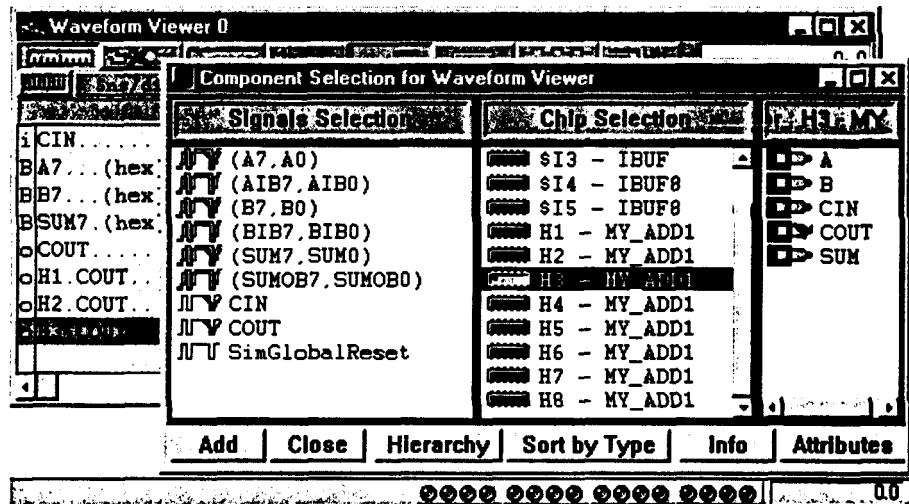



Figure 5.16 Displaying hierarchical signals in the 8-bit adder.

from how your circuit was mapped into the FPLD chip. You can tell the Foundation Series Implementation tools to generate this information for either schematic or HDL mode projects by selecting the **Implementation** → **Options...** menu item in the **Project Manager** window. Then check the **Produce Timing Simulation Data** box in the **Options** window (see Figure 5.17). Then run the Foundation Implementation tool and it will extract the routing and gate delays as it maps the netlist into the FPLD.

Once you have run the Foundation Implementation tools, you can click on the  button in the **Flow** tab of the **Project Manager** window. This brings up the **Logic Simulator** window with the **Timing** mode selected. Other than that, you use the same steps to add signals and do simulations as you did for functional simulations.

The resulting waveforms for the 8-bit ripple-carry adder are shown in Figure 5.18. In this case, we compiled the adder for an XC4005XL FPGA with a -3 speed rating. In the simulation, the CIN input transitions at the 80 ns mark. The COUT output transitions in response at 121 ns. Thus, the delay through the 8 adder bits (including gate, I/O buffer, and wiring delays) is 41 ns. The worst-case combinatorial delays through an XC4005XL-3 CLB range from 1.6 ns to 3.2 ns. That would make the total gate delay through the 8 CLBs of the carry chain between 12.8 ns and 25.6 ns. The remainder of the delay is buffer propagation time and wiring delays.

All of our experiments have been done on ripple-carry adders. The ABEL and VHDL descriptions for a 4-bit carry-lookahead adder are shown in Listings 5.4 and 5.5, respectively. You should be able to see the correspondence between the

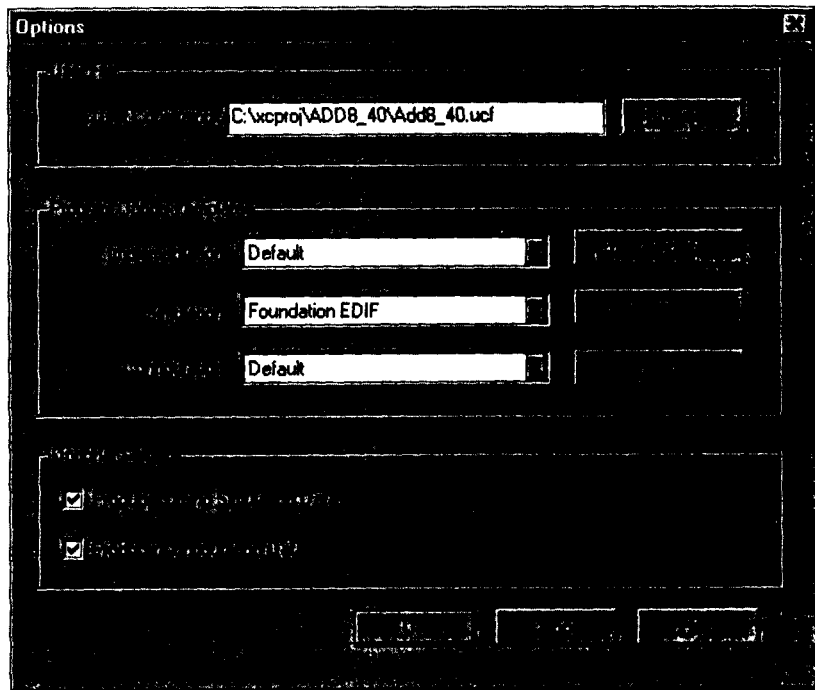


Figure 5.17 Configuring the Foundation Series Implementation tools to produce timing simulation data.

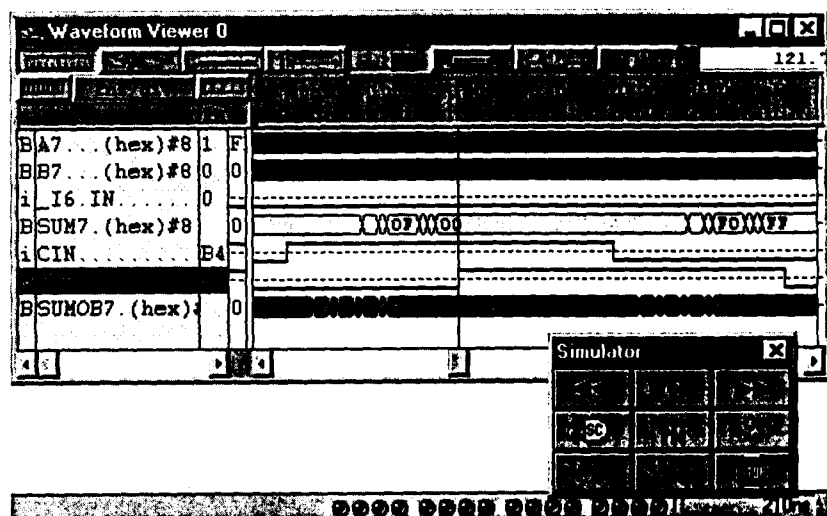


Figure 5.18 An accurate timing simulation of the 8-bit ripple-carry adder in an XC4005XL-3 FPGA.

code and the equations for c_i , G_i , and P_i presented earlier in this chapter. You will compare the propagation delays of the carry-lookahead and ripple-carry adders in the project at the end of this chapter.

Listing 5.4 ABEL code for a 4-bit carry-lookahead adder (LKADD.ABL).

```

001- MODULE lkadd
002- TITLE '4-bit carry-lookahead adder'
003-
004- DECLARATIONS
005-
006- cin PIN;
007- a3..a0 PIN;
008- b3..b0 PIN;
009- sum3..sum0 PIN ISTYPE 'COM';
010- cout PIN ISTYPE 'COM';
011- g3..g0 NODE ISTYPE 'COM'; "carry generate bits
012- p3..p0 NODE ISTYPE 'COM'; "carry-propagate bits
013- c2..c0 NODE ISTYPE 'COM'; "internal carry bits
014-
015- EQUATIONS
016-
017- g0 = a0 & b0;"first adder bit
018- p0 = a0 $ b0;
019- c0 = g0 # p0&cin;
020- sum0 = p0 $ cin;
021- g1 = a1 & b1;"second adder bit
022- p1 = a1 $ b1;
023- c1 = g1 # p1&g0 # p1&p0&cin;
024- sum1 = p1 $ c0;
025- g2 = a2 & b2;"third adder bit
026- p2 = a2 $ b2;
027- c2 = g2 # p2&g1 # p2&p1&g0 # p2&p1&p0&cin;
028- sum2 = p2 $ c1;
029- g3 = a3 & b3;" fourth adder bit
030- p3 = a3 $ b3;
031- cout = g3 # p3&g2 # p3&p2&g1 # p3&p2&p1&g0 #
032- p3&p2&p1&p0&cin;
033- sum3 = p3 $ c2;
034-
035- END lkadd

```

Listing 5.5 VHDL code for a 4-bit carry-lookahead adder (LKADD.VHD).

```
001- -- Look-ahead Adder
002-
003- LIBRARY IEEE;
004- USE IEEE.std_logic_1164.ALL;
005-
006- -- Look-ahead Adder interface description
007- ENTITY lkadd IS
008-     PORT
009-     (
010-         a: IN STD_LOGIC_VECTOR (3 DOWNTO 0);-- 4-bit addend
011-         b: IN STD_LOGIC_VECTOR (3 DOWNTO 0);-- 4-bit addend
012-         cin: IN STD_LOGIC;-- input carry
013-         sum: OUT STD_LOGIC_VECTOR (3 DOWNTO 0);-- 4-bit sum
014-         cout: OUT STD_LOGIC-- carry output
015-     );
016- END lkadd;
017-
018- -- Look-ahead Adder architecture description
019- ARCHITECTURE lkadd_arch OF lkadd IS
020- -- signal vectors for transferring the carry-propagate,
021- -- carry-generate, and carry signals between the adder stages
022- SIGNAL g: STD_LOGIC_VECTOR (3 DOWNTO 0);-- carry-generate
023- SIGNAL p: STD_LOGIC_VECTOR (3 DOWNTO 0);-- carry-propagate
024- SIGNAL c: STD_LOGIC_VECTOR (3 DOWNTO 0);-- carry
025- BEGIN
026- PROCESS (a,b,cin,g,p,c)
027- BEGIN
028- -- use a FOR loop to iteratively connect the logic operations
029- -- for the carry-generate, carry-propagate, and sum signals
030- FOR i IN 0 TO 3
031- LOOP
032- g(i) <= a(i) AND b(i);
033- p(i) <= a(i) XOR b(i);
034- sum(i) <= p(i) XOR c(i);
035- END LOOP;
036- -- compute the carry bits for each stage of the adder
037- c(0) <= cin;
038- c(1) <= g(0) OR (p(0) AND c(0));
039- c(2) <= g(1) OR (p(1) AND g(0)) OR
040- (p(1) AND p(0) AND c(0));
041- c(3) <= g(2) OR (p(2) AND g(1)) OR
042- (p(2) AND p(1) AND g(0)) OR
```


Listing 5.5 VHDL code for a 4-bit carry-lookahead adder (LKADD.VHD). (Cont'd.)

```
043- (p(2) AND p(1) AND p(0) AND c(0));  
044- cout <= g(3) OR (p(3) AND g(2)) OR  
045- (p(3) AND p(2) AND g(1)) OR  
046- (p(3) AND p(2) AND p(1) AND g(0)) OR  
047- (p(3) AND p(2) AND p(1) AND p(0) AND c(0));  
048- END PROCESS;  
049- END lkadd_arch;
```

Projects

1. Compile the 8-bit ripple-carry adder from the previous chapter. Create a simple input pattern that will make the adder exhibit its worst-case delay when a clock waveform is input to the adder as c_{in} . Simulate the adder and observe the delay. Download the adder to your XS40 or XS95 Board and measure the delay with an oscilloscope.
2. Design an 8-bit carry-lookahead adder using the 4-bit carry-lookahead module. Apply the same input pattern to this adder. Simulate the adder and observe the worst-case delay. Download the adder to your XS40 or XS95 Board and measure the delay with an oscilloscope. Compare the worst-case delay and size of this adder with the ripple-carry adder (use the report file). Does a faster design take more space in your FPGA?

Overview

The Timing Analyzer performs static timing analysis of an FPGA or CPLD design. The FPGA design must be mapped and can be partially or completely placed, routed, or both. The CPLD design must be completely placed and routed. A static timing analysis is a point-to-point analysis of a design network. It does not include insertion of stimulus vectors.

The Timing Analyzer verifies that the delay along a given path or paths meets your specified timing requirements. It organizes and displays data that allows you to analyze the critical paths in your circuit, the cycle time of the circuit, the delay along any specified paths, and the paths with the greatest delay. It also provides a quick analysis of the effect of different speed grades on the same design.

The Timing Analyzer works with synchronous systems composed of flip-flops and combinatorial logic. In synchronous design, the Timing Analyzer takes into account all path delays, including clock-to-Q and setup requirements, while calculating the worst-case timing of the design. However, the Timing Analyzer does not perform setup and hold checks; you must use a simulation tool to perform these checks.

The Timing Analyzer creates timing analysis reports, which you customize by applying filters with the Path Filters menu commands.

There are several ways to issue commands to the Timing Analyzer. You can select menus, click toolbar buttons, type keyboard commands in the console window, and run macros. For descriptions of Timing Analyzer commands, see Help Topics.



Floorplanner Guide
Chapter 1: Introduction

Features of the Floorplanner

The Floorplanner provides an easy-to-use graphical interface that offers the following features.

- Interacts at a high level of the design hierarchy, as well as with low-level elements such as I/Os, function generators, tristate buffers, flip-flops, and RAM/ROM
- Captures and imposes complex patterns, which is useful for repetitive logic structures such as interleaved buses
- Automatically distributes logic into columns or rows
- Uses dynamic rubberbanding to show the ratsnest connections
- Finds logic or nets by name or connectivity
- Permits design hierarchy rearrangement to simplify floorplanning
- Groups logic by connectivity or function
- Identifies placement problems in the Floorplan window
- Provides online help



Floorplanner Guide
Chapter 1: Introduction

What is the Floorplanner?

The Floorplanner is a graphical placement tool that gives you control over placing a design into a target FPGA using a “drag and drop” paradigm with the mouse pointer.

The Floorplanner displays a hierarchical representation of the design in the Design Hierarchy window using hierarchy structure lines and colors to distinguish the different hierarchical levels. The Floorplan window displays the floorplan of the target device into which you place logic from the hierarchy. The following figure shows the windows on the PC version.



Figure 1.1 Floorplanner Window

Logic symbols represent each level of hierarchy in the Design Hierarchy window. You can modify that hierarchy in the Floorplanner without changing the original design.

You use the mouse to select the logic from the Design Hierarchy window and place it in the FPGA represented in the Floorplan window.

Alternatively, you can invoke the Floorplanner after running the automatic place and route tools to view and possibly improve the results of the automatic implementation.

[Click here to see first search hit](#)

Title: ADD4, 8, 16



Libraries Guide

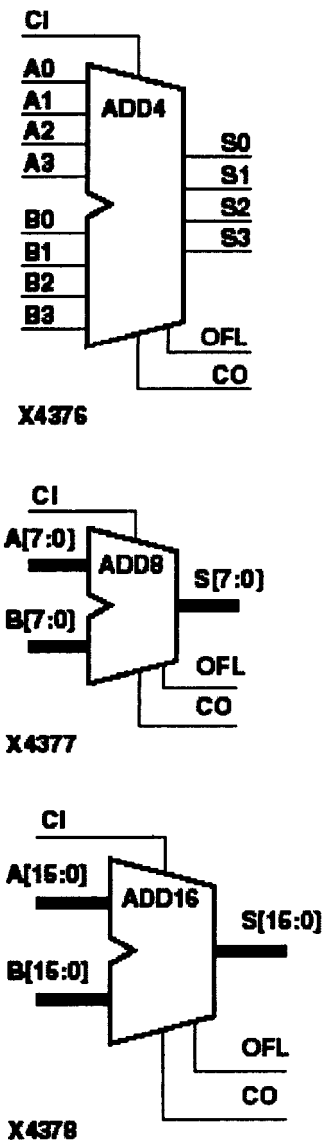
Chapter 3: Design Elements (ACC1 to BYPOSC)

ADD4, 8, 16

4-, 8-, 16-Bit Cascadable Full Adders with Carry-In, Carry-Out, and Overflow

XC3000XC4000EXC4000XXC5200XC9000SpartanSpartanXLSpartan2Virtex

MacroMacroMacroMacroMacroMacroMacroMacroMacro



ADD4, **ADD8**, and ADD16 add two words and a carry-in (CI), producing a sum output and carry-out (CO) or overflow (OFL). ADD4 adds A3 - A0, B3 - B0, and CI producing the sum output S3 - S0 and CO (or OFL). **ADD8** adds A7 - A0, B7 - B0, and CI, producing the sum output S7 - S0 and CO (or OFL). ADD16 adds A15 - A0, B15 - B0 and CI, producing the sum output S15 - S0 and CO (or OFL).

ADD4, **ADD8**, and ADD16 are implemented in the XC4000, Spartan, and SpartanXL using carry logic and relative location constraints, which assure most efficient logic placement.

Unsigned Binary Versus Twos Complement

ADD4, **ADD8**, ADD16 can operate on either 4-, 8-, 16-bit unsigned binary numbers or 4-, 8-, 16-bit twos-complement numbers, respectively. If the inputs are interpreted as unsigned binary, the result can be interpreted as unsigned binary. If the inputs are interpreted as twos complement, the output can be interpreted as twos complement. The only functional difference between an unsigned binary operation and a twos-complement operation is how they determine when "overflow" occurs. Unsigned binary uses CO, while twos-complement uses OFL to determine when "overflow" occurs. Therefore, if you want to interpret the inputs as unsigned binary, you should follow the CO output. If you want to interpret the inputs as twos complement, you should follow the OFL output.

Unsigned Binary Operation

For unsigned binary operation, ADD4 can represent numbers between 0 and 15, inclusive; **ADD8** between 0 and 255, inclusive; ADD16 between 0 and 65535, inclusive. CO is active (High) when the sum exceeds the bounds of the adder.

OFL is ignored in unsigned binary operation.

Twos-Complement Operation

For twos-complement operation, ADD4 can represent numbers between -8 and +7, inclusive; **ADD8** between -128 and +127, inclusive; ADD16 between -32768 and +32767, inclusive. OFL is active (High) when the sum exceeds the bounds of the adder.

CO is ignored in twos-complement operation.

Topology for XC4000, Spartan, SpartanXL

This is the ADD4 (4-bit), **ADD8** (8-bit), and ADD16 (16-bit) topology for XC4000E, XC4000X, Spartan, and SpartanXL devices.

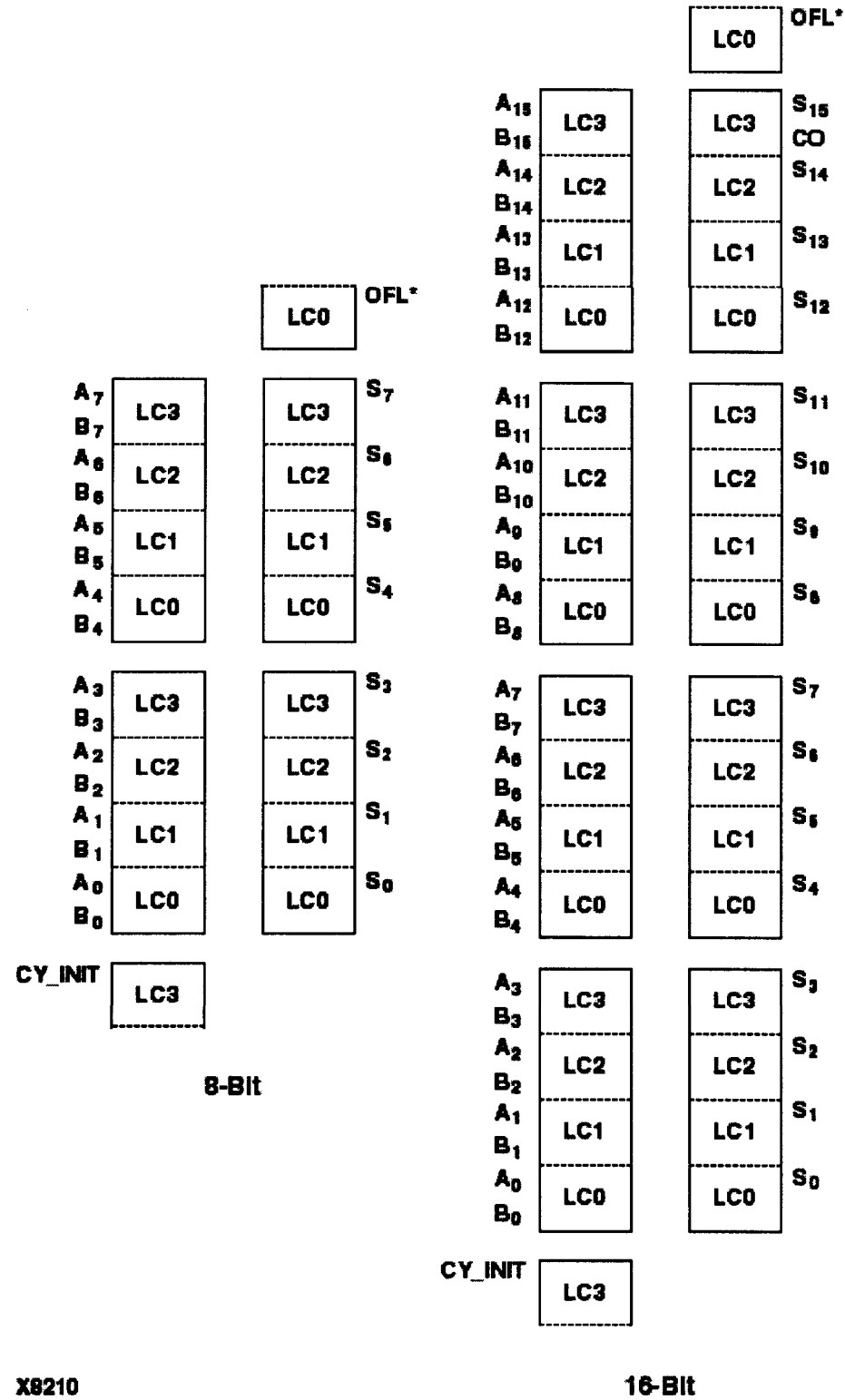


Figure 3.9 ADD8 Implementation XC3000

Figure 3.10 ADD8 Implementation XC4000E, XC4000X, Spartan, SpartanXL

Figure 3.11 ADD8 Implementation XC5200

Figure 3.12 ADD8 Implementation Spartan2, Virtex



Figure 3.13 ADD4 Implementation XC9000



Figure 3.14 ADD8 Implementation XC9000

[Click here to see first search hit](#)