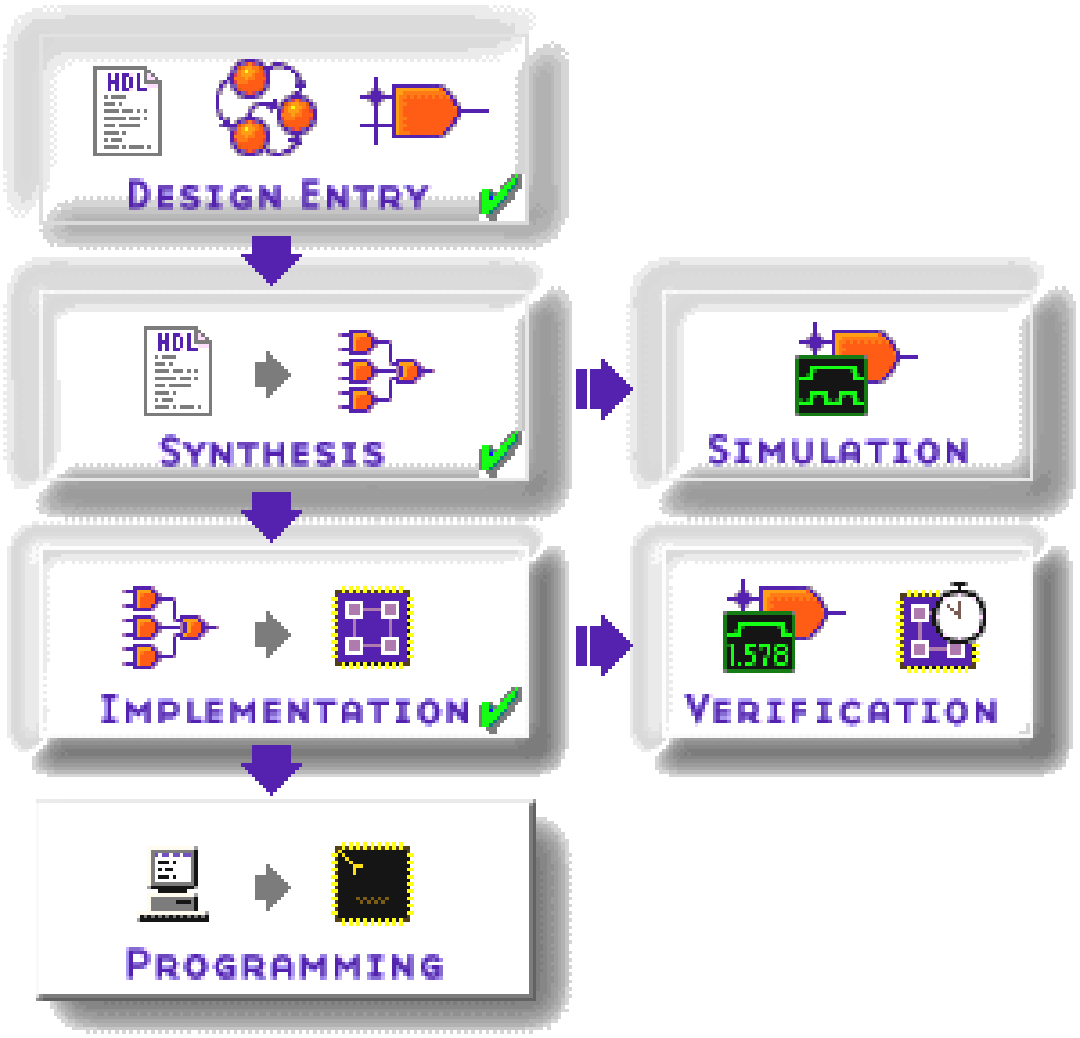
Pragmatic Logic Design

With XILINX Foundation 2.1i



David E. Vanden Bout

XESS Corp

5

Using RAM

In this chapter you will learn how to:

- Interface to an external RAM with a programmable device.
- Use the internal RAMs found in the XC4000 FPGAs.
- Create RAM modules using the Core Generator.

RAM

Eventually you will need to incorporate RAM into one of your designs. While you can build multi-bit registers from the flip-flops in a CPLD or FPGA, it is more efficient to use an external RAM chip or a specialized internal RAM-block when you need to store larger amounts of data.

In this chapter we will build a simple design that reads a set of data bytes from RAM, writes the 2's-complement of the byte values back into the RAM, sums the complemented data values and then displays the sum on the seven-segment LED. We will do two different versions of this design:

1. The first version will store the data values in the external asynchronous, byte-wide RAM found on the XS40 and XS95 Boards.
2. The second version will store the data values using the internal synchronous, distributed RAM contained in the XC4000 FPGA on the XS40 Board.

Using an External Asynchronous RAM

The first version of the RAM summation circuit has the design hierarchy shown in Figure 12. The root module of the design manages the interface to the external asynchronous RAM and sums the data while the lower-level module displays a four-bit hexadecimal value on a seven-segment display.

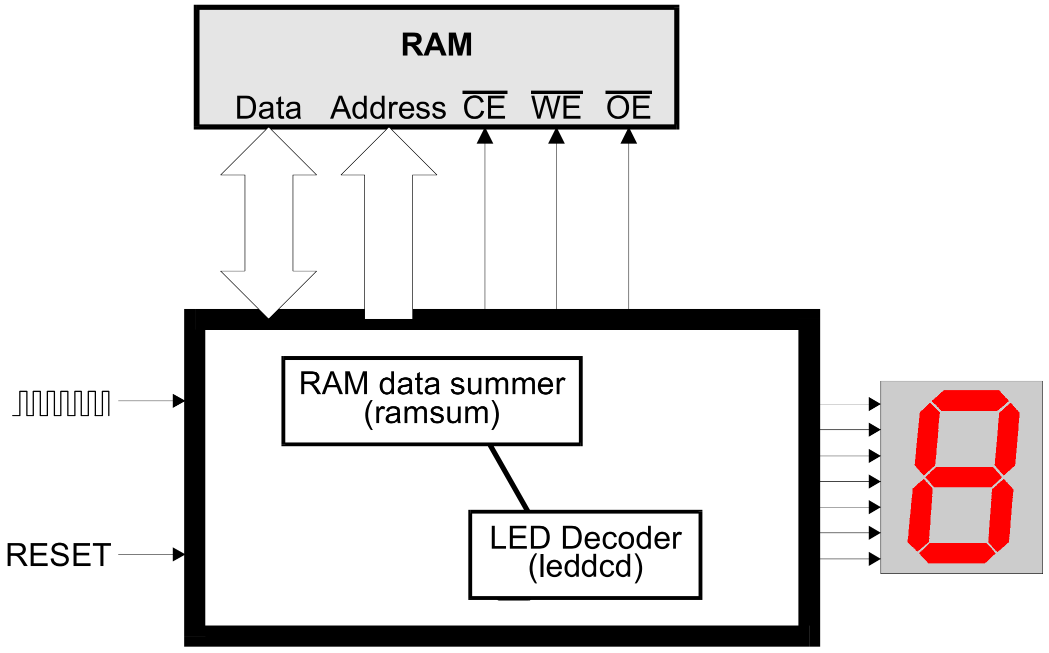
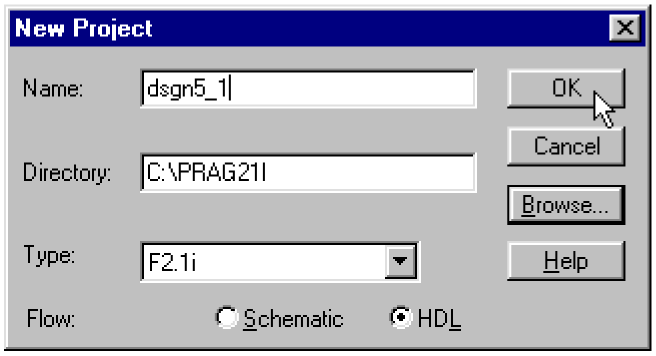


Figure 12: Design hierarchy for a logic circuit that displays the summation of data in the RAM.

Each of these modules is described by a VHDL file stored in the ***dsgn5_1*** project directory that was created as follows.


New Project

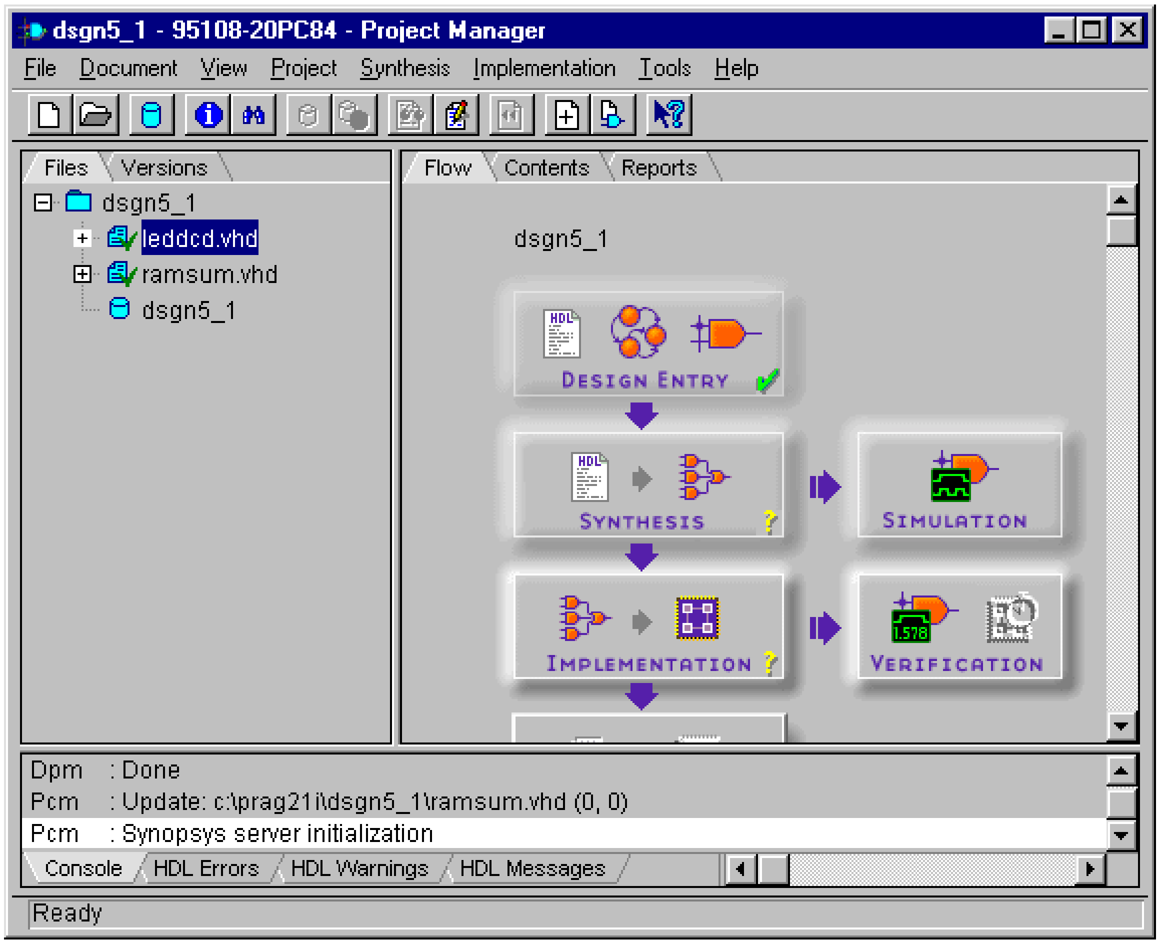
Name: dsgn5_1

Directory: C:\PRAG21I

Type: F2.1i

Flow: Schematic [✓ HDL]

After the VHDL files for the modules were created and added to the project, the **Project Navigator** window appears as follows. Now I will describe the contents of each VHDL file.


dsgn5_1 - 95108-20PC84 - Project Manager
File Document View Project Synthesis Implementation Tools Help
Files Versions
dsgn5_1
leddcd.vhd
ramsum.vhd
dsgn5_1
Flow Contents Reports
dsgn5_1
DESIGN ENTRY
SYNTHESIS
SIMULATION
IMPLEMENTATION
VERIFICATION
Console HDL Errors HDL Warnings HDL Messages

```
Dpm   : Done
Pcm   : Update: c:\prag21i\dsgn5_1\ramsum.vhd (0, 0)
Pcm   : Synopsys server initialization
```

Ready

The LED Decoder Module

This LED decoder circuit is almost identical to the one in Chapter 3 except for the addition of a blanking input signal that causes all the LED segments to turn off. This signal will be used to blank the display to separate the digits when displaying a multi-digit hexadecimal number.

The VHDL code for the LED decoder is shown in Listing 4. This code is stored in the leddcd.vhd file in the ***dsgn5_1*** project directory.

Listing 4: VHDL code for the seven-segment LED decoder module.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

package leddcd_pckg is

component leddcd
  port (
    blank: in STD_LOGIC;   -- active-high blanking input
    d    : in UNSIGNED (3 downto 0);
    s    : out STD_LOGIC_VECTOR (6 downto 0)
  );
end component;

end leddcd_pckg;


library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity leddcd is
  port (
    blank: in STD_LOGIC;  -- active-high blanking input
    d    : in UNSIGNED (3 downto 0);
    s    : out STD_LOGIC_VECTOR (6 downto 0)
  );
end leddcd;

architecture leddcd_arch of leddcd is
signal s_tmp: STD_LOGIC_VECTOR(6 downto 0);
begin
  with d select
  s_tmp <= "1110111" when "0000", -- 0
           "0010010" when "0001", -- 1
           "1011101" when "0010", -- 2
           "1011011" when "0011", -- 3
           "0111010" when "0100", -- 4
           "1101011" when "0101", -- 5
           "1101111" when "0110", -- 6
           "1010010" when "0111", -- 7
           "1111111" when "1000", -- 8
           "1111011" when "1001", -- 9
           "1111110" when "1010", -- A
           "0101111" when "1011", -- b
           "1100101" when "1100", -- C
           "0011111" when "1101", -- d
           "1101101" when "1110", -- E
           "1101100" when others; -- F

  -- zero the outputs if the blanking signal is high,
  -- otherwise output the LED digit bit pattern
  s <= "0000000" when blank='1' else s_tmp;
end leddcd_arch;
```

The Root Module

The root module sequences through three main phases:

Phase 1: Starting from an upper address of RAM and proceeding to address zero, the value stored at each RAM address is read and the two's-complement is computed and written back to the same address.

Phase 2: Restarting from the upper address and proceeding to address zero, each value is read from RAM and added to a sum register.

Phase 3: The sum is displayed on the seven-segment LED by blanking the LED segments for a long interval to signal the start of the sum, then the hexadecimal digit for the upper four bits of the sum are displayed, then the LEDs are blanked for a shorter interval and then the hexadecimal digit for the lower four bits is displayed. Then this four-step display process repeats.

The VHDL code for the root module is in the ramsum.vhd (Listing 5). Some highlights from the code are given below.

Line 4: The root module accesses the component declaration for the LED decoder by using the `leddcd_pckg` package that is part of the `WORK` library. The `WORK` library is an *implicit library* that has every project module as a member. We could have explicitly created a library and added the leddcd.vhd file to it as we did in Chapter 3, but using the `WORK` library is a bit simpler.

Lines 6–17: The interface to the design is declared here. The reset and clock inputs drive the actions of the state machine that controls the operation of the circuit. Data is passed to and from the RAM using the address and data buses along with the chip-enable, write-enable and output-enable control signals. (Note that the RAM data bus is declared as an `inout` since the same signals are used to get data from the RAM as to send data to it.)

Lines 20–21: The four-bit RAM address register is declared on these lines as well as the constant for the address of the upper end of the RAM data that will be summed. For this example, the circuit will complement and sum eleven bytes of data from address zero to ten, inclusive.

Lines 22–23: Two registers that are the same width as the RAM data bus are declared here. One register holds the current value read from the RAM while the other holds the sum of the RAM data values.

Lines 24–27: These lines declare a time delay register and the constants for the time intervals involved with the display of the hexadecimal digits in the sum.

Lines 28–29: These lines declare a four-bit bus for sending the hexadecimal digit to the LED decoder and a control signal to force the LED display to blank.

Lines 31–33: The nine states of the state machine are declared along with a register that holds the current state.

Line 37: This is the start of the process that computes the next state for the state machine given the current state, RAM address and delay timer value. The values for the RAM address, data and control signals are also generated in this process.

Lines 40–50: The default outputs for this process are defined here. The state, RAM address, summation and RAM byte registers all retain their current values unless explicitly changed within the process body. The delay register is decremented. The LED display is blanked. The RAM is enabled, but any read or write operations are disabled. The data bus is tristated to remove any chance of contention between the FPGA or CPLD and the RAM.

Line 52: This is the start of the `case` statement that computes the outputs from this process based on the current state stored in the `st_r` register.

Lines 53–55: The `init` state initializes the state machine for the start of the loop that complements the contents of RAM. The address register is set to point to the upper bound of the RAM data range and the state machine is moved to the start of the two's-complement loop (`invertr`).

Lines 56–59: The `invertr` state activates the outputs of the external RAM. The FPGA or CPLD will tristate its own outputs to the RAM data bus so that there is no contention (the default statement handles this). The value from RAM is complemented and stored in the RAM byte register. Then the state machine is moved to the state where the complemented data is written back to the RAM (`invertw`).

Lines 60–63: The `invertw` state activates the write-enable of the external RAM during the second half of the clock cycle (when the clock is low). The value in the RAM byte register is sent out to the RAM on the data bus. Then the state machine is moved to a NOP state to terminate the RAM write (`invertnop`).

Lines 64–74: The `invertnop` state keeps the RAM address stable while the write-enable returns to its quiescent state (the default operation statements handle this). The value in the RAM byte register remains on the output bus to the RAM for the same reason. If the current RAM address is zero indicating the complementation loop is finished, then the RAM address is reloaded with the starting address of the (now complemented) RAM data. Then the state machine is moved to the start of the summation loop (`add`). If not all the RAM data has been complemented yet, then the RAM address is decremented and the state machine returns to the start of the complementation loop (`invertr`).

Lines 75–85: The `add` state activates the outputs of the external RAM. The value from RAM is added to the summation register. If the current RAM address is zero indicating the summation loop is finished, then the time delay register is loaded with the initial blanking interval for the LED display. Then the state machine is moved to the start of the display loop (`display_blank`). If all the RAM data has not been summed, then the RAM address is decremented and the state machine stays in the `add` state.

Lines 86–91: The `display_blank` state blanks the LED display and decrements the delay timer (the default operation statements handle this). Once the delay timer reaches zero, it is reloaded with the time interval for display of a digit and the state machine is moved into the `display_upper_digit` state.

Lines 92–99: The `display_upper_digit` state unblanks the LED display and sends the upper four bits of the sum to the LED decoder. The delay timer is also decremented. Once the delay timer reaches zero, it is reloaded with the time interval for blanking the display between digits and then the state machine is moved into the `display_interdigit` state.

Lines 100–105: The `display_interdigit` state blanks the LED display and decrements the delay timer (the default operation statements handle this). Once the delay timer reaches zero, it is reloaded with the time interval for display of a digit and the state machine is moved into the `display_lower_digit` state.

Lines 106–113: The `display_lower_digit` state unblanks the LED display and sends the lower four bits of the sum to the LED decoder. The delay timer is also decremented. Once the delay timer reaches zero, it is reloaded with the time interval for blanking the display before the sum is displayed and then the state machine is moved into the `display_blank` state.

Lines 120–134: This process updates the state, address, data, sum and time delay registers with their new values on the rising edge of the clock. The reset input synchronously clears the sum register and transfers the state machine into its starting state.

Line 136: The four-bit RAM address is padded with leading zeroes to form the complete address passed to the external RAM.

Line 137: The LED decoder is passed the blanking signal and the four-bit hexadecimal code for the digit to be displayed. The outputs of the LED decoder drive the seven-segment LEDs.

Listing 5: VHDL code for the root module.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use WORK.leddcd_pckg.all;

entity ramsum is
  port (
     rst  : in STD_LOGIC;                    -- reset
     clk  : in STD_LOGIC;                    -- clock
     a    : out UNSIGNED(16 downto 0);       -- RAM address bus
     d    : inout UNSIGNED(7 downto 0);      -- RAM data bus
     ce_n : out STD_LOGIC;                   -- RAM chip-enable
     we_n : out STD_LOGIC;                   -- RAM write-enable
     oe_n : out STD_LOGIC;                   -- RAM output-enable
     s    : out STD_LOGIC_VECTOR(6 downto 0) -- outputs to LED segments
  );
end ramsum;

architecture ramsum_arch of ramsum is
signal addr_r, next_addr: UNSIGNED(3 downto 0); -- address register
constant maxaddr   : UNSIGNED := TO_UNSIGNED(10,addr_r'length);  -- upper address
signal b_r, next_b : UNSIGNED(d'length-1 downto 0);   -- stores byte from RAM
signal sum_r, next_sum : UNSIGNED(d'length-1 downto 0); -- stores sum of RAM bytes
signal delay_r, next_delay : UNSIGNED(22 downto 0);   -- delay counter
constant blank_dly : UNSIGNED := TO_UNSIGNED(5_000_000,delay_r'length);
constant interdigit_dly : UNSIGNED := TO_UNSIGNED(1_600_000,delay_r'length);
constant digit_dly : UNSIGNED := TO_UNSIGNED(2_500_000,delay_r'length);
signal digit      : UNSIGNED(3 downto 0); -- LED hex digit to display
signal blank      : STD_LOGIC;           -- LED digit blanking signal
-- states for the state machine
type state is (init,invertr,invertw,invertnop,add,display_blank,
        display_upper_digit,display_interdigit,display_lower_digit);
signal st_r, next_st    : state;   -- state register
begin

-- this process computes the actions of the state machine in each state
process(clk,st_r,addr_r,sum_r,b_r,delay_r,d)
begin
   -- default outputs unless otherwise specified
   next_st    <= st_r;-- next state is the same as the current state
   next_addr  <= addr_r; -- don't change the RAM address
   next_sum   <= sum_r;  -- don't update the sum register
   next_b     <= b_r;    -- don't reload the RAM byte register
   next_delay <= delay_r-1;-- decrement the delay counter
   digit   <= TO_UNSIGNED(0,digit'length); -- output a '0' LED digit
   blank   <= '1';    -- blank the LED display
   ce_n    <= '0';    -- enable the RAM
   we_n    <= '1';    -- don't write to the RAM
   oe_n    <= '1';    -- don't read from the RAM
   d       <= (others=>'Z');  -- tristate the but to the RAM

   case st_r is -- case statement for the state machine
   when init => -- initialization state
      next_addr <= maxaddr;   -- start inverting from the upper address
      next_st <= invertr;     -- enter the RAM inversion loop
   when invertr => -- read the contents of the RAM location
      oe_n <= '0';            -- enable the RAM outputs
      next_b <= TO_UNSIGNED(0,next_b'length) - d;  -- invert byte from RAM
      next_st <= invertw;     -- go to RAM-write state
   when invertw => -- write inverted byte value into same RAM location
      we_n <= clk;       -- write RAM in 2nd half of clock cycle
      d <= b_r;          -- output inverted byte value to RAM
      next_st <= invertnop;-- go to RAM no-op state
   when invertnop => -- terminate RAM-write cleanly
      d <= b_r;          -- maintain output of inverted byte value to RAM
      if addr_r = TO_UNSIGNED(0,addr_r'length) then
         -- reached the lower address of the RAM data
         next_addr <= maxaddr;-- reload register with upper address
         next_st <= add;    -- enter the summation loop
      else
         -- haven't inverted all the RAM data yet
         next_addr <= addr_r - 1;-- address the next RAM location
         next_st <= invertr;  -- return to beginning of the inversion loop
      end if;
   when add =>     -- sum the inverted data from RAM
      oe_n <= '0';       -- enable the RAM outputs
      next_sum <= sum_r + d;  -- add the RAM data to the sum
      if addr_r = TO_UNSIGNED(0,addr_r'length) then
         -- reached the lower address of the RAM data
         next_delay <= blank_dly;-- load display interval counter
         next_st <= display_blank;  -- now display the sum
      else
         -- haven't summed all the RAM data yet so stay in this state
         next_addr <= addr_r - 1;-- address the next RAM location
      end if;
   when display_blank =>-- blank the display
      if delay_r = TO_UNSIGNED(0,delay_r'length) then
         -- initial display blanking is complete
         next_delay <= digit_dly;   -- load digit display interval
         next_st <= display_upper_digit; -- now display the upper sum digit
      end if;
    when display_upper_digit => -- display the upper digit of the sum
      blank <= '0';                 -- activate the LED
      digit <= sum_r(7 downto 4);   -- display the upper 4-bits of the sum
      if delay_r = TO_UNSIGNED(0,delay_r'length) then
        -- upper digit display is complete
        next_delay <= interdigit_dly;   -- load inter-digit blanking interval
        next_st <= display_interdigit;  -- blank the display between digits
      end if;
    when display_interdigit => -- blank the display between sum digits
      if delay_r = TO_UNSIGNED(0,delay_r'length) then
        -- inter-digit display blanking is complete
        next_delay <= digit_dly;        -- load digit display interval
        next_st <= display_lower_digit; -- now display the lower sum digit
      end if;
    when display_lower_digit => -- display the lower digit of the sum
      blank <= '0';                 -- activate the LED
      digit <= sum_r(3 downto 0);   -- display the lower 4-bits of the sum
      if delay_r = TO_UNSIGNED(0,delay_r'length) then
        -- lower digit display is complete
        next_delay <= blank_dly;   -- load blanking interval between loops
        next_st <= display_blank;  -- loop and display the sum again
      end if;
    when others =>  -- error state
      next_st <= init;-- re-initialize the state machine on an erroneous state
  end case;
end process;

-- this process updates the registers on every rising clock edge
process(clk)
begin
  if clk'event and clk='1' then -- trigger on rising clock edge
    if rst='1' then    -- synchronous reset
      st_r    <= init;
      sum_r   <= TO_UNSIGNED(0,sum_r'length);
    else    -- update the registers
      st_r    <= next_st;
      sum_r   <= next_sum;
      addr_r  <= next_addr;
      b_r     <= next_b;
      delay_r <= next_delay;
    end if;
  end if;
end process;

a <= "0000000000000" & addr_r;
u1: leddcd port map(blank=>blank, d=>digit, s=>s);

end ramsum_arch;
```

Figure 13 shows the waveforms for the phase when the RAM data is complemented. RAM address N is output at the start of the `invertr` clock cycle and the RAM output-enable is activated. The data stored at address N is output by the RAM to the FPGA or CPLD where it is complemented and stored into the RAM byte register at the start of the `invertw` cycle. The complemented value in the byte register is sent back to the RAM and the RAM write-enable is pulsed low during the second half of the `invertw` cycle, thus writing the complemented data back to address N. The RAM address and data are held stable during the following `invertnop` cycle and then the entire operation is repeated for RAM address N-1.

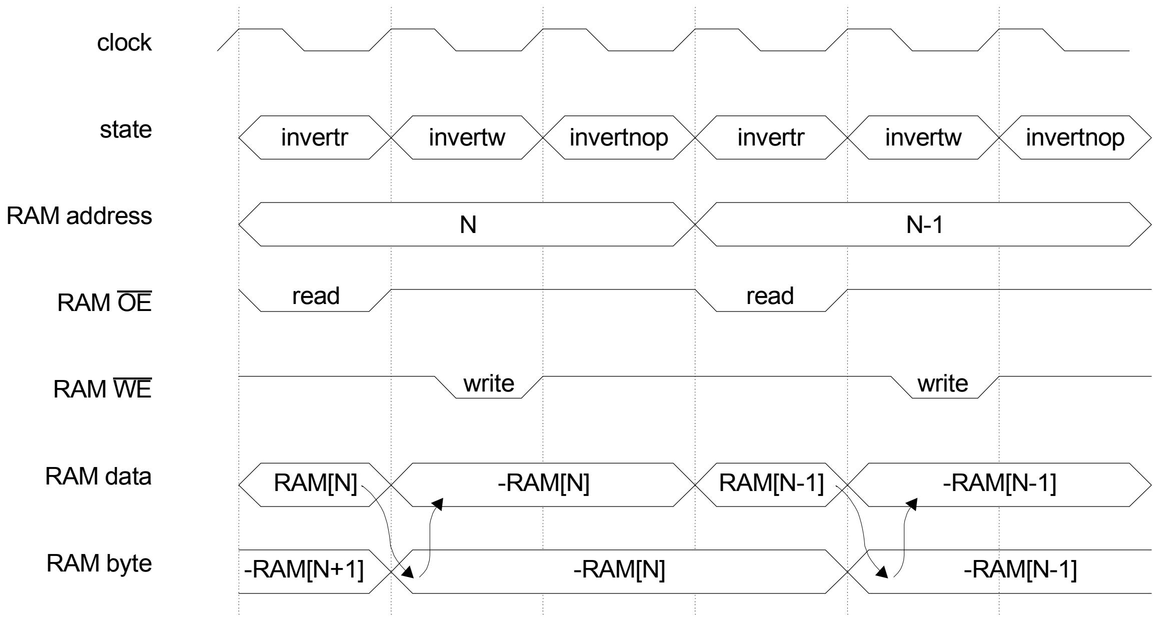


Figure 13: Timing waveforms for the asynchronous RAM summation circuit.

The timing waveforms illustrate the fundamental principles involved when writing to an asynchronous RAM:

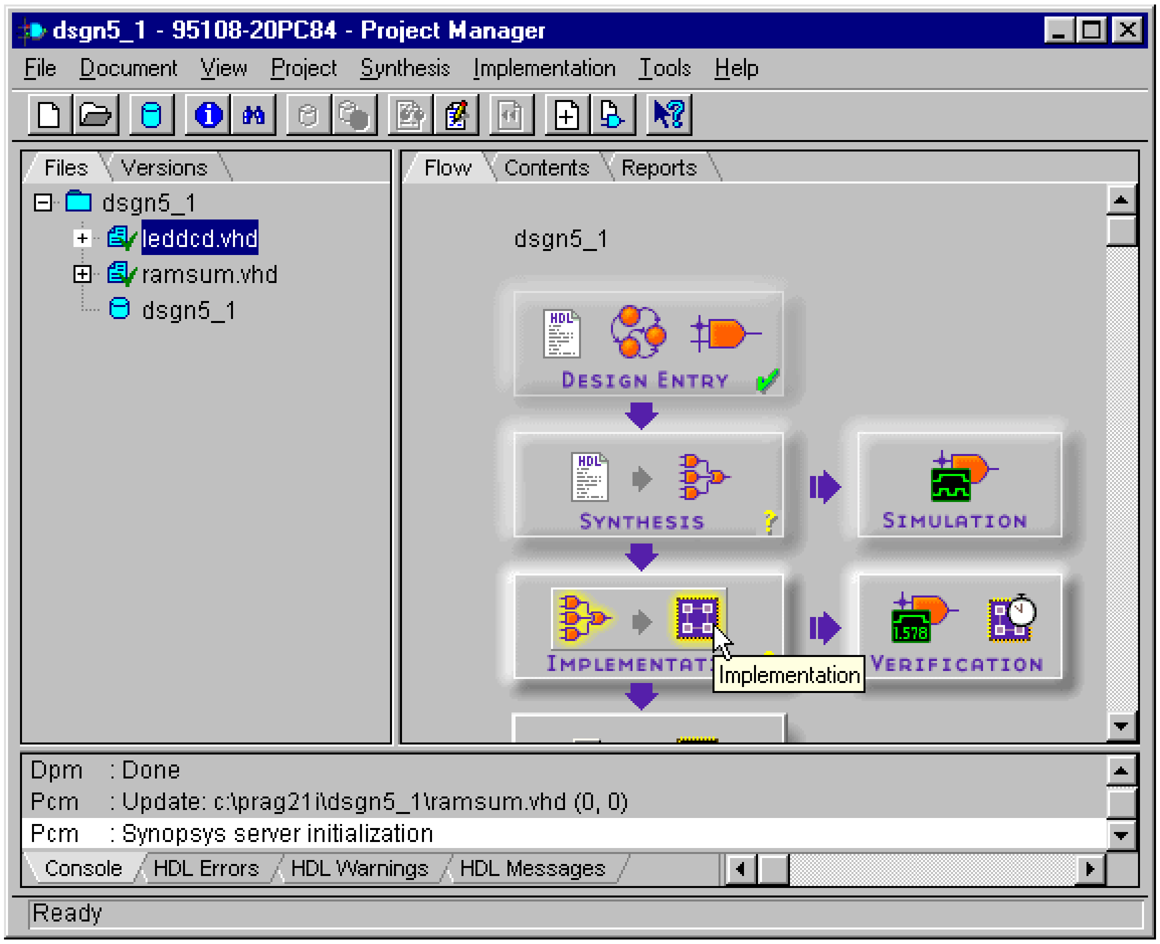
1. The address to the RAM must be held stable during the entire time the write-enable is active-low. Otherwise, data may be erroneously written to some other address instead of, or in addition to, the desired address. That's because write operations to an asynchronous RAM occur as long as the write-enable is low, not just on an edge transition.

2. The data to the RAM must be held stable at the rising edge of the write-enable pulse. Otherwise, an incorrect data value may be written to the RAM address.

For our design, note that the RAM address is stable for an entire cycle both before and after the write-enable pulse, and the RAM data is stable for an entire cycle before and after the rising edge of the write-enable pulse. This allows a large setup time for the address and data before the write-enable pulse, and provides a large hold time after the pulse.

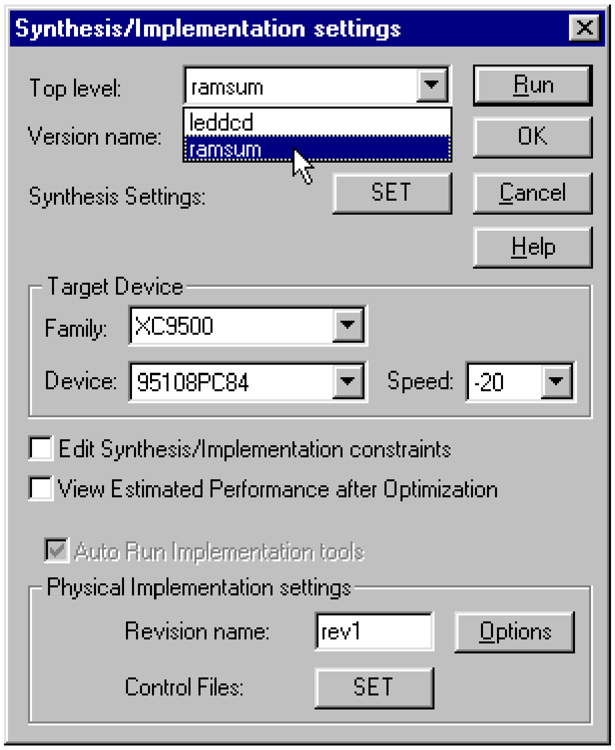
Synthesizing and Implementing the Design

Once the modules are checked for syntax and any errors are removed, we can run the synthesis and implementation tools to create the configuration bitstream for the FPGA or CPLD. Click on the Implementation icon to run the synthesizer and the implementation tools sequentially.



```
Dpm   : Done
Pcm   : Update: c:\prag21i\dsgn5_1\ramsum.vhd (0, 0)
Pcm   : Synopsys server initialization
```

We will target this design to the XS95 Board, so set the target device to be an XC95108PC84 with a –20 speed grade. Then select the **ramsum** module as the top-level module for the design.


Synthesis/Implementation settings

Top level: ramsum

leddcd
ramsum

Version name:

Synthesis Settings: SET

Target Device

Family: XC9500

Device: 95108PC84 Speed: -20

☐ Edit Synthesis/Implementation constraints

☐ View Estimated Performance after Optimization

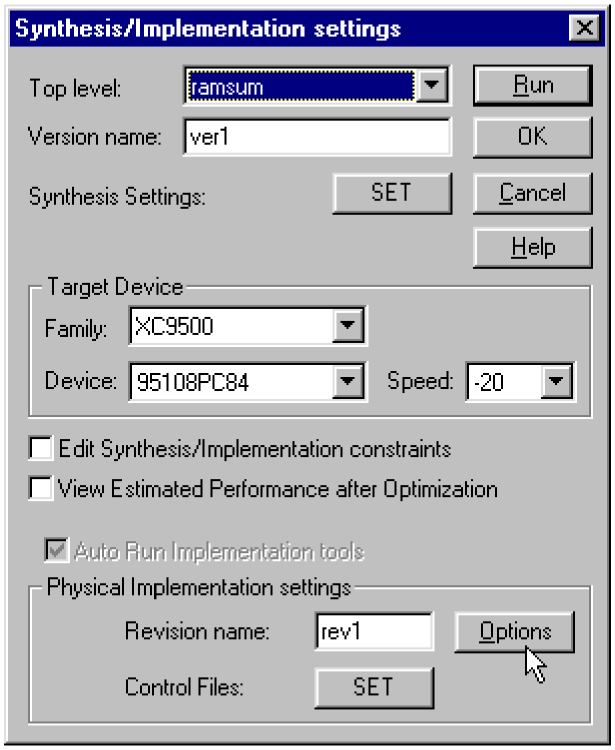
☑ Auto Run Implementation tools

Physical Implementation settings

Revision name: rev1

Control Files: SET

Next, we need to set the options for the implementation tools.


Synthesis/Implementation settings

Top level: ramsum

Version name: ver1

Synthesis Settings: SET

Target Device

Family: XC9500

Device: 95108PC84 Speed: -20

☐ Edit Synthesis/Implementation constraints

☐ View Estimated Performance after Optimization

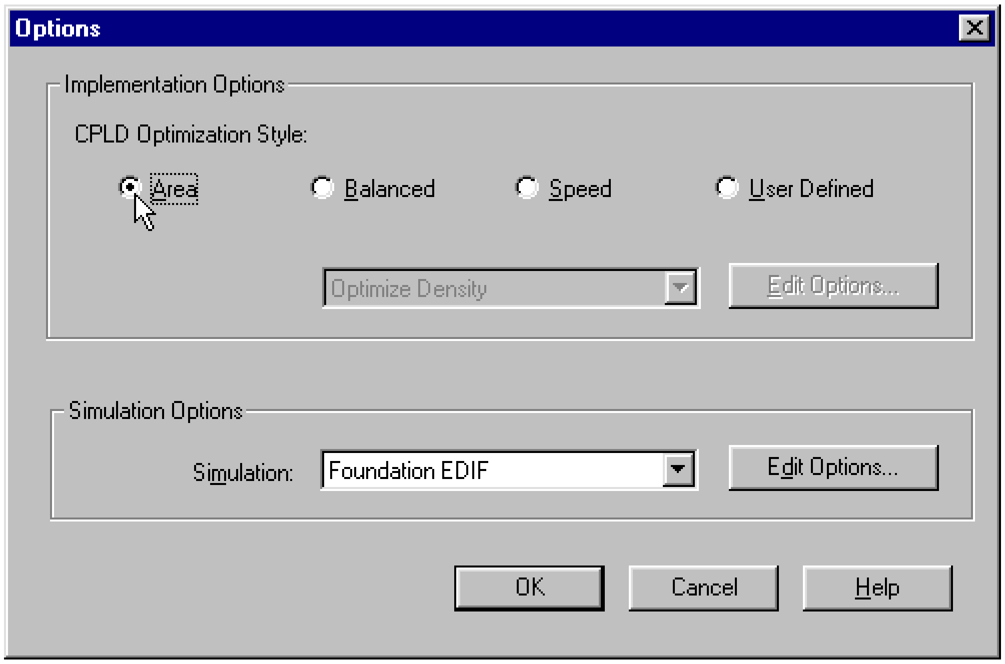
☑ Auto Run Implementation tools

Physical Implementation settings

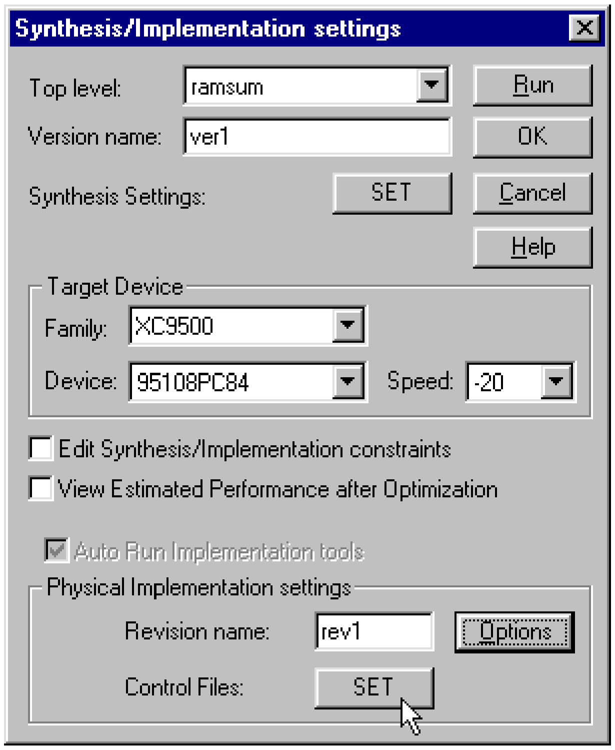
Revision name: rev1

Control Files: SET

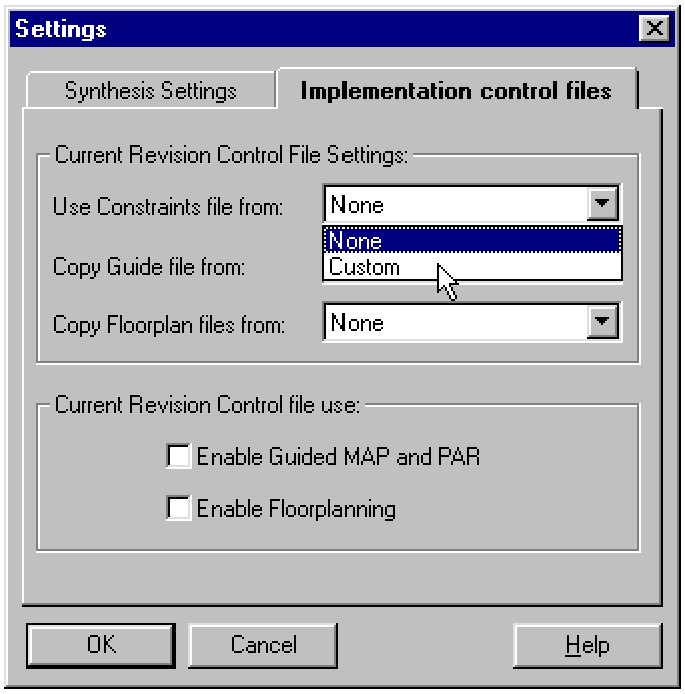
Click on the Area button to make the CPLD fitting tool emphasize logic efficiency over operational speed. This option setting is necessary because the design uses a large time delay counter and several comparators that make it difficult to fit into an XC95108 CPLD. Click on the OK button to close the window.



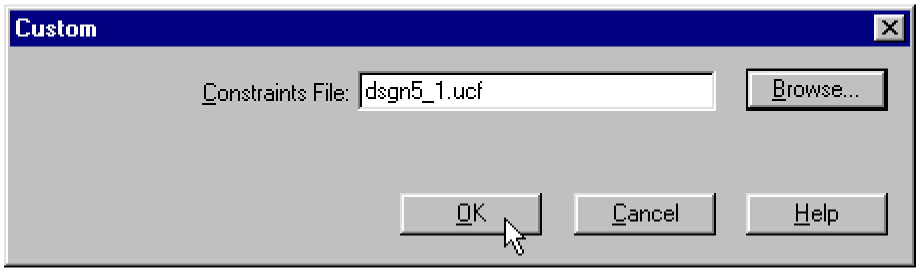
Next, click on the SET button so we can specify the constraint file that lists the pin assignments for the XS95 Board.


Synthesis/Implementation settings
Top level:
ramsum
Run
Version name:
ver1
OK
Synthesis Settings:
SET
Cancel
Help
Target Device
Family:
XC9500
Device:
95108PC84
Speed:
-20
Edit Synthesis/Implementation constraints
View Estimated Performance after Optimization
Auto Run Implementation tools
Physical Implementation settings
Revision name:
rev1
Options
Control Files:
SET

Select the Custom entry in the drop-down list of constraint files.


Settings
Synthesis Settings
Implementation control files
Current Revision Control File Settings:
Use Constraints file from: None
None
Custom
Copy Guide file from:
Copy Floorplan files from: None
Current Revision Control file use:
Enable Guided MAP and PAR
Enable Floorplanning
OK
Cancel
Help

The **Custom** window should appear with the dsgn5_1.ucf file already in the Constraints File field. If not, click on the Browse button, find this file in the top-level directory of the ***dsgn5_1*** project and select it. Then click on the OK button.



The dsgn5_1.ucf file should specify the assignments for the FPGA or CPLD pins that connect to the clock, reset, seven-segment LED and RAM address, data and control pins as shown in Figure 14. The pin assignments for the XS95 Board (which is our target for this example) are shown in Listing 6. The equivalent pin assignments for the XS40 Board are given in Listing 7.

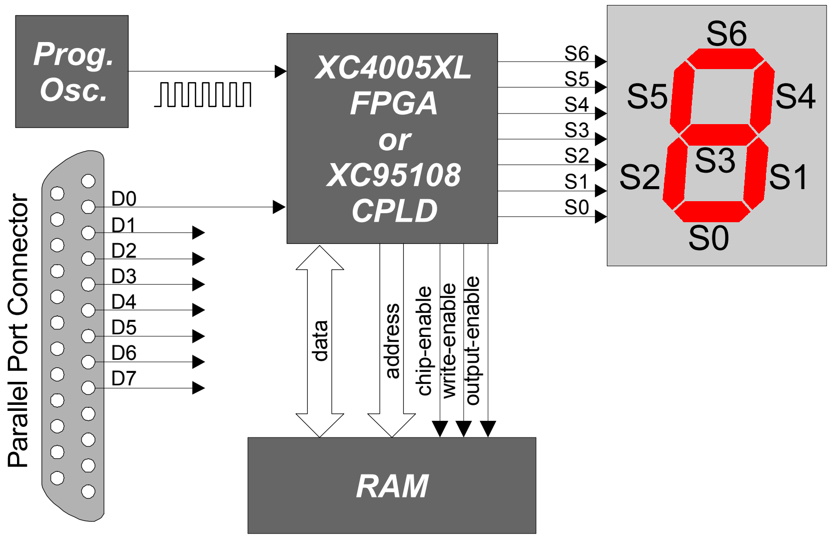


Figure 14: Connection of the external RAM, programmable oscillator, parallel port, and LED digit to the pins of the FPGA or CPLD on the XS40 or XS95 Board.

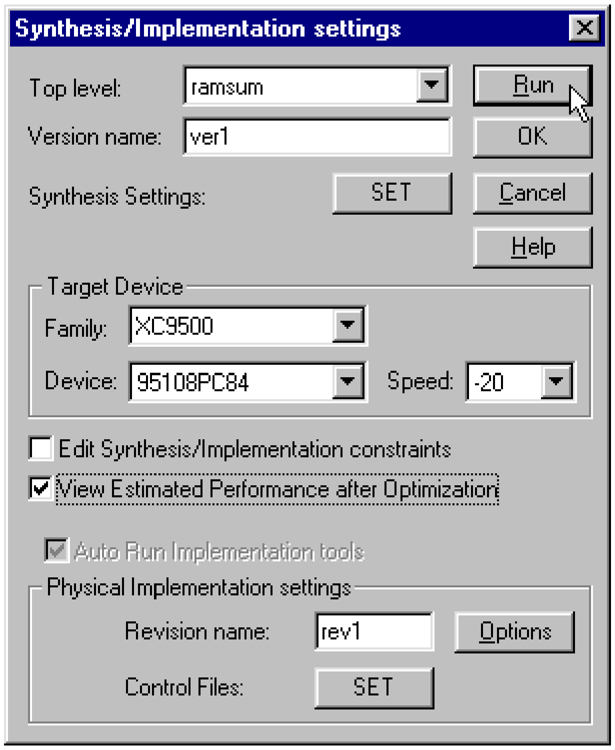
Listing 6: Pin assignments for the XS95 Board.

```
# pin assignemnts for the XS95 Board
net clk loc=p9;  # clock from programmable osc.
net rst loc=p46;   # reset from data pin D0 of parallel port
net d<0> loc=p44; # RAM data pin D0
net d<1> loc=p43; # RAM data pin D1
net d<2> loc=p41; # RAM data pin D2
net d<3> loc=p40; # RAM data pin D3
net d<4> loc=p39; # RAM data pin D4
net d<5> loc=p37; # RAM data pin D5
net d<6> loc=p36; # RAM data pin D6
net d<7> loc=p35; # RAM data pin D7
net a<0> loc=p75; # RAM address pin A0
net a<1> loc=p79; # RAM address pin A1
net a<2> loc=p82; # RAM address pin A2
net a<3> loc=p84; # RAM address pin A3
net a<4> loc=p1;   # RAM address pin A4
net a<5> loc=p3;  # RAM address pin A5
net a<6> loc=p83; # RAM address pin A6
net a<7> loc=p2;  # RAM address pin A7
net a<8> loc=p58; # RAM address pin A8
net a<9> loc=p56; # RAM address pin A9
net a<10> loc=p54;  # RAM address pin A10
net a<11> loc=p55;  # RAM address pin A11
net a<12> loc=p53;  # RAM address pin A12
net a<13> loc=p57;  # RAM address pin A13
net a<14> loc=p61;  # RAM address pin A14
net a<15> loc=p34;  # RAM address pin A15
net a<16> loc=p74;  # RAM address pin A16
net we_n loc=p63; # RAM write-enable
net oe_n loc=p62; # RAM output-enable
net ce_n loc=p65; # RAM chip-enable
net s<0> loc=p21; # LED segment S0
net s<1> loc=p23; # LED segment S1
net s<2> loc=p19; # LED segment S2
net s<3> loc=p17; # LED segment S3
net s<4> loc=p18; # LED segment S4
net s<5> loc=p14; # LED segment S5
net s<6> loc=p15; # LED segment S6
```

Listing 7: Pin assignments for the XS40 Board.

```
# pin assignments for XS40 Board
net clk loc=p13;  # clock from programmable osc.
net rst loc=p44;  # reset from data pin D0 of parallel port
net d<0> loc=p41; # RAM data pin D0
net d<1> loc=p40; # RAM data pin D1
net d<2> loc=p39; # RAM data pin D2
net d<3> loc=p38; # RAM data pin D3
net d<4> loc=p35; # RAM data pin D4
net d<5> loc=p81; # RAM data pin D5
net d<6> loc=p80; # RAM data pin D6
net d<7> loc=p10; # RAM data pin D7
net a<0> loc=p3;  # RAM address pin A0
net a<1> loc=p4;  # RAM address pin A1
net a<2> loc=p5;  # RAM address pin A2
net a<3> loc=p78; # RAM address pin A3
net a<4> loc=p79; # RAM address pin A4
net a<5> loc=p82; # RAM address pin A5
net a<6> loc=p83; # RAM address pin A6
net a<7> loc=p84; # RAM address pin A7
net a<8> loc=p59; # RAM address pin A8
net a<9> loc=p57; # RAM address pin A9
net a<10> loc=p51;  # RAM address pin A10
net a<11> loc=p56;  # RAM address pin A11
net a<12> loc=p50;  # RAM address pin A12
net a<13> loc=p58;  # RAM address pin A13
net a<14> loc=p60;  # RAM address pin A14
net a<15> loc=p28;  # RAM address pin A15
net a<16> loc=p16;  # RAM address pin A16
net we_n loc=p62; # RAM write-enable
net oe_n loc=p61; # RAM output-enable
net ce_n loc=p65; # RAM chip-enable
net s<0> loc=p25; # LED segment S0
net s<1> loc=p26; # LED segment S1
net s<2> loc=p24; # LED segment S2
net s<3> loc=p20; # LED segment S3
net s<4> loc=p23; # LED segment S4
net s<5> loc=p18; # LED segment S5
net s<6> loc=p19; # LED segment S6
```

Once the target device, top-level module, implementation options and constraint file are setup, click on the Run button to start the synthesis and implementation phases.


Synthesis/Implementation settings

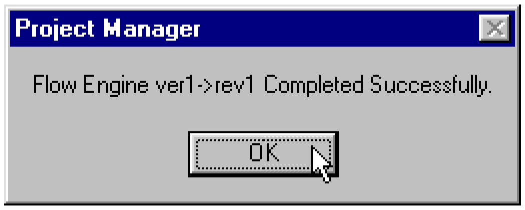
Top level: ramsum
Version name: ver1
Synthesis Settings: SET

Target Device
Family: XC9500
Device: 95108PC84
Speed: -20

☐ Edit Synthesis/Implementation constraints
☑ View Estimated Performance after Optimization
☑ Auto Run Implementation tools

Physical Implementation settings
Revision name: rev1
Control Files: SET

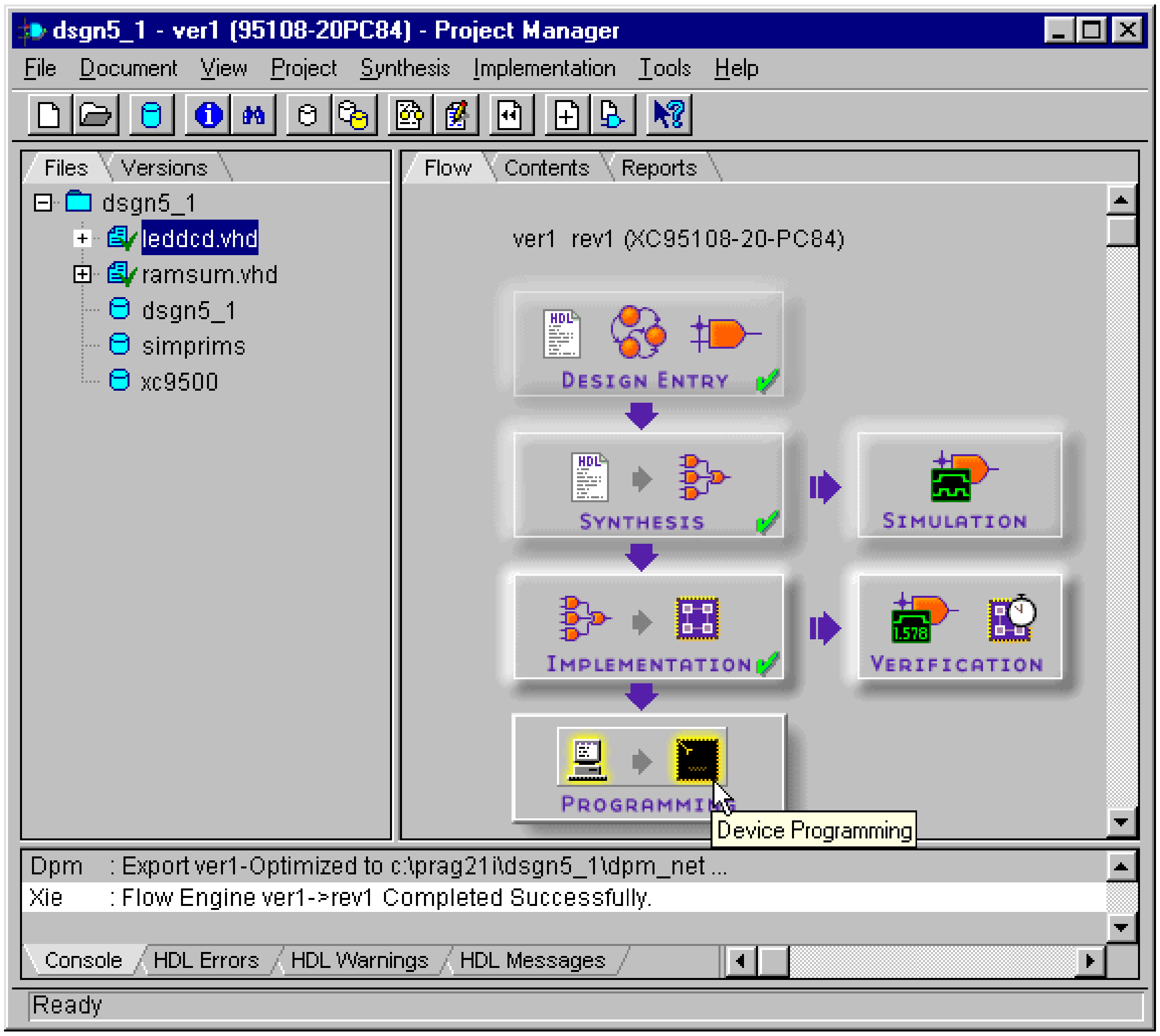
Both phases should complete with no problems.


Project Manager

Flow Engine ver1->rev1 Completed Successfully.

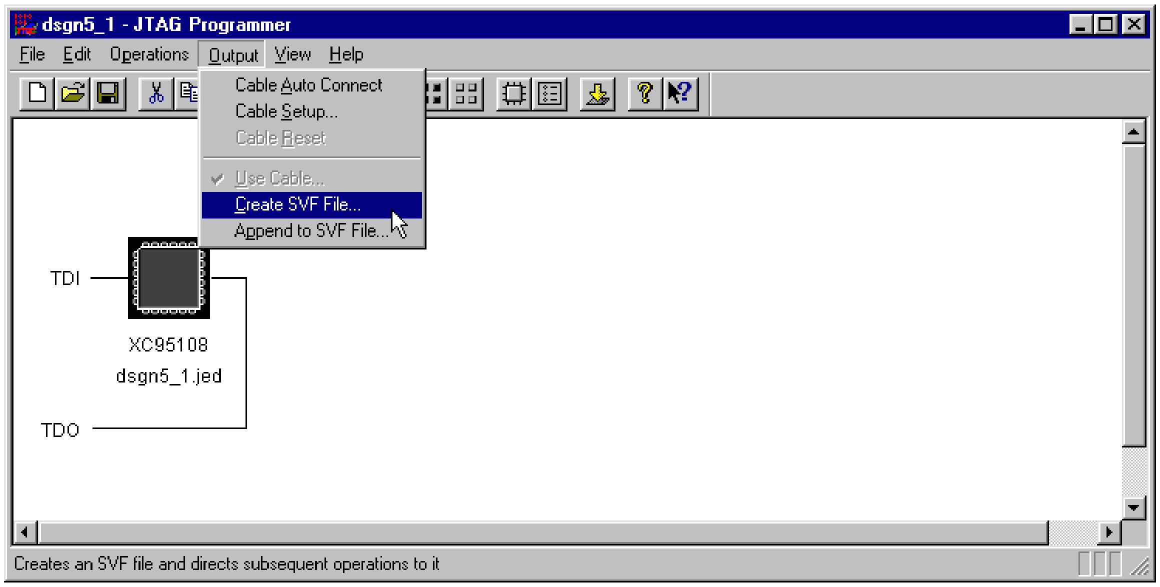
Generating the Bitstream

Once the implementation phase is completed, we can go on to create the SVF file containing the configuration bitstream for the XC95108 CPLD.


dsgn5_1 - ver1 (95108-20PC84) - Project Manager
File Document View Project Synthesis Implementation Tools Help
Files Versions
dsgn5_1
leddcd.vhd
ramsum.vhd
dsgn5_1
simprims
xc9500
Flow Contents Reports
ver1 rev1 (XC95108-20-PC84)
DESIGN ENTRY
SYNTHESIS
SIMULATION
IMPLEMENTATION
VERIFICATION
PROGRAMMING
Device Programming
Console HDL Errors HDL Warnings HDL Messages
Ready

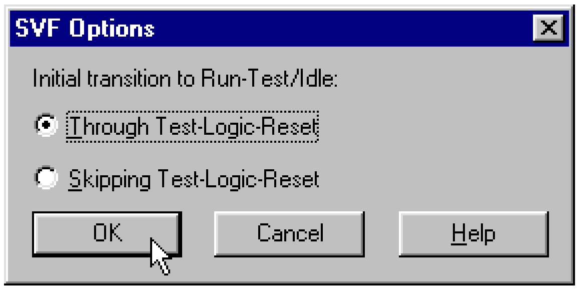
```
Dpm   : Export ver1-Optimized to c:\prag21i\dsgn5_1\dpm_net ...
Xie   : Flow Engine ver1->rev1 Completed Successfully.
```

Select the Output➔Create SVF File... menu item from the **JTAG Programmer** window that appears.



Creates an SVF file and directs subsequent operations to it

As always, specify an initial transition of the JTAG state machine through the Test-Logic-Reset state.

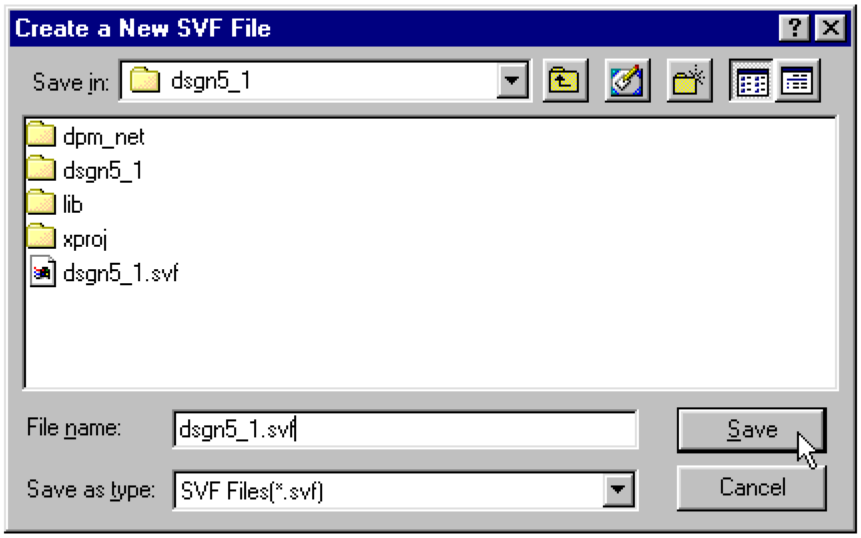


Initial transition to Run-Test/Idle:

[✓ Through Test-Logic-Reset]

Skipping Test-Logic-Reset

Then tell the JTAG Programmer to save the configuration bitstream in the dsgn5_1.svf file in the top-level directory of the **dsgn5_1** project.

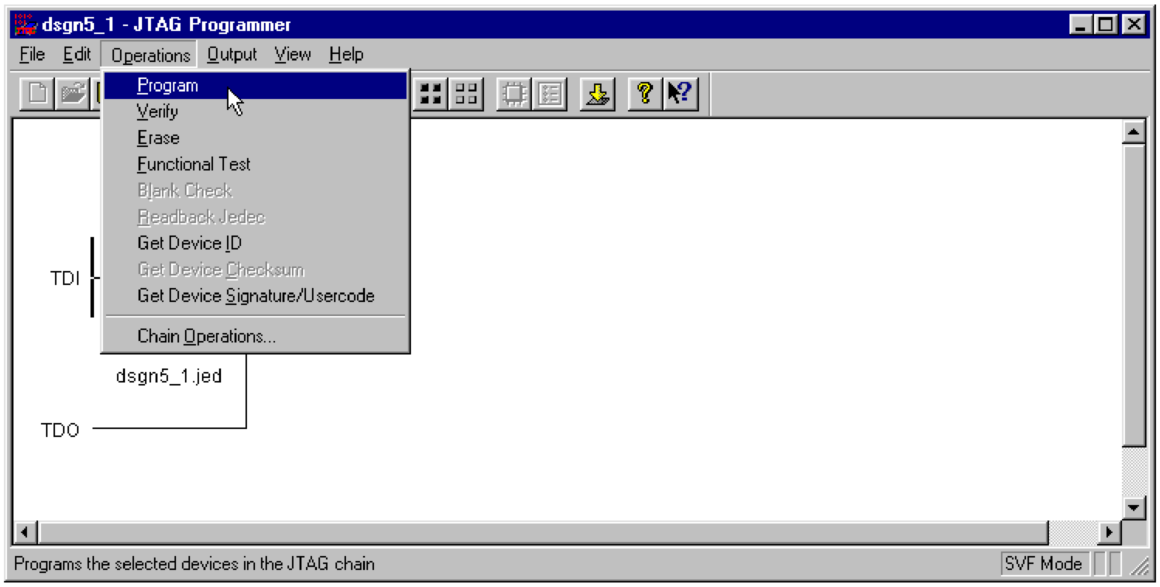


Save in: dsgn5_1

File name: dsgn5_1.svf

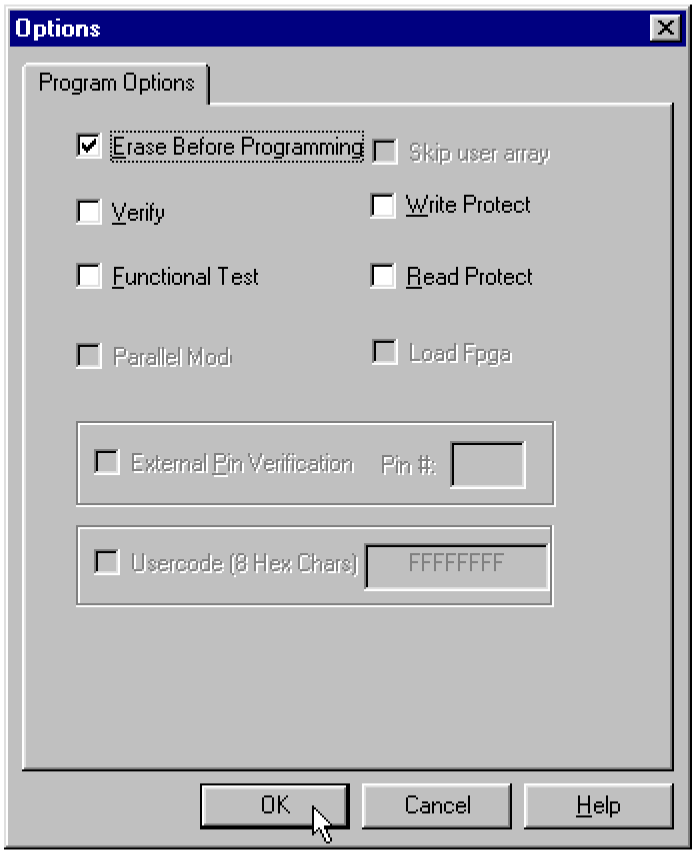
Save as type: SVF Files(*.svf)

Now initiate the generation of the bitstream by selecting the Operations➔Program menu item.



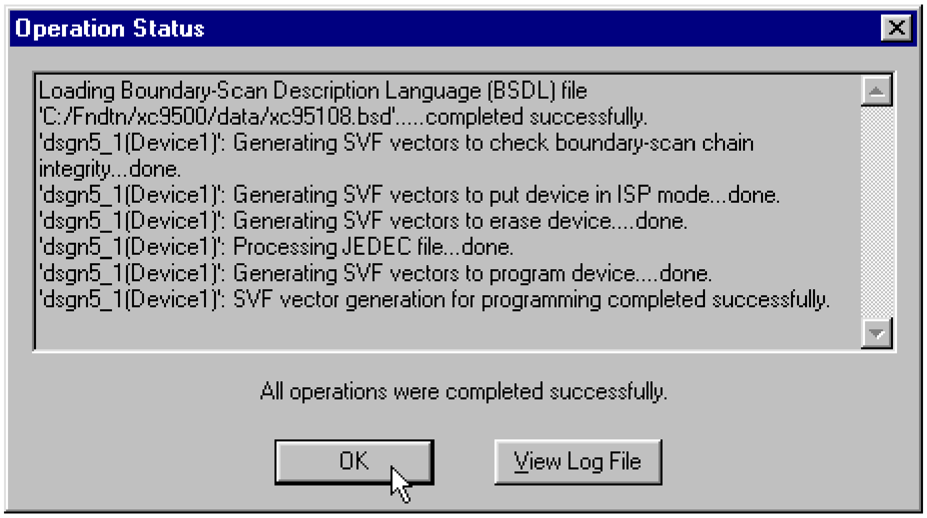
Programs the selected devices in the JTAG chain

Just click on the OK button in the **Options** window to begin generating the bitstream.



[✓] Erase Before Programming
[ ] Skip user array
[ ] Verify
[ ] Write Protect
[ ] Functional Test
[ ] Read Protect
[ ] Parallel Mod
[ ] Load Fpga
[ ] External Pin Verification Pin #:
[ ] Usercode (8 Hex Chars) FFFFFFFF

The bitstream generation should complete without incident.


Operation Status

```
Loading Boundary-Scan Description Language (BSDL) file
'C:/Fndtn/xc9500/data/xc95108.bsd'.....completed successfully.
'dsgn5_1(Device1)': Generating SVF vectors to check boundary-scan chain
integrity...done.
'dsgn5_1(Device1)': Generating SVF vectors to put device in ISP mode...done.
'dsgn5_1(Device1)': Generating SVF vectors to erase device....done.
'dsgn5_1(Device1)': Processing JEDEC file...done.
'dsgn5_1(Device1)': Generating SVF vectors to program device....done.
'dsgn5_1(Device1)': SVF vector generation for programming completed successfully.
```

All operations were completed successfully.

Downloading and Testing the Design

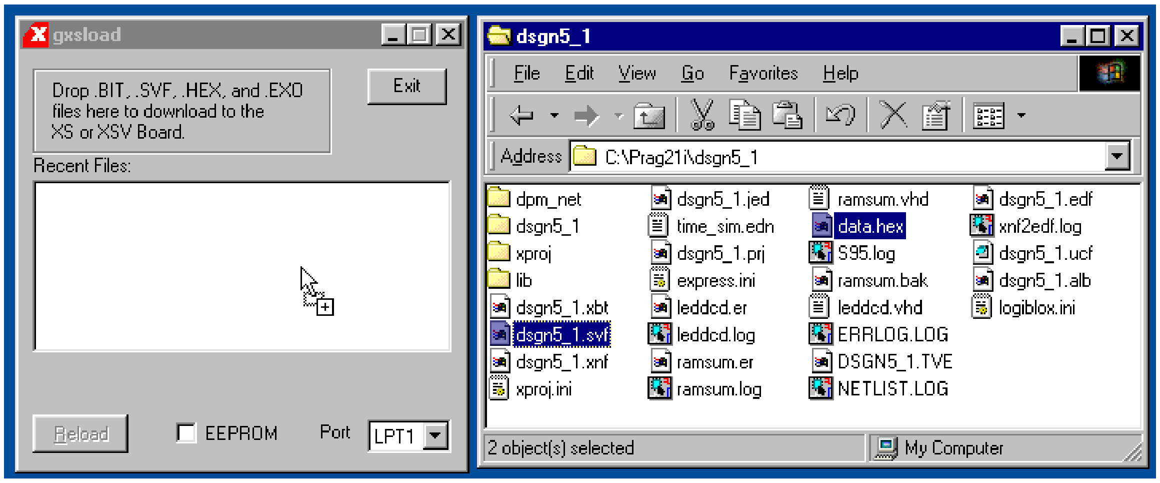
We need some test data to store into the RAM of the XS95 Board in order to test the design. Go to the top-level directory of the ***dsgn5_1*** project and use a text editor to create a file called data.hex containing this single line of text:

```
- 0B 0000 FF FE FD FC FB FA F9 F8 F7 F6 F5
```

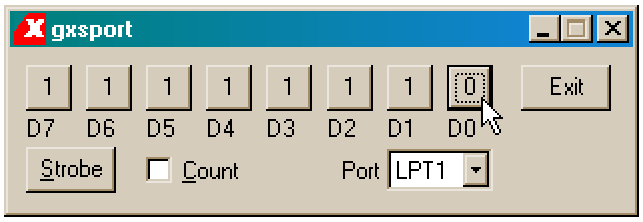
This is a set of eleven data bytes that will be loaded into RAM starting at address zero. (This data is represented in the XESS format.) If you manually complement-and-sum these data values you will get the following result (in two-digit hexadecimal):

```
(-FF)+(-FE)+(-FD)+(-FC)+(-FB)+(-FA)+(-F9)+(-F8)+(-F7)+(-F6)+(-F5) =
(1)+(2)+(3)+(4)+(5)+(6)+(7)+(8)+(9)+(A)+(B) =
42
```

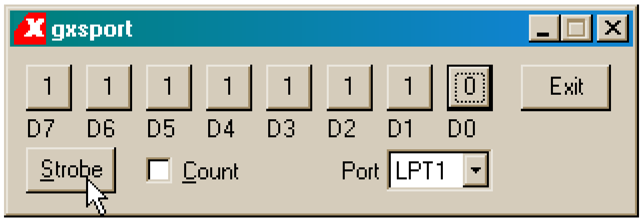
Now that the SVF and test data file are ready, connect an XS95 Board to the PC parallel port and start the GXSLOAD program. Go to the top-level directory for the ***dsgn5_1*** project and select the dsgn5_1.svf and data.hex files. Then drag-and-drop them into the **gxsload** window. The data file will be downloaded into the RAM and then the SVF file will be programmed into the XC95108 CPLD on the XS95 Board.



The reset for the circuit is controlled by data pin D0 of the parallel port. If D0 is at logic 1 after the downloading completes, the circuit will be held in the reset state and the LED will be blank. To release the reset, open the **gxsport** window and click on the D0 button until it displays a zero.

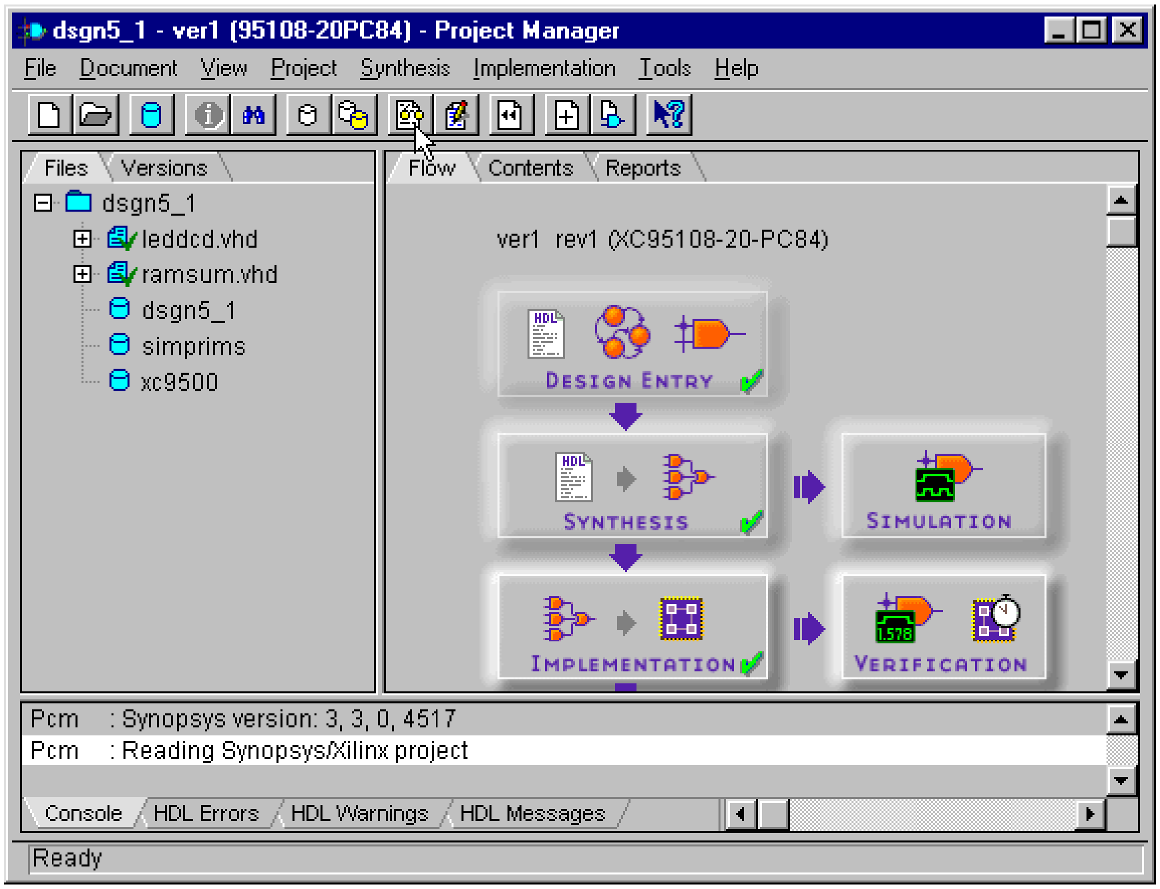


Then click on the Strobe button so the logic 0 value is output on the D0 pin of the parallel port.



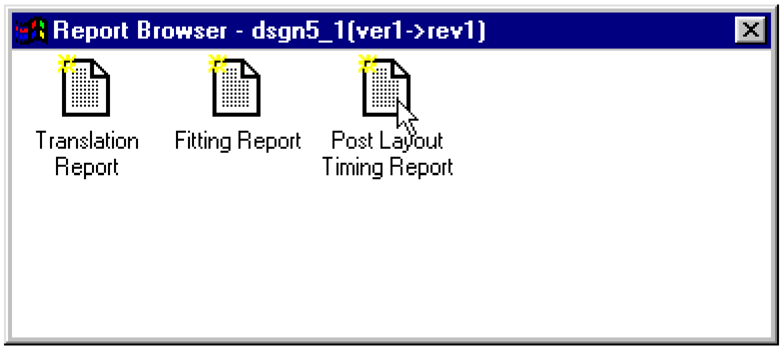
Now you *may* observe the seven-segment LED repeatedly displaying the sequence ……4…2……4…2……. But you probably won't. What went wrong?

The answer is that you are probably running the design with a 50 MHz clock (the default for the XS95 Board). Can this design run that fast? Let's check the timing for the implemented design. Click on the icon for the report files in the **Project Navigator** window.


dsgn5_1 - ver1 (95108-20PC84) - Project Manager
File Document View Project Synthesis Implementation Tools Help
Files Versions
dsgn5_1
leddcd.vhd
ramsum.vhd
dsgn5_1
simprims
xc9500
Flow Contents Reports
ver1 rev1 (XC95108-20-PC84)
Design Entry
Synthesis
Simulation
Implementation
Verification
Console HDL Errors HDL Warnings HDL Messages
Ready

```
Pcm    : Synopsys version: 3, 3, 0, 4517
Pcm    : Reading Synopsys/Xilinx project
```

Then double-click the Post Layout Timing Report in the **Report Browser** window.

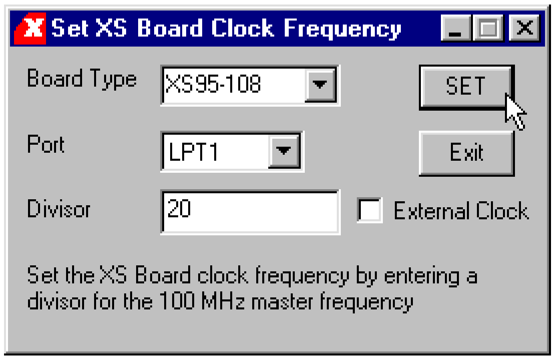

Report Browser - dsgn5_1(ver1->rev1)
Translation Report
Fitting Report
Post Layout Timing Report

The top portion of the timing report is shown in Listing 8 and this tells us what we want to know: the maximum clock frequency for this design is 5.5 MHz. The slow clock is brought about by the long carry propagation times through the complementors and adders in the design. The situation is made worse because the implementation algorithms have packed the logic to emphasize area efficiency and this can add extra propagation delays to the circuit.

Listing 8: Timing report for the design.

```
                         Performance Summary Report
                         --------------------------

Design:     dsgn5_1
Device:     XC95108-20-PC84
Program:    Timing Report Generator:  version C.22
Date:       Sat Jan 05 14:08:28 2002

Performance Summary:

Clock net 'clk' path delays:

Clock Pad to Output Pad (tCO): 73.0ns (4 macrocell levels)
Clock Pad 'clk' to Output Pad 's<4>' (GCK)

Clock to Setup (tCYC):        161.5ns (8 macrocell levels)
Clock to Q, net 'sum_r<0>.Q' to TFF Setup(D) at 'sum_r<7>.D'  (GCK)
Target FF drives output net 'sum_r<7>'

Setup to Clock at the Pad (tSU): 151.5ns (7 macrocell levels)
Data signal 'd<0>' to TFF D input Pin at 'sum_r<7>.D'
Clock pad 'clk' (GCK)

                         Minimum Clock Period: 161.5ns
                      Maximum Internal Clock Speed: 6.1Mhz
                            (Limited by Cycle Time)
```

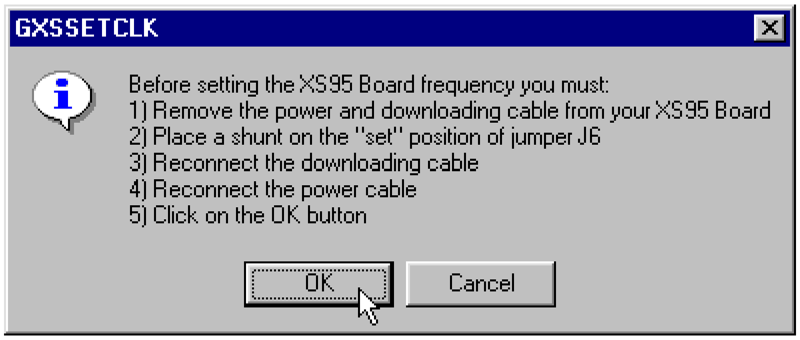
We need to reduce the clock frequency of the XS95 Board to less that 6.1 MHz in order for our design to work reliably. To do this, start the GXSSETCLK program. Place 20 in the Divisor field to reduce the 100 MHz master frequency to 5 MHz. Then click on the SET button.



Board Type: XS95-108
Port: LPT1
Divisor: 20
External Clock: ☐

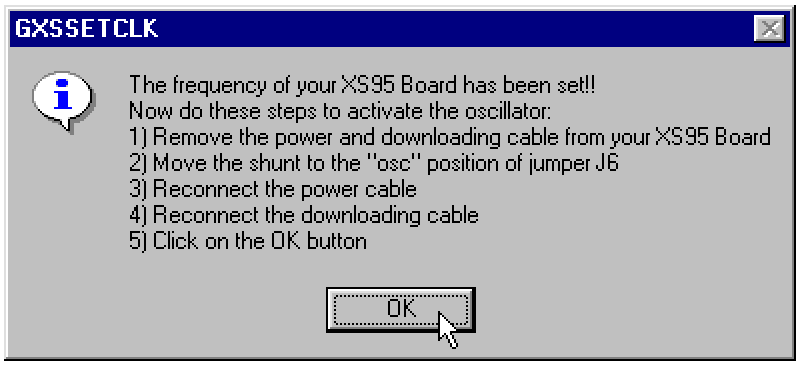
Set the XS Board clock frequency by entering a divisor for the 100 MHz master frequency

A set of instructions will appear that must be followed to adjust the clock frequency of the XS95 Board. After doing these steps, click on the OK button to reprogram the clock.


GXSSETCLK

Before setting the XS95 Board frequency you must:
1) Remove the power and downloading cable from your XS95 Board
2) Place a shunt on the "set" position of jumper J6
3) Reconnect the downloading cable
4) Reconnect the power cable
5) Click on the OK button

Reprogramming the clock takes a minute or two after which the following set of instructions is given to activate the new clock frequency.


GXSSETCLK

The frequency of your XS95 Board has been set!!
Now do these steps to activate the oscillator:
1) Remove the power and downloading cable from your XS95 Board
2) Move the shunt to the "osc" position of jumper J6
3) Reconnect the power cable
4) Reconnect the downloading cable
5) Click on the OK button

After activating the 5 MHz clock frequency, we can download the dsgn5_1.SVF and the data.hex files and release the reset on the circuit. Now we should see the ……4…2……4…2…… sequence displayed on the LED digit. If we set and clear the reset again, then we should see the display change to ……B…E……B…E……. Why? Because the first time the circuit was run it computed the two's-complement of all the data values and wrote them back into the RAM after which it computed the sum of the data. So the second time the circuit was run it was using complemented data and the resulting sum is the two's-complement of the first result: -42 = BE in two-digit hexadecimal. In addition, the sum will toggle between 42 and BE each time the circuit is reset.

Using an Internal Synchronous RAM

The second version of the RAM summation circuit has the design hierarchy shown in Figure 15. The root module of the design sums the data stored in an internal synchronous RAM module while the LED decoder module displays the four-bit hexadecimal digits on a seven-segment display. Only the XC4000 FPGAs have internal RAM so this design can only be done using the XS40 Board. The XC95108 CPLD on the XS95 Board is not suitable for designs, which require large amounts of internal data storage.

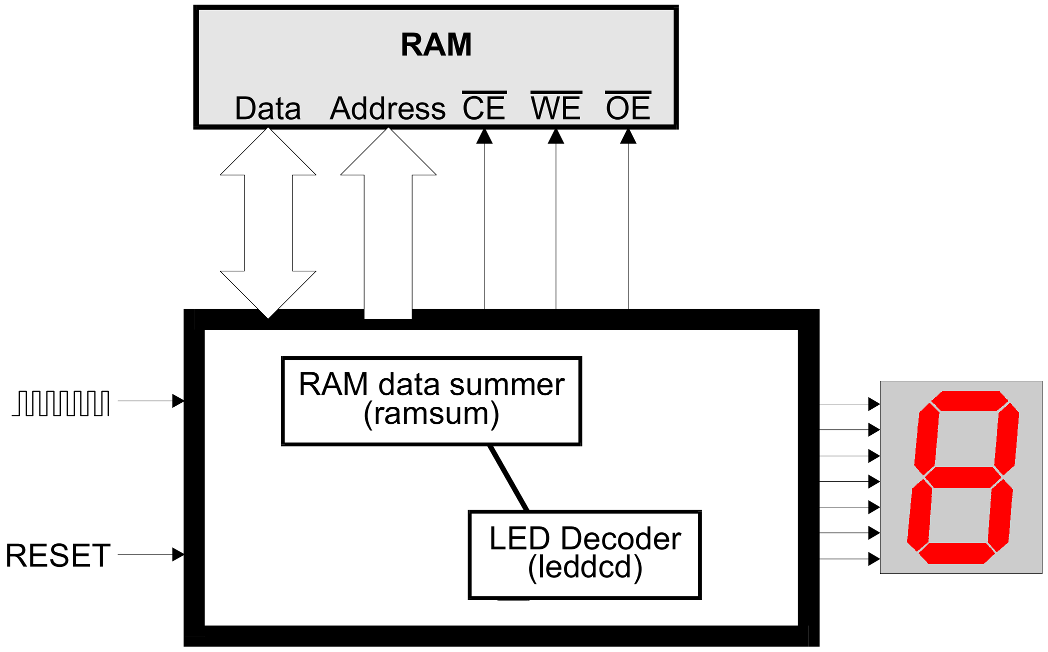
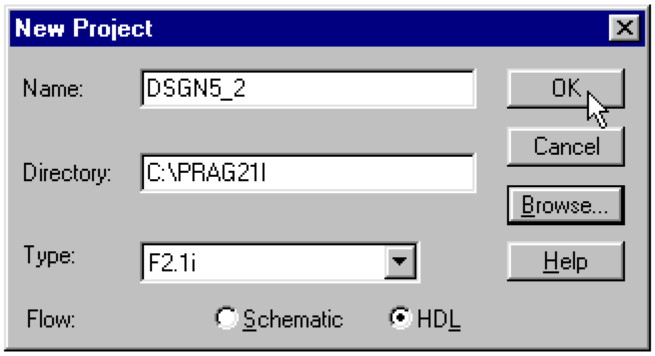


Figure 15: Design hierarchy for a logic circuit that displays the summation of data in an internal synchronous RAM.

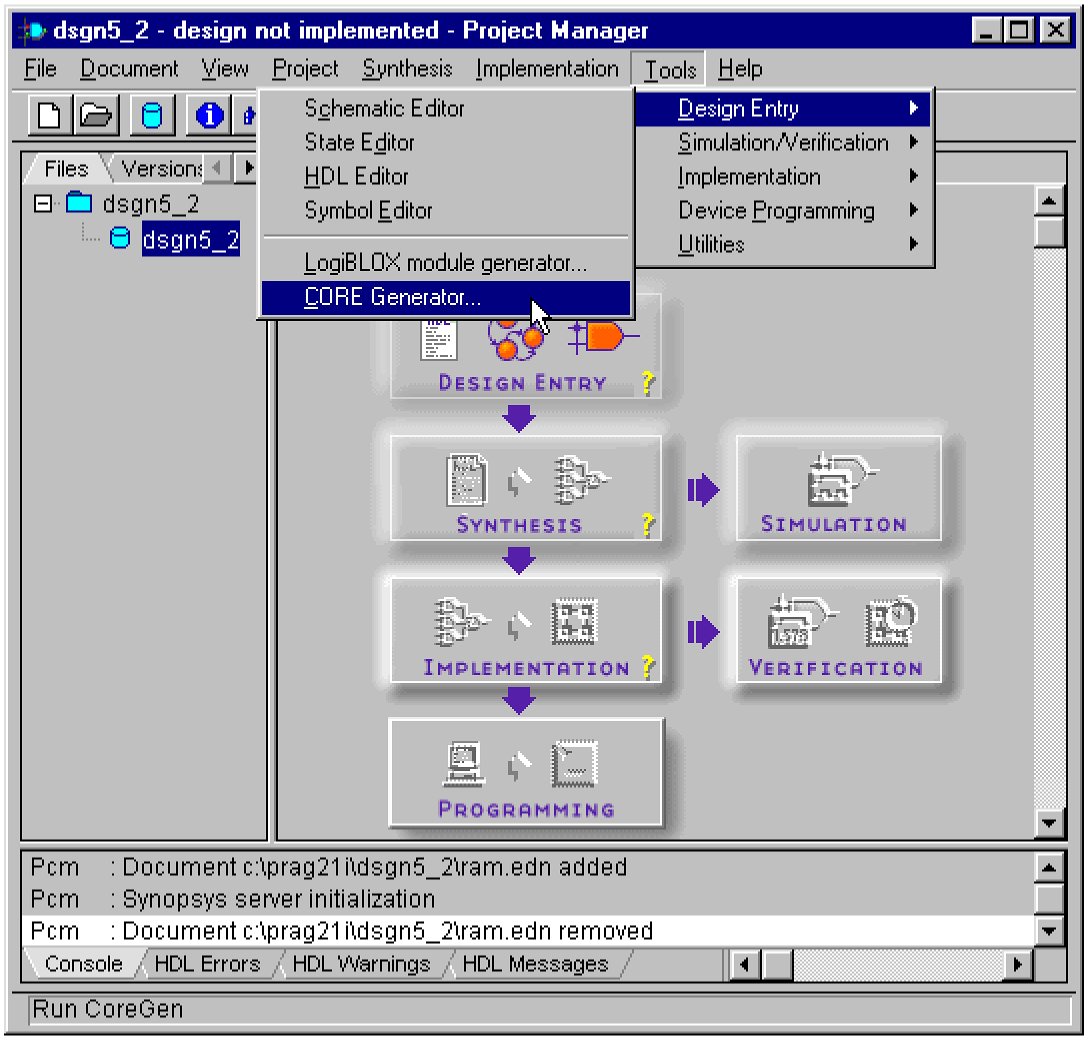
Each of these modules is stored in the ***dsgn5_2*** directory that was created by starting an HDL project follows.



Name: DSGN5_2
Directory: C:\PRAG21I
Type: F2.1i
Flow: Schematic [✓ HDL]

The Internal RAM Module

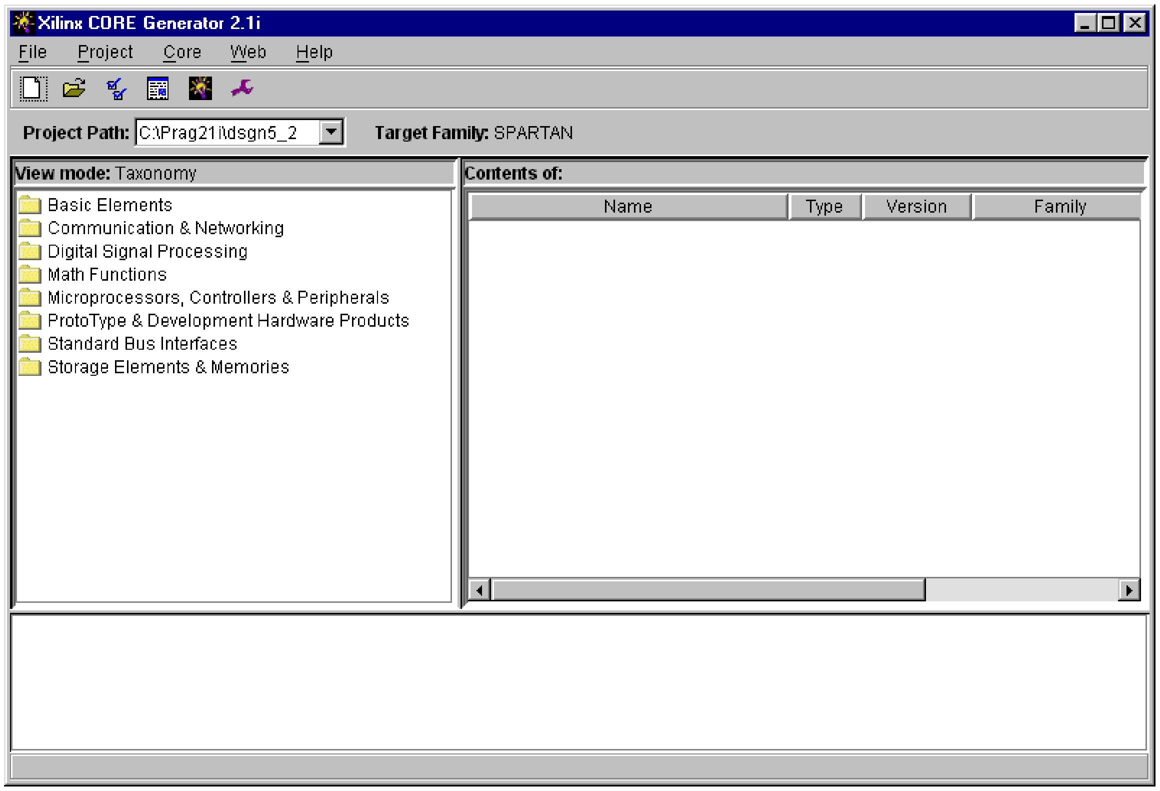
The first module we will add is the internal synchronous RAM. This module is constructed using the CORE Generator. To start this tool, select the Tools➔Design Entry➔Core Generator... menu item.



```
Pcm    : Document c:\prag21i\dsgn5_2\ram.edn added
Pcm    : Synopsys server initialization
Pcm    : Document c:\prag21i\dsgn5_2\ram.edn removed
```

Run CoreGen

The **Xilinx CORE Generator** window will appear. The left-hand pane of the window displays the various families of circuits that the tool can generate. The individual circuits within a highlighted family are shown in the right-hand pane.



Project Path: C:\Prag21i\dsgn5_2 **Target Family:** SPARTAN

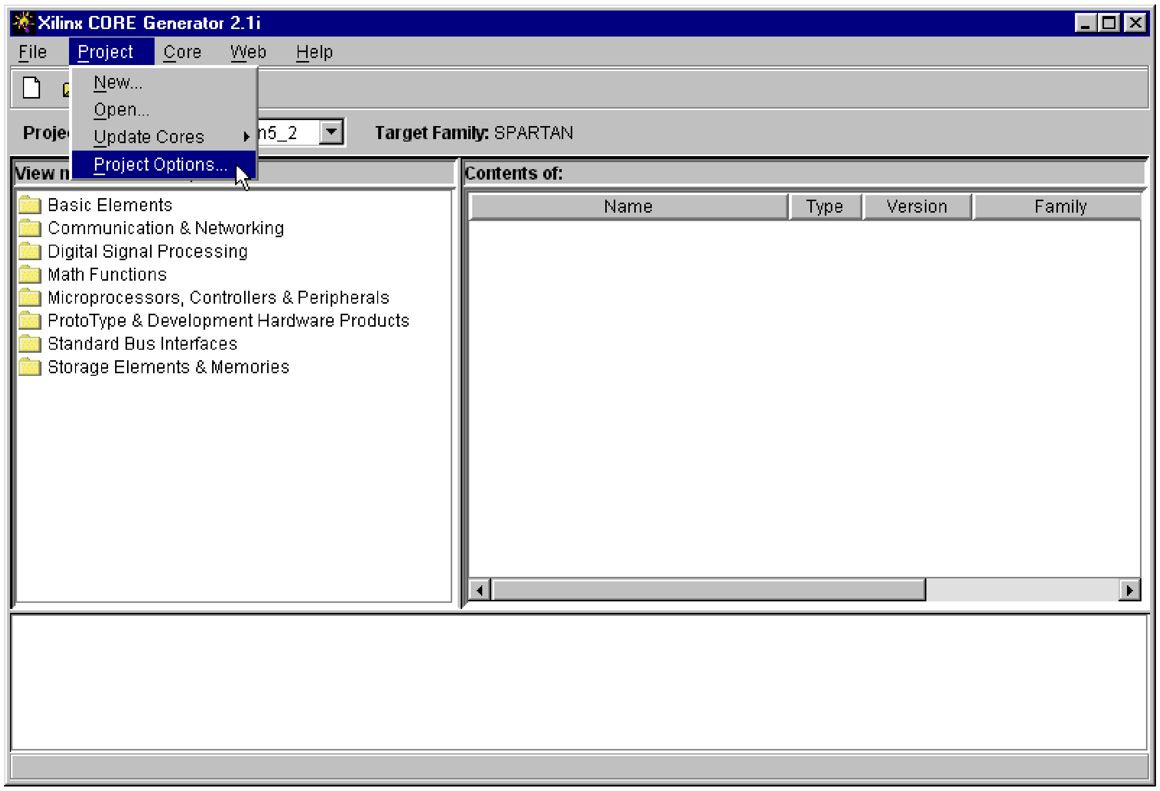
View mode: Taxonomy

- Basic Elements
- Communication & Networking
- Digital Signal Processing
- Math Functions
- Microprocessors, Controllers & Peripherals
- ProtoType & Development Hardware Products
- Standard Bus Interfaces
- Storage Elements & Memories

Contents of:

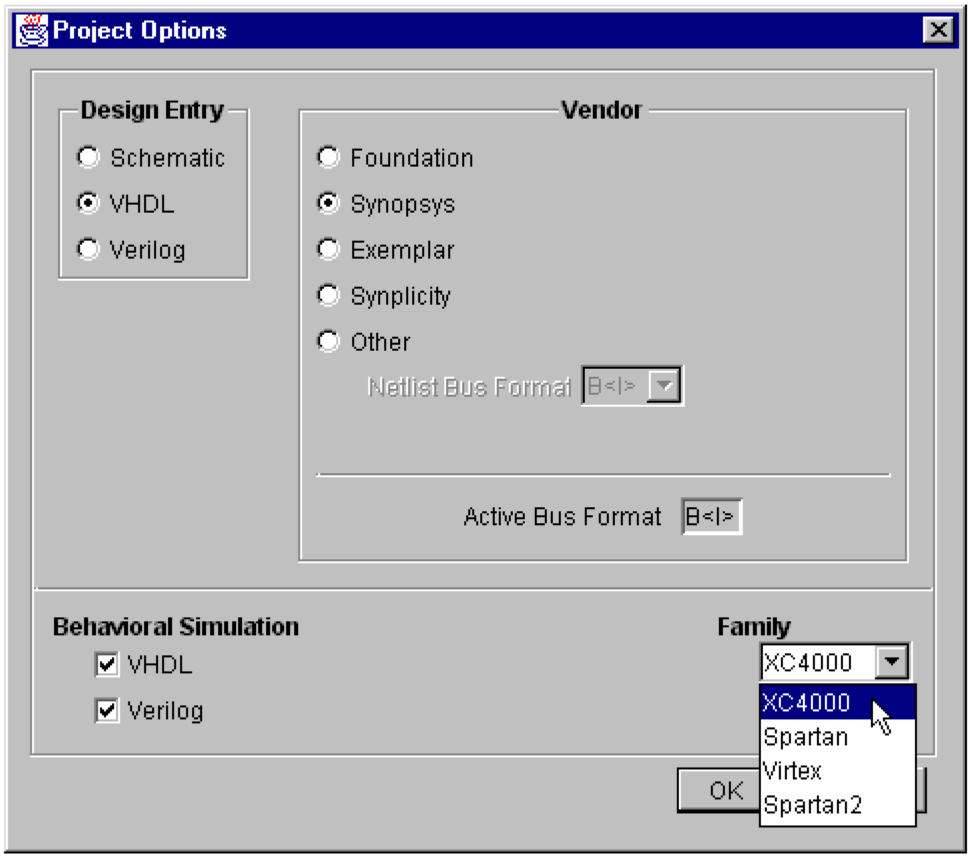
Name	Type	Version	Family

Our first action is to setup the CORE Generator for the target FPGA and type of project we are using in Foundation. Click on the Project➔Project Options… menu item to open the **Project Options** window.



Basic Elements
Communication & Networking
Digital Signal Processing
Math Functions
Microprocessors, Controllers & Peripherals
ProtoType & Development Hardware Products
Standard Bus Interfaces
Storage Elements & Memories

In the **Project Options** window, select VHDL in the Design Entry section since our project will be done using VHDL. Also, click on the Synopsys button in the Vendor section since this is the VHDL synthesis tool used by Foundation 2.1. Finally, select XC4000 as the target FPGA in the Family section.


Project Options

Design Entry

( ) Schematic
(●) VHDL
( ) Verilog

Vendor

( ) Foundation
(●) Synopsys
( ) Exemplar
( ) Synplicity
( ) Other

Netlist Bus Format: B<I>

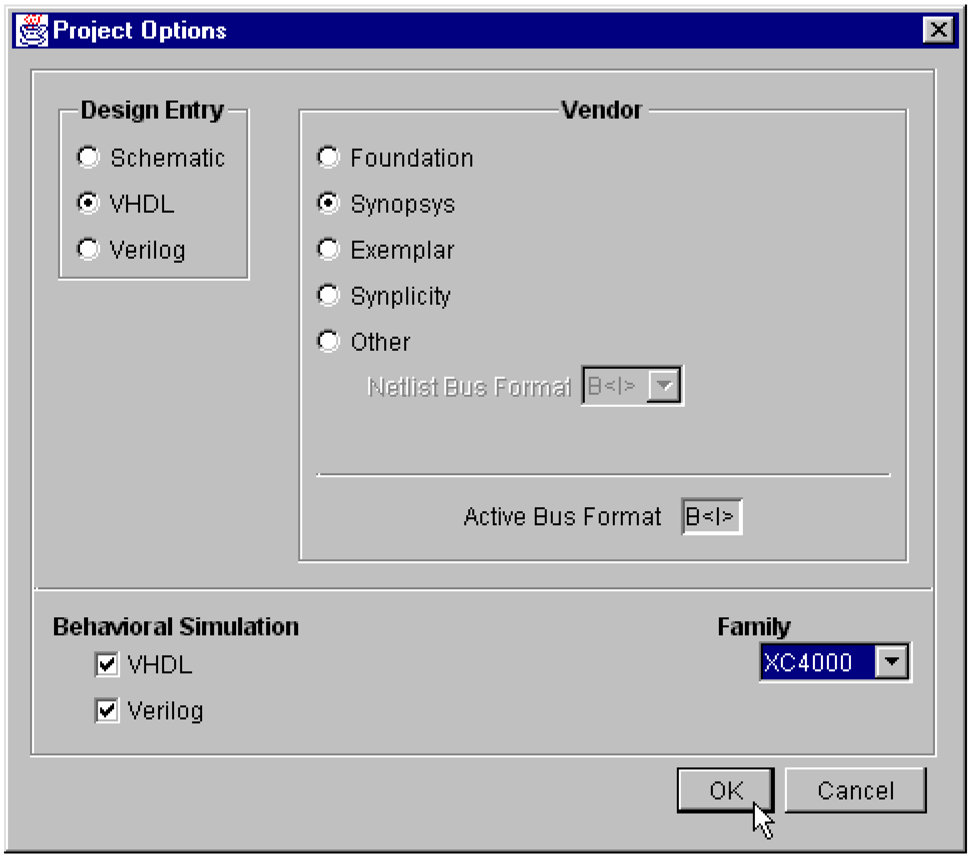
Active Bus Format: B<I>

Behavioral Simulation

✓ VHDL
✓ Verilog

Family: XC4000

Once the options are set as described above, click on the OK button to close the window.


Project Options

Design Entry
- ( ) Schematic
- (●) VHDL
- ( ) Verilog

Vendor
- ( ) Foundation
- (●) Synopsys
- ( ) Exemplar
- ( ) Synplicity
- ( ) Other

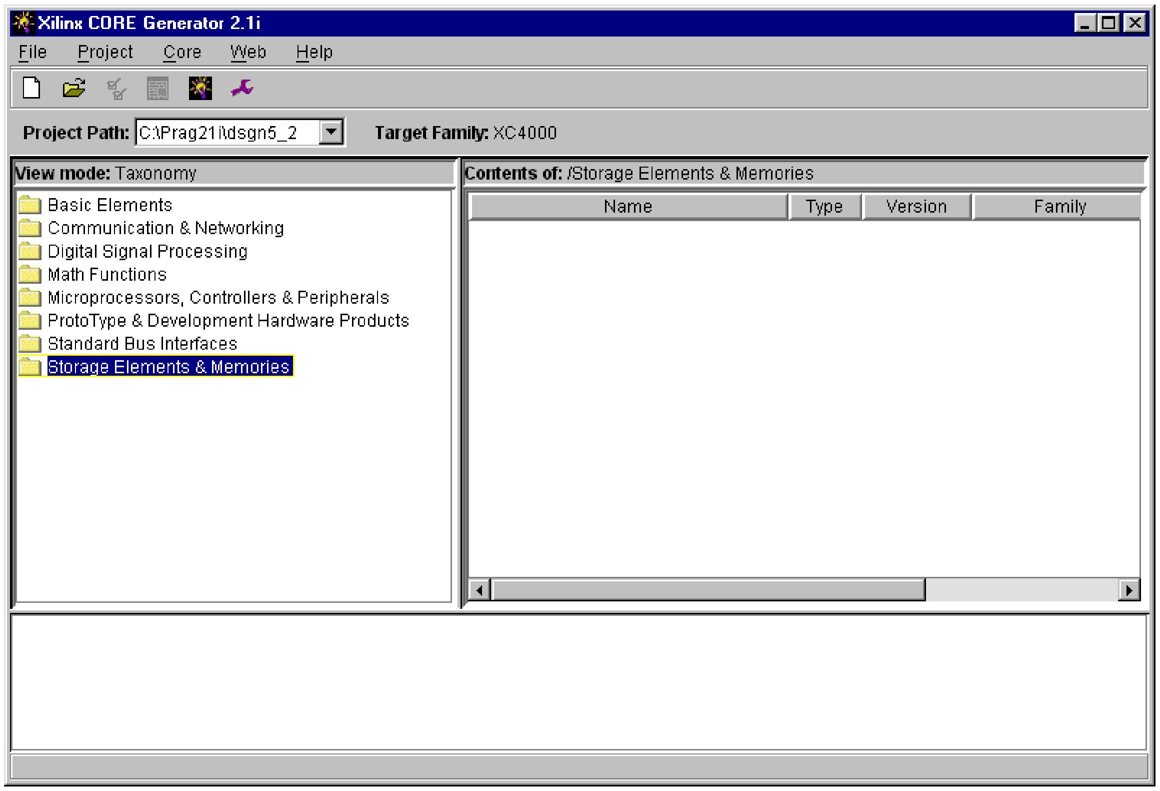
Netlist Bus Format B<I>

Active Bus Format B<I>

Behavioral Simulation
- [✓] VHDL
- [✓] Verilog

Family XC4000

In the **Core Generator** window we can now see the Target Family is listed as XC4000. Next, double-click on the Storage Elements & Memories entry in the left-hand pane.



Project Path: C:\Prag21i\dsgn5_2 Target Family: XC4000

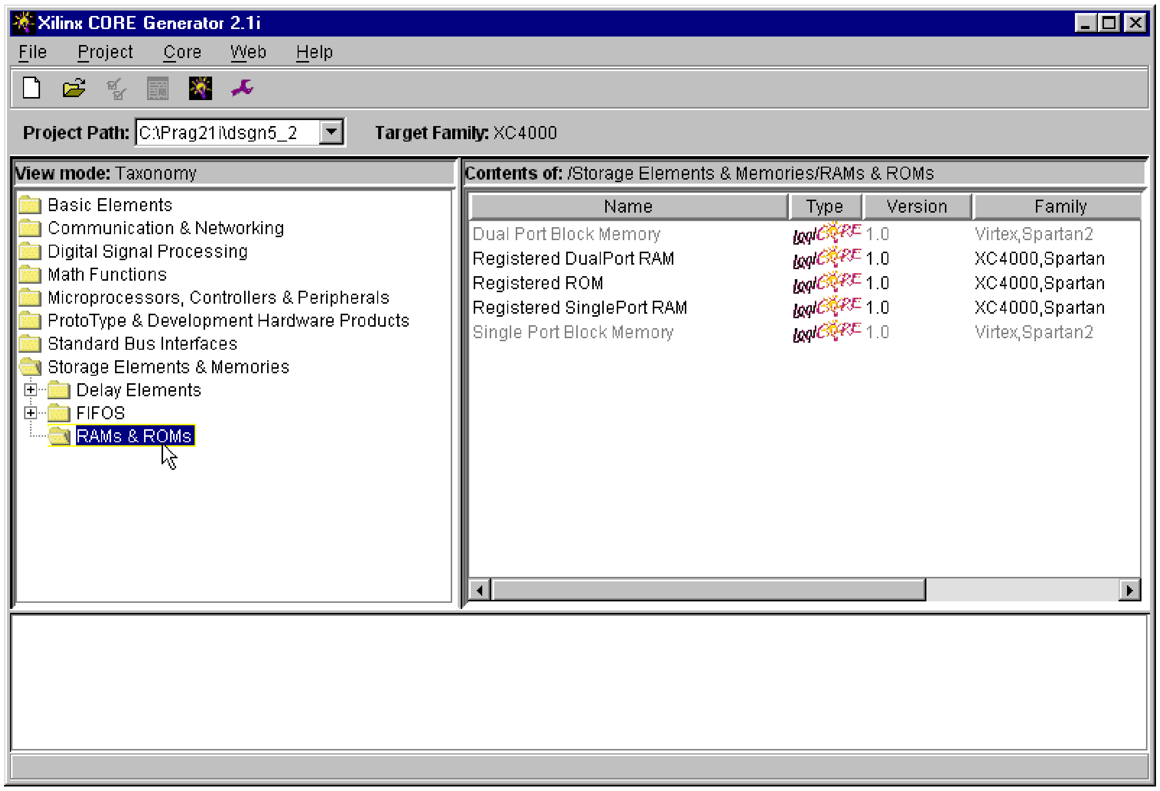
View mode: Taxonomy

- Basic Elements
- Communication & Networking
- Digital Signal Processing
- Math Functions
- Microprocessors, Controllers & Peripherals
- ProtoType & Development Hardware Products
- Standard Bus Interfaces
- Storage Elements & Memories

Contents of: /Storage Elements & Memories

Name	Type	Version	Family

Double-clicking the Storage Elements & Memories entry expands it and exposes three sub-families of modules. Clicking on the RAMs & ROMs entry will display the members of this family in the right-hand pane.



Project Path: C:\Prag21i\dsgn5_2 Target Family: XC4000

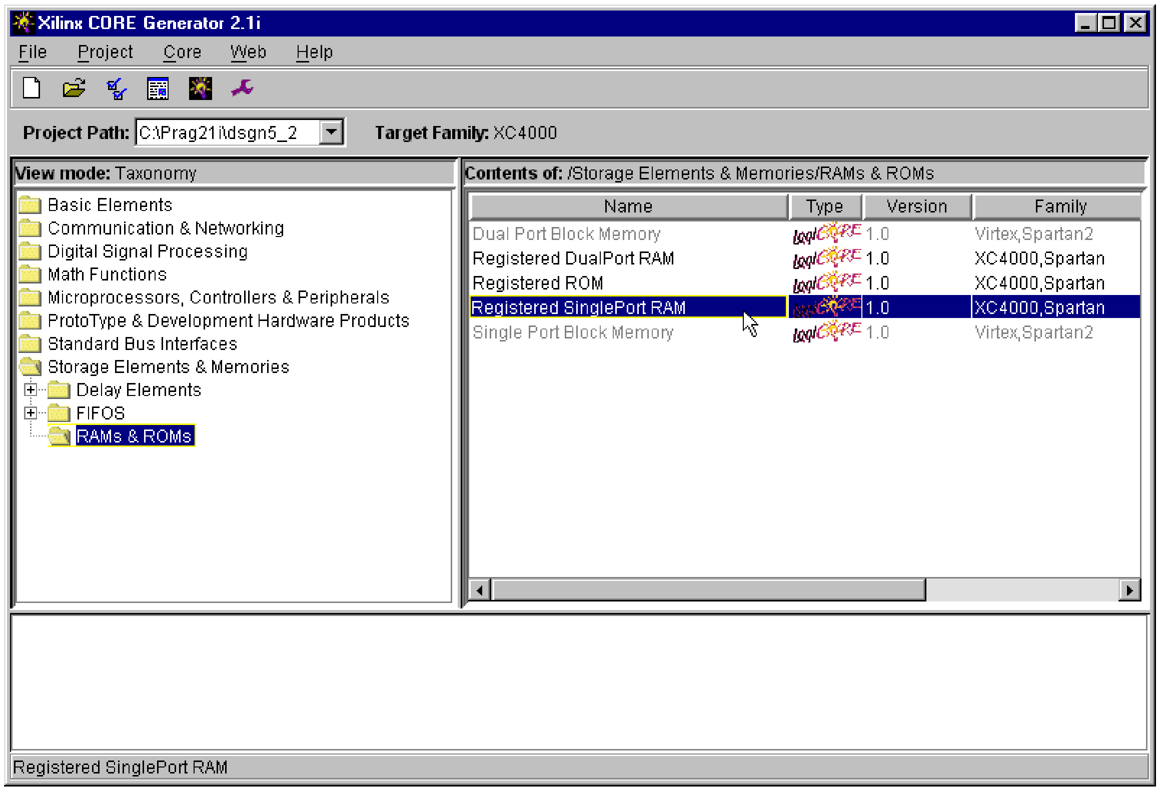
View mode: Taxonomy

- Basic Elements
- Communication & Networking
- Digital Signal Processing
- Math Functions
- Microprocessors, Controllers & Peripherals
- ProtoType & Development Hardware Products
- Standard Bus Interfaces
- Storage Elements & Memories
  - Delay Elements
  - FIFOS
  - RAMs & ROMs

Contents of: /Storage Elements & Memories/RAMs & ROMs

Name	Type	Version	Family
Dual Port Block Memory	LogiCORE	1.0	Virtex,Spartan2
Registered DualPort RAM	LogiCORE	1.0	XC4000,Spartan
Registered ROM	LogiCORE	1.0	XC4000,Spartan
Registered SinglePort RAM	LogiCORE	1.0	XC4000,Spartan
Single Port Block Memory	LogiCORE	1.0	Virtex,Spartan2

We need to both read and write the data values so a RAM should be used rather than a ROM for this application. We will also try to keep this circuit as similar to the one in the previous project so we will use a RAM with a single data port for both read and write operations. For these reasons, the Registered SinglePort RAM is the closest match to what we need so double-click that entry to begin the generation of such a RAM module.



Project Path: C:\Prag21i\dsgn5_2 Target Family: XC4000

View mode: Taxonomy

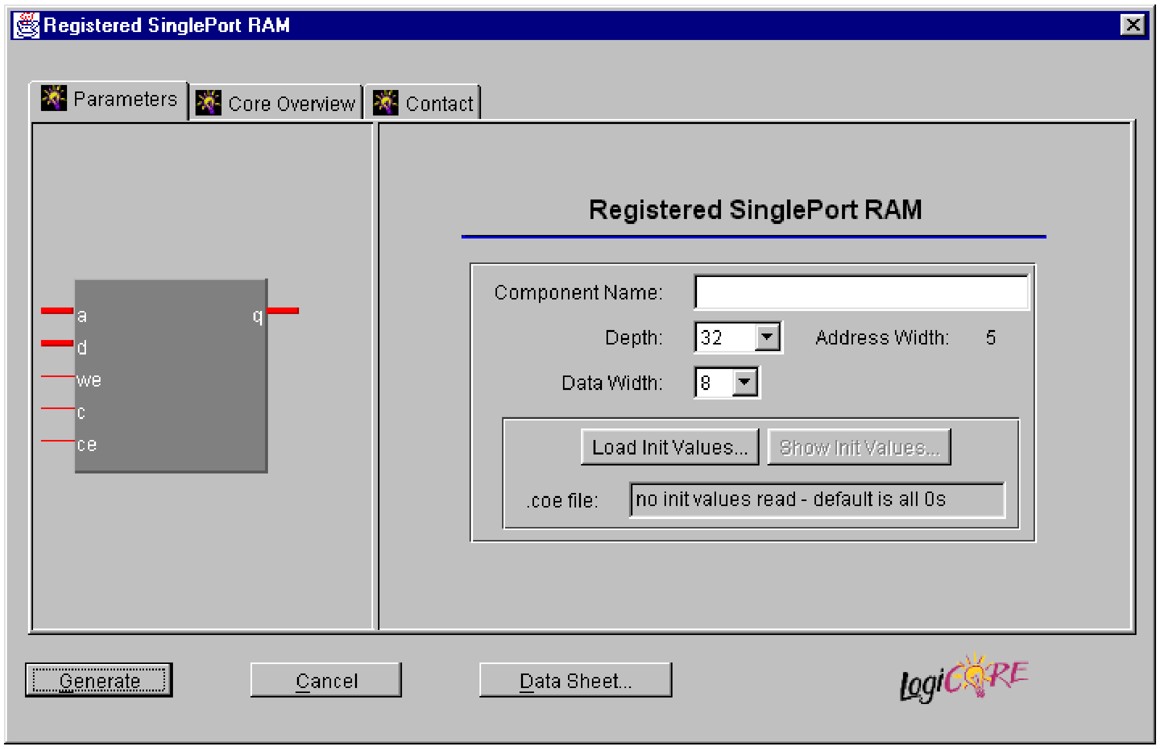
- Basic Elements
- Communication & Networking
- Digital Signal Processing
- Math Functions
- Microprocessors, Controllers & Peripherals
- ProtoType & Development Hardware Products
- Standard Bus Interfaces
- Storage Elements & Memories
  - Delay Elements
  - FIFOS
  - RAMs & ROMs

Contents of: /Storage Elements & Memories/RAMs & ROMs

Name	Type	Version	Family
Dual Port Block Memory	LogiCORE	1.0	Virtex,Spartan2
Registered DualPort RAM	LogiCORE	1.0	XC4000,Spartan
Registered ROM	LogiCORE	1.0	XC4000,Spartan
Registered SinglePort RAM	LogiCORE	1.0	XC4000,Spartan
Single Port Block Memory	LogiCORE	1.0	Virtex,Spartan2

Registered SinglePort RAM

The **Registered SinglePort RAM** window that appears has three tabs. The Core Overview tab displays a general summary of the module while the Contact tab lists the organization that was responsible for designing the module. But the Parameters tab is where we actually personalize the module to fit our particular application.

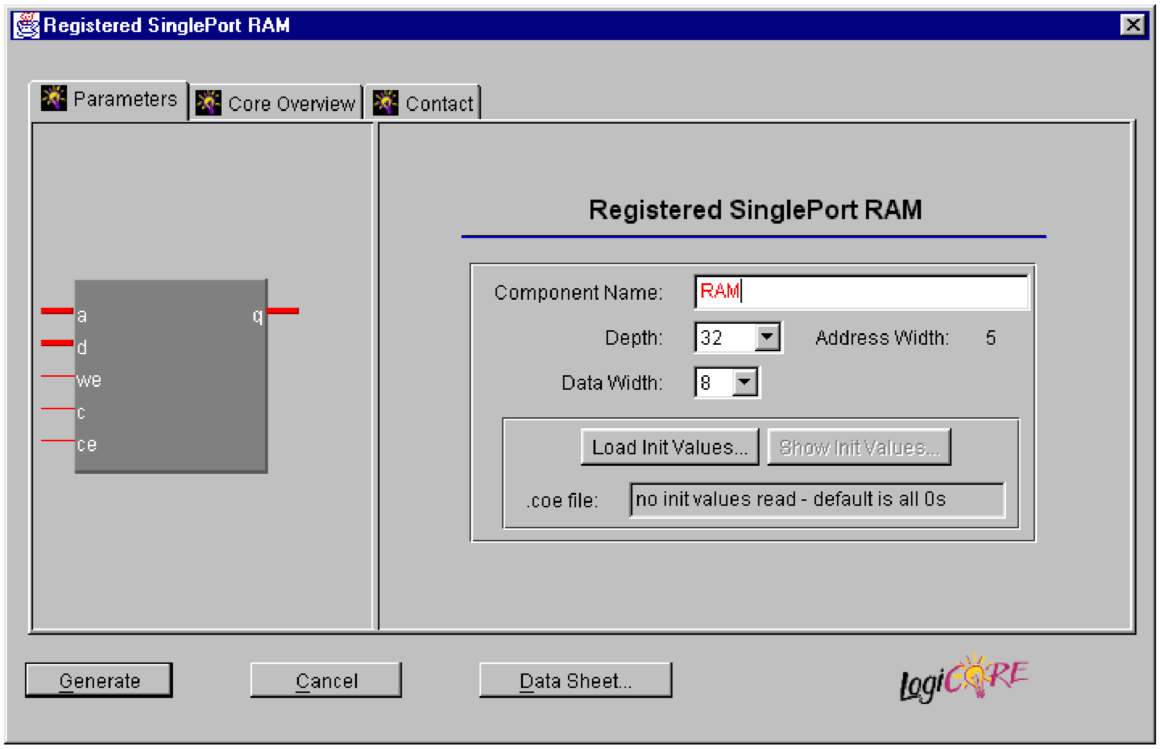

Registered SinglePort RAM

Parameters | Core Overview | Contact

Registered SinglePort RAM

Component Name:
Depth: 32 Address Width: 5
Data Width: 8
Load Init Values... Show Init Values...
.coe file: no init values read - default is all 0s

The first thing to do is to type a name for the module into the Component Name field. We chose the very original name RAM in this case.

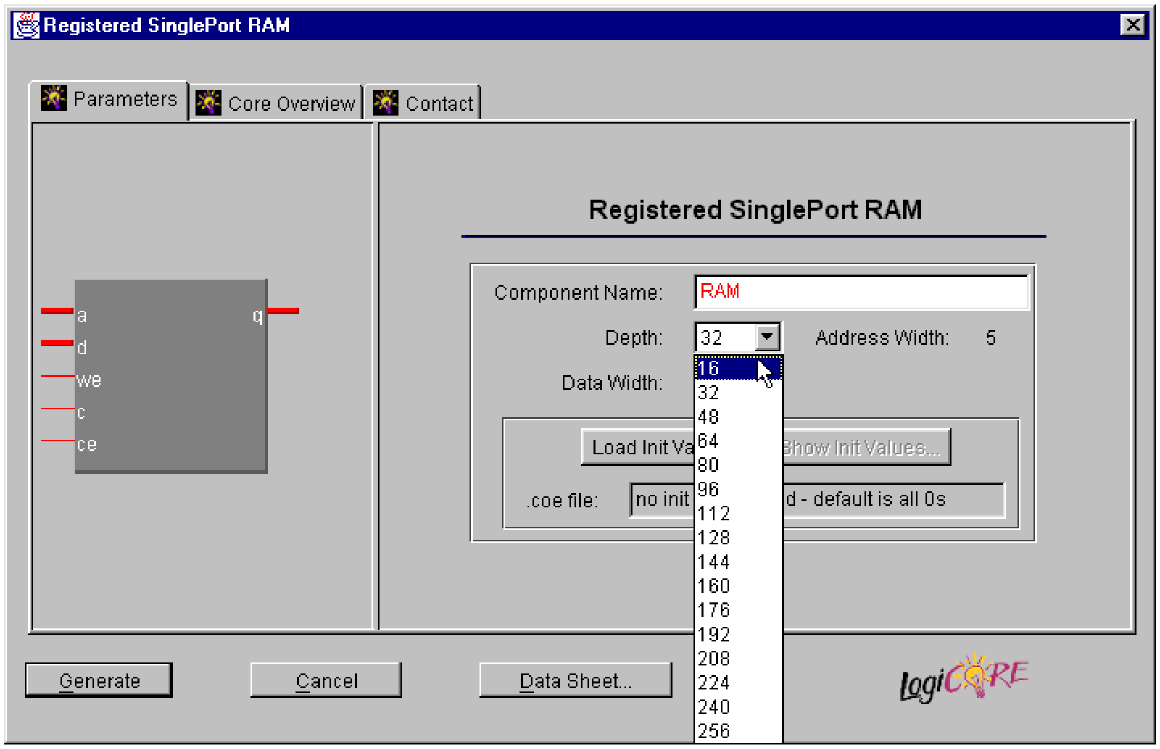

Registered SinglePort RAM

Parameters | Core Overview | Contact

Registered SinglePort RAM

Component Name: RAM
Depth: 32 Address Width: 5
Data Width: 8
Load Init Values... Show Init Values...
.coe file: no init values read - default is all 0s

Next, we set the number of locations in the RAM module. We want to sum as many as sixteen values, so select 16 from the Depth pull-down menu.



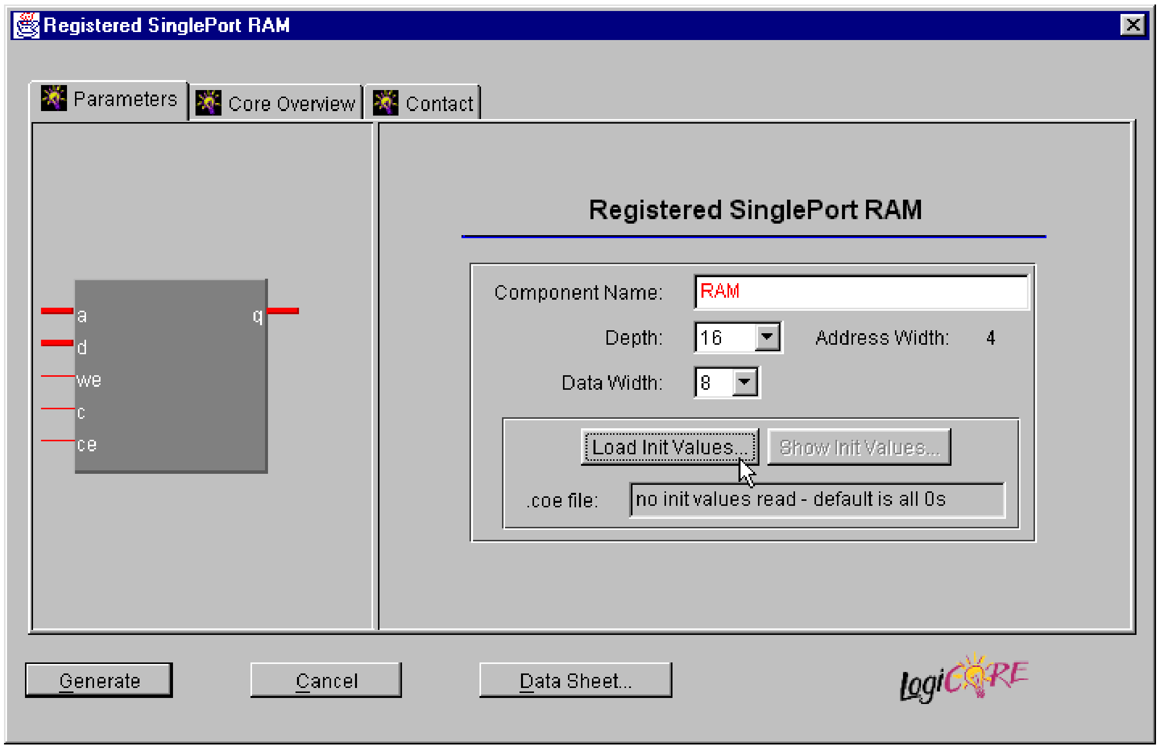
Once the Depth field is set to sixteen, note that the Address Width field changes to four. The Data Width Field is already set to eight so there is no need to change it.

The RAM has to be initialized with the values that will be summed. In the previous example, this initialization was managed by having the GXSLOAD utility load the external RAM with the contents of a HEX file. But in this example, the RAM is contained within the FPGA so there is no way for GXSLOAD to access it and load its contents. Instead, the initial values for the RAM must be inserted into the FPGA configuration bitstream so the RAM contents are initialized at the same time the logic gates on the FPGA are configured. The Core Generator looks for RAM initialization values in .coe files. The contents of such a file for the RAM in this example is shown in Listing 9. The `Radix` field is set to sixteen to indicate the data is represented in hexadecimal form. The `memdata` field stores the initial values of each RAM location starting from address zero and incrementing upwards until all sixteen locations are filled.

Listing 9: Initialization file for a Core Generator RAM.

```
Component_Name=ram;
Data_Width = 8;
Address_Width = 4;
Depth = 16;
Radix = 16;
memdata=FF,FE,FD,FC,FB,FA,F9,F8,F7,F6,F5,F4,F3,F2,F1,F0;
```

The RAM initialization values are stored in a file called ram.coe in the top-level directory of the ***dsgn5_2*** project. To load these values into the Core Generator, click on the Load Init values… button as shown below.



Registered SinglePort RAM

Parameters | Core Overview | Contact

Registered SinglePort RAM

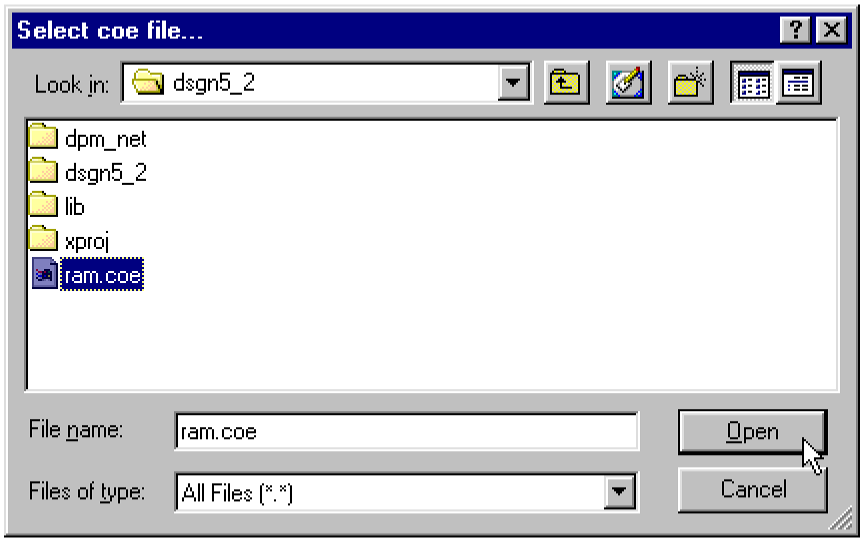
Component Name: RAM

Depth: 16 Address Width: 4

Data Width: 8

.coe file: no init values read - default is all 0s

Next, highlight the ram.coe file in the **Select coe file…** window and click on the Open button. This loads the RAM initialization values into the Core Generator.



Select coe file...

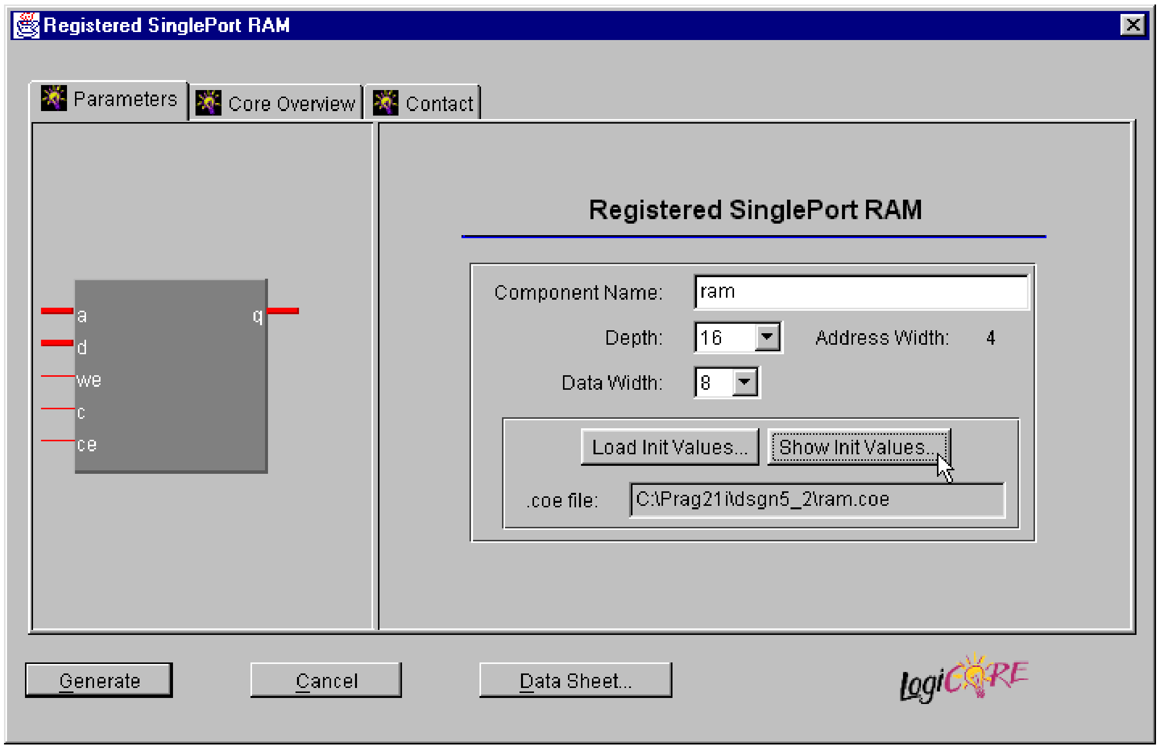
Look in: dsgn5_2

dpm_net
dsgn5_2
lib
xproj
ram.coe

File name: ram.coe

Files of type: All Files (*.*)

Once the initialization values are loaded, click on the Show Init Values... button to view them.



Registered SinglePort RAM

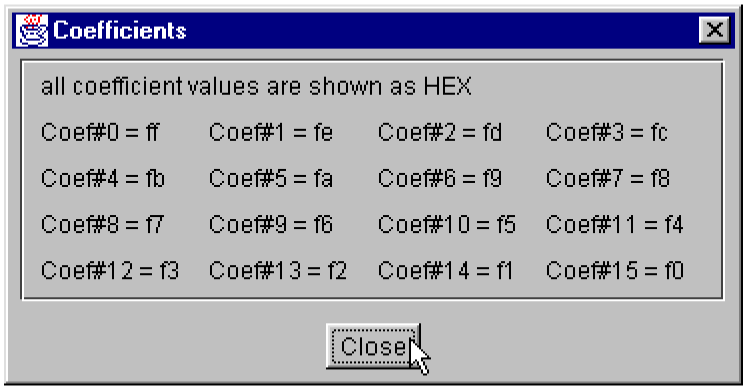
Component Name: ram

Depth: 16 Address Width: 4

Data Width: 8

.coe file: C:\Prag21i\dsgn5_2\ram.coe

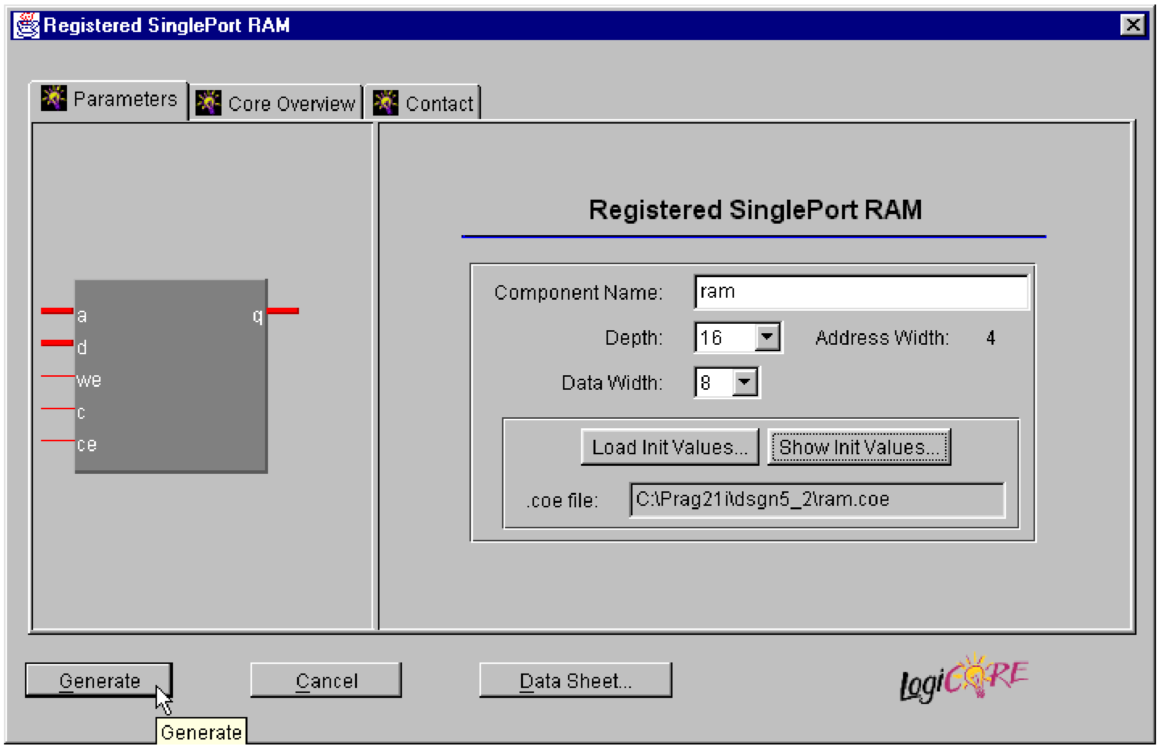
The initial value for each RAM location will appear in the **Coefficients** window. (The locations are labeled Coef# because RAMs inside FPGAs are often used to store tables of coefficients for digital signal processing applications.) Click on the Close button to remove the window.



all coefficient values are shown as HEX

Coef#0 = ff	Coef#1 = fe	Coef#2 = fd	Coef#3 = fc
Coef#4 = fb	Coef#5 = fa	Coef#6 = f9	Coef#7 = f8
Coef#8 = f7	Coef#9 = f6	Coef#10 = f5	Coef#11 = f4
Coef#12 = f3	Coef#13 = f2	Coef#14 = f1	Coef#15 = f0

Once the RAM width, depth and initial values are specified, click the Generate button to have Core Generator assemble the necessary files that describe this RAM.



Registered SinglePort RAM

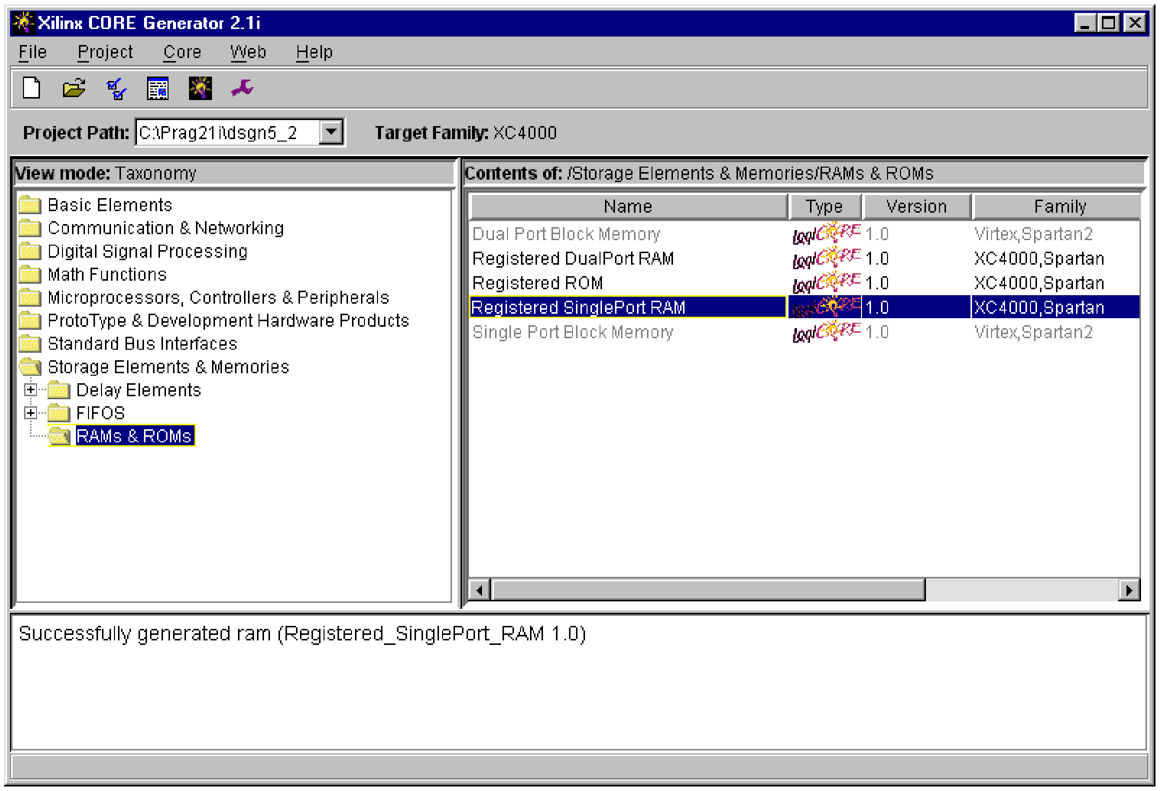
Component Name: ram

Depth: 16 Address Width: 4

Data Width: 8

.coe file: C:\Prag21i\dsgn5_2\ram.coe

The success of the operation will be reported in the bottom pane of the **Core Generator** window.



Xilinx CORE Generator 2.1i
File
Project
Core
Web
Help
Basic Elements
Communication & Networking
Digital Signal Processing
Math Functions
Microprocessors, Controllers & Peripherals
ProtoType & Development Hardware Products
Standard Bus Interfaces
Storage Elements & Memories
Delay Elements
FIFOS
RAMs & ROMs

Project Path: C:\Prag21i\dsgn5_2 **Target Family:** XC4000

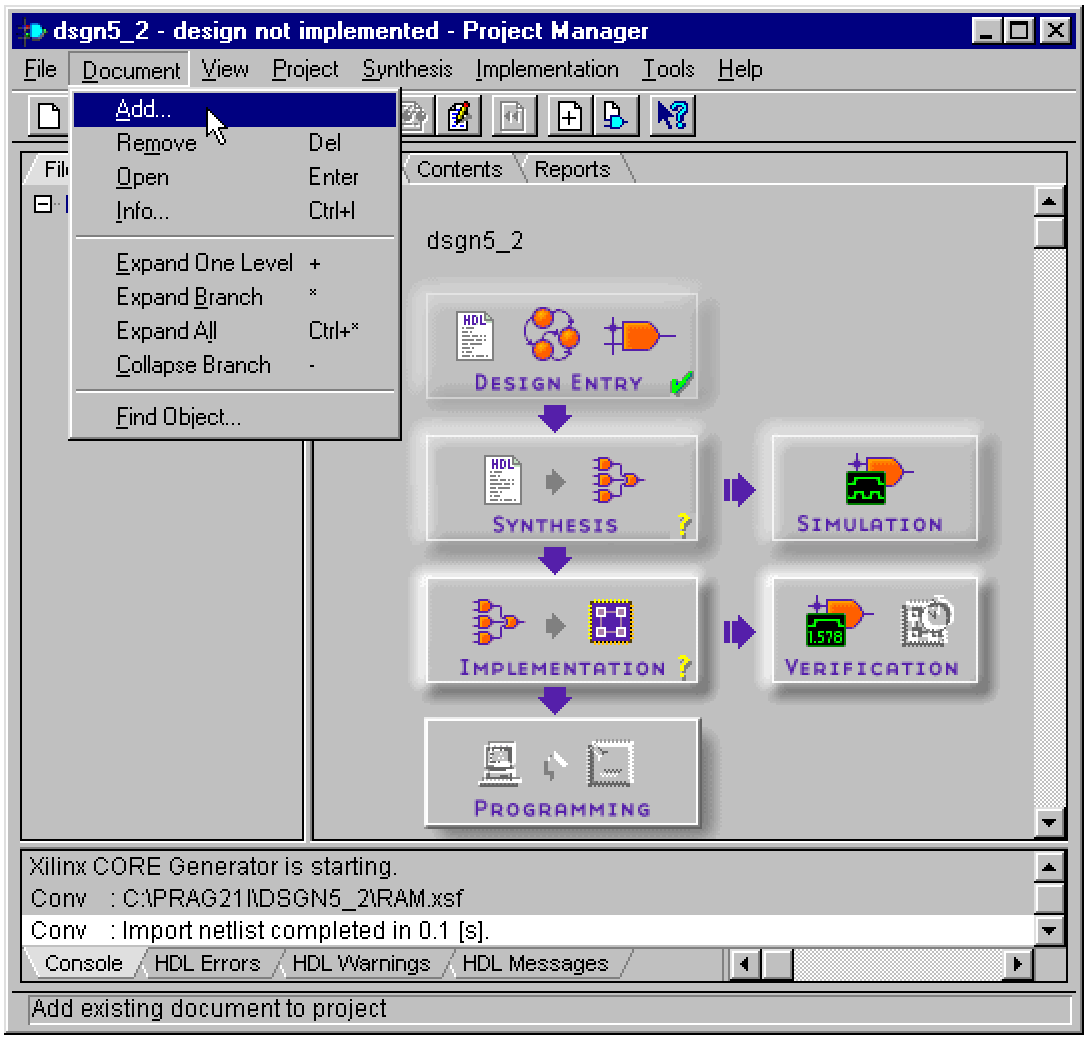
View mode: Taxonomy

Contents of: /Storage Elements & Memories/RAMs & ROMs

Name	Type	Version	Family
Dual Port Block Memory	LogiCORE	1.0	Virtex,Spartan2
Registered DualPort RAM	LogiCORE	1.0	XC4000,Spartan
Registered ROM	LogiCORE	1.0	XC4000,Spartan
Registered SinglePort RAM	LogiCORE	1.0	XC4000,Spartan
Single Port Block Memory	LogiCORE	1.0	Virtex,Spartan2

```
Successfully generated ram (Registered_SinglePort_RAM 1.0)
```

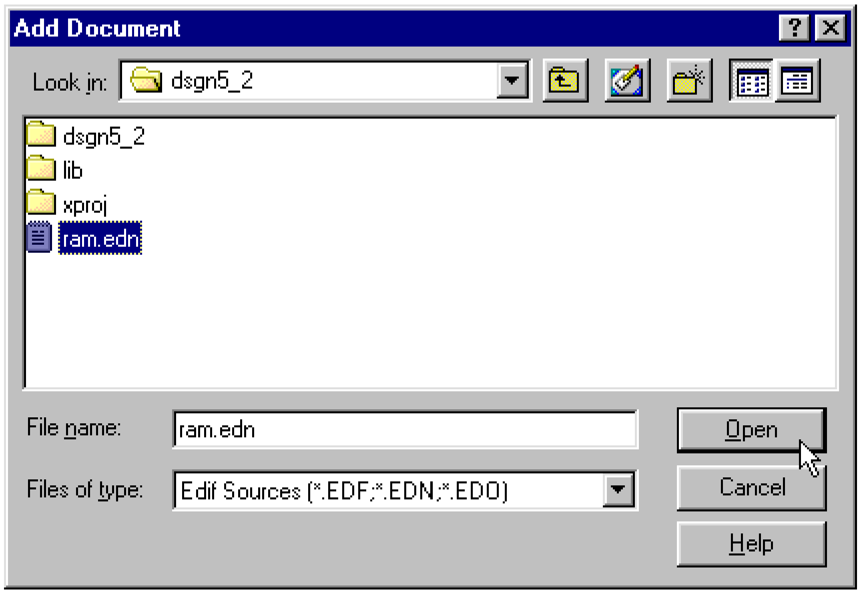
Once the RAM module is generated, we can add it to the project using the Document➔Add... menu command.



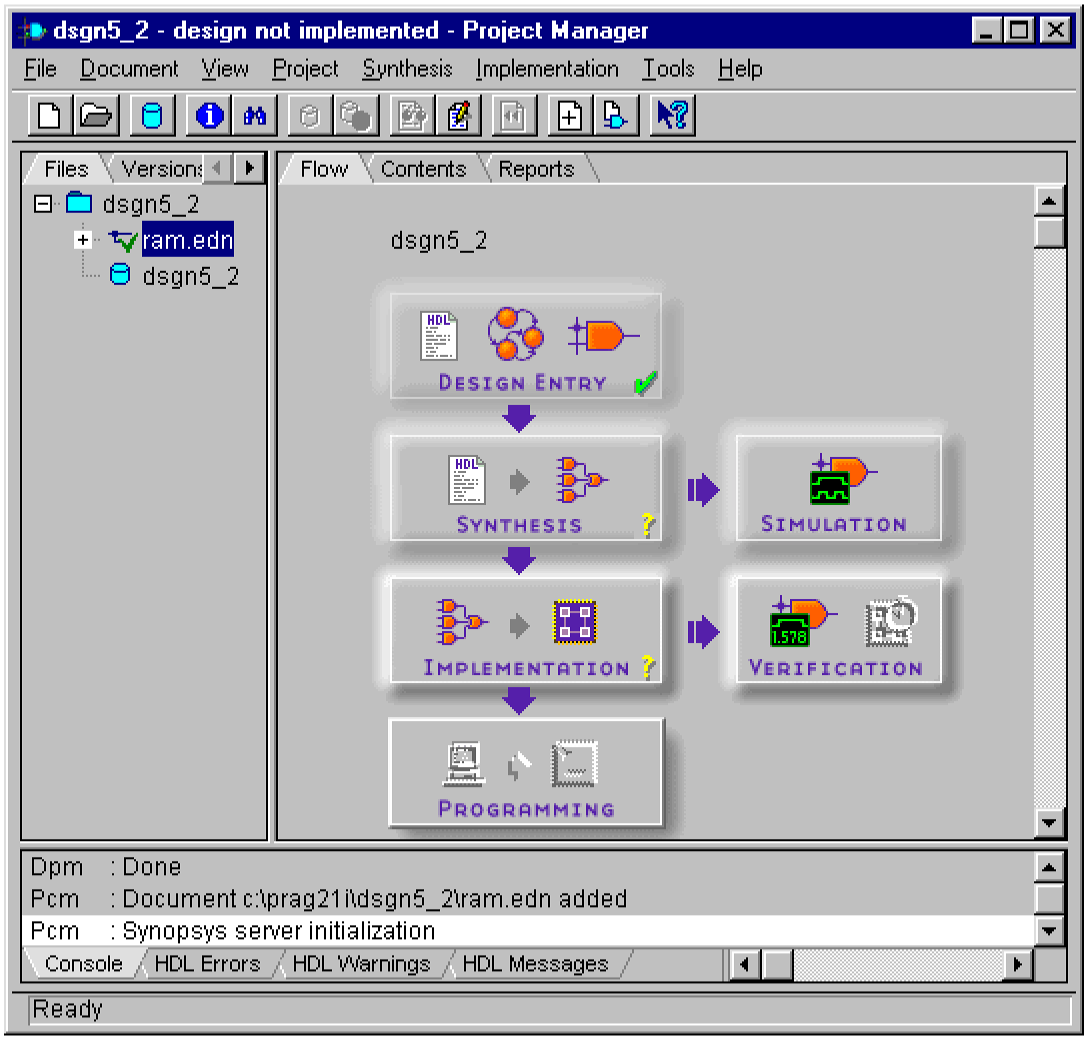
```
Xilinx CORE Generator is starting.
Conv   : C:\PRAG21I\DSGN5_2\RAM.xsf
Conv   : Import netlist completed in 0.1 [s].
```

Add existing document to project

The Core Generator creates modules in the form of EDIF netlist files with the .edn extension. Select Edif Sources in the Files of type field of the **Add Document** window and then you will see the ram.edn file in the top-level directory of the project. Highlight ram.edn and click on the Open button to add the RAM module to the project.



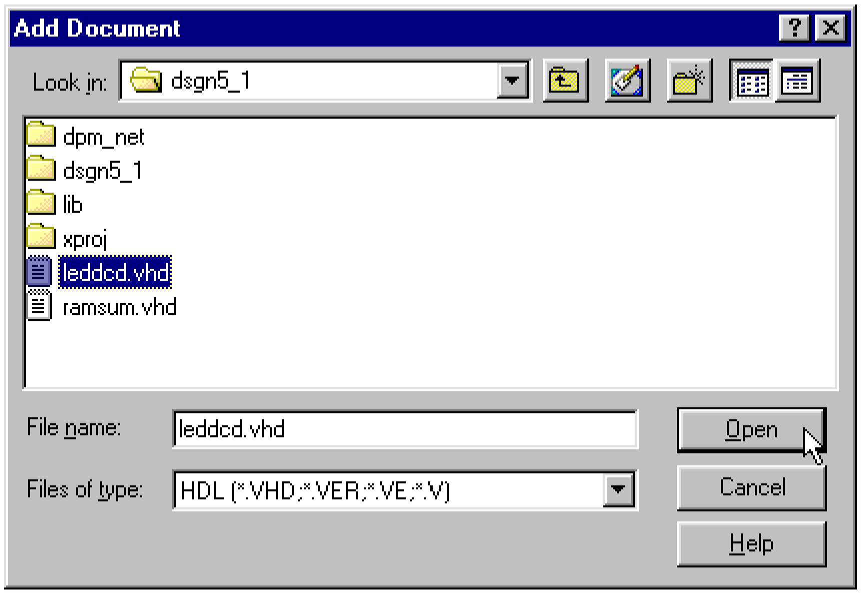
After adding the RAM module, the **Project Navigator** window appears as shown below.


dsgn5_2 - design not implemented - Project Manager
File Document View Project Synthesis Implementation Tools Help
Files Versions
dsgn5_2
ram.edn
dsgn5_2
Flow Contents Reports
dsgn5_2
DESIGN ENTRY
SYNTHESIS
SIMULATION
IMPLEMENTATION
VERIFICATION
PROGRAMMING
Console HDL Errors HDL Warnings HDL Messages
Ready

```
Dpm   : Done
Pcm   : Document c:\prag21i\dsgn5_2\ram.edn added
Pcm   : Synopsys server initialization
```

The LED Decoder Module

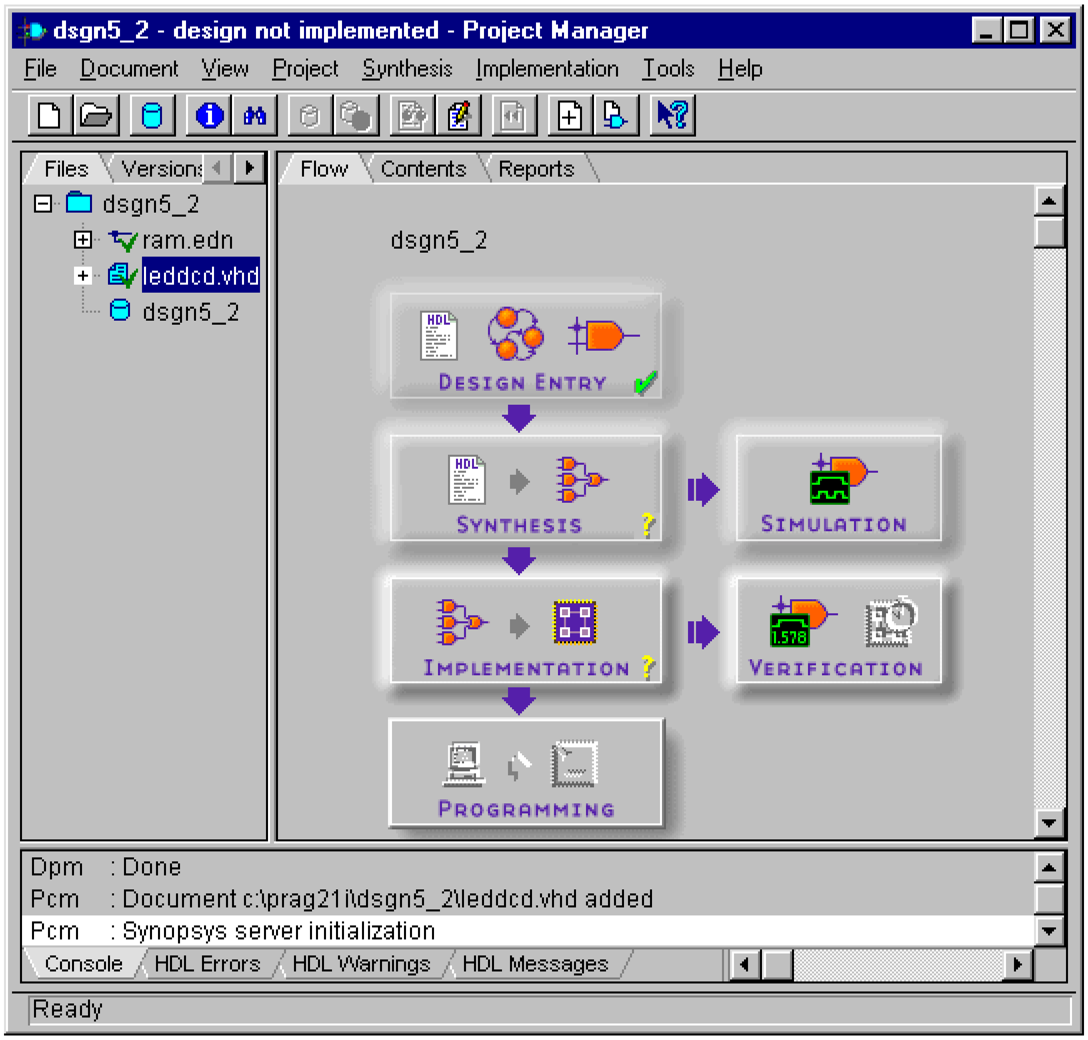
This LED decoder circuit for this project is identical to the one used in the previous project. Use the Document➔Add... menu command and then move to the top-level directory of dsgn5_1. Highlight the leddcd.vhd file and click on the Open button to add the RAM module to the project.



File name: leddcd.vhd

Files of type: HDL (*.VHD;*.VER;*.VE;*.V)

After adding the LED decoder module, the **Project Navigator** window appears as shown below.


dsgn5_2 - design not implemented - Project Manager
File Document View Project Synthesis Implementation Tools Help
Files Versions
dsgn5_2
ram.edn
leddcd.vhd
dsgn5_2
Flow Contents Reports
dsgn5_2
DESIGN ENTRY
SYNTHESIS
SIMULATION
IMPLEMENTATION
VERIFICATION
PROGRAMMING
Console HDL Errors HDL Warnings HDL Messages
Ready

```
Dpm   : Done
Pcm   : Document c:\prag21i\dsgn5_2\leddcd.vhd added
Pcm   : Synopsys server initialization
```

The Root Module

The root module sequences through three main phases just as in the previous example:

Phase 1: Starting from an upper address of RAM and proceeding to address zero, the value stored at each RAM address is read and the two's-complement is computed and written back to the same address.

Phase 2: Restarting from the upper address and proceeding to address zero, each value is read from RAM and added to a sum register.

Phase 3: The sum is displayed on the seven-segment LED by blanking the LED segments for a long interval to signal the start of the sum, then the hexadecimal digit for the upper four bits of the sum are displayed, then the LEDs are blanked for a shorter interval and then the hexadecimal digit for the lower four bits is displayed. Then this four-step display process repeats.

The VHDL code for the root module (Listing 10) was derived from the root module of the previous example is in the ramsum.vhd. The differences between the previous root module and this one are described below.

Lines 6–12: The RAM address, data and control signals are no longer included in the interface definition. That's because the RAM is now internal to the FPGA so we don't need any I/O pins to interface to the external RAM chip.

Lines 15–24: These lines define the interface to the RAM module created by the Core Generator. In addition to the ram.edn netlist file, the Core Generator also creates a ram.vho file that shows the VHDL interface definition for the component and how to instantiate it. We just copied the component declaration from that file.

Lines 25–32: The internal buses for interfacing to the RAM module are declared on these lines as well as the four-bit address register. The address, input data and output data buses used in the state machine are declared with the type `UNSIGNED`. This makes it easier to perform arithmetic operations on their values using the `numeric_std` library. But the RAM module from the Core Generator has input and output buses declared as type `STD_LOGIC_VECTOR` so some intermediary buses are declared to make the type conversion.

Line 33: This line declares the constant for the address of the upper end of the RAM data range that will be summed. As in the previous example, this circuit will complement and sum eleven bytes of data from address zero to ten, inclusive.

Lines 34: A register to hold the sum of the RAM values is declared here. The register to hold the current value read from the RAM in the previous example is no longer needed here because the synchronous RAM outputs will remain stable except on the rising edge of the clock.

Lines 42–43: Only eight states are defined for this design. The `invertnop` state is no longer needed when a synchronous RAM is used.

Lines 57–58: The default values for the active-high RAM read-enable (`ce`) and write-enable (`we`) signals are defined here. The logic 1 on the `ce` input means a rising clock edge will cause the RAM to register the value of the currently addressed location onto its outputs. The logic 0 on the `we` input disables any writes to the RAM.

Line 59: The input data bus to the RAM is set to zero unless it is specifically set to some other value in the state machine. Unlike the example with the external RAM, the input and output data buses of the internal RAM module are separate so we do not need to tristate the bus when it is not in use.

Lines 62–64: The `init` state initializes the state machine for the start of the loop that complements the contents of RAM. The address register is set to point to the upper bound of the RAM data range and the state machine is moved to the start of the two's-complement loop (`invertw`).

Lines 65–68: The `invertw` state activates the write-enable of the RAM. At the time this state is entered, the contents of the RAM address generated in the previous cycle will be available on the RAM output data bus. This value is complemented and written back on the din bus to the same address location. The actual write will take place on the next rising-edge of the clock. Then the state machine is moved to the `invertr` state to read the next RAM location.

Lines 69–78: The `invertr` state determines the next location to be read from RAM depending upon the value of the current address register. If the current RAM address has reached zero, then the address register is reloaded with the starting address of the data range and control branches to the `add` state where the summation of the data takes place. Otherwise, the current address is decremented and control returns to the invertw state so the next data location can be complemented. In either case, the contents at the new address will be available on the outputs of the RAM at the start of the next clock cycle.

Lines 79–88: The `add` state adds the value from RAM to the summation register. If the current RAM address is zero indicating the summation loop is finished, then the time delay register is loaded with the initial blanking interval for the LED display. Then the state machine is moved to the start of the display loop (`display_blank`). If all the RAM data has not been summed, then the RAM address is decremented and the state machine stays in the `add` state. The contents at the new address will be available on the outputs of the RAM at the start of the next clock cycle.

Lines 139–149: The UNSIGNED address and data buses used in the state machine are converted to the STD_LOGIC_VECTOR types and passed to the RAM module created using the Core Generator. As with the component declaration, example code for instantiating the RAM module can be found in the ram.vho file

Listing 10: VHDL code for the root module.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use WORK.leddcd_pckg.all;

entity ramsum is
   port (
      rst   : in STD_LOGIC;                         -- reset
      clk   : in STD_LOGIC;                         -- clock
      s     : out STD_LOGIC_VECTOR(6 downto 0) -- outputs to LED segments
   );
end ramsum;

architecture ramsum_arch of ramsum is
component ram -- 16-byte synchronous RAM from CoreGen
   port (
      a     : IN std_logic_VECTOR(3 downto 0); -- address bus
      d     : IN std_logic_VECTOR(7 downto 0); -- data input bus
      we    : IN std_logic;                     -- write-enable
      c     : IN std_logic;                     -- clock
      ce    : IN std_logic;                     -- read-enable
      q     : OUT std_logic_VECTOR(7 downto 0) -- data output bus
   );
end component;
```

```
-- RAM address, data, control signals
signal     addr_r, next_addr  : UNSIGNED(3 downto 0); -- RAM address reg
signal     din     : UNSIGNED(7 downto 0);    -- RAM data input bus
signal     dout    : UNSIGNED(7 downto 0);    -- RAM data output bus
signal     ce      : STD_LOGIC;               -- RAM chip-enable
signal     we      : STD_LOGIC;               -- RAM write-enable
signal     aa      : STD_LOGIC_VECTOR(addr_r'range);  -- RAM address bus
signal     dd,qq : STD_LOGIC_VECTOR(din'range);     -- RAM data I/O buses
constantmaxaddr : UNSIGNED := TO_UNSIGNED(10,addr_r'length);
signal     sum_r, next_sum : UNSIGNED(din'length-1 downto 0); -- RAM sum
signal     delay_r, next_delay   : UNSIGNED(22 downto 0);  -- delay counter
constantblank_dly  : UNSIGNED := TO_UNSIGNED(5_000_000,delay_r'length);
constant interdigit_dly:UNSIGNED:=TO_UNSIGNED(1_600_000,delay_r'length);
constantdigit_dly  : UNSIGNED := TO_UNSIGNED(2_500_000,delay_r'length);
signal     digit: UNSIGNED(3 downto 0); -- LED hex digit to display
signal     blank: STD_LOGIC;            -- LED digit blanking signal
-- states for the state machine
type state is (init,invertr,invertw,add,display_blank,
        display_upper_digit,display_interdigit,display_lower_digit);
signal     st_r, next_st: state;   -- state register
begin

-- this process computes the actions of the state machine in each state
process(clk,st_r,addr_r,sum_r,delay_r,din)
begin
   -- default outputs unless otherwise specified
   next_st      <= st_r;      -- remain in the current state
   next_addr    <= addr_r;    -- don't change the RAM address
   next_sum   <= sum_r;       -- don't update the sum register
   next_delay <= delay_r-1;-- decrement the delay counter
   digit      <= TO_UNSIGNED(0,digit'length); -- output a '0' LED digit
   blank      <= '1';         -- blank the LED display
   ce           <= '1';          -- always read the RAM
   we           <= '0';          -- don't write to the RAM
   din          <= TO_UNSIGNED(0,din’length);

   case st_r is -- case statement for the state machine
   when init => -- initialization state
      next_addr <= maxaddr;-- start inverting from the upper address
      next_st <= invertw;     -- enter the RAM inversion loop
   when invertw => -- write inverted byte value into same RAM location
      we <= '1';            -- write RAM at the end of this clock cycle
      din <= TO_UNSIGNED(0,din'length) - dout;-- output inverted byte
      next_st <= invertr;     -- now read from next RAM location
   when invertr => -- read byte from RAM
      if addr_r = TO_UNSIGNED(0,addr_r'length) then
         -- reached the lower address of the RAM data
         next_addr <= maxaddr;   -- reload register with upper address
         next_st <= add;         -- enter the summation loop
      else
         -- haven't inverted all the RAM data yet
         next_addr <= addr_r - 1;
         next_st <= invertw;       -- now write to it
      end if;
   when add =>     -- sum the inverted data from RAM
      next_sum <= sum_r + dout;  -- add the RAM data to the sum
      if addr_r = TO_UNSIGNED(0,addr_r'length) then
         -- reached the lower address of the RAM data
         next_delay <= blank_dly;-- load display interval counter
         next_st <= display_blank;  -- now display the sum
      else
         -- haven't summed all the RAM data yet so stay in this state
         next_addr <= addr_r - 1;-- address the next RAM location
      end if;
   when display_blank =>-- blank the display
```

```
      if delay_r = TO_UNSIGNED(0,delay_r'length) then
        -- initial display blanking is complete
        next_delay <= digit_dly;      -- load digit display interval
        next_st <= display_upper_digit; -- display the upper sum digit
      end if;
    when display_upper_digit => -- display the upper digit of the sum
      blank <= '0';                 -- activate the LED
      digit <= sum_r(7 downto 4);   -- display the upper 4-bits of the sum
      if delay_r = TO_UNSIGNED(0,delay_r'length) then
        -- upper digit display is complete
        next_delay <= interdigit_dly; -- load inter-digit blank interval
        next_st <= display_interdigit;  -- blank display between digits
      end if;
    when display_interdigit => -- blank the display between sum digits
      if delay_r = TO_UNSIGNED(0,delay_r'length) then
        -- inter-digit display blanking is complete
        next_delay <= digit_dly;      -- load digit display interval
        next_st <= display_lower_digit; -- display the lower sum digit
      end if;
    when display_lower_digit => -- display the lower digit of the sum
      blank <= '0';                 -- activate the LED
      digit <= sum_r(3 downto 0);   -- display the lower 4-bits of the sum
      if delay_r = TO_UNSIGNED(0,delay_r'length) then
        -- lower digit display is complete
        next_delay <= blank_dly;-- load blank interval between loops
        next_st <= display_blank;  -- loop and display the sum again
      end if;
    when others =>  -- error state
      next_st <= init;-- re-initialize the state machine
    end case;
  end process;

  -- this process updates the registers on every rising clock edge
  process(clk)
  begin
    if clk'event and clk='1' then-- trigger on rising clock edge
      if rst='1' then    -- synchronous reset
        st_r    <= init;
        sum_r <= TO_UNSIGNED(0,sum_r'length);
      else                  -- update the registers
        st_r    <= next_st;
        sum_r <= next_sum;
        addr_r  <= next_addr;
        delay_r <= next_delay;
      end if;
    end if;
  end process;

  -- connect clock, address, data and control to the RAM block
  aa <= STD_LOGIC_VECTOR(next_addr);
  dd <= STD_LOGIC_VECTOR(din);
  qq <= STD_LOGIC_VECTOR(dout);
  u2: ram port map (
          a  => aa,
          d  => dd,
          we => we,
          c  => clk,
          ce => ce,
          q  => qq
          );

  -- output digit on the LED display
  u1: leddcd port map(blank=>blank, d=>digit, s=>s);

  end ramsum_arch;
```

Figure 16 shows the waveforms for the last few cycles of the RAM complementation loop and the first few cycles of the summation loop. Data from RAM address 1 is available at the start of the `invertw` clock cycle. The data is complemented and sent back to the RAM where it is written at the end of the `invertw` cycle. During the following `invertr` cycle, the RAM address is decremented to zero and this is output on the RAM address bus. At the beginning of the next `invertw` cycle, the contents of RAM address 0 become available on the RAM data outputs. The current RAM address of zero is also stored in the address register in the FPGA. The complemented contents of RAM address 0 are written back into the RAM and control returns to the `invertr` state. Since the address register now contains zero, the RAM address is restored back to the start of the data range and control proceeds to the `add` state. During the `add` state the data from the address register location is added to the summation register while the decremented address for the next RAM location is sent to the RAM. The data at the decremented address is available during the next clock cycle and the summation continues until the address register reaches zero.

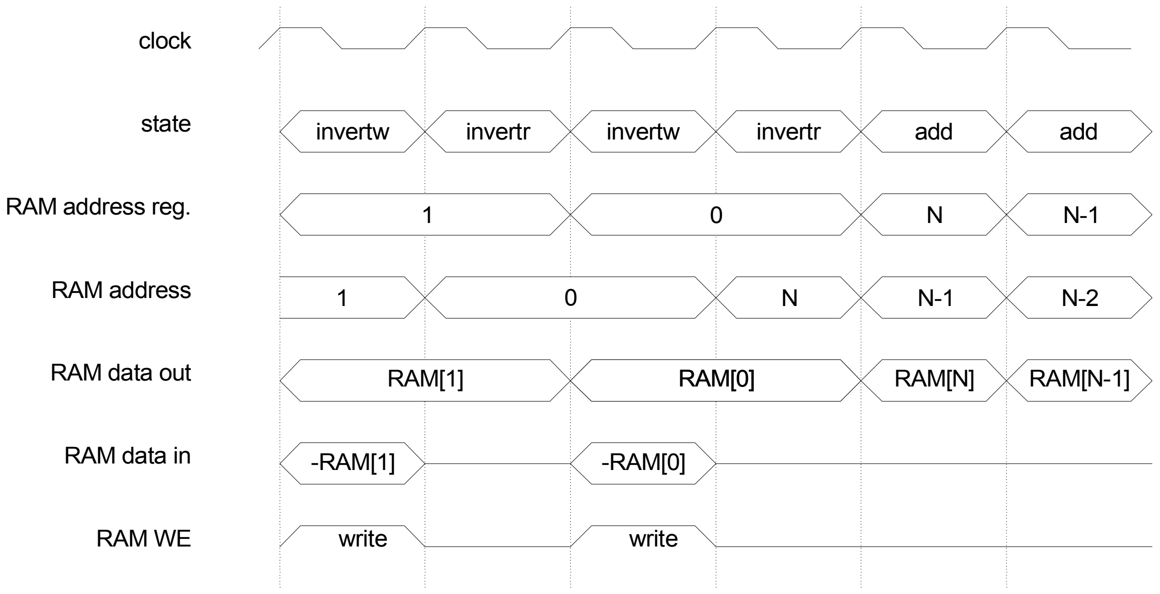


Figure 16: Timing waveforms for the synchronous RAM summation circuit.

The timing waveforms illustrate the fundamental principles involved when writing to a synchronous RAM:

1. The address, data and write-enable signal must be held stable for the setup time before the actual write-operation occurs at the next rising clock edge. Changing the address while the write-enable is active and the clock is either high or low will not cause erroneous writes into other addresses because write operations only occur on a rising clock edge.

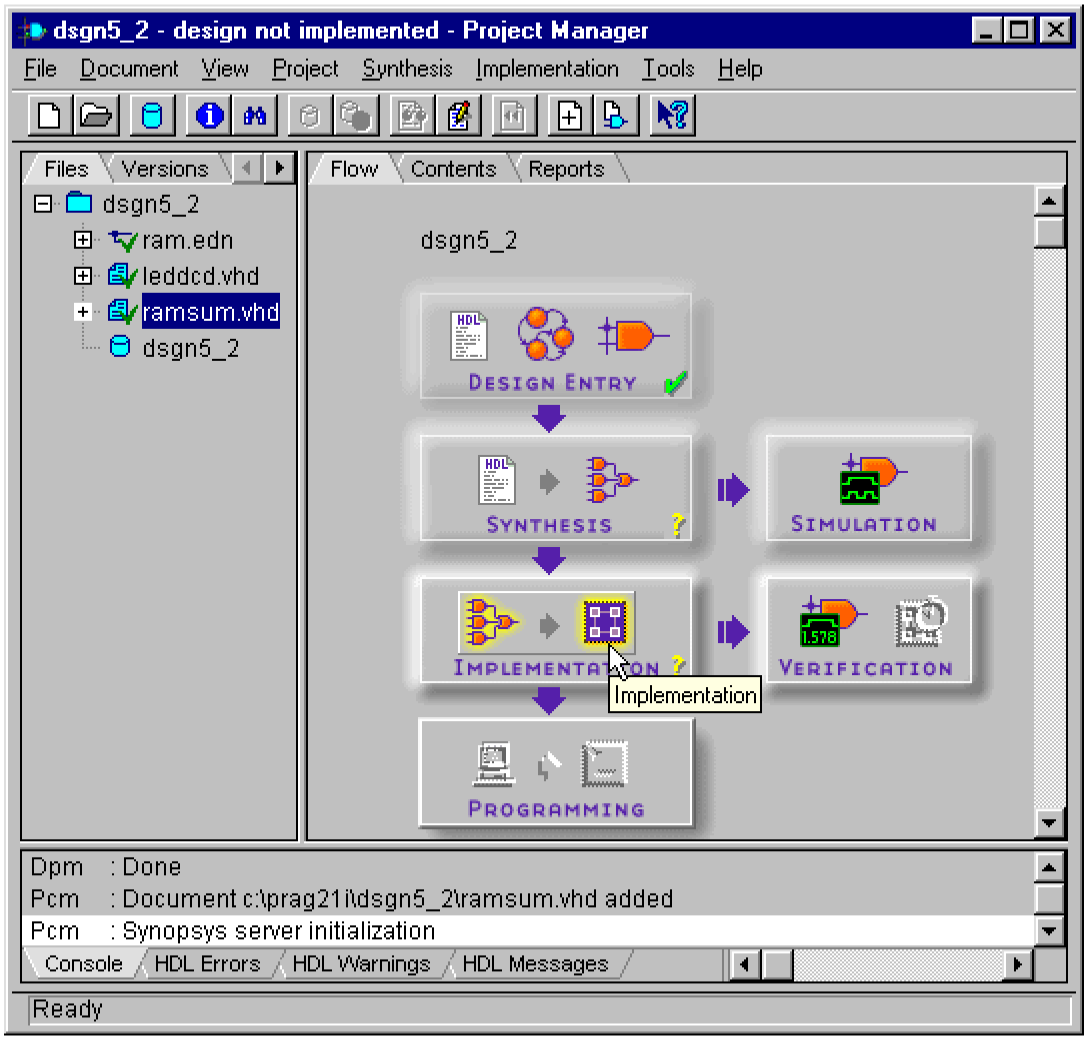
2. There is no need to hold the address or data stable after the rising clock edge during a write operation.

3. For a synchronous RAM with registered outputs, the RAM outputs will show the data that was in the RAM location whose address was present at the previous rising clock edge. These outputs will persist until the next rising clock edge.

For our design, note that the RAM address, data and write-enable are stable before a rising clock edge and then change immediately after the edge. The output RAM data is stable for the entire cycle after the rising clock edge even when the RAM address changes. That means our design can complement the RAM data directly and then send it back to the RAM rather than store it in a register and then operate on it.

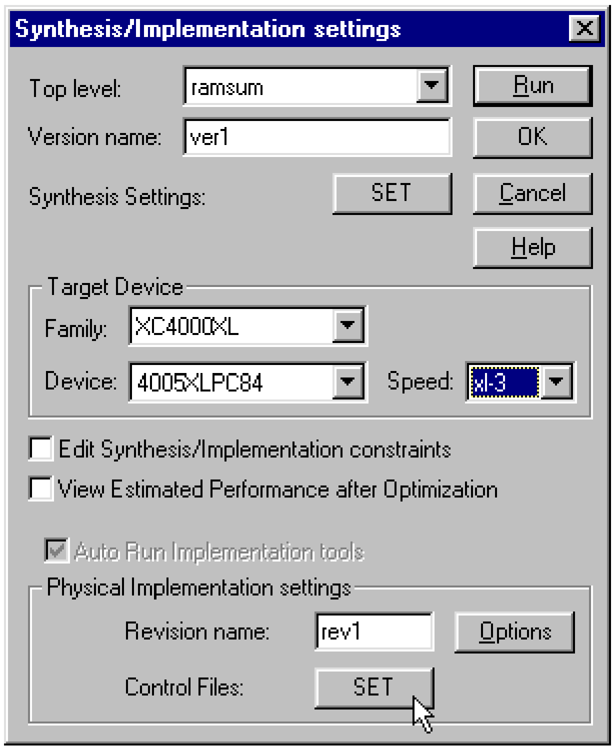
Synthesizing and Implementing the Design

Once the modules are checked for syntax and any errors are removed, we can run the synthesis and implementation tools to create the configuration bitstream for the FPGA or CPLD. Click on the Implementation icon to run the synthesizer and the implementation tools sequentially.



```
Dpm   : Done
Pcm   : Document c:\prag21i\dsgn5_2\ramsum.vhd added
Pcm   : Synopsys server initialization
```

We will target this design to the XS40 Board, so set the target device to be an XC4005XLPC84 with a –3 speed grade. Then select the **ramsum** module as the top-level module for the design. Then click on the SET button so we can specify the constraint file containing the pin assignments.


Synthesis/Implementation settings

Top level: ramsum

Version name: ver1

Synthesis Settings: SET

Target Device

Family: XC4000XL

Device: 4005XLPC84

Speed: xl-3

☐ Edit Synthesis/Implementation constraints

☐ View Estimated Performance after Optimization

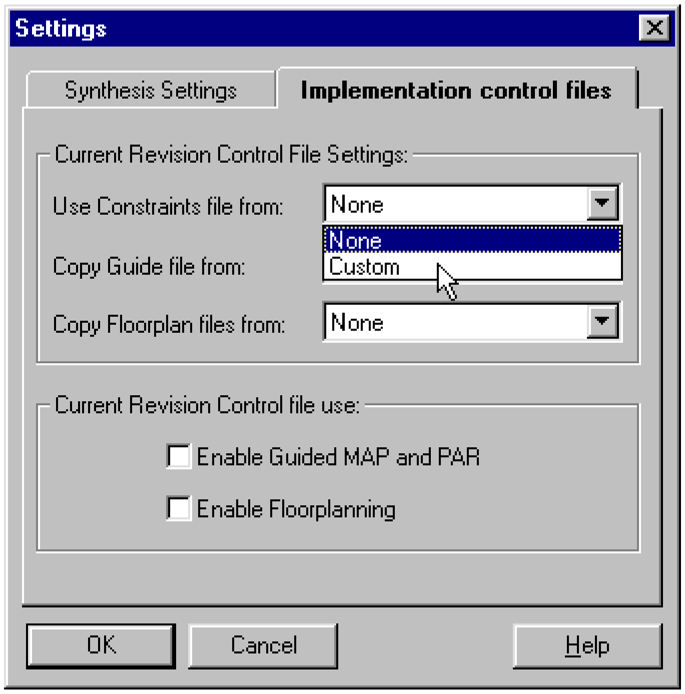
☑ Auto Run Implementation tools

Physical Implementation settings

Revision name: rev1

Control Files: SET

Select the Custom entry in the drop-down list of constraint files.


Settings

Current Revision Control File Settings:

Use Constraints file from: None

None

Custom

Copy Guide file from:

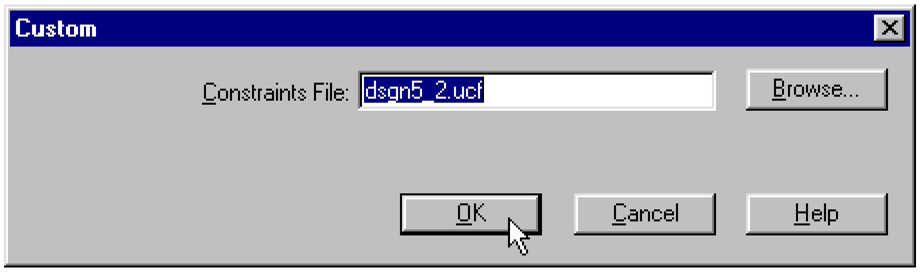
Copy Floorplan files from: None

Current Revision Control file use:

☐ Enable Guided MAP and PAR

☐ Enable Floorplanning

The **Custom** window should appear with the dsgn5_2.ucf file already in the Constraints File field. If not, click on the Browse button, find this file in the top-level directory of the ***dsgn5_2*** project and select it. Then click on the OK button.



The dsgn5_2.ucf file should specify the assignments for the FPGA or CPLD pins that connect to the clock, reset, seven-segment LED and RAM address, data and control pins as shown in Figure 17. The pin assignments for the XS40 Board (which is our target for this example) are shown in Listing 11.

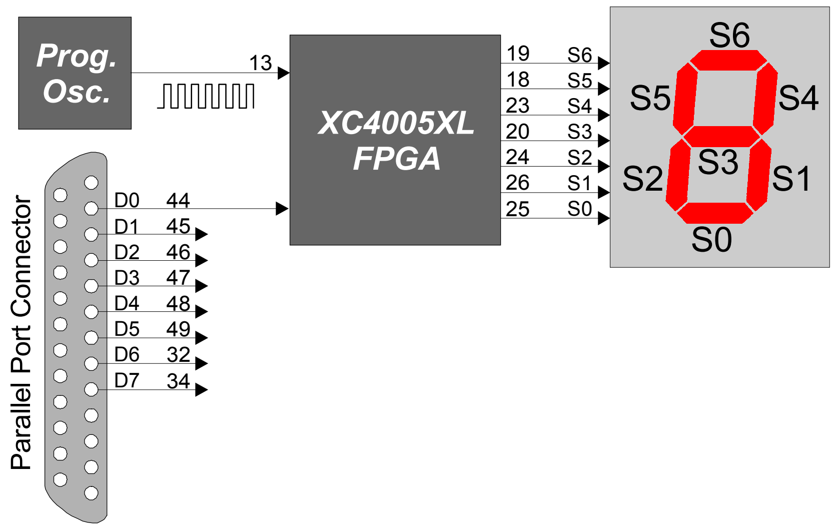
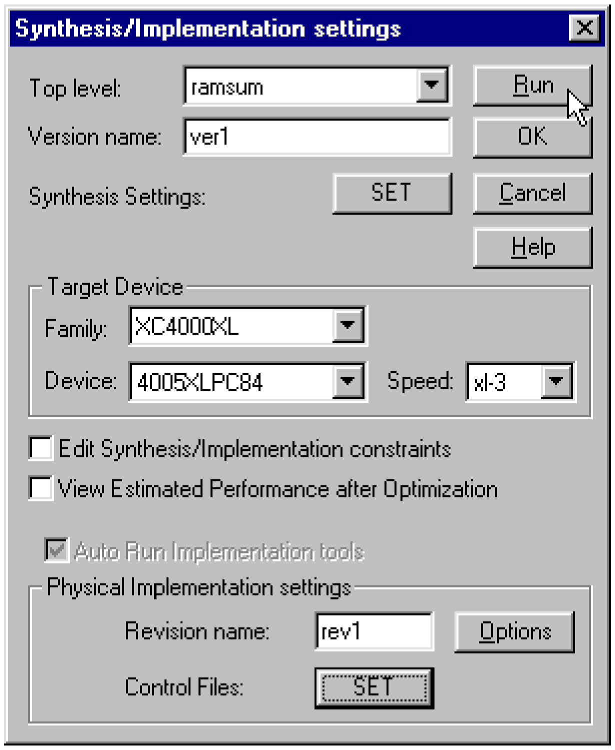


Figure 17: Connection of the programmable oscillator, parallel port, and LED digit to the pins of the FPGA or CPLD on the XS40 Board.

Listing 11: Pin assignments for the XS40 Board.

```
# pin assignments for XS40 Board
net clk loc=p13;  # clock from programmable osc.
net rst loc=p44;  # reset from data pin D0 of parallel port
net s<0> loc=p25; # LED segment S0
net s<1> loc=p26; # LED segment S1
net s<2> loc=p24; # LED segment S2
net s<3> loc=p20; # LED segment S3
net s<4> loc=p23; # LED segment S4
net s<5> loc=p18; # LED segment S5
net s<6> loc=p19; # LED segment S6
```

Once the target device, top-level module, implementation options and constraint file are setup, click on the Run button to start the synthesis and implementation phases.


Synthesis/Implementation settings

Top level: ramsum

Version name: ver1

Synthesis Settings: SET

Target Device

Family: XC4000XL

Device: 4005XLPC84

Speed: xl-3

[ ] Edit Synthesis/Implementation constraints

[ ] View Estimated Performance after Optimization

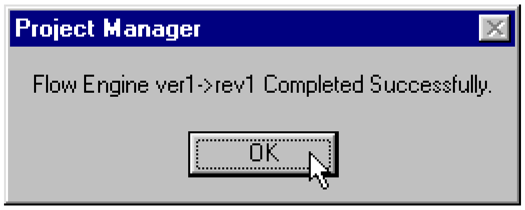
[✓] Auto Run Implementation tools

Physical Implementation settings

Revision name: rev1

Control Files: SET

Both phases should complete with no problems.

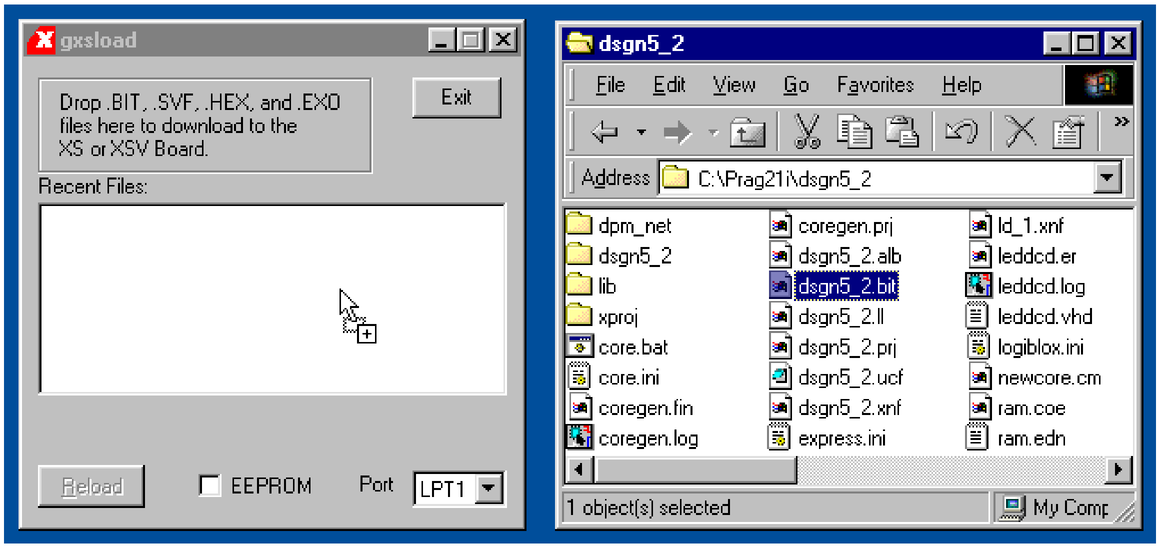

Project Manager

Flow Engine ver1->rev1 Completed Successfully.

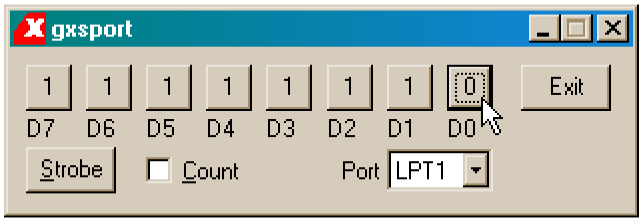
Downloading and Testing the Design

The bitstream file in this example contains both the FPGA configuration and the initial data for the internal RAM, so there is no need to create a separate data file to initialize the RAM as in the last example. The data in the internal RAM is identical to what was used in the previous example, so the result of the complement-and-sum process should be the same: 42 in two-digit hexadecimal.

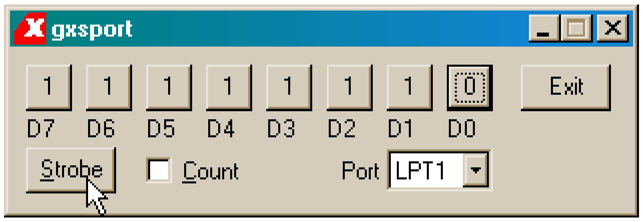
Connect an XS40 Board to the PC parallel port and start the GXSLOAD program. Go to the top-level directory for the ***dsgn5_2*** project and select the dsgn5_2.bit file. Then drag-and-drop it into the **gxsload** window. The bitstream file will be programmed into the XC4005XL FPGA on the XS40 Board.



The reset for the circuit is controlled by data pin D0 of the parallel port. If D0 is at logic 1 after the downloading completes, the circuit will be held in the reset state and the LED will be blank. To release the reset, open the **gxsport** window and click on the D0 button until it displays a zero.

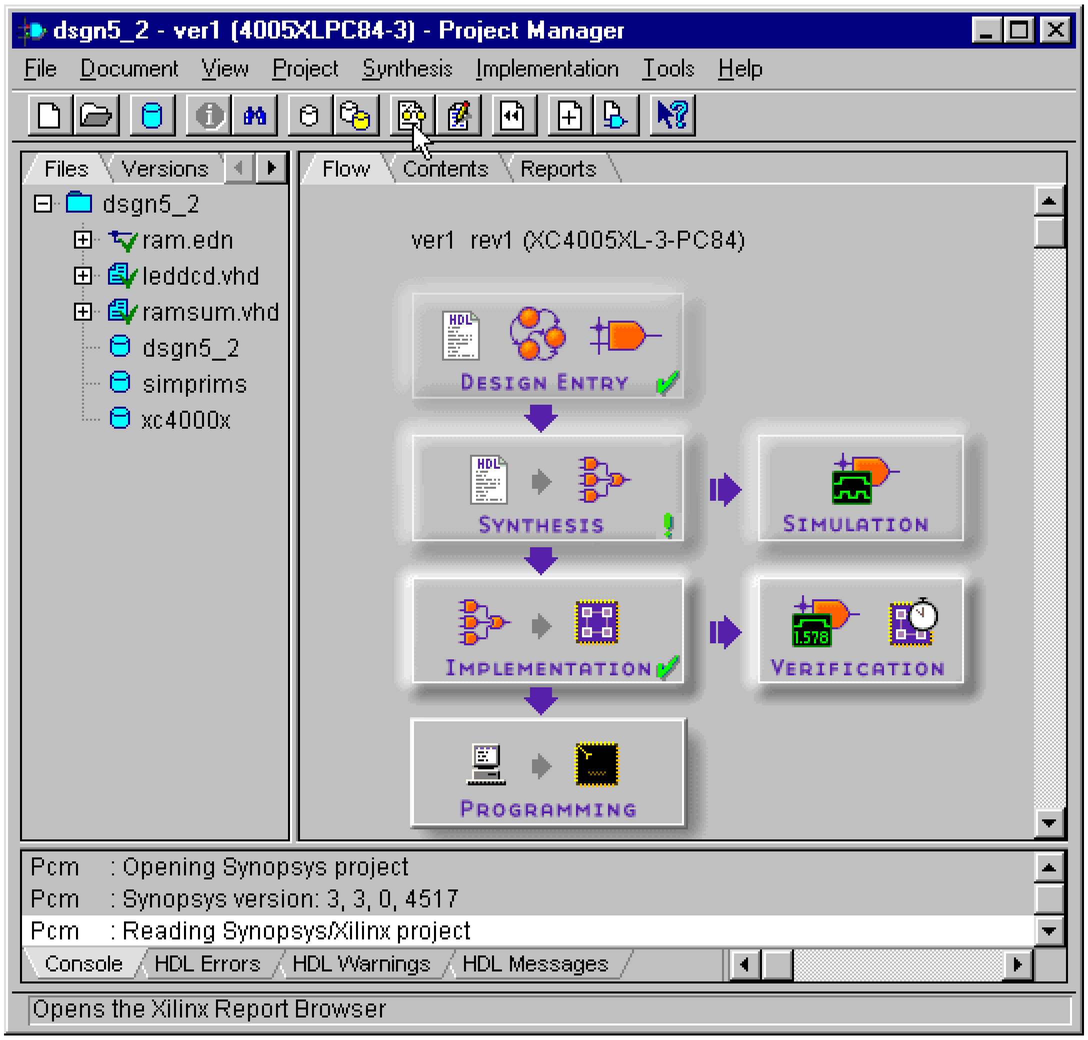


Then click on the Strobe button so the logic 0 value is output on the D0 pin of the parallel port.



Now you will observe the seven-segment LED repeatedly displaying the sequence ……4…2……4…2……. However, it may be blinking too rapidly. Why?

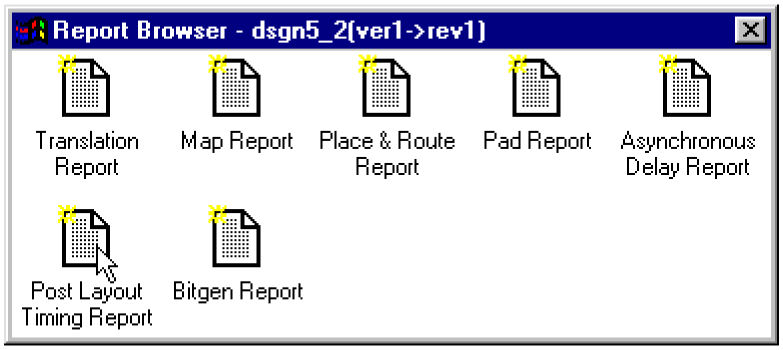
The answer is that you are probably running the design with a 50 MHz clock (the default for the XS40 Board). But the constants that determine the blanking and display intervals for the LED digit were calculated based on a 5 MHz clock. Can this design even run at 50 MHz? Let's check the timing for the implemented design. Click on the icon for the report files in the **Project Navigator** window.


dsgn5_2 - ver1 (4005XLPC84-3) - Project Manager
File Document View Project Synthesis Implementation Tools Help
Files Versions
dsgn5_2
ram.edn
leddcd.vhd
ramsum.vhd
dsgn5_2
simprims
xc4000x
Flow Contents Reports
ver1 rev1 (XC4005XL-3-PC84)
DESIGN ENTRY
SYNTHESIS
SIMULATION
IMPLEMENTATION
VERIFICATION
PROGRAMMING
Console HDL Errors HDL Warnings HDL Messages

```
Pcm    : Opening Synopsys project
Pcm    : Synopsys version: 3, 3, 0, 4517
Pcm    : Reading Synopsys/Xilinx project
```

Opens the Xilinx Report Browser

Then double-click the Post Layout Timing Report in the **Report Browser** window.

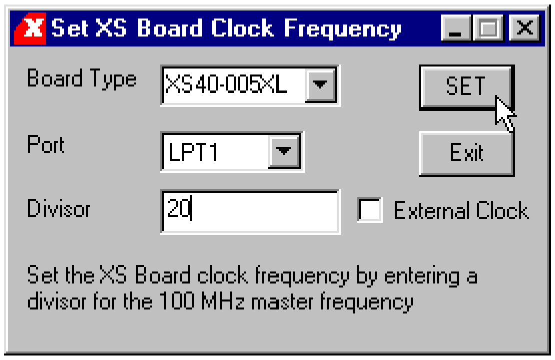

Report Browser - dsgn5_2(ver1->rev1)
Translation Report
Map Report
Place & Route Report
Pad Report
Asynchronous Delay Report
Post Layout Timing Report
Bitgen Report

The top portion of the timing report is shown in Listing 12 and this tells us what we want to know: the minimum clock period for this design is 40.096 ns which translates to a maximum operating frequency of 24.94 MHz. The clock frequency is higher for this design than in the last example because the XC4005XL FPGA has specialized carry propagation circuitry that speeds the addition and complementation operations. It is not surprising that this design runs at 50 MHz when the FPGA is at room temperature and the power supply is optimal.

Listing 12: Timing report for the design.

```
--------------------------------------------------------------------
Xilinx TRACE, Version C.22
Copyright (c) 1995-1999 Xilinx, Inc.  All rights reserved.

Design file:               dsgn5_2.ncd
Physical constraint file: dsgn5_2.pcf
Device,speed:              xc4005xl,-3 (C 1.1.2.2 FINAL)
Report level:              error report
--------------------------------------------------------------------

WARNING:Timing:181 - No timing constraints found, doing default enumeration.

====================================================================
Timing constraint: Default period analysis
 3080 items analyzed, 0 timing errors detected.
 Minimum period is  40.096ns.
--------------------------------------------------------------------

====================================================================
Timing constraint: Default net enumeration
 156 items analyzed, 0 timing errors detected.
 Maximum net delay is  13.074ns.
--------------------------------------------------------------------

…
…
…

Timing summary:
---------------

Timing errors: 0  Score: 0

Constraints cover 3080 paths, 156 nets, and 525 connections (100.0% coverage)

Design statistics:
   Minimum period:  40.096ns (Maximum frequency:  24.940MHz)
   Maximum net delay:  13.074ns

Analysis completed Wed Jan 30 16:45:43 2002
--------------------------------------------------------------------
```

We need to reduce the clock frequency of the XS40 Board to 5 MHz to slow the display of the sum. To do this, start the GXSSETCLK program. Set the Board Type field to XS40-005XL. Place 20 in the Divisor field to reduce the 100 MHz master frequency to 5 MHz. Then click on the SET button.


Set XS Board Clock Frequency

Board Type: XS40-005XL

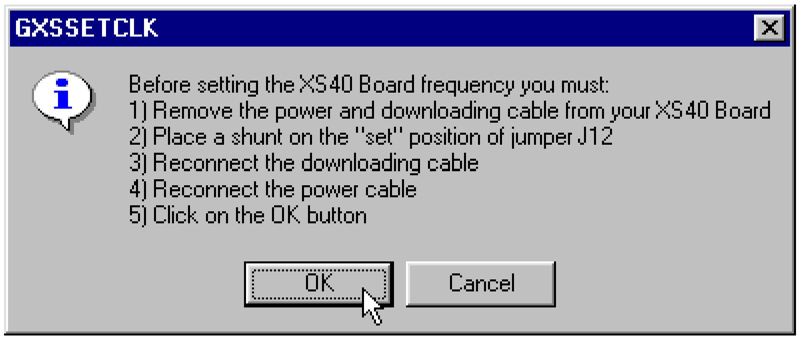
Port: LPT1

Divisor: 20

External Clock

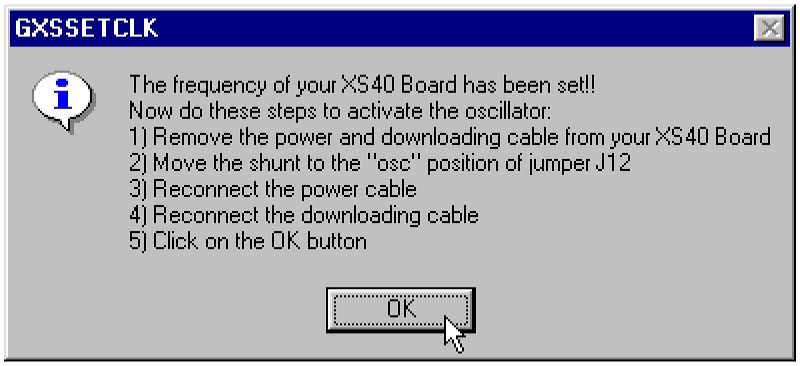
Set the XS Board clock frequency by entering a divisor for the 100 MHz master frequency

A set of instructions will appear that must be followed to adjust the clock frequency of the XS40 Board. After doing these steps, click on the OK button to reprogram the clock.


GXSSETCLK

Before setting the XS40 Board frequency you must:
1) Remove the power and downloading cable from your XS40 Board
2) Place a shunt on the "set" position of jumper J12
3) Reconnect the downloading cable
4) Reconnect the power cable
5) Click on the OK button

Reprogramming the clock takes less than a minute after which the following set of instructions is given to activate the new clock frequency.


GXSSETCLK

The frequency of your XS40 Board has been set!!
Now do these steps to activate the oscillator:
1) Remove the power and downloading cable from your XS40 Board
2) Move the shunt to the "osc" position of jumper J12
3) Reconnect the power cable
4) Reconnect the downloading cable
5) Click on the OK button

After activating the 5 MHz clock frequency, we can again download the dsgn5_2.BIT file again and release the reset on the circuit. Now we should see the ……4…2……4…2…… sequence displayed on the LED digit at a more leisurely pace. And just as with the previous example, if we set and clear the reset again we will see the display change to ……B…E……B…E……