Back-End Design of a Formal High Level Synthesis System

by

Masud-Ul-Hasan

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER ENGINEERING

June, 1993

Back-end design of a formal high level synthesis system

Hasan, Masud-ul, M.S.

King Fahd University of Petroleum and Minerals (Saudi Arabia), 1993

BACK-END DESIGN OF A FORMAL HIGH LEVEL SYNTHESIS SYSTEM

BY

MASUD-UL-HASAN

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE
In

COMPUTER ENGINEERING

JUNE 1993

Dedicated to

My Parents,

Brothers

and

Sisters

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS

DHAHRAN, SAUDI ARABIA

This thesis, written by

Masud-ul-Hasan

under the direction of his Thesis Advisor, and approved by his Thesis committee, has been presented to and accepted by the Dean, College of Graduate Studies, in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER ENGINEERING

Thesis Committee:

Sadiq Sait M 27/July/93

Chairman (Dr. Sadiq M. Sait)

[illegible]

Co-Chairman (Dr. Khalid M. Elleithy)

[illegible]

Member (Dr. Habib Youssef)

27 Sadiq Sait M.

Department Chairman

[illegible]

Dr. Ala H. Rabeh
Dean, College of Graduate Studies

Date: 27. 7. 93

COLLEGE OF GRADUATE STUDIES

Acknowledgment

In the name of Allah, Most Gracious, Most Merciful. Read in the name of thy Lord and Cherisher, Who created. Created man from a {*leech-like*} clot. Read and thy Lord is Most Bountiful. He Who taught {the use of} the pen. Taught man that which he knew not. Nay, but man doth transgress all bounds. In that he looketh upon himself as self-sufficient. Verily, to thy Lord is the return {of all}.

(The Holy Quran, Surah 96)

First and foremost, all praise to the Almighty Allah Who gave me the courage and patience to carry out this work. I am happy to have had a chance to glorify His name in the sincerest way through this small accomplishment and ask Him to accept my efforts. May He guide us and the whole humanity to the right path *(Aameen)*.

Acknowledgement is due to King Fahd University of Petroleum and Minerals for providing support to this work.

My deep appreciation goes to my major thesis advisor Dr. Sadiq M. Sait for his constant help, guidance and the countless hours of attention he devoted throughout

123

the course of this work . His priceless suggestions made this work interesting and learning for me. He was always kind, understanding and sympathetic to me. Working with him was indeed a wonderful and learning experience which I thoroughly enjoyed.

Thanks are also due to my thesis co-advisor Dr. Khalid M. Elleithy and thesis committee member Dr. Habib Youssef for their interest, cooperation, advice and constructive criticism.

I am also indebted to the Department Chairman, Dr. Samir Abdul Jauwad and other faculty members for their support.

Lastly, but not the least, thanks to my family members for their understanding, relatives, fellow graduate students, and all my friends in the campus from whom I learned a lot and who made the long work hours pleasant.

Contents

List of Figures

123

List of Tables

Thesis Abstract

Name: Masud-ul-Hasan

Title: Back-End Design of a Formal High Level Synthesis System

Major Field: Computer Engineering

Date of Degree: June 1993

A complete design and implementation of a cell library has been accomplished in this work. This cell library supports a formal high level synthesis framework. The library contains the logic level models of all the primitive functions of a Realization Specification Language (RSL). Modular design methodology is employed to support the expandibility of basic cells. Examples of a formal adder, multiplier, inner-product and matrix-matrix multiplier are presented.

Master of Science Degree

King Fahd University of Petroleum and Minerals

Dhahran, Saudi Arabia

June, 1993

خلاصة الرسالة

الاسم : مسعود الحسن
عنوان الرسالة : التصميم الخلفى لنظام نمطى للتركيب العالى المستوى
التخصيص : حاسب آلى
تاريخ الشهادة : يونيو ١٩٩٣

لقد تم تصميم و تنفيذ خلية مكتبية فى هذا العمل. وظيفة هذه الخلية المكتبية هو تعضيد اطار نمطى للتركيب العالى المستوى. وتضم هذه الخليـة المستويات المنطقيـة المختلفـة للوظـائف الاوليـة للغة مواضات التحقيق (RSL). ولقد تم توظيـف اطـار مركبى وذلك لتدعيـم تمدديـة الخلايـا الأوليـة. واخيرا تم عرض مجموعة من الامثلة مثل الجامع، الضاربـة، وحـدة الضـرب الداخلـى، وضاربـة المصفوفات.

درجة الماجستير فى العلوم
جامعة الملك فهد للبترول و المعادن
الظهران - المملكة العربية السعودية
يونيو ١٩٩٣ م

Chapter 1

Introduction

Since the production of complex, very large scale integration (VLSI) chips is very expensive and time consuming, it is essential to detect as many design errors as possible, prior to production. Presently, conventional simulation is the principal technique used to detect functional design errors at an early stage of the design process. However, simulation may leave many errors undetected, since exhaustive simulation of complex circuits and systems is not feasible [Yoe90].

Testing is another popular method. It is used to prove the correctness of systems for specific sets of inputs and outputs. But for the current developments in VLSI, no testing procedure is capable of exhaustively testing the complex circuits due to its limitations [Ell90]. To overcome these difficulties, a number of formal high level synthesis techniques are being developed, and they are likely to become practical tools for detection of design errors.

High level synthesis is a conversion or translation of high-level program like specification of the behavior of a circuit into a structural representation. High level synthesis systems accept a behavioral specification in a hardware description language, or a programming language, or as a data flow graph. A behavior may represent a general purpose system, an application specific system (ASIC), or a combination of them. The structural output of high level synthesis is in the form of netlist of components.

Formal high level synthesis is high level synthesis performed within the framework of a suitable formal system, such as first-order logic, higher-order logic, temporal logic or ASL (Algorithmic Specification Language), etc. In formal high level synthesis system the design specifications (also called formal specifications) can be verified for correctness by applying mathematical rules. In simple words, formal high level synthesis system is a system which transforms the formal specifications to implementable hardware.

Any high level synthesis system has two tightly coupled sub-systems, a **front-end** and a **back-end**. The front-end accepts a high level input description and produces an intermediate form. The back-end takes this intermediate form and produces corresponding VLSI layout. The objective of this work is to make the building elements required by the back-end to generate the VLSI layouts, mainly the cell library. The cell library is usually the central part of the back-end of any cell based high level synthesis system.

The high level language used is **ASL** (Algorithmic Specification Language) and the intermediate language is **RSL** (Realization Specification Language) in the system under consideration [Ell90].

The next section briefly overviews the synthesis and various approaches to it. In Chapter 2, we review the related literature. Chapter 3 discusses the implementation of RSL. Chapter 4 discusses the layout design technique used for the library cells and also a complete example of matrix-matrix multiplier. Finally, the conclusions are presented in Chapter 5.

1.1 Synthesis

The synthesis is an automated procedure by which an implementation is automatically derived from a given specification. It is a transformation between different specifications. Beside being much faster than manual design approaches, the designs produced with the high level synthesis approaches are generally correct by construction. Synthesis reduces the design cycle considerably, allowing the designer to experiment with various design options to obtain the desirable correct design. The synthesis process takes into consideration many engineering aspects such as chip area, power, timing constraints, reliability, and testability [Eve87]. Today, synthesis is a growing industry, and commercial implementations of synthesis systems are widely used for production-level design of digital circuits [WC91].

There are different levels of representations of the digital system, which are, physical, structural or behavioral [MC80, WE85]. The behavioral level is the most abstract level while the physical level is the least abstract level. Depending on these different levels, synthesis can be divided into the following main categories [Muk86]:

- **High Level Synthesis:** High Level Synthesis is a conversion or translation of high-level program like specification of the behavior of a circuit into a structural representation. Structural level consists of a set of Register-Transfer level components, such as ALUs, registers, multiplexers, and their interconnection [MPC88, MPC90, Cam90].

  High Level Synthesis raises the level of abstraction to the algorithmic level, allowing a more behavioral-style specification [WC91]. High Level Synthesis techniques can be classified into two main categories:

  1. Techniques to map a given algorithm to a specific architecture [FK80].

2. Techniques to map a given algorithm to a general architecture [Ell90].

High level behavioral specifications are in general shorter than lower level structural specifications, easier to write and to understand, therefore, less error-prone, and faster to simulate. Thus, these specifications considerably facilitate the design of complex systems [WC91].

- **Logic Synthesis:** Logic Synthesis is a conversion or translation of structural design into optimized combinational logic and maps that logic onto the library of available cells [WHJ86, LBK88]. Logic synthesis is the highest synthesis level currently in wide-spread practical use. The input to a logic synthesis system is a Register-Transfer Level (RTL) description and description of an interconnected set of components such as ALUs, adders, registers and multiplexers.

- **Layout Synthesis:** Layout Synthesis converts an interconnected set of cells, which describe the structure of a design, into the exact physical geometry, i.e., layout, of the design. It involves both the placement of the cells as well as their connections [GK83, WT85].

An integrated synthesis system that covers all the three synthesis levels is often referred to as a *silicon compiler* [WC91].

1.1.1 Silicon Compilation

Silicon compilers are used to obtain a VLSI circuit from different levels of representations of the digital system to be synthesized. High level synthesis can be used as a front-end component in silicon compilers [Ell90]. Silicon compilers derive their name by analogy to software compilation, since the input languages to these may be thought of as being analogous to high level programming languages [Dav90].

1.1.2 High Level Synthesis from Formal Specification

The synthesis system under consideration proposed in [Ell90], uses μ-recursive algorithms to model the behavior to be synthesized. These algorithms can be mathematically verified for correctness before being subjected to the task of translation to architecture and then to corresponding VLSI layouts. Therefore this synthesis system is called as **Formal Synthesis System.**

The high level language used in this system is ASL (Algorithmic Specification Language) and the intermediate language is RSL (Realization Specification Language). The elements of the cell library will be the main constructs of ASL. The Figure 1.1 illustrates the different steps involved in this formal high level synthesis system.

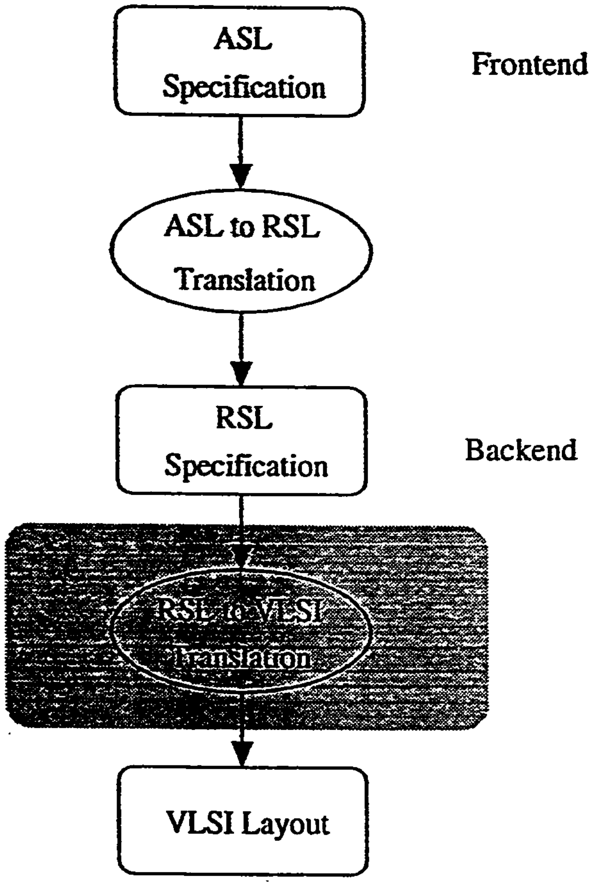

Figure 1.1: ASL based formal high level synthesis system.

This system accepts ASL(Algorithmic Specification Language) as an input and produces VLSI layouts. It has a **front-end** and a **back-end**. The front-end accepts ASL input specification. The ASL can be transformed into RSL which then acts as the specification for the back-end. This back-end would produce VLSI layouts from the RSL specification. The specification method has three initial functions and three operations that can be applied on certain sequences to obtain *any* computable function.

In order to build such a system, it is required that the intermediate representation RSL be transformed into layouts. An algorithm to transform ASL to RSL is also given in [Ell90]. This transformation algorithm is based on using a one-to-one mapping procedure. The algorithm takes an ASL representation and transform each ASL's construct to an equivalent RSL specification and an equivalent architecture implementation.

Chapter 2

Literature Review

The synthesization of hardware from abstract, program-like descriptions have been an active area of research for more than two decades. In this chapter, a review of the different approaches to hardware synthesis is done.

2.1 Synthesis

A number of powerful synthesis tools have been developed for several purposes, mainly direct compilation [DPST81], expert systems [KT83], mixed-integer linear programming [HP82]. Important contributions have also been made in the area of microprocessor synthesis by the CMU-DA project [HP82], the Design Automation Assistant (DAA) [KT83] and the MIMOLA system [Zim79].

The CMU-DA system [HP82] aims to produce a register transfer level design from a behavioral description. Its goals are to convert the behavioral description into an abstract data flow graph, perform optimization on the data flow graph, and select a design style for implementation.

The Design Automation Assistant (DAA) [KT83] uses a knowledge based system approach to aid in the system partitioning, structural selection, and allocation to functional components. The input to the system is a data flow graph derived from a behavioral description, and the output is an architectural description of necessary components and their interconnection.

The MIMOLA [Zim79] system transforms a behavioral description into a set of required hardware resources and schedules them. The target hardware can be restricted and MIMOLA allocates the operations among the defined hardware and generates a micro-program for the hardware. The reviews can be found in [Cam85, Tho86]. Three approaches to correct hardware design i.e., formal verification, synthesis and correctness-preserving transformations are discussed, compared and their relative merits and mutual relationships are investigated in [Eve87]. [CR89] discussed in detail the synthesis of structures from behavioral domain descriptions, including overall synthesis approach, techniques and methods used to solve the main problems.

The behavioral description can be also described in an ordinary programming language. In [GK88] Ada is used to specify the behavior of the hardware. In Flamel [Tri87], Pascal is used to define the required behavior of the hardware. The prototype Flamel imposes some restrictions on the input programs, for example, the input must be a single program without parameters and non-recursive. Only integer, boolean, and one-dimensionless array data types are allowed. Multiplication and division are not allowed. In HARP [TKK89], the behavior of the required hardware is described using FORTRAN. All the synthesis steps can run automatically, except the determination of the facility's wordlength and choosing the function unit allocation strategy. Only certain data types and their operations are allowed. Loops with indefinite iterations are forbidden. Due to these limitations, the original structure cannot be handed down to the synthesis process.

A hierarchically structured framework for analog circuit synthesis called OASYS is presented in [HRC89]. Implementation mechanisms for managing the hierarchy, style selection and translation are also described. The algorithms that translate the behavioral description into structure within a bit-serial compiler are discussed in [HJ88]. The behavioral description includes a full range of logical and arithematic operators and supports previous signal values. A bit-serial (algorithmic) language (BSL) has been used for the input to the compiler.

2.2 Formal Synthesis

The rapid increase in the complexity of the VLSI has generated interest in new methods for synthesizing. One of them is the formal synthesis, using techniques based on a mathematical framework to formally prove that a circuit correctly implements its behavioral specification. The classical text on formal theory and first order logic can be found in [Men64]. A brief introduction to higher-order logic can also be found in [Gor86]. Higher-order logic was originally developed as a foundation for mathematics. [Gor86] shows how it can be used both as a hardware description language and as a deductive system for proving that designs meet their specifications. Examples are also used to illustrate various specifications and verifications techniques. [CP88] has discussed in detail about the different formal systems for hardware representation and also formal logic including first-order logic, higher-order logic and temporal logic.

A more recent introduction to formal logic, which also deals with higher order logics and type theory is also presented in [Hat82]. The use of higher order logic for hardware specification and verification was pioneered by [HD85]. Many of the techniques presented, have been adopted from Ben Moszkowski's work on applying temporal logic to hardware description [HMM83]. Another version of higher order logic can be found in [Gor85]. [Mel88a] contains a brief summary of [Gor85] as well as verification examples, using higher-order logic.

Generally, higher-order logic is distinguished from first-order logic by admitting higher-order predicates, higher order functions and function quantifiers. First-order logic suffices for the expression of many known mathematical theories and has been widely applied for this purpose. But the expressive power of higher-order logic is higher than that of first-order logic. A formal comparison can be found in [Hat82]. The high expressive power of higher-order logic also has its advantages that it easily introduces inconsistencies, and formal reasoning is more difficult than in first-order logic. This difficulty is intended to overcome with the introduction of *types* in higher-order logic. An in-depth exposition of the role of *types* (type theory) in mathematical logic can be seen in [Hat82].

An authentic introduction to temporal logic and its application to the verification is given in [MP81]. Methods of verifying communication protocols, temporal logic, can be found in [Hai82]. For a recent overview of the temporal logic and its applications to verification and artificial intelligence(AI) is given in[Gal87]. The application of temporal logic to hardware verification are also illustrated in [BCDM86, DC86, CLM89, MF85, FTMo85]. The formal specification of the device's behavior must be clear and concise, in order to be seen to reflect the designer's intent. The idea of abstraction is therefore fundamental to formalization of hardware design correctness. [Mel88b] discussed four basic abstraction mechanisms and their formalization in higher order logic, each with an example.

Formal methods produce the maximal benefit in hardware development when the correct technique is identified and embedded in a traditional framework such as simulation or analysis. A first step toward such integration is the use of formal specifications both as simulation tests and as input to appropriate theorem provers. [SBE88] identifies a way in which to obtain maximal length from the use of formal methods in the specification and verification of hardware systems. It attempts to evaluate a set of systems through a case study with two main issues under consideration:

1. Which formalism is most appropriate and,

2. How to make use of the formalism or tool.

It also lists the desirable attributes of formalism intended for use in hardware development. A complete overview on higher order logic is presented in [CS90]. The latest overview on formal theory and temporal logic can be found in [KM91].

A new approach for formal synthesis is presented in [EB90]. It introduces a formal behaviorl framework for synthesis. The given algorithm is represented using a newly developed language, termed Algorithm Specification Language (ASL). ASL is capable of representing any algorithm using a limited number of constructs. Realization Specification Language (RSL) is used to represent the components and connectivity of the synthesized architecture. An algorithm is also given to transform a specification in ASL to RSL.

Chapter 3

Implementation of RSL

3.1 Introduction

The system under consideration accepts ASL as an input and produces an intermediate form RSL. The RSL acts as the specification for the back-end. This back-end would produce VLSI layout from the RSL specification. The specification would have three initial functions and three operations that can be applied on certain sequence to obtain any computable function. In order to build such a system, it is required that the intermediate representation RSL be transformed into layouts. Each basic function is made as a cell and stored in this library. This cell library support the back-end of the cell based formal high level synthesis system.

This chapter deals with the implementation of RSL and with the development of logic level formal cell library as a constituent of the back-end design of the ASL-RSL based formal synthesis system. The hardware models of basic units and their logic level models and simulation outputs of larger functions such as adder, multiplier, inner-product (a×b+c), etc. are presented. The approach used for the implementation of RSL is illustrated in Figure 3.1. It shows that the RSL specifications of a given circuit is translated manually into a logic level model. The cells already stored in the formal cell library are used. This formal cell library contains the basic functions of RSL. In order to make different cells care is taken to achieve modularity and the design can be extended to any desired word length. This logic level model of the given circuit is then simulated using a gate level simulator. The functionally correct model is then given to the input of the physical design system which generates the final layout. This physical design system uses a cell library of pre-designed standard cells. The figures for the RSL architectures are taken from [Ell90].

This chapter deals with the development of logic level formal cell library as a constituent of the back-end design of the ASL-RSL based formal synthesis system.

3.2 Formal Logic Cell Library Design

As mentioned earlier, any synchronous digital system can be expressed using the basic constructs of ASL, viz, zero, projection, successor, composition, recursion and μ-recursion. Corresponding to each construct in ASL is a construct in realization

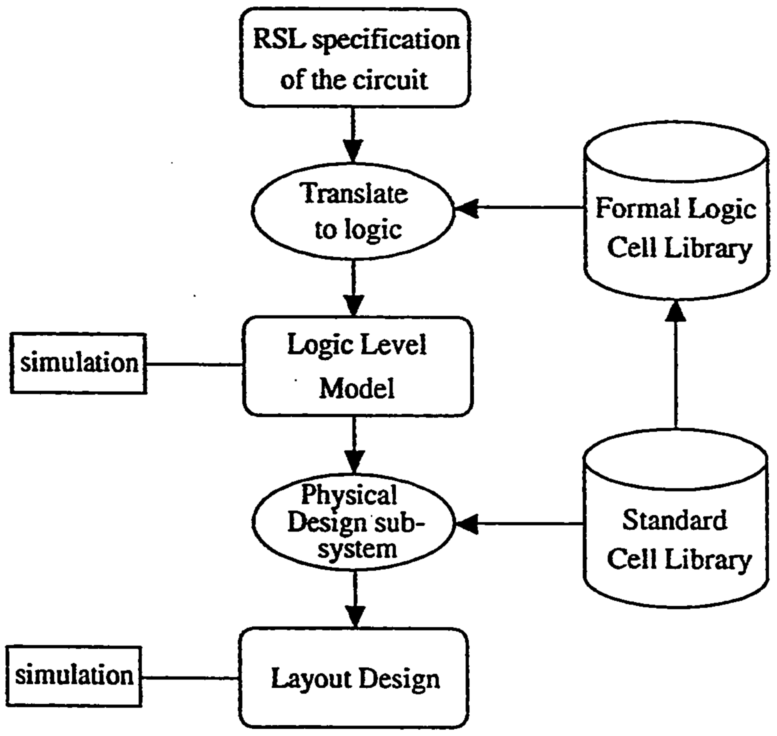

Figure 3.1: Flowchart for the implementation of RSL.

specific language (RSL) that must be mapped to hardware equivalent modules. For each RSL primitive, the library contains a logical description (at the gate/transistor level). Similar to the construction of larger functions at ASL level, with the help of initial elements in the library, it is possible to construct larger logic modules. The synthesized logic circuits are stored in the library for later use. Therefore, for ease of expandibility, they are made modular so that they can be easily connected to build cells of larger functions. Also the design must accommodate varying word lengths. For example, an n-bit successor function is easily made by cascading n 1-bit successor units. Hence, the cell library constructed may be utilized by other systems once the modeling primitives (I/O primitives) are fixed. Thus the cell library designed consists of modular building blocks of logic level macros with the above characteristics. Each primitive logic module (for Zero, Projection, Successor) is carefully synthesized manually. The zero function is a direct connection to ground. The projection function can be easily implemented using multiplexers. The successor function is basically an incrementer that takes an input n and produces an increment $n+1$. The successor function can be used to build an adder, and adder can be used to build multiplier. The adder and multiplier can be used to synthesize the inner-product. The approach used for making the formal cell library is shown in Figure 3.2.

The basic functions, which are *Zero, Projection, Successor*, are modelled as a logic netlist [VLS87]. The logic level netlist is then translated into a transistor-level circuit using a tool called **netlist** and this transistor-level circuit is then translated into a binary file using a translation tool called **presim** [VLS87]. This binary file is required for the simulation done by **rnl**, a switch level simulator [VLS87]. This simulation will verify the correctness of translation. These functionally correct logic level models are then stored in the cell library. Layouts of these functions are then made using a layout design environment such as OASIS [OAS92]. A number of design tools are integrated into the OASIS system. One of them is MAGIC [MAS+90], which is a layout editor. Circuit from layouts are extracted and simulated to verify the functional correctness of layouts. These stored initial functions are used in the definition of larger functions. This approach is useful for building a cell library to support the synthesis of an automated system.

These design tools are briefly discussed in the appendix C.

3.2.1 Clocking strategy

Two-phase non-overlapping clocking scheme is used in the design of all the cells. Input is loaded during ϕ_1 and the output is obtained during ϕ_2. The large modules are made by using the primitive functions or using the small modules which are made by using the primitive functions. Every cell has an input *Control* and an output *Ready*. The input signal *control* is used to determine the starting of the

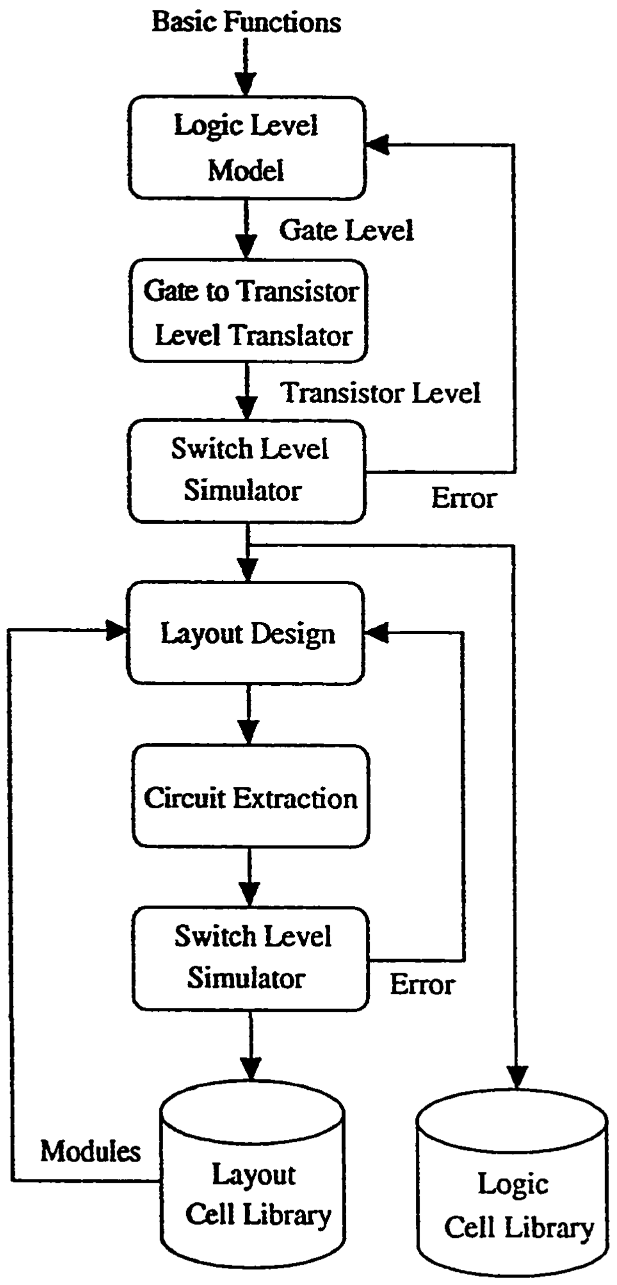

Figure 3.2: Design methodology of the cell library.

operation and the output line *ready* is used to indicate that the circuit has finished its operation. Asynchronous handshaking strategy is used to build the large modules. The cells are connected in this manner that the *Ready* of the cell is connected to the *Control* of the next cell. The cell starts its function when its *Control* gets some signal from the *Ready* of the previous cell.

3.3 Implementation of Primitive Functions

We now describe the primitive functions which are used by the larger functions.

3.3.1 Zero Function

The *zero* function returns the value and has no arguments. A register is used to realize the function which is initialized with the value *zero*.

3.3.2 Projection Function

The *projection* function is used to choose an argument i from n arguments. A multiplexer is used to perform the *projection* function. The n+1 registers are used to store the arguments. An input line *Control* is used to determine the start of the multiplexer operation, and an output line *Ready* is used to indicate that the circuit has finished its operation.

Figure 3.3 shows the hardware of *projection* function. It is a simple 2-to-1 line

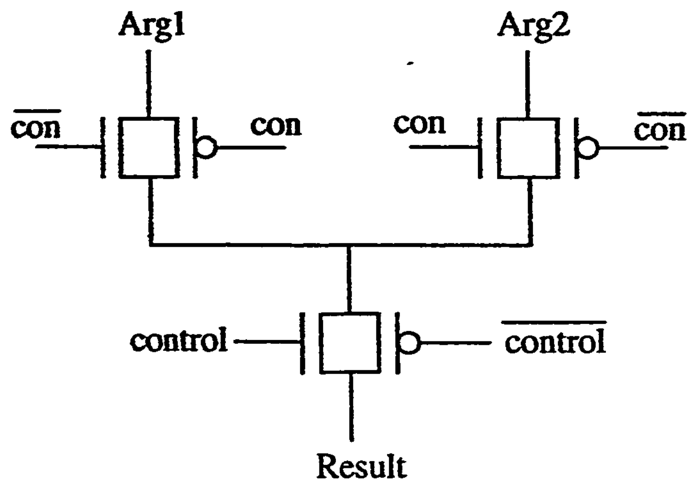

Figure 3.3: Hardware model of 2-to-1 line *projection* function.

multiplexer made by using three transmission gates. There is an input selection line *con* to select one of the two inputs i.e., *in0* and *in1*. The output *out* is obtained by the enable line *control*.

As mentioned earlier, every cell is made modular so that it can be extend to any desired word length. Figure 3.4 shows how this one bit 2-to-1 projection function is extended to 8 bits.

3.3.3 Successor Function

The *successor* function is basically an incrementer that takes an input *n* and produces the output *n+1*. An adder and two registers are used to model the successor function in RSL. One of the registers is for the argument *n* and the other for the value *one*. The operation is done in one clock cycle. The block diagram of a 1-bit successor cell

```
;The Projection Function
;net file for 2-to-1 line projection
;with enable input " control "

(include "oasis_lib.def")
(include "formal_lib.def")

;one cell of multiplexer
(macro mux2(result Arg1 Arg2 con control ready)
        (local conb a b )
     (i1 con conb )
     (i1 control controlb )
     (tgate Arg1 conb con a )
     (tgate Arg2 con conb a)
     (tgate a control controlb result )
)

;net file for 2-to-1 line projection (Bank of n muxs)
;it can multiplex two inputs each of n-bits.
(macro muxn(result Arg1 Arg2 con control ready)
 (repeat i 1 n
     (mux2 result.i Arg1.i Arg2.i con control ready)
 )
)
```

Figure 3.4: Logic level model of 8-bit 2-to-1 line *projection* function.

is shown in Figure 3.5 and the corresponding logic module is shown in Figure 3.6.

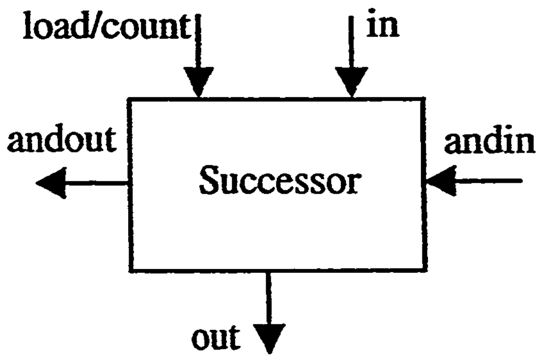

Figure 3.5: Successor function.

This unit is designed as an incrementer and can also be used as an up-counter by feeding back the output of the successor function to one of its two inputs. *Load/Count* control inputs are available to accomplish the necessary function. 2-phase clocking scheme is used in the design of all the cells, input is loaded during ϕ_1 and the output is obtained during the ϕ_2. The cascading of successor units can be done by connecting the *andout* output of a unit to the *andin* input of the adjacent unit as shown in the Figure.

All logic primitives, macro cells, etc., are simulated to verify the correctness of design. The Figure 3.8 shows the simulation results of a 4-bit successor function. As a stimulus, the input is loaded with 7 (0111), on lines in.1-in.4 at the clock ϕ_1. The simulator gives the incremented output i.e., 8 (1000) on lines out.1-out.4 at the clock ϕ_2.

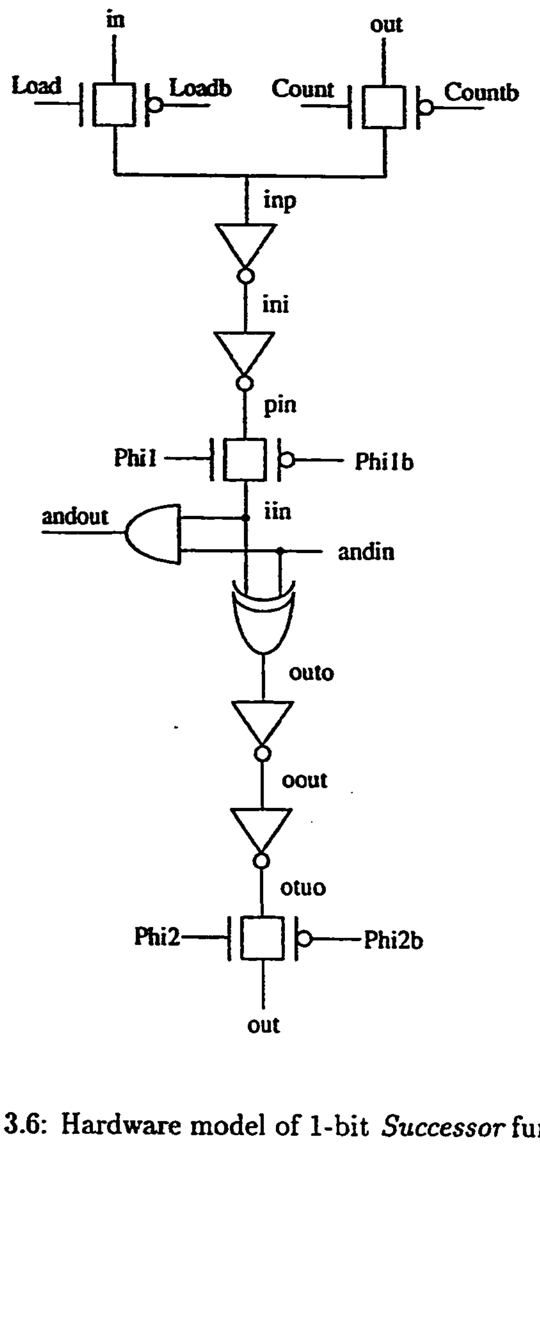

Figure 3.6: Hardware model of 1-bit *Successor* function.

```
; The Successor Function
(include "oasis_lib.def")
(include "formal_lib.def")
(macro successor(out andout andin in phi1 phi2 load count)
          (local a ini oout loadb pin outo otuo countb inp)
     (tgate in load loadb inp )
     (tgate out count countb inp )
     (i1 inp ini )
     (i1 ini pin )
     (tgate pin phi1 phi1b iin )
     (a2 iin andin andout )
     (exor andin iin outo )
     (i1 outo oout )
     (i1 oout otuo )
     (tgate otuo phi2 phi2b out )
)
(node  iin andout andin in phi1 phi2 load count)
(successor  iin andout andin in phi1 phi2 load count)

(macro successorn(out andout andin in phi1 phi2 load count)
  (repeat i 1 n
      (out.i andout.i andin.i in.i phi1 phi2 load count)
  )
)
  (repeat i 2 n
      (connect andout.(i-1) andin.i)
  )
```

Figure 3.7: Logic level model of *Successor* function.

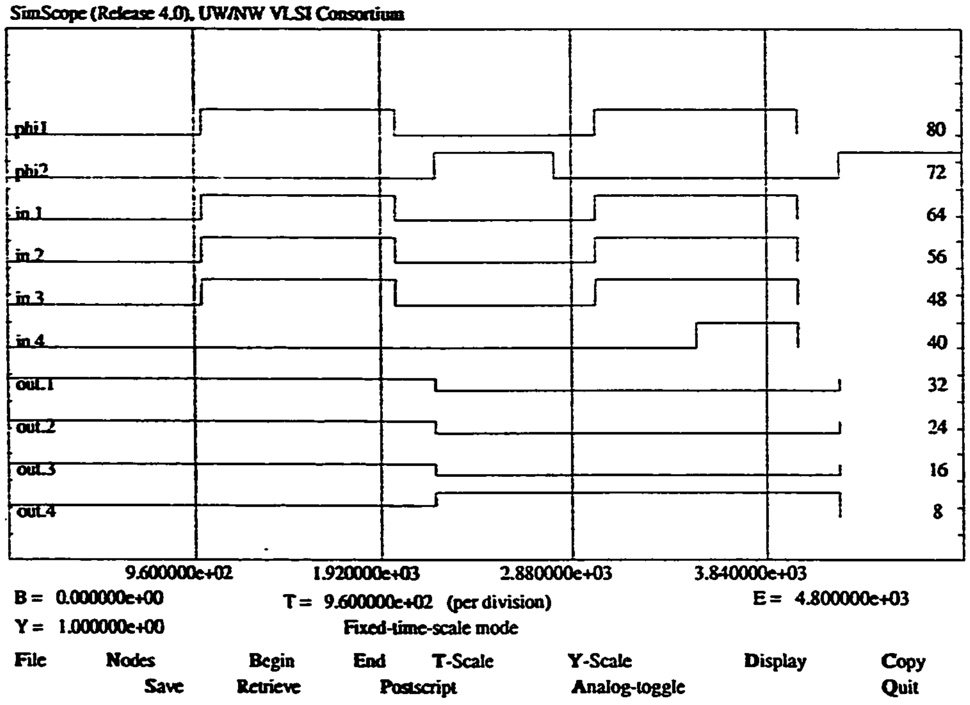

Figure 3.8: Simulation result of a 4-bit *Successor* function.

3.4 Formal Cells

Larger functions are made by using the basic functions. For example, the successor function can be used to build an adder, and adder can be used to build multiplier. The adder and multiplier can be used to synthesize the inner-product. The inner-product is used to implement a matrix-matrix multiplier.

3.4.1 The Add Unit

The unit *add* is used to add two arguments. The direct block diagram representation of *add* function in RSL is shown in Figure 3.9. The actual hardware model of *add* is shown in Figure 3.10. It is implemented using two *Successor* functions and a comparator. Addition is done in a recursive fashion. The *Successor* function which is connected to one of the input of comparator is initialized with an input of *zero*, at the same time other *Successor* function is loaded with one of the two arguments to be added, say n. The other argument, say m of adder, is given to the second input of the comparator. The output *Ready* of the comparator becomes high only when both of its inputs become equal. Both of the successors start incrementing simultaneously and increment m times. When the output value of the successor connected with the comparator reaches the value equal to m, *ready* goes high and both the successors stop incrementing as the *Ready* is connected to the control of both the successors. Evidently the successor function loaded with the initial value of *n* gives the final result as $m+n$. The Figure 3.11 illustrates the logic level model of the *add* function.

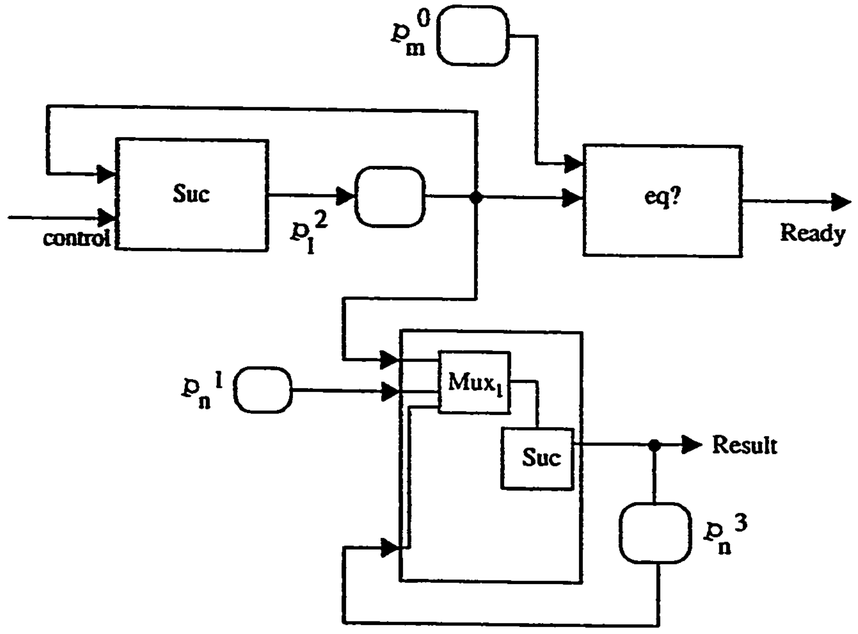

Figure 3.9: The RSL architecture of the unit *add*

The Figure 3.12 shows the simulation result of an 8-bit adder. In this simulation two inputs 15 and 7 are given as the input to the adder and it gives the output 22. It also shows that when the result is obtained *Ready* goes high.

3.4.2 The Pro Unit

The unit *pro* is used to multiply two arguments. The direct block diagram representation of *pro* function in RSL is shown in Figure 3.13. The actual hardware model of *pro* is shown in Figure 3.14. It is implemented by using one successor function,

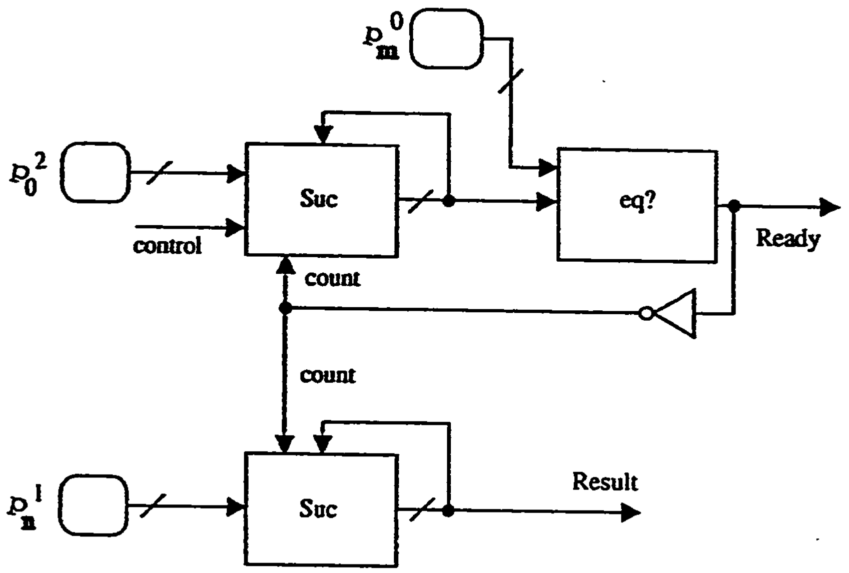

Figure 3.10: Hardware model of n-bit *adder* (add).

one comparator, one add unit and some logic gates as control circuitry. The multiplication is done again in a recursive manner. Suppose m is to be multiplied with n. The successor function which is connected to one of the input of comparator is initialized with an input of *zero*. The argument m is given to the second input of the comparator. The unit *add* is loaded with two arguments *zero* and n. The unit *add* adds n to n, m times, giving the final result $m \times n$. The Figure 3.15 illustrates the designed logic level model of the *pro* function.

The Figure 3.16 shows the simulation result of an 8-bit multiplier *pro*. It also shows that when the result is obtained *Ready* goes high. In this simulation two inputs 3 and 4 are given as the input to the multiplier and it gives the output 12. It also shows that when the result is obtained *Ready* goes high.

```
;The unit "add"

(include "oasis_lib.def")
(include "formal_lib.def")

(macro addn(out1 in phi1 phi2 load b ready in1 load1 )
       (local out count count1 )

;generate "ad1"
  (ad1 out in phi1 phi2 load count b ready)

;generate counter
  (successorn out1 in1 phi1 phi2 load1 count1 )
          (connect readyb count )
          (connect readyb count1 )
)
(node out1 in phi1 phi2 load b ready in1 load1 )
(add out1 in phi1 phi2 load b ready in1 load1 )
```

Figure 3.11: Logic level model of an 8-bit *add* function.

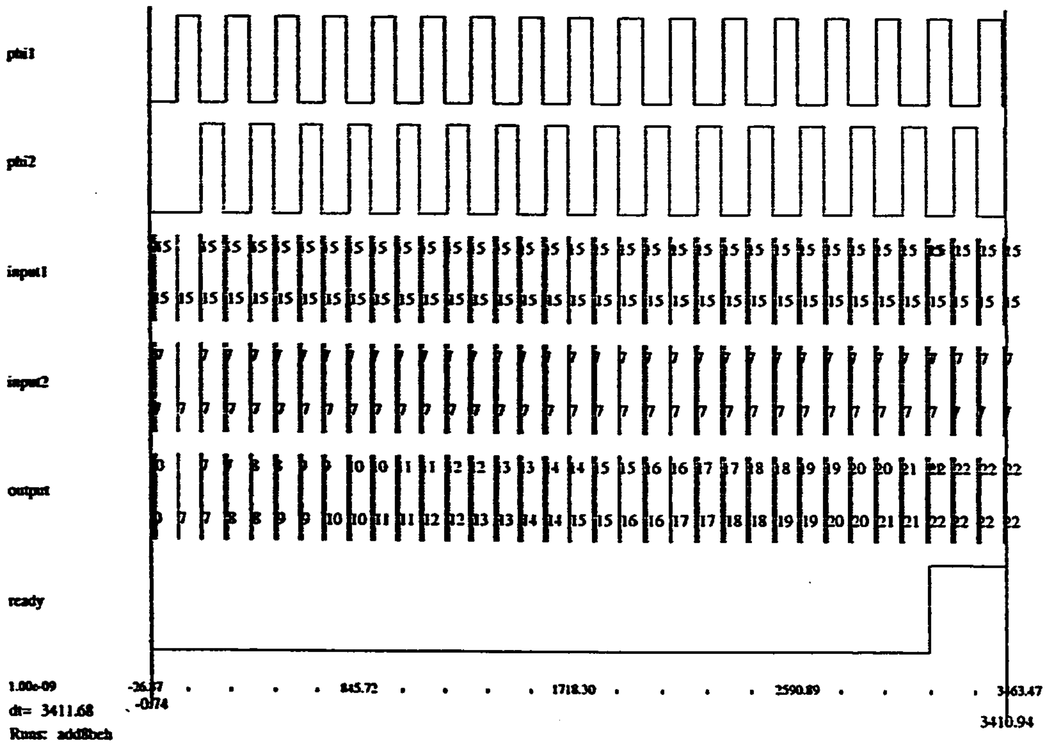

Figure 3.12: Simulation result of an 8-bit *add* function.

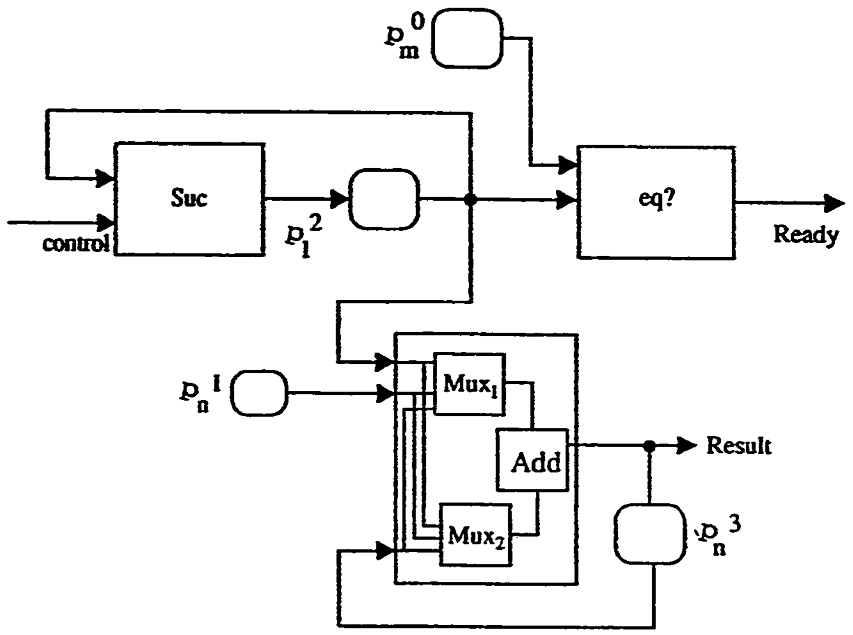

Figure 3.13: The RSL architecture of the unit *pro*

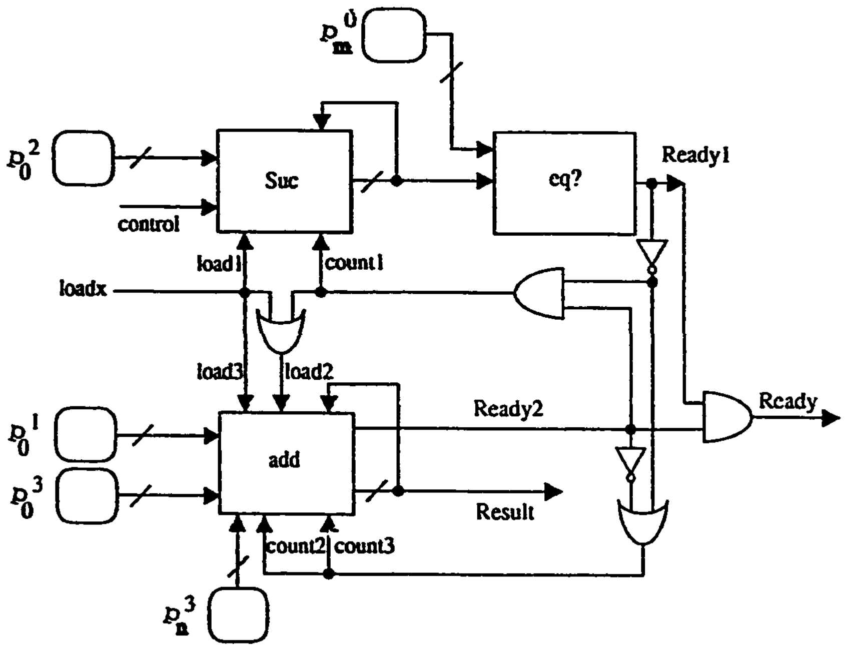

Figure 3.14: Hardware model of n-bit *product* (pro).

```
;The unit "pro"

(include "oasis_lib.def")
(include "formal_lib.def")

(macro pron( out3 in1 phi1 phi2 load b b2 ready )
    (local load1 load2 load3 count1 count2 count3 out2)

;generate "adl"
(adl out1 in1 phi1 phi2 load1 count1 b ready1 )

;generate "dff"
(dff ready1d phi2 ready1)

;generate "add"
(addn out3 out2 in3 phi1 phi2 b2 ready2 ready2b in2)
         (connect load load1 )
         (connect load load3 )
         (a3 ready1b ready2 phi2b count1 )
         (o2 load count1 load2 )
         (o2 ready1b ready2b count3 )
         (connect count2 count3)
         (a2 ready1d ready2 readyt )
)
(node out3 in1 phi1 phi2 load b b2 ready )
(pron out3 in1 phi1 phi2 load b b2 ready )
```

Figure 3.15: Logic level model of 8-bit *pro* function.

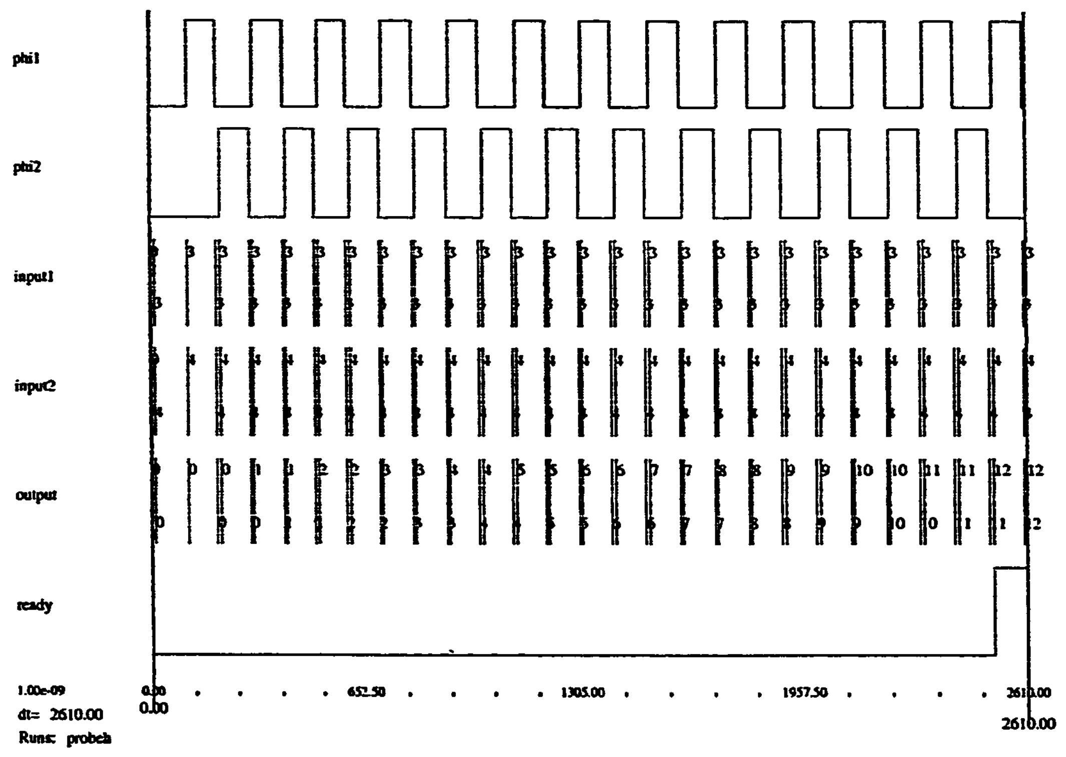

Figure 3.16: Simulation result of an 8-bit *pro* function.

3.4.3 The Inner-Product Unit

The unit *inner-product* is used to multiply two arguments and add the third argument to the result of multiplication. The direct block diagram representation of *inner-product* function in RSL is shown in Figure 3.17. The actual hardware model of *inner-product* is shown in Figure 3.18. It is implemented by using one add circuit and one pro circuit. Also some logic gates as control circuitry. The *pro* multiplies two arguments in a recursive manner and the *add* adds the third argument to its result. Figure 3.19 shows the logic level model of the designed *inner-product* cell.

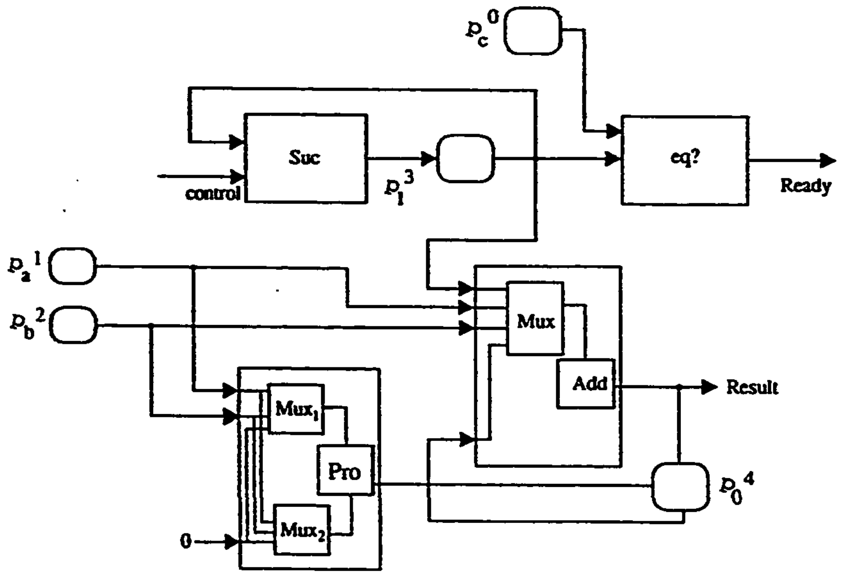

Figure 3.17: The RSL architecture of the *inner-product* cell

Figure 3.20 shows the simulation result of an 8-bit *inner-product* cell. In this example, *a* is taken as 3, *b* as 2 and *c* as 4. The result *output* is equal to (a × b + c) = 10.

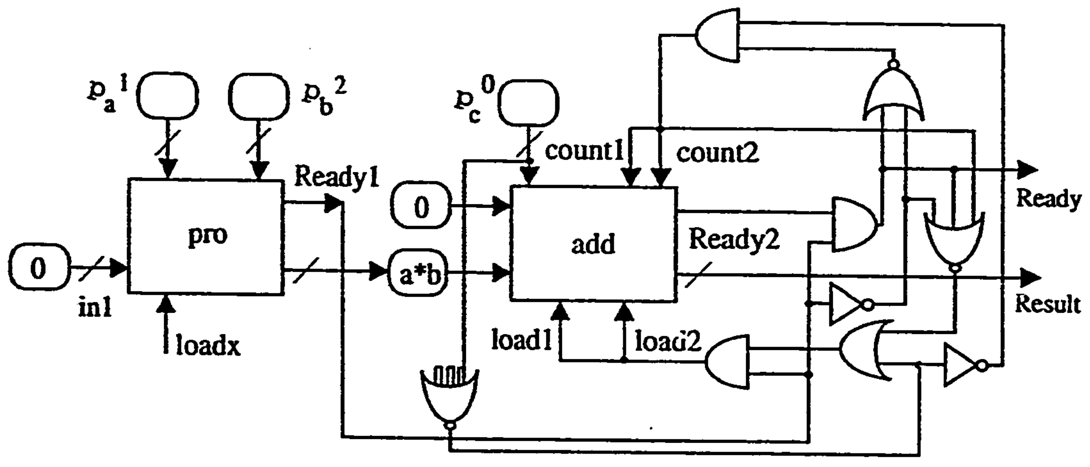

Figure 3.18: Hardware model of n-bit *inner-product.*

```
;The unit "inner-product"
;with 3 inputs a, b, and c.
;result = (a * b + c)

(include "oasis_lib.def")
(include "formal_lib.def")

(macro inner(out3 in1 in5 phi1 phi2 load b4 b3 b2 ready)
     (local rb q load1 load2 count1 count2 counti)
;generate "adl"
       (adl out in5 phi1 phi2 load1 count1 b2 ready0)
;generate counter
        (successorn out3 in4 phi1 phi2 load2 count2 )
;generate "pro"
        (pron out2 in1 phi1 phi2 load b3 b4 ready1)
    (a2 ready0 ready1 ready )
    (i1 ready1 r1b )
    (oi2 ready r1b counti )
    (oi3 r1b ready count1 rb )
    (oi2 rb c0 rb1 )
    (i1 rb1 rb2 )
    (a2 ready1 rb2 load1 )
    (connect load1 load2)
    (o4 b2.1 b2.2 b2.3 b2.4 c01 )
    (o4 b2.5 b2.6 b2.7 b2.8 c02 )
    (o2 c01 c02 c0b )
    (a2 c0b q count1 )
    (connect count1 count2)
  (repeat i 1 n
     (connect out2.i in4.i)
  )
)
;generate globle node names
(node out3 out2 in1 in5 phi1 phi2 load b4 b3 b2 ready)
(inner1 out3 out2 in1 in5 phi1 phi2 load b4 b3 b2 ready)
```

Figure 3.19: Logic level model of an 8-bit *inner-product.*

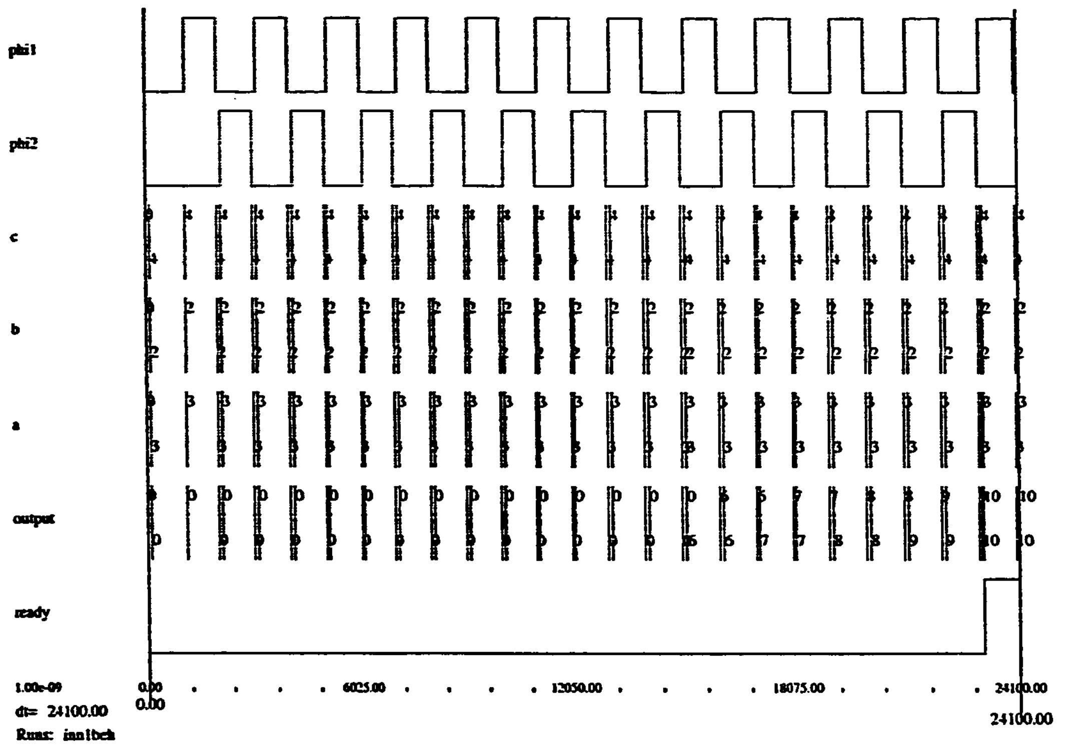

Figure 3.20: The simulation result of an *inner-product* cell

Chapter 4

Cell Layout Design

4.1 Introduction

The previous chapter discussed about the development of the logic level formal cell library. This chapter deals with the layout design methodology of the cells stored in the formal cell library as well as the layouts of the formal cells which used this basic cells in their definition.

4.2 Design Methodology

In a cell based VLSI layout synthesis system, the heart of the physical design unit is the cell library. The approach for such a synthesis system is illustrated in Figure 4.1. The logic level model, using formal logic cell library, of the given circuit is fed to the physical design system. has Two sub-systems of the physical design system are used. The first sub-system is used for placement and routing using standard cell library. It places the cells and performs the global and detailed routing in a plane so as to minimize the layout area. The output of the first sub-system is given to the other sub-system. It assembles the final physical layout. The circuit is then extracted and simulated to verify that the hardware description of the chip under design performs the intended function.

4.3 Physical Design Sub-System

Layouts are made using a layout design environment such as OASIS [OAS92]. A number of design tools are integrated into the OASIS system. OASIS is an abbreviation of Open Architecture Silicon Implementation Software. It is a cell-based system for IC design. The tools integrated into the OASIS system have been developed to automatically translate high-level descriptions of integrated circuits into testable physical layouts, using predesigned standard cells.

One of the premises of the OASIS System is the modularity of the software. New, improved algorithms can be easily substituted in place of the old one's . The entire system is controlled with a single data flow supervisor program to assure data consistency is maintained at all stages of the design. The data flow supervisor is template-driven, the templates used in OASIS can be easily expanded to support

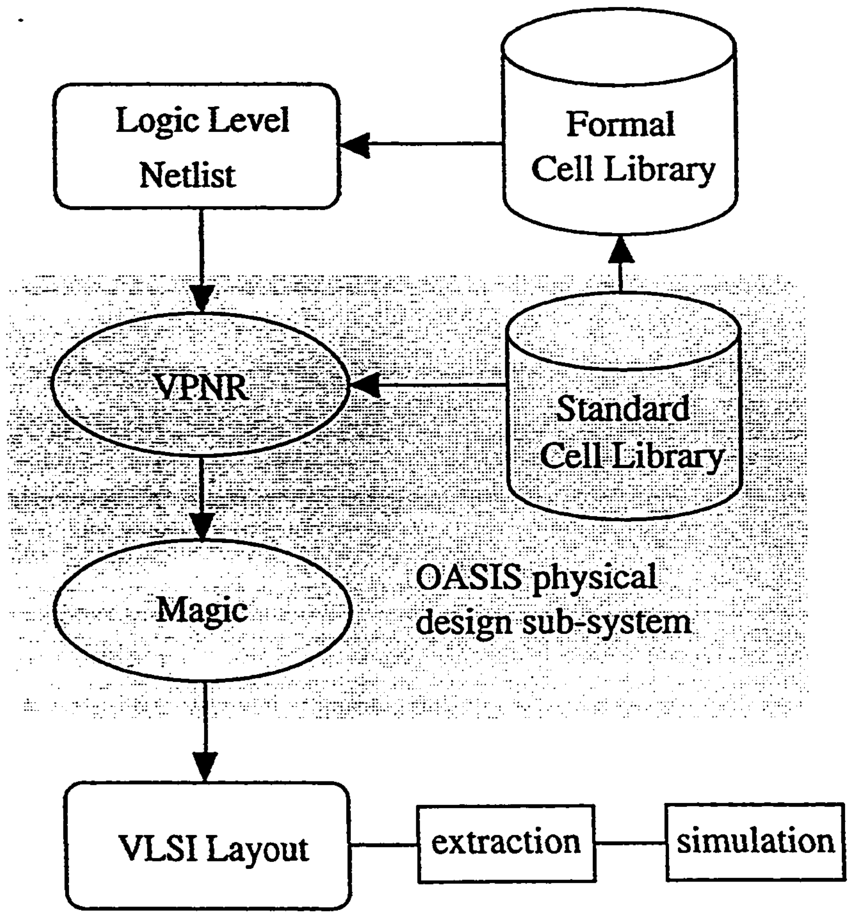

Figure 4.1: Approach for cell based VLSI layout synthesis system

additional software tool, thus providing the desired openness of the system.

4.3.1 Standard cell approach

The layouts produced with the OASIS system utilize standard cells of uniform height. The cells are placed in an array of horizontal rows and all interconnection of signal nets are made by channel routing in the space between the adjacent rows. All connections of the power nets, *VDD* and *GND*, are made by abutting the cell horizontally.

Signal nets connecting cells belonging to non-adjacent rows cross the intervening rows of the cells by utilizing feed through pins (vertical strips of metal running from the top to bottom of a cell) built into some cells, or by inserting feed through cells (cells consisting solely of a single feed-through) whenever appropriate.

A set of scalable CMOS cells compatible with the 2μ SCMOS technology are used. Cell signal input and output ports are made in the second layer of metal, while the power net (VDD and GND) ports are made in the first layer of metal.

The VDD and GND power nets from each row of cells are connected together using power rails running vertically through the entire height of the layout.

4.3.2 Placement

The placement and routing in OASIS is done by a layout subsystem of OASIS called VPNR (Vanilla Place aNd Route). The tools comprising VPNR create physical layouts automatically from netlist descriptions of logic circuits, using a library of pre-designed standard cells of uniform height. The goal of VPNR is to place cells and perform global and detailed routing of interconnections in a plane so as to minimize the layout area.

VPNR is usually invoked at the end of the entire design process after the design has been simulated and verified to implement the desired functionality. Occasionally, the layout may be generated in the early stages of the design to obtain an estimate of the area taken up by the circuit.

VPNR employs placement and global routing algorithms based upon the quadrisection paradigm [SK87]. The combined placement and global routing program receives a netlist of standard cells, partitions it into four quadrants (top-left, top-right, bottom-left and bottom-right), and processes each quadrant in the same manner until each quadrant contains one cell row in the vertical direction. Partitioning is accompanied and directed by approximate global routing.

4.3.3 Routing

After the positions of the cells in rows are determined, the interconnections comprising the scan chain can be chosen. The algorithm applied here is a simple snake-like path threading through the cells consisting solely of a single feed-through wherever appropriate.

The next stage is the detailed global routing. The algorithm constructs a minimum spanning tree for each net, finds the exact crossing locations for nets that need to cross the cell rows, insert feed-through cells if necessary, and assigns sub-nets in channels. The nets are processed sequentially, and the routing of each net takes into account all nets routed previously. After all nets are routed, the global router prepares the data for detailed channel routing.

The global routing derived at the preceding stage does not include the global nets disregarded at the placement stage. These nets are assumed to follow a fixed routing scheme. They cross the channels vertically in the vicinity of the vertical power rails and extend into the channels horizontally to the left and right of the vertical rails. Usually the horizontal extensions of a given net occur in every second channel.The routing of the global net involves inserting the vertical power signals together with the adjacent feed-through cells for the global nets, and adjusting the data for channel routing to accommodate the vertical inserts.

Each channel definition contains a list of groups of pins that will be connected. The channel router determines where to put the wires so the resulting layout occupies the least amount of area. VPNR provides a choice of two channel routers: a greedy router [Ros85] and a left edge based router with channel compaction [LNR88].

4.3.4 Simulation and Verification

The entire purpose of this phase is to verify that the hardware description of the chip under design performs the intended function. The layouts made by the magic can be extracted and simulated using a simulation tool called **irsim** [MAS+90].

4.4 Layouts

Layouts of the different units discussed in the previous chapters are synthesized using layout design environment OASIS [OAS92]. The VLSI layouts of cells are implemented using SCMOS technology. The VLSI layout of successor, adder, multiplier, inner-product and the matrix multiplier are shown in the following figures.

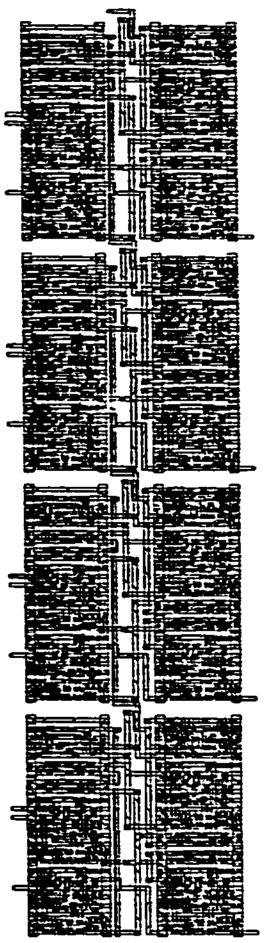

Figure 4.2: VLSI layout of 4-bit *successor* unit.

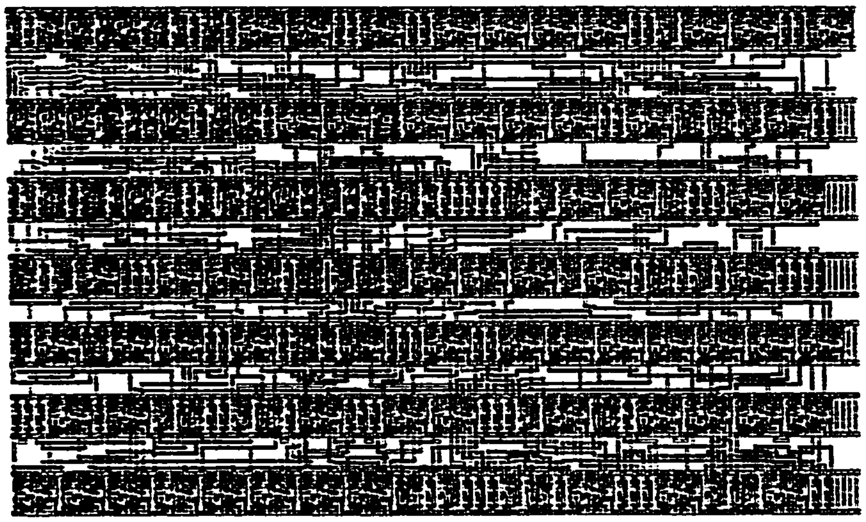

Figure 4.3: VLSI layout of an 8-bit adder (*add*) unit.

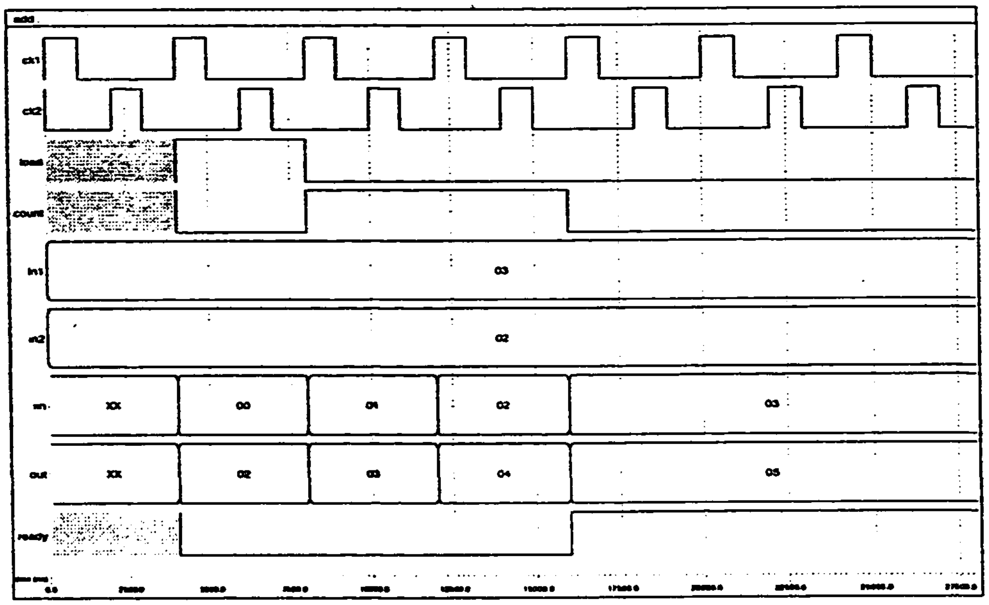

Figure 4.4: Simulation result of an 8-bit adder (*add*) layout.

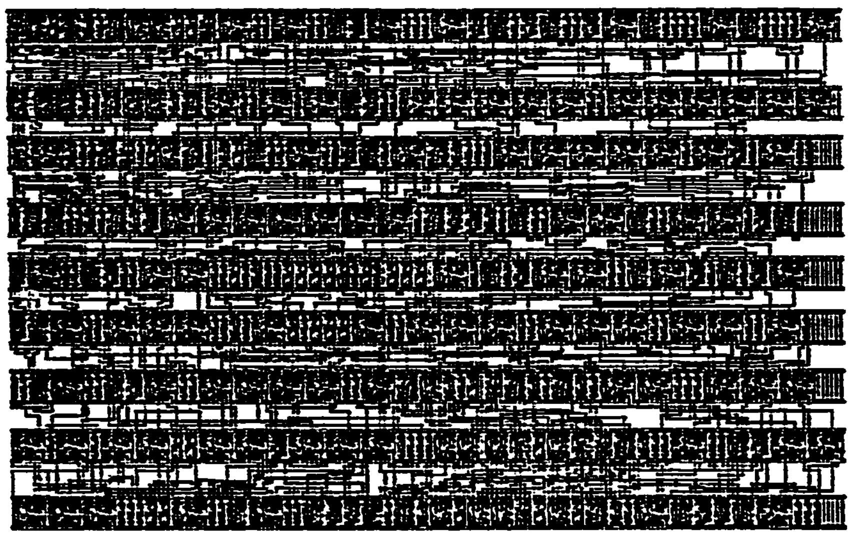

Figure 4.5: VLSI layout of an 8-bit multiplier (*pro*) unit.

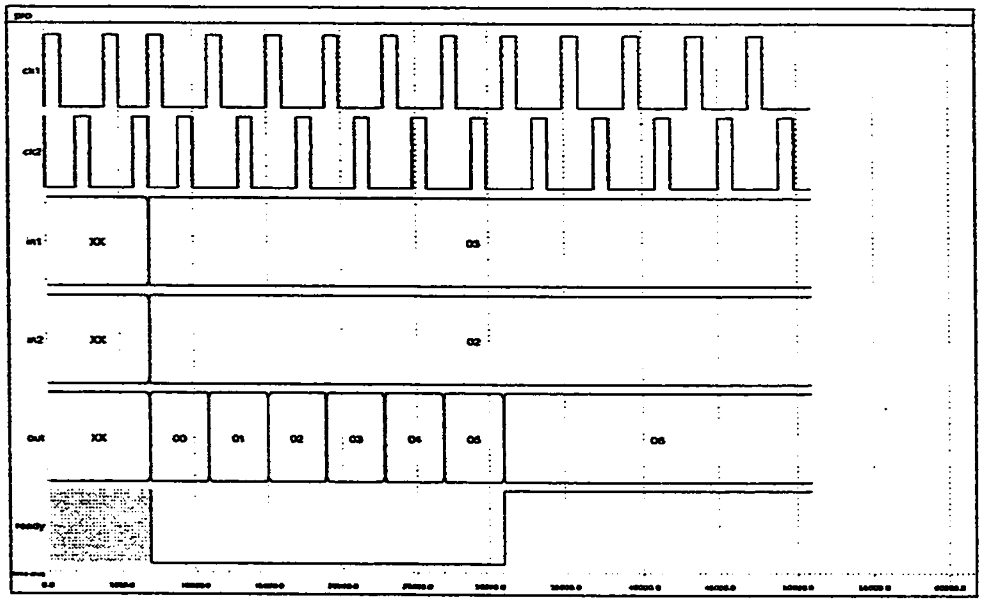

Figure 4.6: Simulation result of an 8-bit multiplier (*pro*) layout.

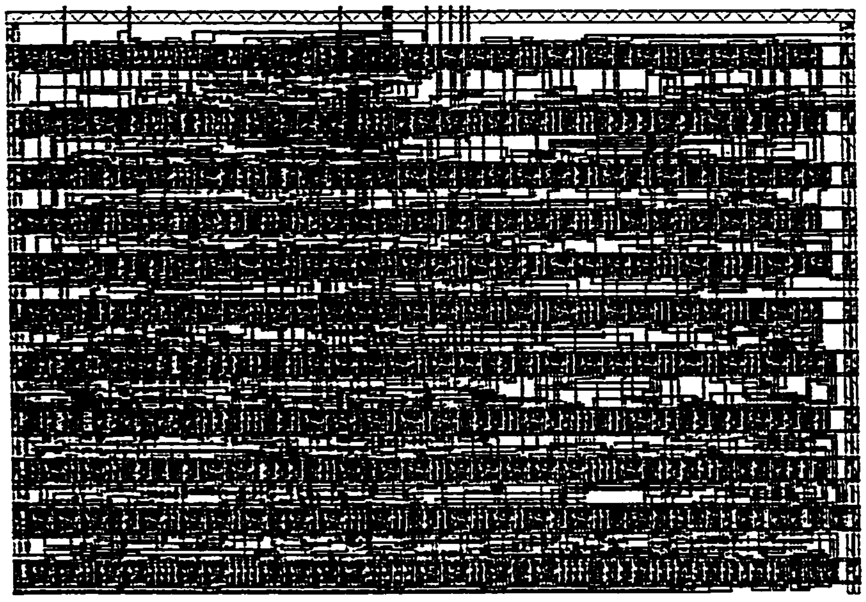

Figure 4.7: VLSI layout of an 8-bit *inner-product* unit.

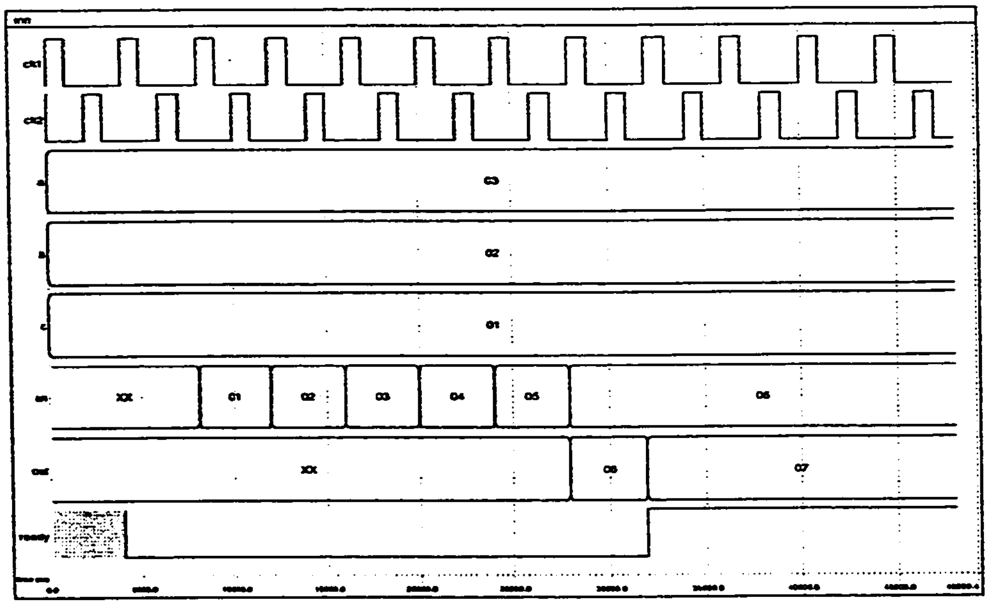

Figure 4.8: Simulation result of an 8-bit *inner-product* layout.

4.5 A Complete Example

An example of formal matrix-matrix multiplier is presented as an application of the cell library. Three different types of architectures i.e., simultaneous recursion, recursion with respect to several variables and fixed number of nestings are used [Ell90]. Each architecture accepts two matrices as input, and a third matrix as an output.

4.5.1 Simultaneous Recursion

Figure 4.9 shows the RSL architecture obtained for matrix multiplication using simultaneous rescursion. It is implemented by applying recursion construct on *inner-product* units. The architecture consists of n^2 inner-product cells. The Figure 4.10 shows the hardware model of this type of *matrix-matrix multiplier* and the Figure 4.11 shows the logic level model of the *matrix-matrix multiplier*. It is made using different cells of the logic cell library i.e., inner-product cell, pro cell, add cell and successor cell.

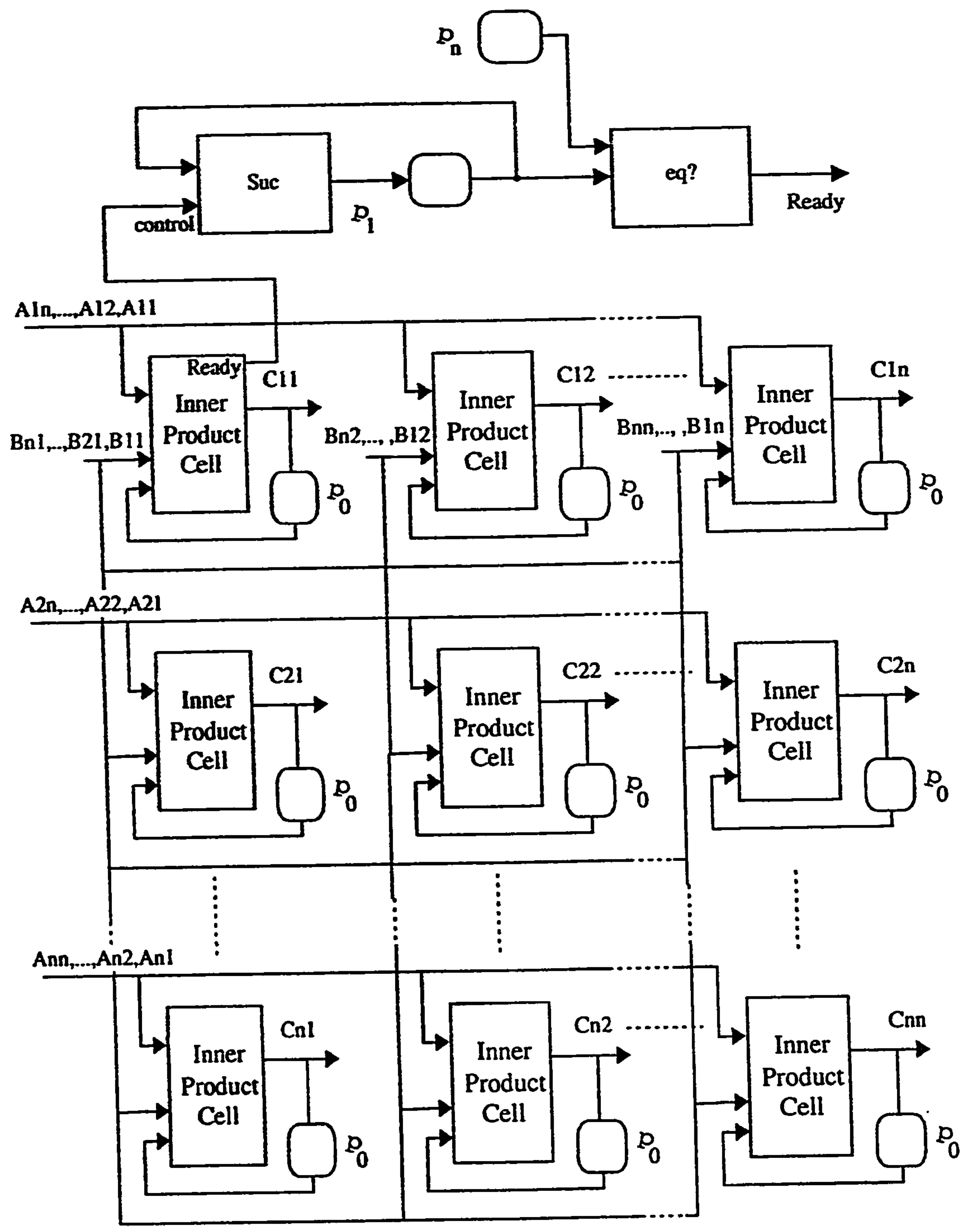

Figure 4.9: Implementation of a *matrix-matrix multiplier* using simultaneous recursion.

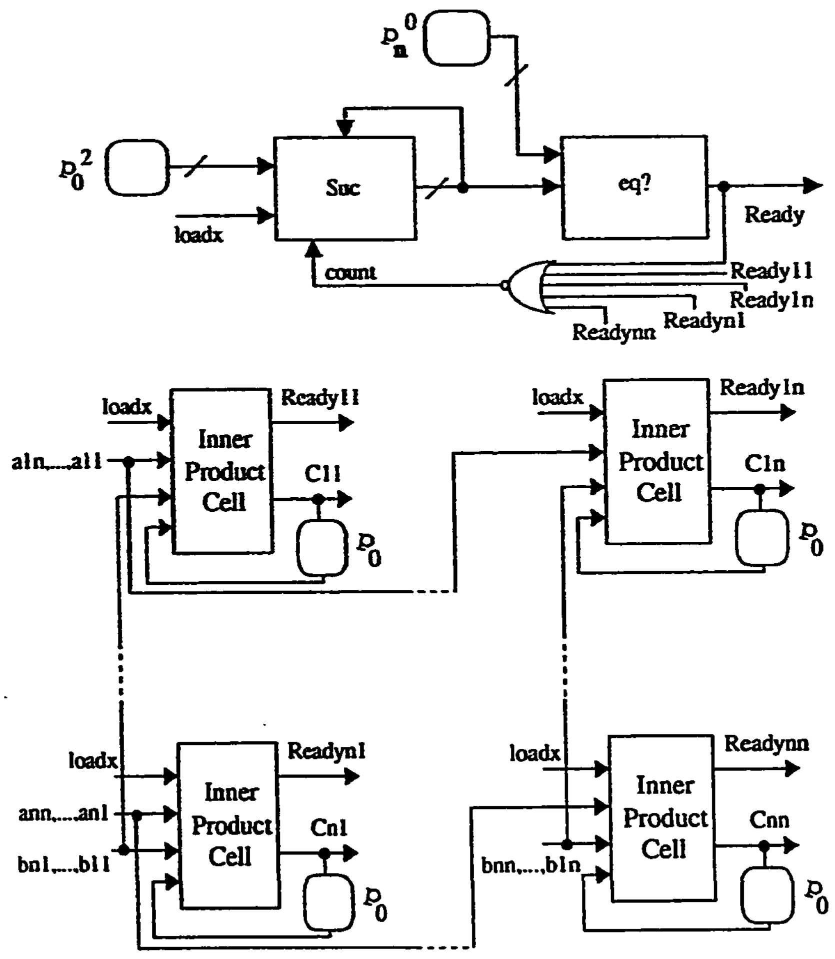

Figure 4.10: Hardware model of a *matrix-matrix multiplier* using simultaneous recursion.

```
; The matrix-matrix multiplier circuit using
; simultaneous recursion

(include "oasis_lib.def")
(include "formal_lib.def")

(macro mul1(o11 o12 o21 o22 i11 i12 i21 i22 a11 a12
a21 a22 b11 b12 b21 b22 phi1 phi2 load b ready )

(local out count con a1 a2 a3 a4 b1 b2 b3 b4 ready1
r1b r11 rb11 r12 rb12 r21 rb21 r22 rb22)
;generate "dff"
        (dff con phi2 out.1 )
;generate "adl8"
        (adl out in1 phi1 phi2 load count b ready1)
;generate "inner-product cells"
(inpro o11 i11 in1 in5 phi1 phi2 load con m1 m2 r11 rb11)
(inpro o12 i12 in1 in5 phi1 phi2 load con m1 m3 r12 rb12)
(inpro o21 i21 in1 in5 phi1 phi2 load con m4 m2 r21 rb21)
(inpro o22 i22 in1 in5 phi1 phi2 load con m4 m3 r22 rb22)

        (o4 rb11 rb12 rb21 rb22 counta )
        (oi2 ready1 counta count )
        (o4 rb11 rb12 rb21 rb22 readya )
        (oi2 ready1b readya ready )

 )
 ;generate globle node names
 (node o11 o12 o21 o22 i11 i12 i21 i22 a11 a12
 a21 a22 b11 b12 b21 b22 phi1 phi2 load b ready)
 (mul o11 o12 o21 o22 i11 i12 i21 i22 a11 a12
 a21 a22 b11 b12 b21 b22 phi1 phi2 load b ready)
```

Figure 4.11: Logic level model of a *matrix-matrix multiplier* using simultaneous recursion.

The Figure 4.12 shows the simulation result of a matrix-matrix multiplier. It can also be observed that when the result is obtained *Ready* goes high. In this simulation following two matrices A and B are given as input to the multiplier and it gives a matrix *out* as an output. The Figure 4.13 shows the VLSI layout of a matrix-matrix multiplier using simultaneous recursion.

$$A \times B = out$$

$$\begin{bmatrix} 1 & 2 \\ 1 & 2 \end{bmatrix} \times \begin{bmatrix} 2 & 1 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 6 & 3 \\ 6 & 3 \end{bmatrix}$$

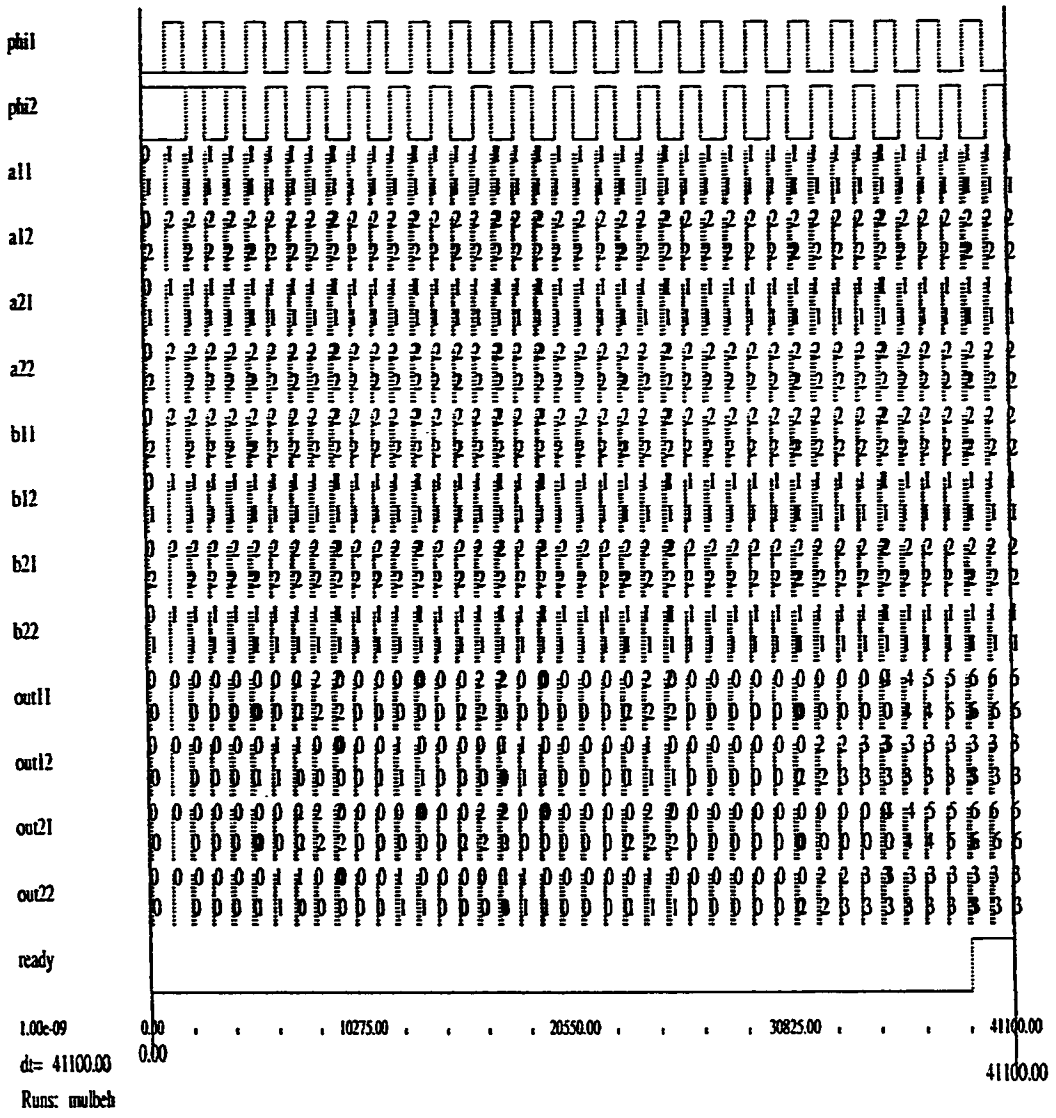

Figure 4.12: Simulation result of a *matrix-matrix multiplier* using simultaneous recursion.

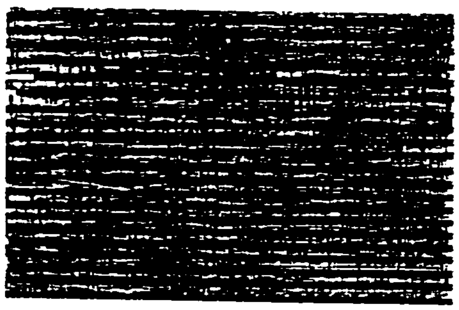

Figure 4.13: VLSI layout of an 8-bit *matrix-matrix multiplier* using simultaneous recursion.

4.5.2 Recursion with Respect to Several Variables

It is implemented by applying recursion construct on *multiplication* and *adder* units. The architecture consists of n multiplication units and 1 adder unit. The Figure 4.14 shows the hardware model of *matrix-matrix multiplier* using recursion with respect to several variables and the Figure 4.15 shows the logic level model of this type of *matrix-matrix multiplier.* The Figure 4.16 shows the simulation result of a matrix-matrix multiplier recursion with respect to several variables. This type of architecture gives output in a serial manner. The output *read* goes high whenever the output is a valid result and the output *ready* goes high after the multiplier give all the results. The Figure 4.13 shows its VLSI layout.

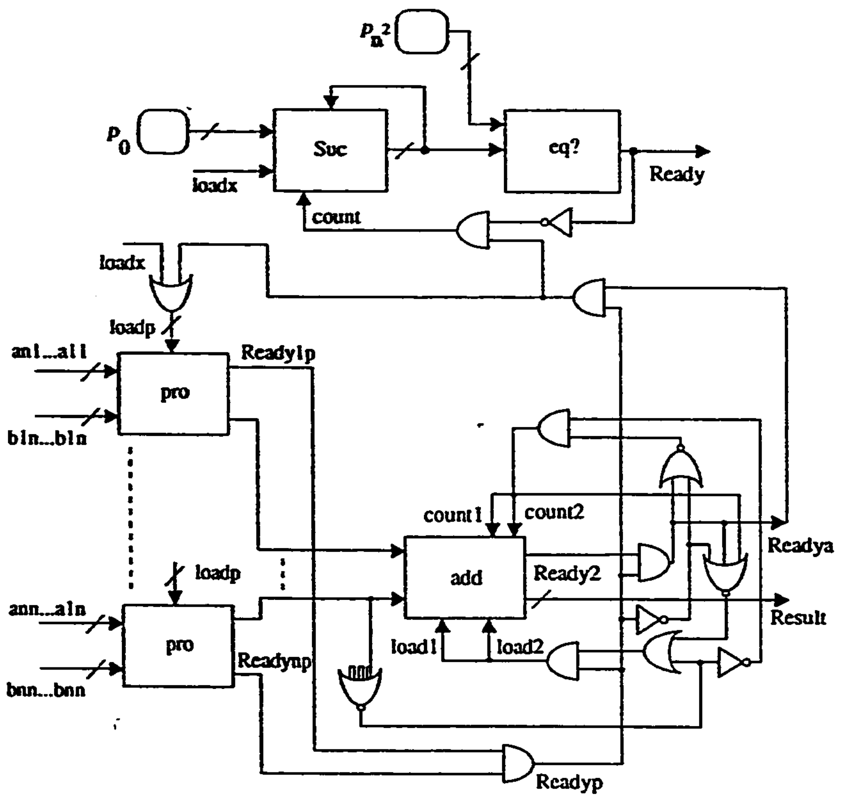

Figure 4.14: Hardware model of *matrix-matrix multiplier* using recursion with respect to several variables.

```
; The matrix-matrix multiplier circuit using
; recursion with respect to several variables

(include "oasis_lib.def")
(include "formal_lib.def")

(macro mul2(iin3 iin2 iin22 iin0 in0 in1 in5 phi1
phi2 a11 a12 a21 a22 b11 b12 b21 b22 load ready)
(local rb rb1 rb2 rb3 c0 c01 c02 c0b load1 load2
count r1b in4 readyab ready1pb ready2pb readyp )
;generate "dff"
   (dff q phi1b counti )
;generate "adl"
   (adl8 out0 in1 phi1 phi2 load count0 rdy)
   (adl8 out in1 phi1 phi2 load1 count1 readya)
;generate counter
   (count8 iin3 in4 phi1 phi2 load2 count2)
;generate "pro"
(pro iin2 in1 phi1 phi2 load ready1p ready1pb)
(pro iin22 in1 phi1 phi2 load ready2p ready2pb)
          (a2 ready1p ready2p readyp )
          (ai2 readya readyp readyb )
          (oi2 ready r1b counti )
          (oi3 r1b ready count1 rb )
          (o2 rb c0 rb2 )
          (ai2 readyp rb2 rb3 )
          (i1 rb3 load2 )
          (o2 load2 lo load1 )
          (a2 c0b q count1 )
          (connect count1 count2 )
          (a2 readya readyp lo )
          (o2 lode loadx loadp )
          (a3 phi1b rdyb lo count0 )
    (repeat i 1 8
         (connect iin2.i in4.i )
         (connect iin22.i b2.i )
    )
)
(node iin3 iin2 iin22 iin0 in0 in1 in5 phi1
phi2 a11 a12 a21 a22 b11 b12 b21 b22 load ready)
(mul2 iin3 iin2 iin22 iin0 in0 in1 in5 phi1
phi2 a11 a12 a21 a22 b11 b12 b21 b22 load ready)
```

Figure 4.15: Logic level model of *matrix-matrix multiplier* using recursion with respect to several variables.

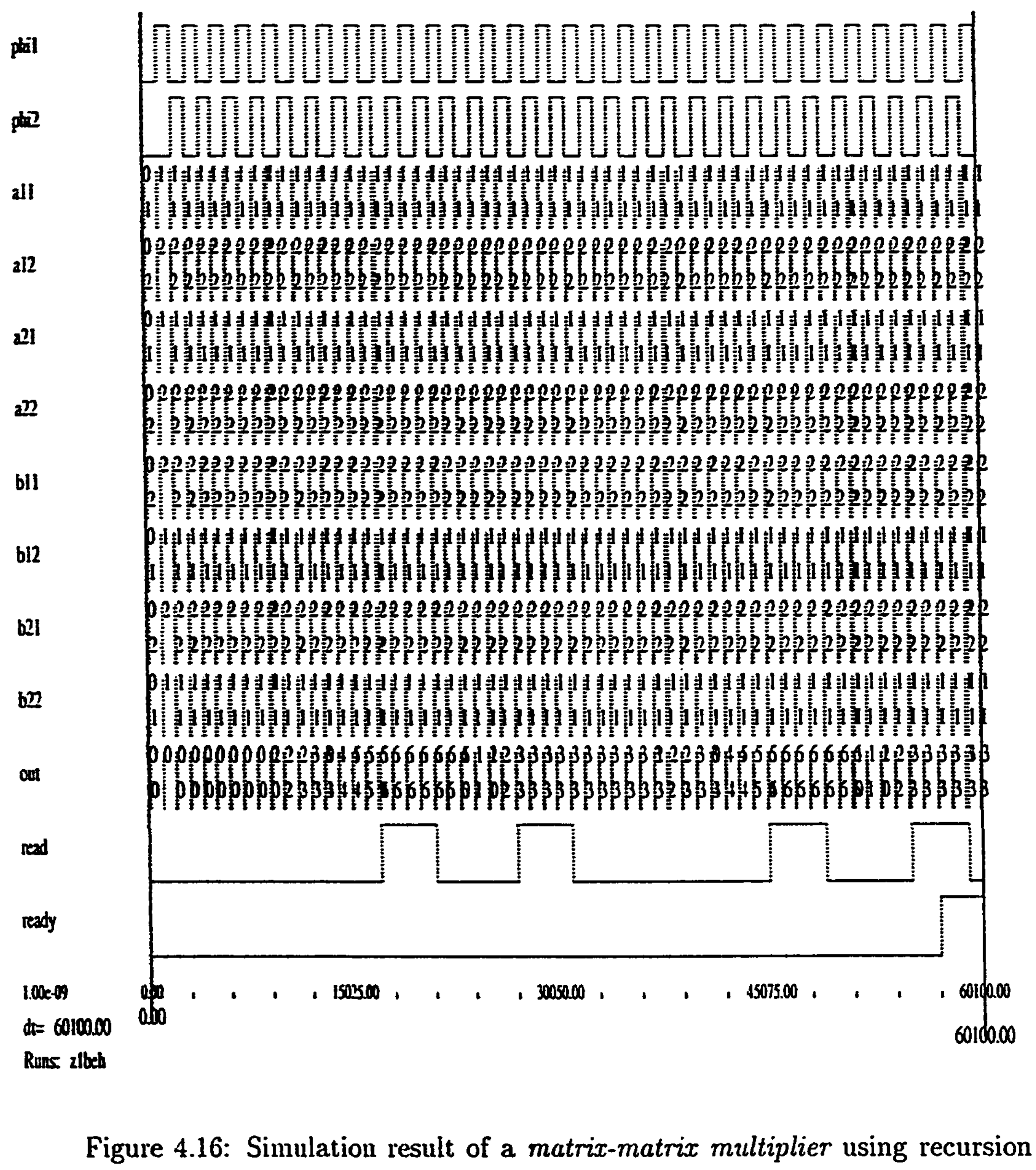

Figure 4.16: Simulation result of a *matrix-matrix multiplier* using recursion with respect to several variables.

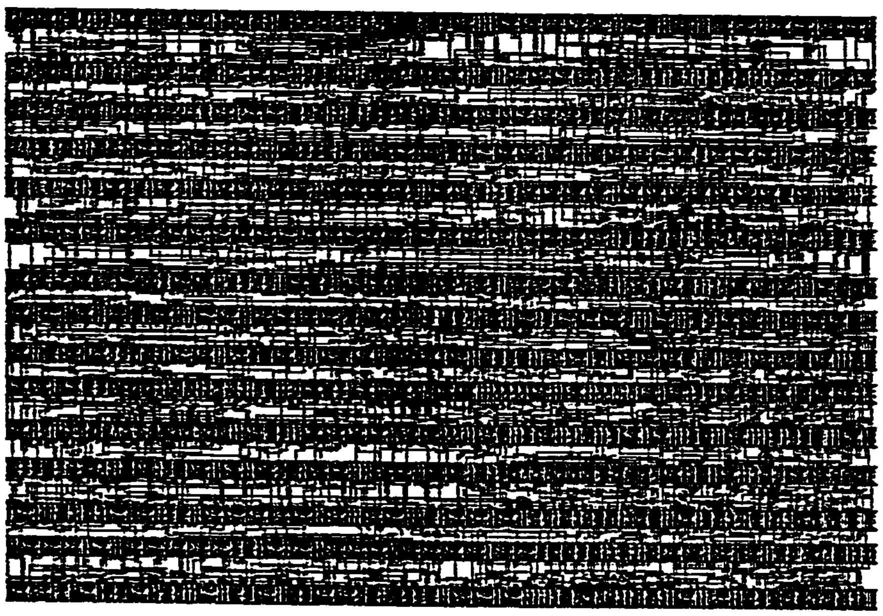

Figure 4.17: VLSI layout of an 8-bit *matrix-matrix multiplier* using recursion with respect to several variables.

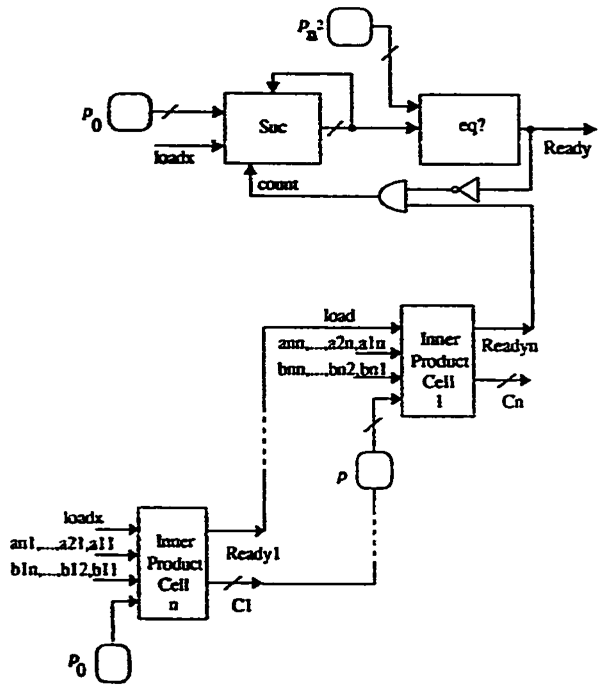

Figure 4.18: Hardware model of *matrix-matrix multiplier* using fixed nesting recursion.

4.5.3 Fixed Nesting Recursion

The architecture consists of n inner-product units. The Figure 4.18 shows the hardware model of *matrix-matrix multiplier* using recursion with respect to several variables and the Figure 4.19 shows the logic level model of this type of *matrix-matrix multiplier*.

The Figure 4.16 shows the simulation result of a matrix-matrix multiplier with fixed nesting recursion. The Figure 4.13 shows the VLSI layout of a matrix-matrix multiplier using fixed nesting recursion.

```
; It is the Matrix-Matrix Multiplier using
; fixed number of nesting

(include "oasis_lib.def")
(include "formal_lib.def")

(macro mul2(iin32 in1 phi1 phi2 load b41 b31
bi21 a11 a12 a21 a22 b11 b12 b21 b22 ready1)
(local loada1 loada2 ready1d ready2d loadi
loadiu readyp1 readyi iin0 count0 )

;generate "adl"
  (adl8 iin0 in1 phi1 phi2 load count0 rdy)

;generate inner-product
(in2 iin32 iin22 in1 in1 phi1 phi2 loadi
readyp2 loada2 ready2 ready2b)
(in iin31 iin21 in1 in1 phi1 phi2 loadi
readyp1 loada1 ready1 ready1b )

(connect readyp1 loada1 )
(a2 ready1 readyp2 loada2 )
(a2 ready1d ready2d readyi )
(o2 readyi loadx loadiu )
(a3 phi1b rdyb ready2 count0 )
   (repeat i 1 8
         (buftri iin31.i loada2 bi22.i )
    )
)

(node iin32 in1 phi1 phi2 load b41 b31
bi21 a11 a12 a21 a22 b11 b12 b21 b22 ready1)
(mul2 iin32 in1 phi1 phi2 load b41 b31
bi21 a11 a12 a21 a22 b11 b12 b21 b22 ready1)
```

Figure 4.19: Logic level model of *matrix-matrix multiplier* using fixed nesting recursion.

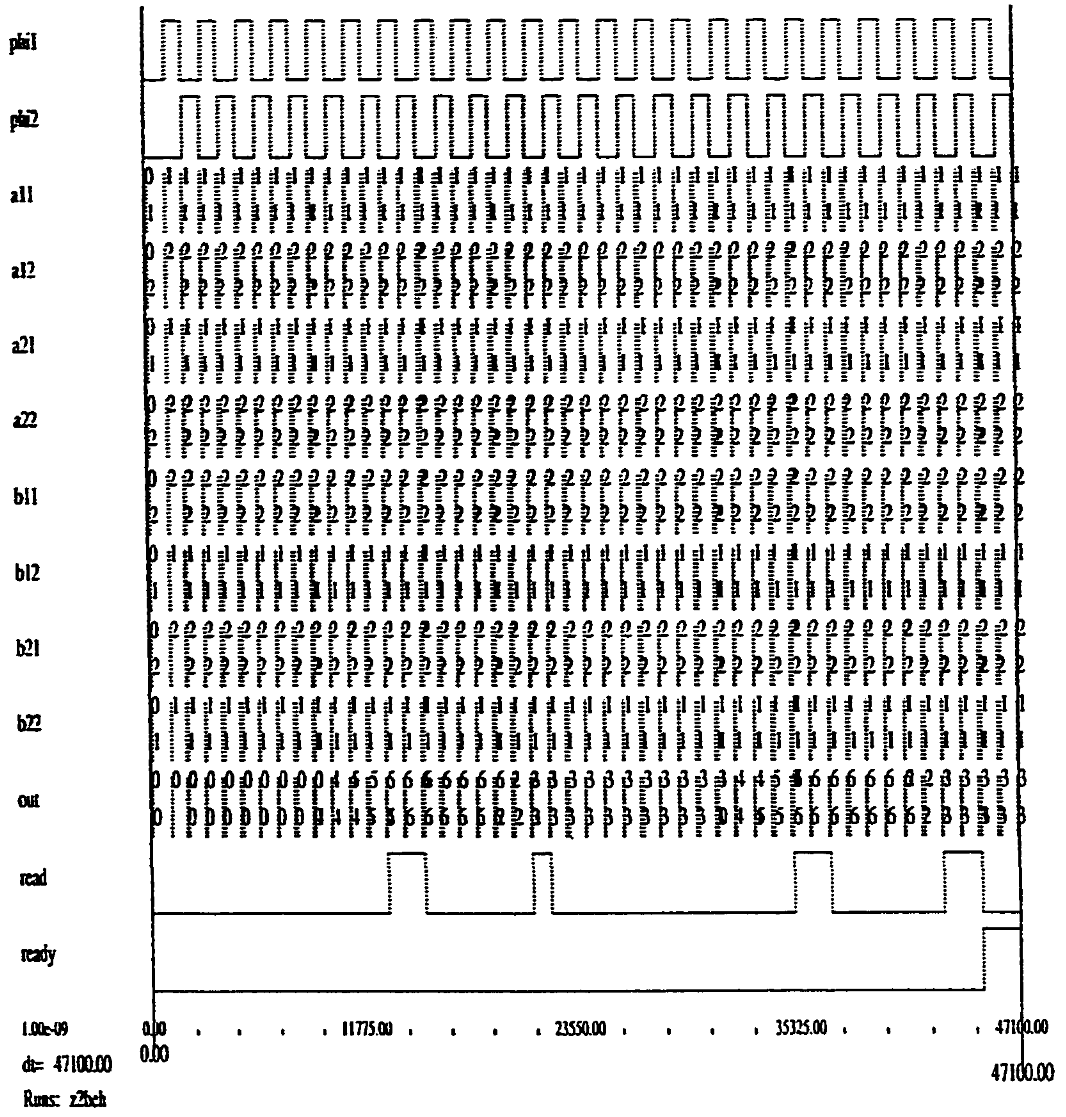

Figure 4.20: Simulation result of a *matrix-matrix multiplier* using fixed nesting recursion.

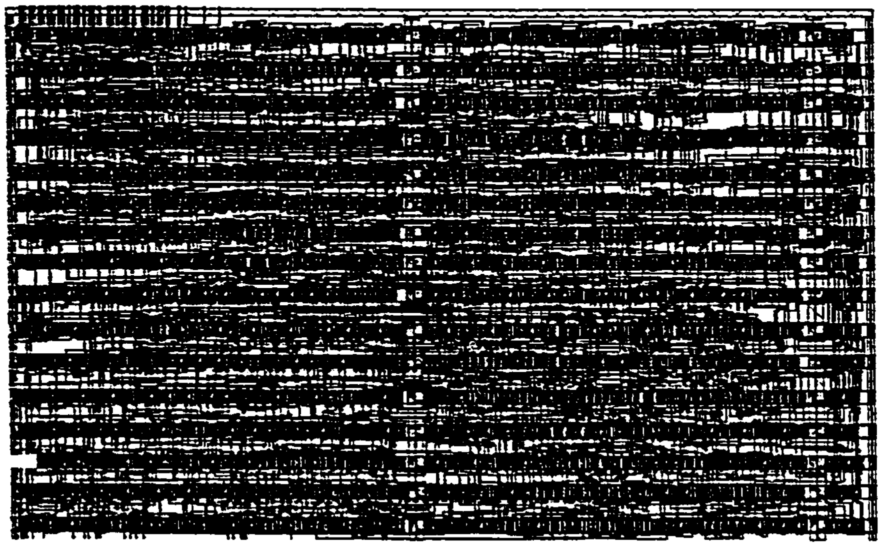

Figure 4.21: VLSI layout of an 8-bit *matrix-matrix multiplier* using fixed nesting recursion.

The architecture of the *matrix-matrix multiplier* using simultaneous recursion has the largest architecture i.e., n^2 *inner-product* units and obviously the largest layout area. But it is the fastest one, gives result parallely. While the other two types have smaller layout area but these are slower, gives result serially. Next chapter shows the comparison between the three types of *matrix-matrix multipliers.*

Chapter 5

Conclusions

A complete design of a logic level cell library for the support of back-end design of a formal high level synthesis system has been presented. The cell library contains the logic level models of basic primitives of RSL, viz., *zero, projection, successor, composition, recursion* and *μ-recursion*, that can be applied recursively to obtain any computable function. All the cells are exhaustively simulated and made modular so that the design is capable of extending to any desired word length. These modules are also used to made the larger functions. A formal matrix-matrix multiplier has been designed using the support of the cell library.

Table 5.1 shows the number of devices and layout area of these units for an 8-bit data bus. Table 5.2 shows the number of clock cycles required by these units for an 8-bit data bus. The time required by the main multiplier is important which is, as mentioned in the table, $max(C_{11}, \cdots, C_{nn})$, where C_{11} is the time required by the inner-product C_{11} and so on.

It can be observed that the area and time is high as compared to the that made by non-formal methods. It is the price paid for the functionally correct hardware made by formal techniques.

Unit	No. of Devices	Area (λ^2)	Other Units Used
counter	336	288320	successor
add	974	739480	incrementer
pro	1674	1284376	add, incrementer
inner-product	2720	2244528	add,pro,incrementer
multiplier1	12060	12325000	inner-product
multiplier2	5864	4218240	add, pro
multiplier3	7244	5279400	inner-product

Table 5.1: Number of devices and layout area of 8-bit units.

Unit	No. of Arguments	Arguments	Time (clock cycle)
counter	1	n	n
add	2	m, n	max(m,n)
pro	2	m, n	m× n
inner-product	3	a, b, c	(a×b) + max(a×b, c)
multiplier1	2($n \times n$)matrices	$A^{n\times n}$, $B^{n\times n}$	$max(C_{11}, \cdots, C_{nn})$ where $C_{11} = (a_{11} \times b_{11} + a_{12} \times b_{21}) + max(a_{11} \times b_{11}, a_{12} \times b_{21})$
multiplier2	2($n \times n$)matrices	$A^{n\times n}$, $B^{n\times n}$	$(C_{11} + \cdots + C_{nn})$ where $C_{11} = 2max(a_{11} \times b_{11}, a_{12} \times b_{21})$
multiplier3	2($n \times n$)matrices	$A^{n\times n}$, $B^{n\times n}$	$(C_{11} + \cdots + C_{nn})$ where $C_{11} = (a_{11} \times b_{11}) + (a_{12} \times b_{21}) + max(a_{11} \times b_{11}, a_{12} \times b_{21})$

Table 5.2: Number of clock cycles required by the 8-bit units to get their final results.

Appendix A

ASL Syntax and Semantics

ASL Syntax

This appendix is taken from [Ell90]. The syntax of ASL is described using the Bakus-Naur Form (BNF). The following meta symbols are used:

:: =	means 'is defined as'.
|	means 'or' .
< >	non terminal names.
[ ]	optional items.
{ }	repetition.

Using the previous meta-symbols, the syntax of ASL is defined as follows:

< ASL specification > ::= < extended mu-recursive specification >
< extended mu-recursive specification > ::= < recursive function name >
| < unbounded minimization > | < basic function name >
< recursive function name > ::= < recursive function > (< argument > {, < argument > })
< recursive function > ::= < initial function > | < operation > (< recursive function > {, < operation > (< recursive function >) })
< operation > ::= < composition > | < primitive recursion >
< composition > ::= < variable name > = < variable name > (< argument . {, < argument > })
< primitive recursion > ::= < case m = 0 > < case m+1 >
< case m = 0 > ::= < variable name > (< argument >,0) = < variable name > (< arguments >)
< arguments > ::= < blank > | <argument name > {, < argument name >}
< case m+1 > ::= < variable name > (< arguments > ,m+1) =
< variable name > (< arguments > ,m, < variable name > (<arguments >))

< initial function > ::= < zero function > | < successor function > | < projection function >
< zero function > ::= ξ ( )
< successor function > ::= λ (< argument >)
< projection function ::= $\eta^{<integer>}_{<argument>}$(< argument > {,< argument > })
< unbounded minimization > ::= unbound < mu-recursive specification >
< argument > ::= < mu recursive specification > | < variable name > | < number > | < boolean >
< basic function name > ::= $\boxed{\text{variable name}}$
< variable name > ::= < upper letter > { < letter > | < digit > }
< argument name > ::= < lower letter > { < letter > | < digit > }
< Number > ::= < integer > | < real >]
< real > ::= < integer > .[< unsigned integer >]
< integer > ::= [< sign >] < unsigned integer >
< unsigned integer > ::= < digit > {< digit >}
< boolean > ::= *true* | *false*
< sign > ::= −|+
< letter > ::= < upper letter > | < lower letter >
< upper letter > ::= $A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z$
< lower letter > ::= $a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z$
< digit > ::= 0|1|2|3|4|5|6|7|8|9
< blank > ::=

ASL Semantics

Let us define a meaning function M that maps sentences of ASL to values and data. For example if X is an integer with a value 4, this is represented using the meaning function as follows:

$$M < X > \equiv_{def} 4$$

1. Zero function:

$$M < \xi > \equiv_{def} 0$$

2. Successor function:

$$M < \lambda(n) > \equiv_{def} M < M < n > +1 >$$

3. Projection function:

$$M < \eta_i^k(arg_1, arg_2, \ldots, arg_k) > \equiv_{def} M < arg_i >$$

4. Composition:

$$M < y(x_1, x_2, \ldots, x_m) > \equiv_{def}$$
$$M < y(M < X_1 >, M < x_2 >, \ldots, M < x_m >)$$

5. Recursion:

$$M < z(arg_1, \ldots, arg_n, 0) > \equiv_{def}$$
$$M < x(M < arg_1 >, \ldots, M < arg_n >) >$$
$$M < z(arg_1, \ldots, arg_n, m+1) > \equiv_{def}$$
$$M < y(M < arg_1 >, \ldots, M < arg_n >, M < m >,$$
$$M < z(M < arg_1 >, \ldots, M < arg_n >, M < m >) >) >$$

6. Unbounded Minimization:

$$M < unbound(arg_1, arg_2, \ldots, arg_n) > \equiv_{def}$$
$$M < min\ m\ s.t\ M < x(M < arg_1 >, M < arg_2 >, \ldots, M < arg_n >, m) = 0 >>$$

7. Basic Functions:

$$M < \boxed{f} > \equiv_{def} M < f >$$

Appendix B

RSL Syntax and Semantics

RSL Syntax

< RSL specification > ::= { < zero specification > | < projection specification > | < successor function > | < composition specification > | < recursion specification > | < mu recursion specification >}

< zero specification > ::= Result = zero
< control statement >

< successor specification > ::= Result = suc (< argument >)
< control statement >

< projection specification > ::= Result = mux (< arguments > # < integer >)
< control statement >

< composition specification > ::= Result = Comp (< arguments > # < function name >)
< complex control statement >

< recursion specification > ::= Result = eq ? (< variable name >, < variable name >)
< parallel initialize >< control statement > I = ρ_1^{n+1} suc(I)
Ready = eq ?(I, m)
Result = comp (< arguments > , I $\rho^{n+2}_{<arguments>}$
Result # <function name >)

< mu recursion specification > ::= < parallel initialize >
Result = ρ_0 suc (Result)
Ready = eq ?(0, < function name > (< arguments >,
Result))

< Parallel initialize > ::= Initp (< register>, < argument >
{; < register >, < argument >})

< control statement > ::= $zero^{Ready} = zero_{Control}$
$|mux^{Ready} = mux_{Control}$
$|suc^{Ready} = suc_{Control}$

< complex control statement > ::= $Comp^{Ready} = And(\{$< function name > $^{Ready}\})$

< arguments > ::= < blank > | < argument name > {, < argument name >}

< function name > ::= < variable name > | < basic function name >

< basic function name > ::= $\boxed{\text{variable name}}$

< variable name > ::= < upper letter > {< letter > | < digit >}

< argument name > ::= < lower letter > {< letter > | < digit >}

< argument > ::= < number > | < boolean > | < function name >

< register > ::= < integer > | < variable name >

< Number > ::= < integer > | < real >

< real > ::= < integer > .[< unsigned integer >]

< integer > ::= [< sign >] < unsigned integer >

< unsigned integer > ::= < digit > {< digit >}

< boolean > ::= $true|false$

< sign > ::= $-|+$

< letter > ::= < upper letter > | < lower letter >

< upper letter > ::= $A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z$
< lower letter > ::= $a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z$
< digit > ::= 0|1|2|3|4|5|6|7|8|9
< blank > ::=

RSL Semantics

Let us use a function called *Connect* to represent the fact that two units are connected. If the output of unit A is connected to unit B, we represent this as:

$$A\ Connect\ B$$

1. Zero function:

$$M < Result = 0 > \equiv_{def} M < M < Result >= 0 >$$

$$M < zero^{Ready} = zero_{Control} > \equiv_{def} M < M < zero^{Control} > M < connect > \\ M < zero_{Ready} >>$$

2. Successor function:

$$M < Result = suc(argument) > \equiv_{def} M < M < argument > +1 >$$

$$M < suc^{Ready} = suc_{Control} > \equiv_{def} M < M < suc^{Control} > M < connect > \\ M < suc_{Ready} >>$$

3. Projection function:

$$M < Result = mux(arg_1, \ldots, arg_n \# < integer >) > \equiv_{def} M < Result =< arg_{integer} >$$

$$M < mux^{Ready} = mux_{Control} \equiv_{def} M < M < mux^{Control} > M < connect > \\ M < mux_{Ready} >>$$

4. Composition:

$$M < Result = Comp(arg_1, \ldots, arg_n \# function) > \equiv_{def}$$

$$M < function(M < arg_1 >, \ldots, < arg_n >)$$

5. Recursion:

$$M < Result = eq?(V_1, V_2) > \equiv_{def} M < eq?(M < V_1 >, M < V_2 >) >$$

$$M < Initp(R1, V1; R2, V2; \ldots; R_n, V_n) > \equiv_{def} M < M < R1 = V1 >, \\ < R2 = V2 >, \ldots, < R_n = V_n >>$$

$$M < I = \rho_1^{n+1} suc(I) > \equiv_{def} M < M < R_{n+1} >= 1 > M < Connect > \\ < M < M < I > +I >>$$

$$M < Result = comp(arg_1, \ldots, arg_n, I, \rho^{n+2} \# function) > \equiv_{def}$$

$$M < function(M < arg_1 >, \ldots, M < arg_n >, M < I >, M < \rho^{n+2} >>$$

6. **Unbounded Minimization:**

$$M < Initp(R1, V1; R2, V2; \ldots; R_n, V_n) > \equiv_{def} M < M < R1 = V1 >, \\ < R2 = V2 >, \ldots, < R_n = V_n >>$$

$$M < Result = \rho_0 suc(Result) > \equiv_{def}$$

$$M < M < R = 0 > M < Connect > M < M < Result > +1 >>$$

$$M < Ready = eq?(0, function(arg_1, \ldots, arg_n, Result)) > \equiv_{def}$$

$$M < eq?(function(M < arg_1 >, \ldots, M < arg_n >, M < Result >), 0) >$$

7. Basic Functions:

$$M < \boxed{\mathrm{f}} > \equiv_{def} M < f >$$

Appendix C

Tools used

Netlist

Netlist is a simple description language for VLSI circuits. It is a logic-level macro-based language for describing networks of sized transistors. Names in *netlist* refer to nodes, which presumably get interconnected by the user through transistors. In addition to transistor macros *netlist* provides macros that allow the user to set node capacitance, specific node delays (in tenths of nanoseconds), and transistor threshold voltages. It also accepts user defined macros.
A node name reference has two forms: One form is

$$n$$

Where n is the name of the network node. When transistor sizes are required they are taken from the appropriate defaults.
The second form is

$$(n\ width\ length)$$

Where n is the name of the network node and, *width* and *length* specify a transistor size. This is used in *netlist* constructs where mention of a name causes creation of a transistor. The netlist creates an output file with the extension ".sim".

Presim

Presim is a translation tool. It converts the ".sim" file (produced by *netlist*) into a binary file suitable for RNL (switch level simulator). Results of *presim* depend on a number of parameter values. The "lambda" parameter is also specified in this file. Resister values not explicitly provided in the configuration file are estimated by linear interpolation. The resister values are sorted first by width, then by length (not by ratio).
Each line of a *config* file has the format

$$parameter\ values\ comments$$

Lines beginning with ";" are treated as all comment.

RNL

RNL is a timing logic simulator for digital MOS circuits. It is an event driven simulator that uses a simple RC (Resistance Capacitance) model of the circuit to estimate the effects of charge shearing. The user interface is a simple LISP interface. This allows both interactive simulation and the programming of complex simulations.

To use RNL, the ".sim" file for the circuit to be simulated is needed. This ".sim" file is then converted to a binary file using *presim*. The RNL uses that binary file for simulation. It is designed to handle the ratioed logic, bidirectionality, and charge sharing/storage. They can be used to determine the functionality and approximate timing behavior of circuits commonly found in digital designs. Basic to the operation of the simulators is the notion of an event. An event specifies

1. a node in the network,
2. a new logic state, and
3. a time at which the node's value is changed to the new logic state.

RNL maintains a list of events, sorted by time, that tells what processing remains to be done. Whenever the input is changed, an event is added to the list; when the list is empty the network has "settled" and RNL waits for further input.
When started with an initial list. RNL sequentially processes the next event on the list, stopping

1. when the list is empty,
2. when a node is need to be traced, or
3. when the specified amount of simulated time has elapsed.

Since nodes are only added to the event list when their values change, portions of the circuit unaffected by the current set of changes to the inputs are not re-evaluated. The algorithm is event-driven, sometimes called selective trace.

OASIS

OASIS is an abbreviation of Open Architecture Silicon Implementation Software.It is a cell-based system for IC design. The tools integrated into the OASIS system have been developed to automatically translate high-level descriptions of integrated circuits into testable physical layouts, using predesigned standard cells.

OASIS is a cell-based design system with five major subcomponent subsystems:

- compilation and logic synthesis,
- simulation and verification,

- automatic test pattern generation,
- automatic layout generation,
- cell binary maintenance.

One of the premises of the OASIS System is the modularity of the software. New, improved algorithms can be easily substituted in place of the old one's. The entire system is controlled with a single data flow supervisor program to assure data consistency is maintained at all stages of the design. The data flow supervisor is template-driven, the templates used in OASIS can be easily expanded to support additional software tool, thus providing the desired openness of the system.

Mextra

Mextra is a Manhattan circuit extractor for VLSI simulation. It reads the file basename.*cif* (generated by *magic*) and creates the circuit description. From this circuit description various electrical checks of the circuit can be done. Mextra creates five new files, basename.*log*, basename.*al*, basename. *sim*, basename.*tbs* and basename.*nodes*. The *.log* file contains general information about the extraction such as number of transistors and the number of nodes and also the messages about the errors, if any. The *.al* file is a list of aliases. The *.tbs* file is a list of transistors in the .sim file that contains the substrate/well node name for each transistor. The *.nodes* file is a list of node names. The *.sim* file is the circuit description for use with simulation programs and electrical rule checkers.

Magic

Magic is an interactive system for creating and modifying VLSI circuit layouts [MAS$^+$90]. Magic is different from other layout editors in many aspects. The most important difference is that Magic is more than just a color painting tool, it understands quit a bit about the nature of circuits and uses this information to provide additional operations. For example, Magic has built-in knowledge of layout rules. It also knows about connectivity and transistors, and contains a built-in hierarchical circuit extractor.

Magic is based on the *Mead-Conway* style of design. This means that it uses simplified design rules and circuit structures. It permits only *Manhattan* designs i.e., those whose edges are vertical or horizontal.

Bibliography

[BCDM86] M. Browne, E. M. Clarke, D. Dill, and B. Mishra. Automatic verification of sequential circuits using temporal logic. *IEEE Transactions on Computer*, pages 1035–1044, December 1986.

[Cam85] R. Camposano. Synthesis techniques for digital systems design. *Twenty-second Design Automation Conference*, pages 475–481, June 1985.

[Cam90] R. Camposano. From behavior to structure: High-level synthesis. *IEEE Design and Test of Computers*, 7(5):8–19, October 1990.

[CLM89] E. M. Clarke, D. E. Long, and K. L. McMillan. *A Language for Compositional Specification and Verification of Finite State Hardware Controllers*, pages 281–295. North-Holland, Amsterdam, The Netherlands, 1989.

[CP88] Paolo Camurati and Paolo Prinetto. Formal verification of hardware correctness. *IEEE Computer*, 16(12):8–19, February 1988.

[CR89] Raul Camposano and Wolfgang Rosentiel. Synthesizing circuits from behavioral descriptions. *IEEE Transactions on Computer Aided Design*, 8(2):171–180, February 1989.

[CS90] Shiu-Kai Chin and Edward P. Stabler. Synthesis arithmetic hardware using hardware metafunctions. *IEEE Transactions on Computer Aided Design*, 9(8):793–803, August 1990.

[Dav90] Bruce S. Davie. *Formal Specification and Verification in VLSI Design.* Prentice-Hall, 1990.

[DC86] D. L. Dill and E. M. Clarke. Automatic verification of asynchronous circuits using temporal logic. *IEE Proceedings*, 133(5):276–282, September 1986.

[DPST81] S. W. Director, A. C. Parker, D. P. Siewiorek, and D. E. Thomas. A design methodology and computer aids for digital VLSI systems. *IEEE Transactions of Circuits System*, CAS-28:634–645, July 1981.

[EB90] Khalid M. Elleithy and Magdy A. Bayoumi. Synthesizing DSP architectures from behavioral specifications: A formal approach. *Proceedings-IEEE International Symposium on Circuits and Systems*, 2:1131–1134, May 1990.

[El190] Khalid M. Elleithy. *A Formal Framework For High Level Synthesis of Digital Designs.* PhD thesis, The Center for Advanced Studies, University of South-Western Lousiana, 1990.

[Eve87] Hans Eveking. *From HDL Descriptions to Guaranteed Correct Circuit Designs,* chapter Verification, Synthesis, and Correctness-Preserving Transformations-Comparative Approaches to correct Hardware Design, pages 229–239. Elsevier Science Publishers B.V., 1987.

[FK80] M. J. Foster and H. T. Kung. The design of special purpose VLSI chips. *Computer,* 13(1):26–40, January 1980.

[FTMo85] M. Fujita, H. Tanaka, and T. Moto-oka. *Logic Design Assistance with Temporal Logic,* pages 129–138. North-Holland, Amsterdam, The Netherlands, 1985.

[Gal87] A. Galton. *Temporal Logic and Their Applications.* Academic Press, London, England, 1987.

[GK83] D. Gajski and R. Kuhn. Guest editor's introduction: New VLSI tools. *Computer,* 16(12):11–14, December 1983.

[GK88] E. F. Girczyc and J. P. Knight. An ada to standard cell hardware compiler based on graph grammars and scheduling. *Proceedings of the 1984 ICCD, New York,* pages 726–731, October 1988.

[Gor85] M. Gordon. Hol: A machine oriented formulation of higher order logic, tech rep. no. 68. Technical report, Computer Laboratory, University of Cambridge, Cambridge, England, 1985.

[Gor86] Mike Gordon. *Why Higher-Order Logic is a Good Formalism for Specifying and Verifying Hardware*, chapter Hardware Verification Using Higher-Order Logic, pages 153–177. Elsevier Science Publishers B.V., 1986.

[Hai82] B. T. Hailpern. *Verifying Concurrent Processes Using Temporal Logic*, chapter Springer-Verlag, Berlin, West Germany. Lecture Notes in Computer Science, Vol. 129, 1982.

[Hat82] W. S. Hatcher. *The Logical Foundations of Mathematics.* Pergamon Press, Oxford, England, 1982.

[HD85] F. K. Hanna and N. Daeche. Specification and verification using higher-order logic. *Proceedings of the Seventh International Conference on Computer Hardware Design Languages, Tokyo, Japan*, 1985.

[HJ88] Richard I. Hartley and Jeffrey R. Jasica. Behavioral to structural translation in a bit-serial silicon compiler. *IEEE Transactions on Computer Aided Design*, 7(8):877–886, August 1988.

[HMM83] J. Halpern, Z. Manna, and Moszkowski. A hardware semantics based on temporal intervals. *Proceedings of the Tenth International Colloquium on Automata, Languages and Programming, Barcelona, Spain*, 1983.

[HP82] L. J. Hafer and A. C. Parker. Automated synthesis of digital hardware. *IEEE Transactions on Computer*, C-31:93–109, Feb. 1982.

[HRC89] Ramesh Harjani, Rob A. Rutenbar, and L. Richard Carley. Oasys: A framework for analog circuit synthesis. *IEEE Transactions on Computer Aided Design*, 8(12):1247–1266, December 1989.

[KM91] R. P. Kurshan and K. L. McMillan. Analysis of digital circuits through symbolic reduction. *IEEE Transactions on Computer Aided Design*, 10(11):1356–1371, November 1991.

[KT83] T. J. Kowalski and D. E. Thomas. The VLSI design automation assistant: Prototype system. *Twenty Design Automation Conference*, pages 479–483, 1983.

[LBK88] R. Lisanke, F. Brglez, and G. Kedem. Mcmap: A fast technology procedure for multi-level logic synthesis. *Proceedings of the 1988 ICCD*, pages 252–256, 1988.

[LNR88] M. J. Lorenzetti, M. S. Nifog, and J. E. Rose. Channel routing for compaction. *Proceedings of the International Workshop on Placement And Routing*, May 1988.

[MAS+90] Robert N. Mayo, Michael H. Arnold, Walter S. Scott, Don Stark, and Gordon T. Hamachi. Decwrl/livermore magic release. Technical report, DECWRL, Digital Western Research Laboratory, September 1990.

[MC80] Carver Mead and Lynn Conway. *Introduction to VLSI Systems.* Addison-Weslay, 1980.

[Mel88a] T. Melham. *Formal Verification and Implementation of a Microprocessor*, chapter The Mechanical Verification of a Microprocessor Design, pages 129–157. Kluwer Academic Publishers, 1988.

[Mel88b] T. F. Melham. *Abstraction Mechanisms for Hardware Verification*, chapter Mathematical Logic, pages 129–157. Kluwer Academic Publishers, 1988.

[Men64] E. Mendelson. *Introduction to Mathematical Logic.* D. Van Nostrand Company, Inc., Princeton, N.J., 1964.

[MF85] F. Maruyama and M. Fujita. Hardware verfication. *Computer*, 16(12):22–32, February 1985.

[MP81] Z. Manna and A. Pnueli. *Verification of Concurrent Programs: The Temporal Framework*, pages 215–273. Academic Press, New York, 1981.

[MPC88] M. C. McFarland, A. C. Parker, and R. Camposano. Tutorial on high-level synthesis. *Proceedings of the 25th Design Automation Conference*, pages 330–336, June 1988.

[MPC90] M. C. McFarland, A. C. Parker, and R. Camposano. The high-level synthesis of digital systems. *Proceedings of the IEEE*, 78(2):301–318, February 1990.

[Muk86] Amar Mukherjee. *Introduction to nMOS and CMOS VLSI Systems Design*. Prentice-Hall, 1986.

[OAS92] MCNC's Center for Microelectronics. *Open Architecture Silicon Implementation Software*, release 2.0 edition, December 1992.

[Ros85] J. E. Rose. Greedy algorithms for wiring in VLSI. Master's thesis, Department of Computer Sceince, North Carolina State University, 1985.

[SBE88] V. Stavridou, H. Barringer, and D. A. Edwards. Formal specification and verification of hardware: A comparative case study. *The Proceedings of the Twenty-fifth ACM/ IEEE Design Automation Conference*, pages 197–204, June 1988.

[SK87] P. R. Suaris and G. Kedem. A new approach to standard cell layout. *International Conference on Computer Aided Design*, pages 474–477, November 1987.

[Tho86] D. E. Thomas. Automatic data path synthesis in design methodologies. *Advances in CAD for VLSI*, 6, 1986.

[TKK89] T. Tanaka, T. Kobayashi, and O. Karatsu. Harp: Fortran to silicon. *IEEE Transactions on Computer Aided Design*, 8(6):649–660, June 1989.

[Tri87] H. Trickey. Flamel: A high-level hardware compiler. *IEEE Transactions on Computer Aided Design*, CAD-6(2):259–269, March 1987.

[VLS87] *VLSI Design Tools Reference Manual.* NW Laboratory for Integrated Systems, release 3.1 edition, February 1987.

[WC91] Robert A. Walker and Raul Camposano. *A Survey of High-Level Synthesis Systems.* Kluwer Academic Publishers, 1991.

[WE85] Neil Weste and Kamran Eshraghian. *Principles of CMOS VLSI Design.* Addison-Wesley, 1985.

[WHJ86] et. al. W. H. Joyner. Technology adaptation in logic synthesis. *Twenty-third Design Automation Conference*, pages 94–100, 1986.

[WT85] R. A. Walker and D. E. Thomas. A model of design representation and synthesis. *Twenty-second Design Automation Conference*, pages 453–459, June 1985.

[Yoe90] Michael Yoeli. *Formal Verification of Hardware Design.* IEEE Computer Society Press, 1990.

[Zim79] G. Zimmermann. The mimola design system: A computer aided digital processor design method. *The Proceedings of the Sixteenth IEEE Design Automation Conference*, pages 53–58, 1979.

Vita

- Masud-ul-Hasan
- Born in Multan, Pakistan
- Received Bachelor's degree in Electronics Engineering from the N. E. D. University of Engineering and Technology, Karachi, Pakistan in 1988.
- Worked as Electronic Engineer in the Electronic Development Cell of M/s Pakistan Steel Mills Corporation, Karachi, from Feb. 1989 till Dec. 1989.
- Completed Master's degree requirements at King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia in June, 1993.