Module 17

Managing the Logic Synthesis Process

cadence

This module just briefly presents the basic steps of the synthesis process and the associated Cadence Encounter® RTL Compiler commands.

Module Objective

In this module, you:

- Go through the Synthesis Flow and manage the Synthesis Process

Topics

- Reading the HDL source
- Elaborating the design
- Applying constraints
- Mapping to technology cells
- Writing the netlist
- Optimizing Arithmetic Expressions
- Sharing/un-sharing Resources
- Boundary optimization, Retiming, Scan Insertion
- Analyzing Results, Report Timing

cadence

This module expands upon the synthesis process that the first module examined. The flow described here is similar to all popular synthesis tools. Any more exact examples derive from the Cadence Encounter RTL Compiler (RC).

Logic Synthesis Goals

Work fast

- Faster than the other guy's synthesis, definitely faster than manual

Work accurately

- Gate-level model functionally equivalent to RTL model
- Cost estimates correspond well with actual values measured later
  - Area, time, power

Maximize Quality of Silicon (QoS)

- Minimize time – maximize frequency and throughput
- Minimize area – cell count and cell size
- Minimize power – switching activity and leakage power

cadence

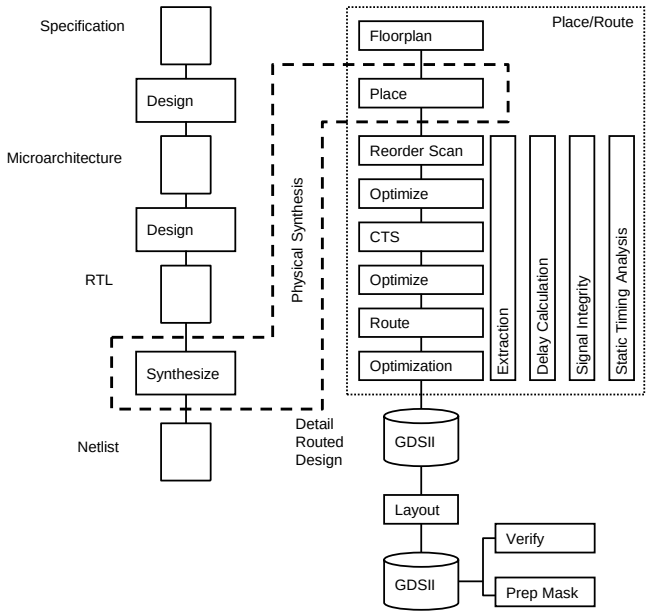
Synthesis tools, like all tools really, have only the two goals to work fast and work well. This slide further splits from the "work well" part the issue of accuracy, for if the tool does not evaluate function and timing well, who cares how wonderful are the false numbers?

- Working fast in a modern highly-competitive industry environment means not just optimizing the design faster than I could do it with pencil and paper but also faster than whatever tool my competitor uses. Here, "faster" means not just tool speed, but also tool usability and tool support that come to play.
- Working accurately, is at least for a modern tool, no longer an issue. Functional accuracy has not been an issue for decades and is now so rare that it is the highest level bug when encountered. Timing accuracy has for long been mainly an issue of how closely the chosen wire-load tables model actual parasitics of the routed design. Modern synthesis tools work in conjunction with layout tools to greatly improve timing accuracy.
- Producing high-quality silicon means minimizing the "cost" factors. Cost factors are foremost timing and only slightly secondarily area and power. All of these cost factors are really just different ways to provide more utility to the customer, i.e., to make the customer want to buy my product instead of the other guy's:
  - Timing is important because higher clock speed means that I can put more functionality into a smaller space and have it respond more quickly to my customer's inputs.
  - Area is important because less area means that I can put more functionality into a smaller space and I can also reduce the price because my own manufacturing and packaging costs are lower.
  - Power is important because less power dissipation means that my customer has lower energy costs, if it is a portable device then the battery can be lighter and stay charged longer, and if it is a handheld device then the customer does not cook their hand while holding it.

Logic Synthesis Basic Process

From RTL input:

- Parse
- Translate
- Optimize
- Map
- Optimize again
- Insert scan
- Optimize again



This module expands somewhat upon the synthesis process that the first module examined, for if you understand something about what's going on inside the flow, then you are likely to provide better-quality RTL inputs.

After every major operation the tool again optimizes the design. Optimization means calculating the design timing, area and power, and if not within your targets, swapping gates and restructuring logic in an attempt to meet your targets.

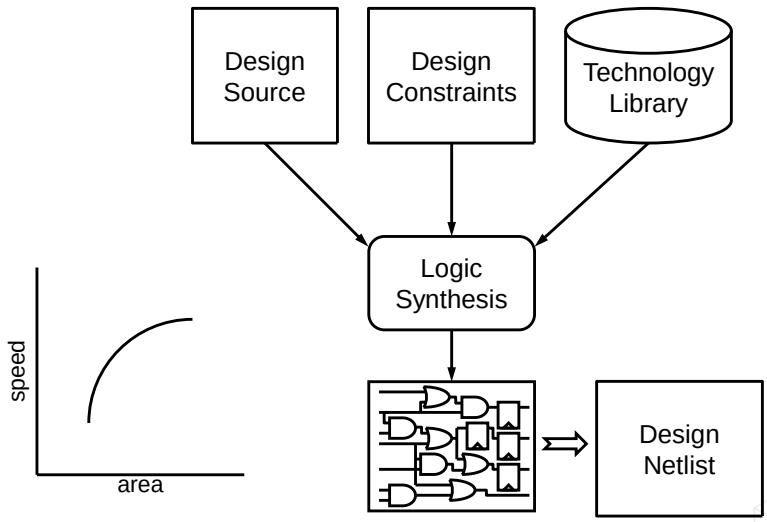
Logic Synthesis Inputs and Outputs

Input

- Synthesizable HDL
- Constraints – (SDC)
- Technology libraries

Output

- Netlist – (usually Verilog)
  - Of technology macros



Inputs to the synthesis process include:

- The design itself, written in the “synthesizable subset” of the HDL.
- Design constraints, including the required clock speed, input drive strength and arrival time with respect to the clock, and output load and required arrival time with respect to the clock. Design constraints can also include power consumption and can sometimes include noise immunity, testability, and factors affecting ease of place and route. Design constraints may be in the Synopsys Design Constraint (SDC) format, and synthesis vendors also have their own proprietary formats.
- At least one technology library, including all pertinent information about the available cells, and factors for estimating interconnect delays.

Outputs from the synthesis process include:

- Most importantly, the design as a structural netlist of technology cells.
- Optionally, post-synthesis timing information for annotation to a gate-level simulation.
- A log file and any other reports you requested from the tool.

Logic Synthesis Basic Flow

Phase	Description	Example RC Command
Read HDL	Parse source code, check syntax	read_hdl
Elaborate	Build data structures and registers	elaborate
Apply constraints	Specify operating conditions, clocks, and I/O timing	define_clock set_input_delay
Map to generic cells	Map to generic cells. Technology-independent optimizations	synthesize to_generic
Map to technology cells	Map to technology cells. Technology-dependent optimizations	synthesize to_map
Optimize (optional)	Improve performance	retime prepare retime min_delay
Insert scan	Build the scan chain	insert_dft synthesize –to_map connect_scan_chains synthesize incremental
Analyze results	Create and analyze reports	report_area report_gates report_timing
Write netlist		write_hdl

cadence

The remainder of this module examines a generic flow from reading the HDL to writing the optimized netlist. Example commands are those of the Cadence Encounter RTL Compiler. Most of these commands have several options that this training does not show. This training also does not show the complete command set. For example, applying constraints is quite a bit more involved then simply defining the clocks.

Reading HDL

Read the HDL source and check syntax.

- Options can include the expected language and version.
- The tool generates an internal parse tree representing the HDL source.

Example

```
read_hdl my_design.v
```


The first step is to read the HDL source and check its syntax. Options might include to specify the expected language and language version, and for Verilog, to define text replacement macros. The tool generates an internal parse tree representing the HDL source.

elaborate

Check semantics and construct a design.

- Construct and connect hierarchy
- Infer sequential elements
- Expand function calls
- Propagate constants
  - Precompute results of expressions having only constant terms
- Remove dead code
  - Remove statements that cannot execute or whose results are not used
- Unroll loops
  - Replace each for loop iteration with its own statement

Example

```
elaborate my_top_module
```


cadence

Elaboration constructs a design.

The elaborator:

- Constructs and connects the hierarchy
- Infers sequential elements
- Expands functions at the point of each call
- Propagates constants
- Removes dead code
- Unroll loops

Example Elaboration Optimizations

Constant Propagation

- Original code

```
a = 0;
b = a + 1;
c = 2 * a;
```

- Optimized code

```
b = 1;
c = 0;
```

Dead Code Removal

- Original code

```
a = x;
b = a + 1;
c = 2 * a;
```

- Optimized code

```
b = x + 1;
c = 2 * x;
```

Loop Unrolling

- Original code

```
for (a=0;a<=2;a=a+1)
  z[a] = x[a] + y[2-a];
```

- Optimized code

```
z[2] = x[2] + y[0];
z[1] = x[1] + y[1];
z[0] = x[0] + y[3];
```

323

cadence

The elaborator can do some optimization as it constructs the design:

- It can propagate constants to eliminate unneeded operators.
- It can remove code that is not executed or that sets variables that are not used.
- It unrolls loops.

Applying Constraints

Inform the tool of the expected operating environment and of any timing paths it can ignore or are longer than one clock cycle in normal operation.

- Clock signal attributes
  - Period, duty cycle, skew, latency
- Arrival times (w.r.t. clock) at I/O ports
- Environmental attributes
  - Input drive, output load
- Timing exceptions
  - False or multi-cycle timing paths

Example

```
define_clock -name 100MHz -period 10000
```


cadence

Timing and design constraints describe the “design intent” and the surrounding constraints, including synthesis, clocking, timing, environmental, and operating conditions.

Set these constraints on start points and end points to ensure that every path is properly constrained to obtain an optimal implementation of the RTL design. A path begin point is from either an input port or a register clock pin, while an end point is either an output port or a register data pin.

Use these constraints to:

- Describe different attributes of clock signals, such as the duty cycle, clock skew, and the clock latency.
- Specify input and output delay requirements of all ports relative to a clock transition.
- Apply environmental attributes, such as load and drive strength to the top-level ports.
- Set timing exceptions, such as multicycle paths and false paths.

Mapping to Generic Cells and Optimize

Do technology-independent RTL optimizations:

- Prune unloaded logic
- Optimize arithmetic expressions
- Share common sub-expressions
- Share and duplicate resources
- Carry-save adder transformations
- Merge and map operators to implementations

Example

```
synthesize -to_generic
```


cadence

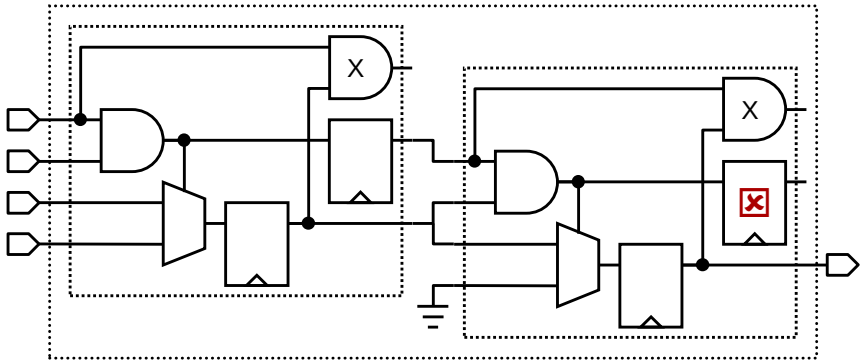
The synthesis tool does technology-independent RTL optimizations as a preliminary step, including:

- Datapath synthesis
- Resource sharing and speculation
- Multiplexor optimization
- Carry-save adder (CSA) optimizations

Pruning Unloaded Logic

Technology-independent optimizations remove unloaded logic.

- Removed cells marked X below do not transitively drive outputs.

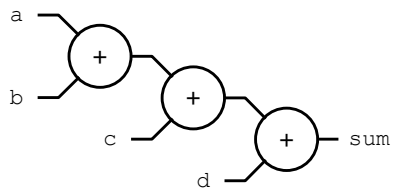


Technology-independent optimizations include the removal of unloaded logic. Here, the cells marked X do not transitively drive outputs so are removed.

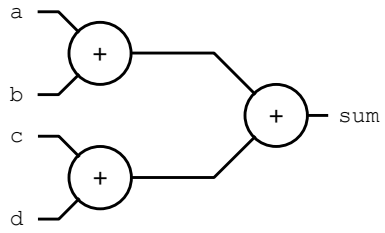
Optimizing Arithmetic Expressions

```
sum = a + b + c + d;
```

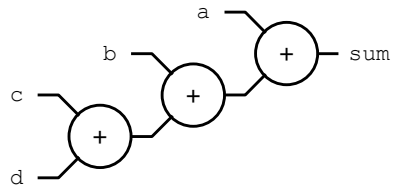
Initial implementation



Optimized for speed if all inputs arrive together



Optimized for speed if input “a” arrives late



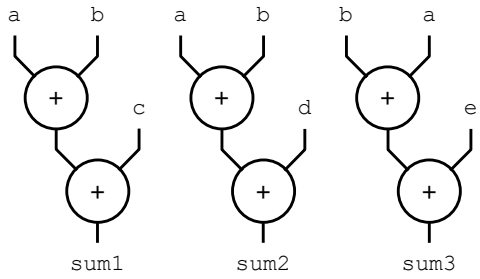
Technology-independent optimizations include the restructuring of arithmetic expressions. The logic tree is restructured to as much as practical balance the arrival times from all inputs. During RTL optimizations the tool estimates path delay based upon expression depth.

Tools typically honor explicit parentheses. Include parentheses to force structure when you know for example that some input will arrive late, and otherwise do not include parentheses.

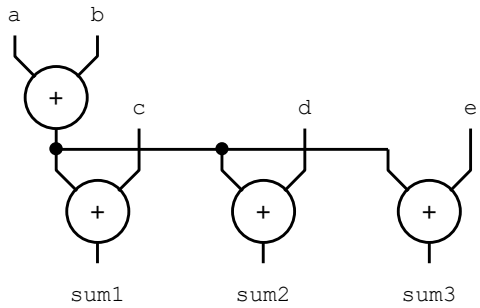
Sharing Common Sub-Expressions

```
sum1 = a + b + c;
sum2 = a + b + d;
sum3 = b + a + e;
```

Initial Implementation



Optimized Implementation



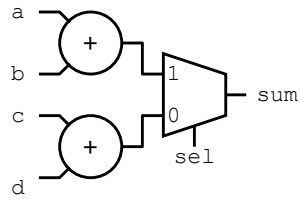
Technology-independent optimizations include sub-expression elimination. Here, the (a+b) sub-expression is common to the three sums, so is shared. The tool may later duplicate sub-expressions to meet timing requirements. The order of the sub-expression operands is immaterial, but for most tools the subexpressions must be in same relative position in its enclosing expression.

Sharing Resources

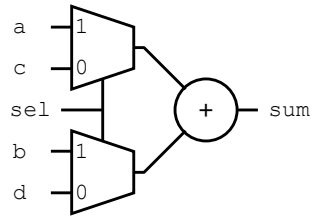
- A resource is a computational element, such as an adder, shifter, or multiplexor.
- Each HDL operator represents a unique resource type.
  - **+** operator requires an adder.
  - **>** operator requires a comparator.
- *Without* sharing, the number of resources is the number of operators.
- *With* sharing, the number of resources can be reduced.
  - Some operators can share the same resource in alternate clock cycles.
  - Some resources can implement more than one type of operation, e.g., **+** and **–.**

```
if (sel==1)
  sum = a + b;
else
  sum = c + d;
```

Possible Implementation



Alternate Implementation



A resource is a computational element, such as an adder, shifter, or multiplexor.

Each HDL operator represents a unique resource type, for example the “plus” (+) operator represents an adder and the “greater than” (>) operator represents a comparator. Some tools can even merge some adjacent operators and map the merged operators to a more optimized resource.

Without sharing, the number of required resources is the number of operators.

With sharing, the number of required resources can be reduced:

- Similar operators that are utilized in non-overlapping clock cycles can be shared.
- Add and subtract operators, for example, can be considered to be similar.

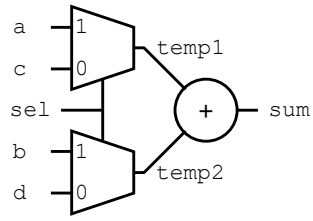
Depending upon the tool, resource sharing may occur by default or you may need to enable it.

Un-Sharing Resources (Speculation)

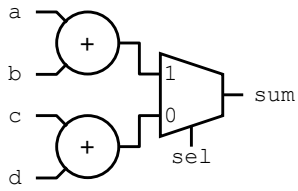
Can also *add* resources to meet timing requirements.

```
begin
 if (sel==1) begin
   temp1 = a;
   temp2 = b;
  end
 else begin
   temp1 = c;
   temp2 = d;
  end
 sum = temp1 + temp2;
end
```

Possible Implementation



Alternate Implementation
(select arrives late)



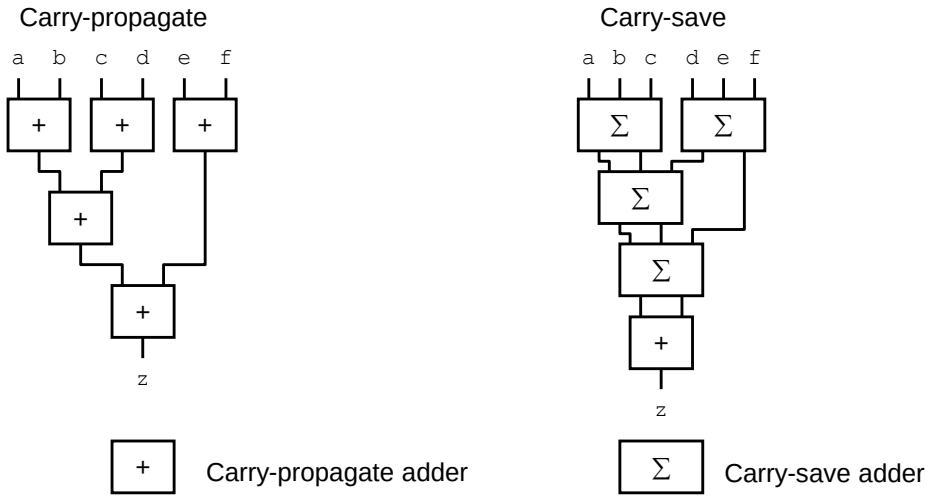
cadence

The tool can alternatively duplicate resources where needed to meet timing requirements. Here, because the select signal arrives late, synthesis duplicated the adder.

Carry-Save Adder Transformations

Carry propagation frequently appears on the critical timing path.

Carry-save adder transformations “save” carry generation until the end.



The most straightforward way to add a set of numbers is to employ an adder tree. Each adder consumes two numbers and produces one. The adder at the root of the tree generates the final sum.

A carry-save adder takes three numbers and produces two outputs, one formed with the sums and the other with the carryouts. Carrysave transformation can greatly improve area and timing.

CSA transformation may apply to any datapath operators whose isolated implementation includes a carry propagate adder. This includes all discrete and relational operators. When the output of one such operator feeds into another datapath operator, it becomes a candidate for CSA transformation. When the output of one mergeable operator feeds into the input of another mergeable operator, it becomes a candidate for CSA transformation.

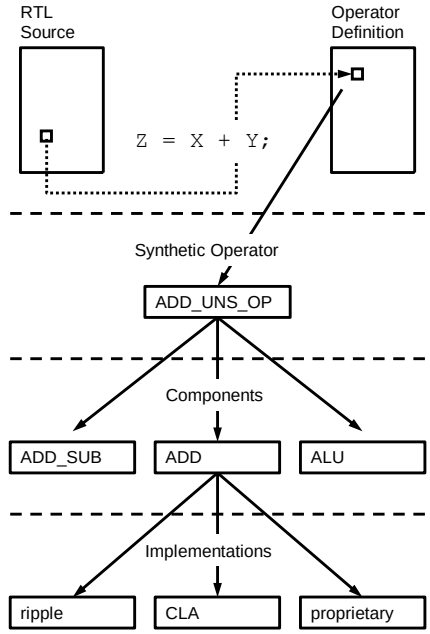
Datapath operators are merged in scenarios such as the following:

- Any combination of Vector Sum, Sum-of-Product, and Product-of-Sum, including intermediary inverted signals.
  - assign cs = a + b + c + d; assign y = cs * p + q;
- Comparator
  - assign p = a + b; assign q = c + d; assign is_greater = (p > q);
- Multiple Fanout
  - cs = a * b; x = cs + c; y = cs + d;
- Multiplexers
  - assign tmp = s ? a*b + c*d : a*c + b*d; assign z = tmp + e;
- Inverter
  - wire [16:0] tmp = a + b; assign z = ~tmp + c;
- Truncated CSA
  - assign p = a + b; assign q = c + d; assign r = p + q;
    assign y = r[17:8];

Selecting the Architecture: ChipWare

Advanced synthesis tools such as the Cadence Encounter RTL Compiler (RC) have libraries of re-usable designs.

- The Cadence ChipWare (CW) library includes:
  - Common combinational and sequential components.
  - Arithmetic components (adders, subtractors, multipliers).
  - Memory components (flip-flops, FIFOs).
- Logic synthesis automatically maps operators to available CW components.
- Many CW components have multiple architectural implementations allowing logic synthesis to make area/time tradeoffs.



A ChipWare library is a set of re-usable functional blocks.

The Cadence Encounter RTL Compiler comes with a large set of pre-defined ChipWare components. You can add your own user-defined ChipWare components, and an IP vendor can package re-usable blocks as user-defined ChipWare components and provide them in a user-defined ChipWare library.

ChipWare involves a four-tier hierarchy:

- 1st – HDL operators (e.g., “+”) and HDL functions.
- 2nd – Synthetic operators, a language-neutral and technology-independent definition of a particular operator or function. For example, ADD_UNS_OP is a synthetic operator representing an addition between two unsigned operands. A synthetic operator can be bound to a selection of ChipWare components.
- 3rd – ChipWare components, which can have multiple implementations.
- 4th – The component implementations.

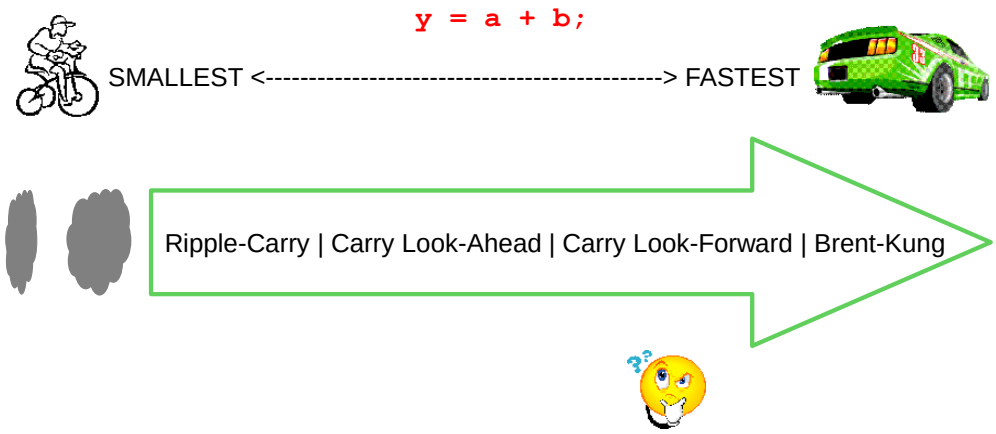
The Cadence RTL Compiler knows about the interface and functionality of these libraries, and for each operator, chooses an implementation based upon QoS calculations.

You can alternatively directly instantiate a ChipWare implementation in your RTL code.

Selecting the Architecture: Tradeoffs

Different architectures have different area and timing characteristics.

Logic synthesis chooses the smallest available architecture meeting timing.



Different resource implementations have different area and timing characteristics.

Logic synthesis chooses from the available implementations the smallest that it expects to meet the timing requirements.

Selecting the Architecture: Manual Instantiation

Sometimes you need to explicitly instantiate a specific technology vendor's library macrocell that the tool does not infer:

- I/O pad
- Memory
- Processor (etc.)

Explicit instantiation creates issues:

- Makes code technology dependent
- Can prevent some optimizations
- Can make the code less readable

```
module my_adder
#(W=4)
 (output [W-1:0] s,
  input  [W-1:0] a,b);
  assign s = a + b;
endmodule
```

```
module my_adder
#(W=4)
 (output [W-1:0] s,
  input  [W-1:0] a,b);
  CW_add #(W) u1 (.A(a), ...);
endmodule
```

Create a "wrapper" module around the macro so that "swapping in" some other IP is merely a configuration change.

cadence

Sometimes you need to explicitly instantiate a specific technology vendor's library macrocell that the tool does not infer, for example the I/O pads, a memory block, or an embedded processor. This explicit macro instantiation makes the code dependent upon the technology. To somewhat preserve technology independence, you can wrap the technology-specific component in a module for which you provide a technology-independent interface. You instantiate your technology-independent module where needed and configure the design to choose one or the other definition of the module.

Sometimes you want to explicitly instantiate a specific technology vendor's library macrocell because you want to force a specific implementation. This explicit macro instantiation can make the code less readable. To preserve readability you can instead retain the operator and utilize tool-specific pragmas to map specific HDL operators to specific implementations.

Map to Technology Cells and Optimize

Map “generic” logic functions to cells that are actually available in the technology library, including:

- Set target timing goals
- Map to technology cells to meet timing goals
- Iteratively remap cells on critical paths
- Incrementally reoptimize critical regions

Example

```
synthesize -to_map
```


Mapping the design to technology cells typically includes concurrent optimizations such as restructuring, splitting, pin swapping, buffering, pattern matching, and isolation.

The tool may then execute multiple remapping phases – some targeted at area optimization and others at timing optimization. The remapping operations are predominantly focused upon resizing the cells.

The tool lastly performs an incremental optimization to improve timing and area and fix design rule (DRC) violations. Timing by default is the highest priority. Incremental optimization can also include iterative resynthesis of fragments of the critical path to improve timing.

Technology-Dependent Optimization

Signal	Formula	Gate	Transistors
d	(a+b)'	nor	4
e	d'	not	2
f	(ab)'	nand	4
g	(ec)'	nand	4
h	f'	not	2
i	g'	not	2
j	(h+i)'	nor	4
k	(h+d)'	nor	4
y	j'	not	2
z	c'k+ck'	xor	8

Implementation of a full adder:

y = ab + ac + bc

z = ab'c' + a'bc' + a'b'c + abc

- A design is technology dependent if implemented as interconnected technology library cells.
- Advantage: Library cells are predefined and highly optimized – area and delay are known, minimal, and accurate.
- Technology-dependent optimizations include:
  - Boundary optimizations.
  - Register retiming.

cadence

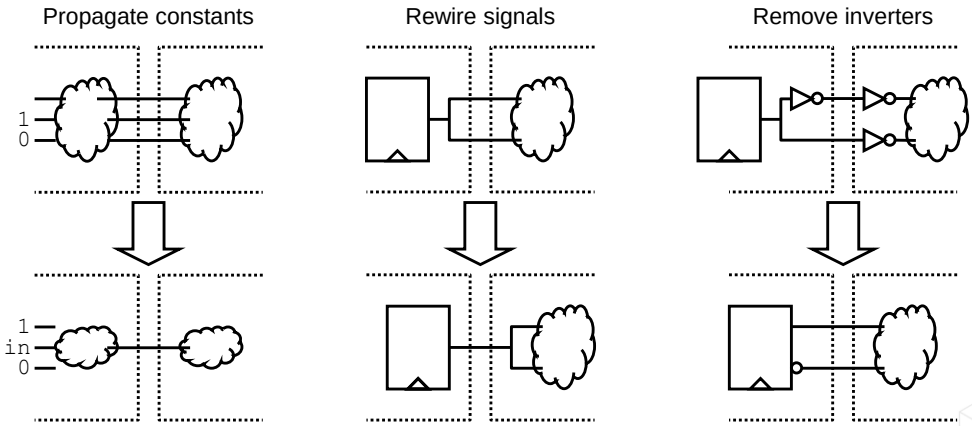
The predefined library cells are highly optimized – area and delay are known, minimal, and accurate.

With and after mapping the design to technology cells, the tool can perform technology-dependent optimizations, including:

- Boundary optimization
- Register retiming

Boundary Optimizations

- Remove gates driving or loading unconnected ports.
- Propagate constants across the hierarchical boundary.
- Rewire equivalent hierarchy-crossing signals.
- Remove inverters duplicated across the hierarchical boundary.



337

cadence

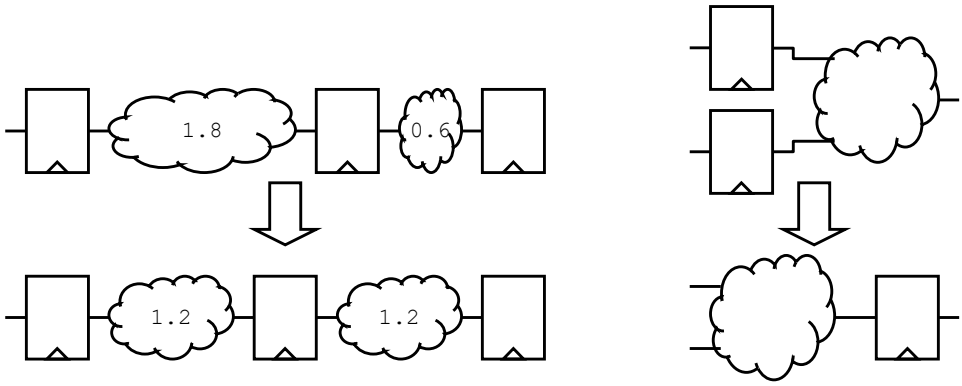
Boundary optimizations include:

- Removing gates driving or loading unconnected ports.
- Propagating constants across the hierarchical boundary.
- Rewiring equivalent hierarchy-crossing signals.
- Removing inverters duplicated across the hierarchical boundary.

Some of these, e.g., constant propagation, the synthesis tool can also do before technology mapping.

Register Retiming

Move registers upstream / downstream to improve results.



Retiming is a technique for improving the performance of sequential circuits by repositioning registers to reduce the cycle time or the area without changing the input-to-output latency. This technique is generally used in datapath designs. The retiming operation distributes the sequential elements at the appropriate locations to meet performance requirements. Thus, retiming allows you to improve the performance of the design during synthesis without having to redesign the RTL. Retiming does not change or optimize the existing combinational logic.

Improving the clock period or timing slack is the most common use of retiming. This can be a simple pipelined design, which contains the combinational logic describing the functionality, followed by the number of pipeline registers that satisfy the latency requirement. It can also be a sequential design that is not meeting the required timing. Retiming distributes the registers within the design to provide the minimum cycle time. The number of registers in the design before retiming may not be the same after retiming because some of the registers may have been combined or replicated.

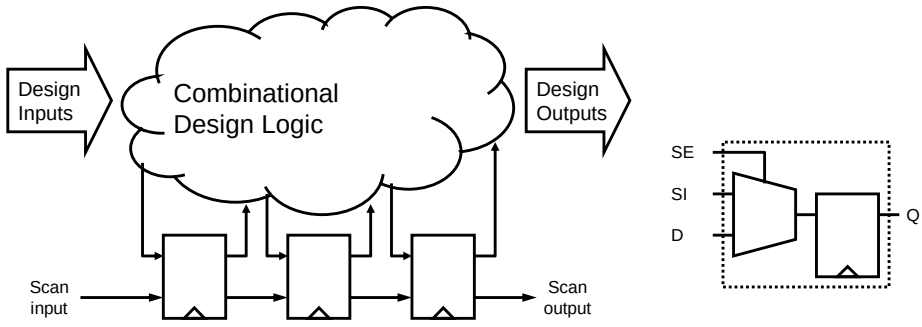
Retiming does not optimize combinational logic and hence the combinational area remains the same. When retiming for area, synthesis moves registers in order to minimize the register count without worsening the critical path in the design.

Retiming is typically not part of the generic flow. You inform the tool which hierarchies or blocks you think would benefit from retiming during synthesis. You can alternatively manually launch a retiming session on a specific block or block region.

If an equivalence checker is part of your flow then you need to inform the equivalence checker about the retimed registers.

Insert Scan

- Internal scan replaces each latch and flip-flop with a scan-equivalent unit.
- Scan-equivalent unit has multiplexor with scan enable and scan input.
- Test mode loads and unloads the scan “chain” serially.
- Makes every latch and flip-flop input and output a virtual I/O pin.
- Reduces test “problem” to just combinational logic.



339

cadence

Manufacturers test ASICs using automated test equipment (ATE), which presents stimulus vectors and clocks to the ASIC inputs and compares the ASIC outputs to expected response vectors.

To improve testability, ASIC developers typically insert internal scan. Internal scan replaces every sequential unit with a functionally equivalent unit having a multiplexed scan data input, and “stitches” the units together to form one or more scan “chains”. This greatly facilitates the automatic test-pattern generation (ATPG) of test vectors, as the tool has absolute control over every sequential unit and needs only to develop combinational test vectors.

Logic synthesis tools automatically insert and connect internal scan. They use ASIC vendor scan macros where available and otherwise build the scan from whatever macros are available. As the scan units are somewhat larger and slower, the logic synthesis tool reoptimizes the design after inserting scan.

Analyzing Results

Command	Description
report area	Area of the synthesized and mapped design
report clocks	Clocks of current design
report datapath	Inferred datapath operators
report design_rules	Design rule violations
report gates	Technology cells used, total area, instance count
report hierarchy	Design hierarchy information
report instance	Instances of current design
report memory	Platform memory used
report messages	Error message summary
report power	Leakage power
report qor	Quality of results
report timing	Timing of current design
report summary	Area, timing, and design rule violations

cadence

Logic synthesis tools report pretty much anything you want to know about the design in a wide selection of formats. This table suggests some of the more obvious things that you might want to report.

Report Timing

The timing report presents arrival time calculations for the critical path.

It reports the path as a series of instance pins with:

- Fanout for each output pin.
- Load and slew for each output pin.
- Incremental delay for each cell.
- Cumulative delay (or arrival time) at each cell.

```
============================================================
  Generated by:           Encounter(r) RTL Compiler v06.20-s019_1
  Generated on:           Jan 23 2008  11:05:20 AM
  Module:                 my_design
  Technology libraries:   lsi500k
  Operating conditions:   lsi_ss_1p08v_125c (balanced_tree)
  Wireload mode:          top
============================================================

         Pin                    Type      Fanout Load Slew Delay Arrival
                                            (fF) (ps)  (ps)    (ps)
------------------------------------------------------------------------
(clock TST_CLK)               launch                                0 R
(DSP.sdc_line_1512)           ext delay                  +8000    8000 F
TST_DIN[0]               <<<  in port         1  1.8 1000   +0    8000 F
p214748365A883/A                                          +0    8000
```


A timing report by default presents detailed information concerning the one most critical path. You can optionally request information for a number of the most critical paths, filter paths by start point, through points and end point, or specify the exact path to report. The example report is that provided by the Cadence Encounter RTL Compiler.

Write Netlist

Write a (usually Verilog) netlist, most often for downstream tools.

Options can include:

- Whether to write the full netlist or just the header.
- Whether to write subdesigns or just the current scope.
- Whether to include tool-specific information for downstream tools.

Example

```
write_hdl
```


Write a netlist. The netlist is almost always a Verilog netlist. The netlist is usually meant for the next tool in the flow, for example verification tools and a place and route tool. Options might include:

- Whether to write the full netlist or just the header.
- Whether to write subdesigns or just the current scope.
- Whether to include additional tool-specific information for the downstream tools.

Module Review

1. Suggest some different general ways that the synthesis tool might implement mathematical operators.
2. Suggest reasons why you might want to use mathematical operators rather than directly instantiate a technology vendor's library macrocell.

cadence

This page does not contain notes.

Module Review Solutions

1. Suggest some different general ways that the synthesis tool might implement mathematical operators.
   - The synthesis tool may instantiate an appropriate macrocell that implements the operator, or it may implement the operator using discrete gates from the target technology library.
2. Suggest reasons why you might want to use mathematical operators rather than directly instantiate a technology vendor's library macrocell.
   - Directly instantiating macro-cells will make the code less readable, and dependent upon the technology that provides the macro-cell.

cadence

This page does not contain notes.

Module Exercise

Logic synthesis cannot add states to your design. Suppose that you know that the sum is not needed until two cycles later. Re-code this design to use two states and one adder. Assume that the inputs remain stable relative to clocks.

```
always @(posedge clk)
  if (do_add)
    sum <= a + b + c;
```

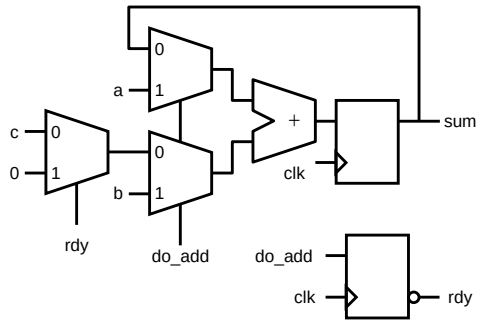

cadence

This page does not contain notes.

Module Exercise Solution

Logic synthesis cannot add states to your design. Suppose that you know that the sum is not needed until two cycles later. Re-code this design to use two states and one adder. Assume that the inputs remain stable relative to clocks.

```
always @(posedge clk)
  if (do_add) begin
      sum <= a + b;
      rdy <= 0;
    end
  else
    if (!rdy) begin
        sum <= sum + c;
        rdy <= 1;
      end
```



cadence

This page does not contain notes.

Labs

There are no labs in this module.

cadence

This page does not contain notes.