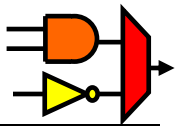


Elettronica dei Sistemi Digitali
Digital Systems Electronics

Lab#3

Data-Path Elements

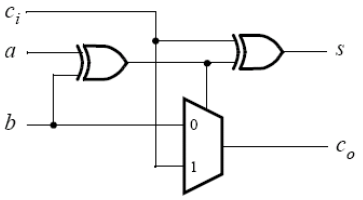
This purpose of this lab is design arithmetic circuits that add, subtract, and multiply binary numbers. Addition, subtraction and multiplication are fundamental operations usually implemented in a Data-Path (DP). The DPs are generally sequential circuits that numerically process some kind of digital input stream. For complex circuits, it is often necessary to combine multiple adders or multipliers.

Contents:

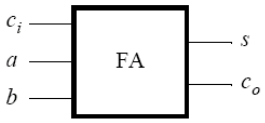
Abbreviations and acronyms:
CBA – Carry Bypass Adder
CSA – Carry Select Adder
DP – Data Path
IC – Integrated Circuit
LED – Light Emitting Diode
MUX – Multiplexer
RCA – Ripple Carry Adder
VHDL – Very high speed integrated circuits Hardware Description Language
[VHDL cookbook: http://www.onlinefreeebooks.net/engineering-ebooks/electrical-engineering/the-vhdl-cookbook-pdf.html]

1 – 4-bit Sequential RCA

Figure 1a shows a possible implementation of a *full adder*. It reads the three inputs a, b, and c_i, and drives the outputs s and c_o. Parts b and c of the figure show a circuit symbol and the truth table of the full adder. This fundamental logic block, used massively in every digital design, generates a 2-bit array of binary sums $<c_o\, s> = a + b + c_i$, where c_o and s are called Carry-Out and Sum, respectively. Figure 1d shows how four instances of this full adder can be used for designing a circuit that adds up two 4-bit numbers. This type of circuit is usually called *ripple-carry* adder. The term "ripple" refers to the way the carry-out is propagated from a full adder to the next one. Actually, in this topology, we have to wait until the carry is propagated until the last Full Adder to have all the five sum bits ready.



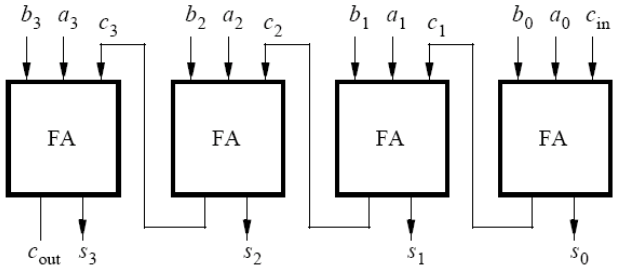
a) Full adder circuit



b) Full adder symbol

b	a	c_i	c_o	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

c) Full adder truth table



d) Four-bit ripple-carry adder circuit

Figure 1 - A ripple-carry adder circuit.

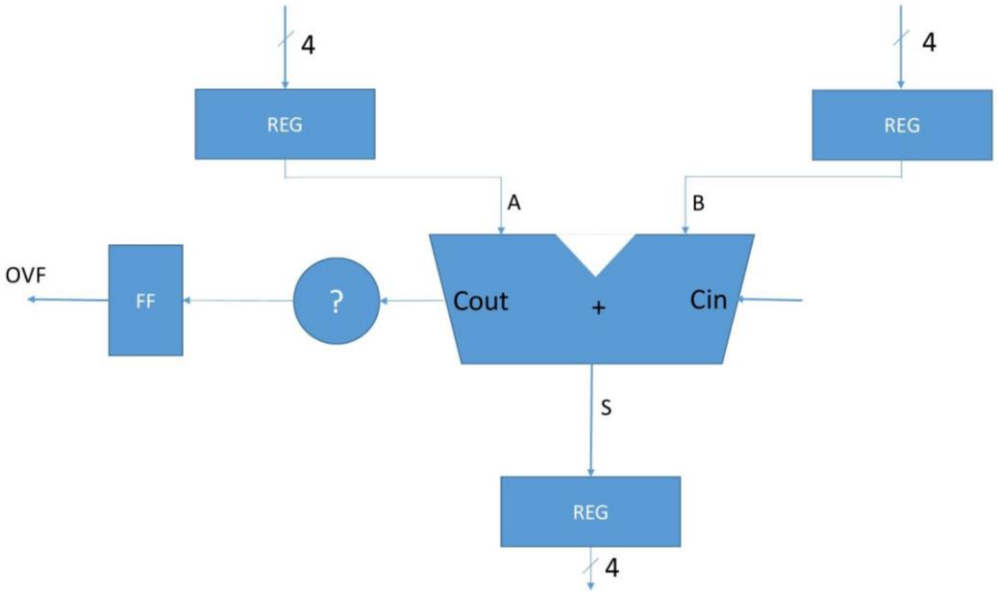
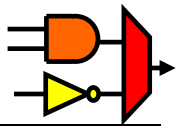


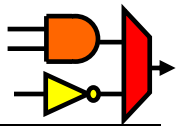
Figure 2 - A 4-bit signed adder with overflow generation and registered inputs and output.

Create a 4-bit version of the adder and include it in the circuit shown in Figure 2. Your circuit should be designed to support signed numbers in 2's-complement form, and the *Overflow* output should be set to 1 whenever the sum produced by the adder does not provide the correct sign value. Notice that, in the unsigned case, the overflow bit is the output carry of the leftmost full-adder. However, for signed numbers, the overflow must be generated differently: the question mark in Figure 2 indicates that you have to modify this part of the design to generate the correct overflow flag.
Do the steps below.



1. **Create a new Modelsim project** and write a VHDL code that describes the circuit in Figure 2. Use the circuit structure in Figure 1 to describe your adder. Use the templates shown in Figure 3 to describe the D-FF and the register. Compile and run the behavioral simulation to test your model.
2. **Create a new Quartus Prime project**, import your code and **include the required input and output ports** in your project to implement the adder circuit on the DE1 board. Connect the inputs *A* and *B* to switches SW_{3-0} and SW_{7-4}, respectively. Use KEY_0 as an active-low asynchronous reset input, and use KEY_1 as a manual clock input. Display the overflow of the adder on the red $LEDR_9$. The hexadecimal values of *A*, *B* and S must be shown on the seven segment displays. Run the synthesis, download the circuit onto the DE1 board and test it by using different values of *A* and *B*. Verify the correct behaviour of the *Overflow* output.
3. Open the Quartus Prime Compilation Report and examine the **results reported by the Timing Analyzer** (read the Appendix A in this document for a short introduction to the use of the timining analysis in Quartus Prime). What is the maximum operating frequency f_{max} of your circuit? What is the longest path in the circuit in terms of delay?
4. **Run the timing simulation** (see last Section in the Lab2 assignment) on Modelsim.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY regn IS
      GENERIC ( N : integer:=4);
      PORT (R                 : IN  SIGNED(N-1 DOWNTO 0);
            Clock, Resetn     : IN  STD_LOGIC;
            Q                 : OUT SIGNED(N-1 DOWNTO 0));
END regn;

ARCHITECTURE Behavior OF regn IS
BEGIN
      PROCESS (Clock, Resetn)
      BEGIN
            IF (Resetn = '0') THEN
                  Q <= (OTHERS => '0');
            ELSIF (Clock'EVENT AND Clock = '1') THEN
                  Q <= R;
            END IF;
      END PROCESS;
END Behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY flipflop IS
      PORT (D, Clock, Resetn  : IN  STD_LOGIC;
            Q                 : OUT STD_LOGIC);
END flipflop;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
      PROCESS (Clock, Resetn)
      BEGIN
            IF (Resetn = '0') THEN -- asynchronous clear
                        Q <= '0';
            ELSIF (Clock'EVENT AND Clock = '1') THEN
                  Q <= D;
            END IF;
```



```
        END PROCESS;
END Behavior;
```

Figure 3 - VHDL code of the register and the D-FF.

2 - 4-bit Sequential Adder/Subtractor

Modify your circuit from Section 1 in such a way it can perform both addition and subtraction of four-bit numbers. Use switch *SW*8 to specify whether addition or subtraction should be done. Connect the switches, LED, and displays as described in section 1.

1. **Simulate your adder/subtractor circuit** and show that it works properly (use Modelsim and a proper test bench).
2. **Download it** onto the DE1 board and test it by using different switch settings.
3. Open the Quartus Prime Compilation Report and examine the **results reported by the Timing Analyzer**. What is the maximum operating frequency f_{max} of your circuit? What is the longest path in the circuit in terms of delay?

3 – 16-bit RCA, Carry-Bypass Adder and Carry-Select Adder

Modify your ripple-carry adder developed for the project of section 1 in such a way that it can perform additions of 16-bit numbers.

1. **Simulate** your Ripple-Carry adder circuit and show that it functions properly.
2. Open the Quartus Prime Compilation Report and examine the **results reported by the Timing Analyzer**. What is the f_{max} of your circuit? What is the longest path in the circuit in terms of delay?

Substitute the RCA with a Carry-Bypass adder (figure 3).

3. **Simulate** your Carry-Bypass adder circuit and show that it functions properly.
4. Open the Quartus Prime Compilation Report and examine the **results reported by the Timing Analyzer**. What is the f_{max} of your circuit? What is the longest path in the circuit in terms of delay?

Substitute the Carry-Bypass adder with a Carry-Select adder (figure 4).

5. **Simulate** your Carry-Select Adder circuit to show that it functions properly.
6. Open the Quartus Prime Compilation Report and examine the results **reported by the Timing Analyzer**. What is the f_{max} of your circuit? What is the longest path in the circuit in terms of delay?
7. **Compare the timing results** and the synthesis reports of the three adders and comment them.

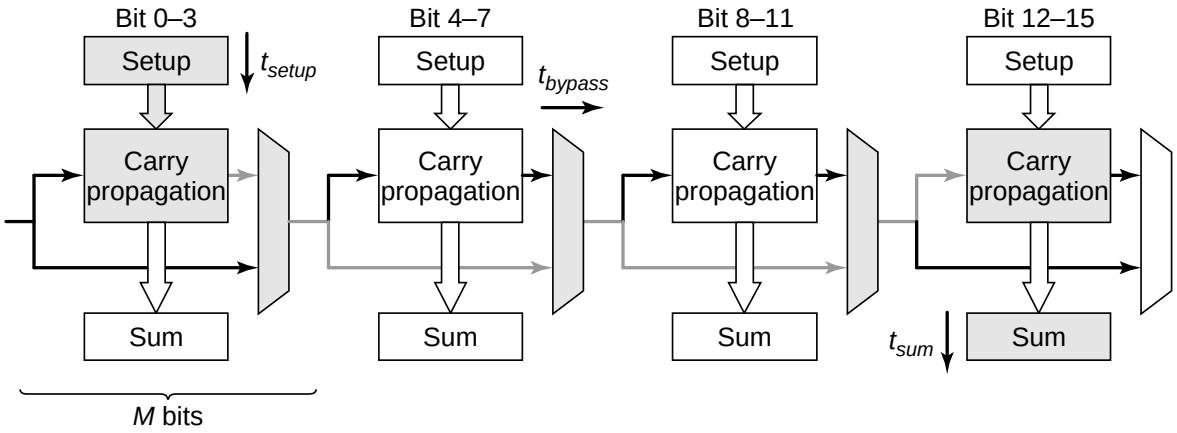


Figure 3 - Carry-Bypass Adder

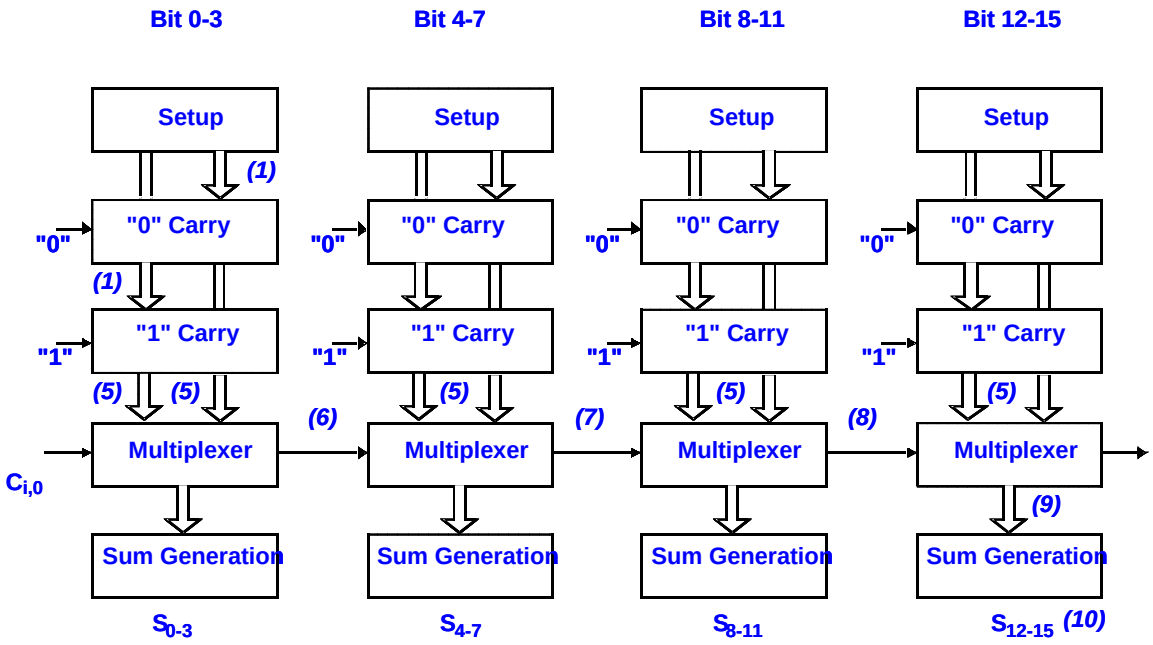


Figure 4 - Carry-Select Adder.

4 - Multiplier

Figure 5a shows the traditional paper-and-pencil multiplication $P = A \times B$, where $A = 12$ and $B = 11$. We need to add two summands that are shifted versions of A to have the product $P = 132$. The part b of the figure shows the same example by using four-bit binary numbers. Since each digit in B is either 1 or 0, the summands are either shifted versions of A or 0000. Figure 5c shows how each summand can be formed by an AND gate on A and the appropriate bit in B.

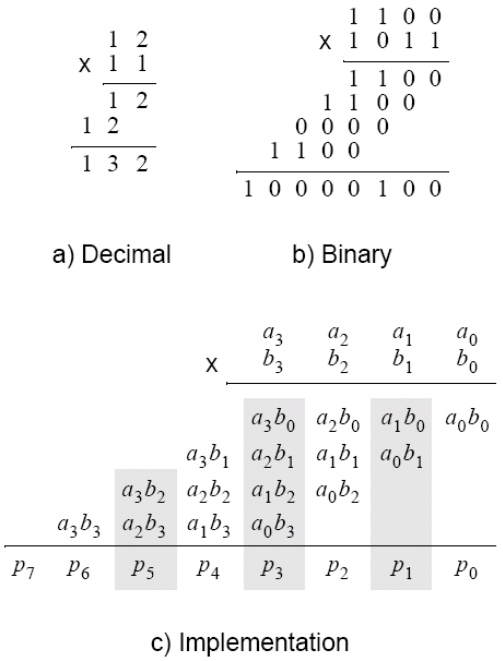


Figure 5 - Multiplication of binary numbers.

A 4-bit circuit that implements $P = A \times B$ is illustrated in Figure 6. Since its structure is regular, this type of multiplier circuit is usually called *array multiplier*. The shaded areas in the circuit correspond to the implementations of the shaded columns in Figure 5c. In each row of the multiplier the AND gates are used to generate the summands and the full adder modules are used to generate the required sums.

Do the following steps and implement the array multiplier circuit:

1. **Create a new Quartus Prime project** which will be used to implement the multiplier on the Altera DE1 board.
2. **Generate the VHDL file**, include it in your project, and compile the circuit.
3. Use **functional simulations** (Modelsim) to verify that your code is correct.
4. **Augment your design** in such a way it uses switches SW_{3-0} to represent the number A and switches SW_{7-4} to represent B. The ***hexadecimal*** values of A and B should be displayed on the 7-segment displays *HEX0* and *HEX1*, respectively. The result $P = A \times B$ has to be displayed on *HEX3* and *HEX2*.
5. **Assign the pins on the FPGA** to route the connections of the switches and of the 7-segment displays as indicated in the User Manual of the DE1 board.
6. **Recompile the circuit** and download it in the FPGA IC.
7. **Test the functionality of your design** by toggling the switches and observing the 7-segment displays.

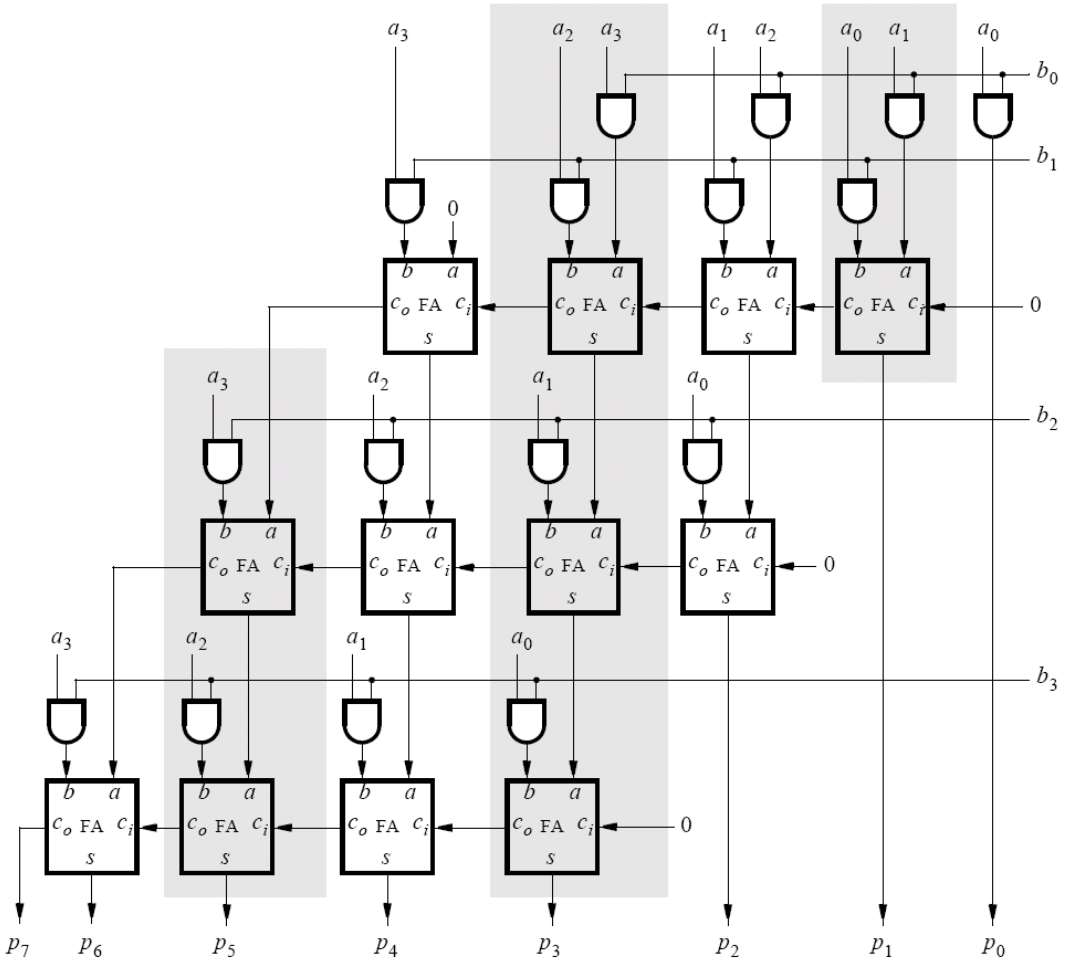
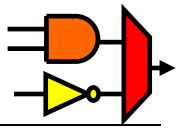


Figure 6 - An array multiplier circuit.



Appendix A –Timing Analysis

Quartus runs the Timing analysis after the Synthesis and Fitter (Place and Route) steps. The tool supports the industry standard Synopsys Design Constraints (.sdc) format for specifying timing constraints.
To include timing constraints in your design, you have to create a .sdc file (use any editor, e.g. notepad) and enter the following lines:

```
#**************************************************************
# Time Information
#**************************************************************

set_time_format -unit ns -decimal_places 3



#**************************************************************
# Create Clock
#**************************************************************

create_clock -name {CK} -period 10.000 -waveform { 0.000 5.000 }[get_ports {CK}]
```

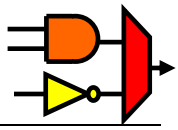
These lines define a clock signal with period of 10 ns and duty cycle 50%. For the clock signal name, use the same name as the one appearing in your VHDL model.
Save the file in the project folder and, on Quartus, double click on *Timing Analisis -> Edit Settings*. This opens a setting window, where you can enter the name of the created sdc file and add it to the project. Once you have included your sdc file, run again the synthesis. The timing results are available under the *Timing Analysis* section, by clicking on *View Report*. In particular, under *Fmax Summary*, you can read the maximum achievable clock frequency of the synthesized circuit.

The general syntax for this sdc command is:

```
create_clock [-add] [-name <clock_name>] -period <value> [-waveform
<edge_list>] <targets>
```

where,
-add: Adds clock to a node with an existing clock
-name <clock_name>: Clock name of the created clock
-period <value>: Speed of the clock in terms of clock period
-waveform <edge_list>: List of edge values
<targets>: List or collection of targets

Example 1: Create a simple 10ns with clock with a 60% duty cycle



```
create_clock -period 10 -waveform {0 6} -name clk [get_ports clk]
```

Example 2: Create a clock with a falling edge at 2ns, rising edge at 8ns,# falling at 12ns, etc.

```
create_clock -period 10 -waveform {8 12} -name clk [get_ports clk]
```

Example 3: Assign two clocks to an input port that are switched externally

```
create_clock -period 10 -name clk100Mhz [get_ports clk]create_clock
-period 6.667 -name clk150Mhz -add [get_ports clk]
```

More details on the timing analysis and on the use of a Timing Analyser are available in the following Appendix B and also in the following Intel FPGA ducuments:

1. Timing Analyzer Quick-Start Tutorial (Intel Quartus Prime Pro Edition) https://www.intel.com/content/www/us/en/programmable/documentation/caf1499898833805.html
2. Intel® Quartus® Prime Pro Edition User Guide - Timing Analyzer https://www.intel.com/content/www/us/en/programmable/documentation/psq1513989797346.html

Appendix B – Introduction to TimeQuest Timing Analyzer

This section demonstrates how to set up timing constraints and obtain timing information for a logic circuit using TimeQuest Timing Analyzer.

Timing analysis is a process of analyzing delays in a logic circuit to determine the conditions under which the circuit operates reliably. One example of a timing analysis computation is to find the maximum clock frequency for a circuit including combinational components and flip-flops. To operate correctly, the clock period has to be long enough to accommodate the delay on the longest path in the circuit. Computing the longest delays in a circuit and comparing these delays to the clock period is a basic function of a timing analyzer.
The timing analyzer can be used to guide computer-aided design (CAD) tools in the implementation of logic circuits. By placing timing constraints on the maximum clock frequency, it is possible to direct the CAD tools to seek an implementation that meets those constraints.

Design Example

As an example we will use an adder that adds three 8-bit numbers and produces a sum output. The inputs are A, B, and C, which are stored in registers reg_A, reg_B and reg_C at the positive edge of the clock. The three registers provide inputs to the adder, whose result is stored in the reg_sum register. The output of the reg_sum register drives the output port sum. The diagram of the circuit is shown in Figure 1A.

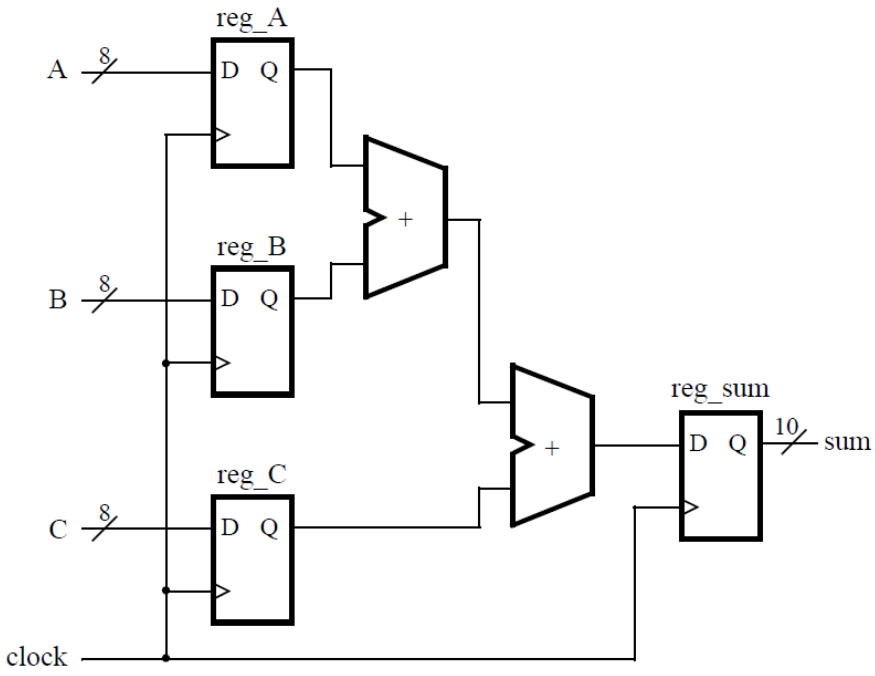
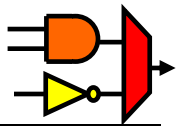


Figure 1A: Example design.

The VHDL source code for the design is given below. Note that the"synthesis keep" comment is included in this code. This comment is interpreted as a directive that instructs the Quartus Prime software to retain the specified nodes in the final implementation of the circuit and keep their names as stated. This directive will allow us to refer to these nodes in the tutorial.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;
ENTITY add_three_numbers IS
PORT ( clock : IN STD_LOGIC;
     A, B, C : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
         sum : OUT STD_LOGIC_VECTOR(9 DOWNTO 0));
END add_three_numbers;
ARCHITECTURE Behavior OF add_three_numbers IS
-- Registers
SIGNAL reg_A, reg_B, reg_C : STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL reg_sum : STD_LOGIC_VECTOR(9 DOWNTO 0);
ATTRIBUTE keep : boolean;
```



```
ATTRIBUTE keep OF reg_A, reg_B, reg_C, reg_sum : SIGNAL IS true;
BEGIN
  PROCESS ( clock )
  BEGIN
    IF (clock'EVENT AND clock = '1') THEN
      reg_A <= A;
      reg_B <= B;
      reg_C <= C;
      reg_sum <= ("00" & reg_A) + ("00" & reg_B) + ("00" & reg_C);
    END IF;
  END PROCESS;
  sum Ç= reg_sum;
END Behavior;
```

To begin the tutorial create a new Quartus Prime project for the design of our example circuit. Select as the target device the usual FPGA chip of the DE1-SoC board. Type the VHDL code above into a file and add this file to the project.
You do not need to make any pin assignments for this example. Compile the project to see the results of timing analysis. These results will be available in the compilation report, once the design is compiled.

Using TimeQuest

Open the *Timequest Timing Analyzer* section of the Compilation Report, and click on the *Clocks* item to select it. In the Clocks display panel that opens on the right-hand side of the Quartus Prime window, notice that the clock signal from the example design has been given a clock period constraint of 1 ns (frequency of 1000 MHz). This is a default constraint that the Quartus Prime CAD tool places on any clock signal in a design project that does not have any user-provided timing constraints. Right-click on the name of the clock signal and select the command *Report Timing* ... (in *Timequest UI*). This action opens the Report Timing dialog shown in Figure 2A.

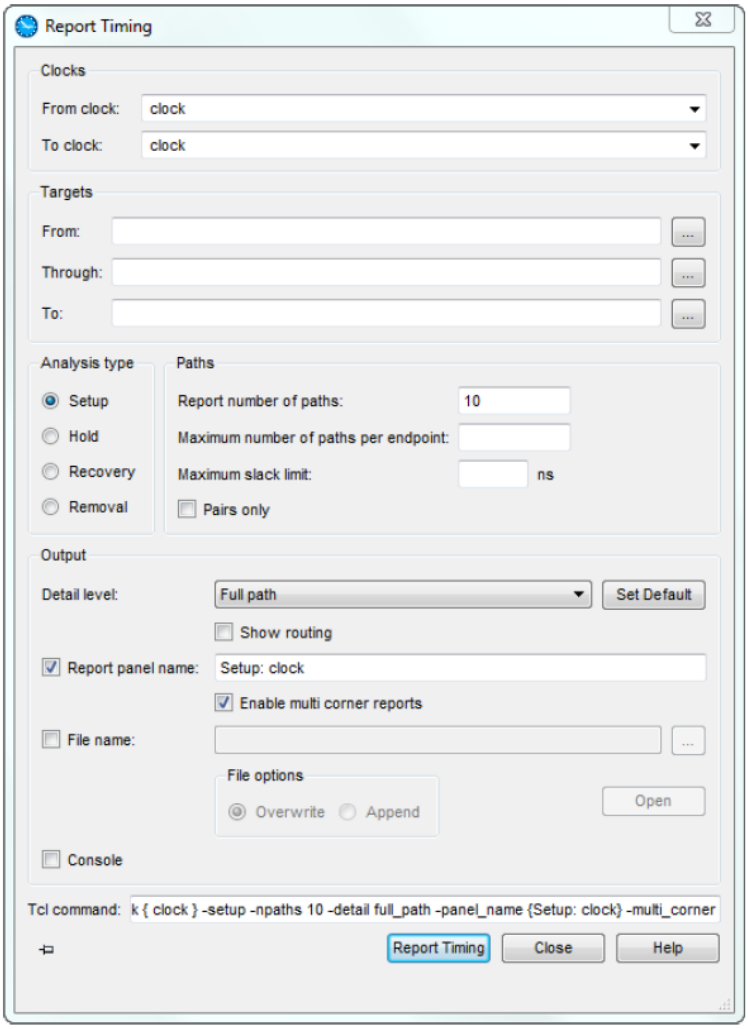


Figure 2A: Report Timing dialog.

Click the drop-down arrow in the *From clock* item and select the clock signal. This selection is used to instruct TimeQuest to analyze all paths in the example circuit that start and end at flip-flops that are clocked by the clock signal. The various settings are displayed in this window: accept all of the default selections and click on the *Report Timing* button. This command opens the *Timequest Graphical User Interface* (GUI) shown in Figure 3A.

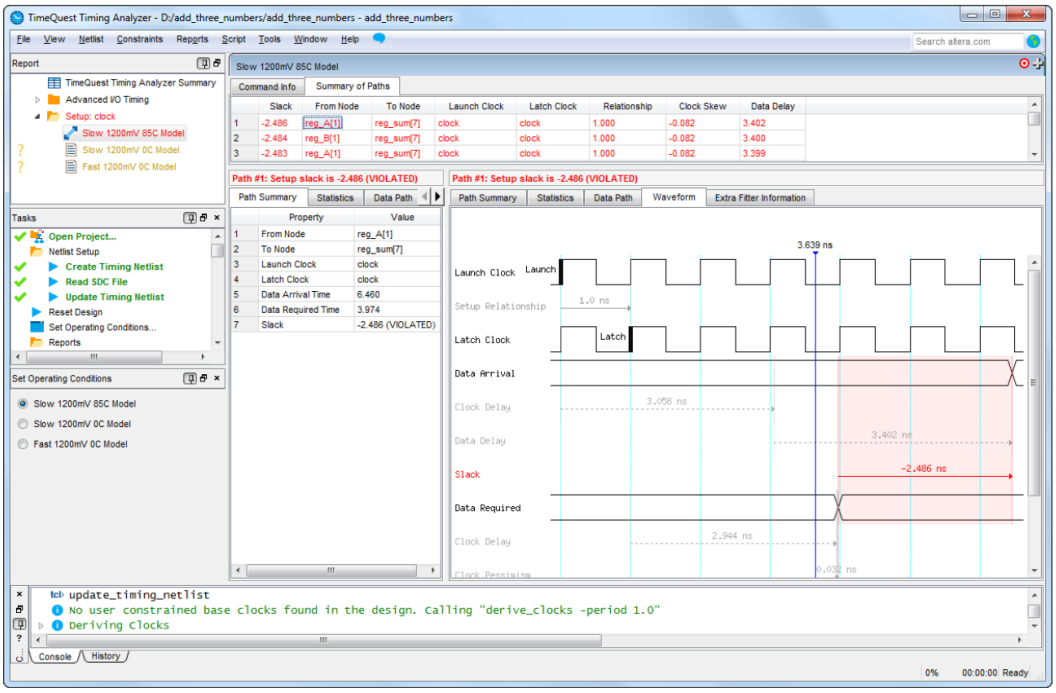


Figure 3A: The TimeQuest GUI.

The TimeQuest GUI consists of several sections. They include the main menu at the top, the Report pane in the top-left corner, the Tasks pane on the left, the detailed results panes in the middle, and the Console display at the bottom of the window. The main menu is used to interact with the TimeQuest tool and issue commands. The Report pane contains any reports generated when using the tool, and the Tasks pane contains a sequence of actions that can be performed to obtain timing reports. The View pane hosts any windows that are opened, such as details about the timing information. The Console window at the bottom provides access to a command line for TimeQuest.

We will focus on the panes in the middle of the TimeQuest GUI, which show detailed results of the timing analysis. The Slow 1200mV 85C Model pane lists the analyzed paths in the circuit from source to destination flip-flops. The first column in this pane shows the Slack of each path with respect to the (default) clock constraint of 1 ns. For each path in the circuit, the slack value represents the difference between the clock period constraint and the path delay:

- a positive slack means that the delay is smaller than the constraint,
- a negative slack represents a delay that is larger than the constraint.

If the slack values in the report are negative, they are shown in red, because the timing results fail to meet the required constraint. If the maximum negative slack value is -2.486 ns, it means that the worst-case delay path is 1 - (-2.486) = 3.486 ns long. This corresponds to a maximum usable clock frequency, F_{max} , of about 286.86 MHz.

The waveforms shown for path #1 in Figure 3A illustrate the detailed timing situation. The waveforms show that the clock signal takes 3.058 ns to propagate from its input pin to the source flip-flop, and then this flip-flop produces data that takes 3.402 ns to reach the destination flip-flop. Also, the clock

signal takes 2.944 ns to reach the destination flip flop. The clock delays at the source and destination flip-flops represent a clock skew t_{skew} = 2.944 - 3.058 = - 0.114 ns.
A value of 0.032 to account for *Clock Pessimism* is added to the clock skew, making the final clock skew value to be - 0.082 ns. The difference in the required arrival time of the data on this path and the actual arrival time is shown by the negative slack value of 1- 3.402 + t_{skew} = - 2.484 ns. TimeQuest adds a value of - 0.002 ns to account for *Clock Uncertainty*, leading to the final slack value of -2.486 ns.

Setting Up Timing Constraints for a Design

In the TimeQuest GUI, select **`Constraints -> Create Clock`**, which leads to the *Create Clock* window shown in Figure 4A. Set the Clock name to *clock* and the Period to 4.000 ns.

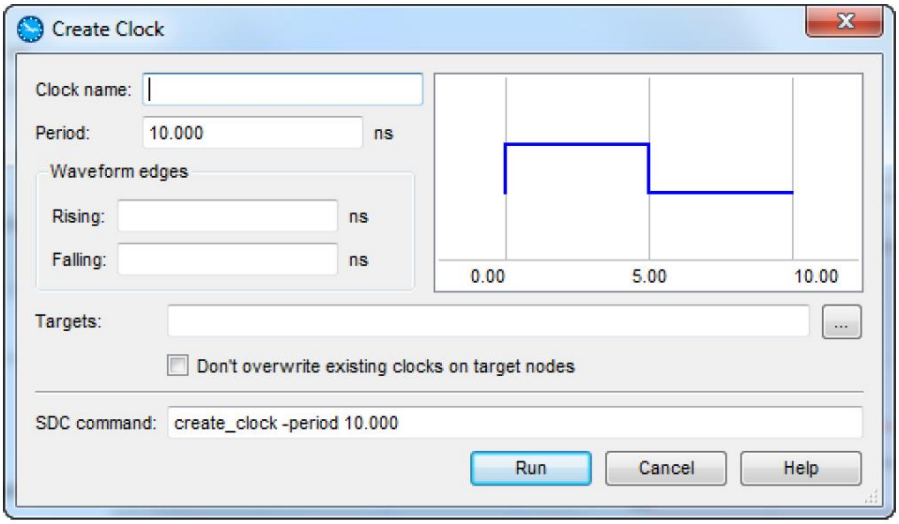


Figure 4A: The Create Clock window.

It is necessary to tell TimeQuest which signal in our design this clock constraint applies to. To do this, click the **...** button to the right of the Targets field, leading to the Name Finder window shown in Figure 5. Click List to show all of the ports in the design. In the list of ports, highlight *clock*, which is the clock signal in our circuit, press **`>,`** then click OK. Finally, click the **`Run`** button in the Create Clock window to apply the constraint.

In order to use this clock constraint for future compilations and timing analysis of this project, we must save the constraint to a file of the type *sdc*, which stands for Synopsys Design Constraint. This file uses an industry standard format for specifying timing constraints. Select **`Constraints -> Write SDC File...`** to write all of the currently set constraints (in our case just the one clock constraint) to an SDC file. This leads to a dialog window, where we specify the `sdc` file name, e.g. *add_three_numbers.sdc*. Note that Quartus will by default try to locate and use the sdc file whose file name matches the project name (except for the .sdc extension).

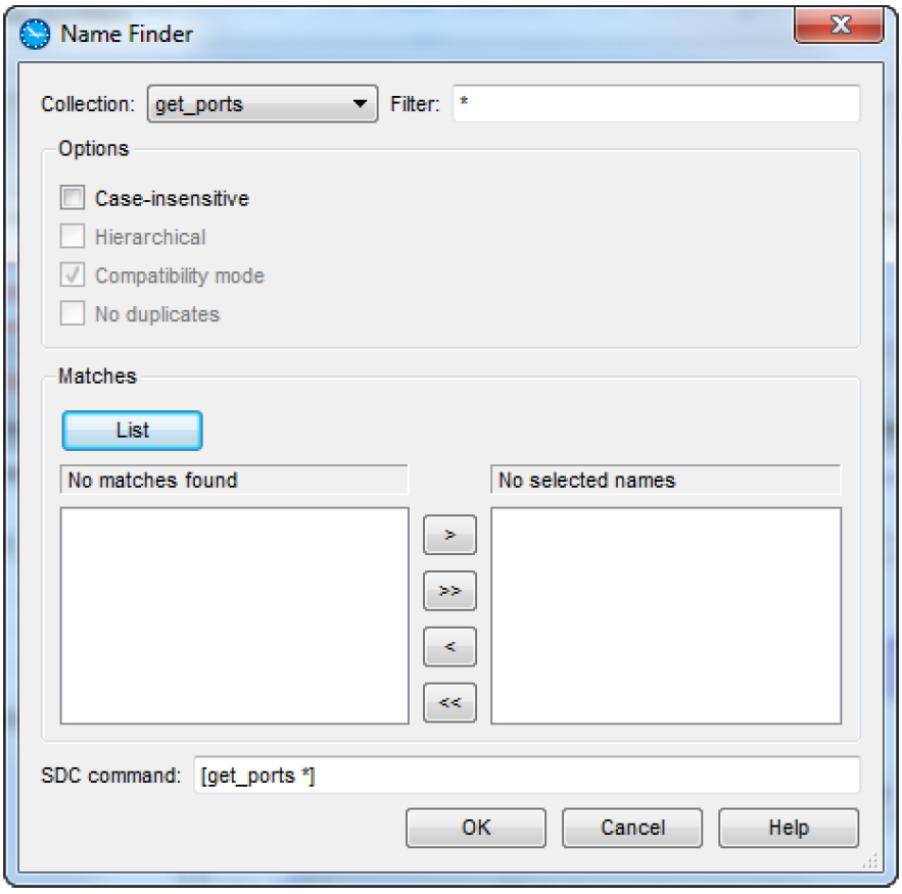


Figure 5A: The Name Finder window.

You will notice that our clock timing report Setup clock is now out of date, as indicated by the yellow font and highlighting. Right-click the report and select Regenerate, to re-run the timing analysis using the new 4 ns clock period constraint. This analysis results in a positive slack of 0.514 ns.

Now, close the TimeQuest GUI. Since we have not recompiled the example design after adding the 4 ns clock period constraint, the timing analysis is based of the circuit that was produced by Quartus Prime when using the (default) 1 ns clock period constraint. To see the effect of the new timing constraint on the compilation results, load the new constraint file and recompile the project. First, select `Project -> Add/Remove Files in Project`, then in the Setting window click on TimeQuest Timing Analyzer in the Category tab; finally, select the SDF file and click OK. This will add the created constraint file to the project. Now, follow the steps described previously to perform a new timing analysis using TimeQuest. The 4 ns timing constraint will cause the Quartus Prime optimization algorithms to make different decisions from those made when the (default) 1 ns constraint was used. In particular, the optimization algorithms will likely take less time to execute, because once the generated circuit has sufficient positive slack to meet the constraint, the algorithms can terminate. Figure 6A shows the results of timing analysis, with a positive slack of 0.600 ns.

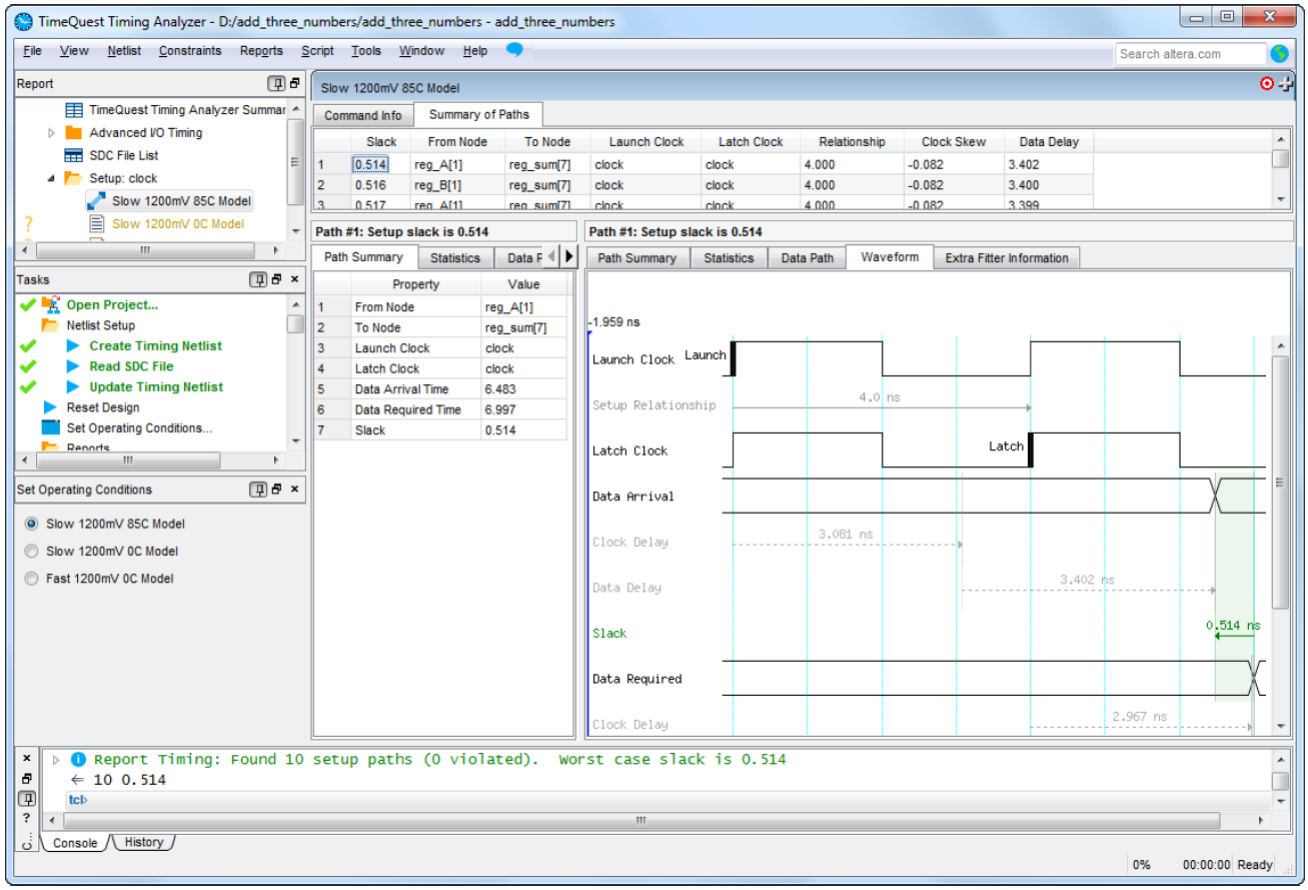


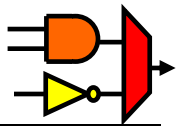
Figure 6A: Updated waveforms.

The TimeQuest Graphical User Interface

In the previous example, we accessed TimeQuest via the Quartus Prime Compilation Report. Another way to open the TimeQuest GUI is to use the command **`Tools > TimeQuest Timing Analyzer`** from the main Quartus Prime window.

There are several commands available in the TimeQuest GUI to obtain timing data.

- In the Tasks pane, we can create a timing netlist, which will be used to perform the analysis, by double-clicking the *Create Timing Netlist* command. Note that while this netlist was generated automatically when performing a timing analysis, as described in the previous sections, the netlist can also be generated manually by using the Tasks pane.
- By double-clicking *Read SDC File*, we instruct the analyzer to read a Synopsys Design Constraints (SDC) file and apply the constraints during analysis. The SDC file can be edited manually (using any text editor) at any time, and the timing analysis can then be re-run using the new constraints.



- By double-clicking the *Update Timing Netlist* command, we use the specified constraints to determine which parts of the circuit fail to meet them. Once the timing netlist is updated, reports can be generated.
- To generate a report, double-click on a report name in the Tasks pane e.g. the *Report Setup Summary*. By right-clicking on the clock name and then selecting *Report Timing...,* we open the *Report Timing dialog* previously shown in Figure 2A. Several fields in this window help specify the data to be reported.
    - The first field is the **`Clocks`** field, which specifies the types of paths that will be reported. More precisely, it specifies the clock signal at the source flip-flops (*From clock*) and the clock signal at the destination flip-flops (*To clock*). This will limit the reporting to the register-to-register paths only.
    - The next field is the **`Targets`** field. It can be used to refine the report by focusing only on certain paths in the design. As an example, we can specify the starting and the ending point of the paths of interest by filling the *From* and *To* fields. In addition, we can look at only the paths that pass through certain nodes in the design.
    - The next two fields are the **`Analysis type`** and **`Paths`** fields. The Analysis type field specifies if the report should contain setup, hold, recovery, or removal information. Each of these analyses looks for distinct timing characteristics in your design. For example, the setup analysis determines if the data arrives at a flip-flop sufficiently early for the flip-flop to store it reliably, given a clock period. On the other hand, the hold analysis determines if the data input at any given flip-flop remains stable after the positive edge of the clock long enough for the data to be stored in a flip-flop reliably. The Paths field specifies the maximum number of paths to be reported and the maximum slack required for a path to be included in the report.
    - The next set of fields specify the **`Output`** format and the level of detail in the report. The output could be to a window or a file.
    - The last field is the **`Tcl command`** field. This field shows a command that will be executed to generate the requested report.
- Timing constraints can be entered by using the Constraints menu in the TimeQuest GUI. To assign a clock constraint, select **`Create Clock...`** from the Constraint menu.
- Once the constraints file is saved, it can be used by Quartus Prime when compiling a project. This is done in Quartus Prime by going into **`Assignments > Settings... > TimeQuest Timing Analyzer`**, and adding the SDC file to the TimeQuest timing analyzer settings. The Quartus Prime project can then be recompiled to use the constraint.

TimeQuest is capable of analyzing circuits that contain multiple clocks. In designs with multiple clocks, it is important to apply constraints to each clock before performing timing analysis.