ECE 351 Syllabus for Spring 2019 Qtr

Instructor: Dr. Garrison Greenwood
email: greenwd@pdx.edu
Office: FAB 20-12
Office hours: W 1500–1700

Prerequisites: ECE 172 (or equivalent)

Course Objectives:

Students must demonstrate the ability to

- describe complex digital designs in the Verilog HDL.
- design testbenches needed to test digital designs
- describe those language syntax that promotes efficient synthesized designs
- apply design automation tools to synthesize designs in FPGAs
- describe good coding practices for efficient FPGA synthesis
- describe how embedded systems are implemented in FPGAs
- be familiar with timing closure and how it is accomplished

Textbook: https://www.oreilly.com/library/view/verilog-hdl-a/0130449113/

1. The course will be supplemented with material, which will be posted on D2L. Supplemental material is testable.
2. Your grade will be based on the two one-hour tests (25 points each), homework (20 points) and a final exam (30 points).
3. The grade scale is

A	$\rightarrow$	≥ 90 pts
B	$\rightarrow$	80–89 pts
C	$\rightarrow$	70–79 pts
D	$\rightarrow$	60–69 pts
F	$\rightarrow$	< 60 pts

4. This class uses a Verilog simulator (Questa-sim) and a synthesizer design suite (Vivado) , which are installed on the PC network in the circuits lab in FAB. D2L contains a complete tutorial on their use.
5. This class uses a FPGA board which has a Xilinx FPGA chip on it. The boards are in the circuits lab in FAB. A tutorial on their use will be posted on D2L

6. General Notes:

- No curve is used in determining grades.
- Any form of cheating will not be tolerated. *If you cheat on an exam, you get a score of zero on that exam.* Character does matter.
- As a courtesy to me and your fellow students, turn off all cell phones before you enter the classroom.
- Homework is assigned each week and is due the following Thursday. All assignments will be posted on D2L.
- Homework is due at the end of the class period. Late homework is not accepted. You can turn in your homework early at the ECE office *but you must get it timestamped.* No timestamp and I consider it late.
- Do not email me your homework; it must be submitted in hardcopy form. Homework submitted directly to the TA will not be graded.
- Some homework assignments will require you to simulate and/or synthesize the design in an FPGA and demonstrate it to the class TA. These demos must be individual efforts (no team demos allowed.) For those assignments requiring demos you must successfully complete the demo or you will not get any points for that homework assignment.
- Late homework is not accepted.
- There are no extra-credit assignments of any kind.
- I introduce Verilog syntax in a specific order. Do not work ahead. If you use a syntax in the homework or lab report or quiz that I have not introduced in class, I will deduct points—*even if the syntax you used was correct.*
- Accommodations are collaborative efforts between students, faculty, and the Disability Resource Center. Students with accommodations approved through the DRC are responsible for contacting the faculty member in charge of the course prior to or during the first week of the term to discuss accommodations. Students who believe they are eligible for accommodations but who have not yet obtained approval through the DRC should contact the DRC immediately. If you are already registered with DRC, I need to see you ASAP to discuss your accommodations.
- Exams/tests taken by students registered with DRC should be taken at the same time as the regular class.
- The two one-hour tests are scheduled for 25 April and 23 May (both on Thursdays). The final exam is on Monday, 10 June from 1015–1205.
- Be sure to check D2L frequently as new material is added from time to time. I will also use D2L for course announcements.

Teaching Approaches

Approach #1

- Teach students HDL programming (Verilog or VHDL)
- Teach students how to simulate HDL code
- Teach students how to program FPGAs
- Rest of course focuses on doing "science fair" programming projects or maybe a team-based large programming project.

Emphasis is on developing HDL programming skills

Teaching Approaches

Approach #2

- Teach students HDL programming
- Teach students how to simulate HDL code
- Teach students how to program FPGAs
- Only one or two homework problems involve programming FPGAs
- Rest of course is focuses on designing with FPGAs

Emphasis is on developing FPGA-based products.

Approach #1 is really cool. Might even be fun because students get lots of experience programming FPGAs.

Problem with that approach is it does a real disservice to the student!! (It doesn't teach them what industry expects them to know)

Why? Because the original coding of a design and programming the FPGA is only about 20% of what is required to get an FPGA design up and running…

The design specification for an FPGA-based design tells you two things:

- functional requirements
- timing requirements

HDLs only describe functional requirements. They cannot describe anything related to timing.

Timing is not addressed until after synthesis (i.e., after you program the device.)

Suppose your design is programmed into an FPGA and it doesn't meet a timing specification (e.g., it won't work with a required 125 MHz clock). What do you do now???

Some possibilities include

- Choose a larger, faster FPGA
- Try different synthesizer options
- Rewrite your HDL code (most likely solution)
  - change the design partitioning
  - change the level of abstraction in some modules
  - do retiming (i.e., move registers across combinatorial logic boundaries)
  - switch from inferring some modules to instantiation of Hard IP

Other Issues

Exactly how is an FPGA programming?

FPGAs are SRAM based. So how do you program FPGAs embedded in hand-held devices such as smartphones or medical instruments?

What kind of support do FPGA vendors provide?

What is the role of 3rd-party IP?

The issues raised on the previous two slides are usually only glossed over (if taught at all) when approach #1 is used to teach the course.

This means just learning an HDL and how to program an FPGA—i.e., approach #1—teaches you only part of what you need to know. You need to make sure your design meets both functional *and* timing requirements.

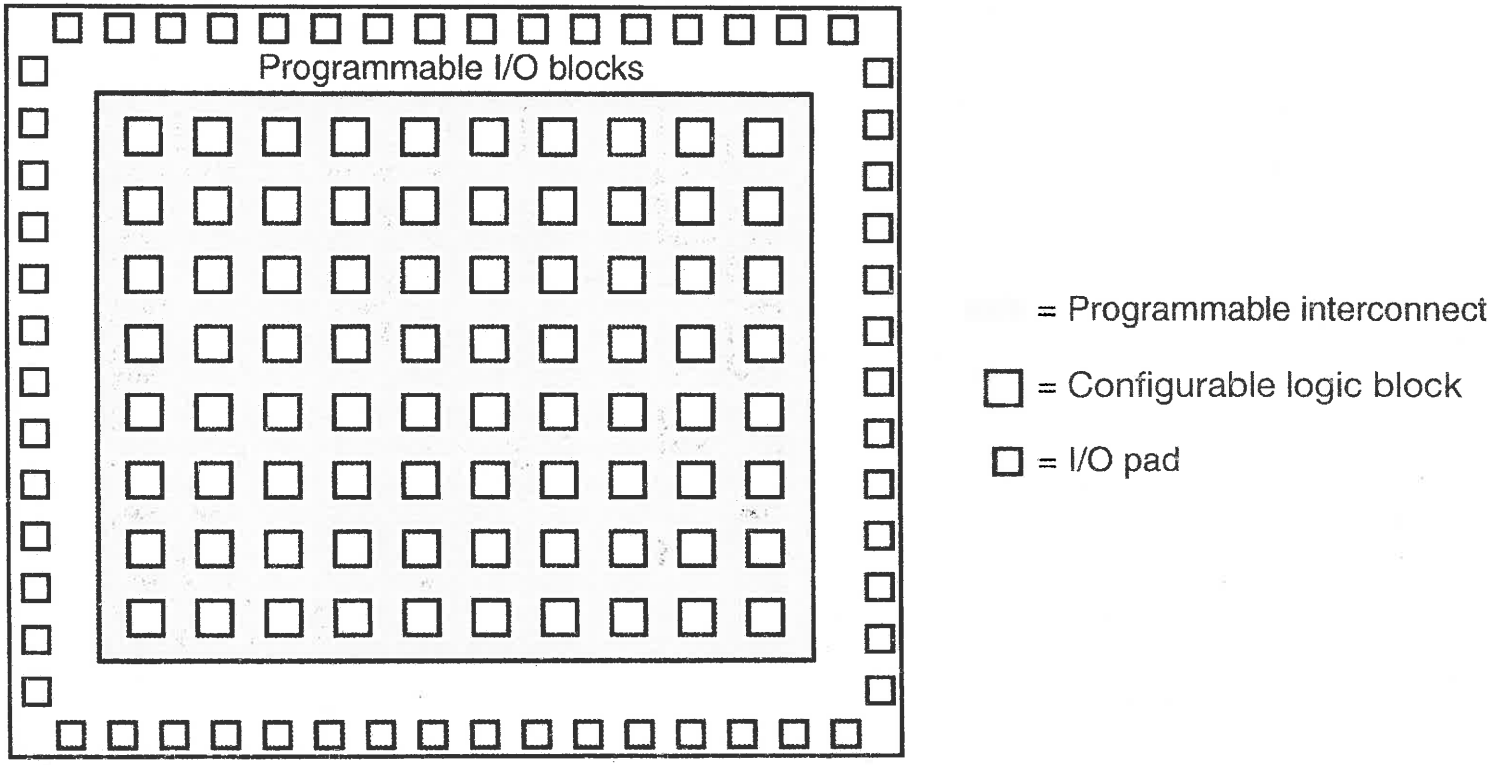
In this class I will spend around 40% of the time teaching the Verilog language. The remaining portion will be spent on what you need to know to build an FPGA-based design.

- **ASIC** (*application specific IC*)

A custom chip designed for a very specific purpose. For example, a chip with a DSP front end designed for a specific model of a cardiac monitoring device mad by only one manufacturer.

- **ASSP** (*application specific standard product*)

A semi–custom chip designed for a specfic purpose, but which can be tailored to individual customer needs. For example, a disk controller chip sold to multiple customers.



General FPGA chip architecture.

2 FPGA applications

- prototyping an ASIC/ASSP design
- target device

ASIC Design Methodology Primer

Abstract

This application note provides an overview of the application-specific integrated circuit (ASIC) design process. Four major phases are discussed: design entry and analysis; technology optimization and floorplanning; design verification; and layout.

Introduction

The *ASIC Design Methodology Primer* provides an overview of the steps involved in application specific integrated circuit (ASIC) design. An ASIC is a collection of logic and memory circuits on a single silicon die. ASICs are used in a wide variety of products ranging from consumer products such as video games, digital cameras, automobiles and personal computers, to high-end technology products such as workstations and supercomputers. The ASIC market, with steady growth over the past decade and continued growth predicted for the next one, is expected to become a $50 billion market by the year 2000 (*Dataquest*, 12/16/96).

This primer is organized into three sections:

- The first section, **Basic Terminology**, defines key terms and the scope of this paper.
- The second and largest section, **Basic Methodology Walkthrough**, covers, at a high level, the four major phases of ASIC design, and is illustrated with real design examples. This discussion also identifies some of the major software vendors who offer ASIC design tools, and lists any process steps unique to IBM.
- The last section, **Design Challenges and Strategies** summarizes the specific strengths and capabilities IBM ASICs brings to the marketplace, and their resulting value to its customers.

Basic Terminology

ASICs are logic chips designed by the end-customers to perform a specific function and thereby meet the specific needs of their application. Customers implement their designs in a single silicon die by mapping their functions to a set of predesigned, preverified logic circuits provided by the ASIC vendor. These circuits are referred to as the ASIC vendor's **library**, and are described in the ASIC vendor's **databook**. These circuits range from the simplest functions, such as inverters, NANDs and NORs, flip-flops and latches, to more complex structures such as static memory arrays, adders, counters and phase-lock loops. Recently vendors have added some highly complex circuits to their ASIC libraries, such as microprocessors, Ethernet® functions, and peripheral component interconnect (PCI) controllers. These complex designs are referred to as **cores** and are fast becoming a major differentiator among ASIC vendors.

ASIC Vendor Selection Criteria

An ASIC designer, seeking to create a new design and select an appropriate ASIC vendor, should consider the following criteria:

- ASIC library content and characteristics:
  - Does the library contain the logic circuits needed to implement the design? Are the circuits fast enough? How many can fit on a single die?

- Design turn-around-time (TAT):
  - How long does the ASIC vendor take to fabricate, package, and test the part once the design is completed?
- Price of the die:
  - How much does the ASIC cost?
    This is an important factor to all designers, but is more crucial to some customers than others. Those in the consumer market may have this as their number one criteria when evaluating an ASIC vendor, whereas a high-end workstation customer may put performance or function ahead of price.
- Power consumption:
  - How much power does the ASIC consume?
    The importance of power utilization has greatly increased over the past several years, and surpasses the importance of cost in some cases, such as in battery-powered applications like cell phones and lap-top computers.
- Miscellaneous aspects:
  - Packaging options, reliability, supply assurance and second-source capabilities are absolutely critical to some customers, and of secondary importance to others.
- Design methodology.
  - Design methodology is the process that a designer must follow to implement a design in an ASIC vendor's library. The ease with which a designer can execute this process can affect time-to-market, design verification and reliability, and the cost of the overall design process. It is this aspect of the ASIC product, **design methodology**, that is the focus of this primer. This criteria is of importance to all ASIC customers.

Design Views

During the course of the design process, the design data exists in several different formats or views. As the design progresses, it becomes less abstract and more specific to, and optimized for, a particular technology. Each step in the design methodology serves a different purpose and requires unique tools.These views evolve through three major phases:

- In the initial phase the design is realized primarily as a technology-independent Hardware Description Language (HDL), a format very similar to a programming language, to describe the design's functionality.
- In the second phase the design is realized as a technology-dependent netlist that consists of a series of instances of circuits from the ASIC vendor's library, interconnected in a manner to implement the functionality described in the previous view.
- In the last phase the design is realized as a physical view, in which the logic circuits described in the previous view are physically placed on a piece of silicon, called a die, and interconnected by various layers of wiring.

Figure 1 on page 3 depicts these three design views.

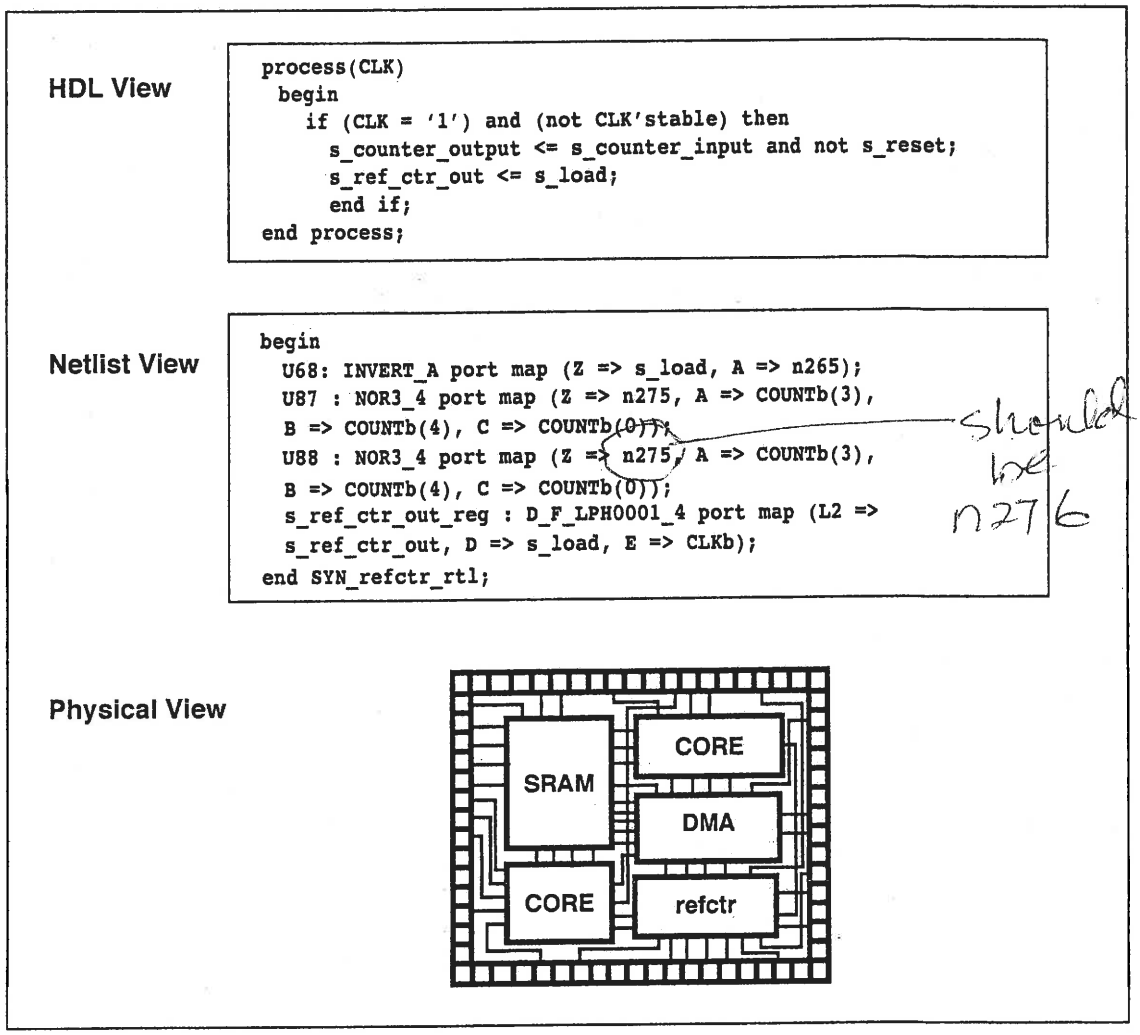


Figure 1. Design Views

Basic Methodology Walkthrough

There are four basic steps that an ASIC design must go through in order to create working silicon:

1. design entry and analysis
2. technology optimization and floorplanning
3. design verification
4. layout

Design Entry

The designer's first task is to describe the design's intended function. Typically this functionality is specified in a document, such as a functional specification, written in a natural language such as English in order to facilitate its development as well as to make it accessible for review by all project team members. Once the specification is finalized, the designer then translates the specification into a form that can be

understood by software tools in order to direct the creation of silicon. The two principal design description methods are:

- Hardware Description Languages (HDLs), generally used for designs of 50 thousand gates or more; and,
- Schematic Capture, an older method, suitable only for sub-50k gate designs and generally less often used today.

The two dominant HDLs are Verilog® and VHDL. Both are entered using a text editor such as *vi* on a UNIX®-based workstation. Verilog and VHDL are languages much like programming languages, such as C or Pascal, but they have been designed specifically for describing hardware behavior. Verilog and VHDL are functionally equivalent. The choice of one over the other is driven primarily by the experience base of the design group, the tool set available to the designers to process the HDL, and, possibly, by organizational dictates, such as those of the US government, which requires that all designs be written in VHDL. Verilog dominates the US merchant ASIC market, whereas VHDL prevails in Europe, the US government, and some large US companies such as IBM.

HDLs allow designers to describe the function of their designs at a high level, often independent of the eventual implementation in silicon, much as a programmer can describe a function in the C language without knowing the specific compiler that will create the executable object code.

With schematic capture, graphical representations of the logic functions are placed on a computer screen and are manually connected by the designer. Schematic capture requires the designer to enter a much lower-level description of the design, implemented directly in the logic circuits available from the ASIC vendor, thereby sacrificing the flexibility of the higher-level description possible with HDLs. Schematic capture may still prevail for some time with very small ASICs (10–40k gates) or those containing analog functions. With the average size of an ASIC in the United States in 1996 exceeding 100k gates, the vast majority of customers will be using VHDL or Verilog as their design entry vehicle. (*Dataquest*).

Design Entry Examples

The following sections provide a brief look at some examples of HDL and a simple schematic.

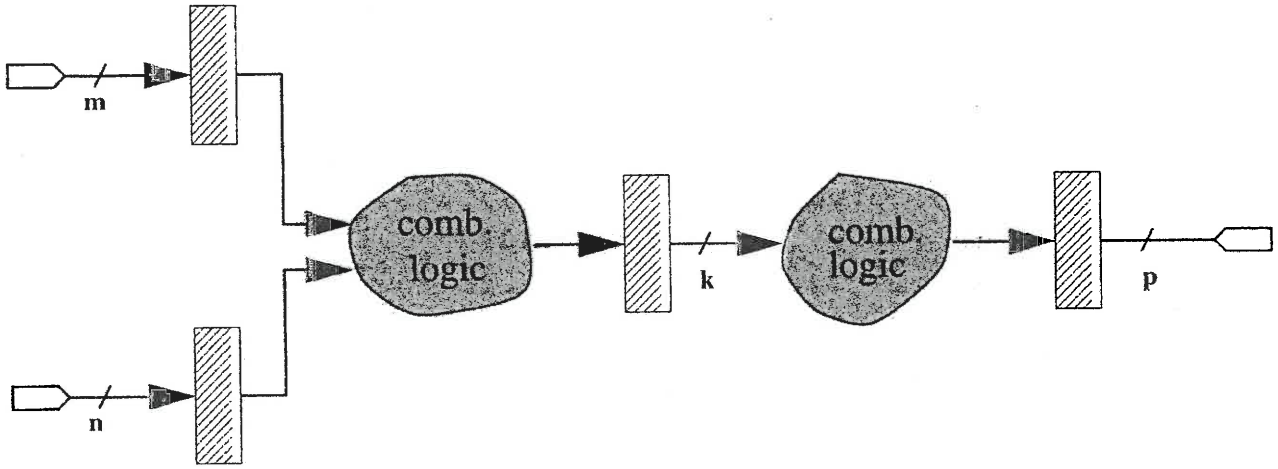
Sample High-Level Hardware Description Language (HDL)

Figure 2 on page 5 contains a portion of a direct memory access (DMA) controller written in two different HDLs: VHDL and Verilog. Notice that though there are syntactical differences between the two languages (for example, VHDL's "entity DMA1..." versus Verilog's "module DMA1..."), the types of language statements and level of description are essentially equivalent. Both HDLs have execution control statements based on the state of a signal called CLK, and both propagate certain design values based on the status of CLK. The language statements are independent of any particular ASIC vendor's library and are at a level of abstraction above any particular logic circuit implementation; for example, such statements might be at a behavioral level or register transfer language (RTL) level. Whatever the level, an HDL can be implemented in several different ways, using different combinations of circuits from any one of a number of different ASIC vendors' libraries.

Sample Schematic

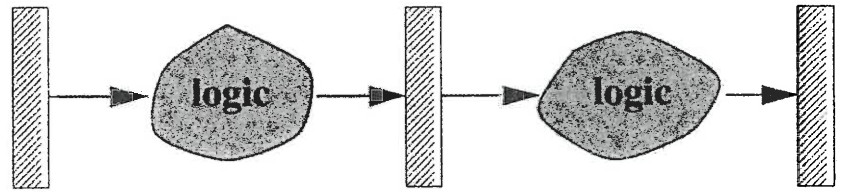
Figure 2 contrasts sharply with Figure 3 on page 5, which provides a schematic representation that is directly mapped into the logic circuits in an ASIC vendor's library. The schematic assembles circuits such as NOR3, AND2 and INVERT and includes explicit connections between inputs and outputs.The logic circuit implementation for this function is completely defined. Because the vast majority of ASIC designs

Definition of Register Transfer Level



- inputs & outputs defined
- registers serve as memory elements
- combinational logic between registers processes data
- circuit specified by operations and data transfers between registers
- defined timing (i.e., operations occur at specified times)

THE RTL MODEL



$$\text{reg2} \leftarrow f_1(\text{reg1})$$

$$\text{reg3} \leftarrow f_2(\text{reg2})$$

These are variables in Verilog

- combinational logic implements the functions
- RTL model shows data movements, but no details about the control structure*

* the control structure generates signals such as decoder enables or mux select line inputs

done at IBM begin with an HDL description rather than schematic entry, this paper focuses primarily on HDL in the analysis phase.

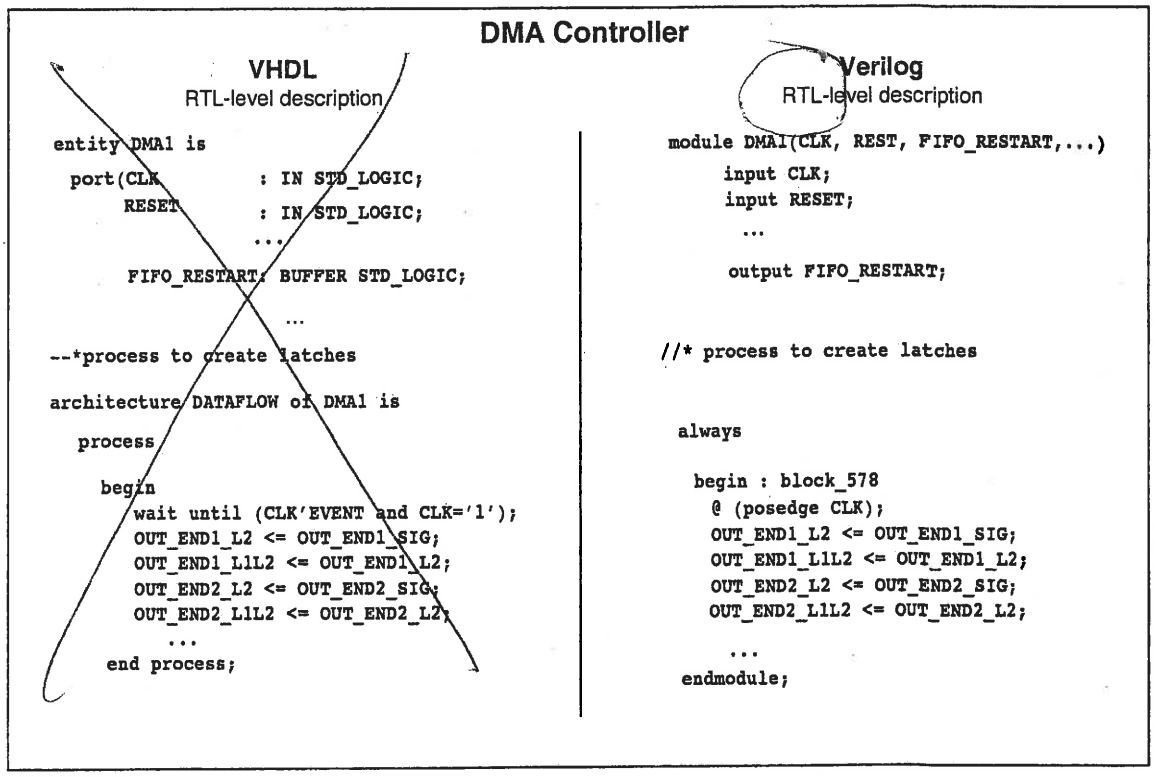


Figure 2. DMA Controller with Two Different HDLs

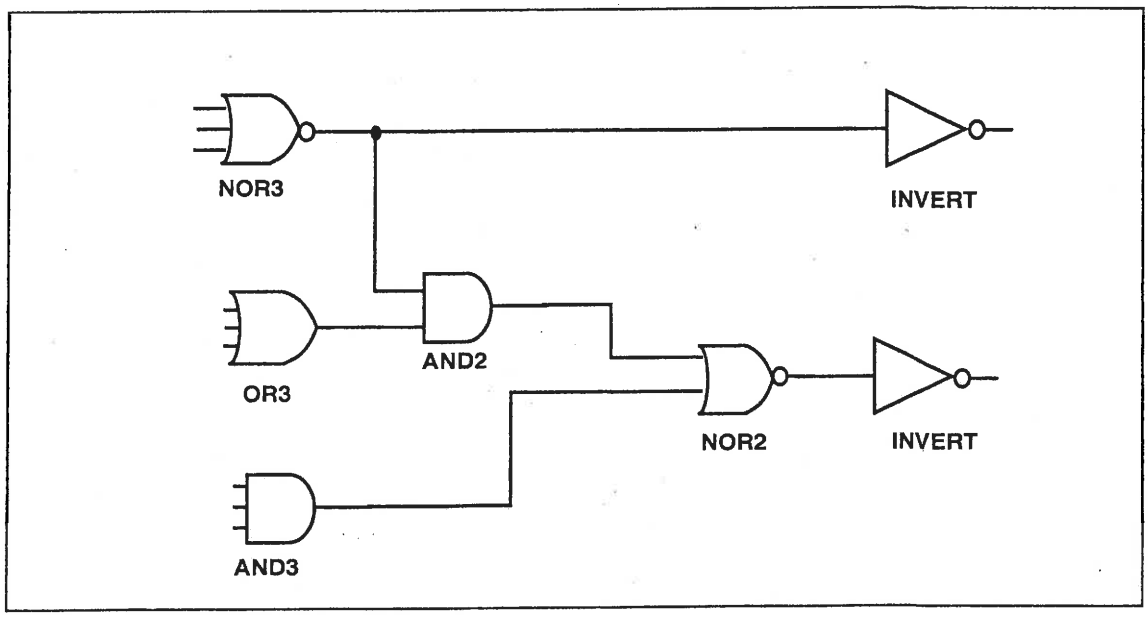
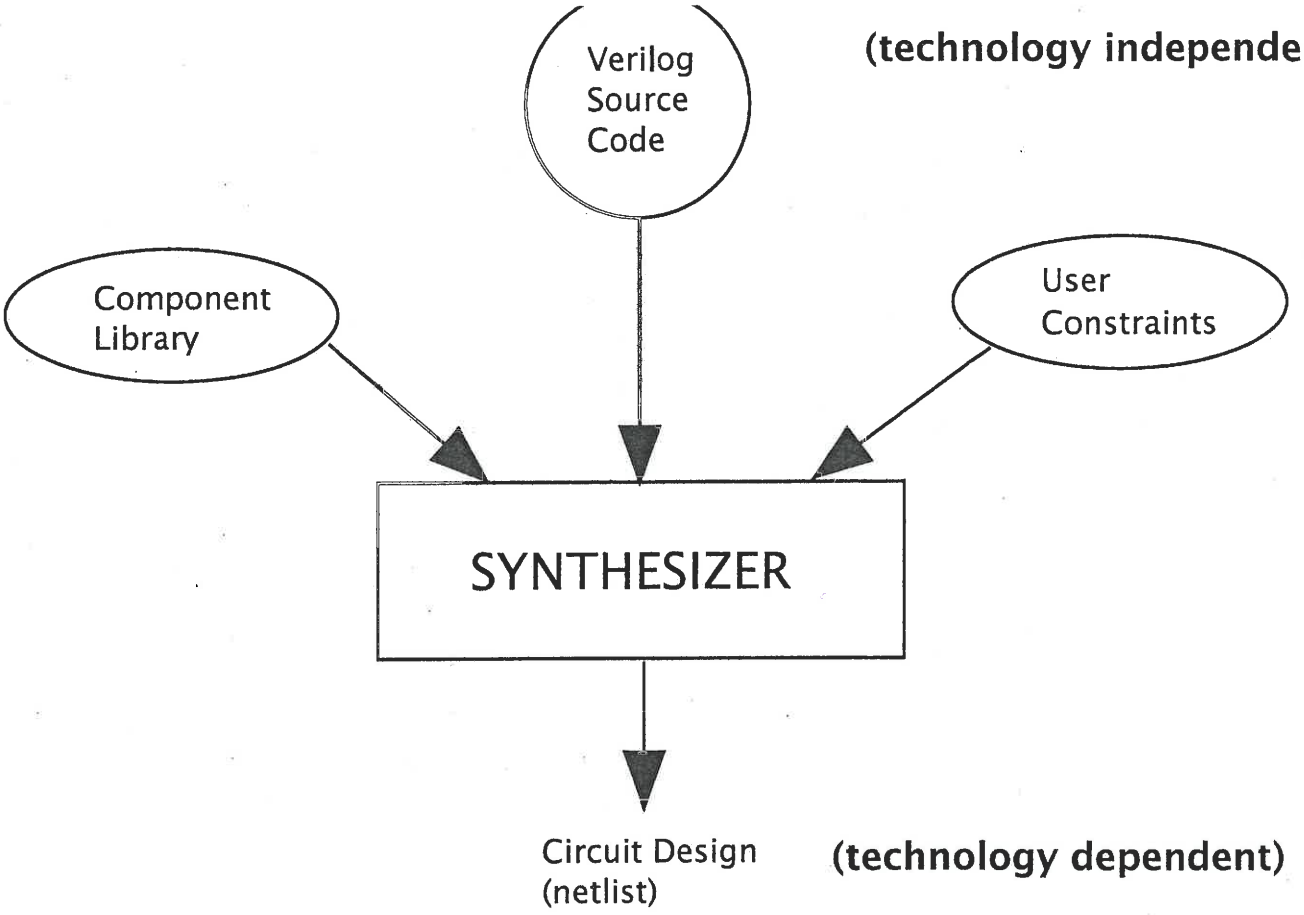


Figure 3. Schematic Representation of Logic Circuits in an ASIC Library


Verilog
Source
Code
(technology independent)
Component
Library
User
Constraints
SYNTHESIZER
Circuit Design
(netlist)
(technology dependent)

Design Analysis

After entering a design in an HDL, the designer begins the process of analyzing what was entered to determine if it correctly implements the intended function. The traditional method is through **simulation**, which evaluates how a design behaves. Simulation is a mature, well-understood process, and there are many simulators available that accept HDLs written in VDHL, Verilog, or increasingly, both languages. IBM ASICs supports many different simulators available from CAD vendors, such as Verilog-XL™ and Leapfrog™ from Cadence; VSS™ from Synopsys; and MTI™ from Model Technology, Inc.

A more recent addition to the design analysis phase is **power analysis**, with many new CAD tools coming to market in the last year. For a growing number of customers, the power consumption and dissipation of their designs are becoming critical factors. Early feedback on the power requirements of a design allows designers to make timely basic design trade-offs in order to achieve power targets. Because this analysis is at the architectural level and is technology-independent, the estimates may not be extremely accurate and may vary as much as 50% from the actual silicon implementation.

Simulation

Figure 4 represents the traditional simulation process. The VHDL or Verilog, which describes the design function, is read into the simulator tool along with a set of **input vectors** created by the designer. The simulator generates **output vectors** that are captured and evaluated against a set of expected values. If the output values match the expected values, then the simulation passes; if the output values differ, then the simulation is said to fail and the design needs to be corrected. Most simulators generate output in two forms: numerically, as 0's and 1's in a file for comparison purposes, and graphically, as waveforms that depict the transition of signals from 0 to 1 and vice-versa.

Note that the simulation at this level is technology-independent. There is usually little or no technology-specific information delivered by an ASIC vendor to support simulation at this phase. Exceptions include high-level behavioral models for large macros such as RAMs, ROMs, or complex cores.

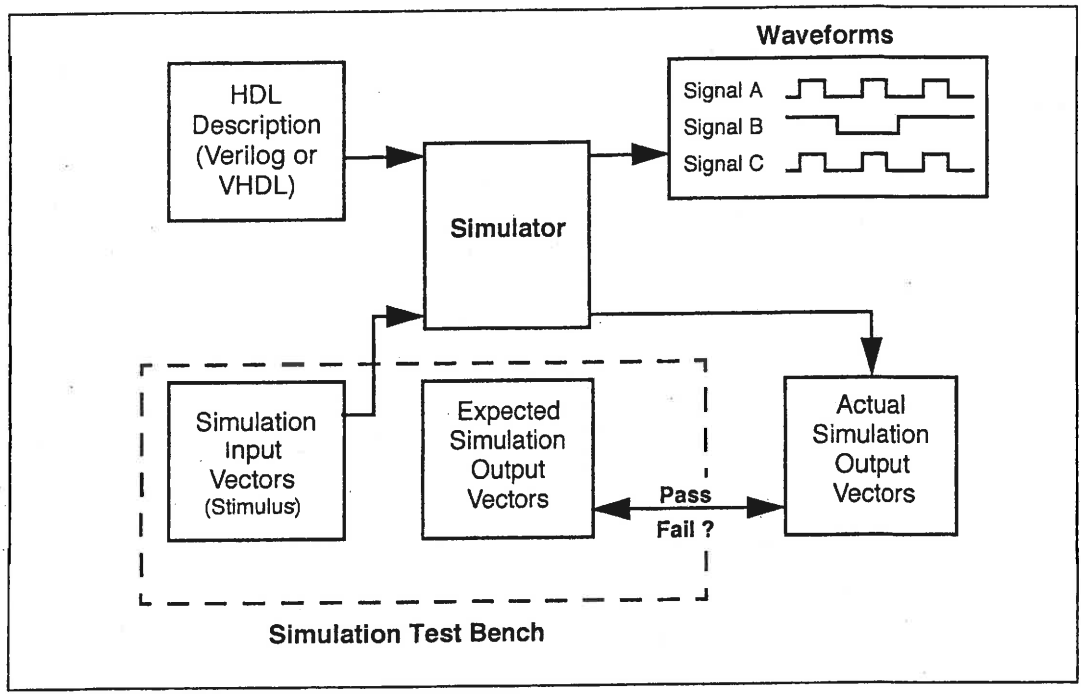


Figure 4. Traditional Simulation Process

Technology Optimization

Technology optimization takes a technology-independent description of a design, and maps it to a library of logic circuits provided by an ASIC vendor, thereby making the design technology-dependent. This phase seeks not just a correct mapping, but the most efficient one in terms of the customer requirements. The optimization process is divided into subprocesses: logic synthesis; test insertion; clock planning and insertion; and floorplanning.

Logic Synthesis

Logic synthesis is the basic step that transforms the HDL representation of a design into technology-specific logic circuits. An ASIC vendor provides the logic circuits in a form called a "synthesis library". As the synthesis tool breaks down high-level HDL statements into more primitive functions, it searches this library to find a match between the functions required and those provided in the library. When a match is found, the synthesis tool copies the function into the design (instantiates the circuit) and gives it a unique name (cell instance name). This process continues until all statements are broken down and mapped (synthesized) to logic circuits. There are potentially hundreds, or even thousands, of different combinations of logic circuits that can implement the same logical function. The combination chosen by a synthesis tool is determined by the synthesis constraints provided by the designer. These constraints define the design's performance, power, and area targets. A design driven primarily by performance criteria may use larger, faster circuits than one driven to minimize area or power consumption. Synthesis has matured over the past 5–8 years in the merchant market and is used in virtually all ASIC design starts today.

The inputs to the logic synthesis process are the HDL design description (VHDL or Verilog), the design constraints, and the synthesis library provided by the ASIC vendor. The output of the synthesis process is a list of circuit instances interconnected in a manner that implements the logical function of the design. This list of interconnected circuit instances is called a **netlist** and can be written in several different formats or languages. The dominant netlist languages are VHDL, Verilog, and Electronic Design Interchange Format (EDIF). The interconnected circuits may also be graphically represented as schematics.

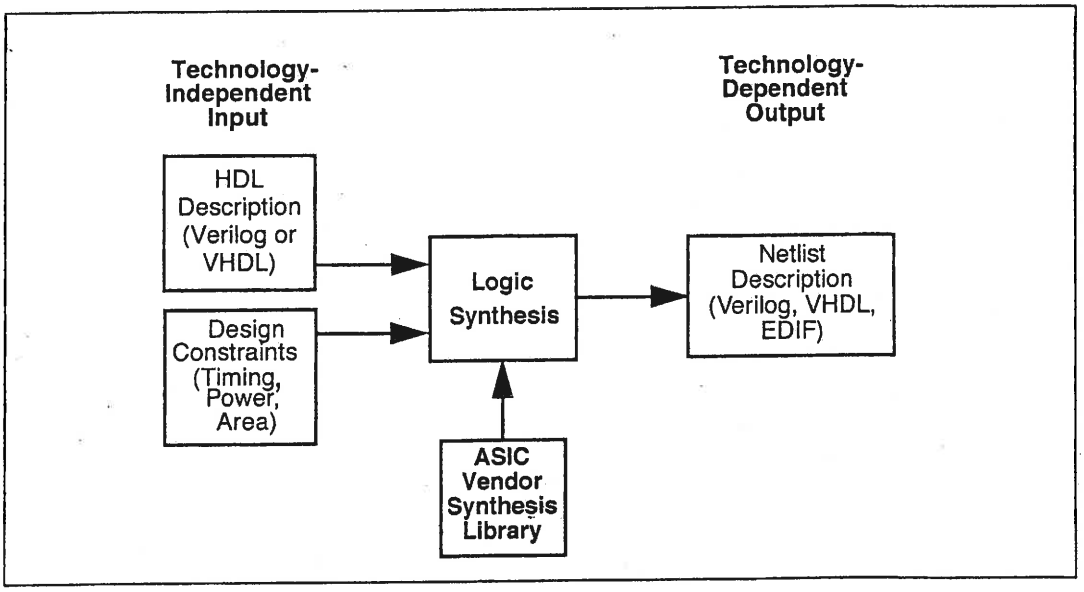


Figure 5. Logic Synthesis Process

The most popular synthesis tool in the external market, accounting for about 85% of the total synthesis

Netlist Gate-Level View of refctr (EDIF)

Electronic Design Interchange Format

The EDIF version of the netlist also contains the exact same information as the schematic, VDHL and Verilog versions in terms of the circuits and their connectivity. The difference is, again, syntactical. EDIF is also more verbose than either VHDL or Verilog, and the data volume of an EDIF netlist is a drawback; nonetheless, EDIF is an industry standard and is accepted by almost every electronic design automation (EDA) tool on the market.

EDIF

```
(cell refctr (cellType GENERIC)
   ...
(contents
   (instance U68
    (viewRef Netlist_representation
     (cellRef INVERT_A(libraryRef IBMCMOS5S_SC))
    )
 )
 (instance U87
  (viewRef Netlist_representation
   (cellRef NOR3_4(libraryRef IBMCMOS5S_SC))
  )
 )

(instance U88
 (viewRef Netlist_representation
  (cellRef NOR3_4(libraryRef IBMCMOS5S_SC))
  )
)
(instance U89
 (viewRef Netlist_representation
  (cellRef AND2_8 (libraryRef IBMCMOS5S_SC))
 )
)

(instance s_ref_ctr_out_reg
 (viewRef Netlist_representation
  (cellRef D_F_LPH0001_4
   (libraryRef IBMCMOS5S_SC))

 )
)
```

EDIF (continued)

```
(net s_load
  (joined (portRef A (instanceRef U74))
   (portRef D (instanceRef s_ref_ctr_out_reg))
   (portRef Z (instanceRef U68))
  )
)
(net CLK
  (joined (portRef CLK)(portRef E (instanceRef
    (s_ref_ctr_out_reg))
    (portRef E (instanceRef s_counter_output_.....
      ...
(net s_ref_ctr_out
  (joined (portRef D0 (instanceRef U90))
    (portRef L2 (instanceRef s_ref_ctr_out_reg))

  )
)
(net 275
  (joined (portRef A (instanceRef U89)) (portRef Z
    (instanceRef U87)))
)
(net 276
  (joined (portRef B (instanceRef U89)) (portRef Z
    (instanceRef U88)))
)
(net 277
  (joined (portRef SD instanceRef u90)) (portRef Z
    (instanceRef U89)))
  )
)))))
```

Figure 10. Gate-Level Netlist View of refctr - EDIF

Test Insertion

Test insertion, the step following logic synthesis, consists of inserting structures into the design to enable a complete and efficient manufacturing test. The IBM ASIC methodology requires that the test structures be inserted in a manner that is compliant with IBM's full-scan design-for-test (DFT) methodology. IBM is a recognized industry leader in DFT, and its incorporation into IBM ASIC flow is an important market differentiator. Compliance with the methodology offers customers significant advantages, such as high-quality test coverage (greater than 99% on average) and automatically-generated test patterns.

Schematic View of refctr

Figure 8 depicts a post-synthesis schematic view of a section of **refctr**. Notice that the design was mapped to specific logic circuit functions, such as INVERT_A, NOR3_4 and D_F_LPH0001_4. These names correspond to circuit names found in the IBM *ASIC CMOS 5S Databook*, SA14-2203-03. Each circuit has a unique name, such as U87 for one instance of NOR3_4, and U88 for another instance of NOR3_4. The instance names U87 and U88 were generated by the synthesis tool as it mapped the HDL function into logic circuits such as NOR3_4.

Signals generated by the synthesis tool as it mapped the HDL to logic circuits appear with names such as n275 and n276. Signal names explicitly named in the HDL, such as sload and CLK, are retained. Notice that sload and CLK feed into a circuit that generates the signal s_ref_ctr_out, as described in the technology-independent source on the previous page (Figure 7).

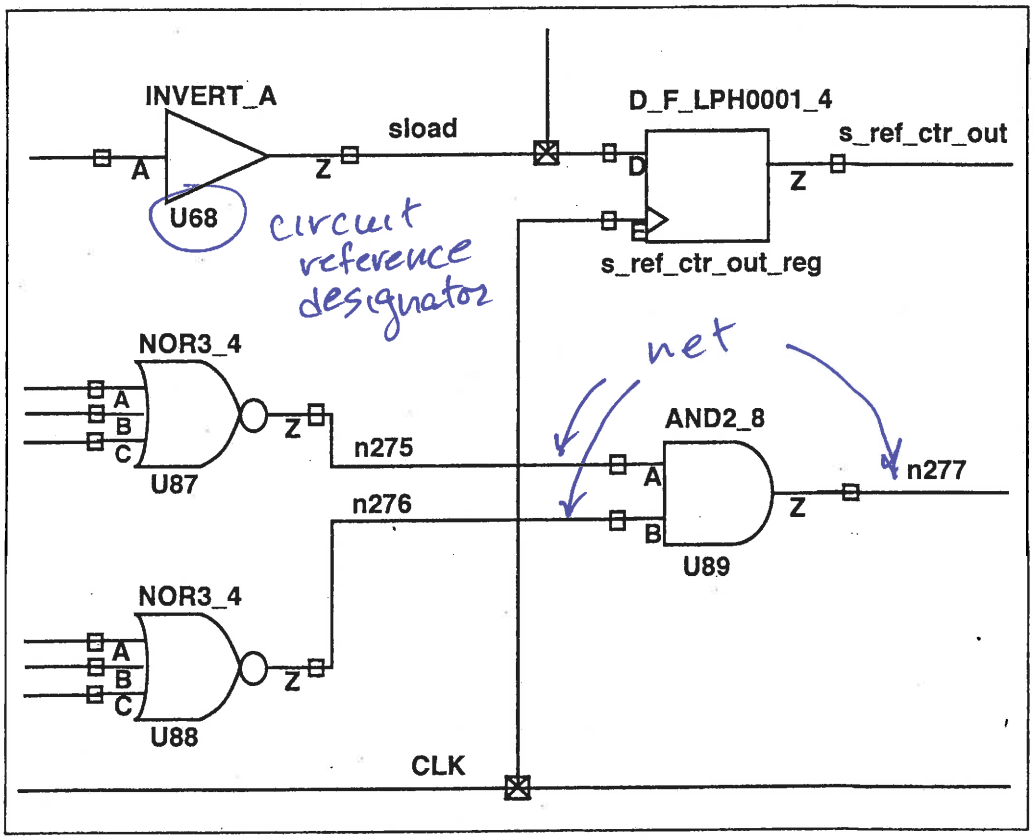


Figure 8. Netlist Schematic View of refctr

Netlist Gate-Level View of refctr (VHDL, Verilog)

Figure 9 contains the post-synthesis netlist of **refctr**, output in both VHDL and Verilog. The circuits described, along with net names and instance names are exactly the same. The difference is in the syntax of the description.

VHDL

```
entity refctr is
    ...
architecture SYN_refctr_rtl of refctr is
    ...
component INVERT_A
 port(Z : out std_logic; A : in std_logic);
end component;
component NOR3_4
 port(Z : out std_logic; A, B, C : in std_logic);
end component;
component AND2_8
 port(Z : out std_logic; A, B : in std_logic);
end component;
component D_F_LPH0001_4
 port(L2 : out std_logic; D, E : in std_logic);
end component;

begin ...
 U68 : INVERT_A port map (Z => s_load, A => n265);
 U87 : NOR3_4 port map (Z => n275, A => COUNT(3),
      B => COUNT(4), C => COUNT(0));
 U88 : NOR3_4 port map (Z => n276, A => COUNTb(5),
      B => COUNT(2), C => COUNT(1));
 s_ref_ctr_out_reg : D_F_LPH0001_4 port map (L2 =>
      s_ref_ctr_out, D => s_load, E => CLK);
end SYN_refctr_rtl;
```

Verilog

```
module refctr (COUNT, CLK, RESET, REF);
  ...
 INVERT_A U68 (.Z(s_load), .A(n265) );
 NOR_4 U87 (.Z(n275), .A(COUNT[3]),
  .B(COUNT[4]), .C(COUNT[0]) );
 NOR3_4 U88 (.Z(n276), .A(COUNT[5]),
  .B(COUNT[2]),.C(COUNT[1]) );
 AND2_8 U89 (.Z(n277),.A(n275),
  .B(n276) );
 D_F_LPH0001_4 s_ref_ctr_out_reg(
  .L2(s_ref_ctr_out), .D(s_load), .E(CLK));
  ...
endmodule;
```

Figure 9. Gate-Level Netlist View of refctr - VHDL/Verilog

A typical design flow for designing VLSI IC circuits is shown in Figure 1-1. Unshaded blocks show the level of design representation; shaded blocks show processes in the design flow.

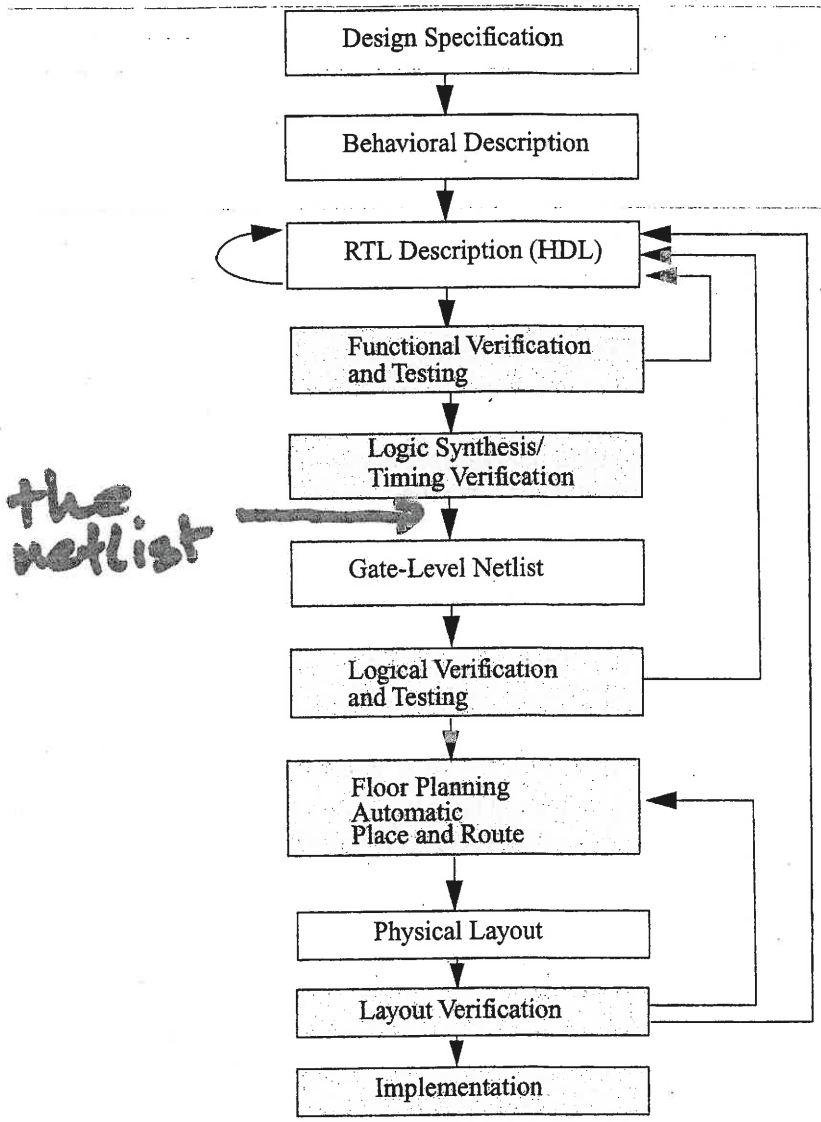


Figure 1-1 Typical Design Flow

All programming languages have

- alphabet
- syntax
- semantics

Defn (alphabet)

the symbols that can be used in the program

e.g. (in C)

- Lowercase ltrs { a, b, c, ... }
- uppercase ltrs { A, B, C, ... }
- numbers { 1, 2, 0, 49, ... }
- special symbols { @, . ! # .... }

(same in Verilog)

Defn (syntax)

the rules for combining alphabetic characters into valid program statements

e.g. (in C)

i++	is okay
+i+	in not okay

Defn (semantics)

the meaning of a valid syntactically correct program statement

e.g. (in C)

syntax	semantics
i++;	increment i by 1

in Verilog,

① `/*` ⋮ `*/` multiline comment

or

`//` one line comment

② operators

e.g.

- unary — `A = ~B;` // $A = \overline{B}$
- binary — `Y = A+B;`
- ternary — `a = b ? c : d ;`