



Synplicity®

Simply Better Results...

Download from: www.FPGA.com.cn

Agenda

- ▶ Product Overview

- ▶ Synplify

- ▶ Synplify Pro

- ▶ Amplify

Agenda

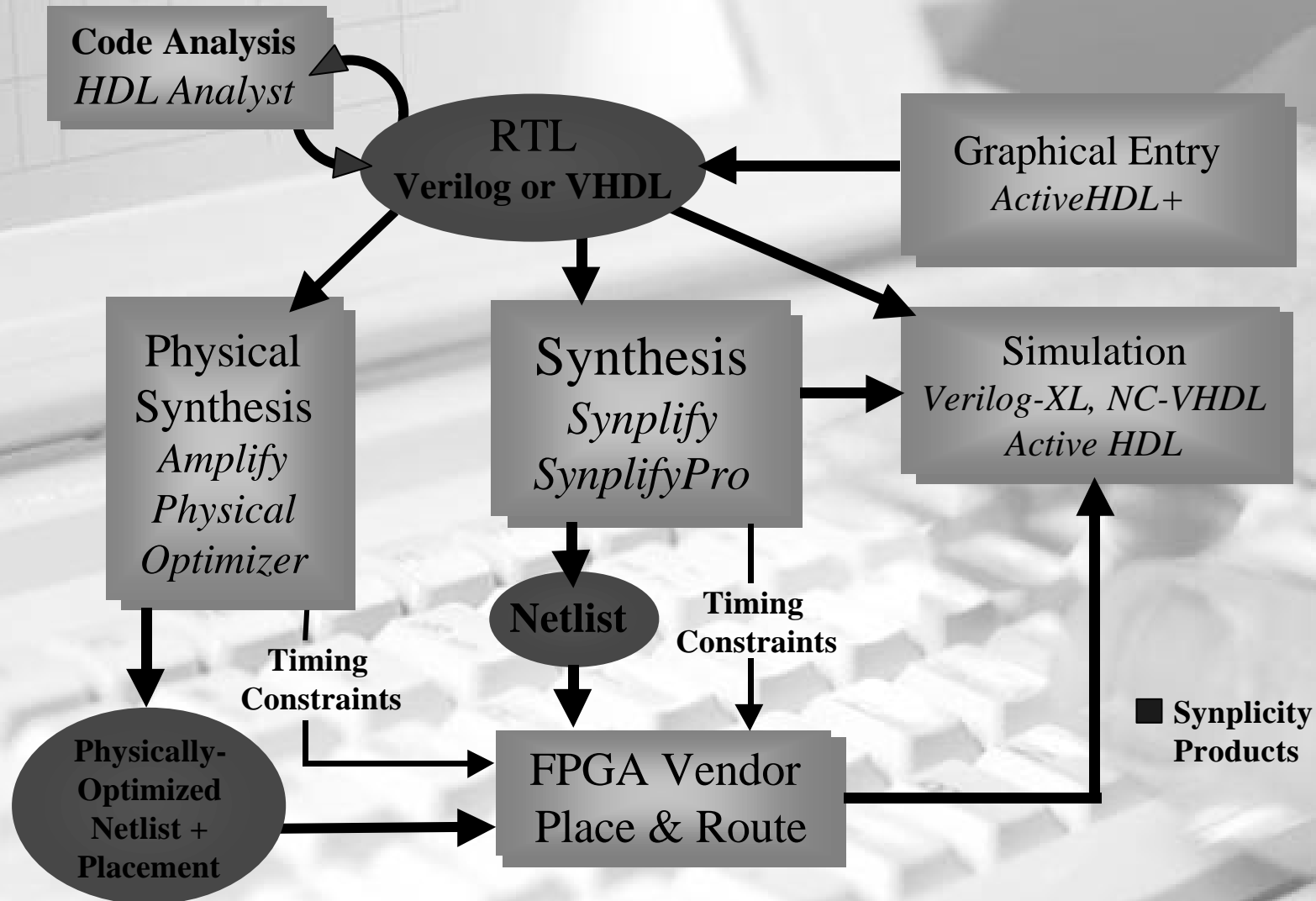
► Product Overview ←

► Synplify

► Synplify Pro

► Amplify

Design Flow Overview



FPGA Product Line Overview

Amplify™ Physical Optimizer™

Physical Synthesis for FPGAs

- **Highest Circuit Performance**
- **Fastest Timing Closure**
- **Option to Synplify Pro**

Synplify® Pro

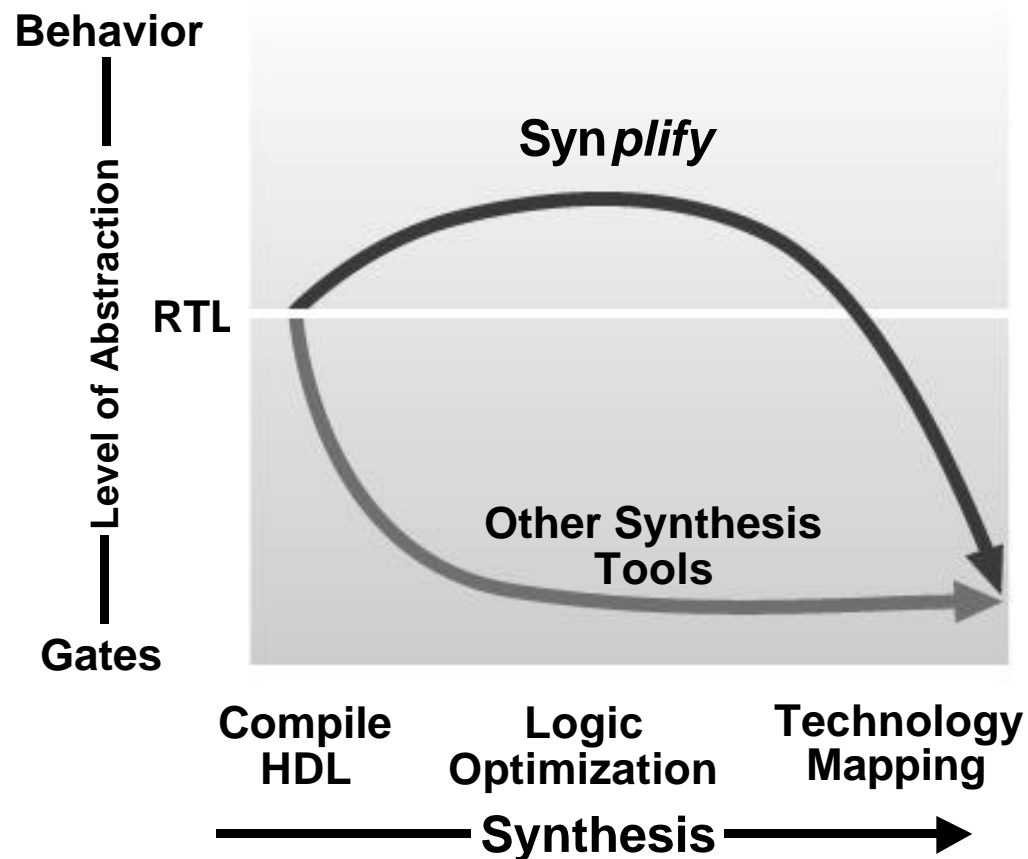
- **Challenging Designs**
- **Complex Projects**
- **The Ultimate in FPGA synthesis**

Synplify®

- **Fast**
- **Easy to Use**
- **Excellent Results**

Why Synplify is BEST for FPGA Synthesis

High-level structure is preserved throughout optimization and mapping



B.E.S.T.TM

Behavior
Extracting
Synthesis
Technology

- Multi-million gate capacity
- Ultra-Fast Compile Times
- Optimization Across Hierarchical Boundaries

Agenda

► Product Overview

► Synplify

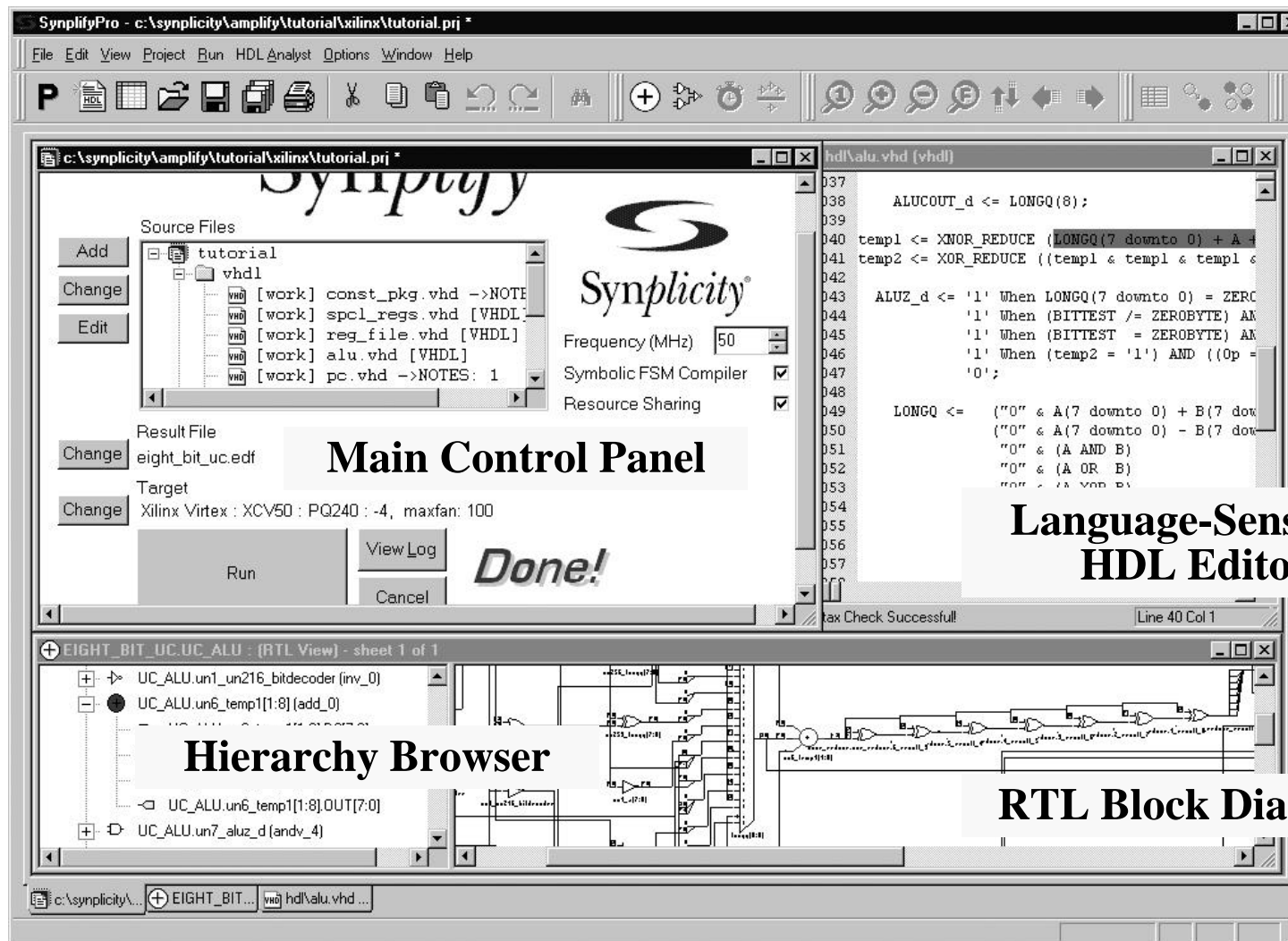


► Synplify Pro

► Amplify

Intuitive User Interface

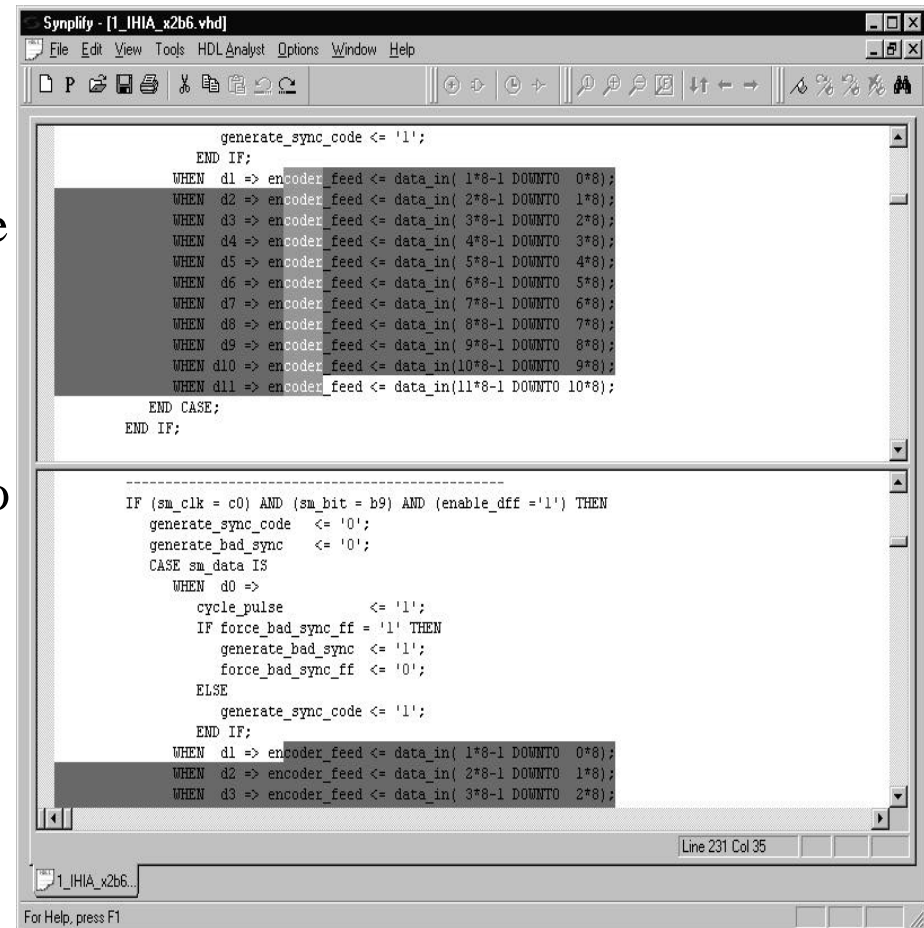
Easy to Learn & Use, Yet Powerful



- ▶ Ultra Fast
- ▶ Easy to Use
- ▶ Excellent Results

HDL Editor

- ▶ Context sensitive
- ▶ Split screen
 - ▶ View multiple portions of the same file simultaneously with instant update of edits
- ▶ Column editing
 - ▶ Use <Alt> and mouse drag to select any rectangular region for editing (copy/cut/paste)
- ▶ Synthesis Check



HDL Analyst: Powerful Graphical Debugging

Language Sensitive Editor

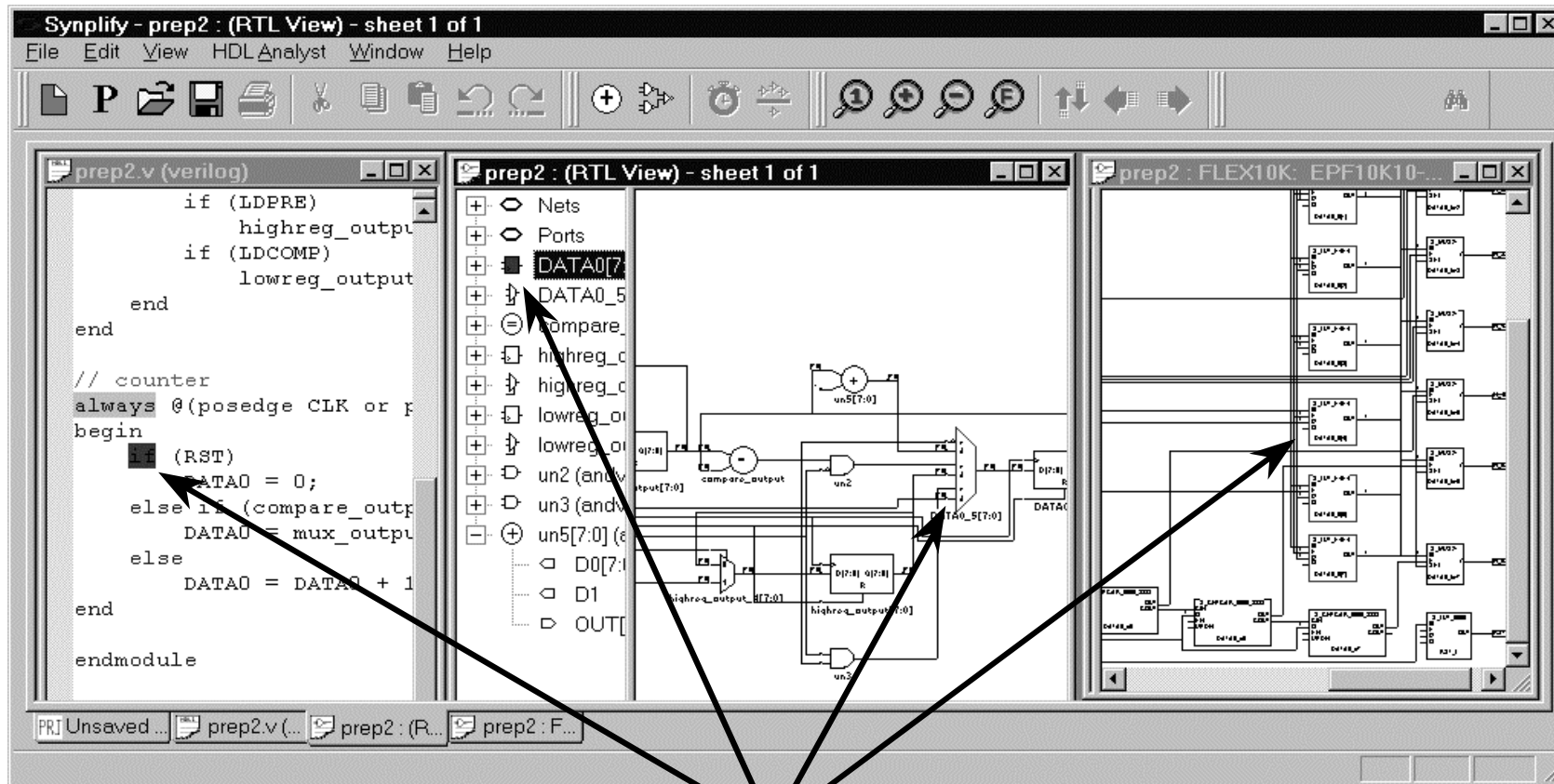
Describe the design
functionality

Unique RTL View

Analyze a technology-
independent block diagram

Technology View

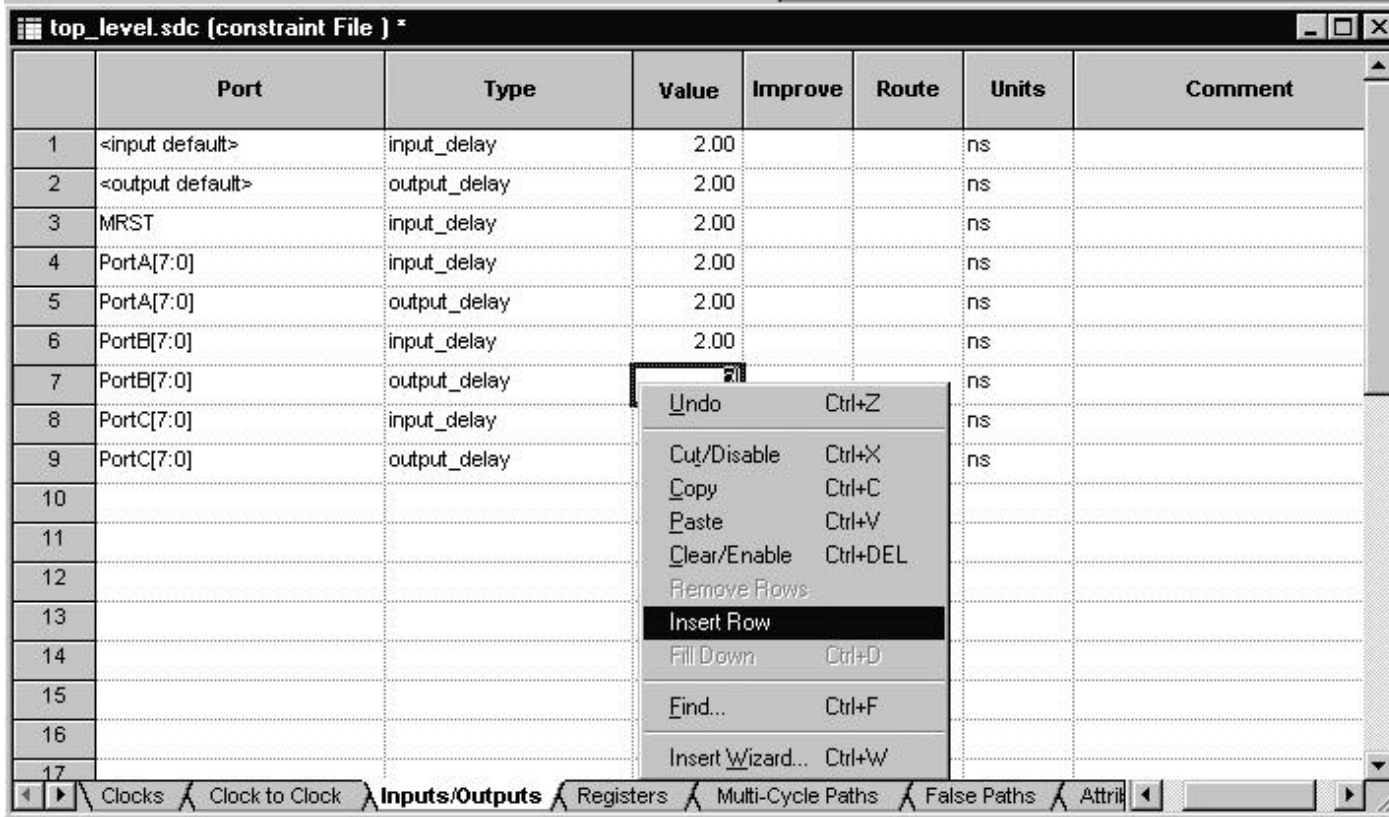
View post-mapped schematic
with annotated timing



Bi-directional cross-probing between all views

SCOPE™ Constraints Editor

Provides Control Over Your Results



The screenshot shows the SCOPE Constraints Editor window titled "top_level.sdc (constraint File) *". It contains a table with 8 columns: Port, Type, Value, Improve, Route, Units, and Comment. The table lists constraints for input and output delays. A context menu is open over the table, showing options like Undo, Cut/Disable, Copy, Paste, Clear/Enable, Remove Rows, Insert Row, Fill Down, Find..., and Insert Wizard... The bottom of the window has a tabbed interface with tabs for Clocks, Clock to Clock, Inputs/Outputs, Registers, Multi-Cycle Paths, False Paths, and Attrs.

	Port	Type	Value	Improve	Route	Units	Comment
1	<input default>	input_delay	2.00			ns	
2	<output default>	output_delay	2.00			ns	
3	MRST	input_delay	2.00			ns	
4	PortA[7:0]	input_delay	2.00			ns	
5	PortA[7:0]	output_delay	2.00			ns	
6	PortB[7:0]	input_delay	2.00			ns	
7	PortB[7:0]	output_delay				ns	
8	PortC[7:0]	input_delay				ns	
9	PortC[7:0]	output_delay				ns	
10							
11							
12							
13							
14							
15							
16							
17							

Context Menu Options:

- Undo (Ctrl+Z)
- Cut/Disable (Ctrl+X)
- Copy (Ctrl+C)
- Paste (Ctrl+V)
- Clear/Enable (Ctrl+DEL)
- Remove Rows
- Insert Row
- Fill Down (Ctrl+D)
- Find... (Ctrl+F)
- Insert Wizard... (Ctrl+W)

Bottom Tabs: Clocks, Clock to Clock, Inputs/Outputs, Registers, Multi-Cycle Paths, False Paths, Attrs

**Clocks(s)
Control**

**I/Os and
Registers**

**Multi-Cycle
& False Paths**

High Levels of User Control

Synthesis Constraints OPTimization Environment

▀ Design Constraints

- ▀ clocks
- ▀ registers
- ▀ I/Os
- ▀ Multi-cycle paths
- ▀ False paths
- ▀ Black Box

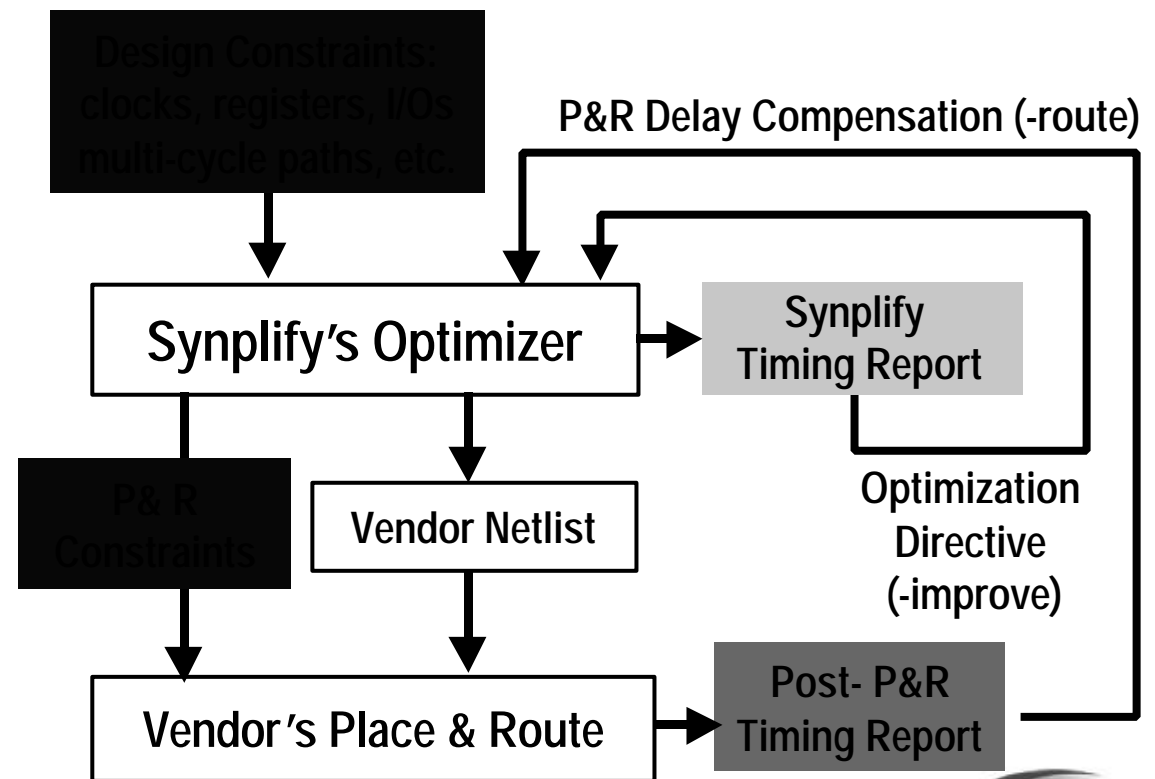
▀ Optimization Directives

- ▀ - improve

▀ P&R Delay Compensation

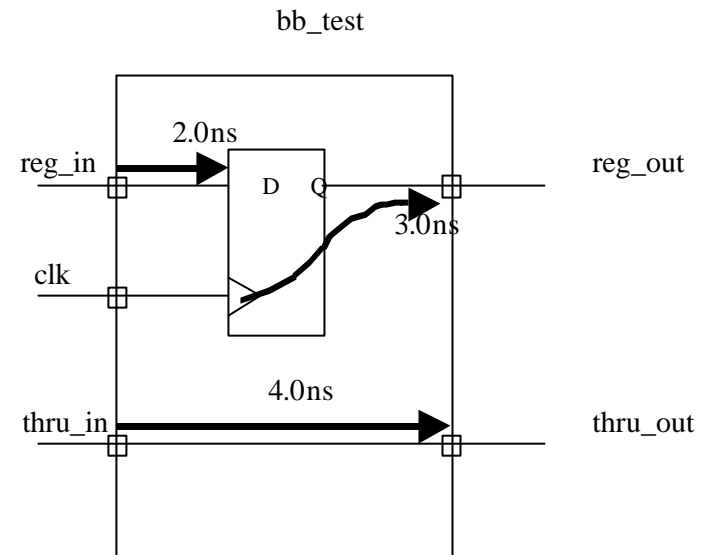
- ▀ - route

SCOPE™



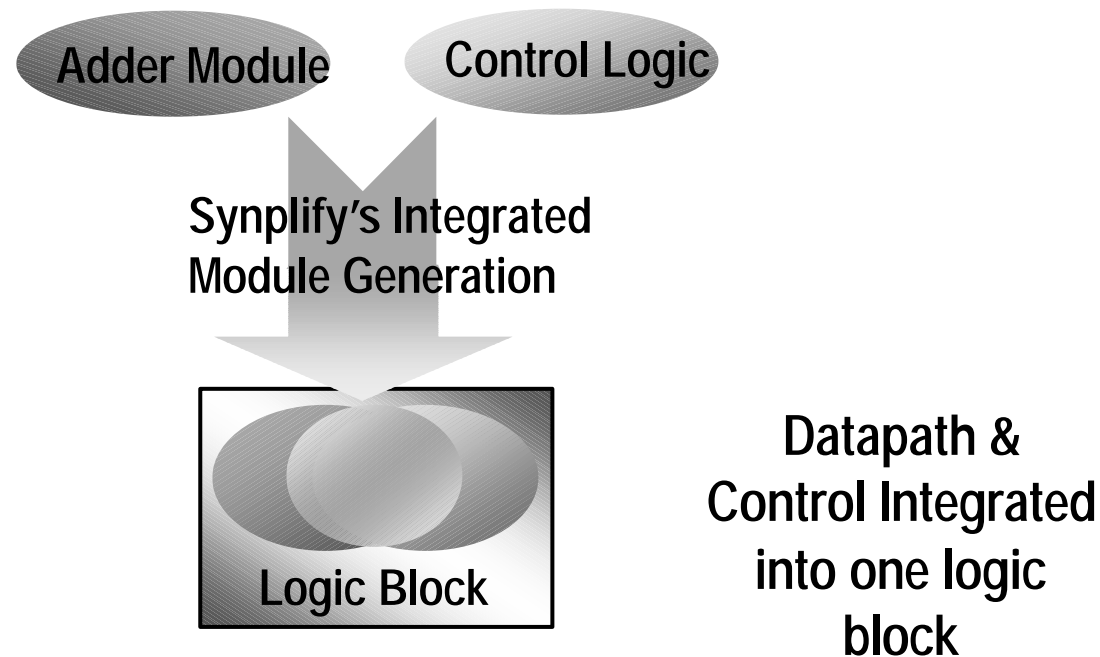
Black Box - Enabling Real IP's

- ▶ Growing Design Gap
- ▶ Timing Arcs Enable Timing Optimization
 - ▶ Are cores in critical path?
- ▶ Strategic Alliances

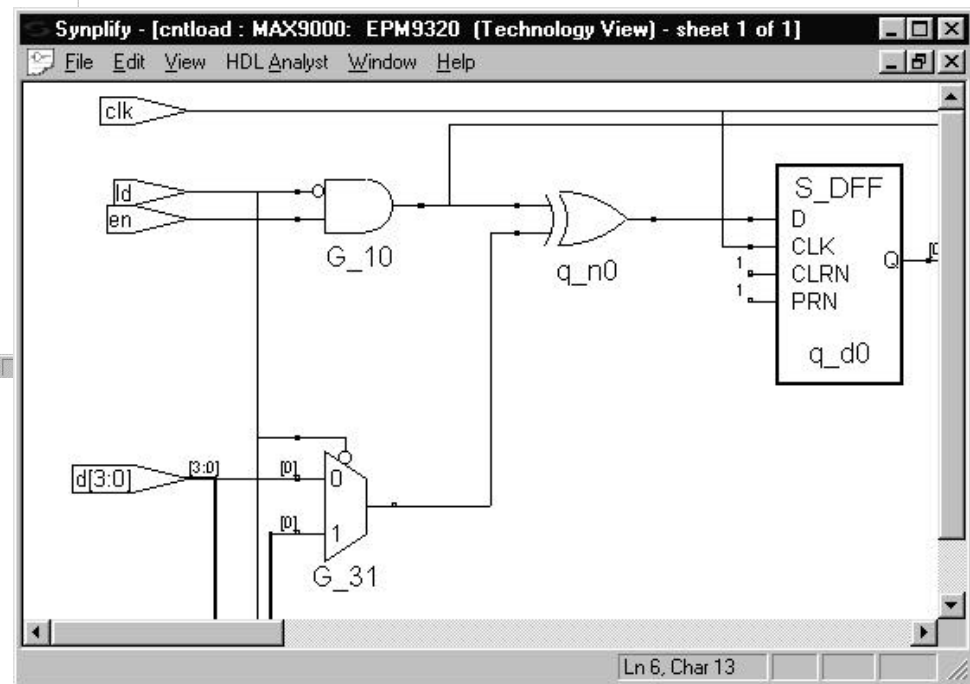
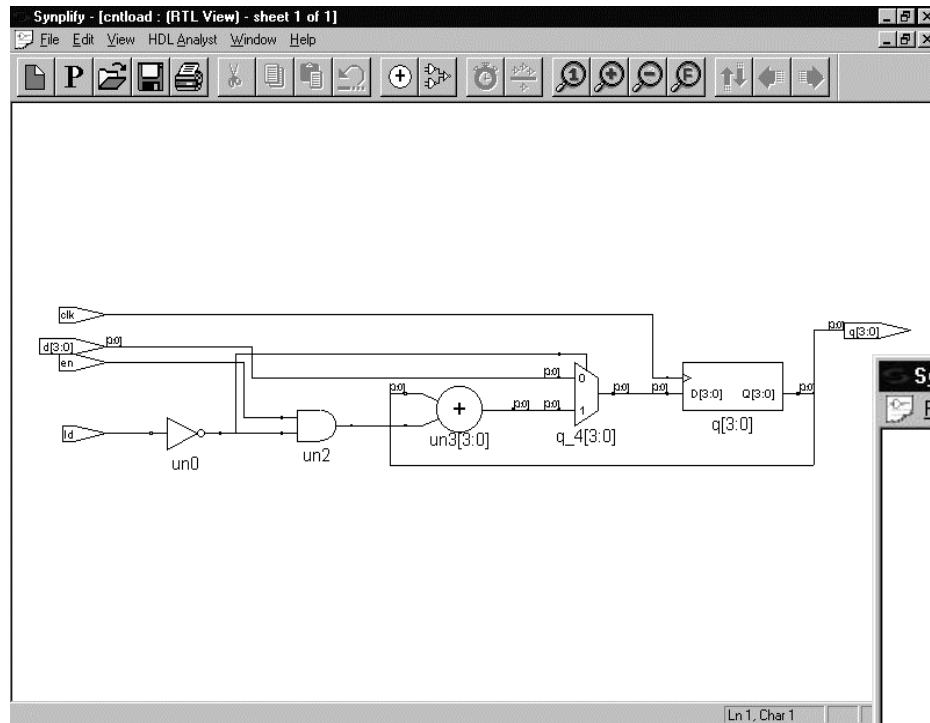


Integrated Module Generation

- ▶ Automatically combine modules with associated random/control logic
- ▶ Take maximum advantage of the resources available in the logic block of a device



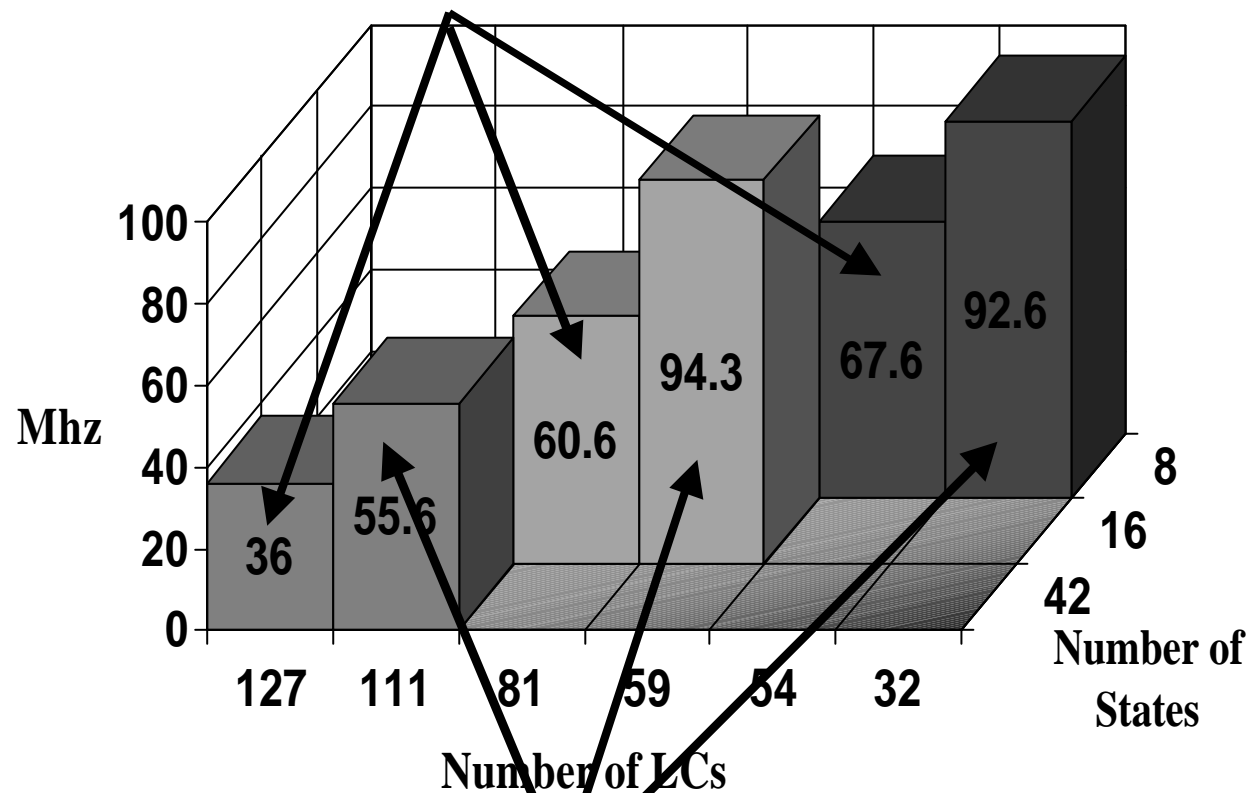
Module Generation Example



Synplify's Unique FSM Compiler

Automatically finds and re-encodes state machines

Without FSM Compiler



With FSM Compiler

FSM Compiler

- ▶ Symbolic FSM Compiler
 - ▶ Auto Inferencing, Encoding, & Optimization
- ▶ How to overwrite FSM compiler encoding
 - ▶ Use syn_encoding in VHDL and Verilog
 - ▶ Sequential
 - ▶ Gray
 - ▶ One Hot

FSM Encoding

VHDL Syntax

```
library synplify;  
use synplify.attributes.all;  
  
signal state : std_logic_vector(1 downto 0);  
signal next_state : std_logic_vector(1 downto 0);  
--attribute state_machine : boolean;  
--attribute syn_encoding : string;  
attribute state_machine of state : signal is true;  
attribute syn_encoding of state : signal is "sequential";
```

Verilog Syntax

```
reg [1:0] state /* synthesis state_machine syn_encoding="sequential" */;  
reg [1:0] next_state;  
reg data_out;
```



FSM VHDL Example

```
library ieee;
use ieee.std_logic_1164.all;

entity FSM1 is
    port(clk, rst, enable : in std_logic;
          data_in : in std_logic_vector(2 downto 0);
          data_out, state0, state1, state2 : out std_logic);
end FSM1;

architecture behave of FSM1 is
    type state_value is (SX, S0, S1, S2);
    signal state, next_state : state_value;
begin

    process (clk, rst) begin
        if (rst = '0') then
            state <= S0;
        elsif (rising_edge(clk)) then
            state <= next_state;
        end if;
    end process;

    process (state, enable, data_in) begin
        state0 <= '0';
        state1 <= '0';
        state2 <= '0';
        data_out <= '0';
        case state is
```

```
            when S0 =>
                if (enable = '1') then
                    state0 <= '1';
                    data_out <= data_in(0);
                    next_state <= S1;
                else
                    next_state <= S0;
                end if;
            when S1 =>
                if (enable = '1') then
                    state1 <= '1';
                    data_out <= data_in(1);
                    next_state <= S2;
                else
                    next_state <= S1;
                end if;
            when S2 =>
                if (enable = '1') then
                    state2 <= '1';
                    data_out <= data_in(2);
                    next_state <= S0;
                else
                    next_state <= S2;
                end if;
            when others =>
                next_state <= SX;
            end case;
        end process;
    end behave;
```



RAM Inference in Synplify

- ▶ Automatic RAM Inferencing
 - ▶ Synchronous RAM only
 - ▶ LPM RAM or Register implementations
- ▶ Advantage of Inferencing
 - ▶ Technology independent coding style
 - ▶ Timing/area Estimates on RAM blocks
- ▶ Coding style
 - ▶ A case structure, or
 - ▶ an assignment to a signal (reg in Verilog) that is an array of array
 - ▶ controlled by a clock edge and a write enable

Single Port RAM Example

```
module test_ram32x2 (clk, we, addr, data_in, data_out);
input clk, we;
input [1:0] data_in;
input [4:0] addr;
output [1:0] data_out;
reg [1:0] mem [31:0];
reg [4:0] addr_reg;
always @(posedge clk) begin
    if (we) mem[addr] = data_in;
    addr_reg = addr;
end
assign data_out = mem[addr_reg];
endmodule
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
entity ramtest is
port (
    q : out std_logic_vector(3 downto 0);
    d, addr : in std_logic_vector(3 downto 0)
    we,clk : in std_logic);
end ramtest;
architecture rtl of ramtest is
type mem_type is array (7 downto 0) of std_logic_vector (downto 0);
signal mem : mem_type;
signal addr_reg : std_logic_vector(3 downto 0);
begin
    q <= mem(conv_integer(addr_reg));
    process (clk, we, addr) begin
        if (rising_edge (clk)) then
            if (we = '1') then
                mem(conv_integer(addr)) <= d;
                addr_reg <= addr;
            end if;
        end if;
    end process;
end rtl;
```



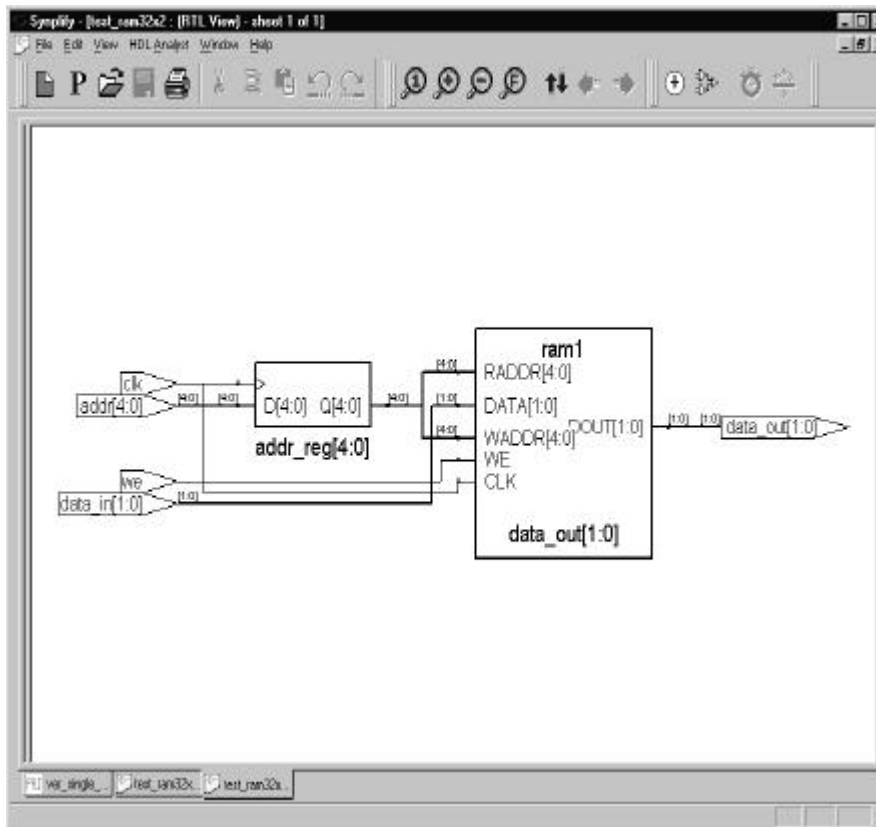
Synplicity

Simply Better Results™

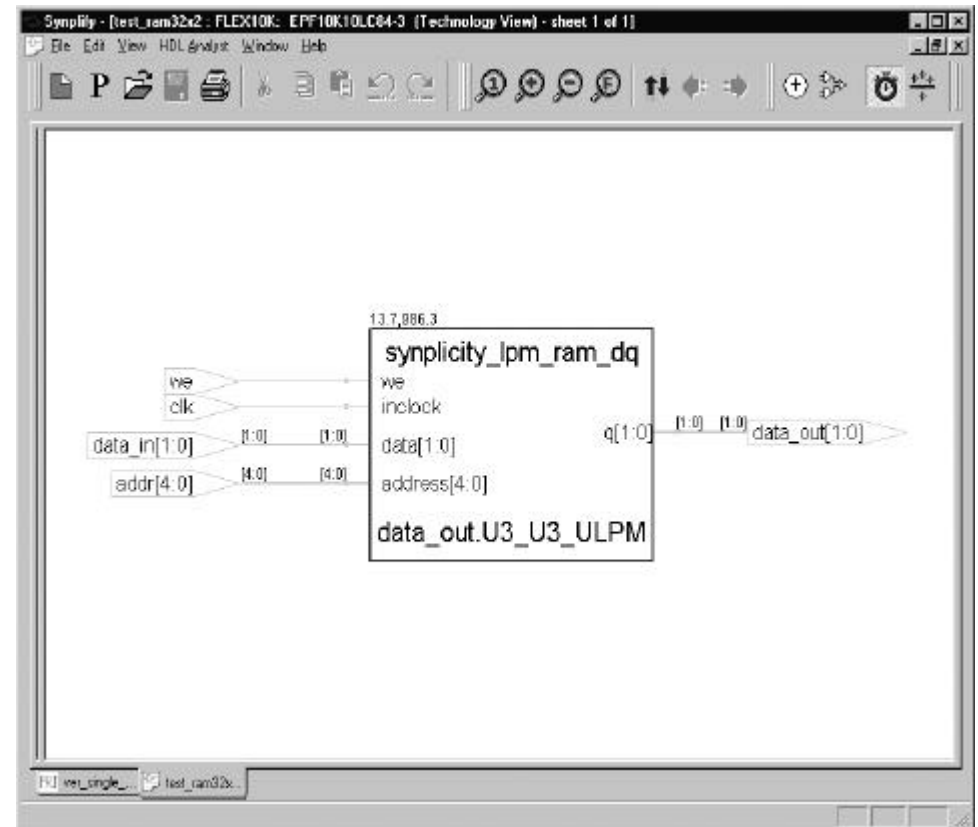
Single Port RAM Example

- ▶ HDL Analyst displays a RAM block in the RTL and Tech view, making the schematic view easy to read

RTL View



Tech View



RAM Implementation

- ▶ Map to RAM Block in CPLD/FPGA
- ▶ Map to standard logic and registers
 - ▶ set syn_ramstyle attribute to “registers.”
define_attribute {data_out[1:0]} syn_ramstyle registers

LPM Support

- ▶ Use generic map and port map statements
 - ▶ No more need to declare a component or define a list of attributes!

- ▶ Disadvantage of LPM
 - ▶ Limits optimization
 - ▶ Timing/area unknown
 - ▶ based on PLD software implementation
 - ▶ Use only when needed
 - ▶ ALU functions operate better in synthesis

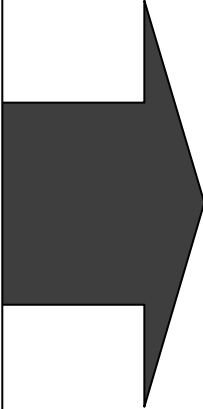
Example: LPM Instantiation Using Generics

```
library ieee;
use ieee.std_logic_1164.all;
entity lpm_mult_old is
    port ( DO          : out std_logic_vector (7 downto 0);
          A, B, SUM    : in  std_logic_vector (3 downto 0);
          CLK , RST    : in  std_logic);
end lpm_mult_old;

architecture struct of lpm_mult_old is
-- Declare the component here
component lpm_mult
    port (dataa, datab, sum : in  std_logic_vector(3 downto 0);
          clock, aclr       : in  std_logic;
          result            : out std_logic_vector(7 downto 0));
end component;

attribute black_box: boolean;
attribute LPM_WIDTHA : integer;
attribute LPM_WIDTHHB : integer;
attribute LPM_WIDTHHP : integer;
attribute LPM_WIDTHHS : integer;
attribute LPM_PIPELINE : integer;
attribute LPM_TYPE     : string;
-- Assign the appropriate attribute values here
attribute black_box      of lpm_mult : component is true;
attribute LPM_WIDTHA     of lpm_mult : component is 4;
attribute LPM_WIDTHHB    of lpm_mult : component is 4;
attribute LPM_WIDTHHP    of lpm_mult : component is 8;
attribute LPM_WIDTHHS    of lpm_mult : component is 4;
attribute LPM_PIPELINE   of lpm_mult : component is 1;
attribute LPM_TYPE       of lpm_mult : component is "LPM_MULT";

begin
-- Instantiate the LPM component here
u1: lpm_mult port map (dataa => A, datab => B, sum => SUM,
                      clock => CLK, aclr => RST, result => DO);
end struct;
```



```
library ieee;
use ieee.std_logic_1164.all;
library lpm;
use lpm.lpm_components.all;
entity lpm_mult_new is
    port ( DO          : out std_logic_vector (7 downto 0);
          A, B, SUM    : in  std_logic_vector (3 downto 0);
          CLK , RST    : in  std_logic);
end lpm_mult_new;

architecture struct of lpm_mult_new is
begin
I0 : lpm_mult generic map (LPM_WIDTHA => 4, LPM_WIDTHHB => 4,
                          LPM_WIDTHHP => 8, LPM_WIDTHHS => 4, LPM_PIPELINE => 1)
    port map (dataa => A, datab => B, sum => SUM,
              clock => CLK, aclr => RST, result => DO);
end struct;
```

Agenda

► Product Overview

► Synplify

► Synplify Pro



► Amplify

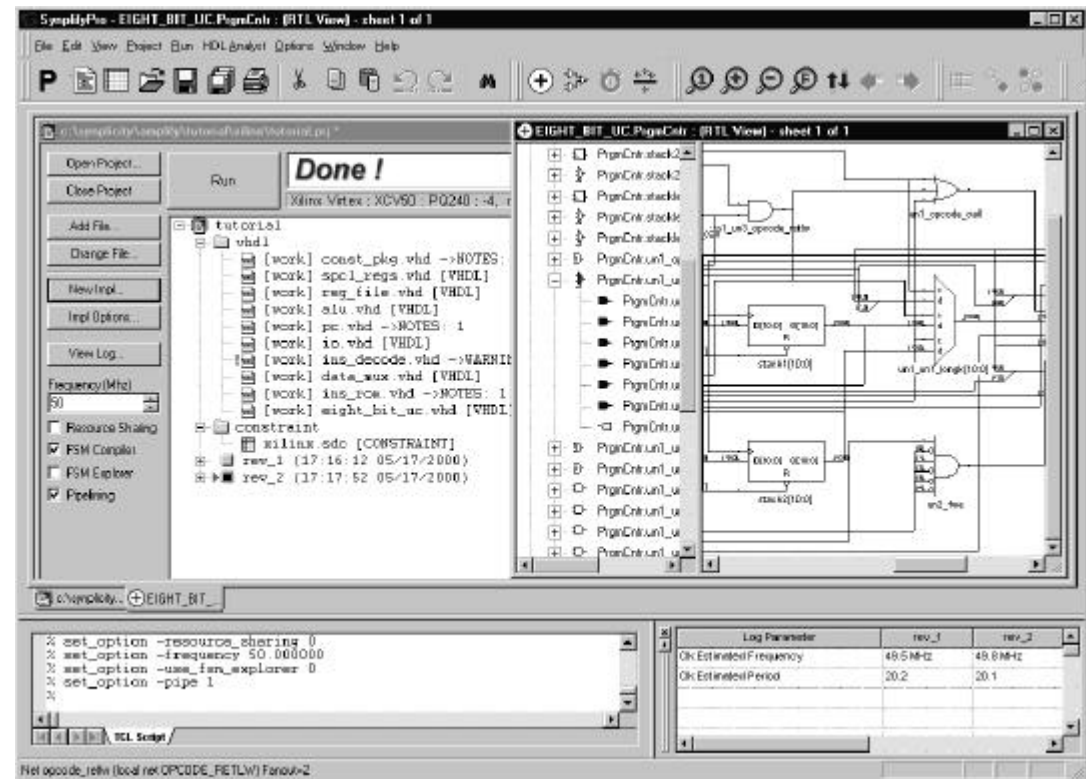
Challenges for Multi-Million Gate FPGAs

- ▶ **Larger Design Teams**
- ▶ **Advanced Project Management**
- ▶ **Incremental Design Techniques**
- ▶ **Integration of Intellectual Property (IP)**
- ▶ **Design Reuse**
- ▶ **Physical Interconnect Effects on Timing Convergence**

Introducing Synplify® Pro

All of Synplify *Plus*...

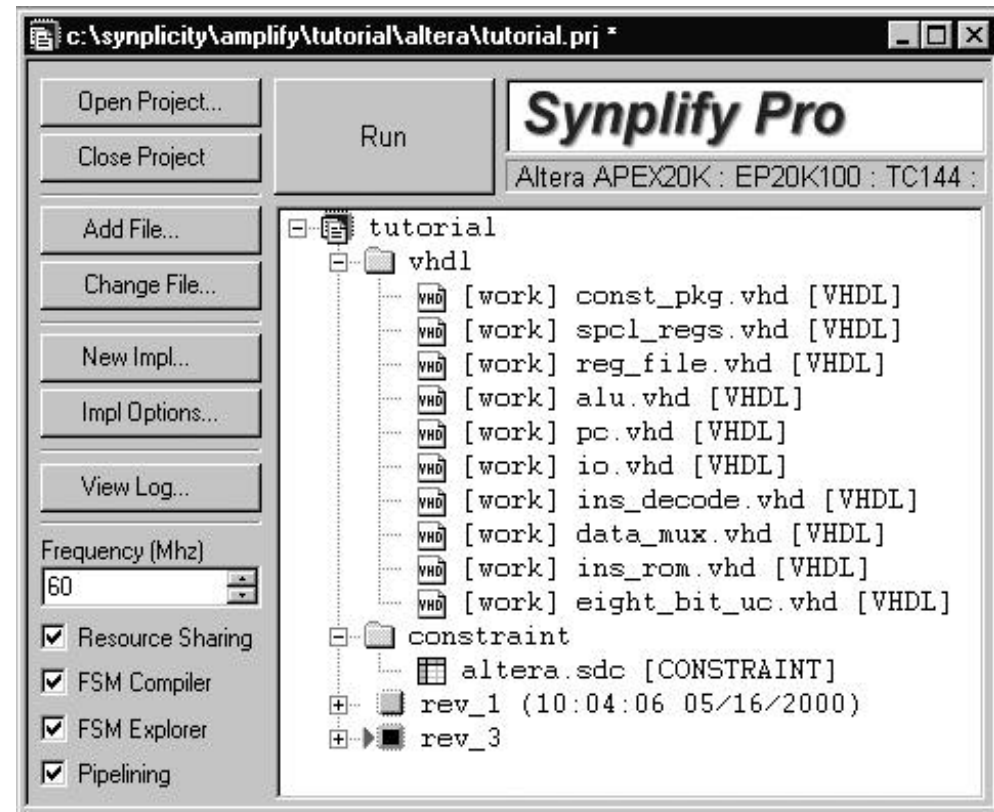
- ▶ HDL Analyst is standard
- ▶ Advanced Project Management
- ▶ Command Line Interface
- ▶ FSM Explorer and State Diagram Viewer
- ▶ Register Balancing for Pipelined Multipliers and ROMs
- ▶ Probe Point Extraction for Test and Debugging
- ▶ Cross-probing to error & any text files
- ▶ Amplify Physical Optimizer Option



*Best in class FPGA
Synthesis*

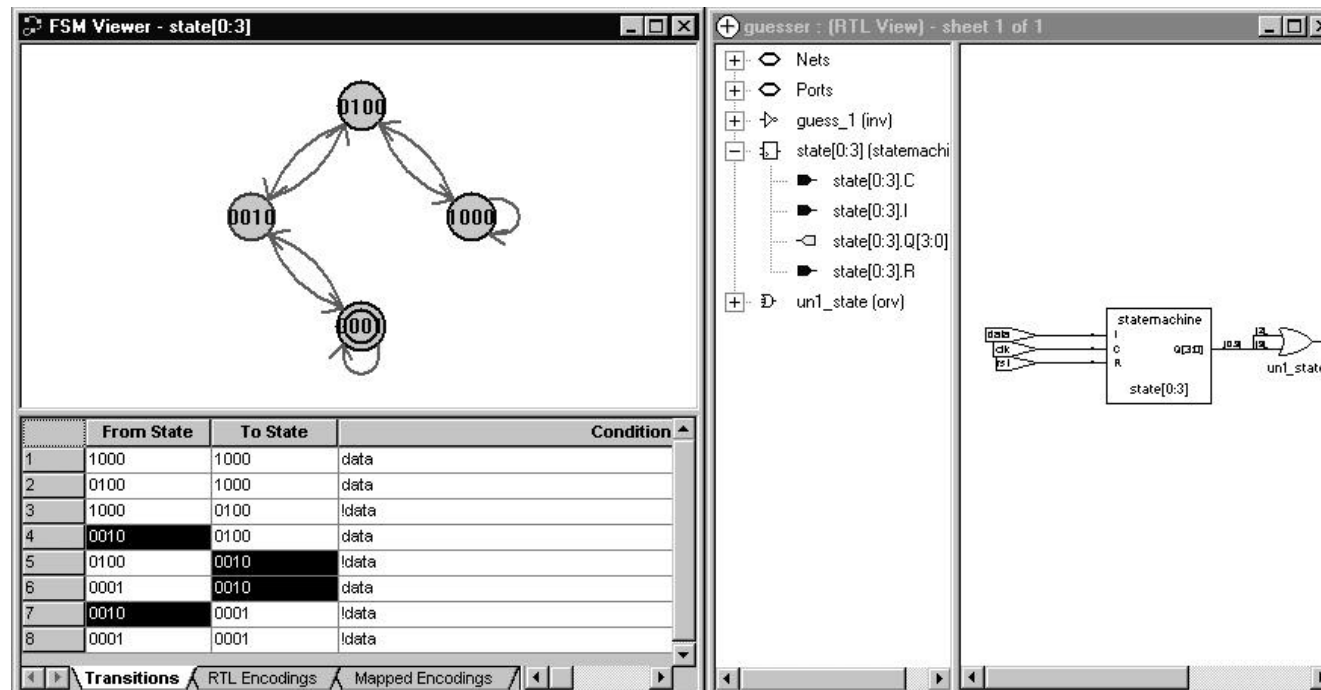
Powerful New User Interface

- ▶ **Advanced Project Management**
 - ▶ Multiple Implementations
 - ▶ Project browser
- ▶ **Command Line Interface**
 - ▶ Expanded TCL command set
- ▶ **Batch Mode**
 - ▶ Floating Licenses
- ▶ **Watch Window**
 - ▶ View results of multiple implementations at once



Log Parameter	rev_1	rev_2
Clk:Estimated Frequency	23.5 MHz	36.9 MHz
Logic resources	999 LCs of 576 (173%)	1001 LCs of 4992 (20%)

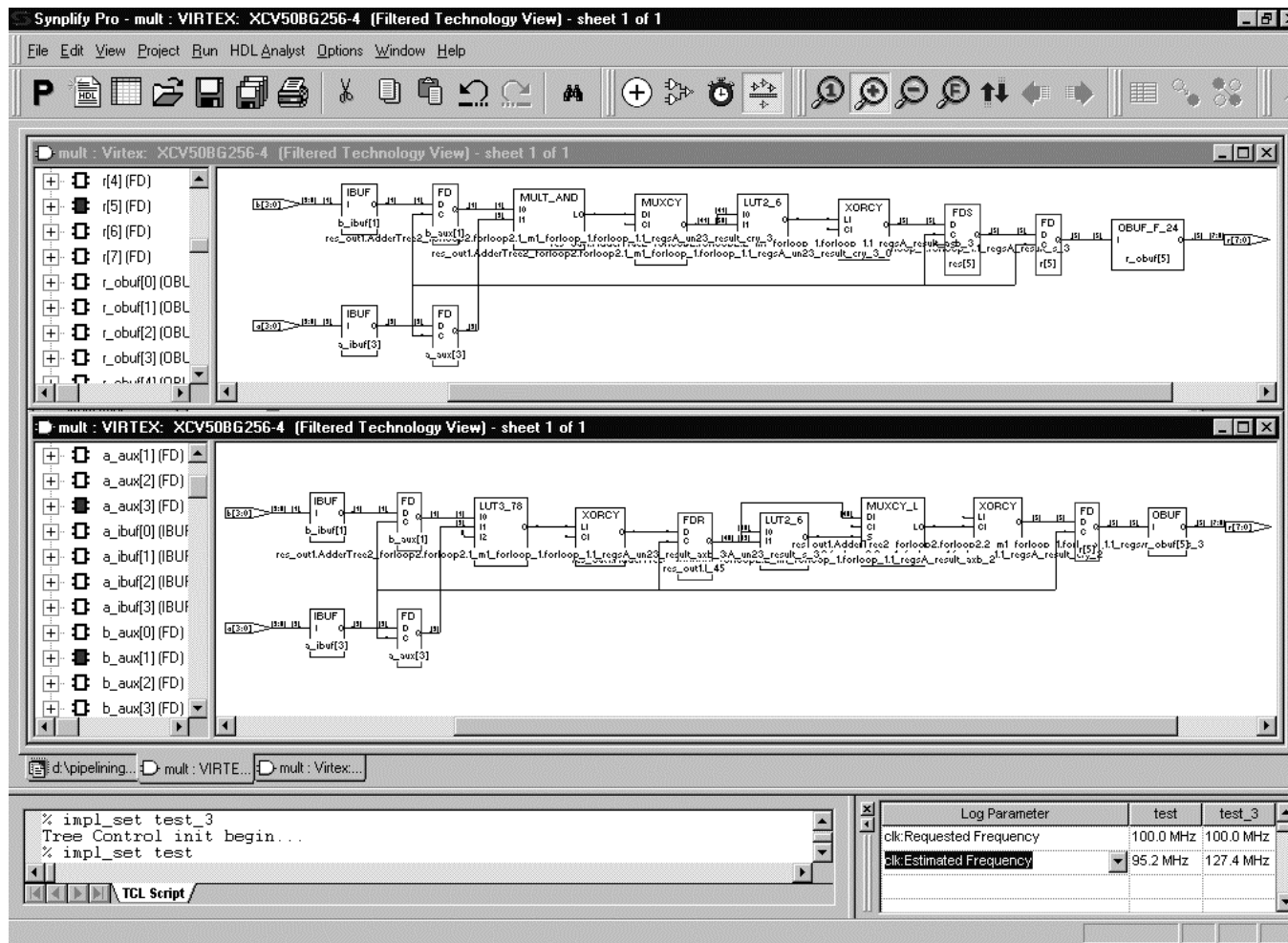
Unique FSM Explorer & Viewer



- ▶ Enhances Synplify's FSM Compiler
- ▶ FSM Explorer automatically selects the best encoding style
 - ▶ Quality of Results benefit
 - ▶ Ease of use benefit
- ▶ FSM Viewer graphically displays FSM bubble diagram
 - ▶ Cross-probing to transitions and states

Register Balancing for Pipelined Multipliers & ROMs

- Automatic register balancing produces pipelined operators for faster circuit performance



Agenda

► Product Overview

► Synplify

► Synplify Pro

► Amplify



Amplify™ Physical Optimizer™

When you need the highest performance possible

► Performance

- Achieves up to 40% better circuit performance

► Productivity

- Saves weeks of time in achieving performance goals
- Maintains Synplify's ease of-use philosophy

► Supports Team Design

- Manages physical hierarchy
- Optimizes across boundaries

