

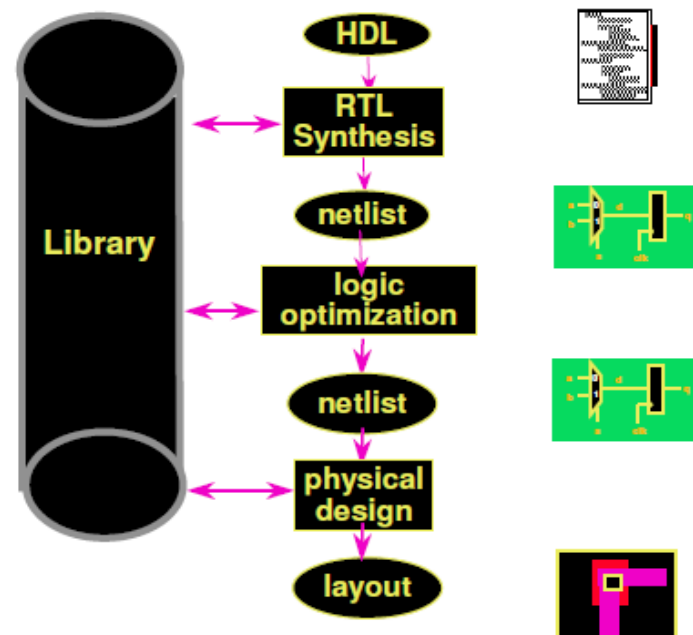
Basics of RTL Design and Synthesis

•Rajendra Pratap, PhD

•Fellow & Head NOIDA Design Center

•eInfochips Ltd, fully owned subsidiary of Arrow Electronics Inc

RTL Synthesis Flow



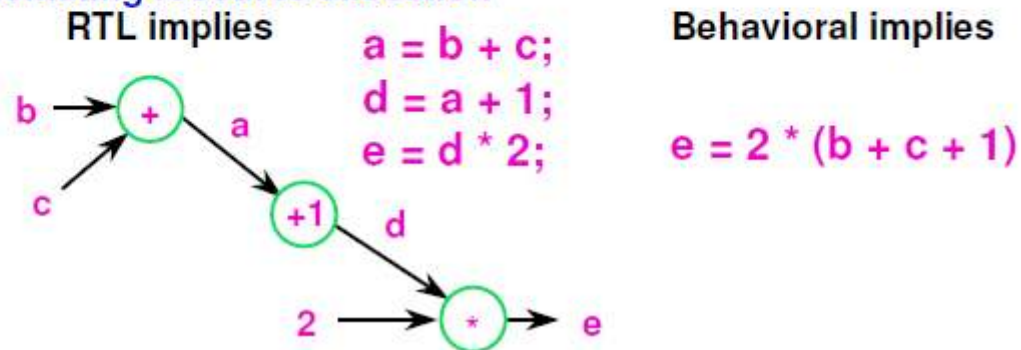
RTL level

An RTL description is always implicitly structural -

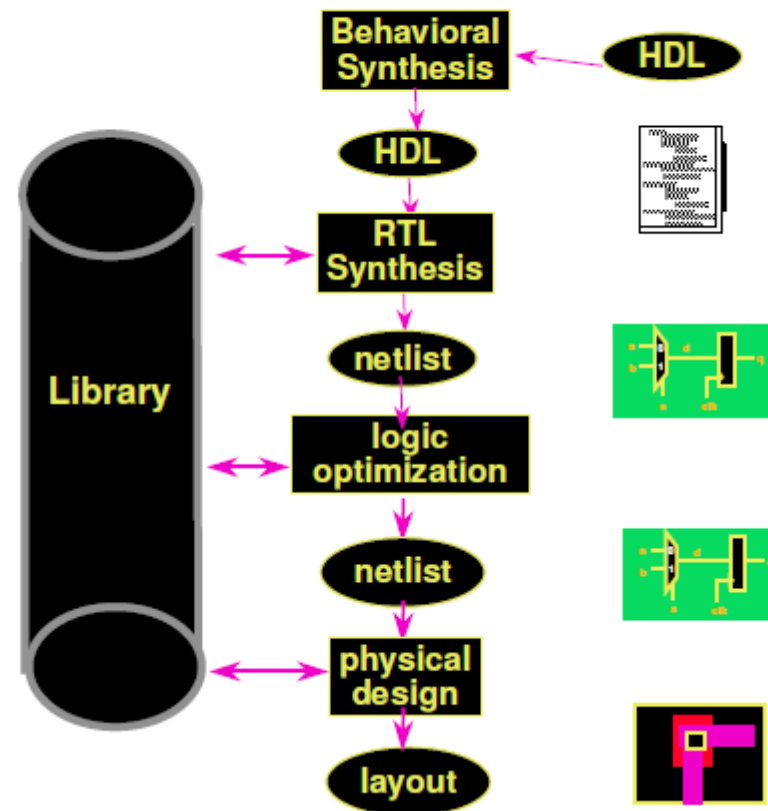
- the registers and their interconnectivity are defined
- Thus the clock-to-clock behavior is defined
- Only the control logic for the transfers is synthesized.

This approach can be enhanced:

- Register inferencing
- Automating resource allocation



Behavioral Synthesis Flow



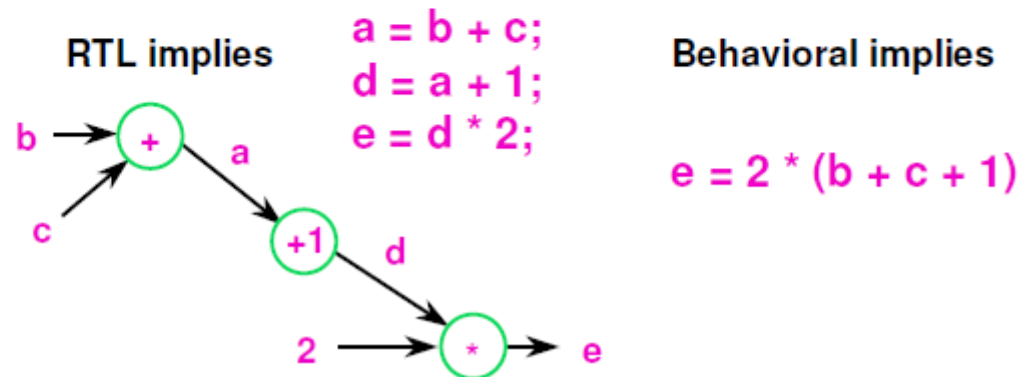
Behavioral level

A behavioral description is always functional

- Temporal relationships are only expressed as precedences
- An entire micro-architecture is synthesized from the behavioral description

The two key elements of behavioral transformation:

- Automating resource allocation (could be RTL also)
- Scheduling



RTL Synthesis

Two Steps:

1. Translation from RTL description into gates
2. Optimization of logic

Ideally, all of the intelligence should be in the optimization step

Unfortunately, starting point for optimization affects results

⇒ Need to write a “good” RTL description

Writing RTL Descriptions

Current scenario: Designer iterates over RTL descriptions using simulation and synthesis tools to verify and evaluate design

PROBLEM: Equivalent RTL descriptions may result in dramatically different logic circuits in terms of area/performance **EVEN AFTER OPTIMIZATION.**

Behavioral Synthesis Features

Scheduling of operations

- Operation Chaining and Multi-cycling
- Different Scheduling Modes, and constraints

Memory Inferencing

- Array's can be mapped directly to RAM
- Automatic control/data/address generation
- RAM tradeoffs (1 port vs. 2 port) are easy

Allocation of resources

- Building your own design components

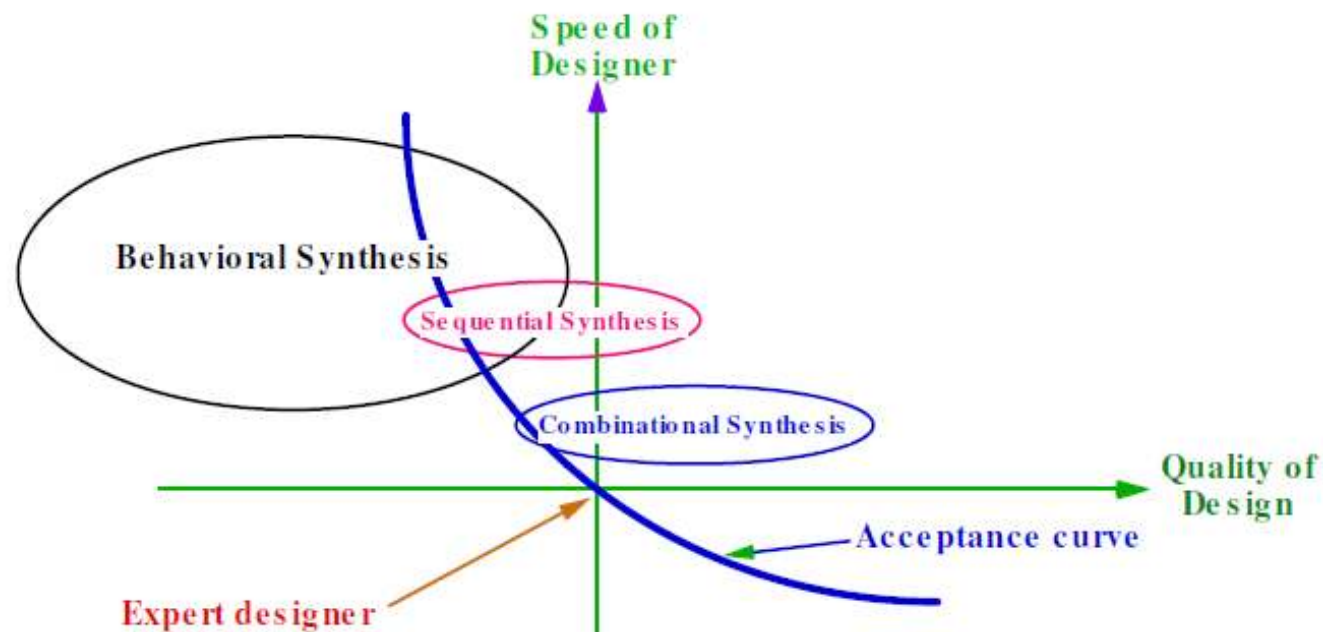
Using Pipelined Components

- Multipliers, Adders, RAMs, others

Pipelining Loop with various Throughput & Latency

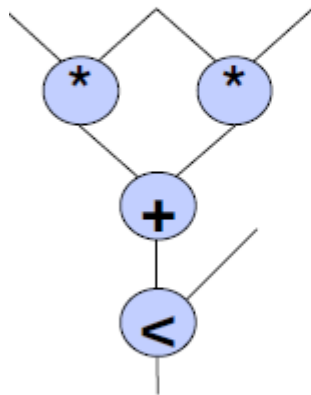
Automatic generation of FSM Controller and integration of Datapath

Level of Acceptance of Synthesis Techniques

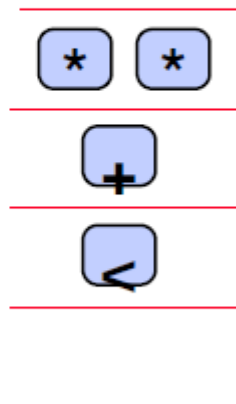


Scheduling Tradeoffs I

Design Tradeoff with Scheduling

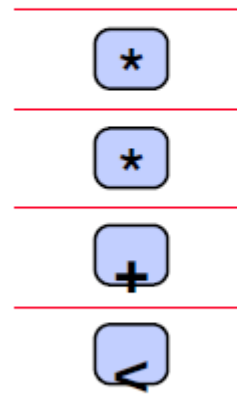


Design #1



clock 50ns
2 fast multipliers
3 cycles

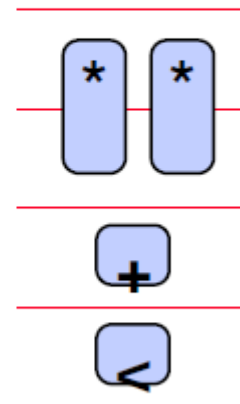
Design #2



clock 50ns
1 fast multipliers
4 cycles

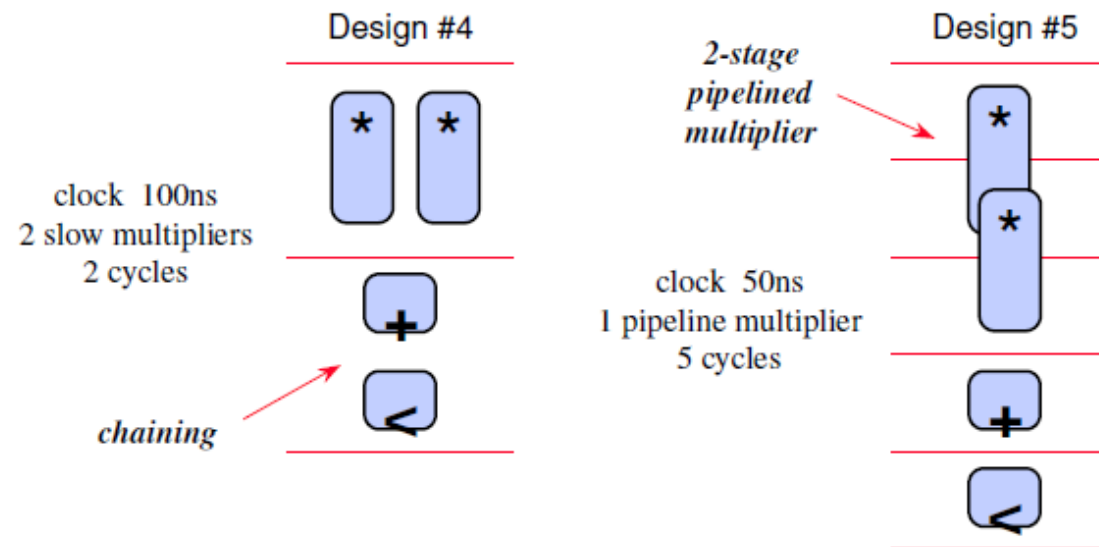
multi-cycling

Design #3



clock 50ns
2 slow multipliers
4 cycles

Scheduling Tradeoffs II



Tradeoff:

Clock speed vs Latency vs Resources

Basics of Synthesis

What is Synthesis

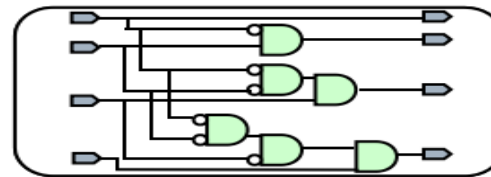
- Synthesis = translation + optimization + mapping

```
if(high_bits == 2'b10)begin  
    residue = state_table[i];  
end  
else begin  
    residue = 16'h0000;  
end
```

**HDL Source
(RTL)**

Translate (HDL Compiler)

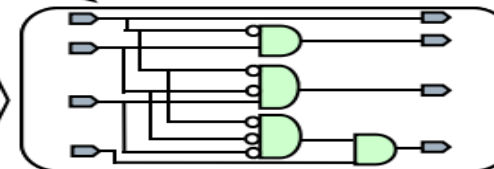
No Timing Info. →



**Generic Boolean
(GTEC)**

**Optimize + Mapping
(Design Compiler)**

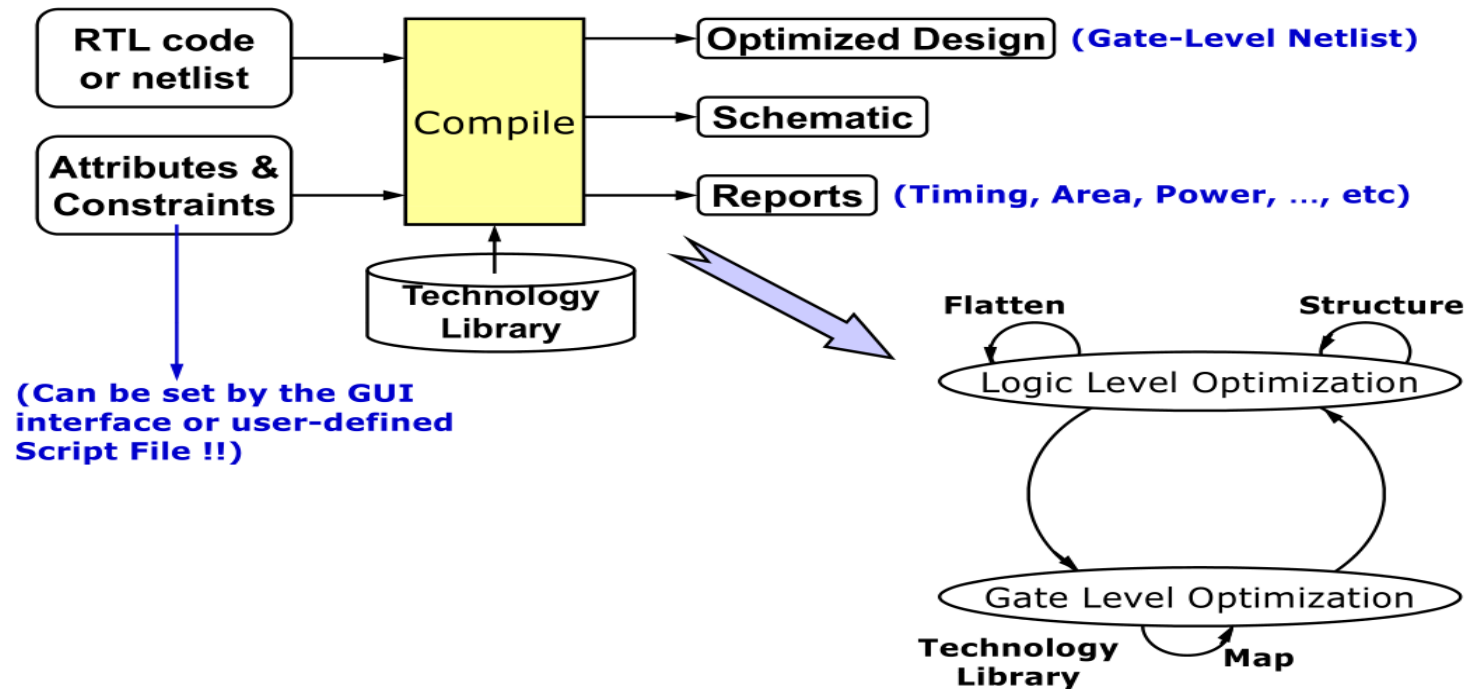
Timing Info. →



Target Technology

**The synthesis is constraint driven
and technology independent !!**

Compile



Stages of Synthesis:

- Identify the Time based State Machine
- Infer the logic and state elements.
- Perform technology independent optimization(logic simplification and state assignment.
- Map elements to the target technology
- Perform technology dependent optimizations (multi level logic optimization , choose gate

Simulation v/s Synthesis

In HDL like Verilog every thing that can be simulated cannot be synthesized. It is not easy to specify synthesizable subset of an HDL.

Note before design and synthesis:

Your RTL design

- Functional verification by some high-level language
- Also, the code coverage of your test benches should be verified
- Coding style checking (i.e. n-Lint)
- Good coding style will reduce most hazards while synthesis
- Better optimization process results in better circuit performance
- Easy debugging after synthesis

Constraints

- The area and timing of your circuit are mainly determined by your circuit/design architecture and coding style.
- There is always a trade-off between the design timing and area.
- In fact, a super tight timing constraint may work while synthesis, but failed in the Place & Route (P&R) procedure.

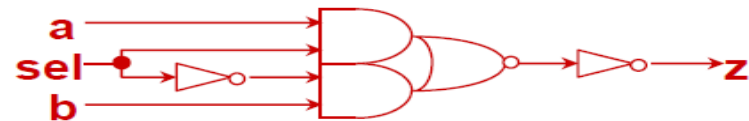
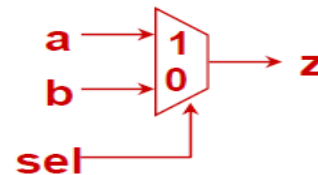
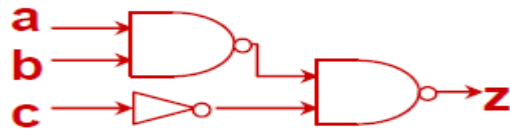
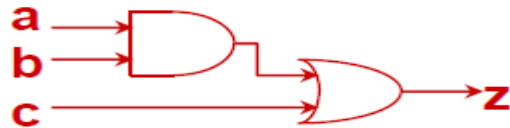
Logic synthesis

Example 1: Represents how can the logic in RTL can be synthesized.

//data flow

assign z = (a & b) | c ;

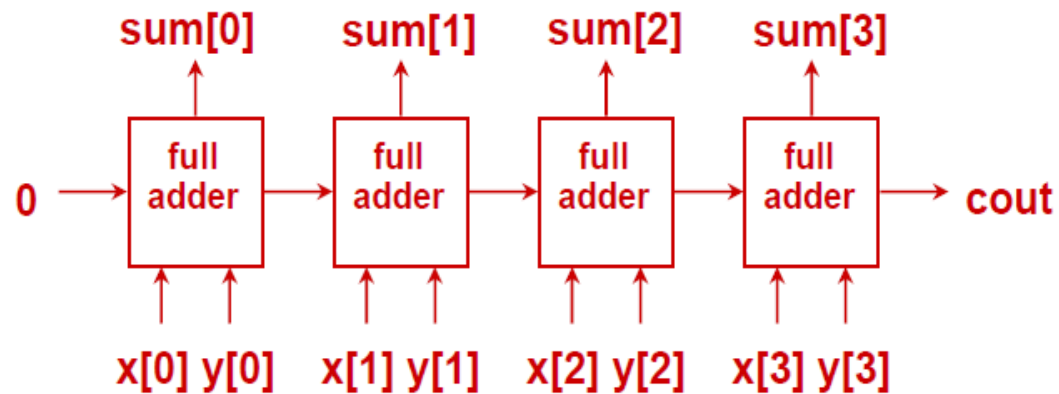
assign z = sel ? a : b ;



Logic synthesis

Example 2: Represents how can the logic in RTL can be synthesized.

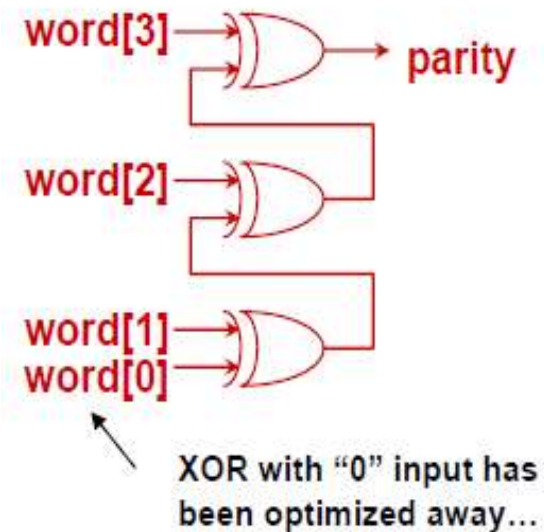
```
wire [3:0] x,y,sum;  
wire cout;  
assign {cout,sum} = x + y;
```



Logic synthesis

Example 3: Represents how can the logic in RTL can be synthesized.

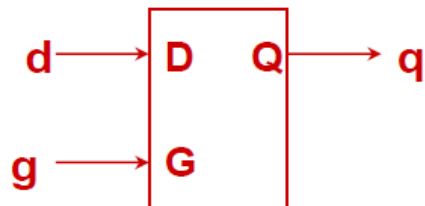
```
module parity (in,p);  
  parameter WIDTH = 2; // default width is 2  
  input [WIDTH-1 : 0] in;  
  output p;  
  //simple approach : assign p = ^in;  
  //here is another , more general approach  
  reg p;  
  always @(in) begin: loop  
    integer i;  
    reg parity = 0;  
    for (i = 0; i < WIDTH; i = i + 1)  
      parity = parity ^ in[ i ];  
    p <= parity;  
  end  
endmodule  
  
wire [3:0] word;  
wire parity;  
parity #(4) ecc (word,parity) //specify WIDTH = 4
```



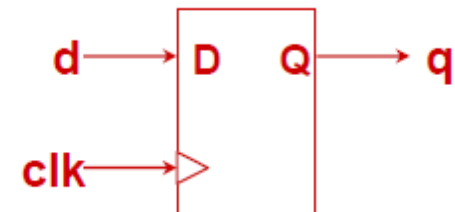
Logic synthesis


Example 4: Represents how can the sequential logic in RTL can be synthesized.

```
reg q;  
// D-latch  
always @(g or d)  
  begin if (g) q <= d;  
end
```



```
reg q;  
// D-register  
always @(posedge clk)  
  begin q <= d;  
end
```



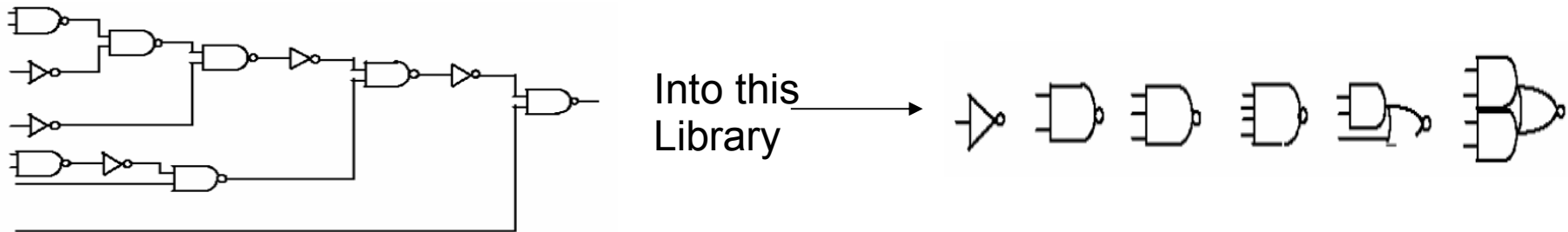
- If q were simply a combinational function of d, the synthesizer could just create the appropriate combinational logic.
- But since there are times when the always block executes but q isn't assigned (e.g., when $g=0$), the synthesizer has to arrange to remember the value of "old" value of q even if d is changing, 
- And it will infer the need for a storage element,(latch, register,...).
- Sometimes this inference happens even when you don't mean it to.
- You have to be careful to always ensure an assignment happens each time through the block if you don't want storage element to appear in your design.

Mapping to target technology






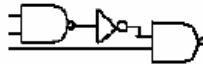

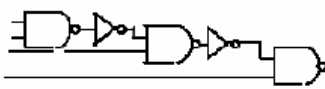
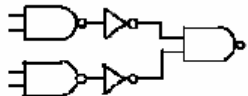

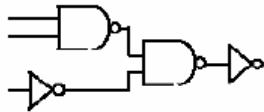

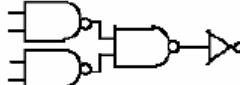
Once we have minimized the logic equations, the next step is mapping each equation to the gates in our target gate library.

Mapping Example:

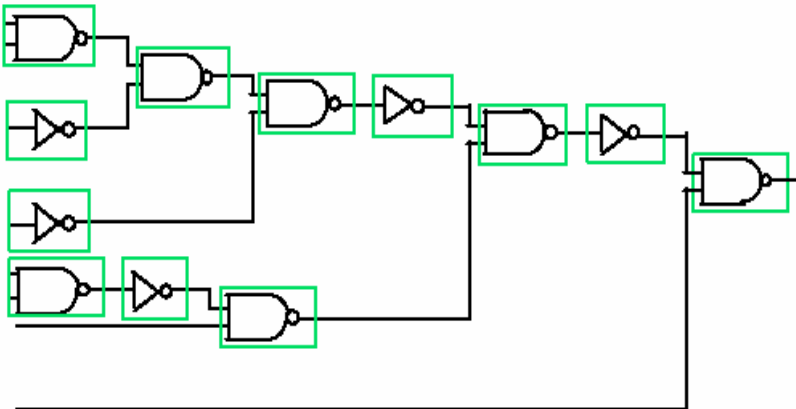
Problem statement: find an “optimal” mapping of this circuit.



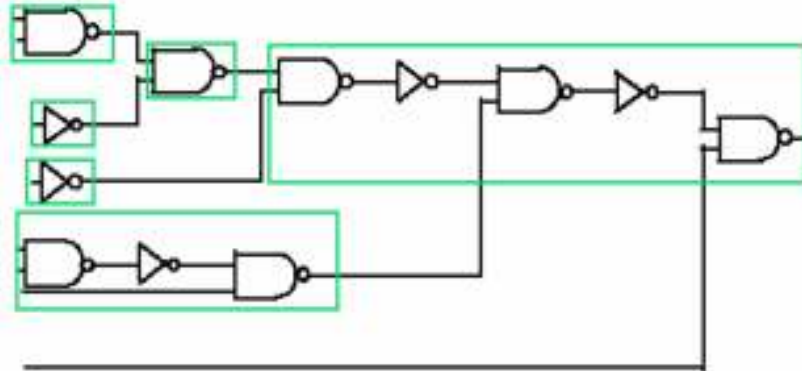
Primitive library gates

	Element/Area Cost		Tree Representation (normal form)	
INVERTER	2			
NAND2	3			
NAND3	4			8
NAND4	5			13
				13
AOI21	4			10
AOI22	5			11

Possible covers



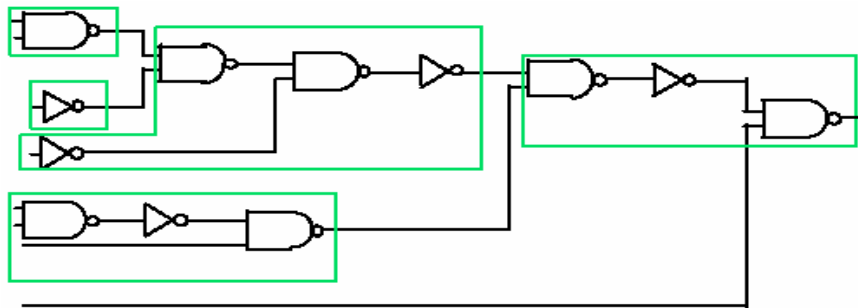
7 NAND2(3) = 21
5 INV (2) = 10
Area cost 31



2 INV =4
2 Nand2 =6
1Nand3 =4
1Nand4 =5
Area cost 19

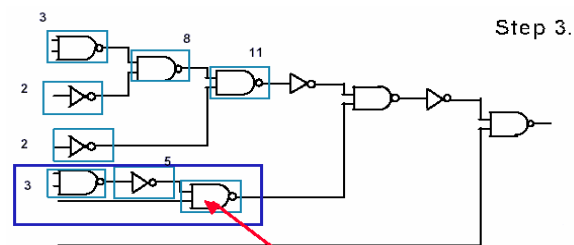
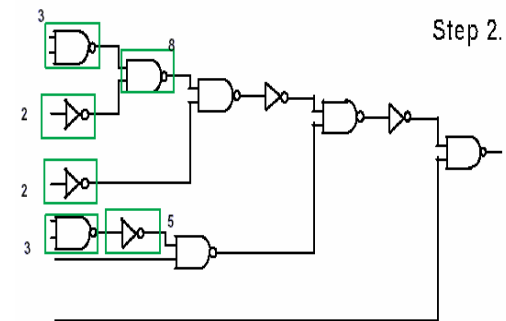
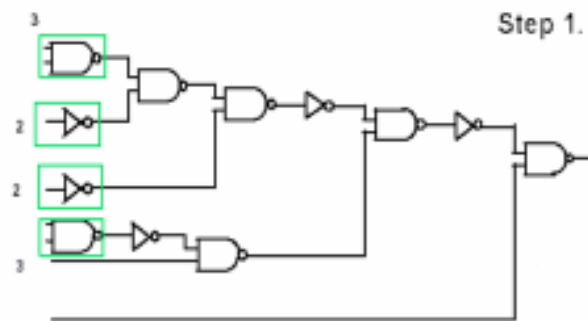
Possible covers

The above example seems promising but is there a systematic and efficient way to arrive at optimal answer?



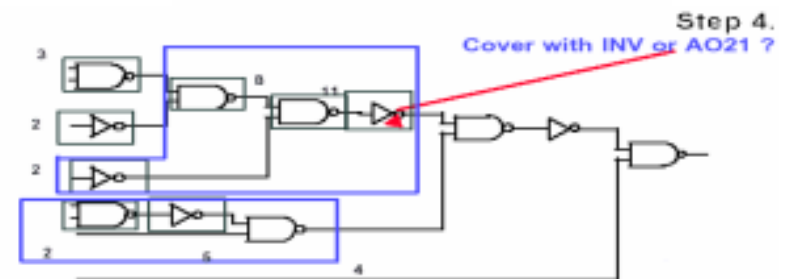
1 INV	=2
1 NAND2	=3
2 NAND3	=8
1AOI21	=4
Area Cost	17

Optimal tree covering example



Cover with ND2 or ND3 ?

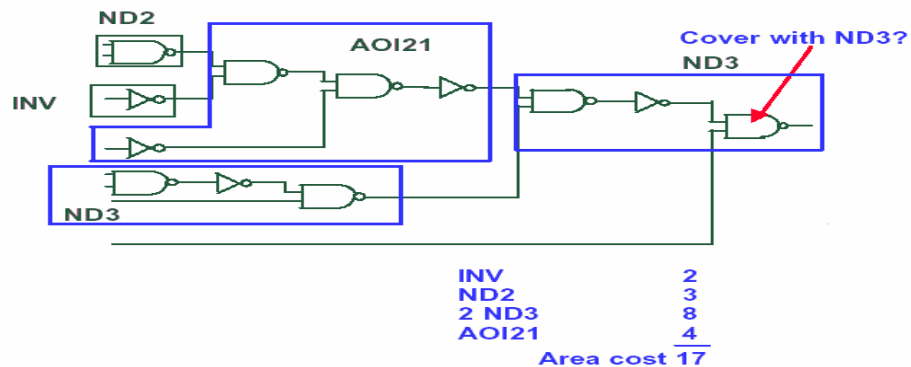
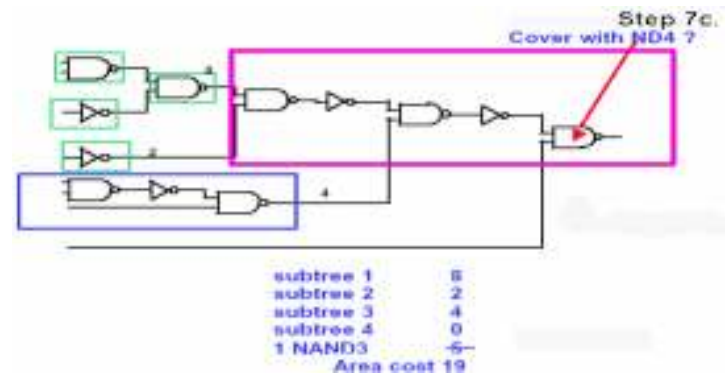
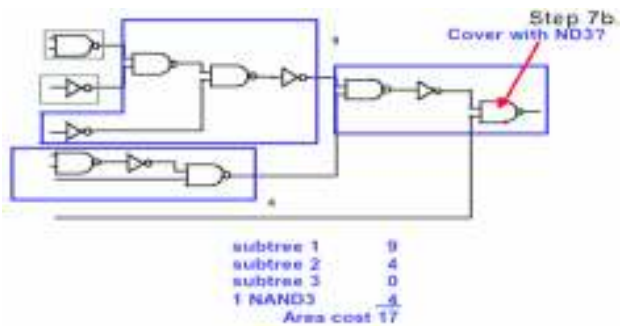
1 NAND2	3	1 NAND3	= 4
+ subtree	5		
Area cost 8			



1 Inverter	2
+ subtree	11
Area cost 13	

1 AO21	4
+ subtree 1	3
+ subtree 2	2
Area cost 9	

Optimal tree covering example

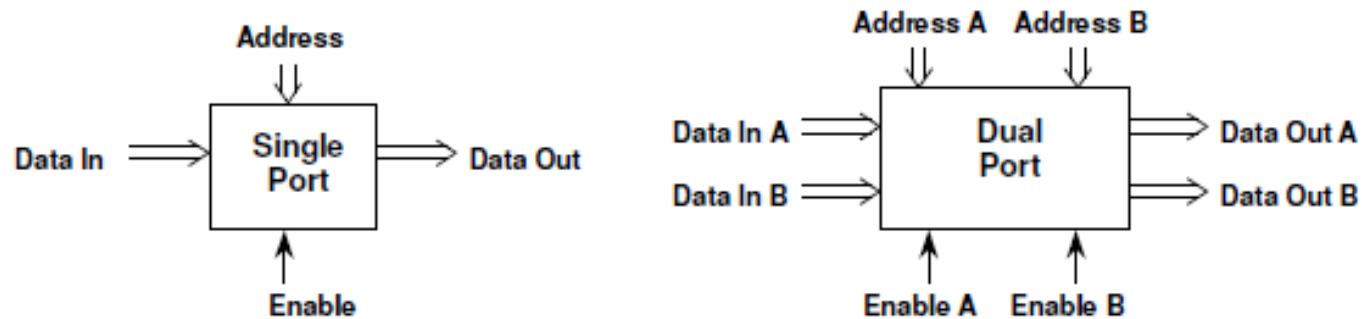


Refinements: timing optimization and area optimization.

Basic Memory Types

Memory Types:

- Synchronous: Inputs & outputs occur at clock edge
- Asynchronous: Unclocked signals
- Single Port: Either read or write to one address location
- Dual/Multi Port: Two fully independent read/write ports



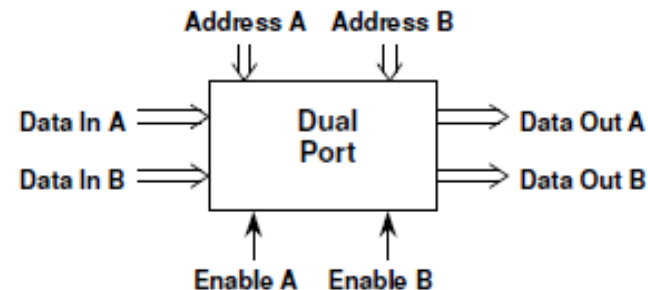
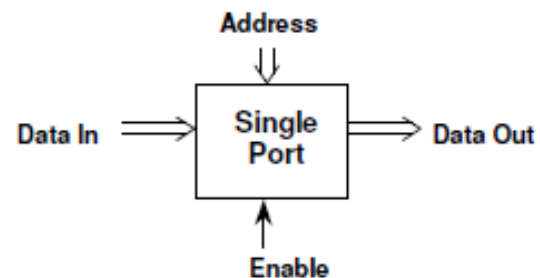
Memory Mapping Example

```

/*****  RAM Declaration Section  *****/
reg [7:0] p[0:63], q[0:63];

/* synopsys resource MY_RAM: variables = "p",
    map_to_module = "DW03_ram1_s_d"; */
/* synopsys resource MY_RAM: variables = "q",
    map_to_module = "DW03_ram1_s_d"; */
. . . . .
begin : counting
  forever begin
    /*****  RAM Read and Write Example *****/
    incm_p : for (i=0; i<63; i=i+1) begin
      q[i] = p[i] + 1;    /* This will be unrolled */
    end
  end
end // counting

```



Technology-dependent optimizations

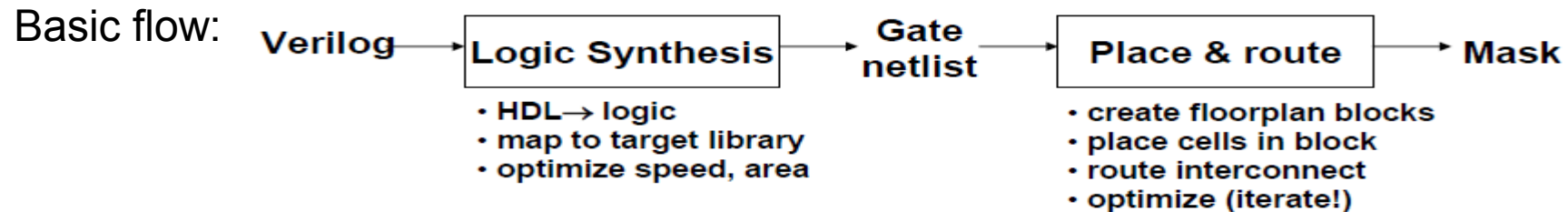
Additional library components: more complex cells may be slower but will reduce area for logic off the critical path.

Load buffering: adding buffers/inverters to improve load-induced delays along the critical path.

Resizing: Resize transistors in gates along critical path

Retiming: change placement of latches/registers to minimize overall cycle time

Increase routability over/through cells: reduce routing congestion.



DC Flow For Synthesis

<.synopsys_dc.setup> File

link_library: the library used for interpreting input description
Any cells instantiated in your HDL code.
Wire load or operating condition modules used during synthesis.

target_library: the ASIC technology which the design is mapped

symbol_library: used for schematic generation

search_path: the path for unsolved reference library

synthetic_path: designware library

Settings for Using Memory

We need the db for the memories which are to be used.

Modify `<.synopsys_dc.setup>` File

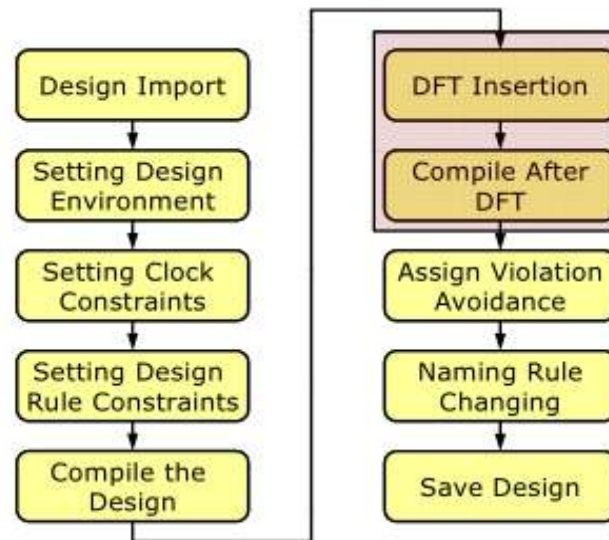
Set `link_library`: “the .db of the memory”

Set `target_library`: “the db of the memory”

Add a “search path” to this file

Before the synthesis, the memory HDL model should be blocked in your netlist.

Synthesis Flow



Read File

Read netlist or other description into Design Compiler
File read

Supported formats

Verilog :. v

VHDL: .vhd

System Verilog: .sv

EDIF

Synopsys internal formats:

DB (binary): .db

Enhance db file : .ddc

Command:

`read_file -format verilog file_name; read_file -format ddc file_name`

Read File

Running the read file Command

The `read_file` command analyzes the design and translate it into a technology-independent (GTECH) design in a single step.

Running the analyse and elaborate Commands

The `analyse` command checks the design and reports error. The `elaborate` command translate the design into a technology-independent design (GTECH) from the intermediate files produced during analysis.

Running the read_verilog or read_vhdl Command

The command checks the code for correct syntax and build a generic technology (GTECH) netlist that Design Compiler uses to optimize the design. You can use the `read_verilog` or `read_vhdl` commands to both functions automatically.

Constraint types

When Design Compiler optimizes your design, it uses two types of constraint:

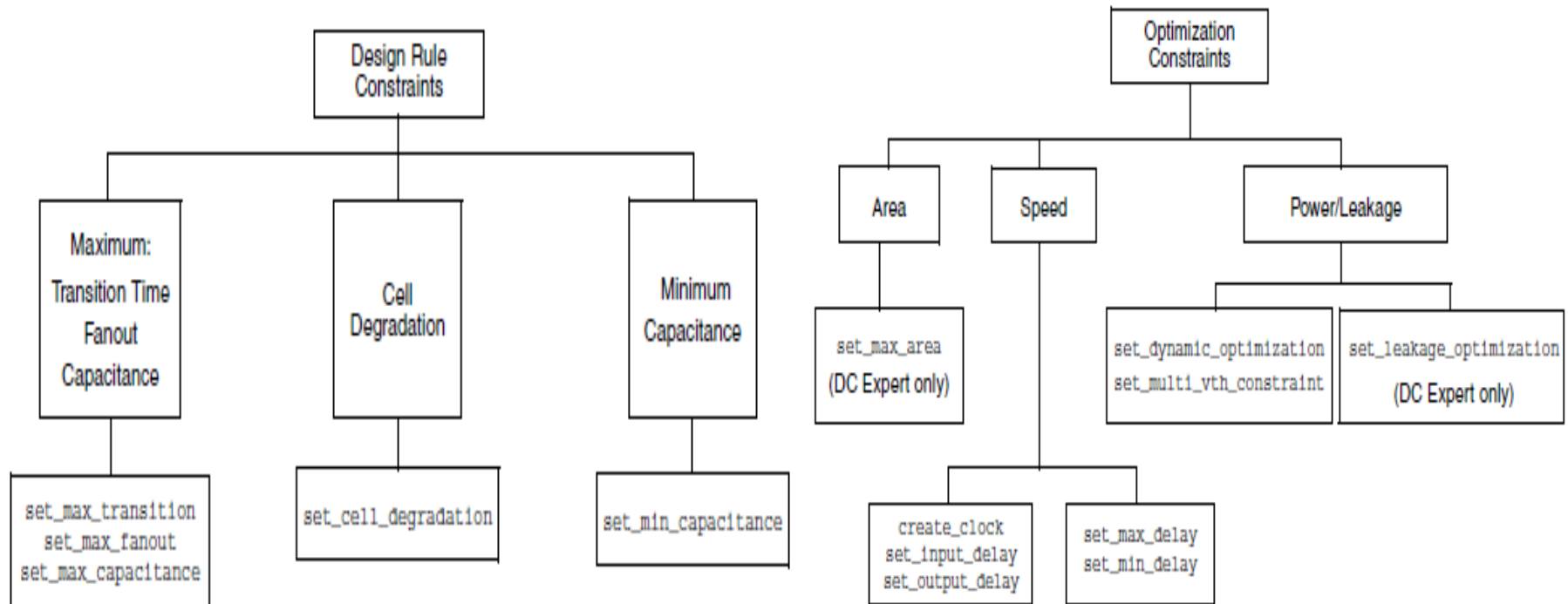
Design Rule Constraints

The logic library defines these implicit constraints. These constraints are required for a design to function correctly. They apply to any design that uses the library. By default, the design rule constraints have a higher priority than optimization constraints

Optimization Constraints

You define this explicit constraints .Optimization constraints apply to the design on which you are working for the duration of the `dc_shell` session and represent the design's goals. During optimization , Design compiler attempts to meet these goals, but no design rules are violated by the process. To optimize a design correctly, you must set realistic constraints.

Constraint types



Using Floorplan Physical Constraints

- Design Compiler in topographical mode supports high level physical constraints such as die area, core area and shape, port locations, cell locations and orientations, keepout margins, placement blockages, preroutes, bounds, vias, tracks, voltage areas and wiring keepouts.
- Using floorplan physical constraint in topographical mode improves timing correlation mode using one of the following methods:
- Export the floorplan information in DEF file or a TCL script from IC Compiler and import this information into Design Compiler.
Create the constraints manually.

Magnet Placement

- Magnet placement improves timing and congestion correlation between Design Compiler and IC Compiler.
- Magnet placement moves standard cells closer to objects specified as magnets.
- You can use magnet placement with any netlist that is fully mapped. Use the `magnet_placement` command to enable magnet placement if `magnet_placement` was used in IC Compiler.

Magnet Placement

- The tool will only perform magnet placement if the standard cells are placed. If you use the `magnet_placement` command before the standard cells are placed, the tool will not pull any objects to the specified magnet. The tool will issue a warning for this condition.
- To perform magnet placement, use the `magnet_placement` command with a specification of the magnets and options for any special functions you need to perform:
`dc_shell-topo> magnet_placement [options] magnet_objects`
- To specify the fanout limit, use the `magnet_placement_fanout_limit` variable. If the fanout of a net exceeds the specified limit, the command does not pull cells of the net toward the magnet objects. The default `magnet_placement` It setting is 1000.
- Magnet placement allows cells to overlap by default. To prevent overlapping of cells, set the `magnet_placement_disable_overlap` variable to true.
- To return a collection of cells that can be moved with magnet placement, use the `get_magnet_cells` command with the options you need:
`dc_shell-topo > get_magnet_cells [options] magnet_list`

Compiling the Design

- When you compile a design, Design Compiler reads the HDL source code and optimizes the design created from that description.
- The tool uses heuristics to implement a combination of library cells that meets the functional, speed, and area requirements of the design according to the attributes and constraints placed on it.
- The optimization process trades off timing and area constraints to provide the smallest possible circuit that meets the specified timing requirements.

The `compile_ultra` Command

- Use the `compile_ultra` command for designs that have significantly tight timing constraints.
- The command is the best solution for timing-critical, high performance designs, and it allows you to apply the best possible set of timing-centric variables or commands during compile for critical delay optimization as well as QoR improvements.

Optimizing the design

- Optimization is the Design Compiler synthesis step that maps the design to an optimal combination of specific target logic library cells, based on the design's functional, speed, and area requirements.
- You use the `compile_ultra` command or the `compile` command to start the compile process, which synthesizes and optimizes the design.
- Design Compiler provides options that enable you to customize and control optimization

Design Compiler performs the following levels of optimization in the following order:

- 1 . Architectural Optimization
- 2 . Logic-Level Optimization
- 3 . Gate-Level Optimization

Architectural Optimization

Architectural Optimization

Architectural optimization works on the HDL description. It includes such high-level synthesis tasks as

- Sharing common subexpressions
- Sharing resources
- Selecting DesignWare implementations (not available in DC Expert)
- Reordering operators
- Identifying arithmetic expressions for datapath synthesis (not available in DC Expert)

Except for DesignWare implementations, these high-level synthesis tasks occur only during the optimization of an unmapped design. DesignWare selection can recur after gate-level mapping.

High-level synthesis tasks are based on your constraints and your HDL coding style. After high-level optimization, circuit function is represented by GTECH library parts, that is, by a generic, technology-independent netlist

Logic level optimizations

Logic-Level Optimization

Logic-level optimization works on the GTECH netlist. It consists of the following two processes:

- **Structuring**

This process adds intermediate variables and logic structure to a design, which can result in reduced design area. Structuring is constraint based. It is best applied to noncritical timing paths. During structuring, Design Compiler searches for subfunctions that can be factored out and evaluates these factors, based on the size of the factor and the number of times the factor appears in the design. Design Compiler turns the subfunctions that most reduce the logic into intermediate variables and factors them out of the design equations.

- **Flattening**

The goal of this process is to convert combinational logic paths of the design to a two-level, sum-of-products representation. Flattening is carried out independently of constraints. It is useful for speed optimization because it leads to just two levels of combinational logic.

During flattening, Design Compiler removes all intermediate variables, and therefore all its associated logic structure, from a design.

Gate level Optimization

Gate-Level Optimization

Gate-level optimization works on the generic netlist created by logic synthesis to produce a technology-specific netlist. It includes the following processes:

- **Mapping**

This process uses gates (combinational and sequential) from the target libraries to generate a gate-level implementation of the design whose goal is to meet timing and area goals. You can use the various options of the `compile_ultra` or `compile` command to control the mapping algorithms used by Design Compiler

- **Delay optimization**

The process goal is to fix delay violations introduced in the mapping phase. Delay optimization does not fix design rule violations or meet area constraints..

- **Design rule fixing**

The process goal is to correct design rule violations by inserting buffers or resizing existing cells. Design Compiler tries to fix these violations without affecting timing and area results, but if necessary, it does violate the optimization constraints.

Gate level Optimization

Area optimization

- The process goal is to meet area constraints after the mapping, delay optimization, and design rule fixing phases are completed. However, Design Compiler does not allow area recovery to introduce design rule or delay constraint violations as a means of meeting the area constraints.
- You can change the priority of the constraints by using the `set_cost_priority` command. Also, you can disable design rule fixing by specifying the `-no_design_rule` option when you run the `compile_ultra` command or `compile` command.
- However, if you use this option, your synthesized design might violate design rules.

Data Path Optimization

- Datapath design is commonly used in applications that contain extensive data manipulation, such as 3-D, multimedia, and digital signal processing (DSP).
- DC Ultra enables datapath optimization by default when you use the `compile_ultra` command.
- Datapath optimization is comprised of two steps:
 - ❑ Datapath extraction, which transforms arithmetic operators (for example, addition, subtraction, and multiplication) into datapath blocks.
 - ❑ Datapath implementation, which uses a datapath generator to generate the best implementations for these extracted components.

Note:

Datapath optimization requires a DC Ultra license and a DesignWare license. The `DesignWare` license is pulled when you run the `compile_ultra` command

Data Path Optimization

analyze_datapath

Provides a datapath analysis report that lists the resources and datapath blocks used in the current design.

SYNTAX

analyze_datapath

[-file *file_name*]

ARGUMENTS

-file *file_name*

Specifies the file name of a log file that contains the **report_resources -hierarchy** and **report_area -designware** reports from a previously compiled design. This option enables you start the tool and use the **analyze_datapath** command to generate a datapath analysis report for a design without having to read in and compile the design a second time.

Data Path Optimization

DESCRIPTION

- The **analyze_datapath** command provides a datapath analysis report that lists a summary of the resources and datapath blocks used in the current design.
- The report includes information reported by the **report_resources -hierarchy** command and the **report_area -designware** command.
- The report displays the area of singleton DesignWare designs and extracted datapath designs.
- The report summarizes the number of singleton DesignWare components, summarizes the number of datapath designs, and lists the designs output width ranges.

Uniquify

Select the most top design of the hierarchy:

There might be some instances which are called multiple times in the module but they have the same name so during compilation the tool cannot identify which instance to compile or optimize. Uniquify assign a unique name to those instance which are called multiple times.

Name ▾	Design Area	Dont Touch
MEM		0 undefined
ROM		0 undefined
SES_ID		0 undefined
SYN_DEC_8_0		0 undefined
SYN_DEC_8_1		0 undefined
SYN_DEC_8_2		0 undefined
SYN_DEC_8_3		0 undefined
SYN_DEC_8_4		0 undefined
SYN_DEC_8_5		0 undefined
SYN_DEC_8_6		0 undefined
SYN_DEC_8_7		0 undefined
addr_present		0 undefined
addr_previous1		0 undefined
addr_previous2		0 undefined
b_to_g_0		0 undefined
b_to_g_1		0 undefined

(Design View)

```
design_vision-xg-t> uniquify
Removing uniquified design 'b_to_g'.
Removing uniquified design 'SYN_DEC_8'.
Uniquified 2 instances of design 'b_to_g'.
Uniquified 8 instances of design 'SYN_DEC_8'.
```

(Log Window)

Save Design

Five Design Files:

- *.sdc timing constraint file for P&R
- *.vg gate-level netlist for P&R
- *.sdf timing file for Verilog simulation
- *.db binary file (all the constraints and synthesis results are recorded.

{ Command Line }

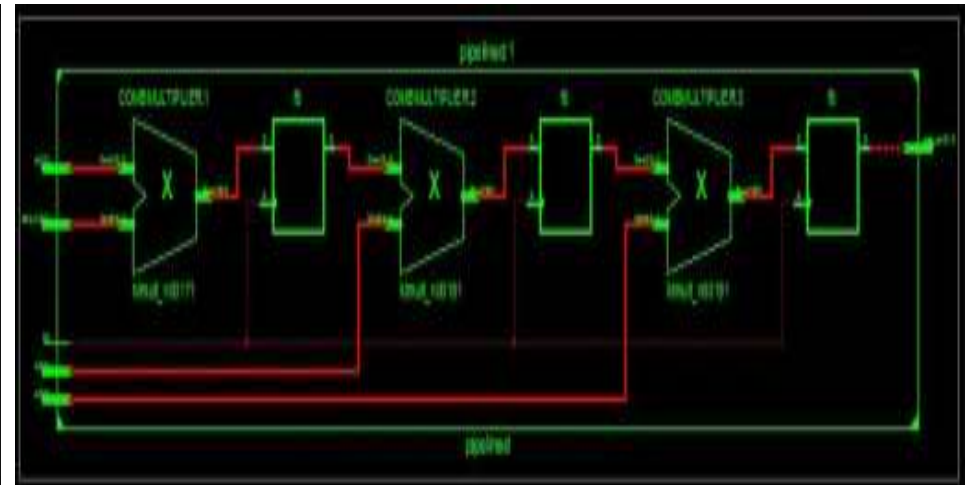
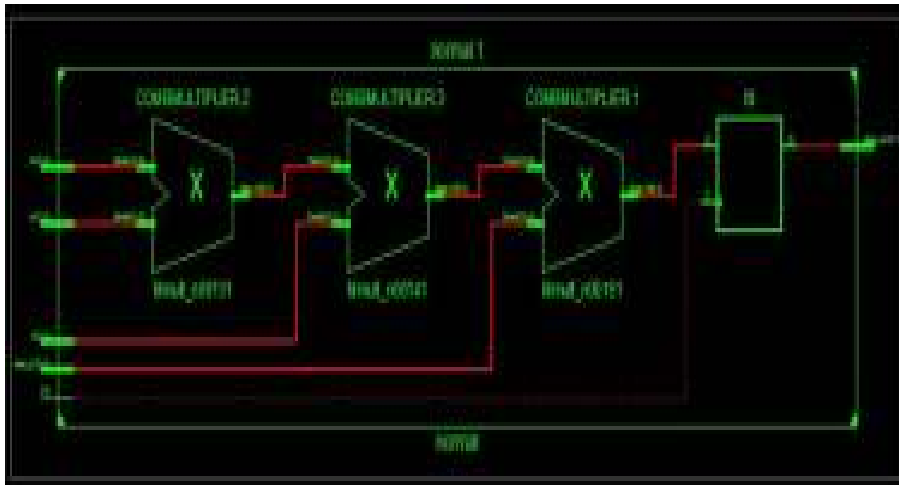
```
write_test_protocol -f stil -out "CHIP.spf"  
write_sdc CHIP.sdc  
write -format verilog -hierarchy -output "CHIP.vg"  
write_sdf -version 1.0 CHIP.sdf  
write -format db -hierarchy -output "CHIP.db"
```

Synthesis Report

- Report Design Hierarchy
- Report Area
- Design View
- Report Timing
- Critical Path Highlighting
- Timing Slack Histogram

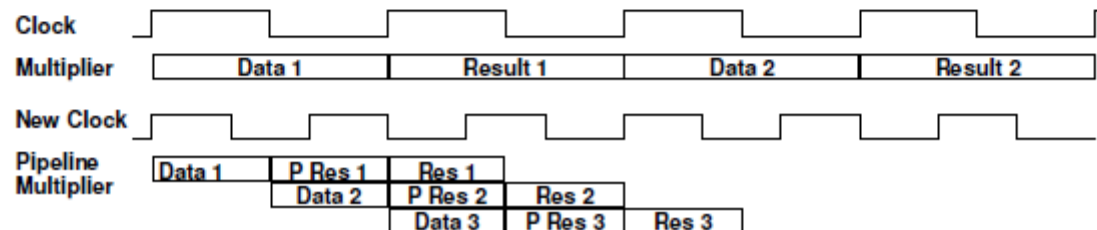
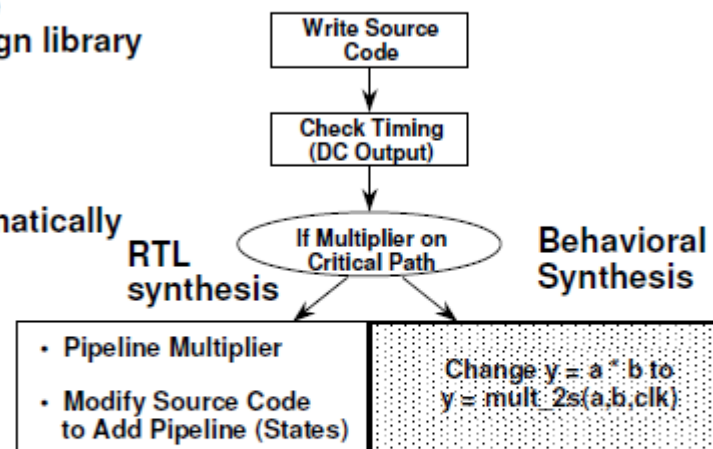
What is Pipelining?

- A **pipeline** is a set of data processing elements connected in series, so that the output of one element is the input of the next one.
- In most of the cases we create a pipeline by dividing a complex operation into simpler operations.
- We can also say that instead of taking a bulk thing and processing it at once, we break it into smaller pieces and process it one after another



Pipelined Components

- If faster multiplier required, use pipelined multipliers from design library
 - Two-stage
 - Three-stage
- Benefit of using design library
 - Behavioral synthesis automatically generates control logic
 - Better area efficiency
 - Resource sharing



Design Methodologies

System Design & Synthesis

- Lecture Set 13: Design Reuse and Intellectual Property Cores
- Design Reuse
- System-on-a-Chip (SoC) Design
- System Design Rules and Guidelines
- RTL Coding Guidelines
- Macro Synthesis Guidelines
- Development of Hard Macros
- Macro Deployment
- System Integration

Design Reuse - 1

- Motivation
 - High cost of design and verification
 - Shorter design cycles
 - Higher quality demands
 - Emerging System-on-a-Chip (SoC) designs
 - Very short design cycles
 - Large numbers of distinct designs
 - Analogous to board design today

Design Reuse - 2

- Requirements
 - Correct and robust
 - Well-written, well-documented, thoroughly-commented code
 - Well-designed verification suites and robust scripts
 - Solves a general problem
 - Easily configurable; parameterized
 - Supports multiple technologies
 - Soft macros: synthesis scripts span a variety of libraries
 - Hard macros: porting strategies to new technologies

Design Reuse - 3

- Requirements (continued)
 - Simulates with multiple simulators
 - Both VHDL and Verilog version of models and test-benches
 - Work with all major commercial simulators
 - Accompanied by full verification environment
 - Test benches and verification suites that provide high levels of verification coverage
 - Verified rigorously before release
 - Includes construction of actual prototype tested in actual system with real software

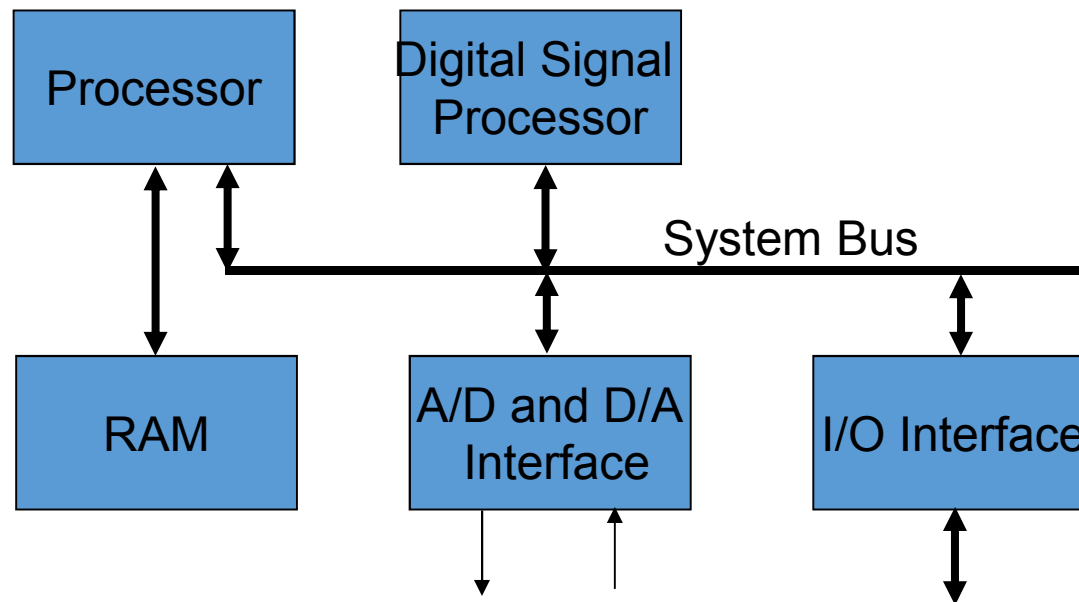
Design Reuse - 4

- Requirements (continued)
 - Documented in terms of applications and restrictions
 - Valid configurations and parameter values
 - Interfacing requirements and restrictions

System-on-a-Chip (SoC) Design

- Overview
 - Example
 - Design Paradigms
 - The Role of Reuse
 - Macros, Cores, and Blocks

Example SoC



Design Paradigms - 1

- Design Flow - Spiral model instead of Waterfall model
- Top-down/Bottom-up Mixture instead of Top-down
- Construct by correction instead of Correct by construction

Design Paradigms - 2

- **Waterfall model**

- Design flows from one phase to another
- Phases such as algorithm development, RTL coding and functional verification, and synthesis and timing verification, and physical design all performed by different teams
- Limited reverse flow in design

Design Paradigms - 3

- Spiral model
 - Teams work on multiple aspects simultaneously, performing incremental improvement
 - Concurrent development of HW and SW
 - Parallel verification and synthesis
 - Floorplanning and place and route at synthesis
 - Modules developed only if predesigned hard or soft macro unavailable
 - Planned iteration throughout

Design Paradigms - 4

- **Top-down Design**

- Assumes that lowest level blocks can be designed and built
- If not, start over
- Not compatible with maximum reuse of macros

- **Top-down/Bottom-up Mixed Design**

- Design downward, but target macros or combination of macros build upward

Design Paradigms - 5

- Correct by construction
 - Focus on one pass design with goal of completely correct during this pass
- Construction by correction
 - Begin with the realization that multiple complete iterations will be required
 - First pass is quick to see the problems at various levels caused by the decisions at prior levels
 - Performed design refinement several times

The Role of Reuse

- Redesign of cores such as processors, bus interfaces, DSP processors, DRAM controllers, RAMS, etc. is not cost-effective
- Redesign of common blocks such as ALUs, barrel shifters, adders, and multipliers, likewise, not cost effective
- Availability of well-designed macros particularly parameterizable versions can greatly reduce cost

Macros, Cores and Blocks - 1

- Terms in header used synonymously
- Other terms:
 - Subblock - subcomponent of a macro, core, or block - too small or specific to be a stand-alone component
 - Hard macro - A hard macro is delivered to integrator as a GDSII file (tape-out file for fabrication) - fully designed, placed and routed by the supplier -

Macros, Cores and Blocks - 2

- Soft macro - Delivered to integrator as synthesizable RTL code - may also include test benches, synthesis scripts, etc.
- Firm macro - (defined by Virtual Socket Interface Alliance) - RTL code with supplemental physical design information

System Design Rules and Guidelines - 1

□ Timing and Synthesis Issues

- Rule - synchronous and register-based
- Rule - document clock domains and frequencies
 - Guideline - use the smallest possible number of clock domains
 - If phase-locked loop (PLL) used, have disable or by-pass.
- Rule - document reset strategy

System Design Rules and Guidelines - 2

- Rule - document reset strategy
 - Guideline - if asynchronous, must be de-asserted synchronously
 - Guideline - Synchronous preferred
- Rule - overall design goals for timing, area, and power should be documented before macros designed or selected - overall synth methodology planned early

System Design Rules and Guidelines - 3

- Functional Design Issues
 - Rule - Design of on-chip buses that interconnect the various blocks must be an integral part of macro selection and the design process
 - Rule - Develop strategy for bring-up and debug early in the design process
 - Guideline - provide controllability and observability, the keys to easy debug

System Design Rules and Guidelines - 4

- Physical Design Issues
 - Rule - Floor-planning, placing and routing of a combination of hard and soft macros must be developed before hard macros are selected or designed
 - Comment - Hard macros can be very detrimental to place and route

System Design Rules and Guidelines - 5

- Rule - Floorplanning must begin early in the design process
- Decide on the basic clock distribution structure early in the design process
 - Guideline - Low speed synchronous bus between modules - High speed local clocks synchronous to the bus clock by PLLs or buffering - multiple of bus clock

System Design Rules and Guidelines - 6

- Verification

- Rule - strategy must be developed and documented before macro selection or design begins
 - Guideline - selection of verification tools can affect the coding style of macros and the design - testbench design must be started early in the design process

System Design Rules and Guidelines - 7

- Manufacturing Test Strategies
 - Rule - system-level chip manufacturing test strategy must be documented
 - Guideline - On-chip test structures are recommended for all blocks - different test strategies for different blocks - master test controller

System Design Rules and Guidelines - 8

- Guideline - Built-In Self-Test (BIST)
recommended for RAMs - also non-BIST data retention tests
- Guideline - microprocessor tests usually involve parallel vectors and serial scan - test controller must provide for both
- Guideline - for other blocks, full scan is good choice - sometimes with logic BIST

RTL Coding Guidelines - 1

- Fundamental Principles
- Basic Coding Practices
- Coding for Portability
- Guidelines for Clocks and Resets
- Coding for Synthesis
- Partitioning for Synthesis
- Designing with Memories

RTL Coding Guidelines - 2

- Fundamental Principles
 - Use simple constructs, basic types (for VHDL), and simple clocking schemes
 - Be consistent in coding style, naming, and style for processes and state machines
 - Use a regular partitioning method with module outputs registered and modules of about the same size
 - Use comments, meaningful names and constants and parameters instead of numbers

RTL Coding Guidelines - 3

- Basic Coding Practices
 - Rule - Develop a naming convention for the design
 - Use lowercase letters for signal names, variable names, and port names
 - Use uppercase letters for constants and user-defined types
 - Use meaningful names
 - Keep parameter names short, but descriptive

RTL Coding Guidelines – 4

- Basic Coding Practices (continued)
 - Use clk for the clock signal. If multiple clocks, use clk as the prefix for all clock signals.
 - Use the same name for all clk signals driven by same source
 - For active low signal, end with underscore _n for standardization
 - Use rst for reset signals - if active low, use rst_n
 - For buses, use consistent bit ordering; recommend (y downto x) (VHDL) or (x:0) (Verilog)
 - Use same name for connected ports and signals
 - Use *_r for register output, *_a for asynchronous signals, *_pn for signals in phase n, *_nxt for data in to register *_r, and *_z for internal, 3-state signal.
- Many more!

RTL Coding Guidelines - 5

- Coding for Portability
 - Rule (VHDL) - Use only IEEE standard types
 - Use std_logic instead of std_ulogic
 - Be conservative re number of created types
 - Do not use bit or bit_vector (no built in arithmetic)
 - Do not use hard-coded values
 - (VHDL) Collect all parameter values and function definitions into a package *DesignName_package.vhd*
 - (Verilog) Keep 'define statements in a separate file *DesignName_params.v*

RTL Coding Guidelines - 6

- Avoid embedding dc_shell scripts to avoid unintended execution with negative impact and obsolescence
- Use technology-independent libraries to maintain technology independence (e.g., DW in Synopsys)
- Avoid instantiating gates in designs
- If technology-specific gates must be instantiated, isolate in separate module
- If gate instantiated, use technology-independent library (e.g., GTECH in Synopsys)

RTL Coding Guidelines - 7

- Code for translation between VHDL and Verilog
 - Do not use reserved keywords from Verilog as identifiers in a description in VHDL and vice-versa.
 - In VHDL, do not use:
 - **generate**
 - **block**
 - Code to modify **constant** declarations

RTL Coding Guidelines - 8

- Guidelines for Clocks and Resets
 - Avoid mixed clock edges
 - Duty cycle of clock becomes critical in timing analysis
 - Separate serial scan handling required
 - If required, worst case duty cycle(s) must be accurately modeled, duty cycle documented, and + and - edge flip-flops in separate modules
 - Avoid clock buffers (done in physical design)
 - Avoid gated clocks (technology specific, timing dependent, and non-scannable)

RTL Coding Guidelines – 9

- Avoid internally-generated clocks (logic they clock cannot be scanned; synthesis constraints difficult to write)
- Avoid internally generated resets
- If gated clocks, or internally-generated clocks or resets required, do in separate module at the top level of the design and partition into modules using single clock and reset.
- Model gated clocks as if registers enabled.
- Model complex reset by generating reset signal in separate module

RTL Coding Guidelines - 10

- Coding for Synthesis
 - Infer registers
 - Avoid latches
 - Avoid combinational feedback
 - Specify complete sensitivity lists
 - In Verilog, always use non-blocking assignments in `always@(*edge clk)`
 - In VHDL, signals preferred to variables, but variables can be used with caution

RTL Coding Guidelines - 11

- Coding for Synthesis (continued)
 - Use case over if-then-else whenever priority structure not required.
 - Use separate processes for sequential state register and combinational logic
 - In VHDL, create an enumerated type for state vector. In Verilog, use 'define. Why 'define rather than parameters?
 - Keep FSM logic separate

RTL Coding Guidelines - 12

- Partitioning for Synthesis
 - Register all outputs
 - Keep related logic together
 - Separate logic with different design goals
 - Avoid asynchronous logic
 - Keep mergeable logic within a given module
 - Avoid point-to-point exceptions and false paths such as multicycle paths and false paths
 - Avoid top-level glue logic

RTL Coding Guidelines - 13

- Designing with Memories
 - Partition address and data registers and write enable logic in a separate module (allows use with both synchronous and asynchronous memories)
 - Add interface module for asynchronous memory

Macro Synthesis Guidelines - 1

- Basic timing budget for macro must be specified
- Timing budgets must be developed for each subblock in the macro
- Initially synthesized to single tech library
- In productization, synthesized to multiple tech libraries

Macro Synthesis Guidelines - 2

- Subblock Synthesis
 - Typically compile-characterize-write script- reoptimize approach
- Macro Synthesis
 - Compile individual subblocks
 - Characterize-compile overall
 - Perform incremental compile
- Use of RAM Compiler and Module Compiler

Developing Hard Macros

- Hard Macro Design Process
- Models and Documentation
 - Behavioral
 - Functional
 - Timing
 - Floorplan

Macro Deployment - 1

- Deliverables
 - Soft Macros
 - RTL code
 - Support files
 - Documentation
 - Hard Macros
 - Broad set of integration models
 - Documentation for integration into final chip
 - Design archive of files

Macro Deployment - 2

- Deliverables
 - Soft Macros
 - RTL code
 - Support files
 - Documentation
 - Hard Macros
 - Broad set of integration models
 - Documentation for integration into final chip
 - Design archive of files

System Integration

- The integration process
 - Selection of macros
 - Design of macros
 - Verification of macros
 - The design flow
 - Verification of design
- Overall: Verification important throughout the macro design and system design



Thank You



The Solutions People

