Implementation and Evaluation of a Power Prediction Model for a Field Programmable Gate Array

A Master's Thesis

Timothy A. Osmulski

Department of Computer Science

Texas Tech University

John K. Antonio (Chairperson)

William M. Marcy

Noe Lopez-Benitez

ACKNOWLEDGEMENTS

I wish to express my sincere gratitude to Dr. John Antonio, my committee chairperson. He has been of invaluable assistance, and I doubt this research would have been so successful without the benefit of his knowledge and experience. I also wish to thank the other members of my thesis committee, Dr. William Marcy and Dr. Noe Lopez-Benitez, for their suggestions and advice.

The success of this research depends largely on the efforts of the other members of my research lab, including Nikhil Gupta, Jack West, Jeff Muehring, and Brian Veale. Nik created several of the FPGA configurations used to test the prototype simulator, and a large portion of Appendix A was taken directly from his work. I also wish to thank Jason Bivens, who created the data acquisition system that allowed us to perform our temperature measurements.

I would like to thank my parents, Ted and Nancy, and my sister Amy, for a good 18 years of growth and learning. Achieving my educational goals would have been impossible without the support of my family.

Finally, I want to express my appreciation to the Computer Science Department at Texas Tech University, including all the professors whose classes I had the pleasure to take. Teachers, after all, often deserve more thanks than they get.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS ii
ABSTRACT iv
LIST OF FIGURES iv
I INTRODUCTION 1
II CONCEPTS OF POWER MODELING 3
III TRADEOFFS AMONG DSPS, ASICS, AND FPGAS 6
3.1 Overview 6
3.2 Capacitance Charging Power 8
3.3 Summary 9
IV PREVIOUS RESEARCH IN POWER PREDICTION 11
4.1 Time-Domain Techniques 11
4.2 Probabilistic Techniques 13
4.3 Previous Application of Probabilistic Power Prediction 17
V RESEARCH PERFORMED 19
5.1 Model Design 19
5.1.1 Design of Internal Structures 19
5.1.2 Design of Signal Classes 21
5.1.3 Feedback Resolution 28
5.2 Implementation and Results 29
5.2.1 Implementation Language 29
5.2.2 Convergence of Signal Stabilities 30
5.2.3 Infeasibility of a Large Number of Inputs 31
5.2.4 Proof of Concept: Measuring Actual Power Consumption 32
VI CONCLUSIONS AND FURTHER RESEARCH 34
REFERENCES 36
APPENDICES 37
A THE XILINX 4000 SERIES FPGAS 37
B SELECTED PORTIONS OF THE SOURCE CODE 44

ABSTRACT

As configurable computing devices like FPGAs gain popularity as application-specific computing solutions, the special problems presented by these devices must be considered. For example, the power consumed by the FPGA can be heavily dependent upon the dynamic nature of the input data. In addition, the routing resources that allow FPGAs to be reconfigured can be a significant source of power consumption. Because changes in the device's configuration can affect both of these issues, the opportunity exists to optimize the utilization of the FPGA's resources before the device is actually configured. A fast method of modeling an FPGA's power consumption would be an important tool to achieve this end.

This thesis explores the problem of modeling power within a reconfigurable device. A probabilistic method for modeling power consumption within an FPGA is presented. The implementation of this method is the basis of the developed power prediction simulator.

LIST OF FIGURES

3.1 Leakage current 6
3.2 Switching transient current 7
3.3 Capacitance charging current 7
3.4 Equivalent model for a transistor gate 8
4.1 CMOS implementation of the Boolean function $y = x_1 x_2 \overline{x_3}$ 11
4.2 Time domain modeling 12
4.3 Obtaining probability measures from time-domain signals 14
4.4 Obtaining activity measures from time-domain signals 14
4.5 Using activity measures to estimate consumed power 16
4.6 Theoretical row-based FPGA 17
5.1 Simulator inheritance diagram 20
5.2 Simulator ownership diagram 20
5.3 Signal class inheritance diagram 21
5.4 Signal class ownership diagram 22
5.5 Local/Remote signal relation 22
5.6 Example physical Logic Block configuration 23
5.7 Modeling a physical configuration with Local/Remote signals 23
5.8 Reconvergent fan-out 25
5.9 Illustration of signal feedback and convergence of signals 28
5.10 Convergence times for symbolic probabilities 30
5.11 Chip temperature over a given frequency range 32
A.1 Diagram of a Xilinx 4000 series FPGA 37
A.2 A Xilinx 4000 series CLB 38
A.3 Carry logic of a Xilinx 4000 series FPGA 39
A.4 A Xilinx 4000 series IOB 40
A.5 Diagram of a Programmable Switch Matrix 41
A.6 Illustration of a CLB with routing channels and switching matrices 42
A.7 Illustration of double-length routing segments 42

CHAPTER I

INTRODUCTION

The use of configurable devices such as programmable logic devices (PLDs) and field programmable gate arrays (FPGAs) is, in some instances, becoming a more popular approach to implementing application-specific computing systems than designing and manufacturing application-specific integrated circuits (ASICs). For special-purpose computing, configurable devices may offer a cost-effective alternative to the use of ASICs. This is especially true for special-purpose applications in which relatively few copies of the chip(s) need to be produced. In addition to configurable devices, dynamically *reconfigurable* computing devices such as SRAM (static random-access memory) based FPGAs are beginning to challenge digital signal processing (DSP) chips in the areas of high-bandwidth, real-time, and embedded signal processing applications [4, 5].

In many cases, a device's performance (i.e., speed) is the principal design consideration, but power consumption is of growing concern as the logic density and clock rate of ICs increase. This is notably true for battery-operated equipment such as cellular telephones and GPS receivers, and for remote devices housed in satellites and aircraft, where power is a premium. The measurement and prediction of an IC's power consumption therefore can be an important issue, especially when designing a system using configurable devices. In particular, the characteristics that make these devices configurable also introduce timing and power considerations not present with the use of traditional IC design and implementation techniques.

This thesis addresses the problem of power prediction for configurable devices, with emphasis on predicting power for a class of FPGAs. To set the context of the research, an overview of tradeoffs in the use of DSPs, ASICs, and FPGAs is first provided. This overview is followed by a brief survey of relevant research on power consumption in CMOS (complementary metal-oxide semiconductor) ICs. A method for predicting the power consumption of a specific class of FPGAs is then outlined. The method is implemented, in Java, for a class of FPGAs (the Xilinx 4000 series). Preliminary evaluation of the method is performed by comparison of predicted power and heat measurements taken from the surface of an FPGA.

CHAPTER II

TRADEOFFS AMONG DSPS, ASICS, AND FPGAS

A decade ago, when an embedded system design required special-purpose computing hardware, the hardware was either built using small-scale integrated circuits or designed into a special chip (i.e., an ASIC). Use of an ASIC typically results in a device that is smaller in size and higher in reliability than a board of many smaller (i.e., less dense) ICs. However, the manufacturing of an ASIC can be prohibitively expensive; if only a few hundred ASICs are manufactured, as is often the case for special-purpose applications, then each chip may cost thousands of dollars. In contrast, if several million chips are to be made, which is typically the case for general-purpose devices, the high initial manufacturing cost can be offset by the vast quantity of chips sold [9].

One way to avoid the high cost of ASIC manufacturing is to use relatively general-purpose chips, e.g., DSPs, along with application-specific software to control these chips. Because thousands of copies of the general-purpose chips have likely been produced, they are relatively inexpensive. This approach shifts costs of the application-specific computing platform from hardware design and manufacturing to software development. It also provides the benefit of being able to correct some initial design errors, or even to adapt the system for future applications.

DSP chips offer relatively high performance for certain signal processing functions such as Fourier transforms and convolutions. Because they are highly optimized to perform various signal-processing functions, they can be very useful in application areas such as image and radar processing [5]. Although devices like DSPs

are often well tuned for embedded applications, these devices are still relatively general in the sense that they are not designed specifically for any one application. Consequently, their use may carry with it a penalty of having "excess silicon complexity" that is not explicitly required for any given application in which they are used. Thus, performance may not be as high as it could be if ASICs were used in the implementation. In addition, in some instances this excess complexity can result in implementations in which the power consumption of the platform is too high (i.e., does not meet system requirements).

The use of configurable devices is a relatively new alternative to both DSP- and ASIC-based designs. PLDs and FPGAs have traditionally been used as "glue-logic" in complex systems, alleviating the need for large quantities of small-scale ICs. In recent years, however, manufacturers of configurable devices have responded to the market demand of using configurable hardware for some core components of high-performance computing systems. This has led to improvements in configurable hardware design to the point that FPGAs are now a viable implementation alternative.

Instead of application software controlling the processor(s), as is the case in DSP-based designs, "software" is actually used to configure the FPGA chip, thereby defining its functionality. The software used in this process is typically an encoding expressed using a hardware description language such as VHDL (*V*ery High-Speed Integrated Circuit *H*ardware *D*escription *L*anguage) [8]. The VHDL "program" is first passed through a synthesis process in which the encoded design is translated into an implementation suitable for the FPGA being targeted. Next, the output from the synthesis process, which is called a netlist, serves as input to a process called place and

route, in which functional blocks associated with the netlist are assigned to physical logic blocks on the FPGA. The routings (i.e., connections) among the logic blocks are also determined during this phase.

The use of FPGAs offers the ability to specifically match a hardware design with the processing needs of the application, with the design directly implemented in hardware. As with traditional software-controlled computing engines (e.g., DSPs), the hardware cost of FPGA-based implementations is low relative to ASIC-based systems. By coupling the advantages of a "soft" design methodology with the flexibility of full implementation in hardware, configurable computing devices offer an attractive alternative to the implementation of special-purpose computing platforms using ASICs.

Although FPGAs can provide performance at or near ASIC levels, the flexibility of an FPGA is obtained by the use of generic and programmable logic blocks and routing resources. The generality of these internal structures can translate into implementation inefficiencies, which potentially can have a net effect of increasing power consumption (as compared with ASIC implementations). A model capable of predicting power consumption of the device would therefore be instrumental in optimizing utilization of its components (i.e., logic blocks and routing resources) so as to minimize power consumption for a given design.

CHAPTER III

CONCEPTS OF POWER MODELING

3.1. Overview

Modeling power in an electric device is fundamentally a matter of predicting current flows and voltage differences throughout the device. In CMOS devices, power consumption is due to the following three types of current flow [9]:

- *Leakage Current*. The field-effect transistors (FETs) used in CMOS devices rely on a voltage difference to produce an electric field. This field is responsible for controlling the transistor's state. The FETs are imperfect, however, and some current leaks through to other parts of the transistor, resulting in power loss (see Fig. 3.1).

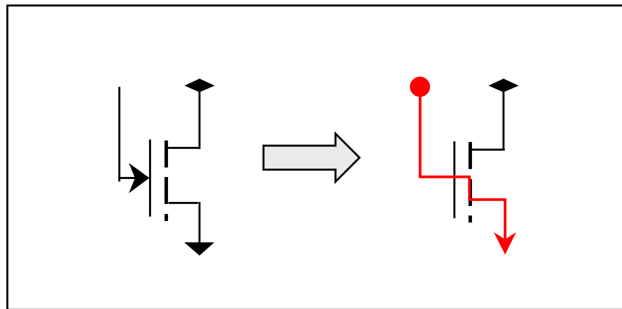

Fig. 3.1: Illustration of leakage current, shown in red, from the gate to the drain of an FET.

  The power loss due to leakage current in CMOS is minute and is rarely considered when determining power consumption.

- *Switching Transient Current*. CMOS gates are composed of pairs of complimentary MOSFETs. When a gate changes state, more than one of these

can be in the *on* state at the same time, resulting in a very brief short-circuit through the gate (see Fig. 3.2).

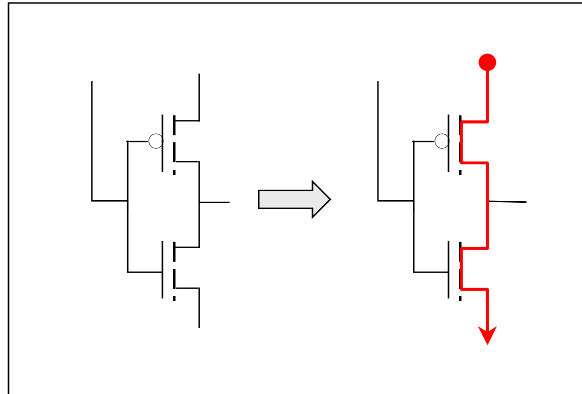

Fig. 3.2: Illustration of switching-transient current due to two complementary FETs being in the *on* state at the same time.

Power loss due to switching transient current is more significant than that resulting from leakage current, and is dependent on the switching frequency of the gate.

- *Load Capacitance Charging Current*. When MOSFET transistors change state, current flow is necessary to charge the capacitance associated with the transistor's gate (see Fig. 3.3).

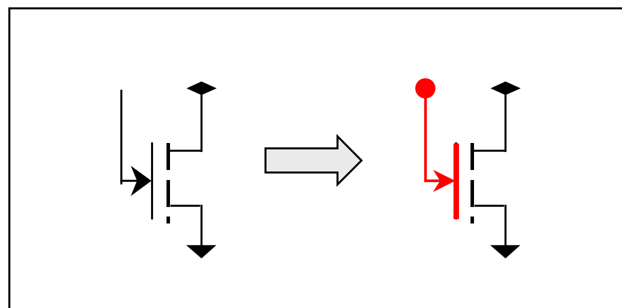

Fig. 3.3: Illustration of capacitance charging current.

The charging and discharging of load capacitances is the dominant form of power consumption in CMOS ICs.

3.2. Capacitance Charging Power

In order to derive the power consumption associated with capacitance charging, two assumptions must be made concerning the CMOS transistors. First, there is a non-zero resistance in the electrical connection to the gate of each transistor. Second, the rate of switching must be such that there is time for the capacitor to completely charge or discharge. Fig. 3.4 shows an equivalent model for the switching of a CMOS transistor.

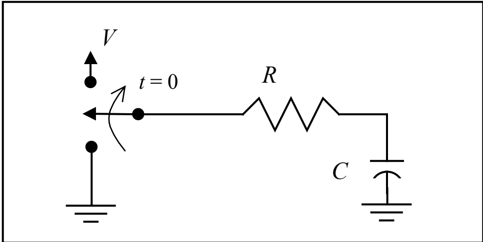

Fig. 3.4: Equivalent gate model for a CMOS transistor.

The source voltage is denoted by *V*, the resistance in the connection to the gate is *R*, and the capacitance of the gate is *C*. The switch in the diagram toggles between *V* and ground in accordance to the logic value of the signal that is driving the transistor gate.

Assume the capacitance is initially uncharged, and the switch moves to *V* at $t = 0$. For $t \geq 0$, the expressions for the voltage drops across the capacitor and resistor are given by:

$v_C(t) = V\left(1 - e^{\frac{-t}{RC}}\right)$ and

$v_R(t) = Ve^{\frac{-t}{RC}}$.

The power dissipation of the resistor is calculated below:

$$p_R(t) = \frac{V^2 e^{\frac{-2t}{RC}}}{R}.$$

Let τ denote the time that the switch remains at *V* before switching to ground. Then the average power dissipated through the resistor is:

$$p_{avg} = \frac{1}{\tau}\int_0^{\tau} \frac{V^2 e^{\frac{-2t}{RC}}}{R} dt = \frac{CV^2}{-2\tau}\int_0^{\tau} \frac{-2}{RC} e^{\frac{-2t}{RC}} dt.$$

Assuming that $\tau \geq 4RC$, then

$$p_{avg} = \frac{CV^2}{-2\tau} e^{\frac{-2t}{RC}}\Bigg|_o^{\tau} \approx \frac{1}{2\tau} CV^2.$$

Therefore, provided there is some non-zero resistance, which there must be, and that the transistor fully changes state, which it must for correct operation, then all of the energy stored in the capacitor is dissipated through the resistor during the time interval τ. Thus, although resistance actually causes the power dissipation, the amount of power dissipated when the transistor changes state is dependent only on the amount of gate capacitance. Note that if the value of τ is decreased (but still satisfies $\tau \geq 4RC$), then the power dissipation increases. This is an intuitive observation that indicates that the faster a transistor is switched (i.e., smaller τ), the greater the power requirement.

3.3. Summary

The three types of current flow in CMOS ICs that were previously described can be partitioned into two categories: static (leakage current) and dynamic (switching transient and capacitance charging current). Dynamic current originates from the process

of switching transistors on and off, where static current is due to imperfect (i.e., “leaky”) transistors. At the speeds that high-performance CMOS devices are commonly driven, dynamic current is by far the more dominant cause of power consumption.

The modeling and prediction of power consumption in CMOS devices is complicated, therefore, by the fact that consumption is strongly dependent upon the dynamic nature of input data [3]. Changes in the relative frequency ($1/\tau$ in the previous derivation) of the device’s data signals can have a significant impact on how much power the IC consumes.

Power prediction is further complicated in configurable CMOS devices, such as FPGAs, because changes in the configuration can affect the structure of internal signals and how the device processes these signals. Other aspects of a configurable device present unique problems not found in non-configurable ICs. For example, in some FPGAs more than 50% of the gates are used to implement the routing fabric that is used to “program” how the chip’s logic blocks are interconnected [10]. As signals traverse this routing fabric, there can be significant power consumption due to the capacitance associated with the interconnections. Thus, there is the potential to optimize routing resources to reduce power consumption for FPGAs and other configurable devices.

CHAPTER IV

PAST RESEARCH IN POWER PREDICTION

4.1. Time-Domain Techniques

As stated previously, the primary source of power consumption in CMOS ICs is due to the current required to charge capacitance associated with each transistor during state transitions. The time-domain representation of all signals is therefore sufficient to estimate consumed power. This approach requires the simulation of time-domain data signals throughout the device, over the entire interval corresponding to the input data stream.

For example, consider the logic function $y = x_1 x_2 \overline{x_3}$, whose CMOS implementation is shown in Fig. 4.1.

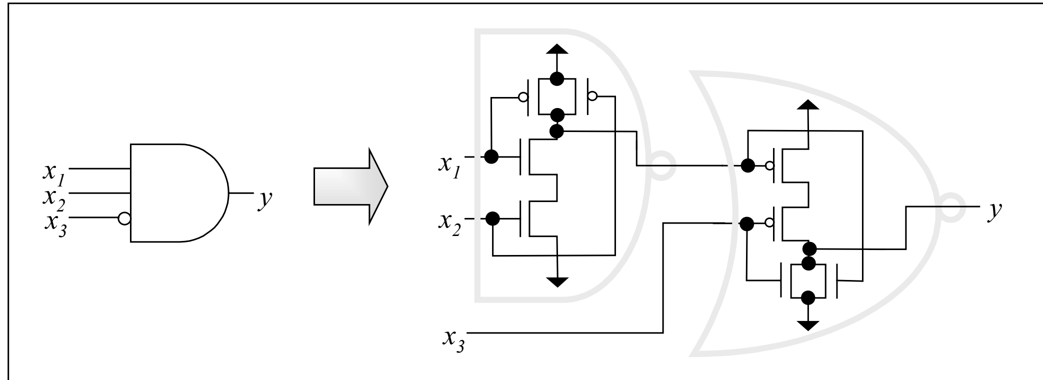

Fig. 4.1: Illustration of a CMOS implementation for the Boolean function $y = x_1 x_2 \overline{x_3}$.

The input signals to the logical gates correspond to voltage signals for the gates of transistors. The time-domain approach to simulate this function requires that the voltages associated with each of the transistors be modeled. This is illustrated in Fig. 4.2.

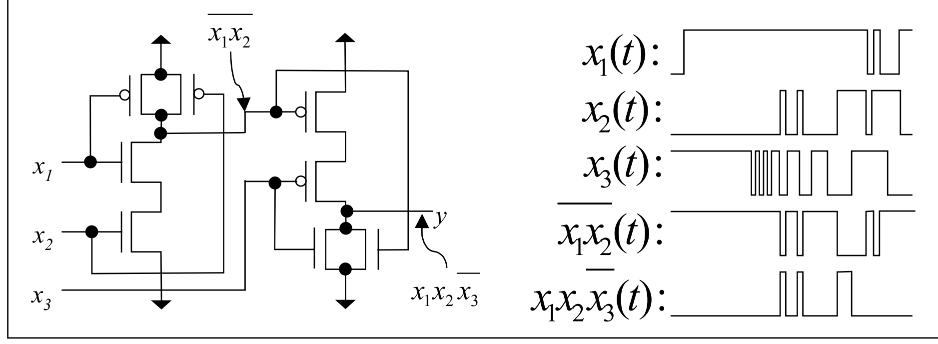

Fig. 4.2: Illustration of time-domain modeling of CMOS circuit signals.

After voltage signals are known for each transistor being modeled, average power is computed for each gate g based on the number of transitions the gate experiences over the period of interest. This value is denoted N_g. For example, the signal $x_1(t)$ shown in Fig. 4.2 transitions five times, so N_1=5. Based on the derivation in the previous chapter, the average power consumed by gate g is

$$\frac{1}{2T} CV^2 N_g ,$$

where T is the length of time over which power is to be estimated.

The average power consumed by all gates in the device is, therefore, given by

$$P_{avg} = \sum_{g \in all\ gates} \frac{1}{2T} CV^2 N_g = \frac{1}{2} CV^2 \sum_{g \in all\ gates} \frac{N_g}{T} .$$

This derivation assumes that capacitances are the same for all gates. The term $\frac{N_g}{T}$ represents the number of state transitions experienced by gate g over the interval of simulation, and serves as a measure of the gate's signal frequency. Let f denote the system clock frequency. For the remainder of this thesis, $\left(\frac{N_g}{T}\right)\Big/ f$ will be denoted as A_g,

which denotes the activity of the signal at gate *g* relative to the gate frequency *f*. Thus, the value of A_g is normalized between zero and one. A value of unity indicates the signal transitions at every clock cycle; a value of 0.25 would correspond to a signal that transitions every fourth clock cycle (on the average).

Calculating A_g is straightforward if the time-domain signals driving each transistor gate *g* are known. However, determination of the exact time-domain signals is computationally expensive, and therefore simulations making use of this method are slow to complete, given the number of signals present a realistic design. If known activities of signals could be used to determine unknown activities of other signals, power calculations could be greatly simplified. This is the basis of the approach discussed in the next section.

4.2. Probabilistic Techniques

To obtain acceptable results in a reasonable amount of time, probabilistic techniques have been proposed for predicting power consumption of CMOS circuits. The idea behind a probabilistic model is to distill important probability-domain information from the time-domain input data. This probability-domain information is then used in calculations, instead of actual time-domain signal values, to estimate average power consumption. The use of probabilistic modeling techniques removes the dimension of time, resulting in a significant reduction in the complexity of IC power calculations.

Figs. 4.3 and 4.4 illustrate how the logical values of time-domain digital signals are represented using two probabilistic parameters: the signal probability and the signal

activity. The signal probability, denoted $p(s)$ for signal s, represents the percentage of time that a signal attains a logical value of one.

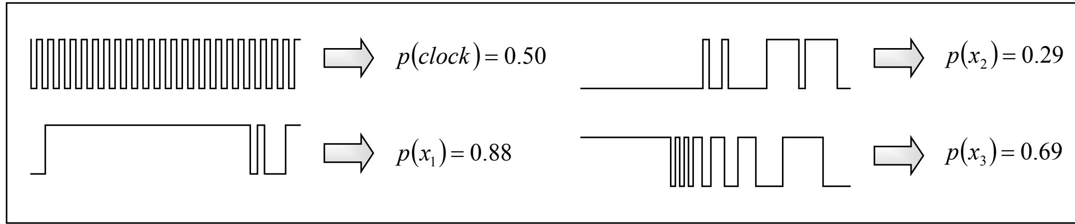

Fig. 4.3: Illustration of signal value probability measures associated with various time-domain signal data.

In their study, Parker and McCluskey [1] describe a symbolic method to relate operations on Boolean data to corresponding operations on probabilistic data. This method allows a digital circuit simulation algorithm to operate on probabilistic information as if it were Boolean data. Below is an example for the Boolean function $y = x_1 x_2 \overline{x_3}$ shown in Fig. 4.1.

$$\begin{aligned} p(y) &= p(x_1)\cdot p(x_2)\cdot(1-p(x_3)) \\ &= (0.88)\cdot(0.29)\cdot(1-0.69) \\ &= 0.079. \end{aligned}$$

In [3], a probabilistic measure of signal frequency (i.e., activity) is introduced. This activity measure is a normalized fraction of the device's clock frequency (see Fig. 4.4).

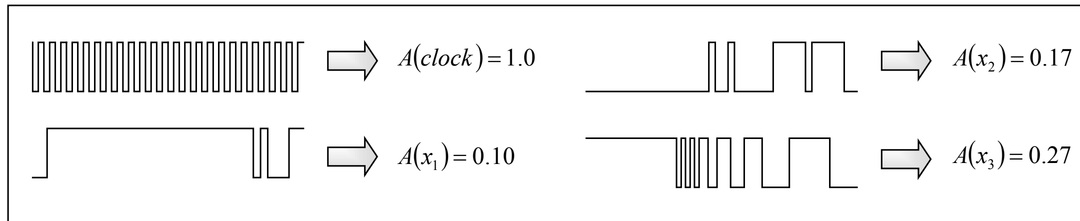

Fig. 4.4: Illustration of signal activity measures associated with various time-domain signal data.

Signal activities are "transformed" as they pass through logic gates (because the underlying time-domain signals are transformed). The transformation associated with activities, which is derived in [11], is more complicated than the probability transformations of [3], and is defined below by equation 4.1.

$$A(y)=\sum_{X}P(X)\cdot\left\{\begin{array}{l}\sum_{i=1}^{n}\left(f(X)\oplus f(X;z_i)\right)P(z_i|x_i)\prod_{j\neq i}\left(1-P(z_j|x_j)\right)\\+\frac{1}{2!}\sum_{\substack{i\neq j\\ i=1,j=1}}^{n}\left(f(X)\oplus f(X;z_i,z_j)\right)P(z_i|x_i)P(z_j|x_j)\prod_{k\notin\{i,j\}}\left(1-P(z_k|x_k)\right)\\+\frac{1}{3!}\sum_{\substack{i\neq j\neq k\\ i=1,j=1,k=1}}^{n}\left(f(X)\oplus f(X;z_i,z_j,z_k)\right)P(z_i|x_i)P(z_j|x_j)P(z_k|x_k)\prod_{l\notin\{i,j,k\}}\left(1-P(z_l|x_l)\right)\\+\ldots\end{array}\right\} \quad (4.1)$$

The symbols of this equation are defined below:

$X=x_1\,x_2\cdots x_n$: Boolean input vector

f : Boolean function for output y

n : number of inputs for function f

$P(X)$: probability of the occurrence of input vector X

$f(x_1\,x_2\cdots x_n;z_1)=f(\overline{x_1}\,x_2\cdots x_n)$

$f(x_1\,x_2\cdots x_n;z_1,z_2)=f(\overline{x_1}\,\overline{x_2}\cdots x_n)$

$f(x_1\,x_2\cdots x_n;z_2,z_3)=f(x_1\,\overline{x_2}\,\overline{x_3}\cdots x_n)$

$P(z_i|x_i)=\dfrac{\frac{1}{2}A(x_i)}{p(x_i)}$.

It should be noted from Equation 4.1 that signal probabilities must be known before the activities can be calculated. Below is an example of activity calculation for the activity value of y shown in Fig. 4.1. The assumed probabilities and activities for x_1, x_2, and x_3 are as defined in Figs. 4.3 and 4.4.

$$\begin{aligned}A(y) &= (0.0264)\left\{(0)+\frac{1}{2!}(0.0564)+\frac{1}{3!}(0)\right\}+(0.194)\left\{(0.0637)+\frac{1}{2!}(0)+\frac{1}{3!}(0)\right\}\\ &+(0.0110)\left\{(0.166)+\frac{1}{2!}(0)+\frac{1}{3!}(0)\right\}+(0.0791)\left\{(0.469)+\frac{1}{2!}(0.295)+\frac{1}{3!}(0.0435)\right\}\\ &+(0.0588)\left\{(0)+\frac{1}{2!}(0)+\frac{1}{3!}(0.0586)\right\}+(0.431)\left\{(0)+\frac{1}{2!}(0.0442)+\frac{1}{3!}(0)\right\}\\ &+(0.0240)\left\{(0)+\frac{1}{2!}(0.115)+\frac{1}{3!}(0)\right\}+(0.176)\left\{(0.130)+\frac{1}{2!}(0)+\frac{1}{3!}(0)\right\}\\ &= 0.0986\end{aligned}$$

Signal activity measures are ultimately used to model signal frequencies at the gates of transistors [3]. This provides a straightforward way to estimate consumed power at the transistor level, as shown in Fig. 4.5.

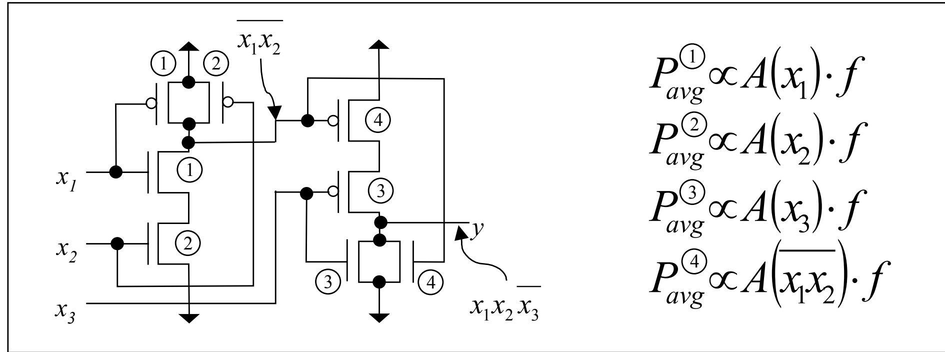

Fig. 4.5: Using signal activity measures to estimate consumed power.

After signal probabilities and activities have been calculated, average power is calculated for each gate using the equation

$$P_g = \frac{1}{2} V^2 C A_g f \ .$$

Note that this equation is virtually the same as the previously derived power equation for time-domain modeling of signal transitions. The individual power values are summed for all gates in the device, yielding the device's power consumption.

4.3. Previous Application of Probabilistic Power Prediction

In [7], a probabilistic method is used to define a cost function to optimize the place and route characteristics of a hypothetical row-based FPGA. A diagram of this row-based FPGA is shown in Fig. 4.6.

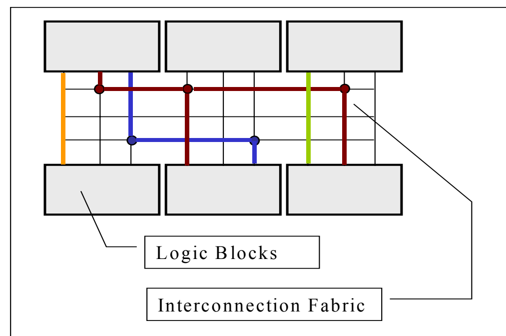

Fig. 4.6: Illustration of the row-based FPGA of [7].

The FPGA of [7] consists of rows of logic modules separated by layers of interconnection fabric. Signals propagate downward, from row to row, through the modules and routing channels of the FPGA.

The study involved optimizing power consumption by altering routing choices made within the interconnection layers. The minimization algorithm was based on simulated annealing. It was shown that up to a 40% reduction in power consumption could be obtained by considering the effects of routing decisions during place-and-route mapping.

CHAPTER V

RESEARCH CONDUCTED

The research involved the design and implementation of a power consumption simulator, based on the model of [7], for the Xilinx 4000 series FPGA [10]. An overview of this device is given in Appendix A. The simulator uses the probabilistic approach to compute power consumption, given the FPGA configuration details and a probabilistic characterization of the input data. The configuration details come from a design compiled by the Xilinx development software. The input data signals to the FPGA are modeled using probabilistic parameters (i.e., probabilities and activities), that in turn are used to calculate the FPGA's internal signal characteristics. The user must supply the probabilistic values for the input data signals to the simulator.

This chapter will detail the design and implementation of the simulator and the results obtained. Problems encountered during the design and implementation will be discussed, as will the solutions to those problems.

5.1. Model Design

5.1.1 Design of Internal Structures

An object-oriented methodology was employed to design the simulator. For purposes of design, the internal structures of the FPGA were divided into the following two categories: blocks that modify signals and an interconnection fabric that route signals. Both of these components consume power, but only the blocks modify signal characteristics.

Classes were created for the FPGA itself and each of its major internal components. The results of the initial design are shown in Figs. 5.1 and 5.2

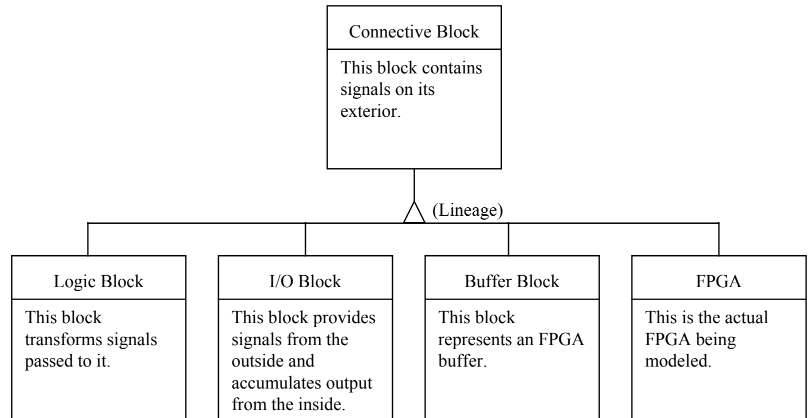

Fig. 5.1: Inheritance diagram of the simulator.

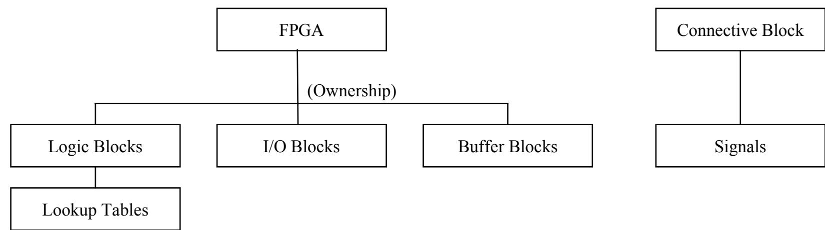

Fig. 5.2: Ownership diagram of the simulator.

Note that the orthogonal and hierarchical nature of the FPGA's internal blocks allows them to be modeled easily.

No classes were created to model the interconnection fabric. One reason for this choice was that the connections between the FPGA's blocks do not alter the signals they carry. Another reason stems from the observation that the Xilinx place and route tool implements shortest-path routing where possible. This implies that the power consumed by a given connection is less dependent upon the exact connection path than on the

Manhattan distance the connection traverses. Therefore, the exact layout of the interconnection fabric was not considered for this study; modeling it would have led to an inefficient (slow) implementation. Because there is no interconnection class for the signal routing information, the necessary probabilistic information is stored in the signal objects themselves. This approach is described below.

5.1.2 Design of Signal Classes

Because over 4000 individual connections can be made within the FPGA, an efficient means of simulating connections through the FPGA was necessary. In addition, the simulated connections must be extensible to future modifications and as hardware independent as possible. In order to realize these objectives, two classes of signals were designed, as illustrated in Figs. 5.3 and 5.4.

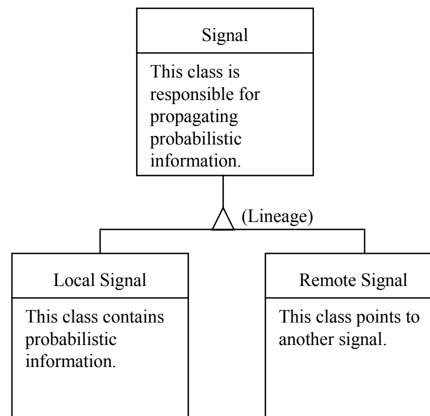

Fig. 5.3: Inheritance diagram of the signal class.

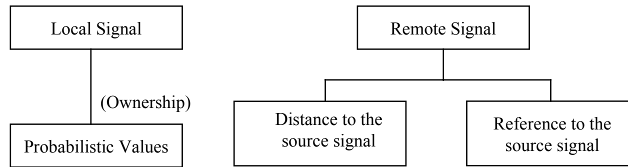

Fig. 5.4: Ownership diagram of the signal class and its descendents.

As can be seen from the illustration, only Local Signals contain probabilistic information. Remote Signals contain a reference to a source signal and the Manhattan distance from that signal. Using this scheme, complex connection networks of high fan-out can be created. An example is shown in Fig. 5.5.

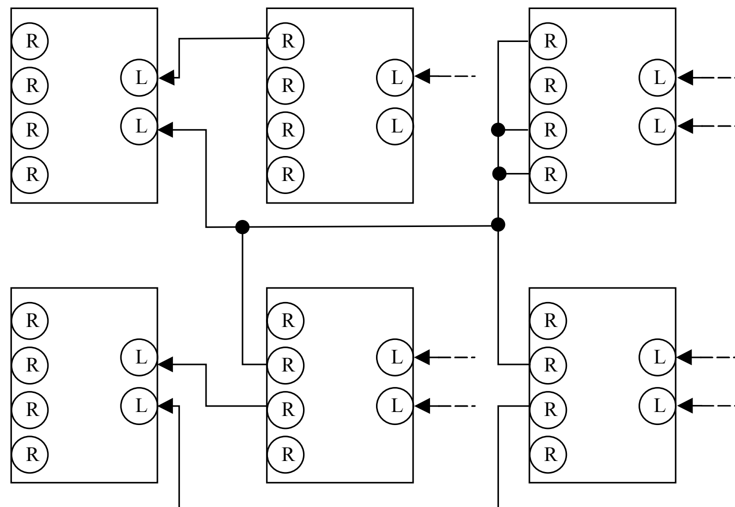

Fig. 5.5: Illustration of Local/Remote signals to create complex, high fan-out connection networks among the logic blocks of the FPGA. Arrows show connections that link Remote Signals back to Local Signals. Signals actually flow in the reverse direction (i.e., signals enter at the left side of the blocks).

In addition to routing signals between Logic Blocks, Local and Remote signals are used to route signals inside the blocks themselves. For example, consider the Logic Block implementation shown in Fig. 5.6. The red lines represent the signal paths associated with a configuration.

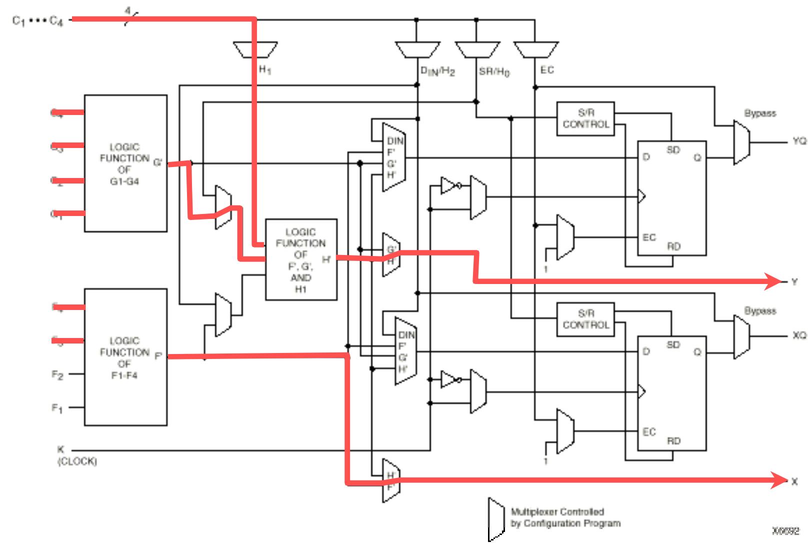

Fig. 5.6: An example of a physical configuration for a Logic Block.

By using Remote Signals to describe the Logic Block's programmable multiplexers, the Logic Block can be modeled as shown in Fig. 5.7.

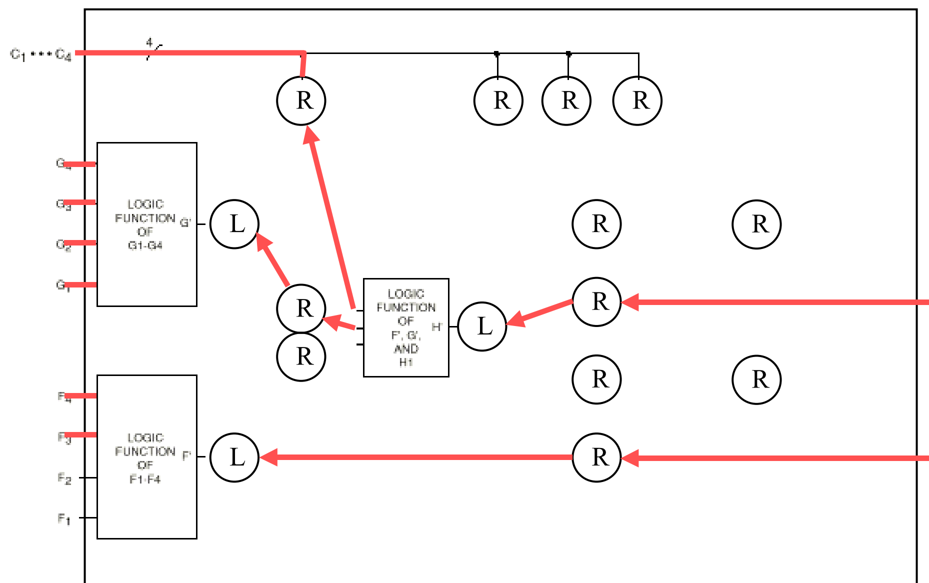

Fig. 5.7: Modeling a Logic Block using Local and Remote Signals.

One can note the following advantages to using this method:

- A signal's probabilistic information can be determined at any point in the connection fabric, even though the actual data is stored in only one place.
- High fan-out connections can be realized without additional overhead or storage requirements.
- The distance a signal has traveled can be computed by summing the distances back to a Local Signal.
- By using Remote Signals to model configuration multiplexers, a block's internal routing can be modeled without the need for additional simulation logic.

Now that propagation of the FPGA's signals has been covered, the actual design of the signals will be explored. Because the primary function of the simulator is manipulating the probabilistic information contained within these signals, most of the model's functionality resides at the signal level. Recall from Chapter IV that there are two probabilistic factors contained inside each signal, the signal probability and the signal activity. The process of calculating these values is distinct, as described below.

Before signal activities can be computed, signal probabilities must be known throughout the chip. Propagation and calculation of signal probabilities is complicated at sites of reconvergent fan-out. At such sites, the input signals do not contain mutually independent probabilities. Gate probabilities, therefore, cannot be computed numerically at these sites. To illustrate this problem, consider the circuit shown in Fig. 5.8.

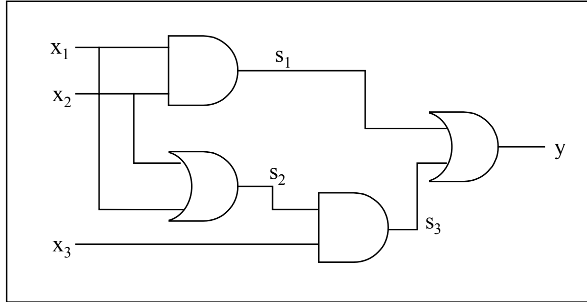

Fig. 5.8: Illustration of reconvergent fan-out.

Table 5.1 below shows two ways to calculate the output probability *y*.

Table 5.1: Symbolic versus numeric calculation of signal probabilities.

Node	Algebraic Probability Expression	Algebraic Probability	Symbolic Probability Expression	Symbolic Probability
x1	a	0.5	a	0.5
x2	b	0.5	b	0.5
x3	c	0.5	c	0.5
s1	ab	0.25	ab	0.25
s2	a+b–ab	0.75	a+b–ab	0.75
s3	ac+bc-abc	0.375	ac+bc-abc	0.375
Y	$ab+ac+bc-abc-a^2bc-ab^2c+a^2b^2c$	0.53125	ab+bc+ac-2abc	0.5

Values in the Algebraic Probability column were computed algebraically from the input values. Values in the Symbolic Probability column were computed using the correct symbolic expressions of the input values as described in [1]. Notice that algebraic calculation of probabilities fails at points of reconvergence. Note also that the algebraic expression for *y* is $ab+ac+bc-abc-a^2bc-ab^2c+a^2b^2c$. Evaluating this expression gives the incorrect result of 0.53125. However, dropping the exponents on all terms in the algebraic expression, as described in [1], yields the correct answer of 0.5. The complication of reconvergent fan-out requires that the FPGA's probability factors be propagated symbolically.

The propagation of symbolic information requires a special data structure to hold and manipulate the information. The approach used is similar to the one described in [1], and takes into account the fact that suppressing exponents on product terms yields correct probability expressions. This approach is outlined below.

- A probability expression is composed of a list of product terms.
- Each product term is composed of an integer coefficient and a string of probability terms.
- Because each probability term is either present or absent, a single bit can be used to represent it.

Using the above approach to encode the expression $ab+ac+bc-2abc$, the representation {1:110, 1:101, 1:011, -2:111} is produced. A list such as this one is present within each Local Signal, and serves as the symbolic probability representation of that signal.

To facilitate the probabilistic transformations that are performed at each Logic Block, the operations of addition, negation, and multiplication were defined and implemented. Given two expressions each having l terms, multiplication and addition of the two symbolic probabilities takes $O(l^2)$ time. Negation takes $O(l)$ time for an l term expression.

Activity factors have a much simpler representation within the model. Because they are not subject to symbolic calculation, they are represented as floating point numbers. However, the activity transformations are as complex as probability transformations, as is evident from Equation 4.1. Given n input signals, the complexity

of evaluating this equation for a single activity factor is $O(n^n)$. Fortunately, because the logic of the FPGA is partitioned into 4-input lookup tables, n never grows above 4.

The actual transformations that both probability and activity factors must undergo are defined by a Xilinx configuration file. The simulator parses this file as it constructs the internal structure and interconnections of the model. The transformations within the file are represented as infix expressions of lookup table inputs. The simulator converts these expressions to postfix and stores them in the Lookup Tables within each Logic Block. These postfix expressions are evaluated at each iteration of the simulator, resulting in new signals being created from old ones.

As noted in the above discussion, there are differences between the internal representation of and set of operations performed on the two signal factors. These differences lead to the two-phase modeling approach described below:

1. Propagate symbolic expressions for signal probabilities throughout the FPGA.
2. Evaluate the symbolic expressions to produce numeric probabilities.

} *Probability Propagation* (steps 1–2)

3. Propagate numeric activities throughout the FPGA.

} *Activity Propagation* (step 3)

4. Compute power consumption.

Note that only after both phases (probability and activity propagation) are complete can power consumption be estimated.

5.1.3 Feedback Resolution

Due to the block-based nature of the Xilinx FPGAs, signal feedback is possible within the chip. This complicates the issue of signal propagation, because the point at which the propagation process should be considered complete may not be readily apparent. To solve this problem, a concept was borrowed from back-propagation neural network learning [13]. Specifically, a delta value was associated with each signal in the simulated chip. As signal propagation progresses, the rate at which signal values mature can be monitored. When a signal ceases to change beyond a predetermined epsilon value, it is considered stable. For this implementation, when over 99% of the signals being propagated become stable, the propagation process ceases. An example of this idea is shown below in Fig. 5.9.

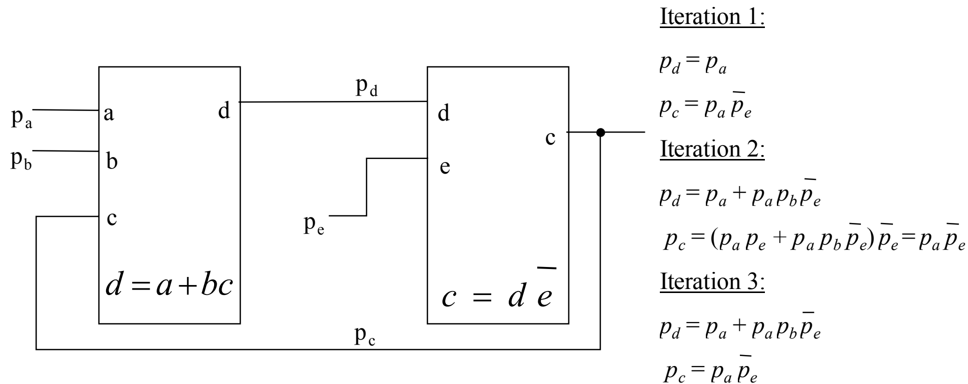

Fig. 5.9: Illustration of symbolic signal feedback and convergence of signals.

In the first iteration shown in Fig. 5.9, the probability value of *c* is unknown, and so it is not taken into account by the system's equations. Thus the initial result of probabilistic propagation may not be completely correct. However, in the second

iteration c has taken on a new value that can be utilized in subsequent calculations. Although the correctness of this new value of c may be questionable initially, it will, through subsequent iterations, take on other more reasonable values. Thus, the probability values in the system will eventually converge and become stable, as is demonstrated in the figure.

Although convergence is illustrated in the example of Fig. 5.9, use of an algorithm such as this raises the general issue of signal convergence. Because the algorithms for transforming probability and activity factors may not have been designed for recursive use, it is not certain whether configurations that contain feedback would converge during simulation. It is possible that the probabilistic factors within the signals could oscillate or diverge during propagation. This was thought to be especially likely for the probability factor, due to its symbolic nature.

5.2. Implementation and Results

5.2.1 Implementation Language

The simulator was implemented in the Java language. The choice of this language centered on the need for platform independence and immediate distribution and testing of the prototype simulator. It was determined through early experimentation that any decrease in execution speed due to the use of Java would not be significant. Some selected listings of the source, which is over 1500 lines of code, are given in Appendix B.

After implementation, the simulator was run with several test cases to measure its effectiveness and utility. Below are some of the results.

5.2.2 Convergence of Signal Stabilities

It was found that signals became stable and converged for all test cases considered. The number of iterations to converge was somewhat dependent on the amount of real estate used by the configuration, and heavily dependent on the amount of feedback present in the design. Fig. 5.10 is a graph of stability convergence results.

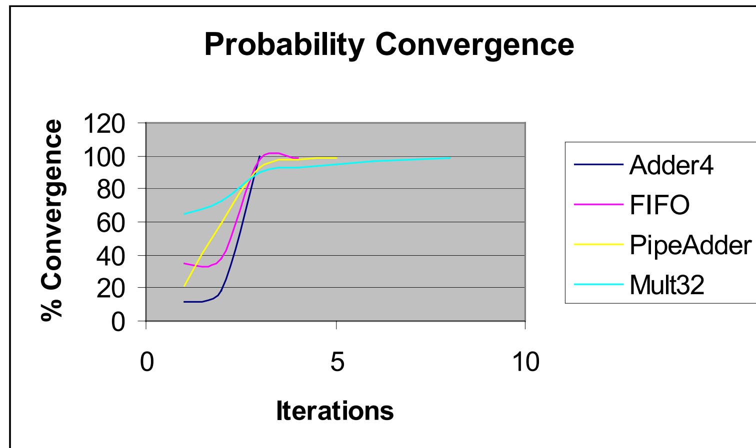

Fig. 5.10: Number of iterations for the symbolic probability to converge.

Note from Fig. 5.10 that all symbolic probabilities rapidly attain a plateau value and then converge more slowly to 100%. This effect is due to the presence of feedback within the configuration. As probabilistic information enters the chip at the start of simulation, it quickly propagates through the entire chip. However, signals whose values depend on feedback will necessarily continue to change slowly until convergence. The configurations that contain the most feedback, a 32-bit multiplier and a pipelined adder (*Mult32* and *PipeAdder*), exhibit a shallower plateau region.

5.2.3 Infeasibility of a Large Number of Inputs

As test cases became more complex, a behavior was discovered that initially was thought to be divergence of signal factors. Upon investigation, however, it was determined to be an explosion in the size of symbolic expressions. It was found that the size of a signal's symbolic probability expressions approached an asymptotic value exponentially related to the number of input signals. In other words, the symbolic probability expressions converged after a given time, but that time grew exponentially with the number of input signals.

To understand why this is the case, recall that symbolic expressions are represented within the simulator as lists of product terms. Each product term can be represented as an *n*-bit number where *n* is the number of input signals. Therefore, there can be at most 2^n distinct terms in each list. In configurations where input signals are thoroughly "mixed" by the FPGA's logic, the resulting product terms will become excessively complex.

For example, it was found that providing 8 or 16 bits of data to a 16-bit FIFO had no effect on the amount of time needed for simulation. This is because a FIFO does not "mix" the input values. However, providing 8, 16, 24, and 32 bits to a 32-bit pipelined adder resulted in a noticeable difference in simulation time and a small difference in the number of iterations required for convergence. Providing a 32-bit multiplier with only 8 bits of non-zero input data allowed the simulator to complete in an acceptable amount of time. However, modeling 32 or even 16 bits on the multiplier resulted in the simulator taking an unacceptable amount of time to finish. This is because the multiplier thoroughly "mixes up" the input signals by multiplying them, creating large and complex

symbolic product terms. In the case just mentioned, some of these symbolic products had more than 500 terms.

5.2.4 Proof of Concept: Measuring Actual Power Consumption

In order to evaluate the concepts on which this study is based, the power consumption of an actual FPGA was estimated during runtime. The Xilinx FPGA was part of a WildOne reconfigurable computing engine [12]. Power from the physical FPGA was estimated by taking temperature measurements from the surface of the chip.

The configuration chosen for this experiment was a 32-bit pipelined adder. Following is a graph showing the temperature of the chip over several operating frequencies (Fig. 5.11). The test was run on data using 50% probability and activity factors.

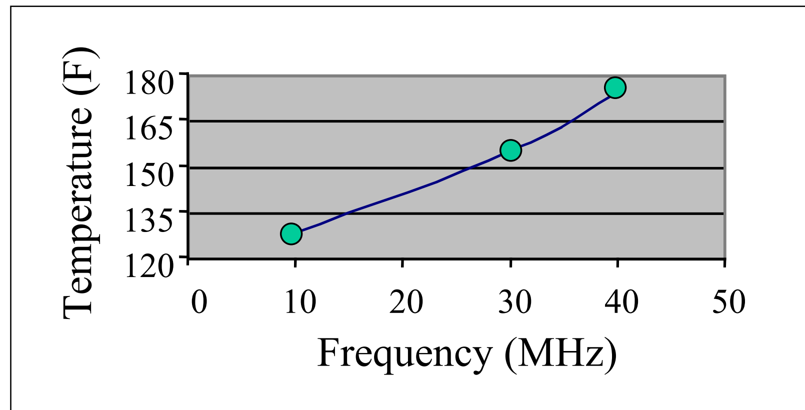

Fig. 5.11: Chip surface temperature over a given frequency range.

As expected, higher operating frequencies cause the chip to consume more power and produce more heat. This is consistent with the output of the simulator, which reports that

power consumption rises linearly with clock frequency. In particular, the output of the simulator for this case is 125mW +43.6mW/MHz.

CHAPTER VI

CONCLUSIONS AND FURTHER RESEARCH

This study has demonstrated the feasibility of using a probabilistic model to compute power consumption for a particular commercially available FPGA. The results of the study show that it is feasible and beneficial to model an FPGA's power consumption probabilistically. In most cases, the simulator was able to complete power calculations in a few seconds. Some of the configurations presented in this study would take days to simulate using time-domain techniques.

The study introduced the concept of how to deal with feedback in probabilistic modeling. The approach used was novel, and has not been found previously in the literature for this application domain. It was shown, empirically, that systems containing probabilistic feedback do converge. This suggests something fundamental about probabilistic feedback, and opens the door for further research and analysis. It would be interesting to discover if probabilistic feedback always converges, or can be proven to converge. A proof of the validity of data that has been fed back through transformations would also be beneficial to the area of probabilistic modeling.

The idea of symbolic probability expressions is well known in the literature, but the complexity explosion that results from increases in the number of input signals has not been discussed much. Because symbolic expressions are important to probabilistic modeling, improvement in this area will be very important as the size and complexity of logic designs grow. Further research in this area might include better ways of

representing symbolic data and different methods for dealing with the reconvergent fan-out problem.

The concepts that this study espouses have been proven through experimentation with the actual chip that the simulator attempts to model. An obvious area of further research is to refine and tweak the model so that it accurately models the power consumption of the FPGA.

REFERENCES

[1] K. P. Parker and E. J. McCluskey, “Probabilistic Treatment of General Combinatorial Networks,” *IEEE Trans. Computers*, vol. C-24, pp. 668-670, June 1975.

[2] Tan-Li Chou, Kaushik Roy, Sharat Prasad, “Estimation of Circuit Activity Considering Signal Correlations and Simultaneous Switching," *IEEE International Conference on Computer-Aided Design*, pp. 300-303, Nov. 1994.

[3] Kaushik Roy, Sharat Prasad, “Circuit Activity Based Logic Synthesis for Low Power Reliable Operations,” *IEEE Trans. VLSI Systems*, vol. 1, no. 4, pp. 503-513, Dec. 1993.

[4] J. Meuhring, J. Antonio, “Optimal Configuration of an Embedded Parallel System for Synthetic Aperture Radar Processing,” *Proceedings, 7th International Conference on Signal Processing Applications and Technology*, pp. 1489-1494, Oct. 1996.

[5] Peter M. Athanas, A. Lynn Abbott, “Real Time Image Processing on a Custom Computing Platform,” *IEEE Computer*, vol. 28, no. 2, pp. 16-24, Feb. 1995.

[6] William H. Magnione-Smith, Brad L. Hutchings, “Configurable Computing: The Road Ahead,” *4th Reconfigurable Architectures Workshop,* Geneva, April 1997, pp. 81-96.

[7] Kaushik Roy, “Power Dissipation Driven FPGA Place and Route under Timing Constraints,” *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, accepted for publication.

[8] Kevin Skahill, *VHDL for Programmable Logic*, Cypress Semiconductor, Addison Wesley, Menlo Park, CA, 1996.

[9] Smith, *Application-Specific Integrated Circuits*, Addison Wesley, Menlo Park, CA, 1997.

[10] Xilinx, “XC4000 Series Field Programmable Gate Arrays,” Xilinx, Inc., San Jose, CA, Sep 18, 1996 (http://www.xilinx.com/partinfo/4000.pdf).

[11] Kaushik Roy, personal communication.

[12] Annapolis Micro Systems, http://www.annapmicro.com.

[13] Simon Haykin, Neural Networks: A Comprehensive Foundation, MacMillan, New York, NY, 1994.

APPENDIX A

THE XILINX FPGAS

The Xilinx series FPGAs consist of a 2D array of programmable logic cells called configurable logic blocks (CLBs). They are interconnected via channels that run vertically and horizontally between the CLBs. A ring of input/output blocks (IOBs) surrounds the array of logic blocks and provides the internal logic of the FPGA access to the package pins. The arrangement of the CLBs and IOBs is illustrated below in Fig. A.1. Note that although a 9x9 array of CLBs are shown in the figure, the actual number of CLBs depends on the particular part number of the device. For instance, the 4028 chip contains a 32x32 array of CLBs.

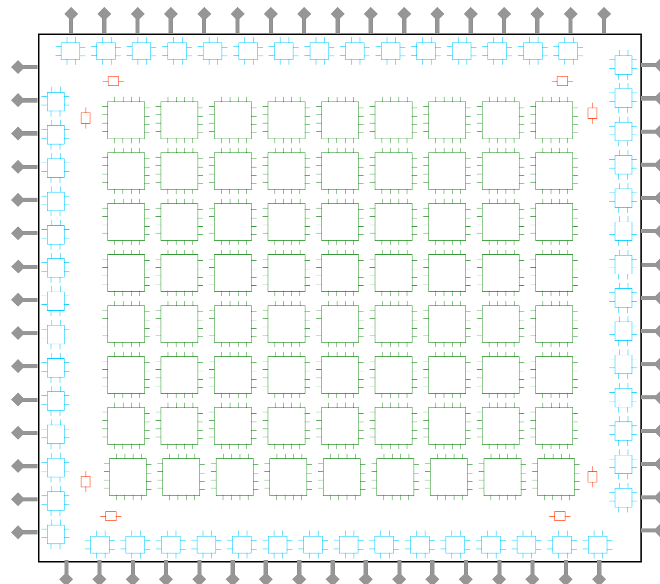

Fig. A.1: Diagram of a Xilinx 4000 series FPGA.

Although there are other components within the Xilinx FPGAs, these are the most important to this study. In the following sections, each of these components is discussed in more detail. The material presented is summarized from [10].

A.1 The XC4000 Series Configurable Logic Block

The primary components of the 4000 series FPGA are the Configurable Logic Blocks. These blocks implement the majority of the logic in the device. The blocks collectively form a "sea of gates" the can be utilized to construct a given implementation. A diagram of a CLB is shown in Figure A.2.

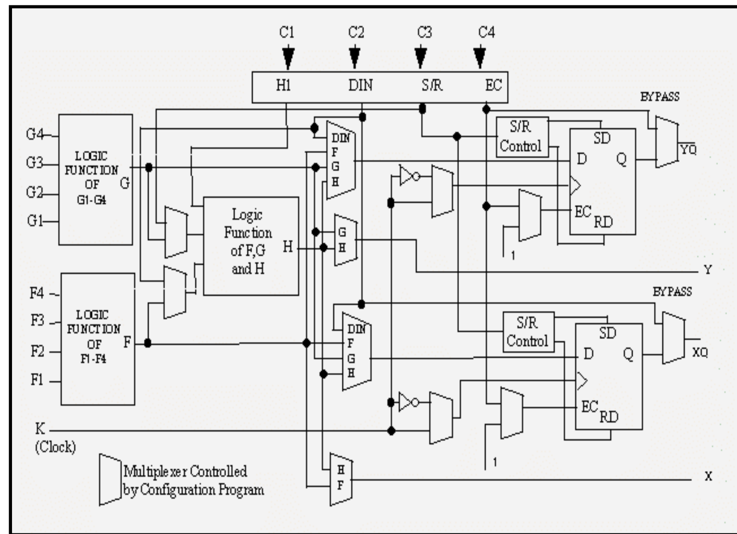

Fig. A.2: A Xilinx 4000 series CLB.

Each CLB contains two independent 4-input function generators (denoted *F* and *G*) and one 3-input function generator (denoted *H*). These function generators are implemented within the CLB as 16x1 lookup tables. They can be configured to function independently, or can work together to implement certain functions with up to nine inputs. Each CLB also contains two flip-flops that can be used to register the function generator outputs. The flip-flops and function generators can be configured to function together or independently.

In order to make the 4000 series FPGAs' performance competitive with that of DSPs, the CLBs contain dedicated arithmetic carry logic in addition to the logic depicted in Fig. A.2. This additional logic is illustrated below in Fig. A.3.

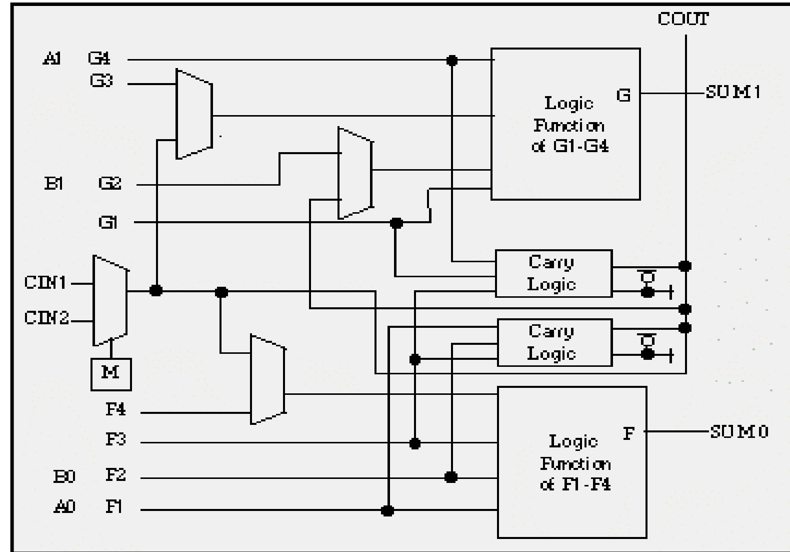

Fig. A.3: The carry logic of a Xilinx 4000 series CLB.

The carry logic can be used to implement fast adders and subtracters. It is commonly enabled during configuration as part of a numerically oriented configuration, and serves as a transparent way to speed up numeric calculations. The carry signals generated by this logic are passed onto the next CLB function generator above or below itself. The routing resources used by carry chains are independent of other routing resources on the chip. The carry circuitry is commonly used where numerical calculations are required, as it is very efficient and renders other speedup methods of only marginal benefit.

A.2 The XC 4000 Series Input/Output Blocks

The IOBs that line the periphery of the XC4000 series FPGAs provide the interface between external package pins and the internal logic. Each IOB is connected to one package pin. An IOB can be configured to handle input, output, or bi-directional signals. Fig. A.4 shows a simplified block diagram of the XC4000 IOB.

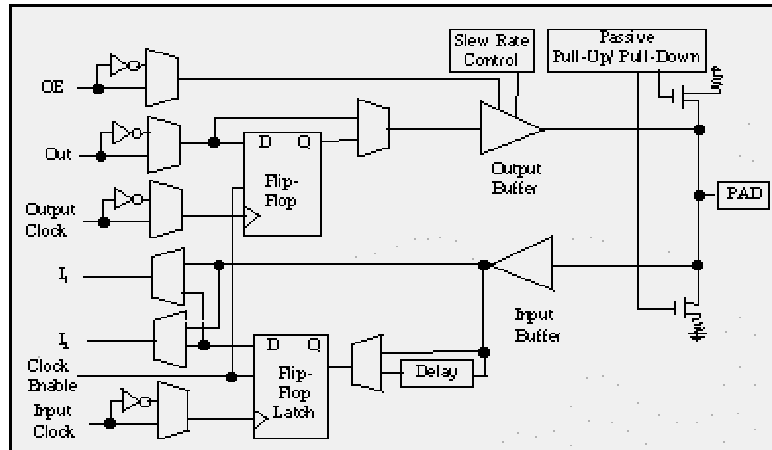

Fig. A.4: A Xilinx 4000 series IOB.

The package pin is labeled *PAD* in the figure. The lines labeled I_1, I_2, and *Out* are connected to the FPGA's interconnection fabric. Registers are provided within each IOB for registering input and output signals, and can be configured as either edge-triggered flip-flops or level-sensitive latches.

A.3 The XC4000 Series Programmable Interconnect

The programmable interconnect consists of a hierarchical matrix of routing resources running vertically and horizontally between the CLBs. The connection fabric also extends outside the CLB matrix to the IOBs. Thus, all components within the FPGA can communicate with each other via the routing fabric.

The routes themselves are composed of metal wire segments and configurable switching points and switching matrices. The switching matrices determine the exact nature of the routing. Fig. A.5 shows a diagram of a switching matrix.

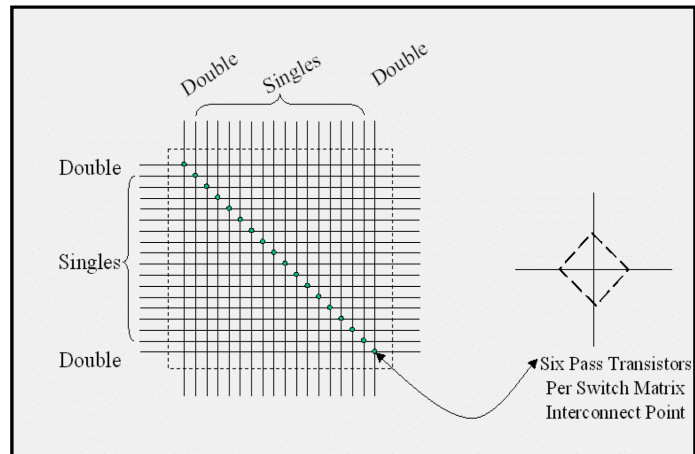

Fig. A.5: Diagram of a Programmable Switching Matrix.

Each switch matrix consists of configurable *n*-channel pass transistors used to establish connections between the wire segments. For example, a signal entering on the right side of the switch matrix can be routed to a wire segment on the top, left or bottom sides, or any combination thereof. There is a switching matrix for each CLB, as shown below in Fig. A.6.

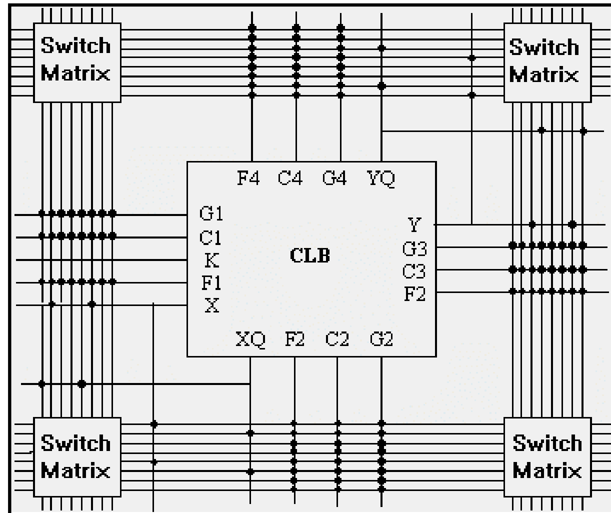

Fig. A.6: Illustration a CLB, routing channels, and associated switching matrices.

As noted before, the routing resources are divided hierarchically into several classes. Single-length lines, for example, run directly from one CLB to the next. This is the situation depicted in Fig. A.6. Double-length lines connect every other CLB, as shown in Fig. A.7.

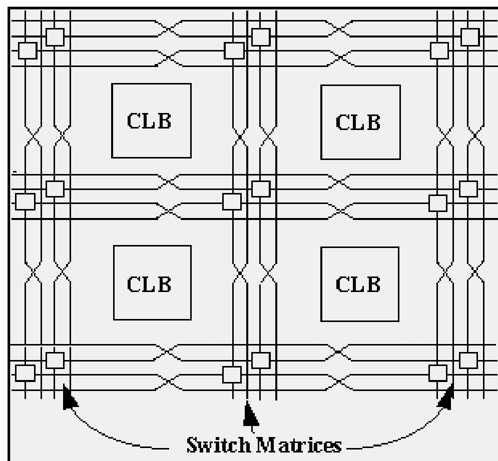

Fig. A.7: Illustration of double-length routing segments.

Note that the switching matrices for double-length lines are staggered on every other block.

In addition to single- and double-length lines, there are Longlines, which run the entire length of the CLB matrix. There are also eight global buffers driving low-skew nets that service the entire chip. These global buffers are most often used for clocks or global control signals. All of these routing hierarchies are combined to provide efficient routing of signals around the FPGA.

APPENDIX B

SELECTED PORTIONS OF THE SIMULATOR SOURCE CODE

This section provides details on a few selected portions of the simulator's source code.

B.1 Symbolic Probability Class

This class models the symbolic probability values that must be propagated through the FPGA. Below is the code, with comments interspersed.

```
/*  This class represents a symbolic probability. */
class SymbolicProbability {
    // root of a linked list of terms...
    private ProbabilisticProductTerm root;

    // constructors...
    public SymbolicProbability() {
        root = null;
        } // end default constructor
    public SymbolicProbability(SymbolicProbability
new_prob) {
        ProbabilisticProductTerm temp = new_prob.root;
        while (temp != null) {
            add_term(new ProbabilisticProductTerm(temp));
            temp = temp.next;
            } // end while
        } // end of copy constructor
    public SymbolicProbability(int pin_num) {
        root = new ProbabilisticProductTerm(pin_num);
        } // end initialization function

    // private methods...
```

Adding a product term to a symbolic expression is complicated by the fact that duplicate terms must be merged. This is demonstrated by the following method, which adds a single term to the object's expression.

```
    /*  This method adds a term to the list.  If the term
already exists in the expression,
        the new term and existing term are combined. */
```

```
    private void add_term(ProbabilisticProductTerm
new_term) {
        // If the coefficient is 0, don't bother adding
it...
        if (new_term.coefficient == 0)
            return;
        // go through the list, looking for this term...
        ProbabilisticProductTerm last = root, temp = root;
        while (temp != null) {
            if (new_term.bits.equals(temp.bits)) {
                // add this term to the current term...
                temp.add_in(new_term);
                // Is this term now a zero?  If so, axe
it...
                if (temp.coefficient == 0)
                    if (temp == root)
                        root = root.next;
                    else
                        last.next = temp.next;
                return; // no need to continue
                } // end if
            last = temp;
            temp = temp.next; // go to the next one
            } // end look through list
        // At this point, the term was not in the list, so
add it...
        new_term.next = root;
        root = new_term;
        } // end of add_term

    // public methods...

    /*  This method multiplies the given signal with the
current one.  The answer is returned
        to the caller.  This signal remains unchanged. */
    public SymbolicProbability multiply(SymbolicProbability
new_signal) {
```

Multiplication of two symbolic expressions is shown below. Notice that duplicate product terms are merged by the previously discussed method.

```
        /*  Multiply each term of both probabilities
together, and add the resulting terms
            to this probability object... */
        ProbabilisticProductTerm i, j;
```

```
        SymbolicProbability result = new
SymbolicProbability();
        i = root; // start at the first term
        while (i != null) {
            j = new_signal.root; // start at the first term
            while (j != null) {
                result.add_term(new
ProbabilisticProductTerm(i.multiply(j)));
                j = j.next; // go to the next term
                } // end while j != null
            i = i.next; // go to the next term
            } // end while i!= null
        return result;
        } // end of multiply
```

Adding two symbolic expressions should take only $O(n)$ time, theoretically. However, because a multiplication is involved to mask out the intersection of the two expressions' terms, it actually takes $O(n^2)$ time, as shown below.

```
    /*  This method adds the given signal with the current
one.  The answer is returned
        to the caller.  This probability remains unchanged.
*/
    public SymbolicProbability add(SymbolicProbability
new_signal) {
        SymbolicProbability temp, result = new
SymbolicProbability(this);
        ProbabilisticProductTerm i = new_signal.root;
        // First, add in the new terms...
        while (i != null) {
            result.add_term(new
ProbabilisticProductTerm(i)); // add this term in
            i = i.next; // go to the next one
            } // end while
        // Next, subtract out the intersection...
        temp = multiply(new_signal);
        i = temp.root;
        while (i != null) {
            result.add_term(new
ProbabilisticProductTerm(i.negate())); // add this term in
            i = i.next; // go to the next one
            } // end while
        return result;
```

```
        } // end of multiply

    /*  This method negates this signal. */
    public SymbolicProbability negate() {
        SymbolicProbability result = new
SymbolicProbability();
        ProbabilisticProductTerm i = root;
        while (i != null) {
            result.add_term(new
ProbabilisticProductTerm(i.negate()));
            i = i.next;
            } // end while
        result.add_term(new ProbabilisticProductTerm(0));
// add +1
        return result;
        } // end of negate

    /*  This method returns a reference to the first term
in the list. */
    public ProbabilisticProductTerm get_first_term() {
        return root;
        }

    /*  This method checks the current probability with the
given one to see
        if they are equal. */
    public boolean equals(SymbolicProbability operand) {
        boolean matching_term = true;
        ProbabilisticProductTerm j, i = root; // start at
the first term

        // Degenerate case...
        if ((root == null || operand.root == null) &&
(!(root == null && operand.root == null)))
            return false;

        // Try to find discrepencies...
        while (i != null && matching_term == true) {
            j = operand.root; // start at the first term
            matching_term = false;
            while (j != null && matching_term == false) {
                if (i.bits.equals(j.bits) && (i.coefficient
== j.coefficient))
                    matching_term = true;
                else
                    j = j.next; // go to the next term
```

```
                } // end while j != null
            i = i.next; // go to the next term
            } // end while i!= null
        return matching_term;
        }

    } // end of class symbolic probability
```

B.2 Probabilistic Product Term Class

This class models the symbolic probability product terms that form the expressions manipulated in the above class.

```
/*  This class is a record that contains a single
probabilistic term. */
class ProbabilisticProductTerm {
    // information for the term...
    public BitSet bits;
    public int coefficient;
    // link to the next term...
    ProbabilisticProductTerm next;

    /*  This copy constructor makes a copy of this object
and returns it. */
    public
ProbabilisticProductTerm(ProbabilisticProductTerm new_term)
{
        bits = (BitSet)new_term.bits.clone();
        coefficient = new_term.coefficient;
        next = new_term.next;
        } // end of copy constructor
    public ProbabilisticProductTerm(int pin_num) {
        bits = new BitSet();
        bits.set(pin_num);
        coefficient = 1;
        next = null;
        } // end of initialization constructor

    /*  This method adds the given Probabilistic Product
Term to this one.  The answer
        is returned to the caller.  This one remains
unchanged. */
    public ProbabilisticProductTerm
add(ProbabilisticProductTerm new_term) {
```

```
        ProbabilisticProductTerm result = new
ProbabilisticProductTerm(this);
        result.coefficient += new_term.coefficient;
        return result;
        } // end of add

    /*  This method adds the given Probabilistic Product
Term to this one.  The answer
        is stored in this one, replacing the previous
answer. */
    public void add_in(ProbabilisticProductTerm new_term) {
        coefficient += new_term.coefficient;
        } // end of add
```

Multiplication of two product terms is effectively a Boolean OR of the two bit vectors that represent the terms. Note that bit 0 is used to represent *power* or a logical *TRUE*. This is needed in many instances during probabilistic transformations.

```
    /*  This method multiplies the given Probabilistic
Product Term by this one.  The
        answer replaces this one. */
    public ProbabilisticProductTerm
multiply(ProbabilisticProductTerm new_term) {
        ProbabilisticProductTerm result = new
ProbabilisticProductTerm(this);
        result.coefficient *= new_term.coefficient;
        result.bits.or(new_term.bits);
        // Is either operand a "1"?
        if (bits.get(0) || new_term.bits.get(0))
            // If both are not "1", then mask the "1" out
of the result...
            if (!(bits.get(0) && new_term.bits.get(0)))
                result.bits.clear(0);
        return result;
        } // end of multiply

    /*  This method negates this term. */
    public ProbabilisticProductTerm negate() {
        ProbabilisticProductTerm result = new
ProbabilisticProductTerm(this);
        result.coefficient = - coefficient;
        return result;
        } // end of negate
```

```
    /*  This method returns the bits associated with the
this term. */
    public BitSet get_bits() {
        return bits;
        }

    /*  This method returns the coefficient associated with
this term. */
    public int get_coefficient() {
        return coefficient;
        }

    } // end of probabilistic product term
```