

High Level Synthesis for the Digital System Designer

Miriam Leeser and Shantanu Tarafdar

DRAFT: Do not distribute without permission of the authors
Copyright 1998 by Miriam Leeser and Shantanu Tarafdar

Chapter 1

High Level Synthesis: An Introduction

High-level synthesis, also known as behavioral or architectural synthesis, is an exciting branch of computer-aided design for digital systems that has emerged over the last decade. During its infancy, interest in high-level synthesis was restricted to the academic, research community. Today, with the release of several commercial high-level synthesis tools, the technology has proven to be robust enough for industry use.

In this chapter, we will quickly tour the design flow for digital systems – specifically, application specific integrated circuits (ASICs) – and be introduced to what a high-level synthesis system can do for us.

1.1 Designing ASICs

An ASIC is an integrated circuit (IC) built to execute a specific application, as opposed to a microprocessor which is an IC built to execute many applications.

Most ASIC design methodologies follow the steps shown in Figure 1.1. The task that the ASIC is expected to perform is described as a functional specification. The functional specification must be implemented in the ASIC.

The very first design step is to design an algorithm that performs this function. Once the algorithm has been determined, the architecture for the ASIC is designed. The architecture specifies how high-level components – such as ALUs, multipliers, registers, and memories – work together to execute the algorithm. The algorithm specifies the behavior and the architecture executes the behavior. After the architecture has been designed, it is converted into gate-level circuits. These are then translated into transistor-level circuits which are finally placed, routed and laid out to form the specification for fabrication masks for a silicon die. The mask specification is then sent to a foundry where the masks are created and the ASIC is fabricated.

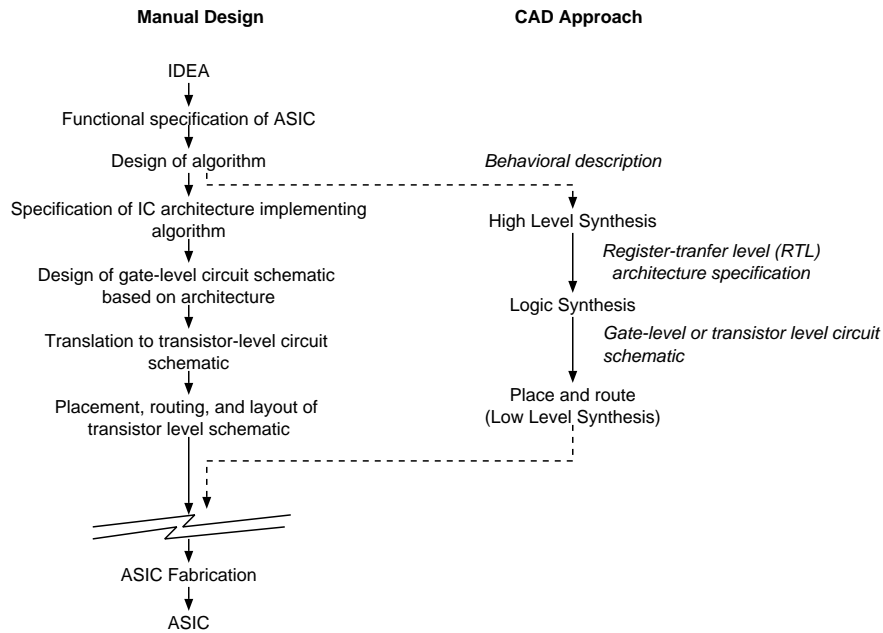


Figure 1.1: ASIC design flow

As fabrication technology advances, increasingly complex designs can be mapped to ASICs. Today, designs with transistor counts in the millions are commonplace. As this complexity grows, managing ASIC designs manually becomes untenable. More and more of the design process is being entrusted to computer-aided design (CAD) tools. Figure 1.1 shows the various CAD stages in an ASIC design flow. The stages are matched in the figure with the manual design tasks that they automate.

Over the years, the tasks of ASIC design that have been automated have moved to higher levels of design. Initially, computers were used by ASIC designers to aid in manually laying out transistor-level circuits. Later, they were used to perform a limited amount of placement and routing. As designs grew and as computers became more powerful, computers began to be used to synthesize logic for architectures. The level of specification to CAD tools moved up from schematic capture to higher levels of abstraction like the register-transfer level (RTL). Today, computers are being used to synthesize the architectures themselves and specifications have reached the level of behavior. The automated translation of a behavior to an architecture is called high-level synthesis. Other names for this step are behavioral synthesis or architectural synthesis.

High-level synthesis is just starting to be adopted by ASIC designers in industry. Behavioral Compiler from Synopsys Inc., was introduced in 1994 and the first commercial ASICs designed by Behavioral Compiler appeared in late 1996.

1.2 A high-level synthesis tutorial

High-level synthesis (HLS) is a very complicated optimization process that solves several design problems. In this section, we introduce each of the main design problems that a high-level synthesis system has to solve. In the chapters to follow, we will look at each of these problems in more detail and study techniques for automatically solving them.

1.2.1 What an HLS system does

An HLS system accepts as input a *behavioral description* of an ASIC. This description is typically the algorithm that the ASIC is to implement and is usually written in a hardware description language (HDL) such as VHDL or Verilog. Figure 1.2 shows a small VHDL description that represents a 2nd-order infinite impulse (IIR) filter, a common design from digital signal processing.

Throughout this book, we use VHDL as the input language for describing the behavior of a design. Many different levels of specification can be described in VHDL [1]. The identifying characteristics of a behavioral description are that the number and kind of datapath components are not specified, and neither is the cycle-by-cycle execution of operations by the ASIC. The behavioral description is meant to capture the algorithm alone. For example, in our description of an IIR2 filter, there are five separate multiply operations. These are the operations that must be performed; these statements do not imply that there are five multipliers in the actual datapath. Depending on design constraints and data dependencies the synthesized design may contain as many as five multipliers – or as few as one. Similarly, the entire loop of the filter is described as happening in a single clock cycle. The final synthesized design will contain several clock cycles. The exact number is decided by the HLS system. The designer does not specify what happens in each cycle, but rather just specifies the algorithm. Behavioral VHDL is described in more detail in Section 3.2.

An HLS system synthesizes and outputs a complete architecture for the input behavioral description. Figure 1.3 shows an example of what an architecture synthesized by an HLS system for our IIR2 example would look like. Note that this particular architecture contains two multipliers to implement the operations in Figure 1.2. It consists of a controller and datapath. The architecture is usually output as a *register-transfer level (RTL) specification* described in an HDL.

The controller is usually a finite state machine. The FSM controls the cycle-by-cycle operation of the datapath and ensures it executes the proper behavior, namely the behavior specified in the input behavioral description. There may be several combinations of datapath and cycle-by-cycle operation that executes the input behavioral description. The goal of an HLS system is to find the best one given the timing and hardware constraints imposed by the ASIC designer.

The datapath consists of an execution unit of data processing elements (or functional units) like adders and multipliers; a storage unit of storage elements like register files and RAMs; and a data transfer subsystem consisting of buses,

```

ENTITY iir2 IS
  PORT (
    x : IN  INTEGER
    y : OUT INTEGER
    clk : IN  BIT
    reset : IN BIT
  );

  ARCHITECTURE behav OF iir2 IS
  BEGIN
    main : PROCESS
      VARIABLE s1, s2 : INTEGER;
    BEGIN
      reset_loop : LOOP
        s1 := 0;
        s2 := 0;
        WAIT UNTIL clk='1' AND clk'EVENT;
        IF reset='1' THEN EXIT reset_loop; END IF;
      main_loop : LOOP
        s0 := x - 2.0*s1 - 0.5*s2;
        y := 1.5*s0 + 0.72*s1 + 0.25*s2;
        s2 := s1;
        s1 := s0;
        WAIT UNTIL clk='1' AND clk'EVENT;
        IF reset='1' THEN EXIT reset_loop; END IF;
      END LOOP main_loop;
    END LOOP reset_loop;
  END PROCESS main;
END behav;

```

Figure 1.2: A behavioral description of an IIR filter in VHDL

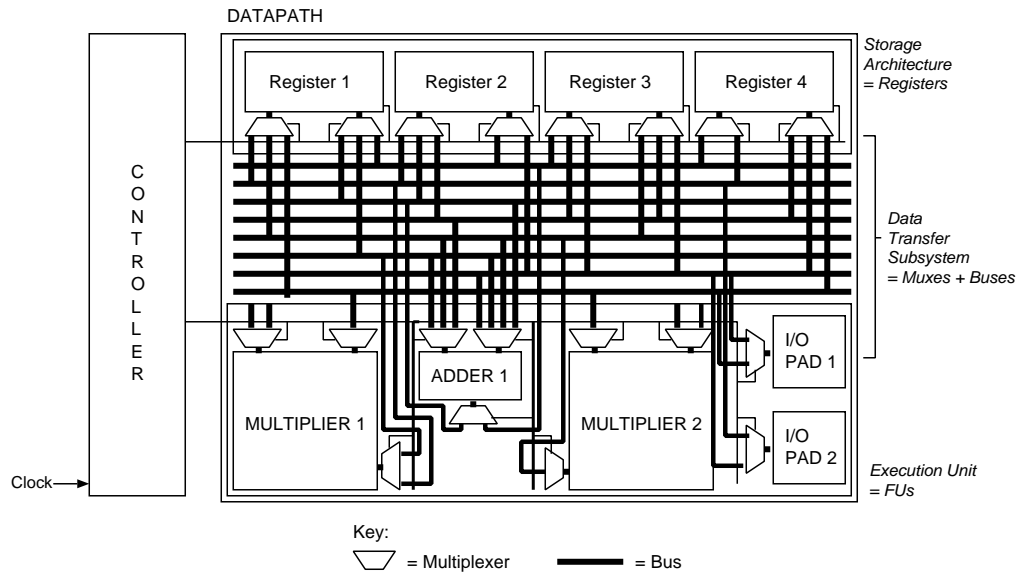


Figure 1.3: A synthesized architecture for the IIR2 filter

interconnect, and multiplexers. The execution unit processes data, the storage unit stores it as long as is needed, and the data transfer subsystem makes sure it gets to where it is going.

The designer typically guides the HLS system by providing it with design constraints. There are two kinds of constraints: timing constraints and resource constraints.

Timing constraints allow the designer to specify timing relationships between operations in the behavior that must be obeyed in the synthesized architecture. For example, a timing constraint could say two operations must happen 5 clock cycles apart or that a loop must execute in 10 clock cycles.

Resource constraints tell the HLS system about restrictions on the resources in the synthesized architecture. For example, constraints might specify that an operation has to be executed on a certain functional unit or that the architecture has only 3 adders and 2 multipliers available.

The HLS system must also be told what the target technology for the ASIC being design is – for example, we might want our ASIC fabricated in LSI 10K technology or maybe on an Altera FPGA. Communicating the target technology to an HLS system is usually done by supplying it with a technology library.

A *technology library* contains information about basic design parts which the HLS system can use during synthesis. The library contains area and timing information for these parts. This information is critical to an HLS system since the system needs to ensure that the architecture it synthesizes meets both timing and area constraints specified for the ASIC.

1.2.2 The Optimization Goals of an HLS system

An HLS system solves an intricate optimization problem. The two basic properties that an HLS system will try to optimize are execution time of the architecture and the total area of the architecture.

The execution time of the architecture is the total time it takes the architecture to execute the behavior. This is the number of clock cycles required multiplied by the period of each clock cycle. In some cases, the clock cycle period is supplied as a timing constraint by the user. In others, the clock period is part of the HLS optimization problem.

The area of the architecture is the silicon area that will be required by the ASIC that implements the architecture. ASIC area depends on design steps that follow high-level synthesis, namely logic synthesis, placement and routing. HLS systems typically estimate ASIC area as a function of the various resources used in the architecture.

There are two classic formulations of the optimization problem based on which property of the ASIC is being optimized:

- In *resource-constrained HLS*, the HLS system minimizes the execution time of the behavior given an upper bound on the ASIC area available to the architecture.

- In *time-constrained HLS*, the HLS system minimizes the ASIC area needed by the behavior given an upper bound on the execution time for the behavior.

In a real design environment, the HLS optimization problem is usually a mix of constraints and optimization goals. Users might specify both resource and timing constraints and require that both execution time and area be minimized, with one taking precedence over the other.

Sometimes other properties need to be optimized as well. Recently power has become important as an optimization criterion, adding another axis to the HLS optimization problem.

1.2.3 The challenges of HLS

The optimization problems mentioned above are *NP-hard*. This means that there is no known algorithm to solve these problems in *polynomial* time. The time taken to solve the problem grows exponentially as the size of the input behavior grows. For example, if it takes 1 second to perform HLS on an input behavior with 10 operations, it might take 17 minutes to perform HLS on an input behavior with 20 operations, i.e. $\frac{2^{20}}{2^{10}}$ seconds. Extending the example, it might take 291 *hours* to perform HLS on an input behavior with 30 operations, i.e. $\frac{2^{30}}{2^{10}}$ seconds. NP-hard problems are discussed in more detail in Section 4.3.

The nature of NP-hard problems dictates that any algorithm that would always produce an optimal solution for HLS would *have to* run in exponential time in the worst-case. As a result, HLS systems typically produce approximate solutions for the HLS optimization problems. These approximations are sub-optimal but, more importantly, they can be generated in a reasonably short time. One of the main challenge in designing an HLS system is to find algorithms that generate consistently good solutions and still run in a reasonable amount of time.

1.2.4 Breaking HLS into subproblems

HLS can be conceptually broken down into a series of input processing steps and optimization subproblems. The solutions to each are combined to form the solution to the HLS problem.

Figure 1.4 shows the design flow through a typical HLS system. The HLS subproblems shown in the figure interact. Different HLS system choose to resolve the interactions in different ways. Some HLS systems choose to solve the subproblems one after the other with the solutions to one being used by the solvers for the next. Others choose to solve them iteratively – sequentially solving the subproblems and then repeating the processing with the solutions of the last iteration. Some HLS solve them all simultaneously. The challenge once again is to generate a good architecture in a reasonable amount of time.

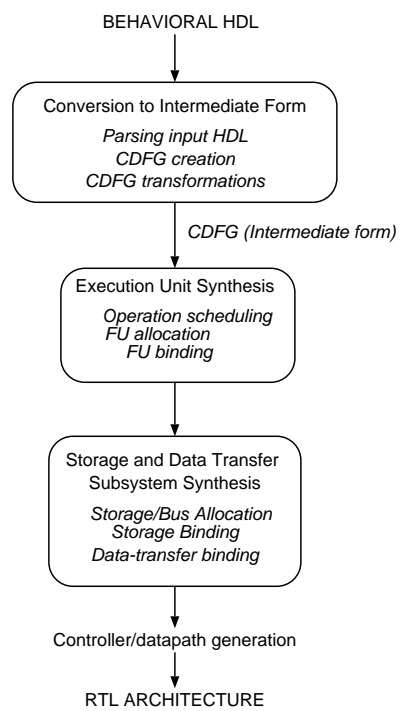


Figure 1.4: A typical HLS system

Conversion to intermediate form The hardware description language (HDL) behavioral description is first translated into an intermediate form. This is usually a directed graph composed of nodes and edges connecting pairs of nodes. Nodes in the graph represent operations in the behavior and directed edges between nodes represent data that is the output of one operation being input to another. This type of edge specifies a precedence timing constraint between the two operations. The first must happen before the second when the synthesized architecture is executing. The graph is called a data flowgraph (DFG) since it specifies the flow of data in the behavior. A second type of edge, called a control edge, may be added to the graph if the behavior has control constructs like `if...then...else` structures, loops or hierarchy. In that case, it is called a control/data flowgraph (CDFG). Figure 1.5 illustrates how a DSP representation for the kernel of the 2nd-order IIR filter (IIR2) (in Figure 1.2) is converted to a VHDL representation which is then parsed into a DFG representation. We discuss generating the CDFG in Chapter 5.

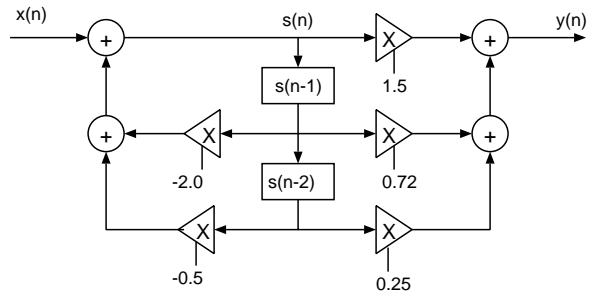
Operation scheduling The execution time of the architecture is divided into control steps (csteps). Csteps may be clock cycles or, more generally, states in the execution of the controller (FSM) of the architecture. Operation scheduling assigns each operation in the behavior to a cstep or csteps in which it will be executed by the architecture. This defines the *schedule* of the architecture. Operation scheduling must satisfy all timing and resource constraints. For example, if *op1* produces data that *op2* uses, *op1* must not be scheduled after *op 2*. Otherwise a precedence timing constraint would be violated. Another example arises when there is a resource constraint that limits the number of adders in the architecture to two. In this case the scheduler should not schedule more than two add operations at the same cstep. Doing so would violate the resource constraint since all operations scheduled at the same cstep must be executed on different functional units. Figure 1.6 illustrates a possible schedule for our IIR2 example. Algorithms for scheduling are discussed in Chapter 6.

Functional unit allocation In FU allocation, the maximum number of each type of functional unit available in the architecture is specified. There must be at least one of each type of functional unit required by the operations in the behavior.

Functional unit binding Functional unit binding assigns a functional unit to every operation. Figure 1.7 shows a table that contains a list of allocated functional units and the operations that are bound to each functional unit at csteps in the schedule. This table is called a *resource reservation table*. Allocation and binding are discussed in Chapter 7.

Storage/bus allocation The solution to this subproblem specifies the maximum numbers of various types of storage elements, like RAMs, ROMs, and register files, and the number of buses available in the architecture.

2nd-order digital IIR filter (Digital signal processing representation)



Behavioral HDL

```

s0 := x - 2.0*s1 - 0.5*s2;      -- s0 = s(n)
y  := 1.5*s0 + 0.72*s1 + 0.25*s2; -- s2 = s(n-2)
s2 := s1;                      -- s2 = s(n-2)
s1 := s0;                      -- s1 = s(n-1)

```

Dataflow graph (DFG)

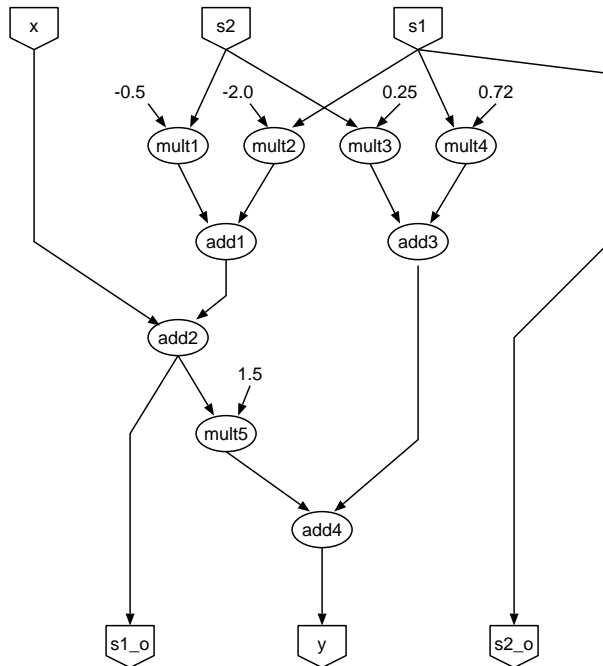


Figure 1.5: Converting HDL for IIR2 to a DFG

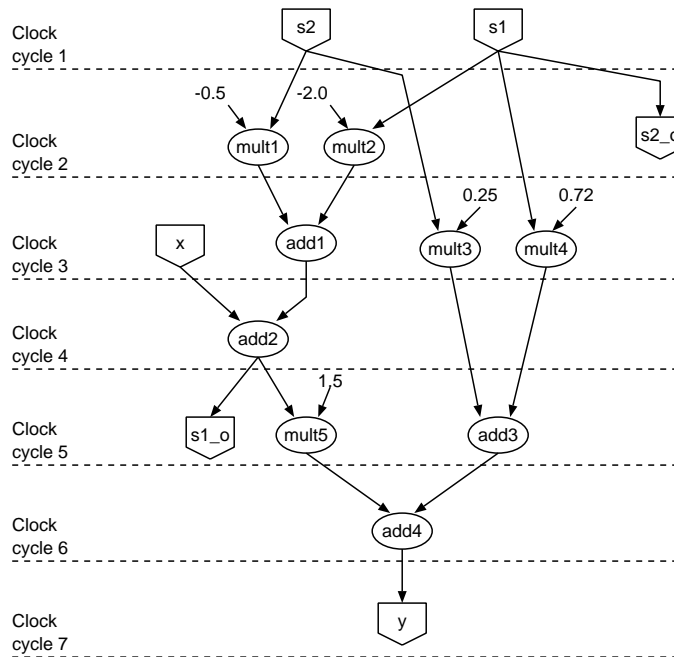


Figure 1.6: Scheduling the IIR2

CSteps	I/O pad 1	I/O pad 2	Multiplier 1	Multiplier 2	Adder 1
1	s2	s1			
2		s2_o	mult1	mult2	
3	x		mult3	mult4	add1
4					add2
5	s1_o			mult5	add3
6					add4
7		y			

Functional unit allocation:
 2 I/O pads
 2 multipliers
 1 adder

Figure 1.7: Functional unit binding of the IIR2

Storage binding Data that is used in a cstep other than the one in which it is produced must be stored in a storage element: It must be assigned to individual registers or locations in storage elements like register files, RAMs and ROMs. In addition, the data must be held in storage from the time it is produced to the time it is needed last. Figure 1.8 shows a reservation table expressing storage allocation and binding. In our example, all of the storage elements are simple, edge-triggered registers. The data generated by an operation is assigned to a register and may reside in the register for multiple csteps.

CSteps	Register 1	Register 2	Register 3	Register 4
1				
2	<s2>	<s1>		
3	<s2>	<s1>	<mult1>	<mult2>
4	<x>	<add1>	<mult3>	<mult4>
5	<a2>		<mult3>	<mult4>
6		<mult5>	<add3>	
7		<add4>		

Key: <i> represents the data produced by operation i

Storage allocation:
4 registers

Figure 1.8: Storage binding of the IIR2

Data-transfer binding In addition to being produced and stored, data needs to be moved to where it is needed. Data must be bound to the input and output terminals of functional units that use or produce it, and it must be bound to read or write ports of storage elements from which it is read or into which it is written. Data transfers from a source to destinations must be assigned to buses in the architecture. The combination is data-transfer binding. Figure 1.9 shows a sample bus binding for the IIR2 example above. Notice how the data produced by an operation may need to be transferred multiple times depending on when it is produced and used. Once data-transfers are bound, multiplexers can be assigned to the inputs and outputs of functional units and storage elements to ensure the terminals and ports are connected to the correct buses at the various csteps in the schedule. Issues in storage and interconnect allocation are discussed in Chapters 7 and 8.

Controller/datapath generation Once the solutions to all the above sub-problems are available, the HLS system has enough information to specify the architecture as a controller and datapath. The architecture is usually expressed in a register-transfer level HDL description. This is the input to a logic synthesis CAD tool in the next step in the ASIC design flow. The datapath synthesized for our example is the same as the one we showed at the beginning of this tu-

CSteps	Bus1	Bus2	Bus3	Bus4	Bus5	Bus6	Bus7	Bus8
1	<s2>	<s1>						
2	<s2>	<s1>	<mult1>	<mult2>				
3	<s2>	<s1>	<mult1>	<mult2>	<x>	<add1>	<mult3>	<mult4>
4		<x>	<add1>	<add2>				
5		<add2>	<mult3>	<mult4>	<mult5>	<add3>		
6					<mult5>	<add3>	<add4>	
7	<add4>							

Key: <i> represents the data produced by operation i

Interconnect allocation:
8 buses

Figure 1.9: Bus binding of the IIR2

torial. Figure 1.3 illustrates the architecture synthesized in this tutorial for the IIR2 example.

1.3 Advantages and Disadvantages of High Level Synthesis

We have seen how HLS fits in an ASIC design flow and have illustrated the basic steps in HLS by stepping a small example through the HLS flow and synthesizing an architecture for it. Our treatment so far has been very elementary and has used many simplifying assumptions. Real, industry-strength HLS systems are much more complex. For example, they must deal with control flow, I/O timing, and a host of other issues. We will discuss some of these issues in more detail throughout this book.

As with any other approach, HLS has its advantages and disadvantages. The main advantages of HLS result from being able to describe designs at a higher level of abstraction. Specifications are short and can be simulated quickly. In addition, computers generally do a more thorough and rapid job of searching the design space than a human designer. The disadvantages are also a result of the higher level of abstraction of the input description. The designer can lose control of the synthesized architecture and may not get the performance expected or required from the resulting design.

1.3.1 Advantages of HLS

The trend in design automation has been to move to higher and higher levels of the ASIC design flow in Figure 1.1. As more of the design is relegated to the computer, the designer is freed to concentrate more on the specification of the design, and less on the implementation details. This has several advantages in HLS.

As a designer using a HLS tool, you don't have to specify architectural details. The synthesis tool does it for you. You can concentrate on getting the behavior to be what you want. The tool will automatically generate an architecture, and make decisions such as how many of each kind of functional unit are required. If you move to a new technology that gives you more silicon area, you can use the same specification, and simply by changing the constraints generate a new design that targets the new technology and makes use of the newly available resources. Similarly, a high level synthesis system generates the schedule of operations for you. The system, not the designer, determines which operations occur at each cycle during execution. This frees you to concentrate on what you are designing, and also allows new and different designs to be easily generated from the same behavioral specification.

Since the tool is automatically generating a detailed architecture more rapidly than a human designer, the designer can use the saved time to perform a significant amount of design space exploration. For example, the filter in section 1.2 uses two multipliers. A better design might use one multiplier or even three or four. Using HLS, a designer can generate all these architectures and determine which is best for that particular design. A designer working at the RTL level tends to choose an architecture and stick with it due to the difficulty of changing the details to generate a new architecture. Exploring different architectures is too time consuming at the RTL level to be practical, but is made possible by moving to a behavioral level of design.

In behavioral design, since the designer is specifying fewer details, the specifications tend to be significantly shorter. As a result, there are fewer chances for a designer to introduce errors. In contrast, at the RTL level the designer must specify all the details of cycle-by-cycle execution of the architecture and sometimes even the connectivity of components. It becomes easy to introduce errors at this level, especially for larger designs, since there are a multitude of details to track. By working at the behavioral level the designer shifts the responsibility for this to the computer, which is well suited to accurately managing large amounts of tedious detail. The result is a less error prone design flow.

Behavioral specification can be simulated more rapidly than a full RTL design. This allows you to find errors quicker and earlier in the design cycle. However, since less detail is being simulated, behavioral simulation does not preclude testing at the RTL level. For example, behaviors are usually specified as if everything happens at one clock cycle, so simulation at the behavioral level will not provide any timing information or the cycle-by-cycle execution of an ASIC. But testing early in the design cycle does allow you to check your behavior and find errors early. This is useful for checking your algorithm and for integrating your design into a larger system.

1.3.2 Disadvantages of HLS

The main advantage of HLS, the higher level of design abstraction, is also its main disadvantage. Designers can lose control over the architecture of their design. If the synthesized architecture meets the requirements, then a designer

is usually happy to use the HLS tool. Problems arise when the design tool generates designs that don't meet the constraints imposed by the designer, or if the tool generates an implementation that is clearly worse than the designer would have developed manually.

There are certain circumstances when a designer requires tighter control than that available from HLS systems. For example, designs with stringent timing requirements or very high speed requirements may not be good candidates for HLS systems. For some applications, different architectures have been studied widely and advantages and disadvantages of particular architectural implementations are well known. For example, in many high throughput DSP applications, a designer knows which is the best architecture for their application. In such cases, HLS may not be the best approach for implementing a design, since it may result in a sub-optimal architecture. Some HLS systems provide for user control of synthesis and have features for incremental synthesis. Both these features may help designers guide the HLS system to efficient architectures for these special applications.

While a designer has tighter control of the architecture by designing at the RTL level, this control comes at a cost. The specification required is longer, and the designer has to attend to many more details than at the behavioral level. This lengthens the design cycle, as well as giving the designer more opportunity to make errors in the specification. Longer specifications also require longer simulation times. In addition, changes are difficult to make. If, in the future, you wish to change the architecture or migrate to a new technology, a great deal of tedious book-keeping may have to be dealt with by the designer.

The complete design space for a specification is huge. Most HLS systems do not attempt to explore the entire design space, but rather assume a specific architectural model. A model with a datapath and a state machine controller is common. The datapath and the controller are each constructed according to certain assumptions. For example, a specific system may assume the storage and data transfer architectures consist of individual registers, muxes and wires. Such a system would not generate a very good design for a microprocessor whose best architecture consists of register files and shared buses. By choosing an HLS system, the designer is choosing the architectural model of his or her design. It is important to understand the underlying model, and use the HLS system for applications that are appropriate.

We refer to the behavioral description as a *specification* of the design. However, this specification is not entirely independent of the resulting architecture. Changing the specification slightly can change the synthesized architecture, even if the new specification expresses the same behavior as the original one. For example, there are many equivalent ways to express a specific behavior in VHDL. Each can produce a different result from HLS. The sensitivity of the results of HLS to changes in the input specification is often exploited by experienced users as a feature; they guide the HLS system by modifying specifications to influence the synthesized architecture directly. This is a way to control the output of the HLS tool. However, this sensitivity can also be frustrating to a first-time user of HLS since seemingly minor changes in the specification can cause large

changes to the implementation. The independence between input specification and synthesized architecture is lost.

In our experience, for a large class of applications, the advantages of HLS outweigh the disadvantages. In addition, as design tools evolve, the quality of the results tend to improve, the sensitivity of the tools to the input reduces, and controllability increases. This has proven to be the case in mature logic synthesis tools and is a trend appearing in modern HLS tools as well.

Most designers use the best tools available that give them the performance they require. High-level synthesis tools are the best solution for many applications, provided they are used appropriately. In this book, we will provide you with an appreciation for the internals of HLS systems. This will help you use HLS tools with greater insight and understanding.

1.4 Construction of a toy HLS system

Throughout this book, we rely on programming exercises to emphasize the points made in the chapters. The programming exercises allow you to quickly implement algorithms described in the book and test them on inputs of your own creation. This will help you internalize the concepts discussed and also gain an appreciation for the strengths and limitations of the techniques discussed.

Only a rudimentary knowledge of C is essential. We provide you with a library of C data structures and functions with which to manipulate them. This enables you to concentrate on the algorithms you are developing and not the details of the underlying data structures.

Figure 1.10 is a program which implements an algorithm called depth-first search on a data structure called a graph. The data structures `Graph`, `Node`, `Edge`, and `List` are provided in the library which is called `libHLS`. In the program, they are manipulated by functions like `graph_nodes()`, `node_setattr()`, `edge_dst()` and `l_append()`.

Many more functions are provided in `libHLS` which allow these and other data structures to be manipulated and interact with one another in a controlled, consistent manner. These allow you to construct both generic algorithms (like the depth-first search) as well as HLS-specific algorithms (like schedulers and allocators). We present `libHLS` and some basics of data structures used in HLS in Chapter 4.

As you read this book, you will encounter a sequence of programming assignments. You will start with simple exercises in which you construct some generic algorithms for graph and list data structures. These will introduce you to `libHLS` and will also be used as functions in later exercises. The assignments help you construct each piece of an HLS system. Finally, you will combine all the assignments to build up a toy HLS system. The HLS system will accept as input a CDFG, schedule and allocate the datapath, and output a datapath and controller specification. Although the system will not have anywhere near the power of an industry-strength HLS system, constructing it will help you understand the principles of HLS more clearly.

```

List *graph_dfs(Graph *g)
{
    List *dfs_nl; /* DFS nodelist */
    Node *n;      /* Node of graph */

    /* Create the DFS nodelist */
    dfs_nl = l_new();

    /* Ensure the "visited?" attribute is un-set on all nodes */
    l_foreach(graph_nodes(g), n) {
        node_delattr(n, "visited?");
    } l_endfor;

    /* For every node, do */
    l_foreach(graph_nodes(g), n) {
        visit_node(n, dfs_nl); /* Start the DFS here */
    } l_endfor;

    return dfs_nl;
}

void visit_node(Node *n, List *dfs_nl)
{
    char *b; /* Value of "visited" attribute */
    Edge *e; /* Outedge of node n */
    Node *c_n; /* Child node of node n */

    /* Ignore the node if it has been or is being visited */
    if (node_getattr(n, "visited?")) return;

    /* We need to visit this node, so mark it as being visited */
    node_setattr(n, "visited?", "is being visited");

    /* Otherwise, visit all the child nodes first */
    l_foreach(node_outedges(n), e) {
        c_n = edge_dst(e); /* Get the child node */
        visit_node(c_n, dfs_nl); /* Visit the node -- RECURSION!!!! */
    } l_endfor;

    /* All children have been visited. Mark this node as visited */
    node_setattr(n, "visited?", "yes");

    /* Add it to the end of the DFS nodelist */
    l_append(dfs_nl, n);
}

```

Figure 1.10: libHLS code for depth-first search

We believe that learning by doing will help internalize the concepts of HLS. Also, implementing and experimenting with a toy HLS system will make you a more educated, and hopefully more effective, user of industry-strength HLS tools.

1.5 Conclusions

We have presented an introduction to High Level Synthesis and a brief tutorial. We also recommend an excellent early tutorial on the subject [2]. The remainder of the book goes into the material touched on in this introduction in more detail. The next section of the book contains three chapters on the essentials of HLS. These include hardware models used by different systems, behavioral descriptions and internal formats, and the basics of data structures and algorithms used for HLS. The next section goes into some of the core algorithms used in HLS for scheduling, allocation, binding, and storage synthesis. In the last section we look at several case studies including a toy synthesis system built out of the programming assignments in this book, an academic system called HYPER, and a commercially available HLS system from Synopsys, Behavioral Compiler.

Bibliography

- [1] IEEE, *VHDL Language Reference Manual*, 1993.
- [2] M. C. McFarland, A. C. Parker, and R. Camposano, "The High-Level Synthesis of Digital Systems," *Proceedings of the IEEE*, vol. 78, pp. 301–318, February 1990.