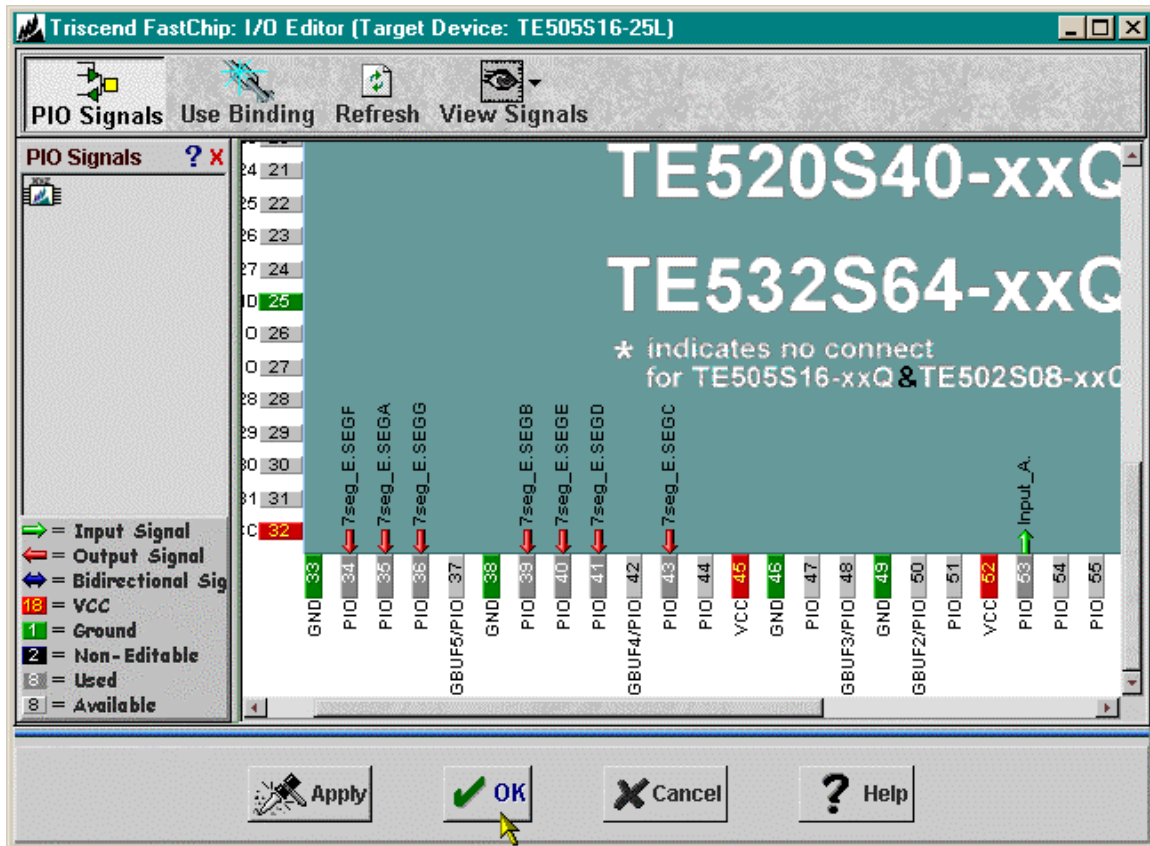


After you make the pin assignments, the **I/O Editor** window will appear as follows:

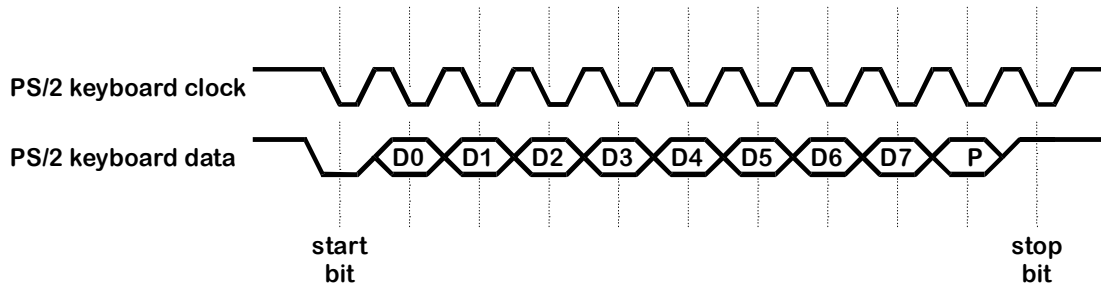


Now that the pin assignments are complete, click on OK to return to the project window. Then bind and download the timer circuit to your CSoc Board. Once the circuit is in the CSoc Board, you can make the timer run by placing DIP switch #1 into its lower position. You should see all sixteen hexadecimal numerals displayed repeatedly on the CSoc LED in a 5 second loop. Raising DIP switch #1 will halt the incrementing of the LED digit.

## Design 1.4 - PS/2 Keyboard Scanner

This example creates a circuit that accepts scan codes from a keyboard attached to the PS/2 interface of the CSoc Board. If a scan code for one of the keys "0"–"9" arrives, then the numeral will be displayed on the LED digit of the CSoc Board.

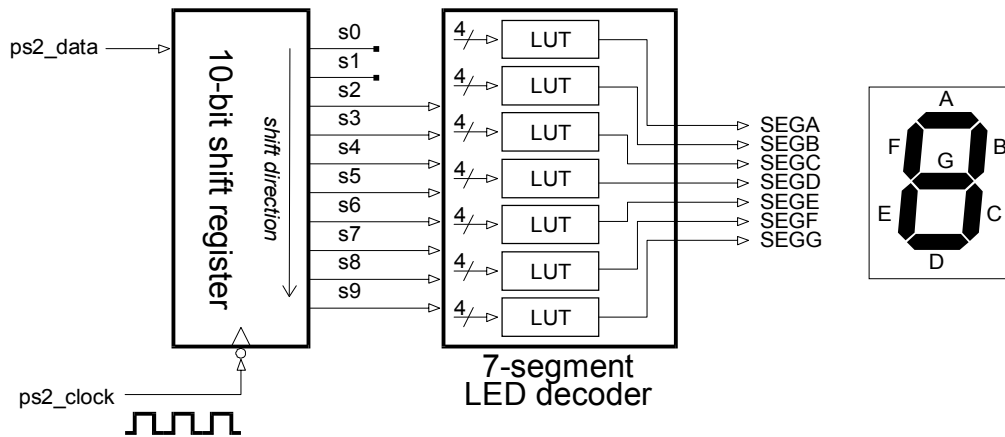
The format of the scan code transmissions from the keyboard are shown in Figure 9. The keyboard drives the clock and data lines. The start of a scan code transmission is indicated by a low level on the data line at the falling edge of the clock. The eight bits of the scan code follow on successive falling clock edges (starting with the least-significant bit). These are followed by an odd-parity bit and then a high-level stop bit.



**Figure 9: Keyboard data transmission waveforms.**

The circuitry for the keyboard scanner is shown in Figure 10. The data bits for the scan code enter a ten-bit shift register on the falling edges of the keyboard clock. Then a decoder converts the eight-bit scan code into seven outputs that drive an LED digit. The LED digit will display the numeral associated with the scan code.

Note that the LED decoder in Figure 10 is not the standard seven-segment LED decoder you have used in the previous designs. The scan codes and the keyboard keys do not have a simple mapping. You will have to construct a decoder using the four-input LUTs in the CSoc CSL matrix. You will see that selecting the correct subset of scan code bits lets you get away with using only seven LUTs for the decoder.



**Figure 10: Block diagram of a PS/2 keyboard scanner.**

The data bits enter the shift register starting with the least significant bit. That means bit D0 appears on output **s9** and D7 appears on output **s2** after the entire scan code has been received. The parity bit and the stop bit are output on **s1** and **s0**, respectively, but you can ignore those. The relationship between the keyboard key that is pressed, the scan code bits which appear on the shift register outputs, and the activation levels for the LED digit segments is shown in Table 4.

**Table 4: PS/2 keyboard scan codes and LED segment activations.**

Key	Shift Register Output								LED Segment						
	s2	s3	s4	s5	s6	s7	s8	s9	A	B	C	D	E	F	G
"1"	0	0	0	1	0	1	1	0	0	1	1	0	0	0	0
"2"	0	0	0	1	1	1	1	0	1	1	0	1	1	0	1
"3"	0	0	1	0	0	1	1	0	1	1	1	1	0	0	1
"4"	0	0	1	0	0	1	0	1	0	1	1	0	0	1	1
"5"	0	0	1	0	1	1	1	0	1	0	1	1	0	1	1
"6"	0	0	1	1	0	1	1	0	1	0	1	1	1	1	1
"7"	0	0	1	1	1	1	0	1	1	1	1	0	0	0	0
"8"	0	0	1	1	1	1	1	0	1	1	1	1	1	1	1
"9"	0	1	0	0	0	1	1	0	1	1	1	1	0	1	1
"0"	0	1	0	0	0	1	0	1	1	1	1	1	1	1	0

There are only ten rows in the table, so it might be possible to distinguish between the ten keys using only four of the scan code bits in the shift register. It turns out that the bits that appear on the **s4**, **s5**, **s6**, and **s9** outputs of the shift register do uniquely distinguish between all ten keys. Therefore, the logic equations that describe the activation of the LED segments can all be written as functions that use only **s4**, **s5**, **s6**, and **s9** as inputs. From Table 4, you can write the logic equation for each LED decoder output as shown below (the logic operators are represented as:  $\sim$  = INVERT,  $\&$  = AND,  $|$  = OR). To make the equations simpler, the decoder outputs that control segments A, B, C, D, F, and G are written using the maxterms in their columns of Table 4. The decoder output that drives segment E is written using the minterms in its column.

$$\sim\text{SEGA} = \sim s4 \& s5 \& \sim s6 \& \sim s9 \quad | \quad s4 \& \sim s5 \& \sim s6 \& s9$$

$$\sim\text{SEGB} = s4 \& \sim s5 \& s6 \& \sim s9 \quad | \quad s4 \& s5 \& \sim s6 \& \sim s9$$

$$\sim\text{SEGC} = \sim s4 \& s5 \& s6 \& \sim s9$$

$$\sim\text{SEGD} = \sim s4 \& s5 \& \sim s6 \& \sim s9 \quad | \quad s4 \& \sim s5 \& \sim s6 \& s9 \quad | \quad s4 \& s5 \& s6 \& s9$$

$$\text{SEGE} = \sim s4 \& s5 \& s6 \& \sim s9 \quad | \quad s4 \& s5 \& \sim s9 \quad | \quad \sim s4 \& \sim s5 \& \sim s6 \& s9$$

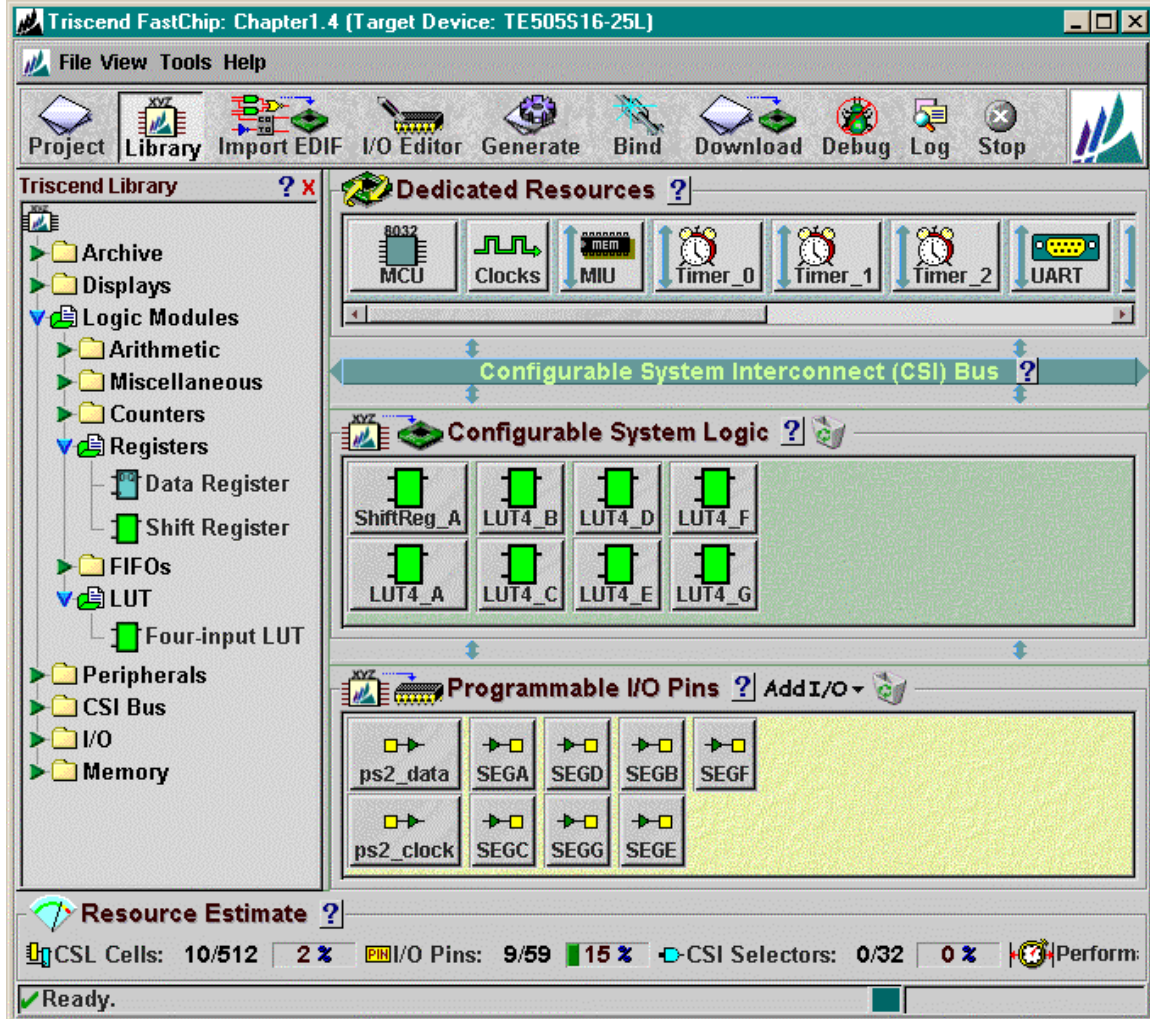
$$\sim\text{SEGF} = \sim s4 \& s5 \& \sim s9 \quad | \quad s4 \& \sim s5 \& \sim s6 \& \sim s9 \quad | \quad s4 \& s5 \& s6 \& s9$$

$\sim\text{SEGG} = \sim s4 \ \& \ s5 \ \& \ \sim s6 \ \& \ \sim s9 \ | \ s4 \ \& \ s5 \ \& \ s6 \ \& \ s9 \ | \ \sim s4 \ \& \ \sim s5 \ \& \ \sim s6 \ \& \ s9$

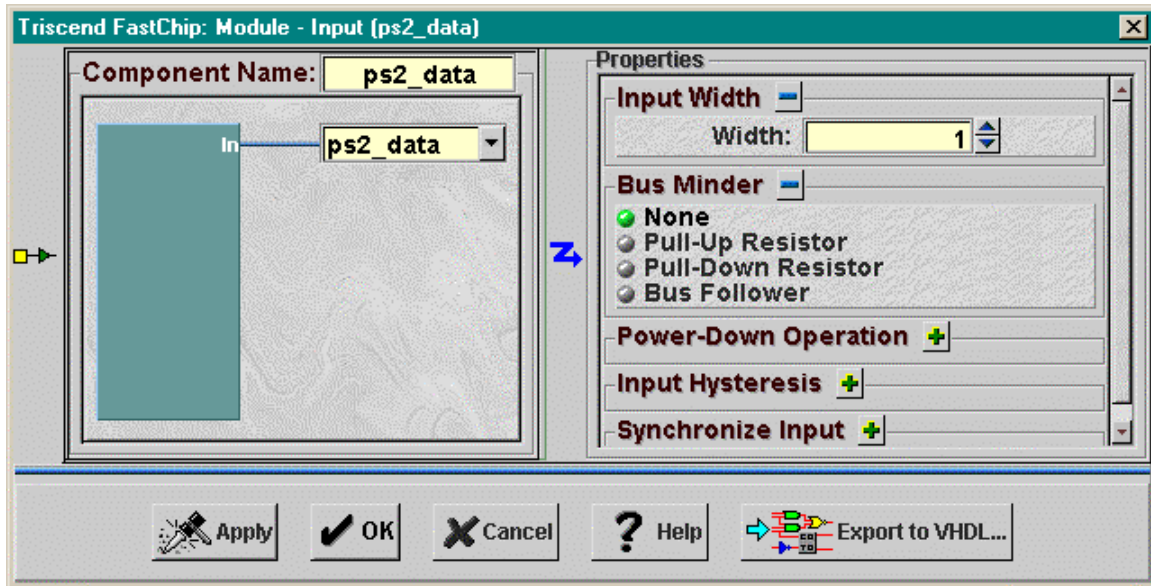
With all the preliminary design work done, start the FastChip software and type Chapter1.4 in the Project Name field. Then use the Target Device area to select the E5 device in a 128-pin LQFP with a maximum clock frequency of 25 MHz. Click on OK to get to the project window. Click on the Library icon and then drag-and-drop the following modules:

- Place two Input and seven output I/O modules in the Programmable I/O Pins area.
- Place one shift register module in the Configurable System Logic area.
- Place seven four-input LUT modules in the Configurable System Logic area.

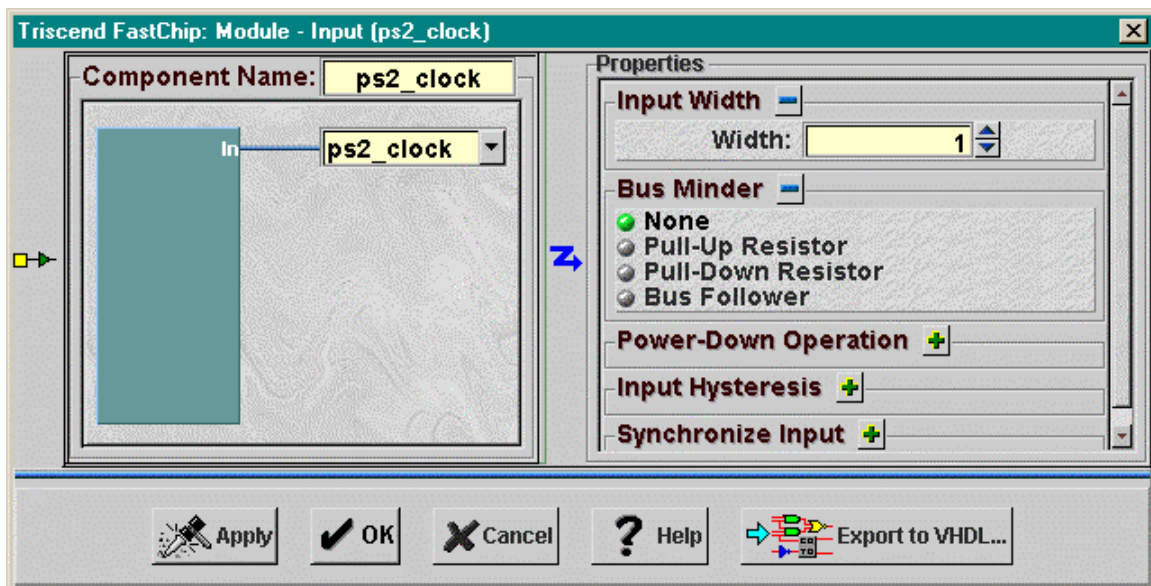
Your project window should now appear like the following (although some of your modules may have different names):




Now you can set-up the modules. Click on one of the input module icons and type `ps2_data` in the Component Name field of the resulting window. That way you can always tell what input enters your circuit through this module. Also, type `ps2_data` in the `<input>` field to name the signal that comes out of the module. Then click on OK to complete the set-up for this input.



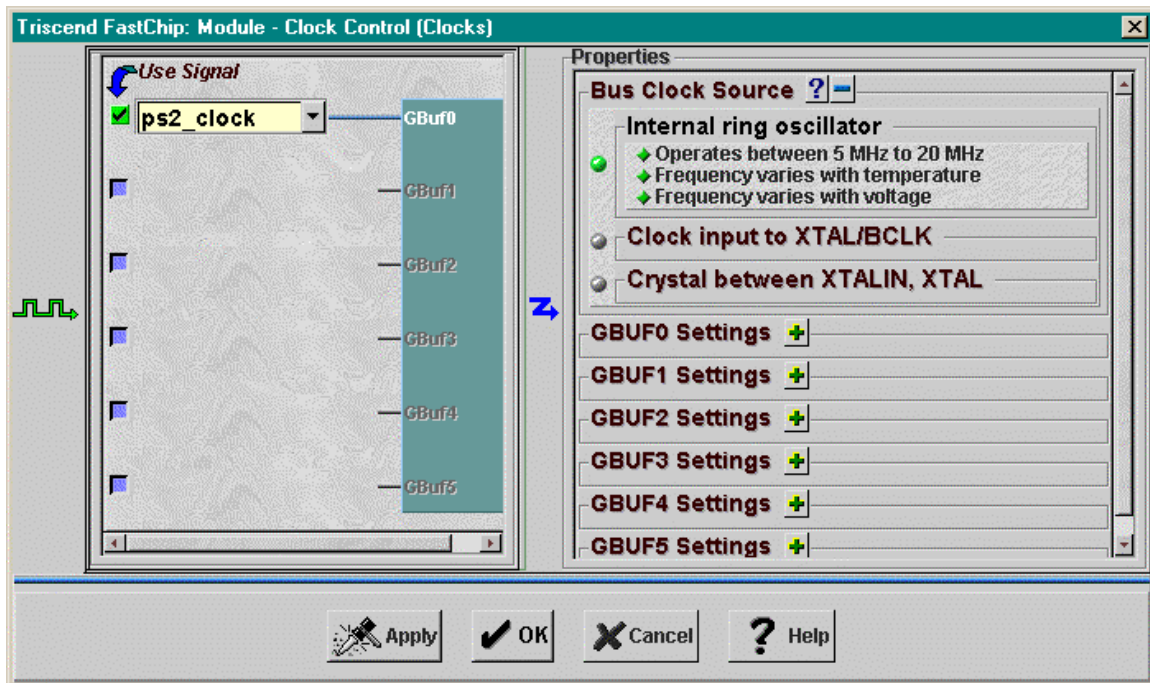
Next, click on the other input module icon and set the Component Name and `<input>` fields to `ps2_clock`. Since the clock for the shift register enters through this input, you should also open the Input Hysteresis control and activate this option. Hysteresis will reduce problems caused by noise on clock edges with slow transitions. Then click on OK to complete the set-up for this input.





To continue configuring the clock input for your circuit, click on the Clocks icon in the project window toolbar. Click on the box to the left of GBuf0 to activate this global clock buffer. Then click on the  button and select `ps2_clock` as the input to the clock buffer. The clock buffer will distribute the **ps2\_clock** signal throughout the CSL on low-skew wiring. Low-skew clocks are necessary for getting the shift register in your design to operate reliably.

You will not use the **BusClock** signal in this design, so you don't need to change the source for this clock from its default setting. Click on OK once you have completed the set-up for the clock buffer.

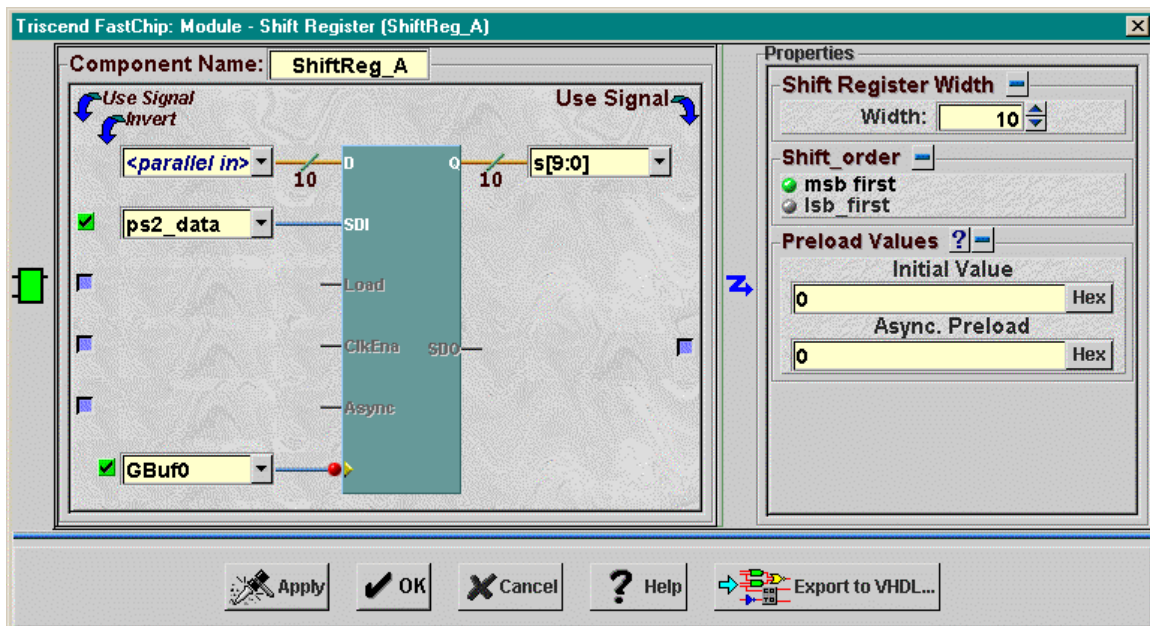


Click on the ShiftReg\_A icon next to configure the shift register module. In the Shift Register Width area, set the number of bits in the register to 10. Then type `s` in the <parallel out> field to declare the shift register output signals `s0`, `s1`, ..., `s9`.

Next click the box to the left of SDI to activate the serial data input to the shift register. Then select `ps2_data` from the drop-down list as the signal connected to this input.

Finally, select the output of global clock buffer `GBuf0` as the clock input for the shift register. `GBuf0` distributes the `ps2_clock` signal, so data bits on the `ps2_data` input will enter the shift register correctly. Since each data bit is valid on the falling edge of `ps2_clock`, you must check the box to the left of the <clock> field. This inverts the clock input to the shift register so serial data enters on the falling edge. The inversion is represented by a solid-red circle on the clock input to the shift register block.

Once the shift register is configured, click on OK.



The **s4**, **s5**, **s6**, and **s9** outputs from the shift register are inputs to the four-input LUTs. Click on one of the LUT icons (**LUT\_A**, for example) and connect the **s4**, **s5**, **s6**, and **s9** signals to the **I0**, **I1**, **I2**, and **I3** inputs, respectively, using the drop-down lists. Then type the name of one of the signals connected to an LED segment (**SEGA** in this example) into the <output> field.

Finally, type the equation for the logic function into the LUT Equation field. Rather than using the variables **s[4]**, **s[5]**, **s[6]**, and **s[9]**, the equation must be expressed in terms of the indices of the **I0**, **I1**, **I2**, and **I3** inputs to the LUT, respectively. So the equation

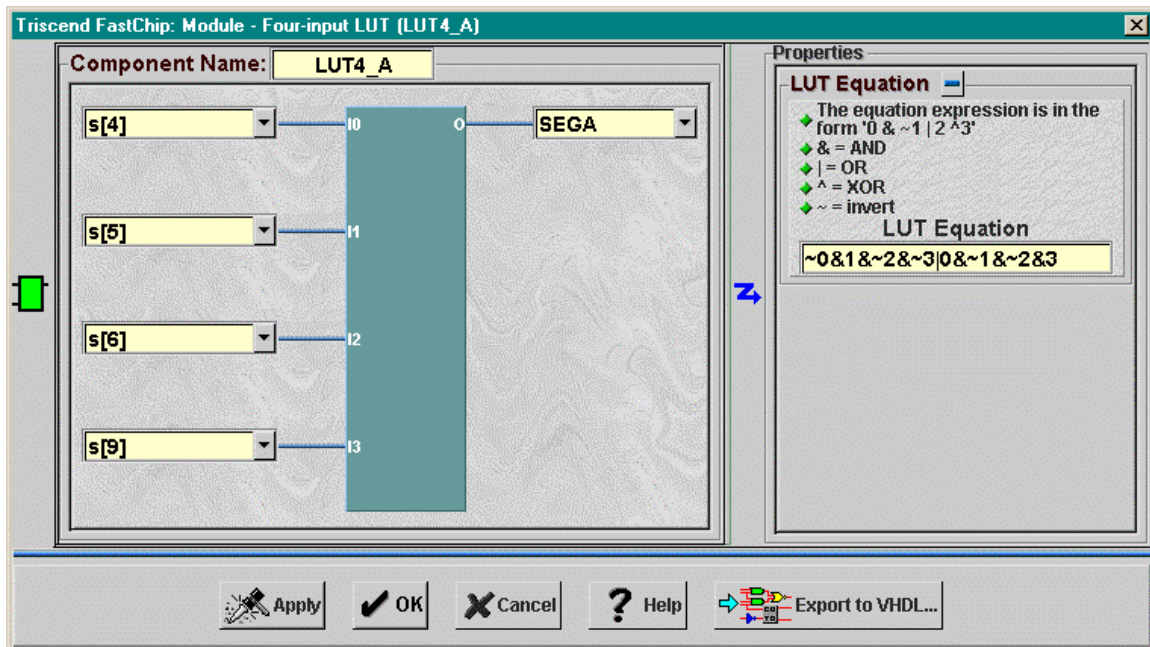
$$\sim\text{SEGA} = \sim s4 \& s5 \& \sim s6 \& \sim s9 \mid s4 \& \sim s5 \& \sim s6 \& s9$$

translates to

$$\sim\text{SEGA} = \sim 0 \& 1 \& \sim 2 \& \sim 3 \mid 0 \& \sim 1 \& \sim 2 \& 3$$

Enter the right-hand side of the translated equation into the LUT Equation field. Don't worry about the logical inversion of **SEGA** right now. You will handle that later using the inverter in an output buffer.

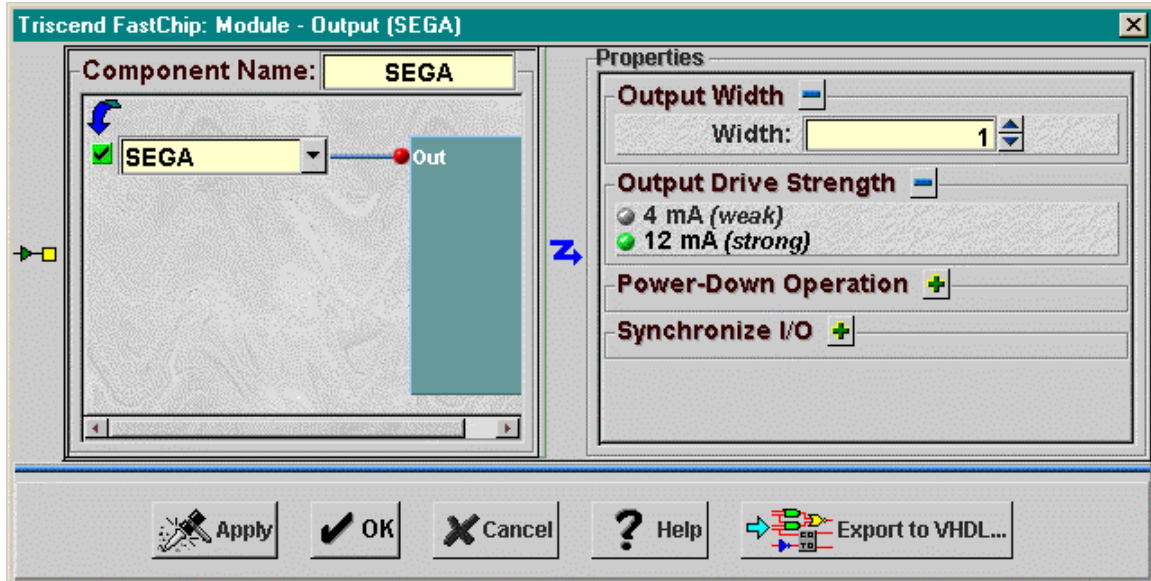
Repeat these operations for the other LUTs to generate the **SEGB**, **SEGC**, ..., **SEGG** signals using the logic equations shown previously.



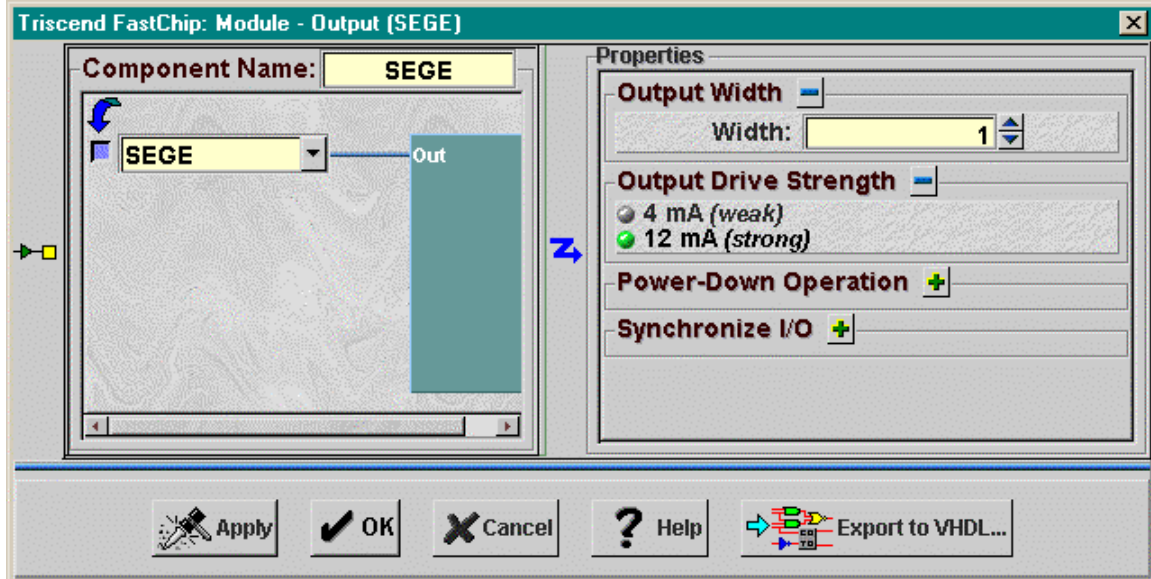
Once all the LUTs are configured, click on one of the output module icons. Connect one of the LUT output signals (**SEGA**, for example) to the output module using the drop-down list of the <output> field. Since the **SEGA** equation is logically inverted, click on the checkbox to the left of the signal name. This activates an inverter in the output module. Next, click on the 12 mA button to increase the current capabilities of the output module since it will be driving an LED segment. Then type the output signal name into



the Component Name field so you can identify which output module drives which LED segment. Then click on OK.



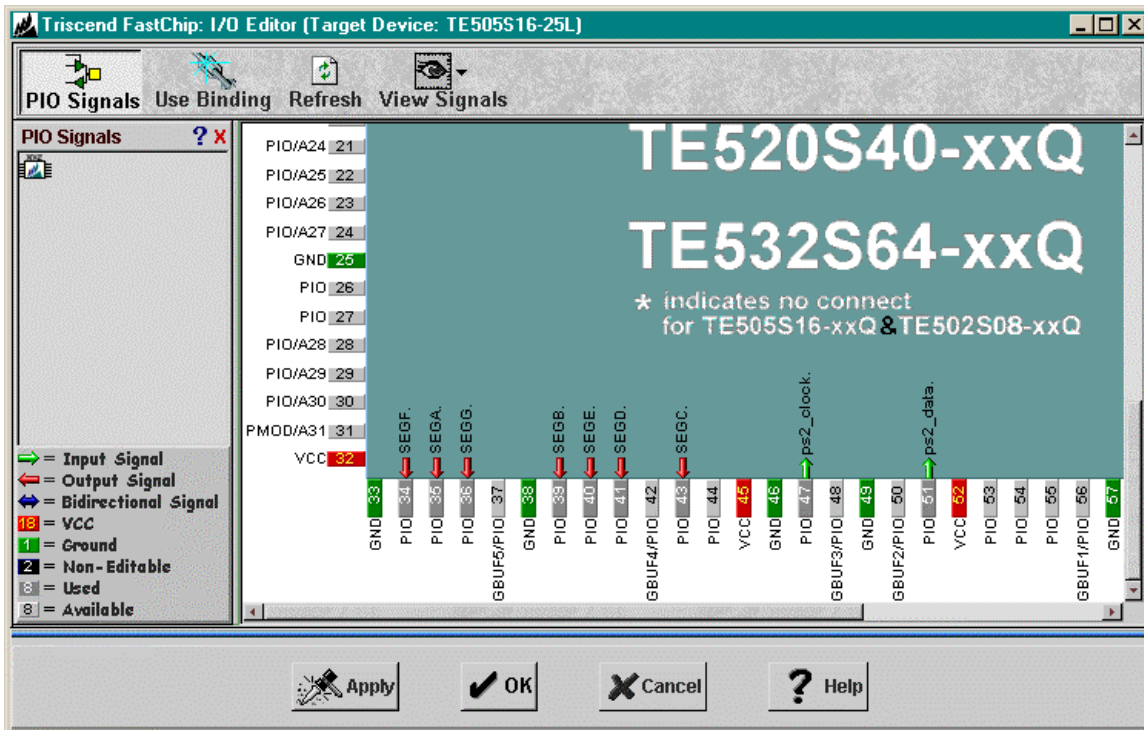
Repeat the steps described above for each of the **SEGB**, **SEGC**, **SEGD**, **SEGF**, and **SEGB** outputs. Also use the same steps for the **SEGE** output except do not activate the output inverter as it is not required by this logic equation.



Now that all the modules are internally interconnected, open the **I/O Editor** window and assign I/O pins as follows:

**Table 5: Pin assignments and functions for the keyboard scanner design.**

Signal	Pin	CSoc Board Resource
ps2_clock	47	PS/2 connector clock input
ps2_data	51	PS/2 connector data input
7seg_E. SEGA	35	LED digit segment A
7seg_E. SEGB	39	LED digit segment B
7seg_E. SEGC	43	LED digit segment C
7seg_E. SEGD	41	LED digit segment D
7seg_E. SEGE	40	LED digit segment E
7seg_E. SEGF	34	LED digit segment F
7seg_E. SEG G	36	LED digit segment G

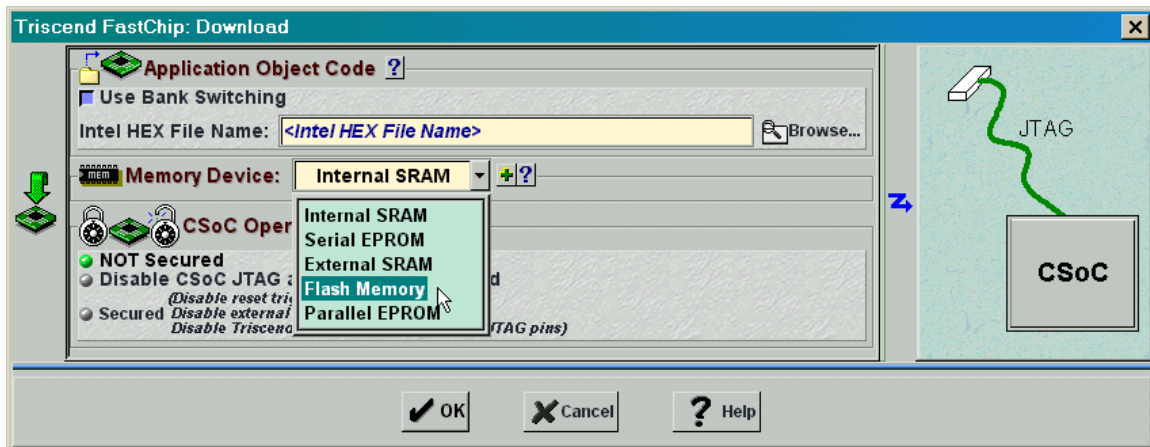


When the pin assignments are complete, run the bind and download operations. Then connect a keyboard with a PS/2 plug to the PS/2 connector of the CSoC Board. When you press keys "0"–"9", you will see the corresponding numeral displayed on the LED of the CSoC Board.

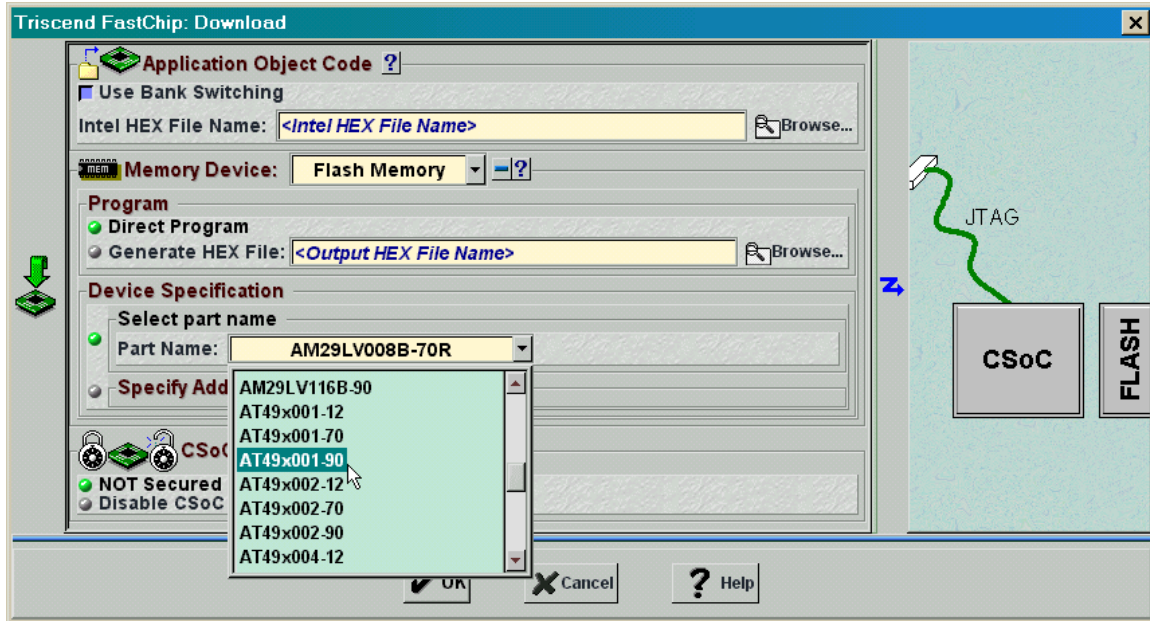
## Storing Designs in the CSoC Board Flash RAM

Up to now, all your designs have been stored in the internal memory of the CSoC. When you remove power from the CSoC Board, the CSoC configuration is lost and has to be reloaded the next time you use the board. But the CSoC Board does have a Flash RAM which can store the configuration even when the power is off and restore it to the CSoC when power returns.

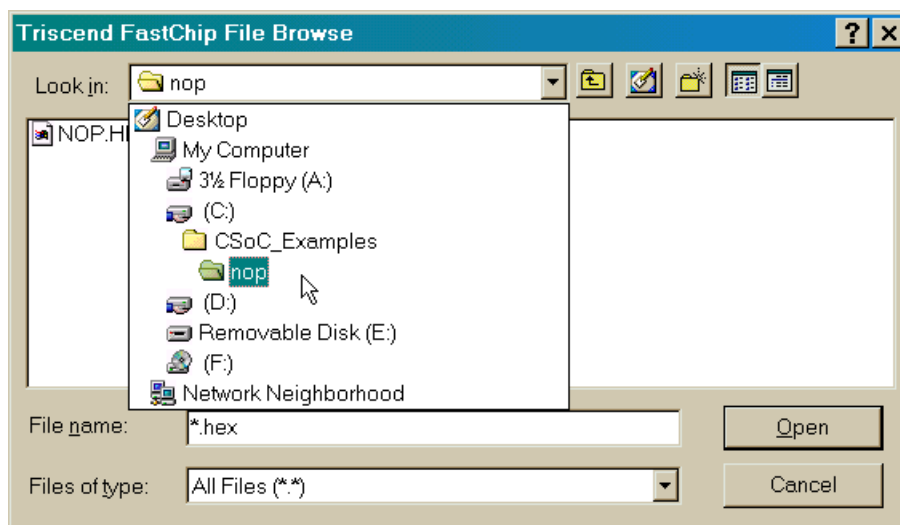
To store the configuration for a design in the Flash RAM, open the **Download** window and select Flash Memory from the Memory Device drop-down list.



Now a Device Specification area will appear in the Download window. The Select part name area has a drop-down list with the identifiers for many types of Flash RAM chips. Scroll down the list and highlight AT49X001-90 to select the type of Flash RAM device on the CSoC Board (an Atmel 1 Mbit Flash RAM with 90 ns access time).

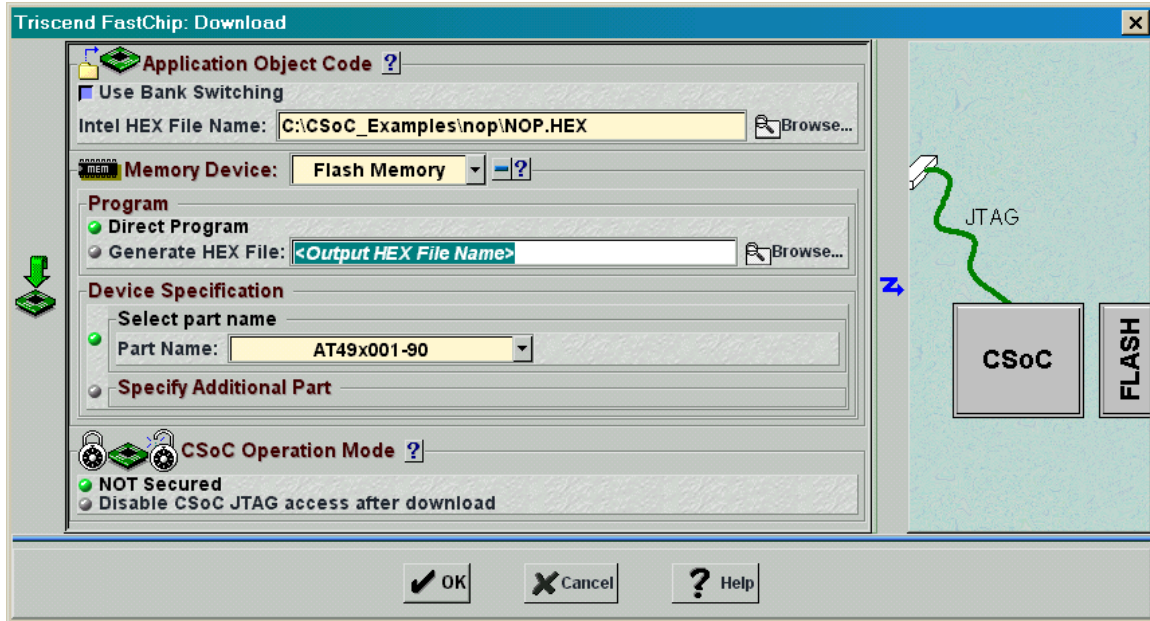


At this point, you could click on the OK button and the configuration file would be programmed into the Flash RAM on your CSoC Board. However, you may find your design operates erratically when the CSoC configures itself from the Flash RAM. That's because you are programming the CSL without also loading a program to be executed by the 8032 microcontroller in the CSoC. Therefore, the microcontroller will execute whatever it finds in the Flash RAM (which is usually garbage). This can lead to strange behavior. So you need to load a simple program into the Flash RAM that will keep the microcontroller busy and out of trouble. I have provided a simple program in the nop folder that keeps the microcontroller in an infinite-loop wherein it does nothing. You should click on the Browse button in the Application Object Code area of the **Download** window and steer your way into the nop folder as shown below.



Highlight the NOP.HEX file and click Open. The path to the NOP.HEX file will appear in the Intel HEX File Name box of the **Download** window.





Before initiating the download, make sure your CSoC Board is attached to the 9V DC power supply and it is connected to the parallel port of your PC with the downloading cable. Also check that the shunts on jumpers J8 and J9 of your CSoC Board are set for programming the Flash RAM (see Appendix A2). Click on the Download icon on the toolbar and then click on OK in the **Download** window that appears. This initiates the downloading of the configuration file into the Flash RAM of your CSoC Board. FastChip will initiate a connection to the CSoC Board, erase the Flash RAM, and then program the Flash with the CSL circuit configuration and the microcontroller instructions in the NOP.HEX file. You will notice that it takes longer to download into the Flash RAM than downloading into the internal CSoC RAM.

After the Flash RAM is programmed, you can remove power from your CSoC Board. When you restore power to your board, you will find your design is still programmed into it. For example, the CSoC Board could display the keys pressed on a PS/2 keyboard without first loading Design 1.4 into the board (provided you stored Design 1.4 in the Flash RAM). That's because the CSoC automatically loads itself from the Flash RAM whenever power is applied.