AREA ESTIMATION OF VLSI CIRCUITS

by

Fadi Joseph Kurdahi

Technical Report CRI-87-40

A Dissertation Presented to the

FACULTY OF THE GRADUATE SCHOOL

UNIVERSITY OF SOUTHERN CALIFORNIA

In Partial Fulfillment of the

Requirements for the Degree

DOCTOR OF PHILOSOPHY

(Computer Engineering)

August 1987

UNIVERSITY OF SOUTHERN CALIFORNIA
THE GRADUATE SCHOOL
UNIVERSITY PARK
LOS ANGELES, CALIFORNIA 90089

This dissertation, written by

Fadi Joseph Kurdahi

under the direction of h.i.s..... Dissertation Committee, and approved by all its members, has been presented to and accepted by The Graduate School, in partial fulfillment of requirements for the degree of

DOCTOR OF PHILOSOPHY

Barbara Solomon

Dean of Graduate Studies

Date July 17, 1987

DISSERTATION COMMITTEE

Chairperson

Dedication

To my family.

Acknowledgments

I would like to thank my advisor, Dr. Alice Parker for her guidance, support and encouragement. During the course of my Ph.D. studies, I developed a feeling of deep respect and admiration for her as a researcher as well as a person. She was always there to listen to my problems, be it research or personal, and to provide valuable suggestions, comments and ideas. Without the help of her insight and understanding of the problems in design automation this work would have never matured into a Ph.D. level dissertation. It is my privilege to have worked with her. For all these reasons she has my most sincere gratitude and my everlasting thanks.

I would also like to thank Prof. Melvin Breuer for his invaluable advice and comments. His excellent critiques of my work throughout my doctorate years have enhanced the presentation of this thesis.

I also thank Prof. Francesco Parisi-Presicce for serving on my dissertation committee and for taking the time to read my thesis, and Prof. John Silvester for serving on my guidance committee.

My deep thanks also go to two people who, in addition to being great friends, have, through their research, greatly affected the course of this work. Sarma Sastry has allowed me to draw on his experience in the complex mathematics of probability theory and his research on wireability analysis has influenced the first part of my thesis. Nohbyung Park has been a constant source of encouragement, moral support and great technical discussions. His brilliant work on pipelined synthesis has helped shape the later part of my thesis.

I thank my parents and my sister, for their undying love, unrelenting support, and endless sacrifices. They have raised me with all the love and care that a child can have and provided the warm atmosphere of a loving home under which roof I learned the true meaning of love. It is to them that I dedicate this thesis with all the love, respect and gratitude that I could never express with mere words.

My heartfelt thanks go to Amal, Charles, Ameer and Reema Marks, for their love and care. They were my family away from home. They cheered for me when I needed support, and brightened my life when the loneliness and being away from home made it look darker and darker. For that they have my deep gratitude and love.

I thank my friend and dear cousin Sani Nassif for his constant support and encouragement.

I thank my roommate and friend Nadim Tohme for being supportive and for coping with me when the pressures of research seemed too much to bear.

I thank my dear friends Nicholas Kozma and Walid Najjar for being there when I needed them and for lending me a shoulder to cry on. I also thank Charles Saleh, Aiman Abdel-Malek, and Jim Palumbo for the great times we had.

One of the things that I have enjoyed at USC are the many friends that I have made within the EE department: Magdy Abadir, John Granacki, Christoph Scheuric, Mitch Mlinar, Sally Hayati, Jorge Pizarro, Rajiv Jain and Esther Brotoatmodjo. Thanks to all of you for your friendship and for being such great colleagues and lunch companions.

I am also indebted to the great staff of the Electrical Engineering Department at USC for being cheerful and helpful, particularly, Carol Gordon, Joyce Tsuchida, Shirin Mistry, Diane Demetras, and Christine Estrada.

Finally, I gratefully acknowledge the financial support that I received from the ARMY Research Office (Grant DAAG29-80-K-0083 and Grant DAAG29-83-K-0147) and from the Lebanese National Council for Scientific Research.

Contents

List of Figures

List of Tables

Abstract

Traditionally, the process of designing VLSI chips proceeded through specification all the way to chip layouts. At this point the resulting layout might have been too big to fit on a chip due to yield considerations which limit the size of the die. Under such circumstances the designer will have to redesign the circuit with modifications. This design-redesign sequence could be repeated several times, consuming time and money. The research presented in this thesis aims at reducing the design time by providing the designer with means of predicting the outcome of the various design steps prior to their execution. These predictions usually take little time to produce compared to the actual design steps.

There are two components which contribute to the area of a chip: functional (or logic) and wiring. In the first part of this thesis, we present a probabilistic model for estimating the wiring area of standard cell type chip layouts. The model is verified through simulation. Validation with respect to real chip layouts indicated that the model estimates are accurate to within 10% of the actual area. Experimental studies on chip layouts aimed at characterizing the procedures used for layout as well as the type of circuits being laid out using a well known empirical relationship known as Rent's rule. We investigated the relation of Rent's rule to the average wire length, an important input parameter to our model.

Estimating the size of the logic in a circuit is investigated in the later part of the thesis. We assume that the abstract functionality of a design is given in the form of a dataflow graph, where node are operations and edges are values.

The design cost is bounded by bounding the costs of the cheapest design and the fastest design.

The automatic synthesis programs used by the ADAM system at USC produce a partial register-transfer level design where the operator cost is known. Estimating the total functional cost of the design is now reduced to estimating the number of registers required for storing the values. We borrowed an algorithm used in channel routing to allocate values to registers. The algorithm is time efficient, so estimates of register count can be produced by actually performing the allocation.

Chapter 1

Introduction

When a machine begins to run without
human aid, it is time to scrap it...
–ALEXANDER CHASE, *Perspectives* (1966)

1.1 Motivation

Automating the design process has become an inevitable necessity as chips increase in size and complexity, and human designers can no longer keep track of several hundreds of thousands of components on a chip while guaranteeing the correctness of the chip function. Producing good digital designs automatically is only possible if accurate estimates of chip area and speed are possible at the early stages of the design process. We can state the following reasons in justifying this statement:

1. **Yield considerations:** Integrated circuit chips are built on *dies* of silicon. Dies are cut from *wafers*. Not all the dies on a wafer are guaranteed to be functional due to the presence of *defects* in the silicon at some regions of the wafer. The *yield* of the wafers, defined as the ratio of functional (defect free) dies to the total number of dies, has been found to be an exponentially decreasing function of the die size [Mur82]. The larger the die, the higher the probability is of finding defects in it. Since cost is directly affected by

yield, estimating the size of the chip before performing the time-consuming and costly physical design will enable the designer to assess the cost and feasibility of the design at hand in the early stages of the process.

2. **Floor planning considerations:** During the design process, and prior to starting the layout procedure, the designer usually tries to *floor plan* the design. Informally, floor planning means roughly allocating space on the chip for each of the major parts of the design (such as the data path, memory, microcode and control parts in the case of a microprocessor) such that the total chip area is as small as possible, the chip aspect ratio is within a certain range, and the global communication length between the major design parts is as small as possible. Area and shape estimation of the major design parts at the floor planning stage is a useful tool because good estimates mean floor plans that do not have to be greatly modified during layout.

3. **Synthesis considerations:** Area estimation could also be a useful tool during logic synthesis. In logic synthesis, there are usually some constraints on the final design. In many cases, such constraints involve the area of the design. Having an area estimate of the design early in its design process is useful in predicting if the design would satisfy the area constraints and hence avoiding a lot of time that would be spent on further carrying out the synthesis of a design which would not satisfy such constraints. This argument will be further discussed in the following section.

1.1.1 Design descriptions

With the advent of VLSI, it has become almost impossible to manage the design information at one level of description, hence the need of representing the design in different *level.* For our purposes, the design can be described at the level depicted in Figure 1.1 :

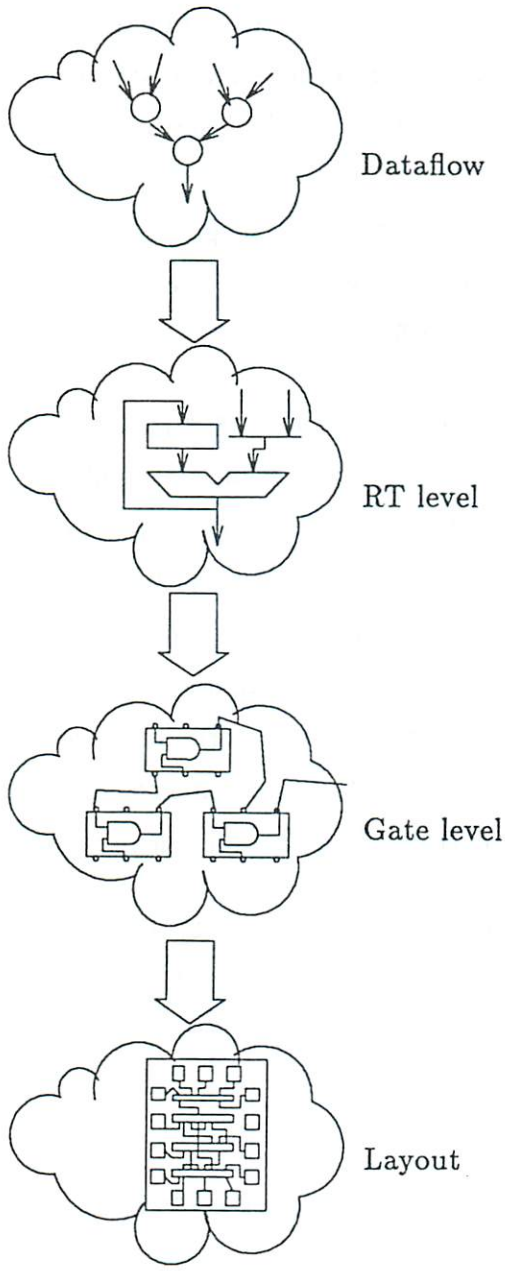


Figure 1.1: Different design descriptions

- **The Functional Description** is the most abstract view of the design. It is usually the initial input specification to the design process. At this level of description, the abstract function of the system is specified in terms of abstract entities such as operations and values, without really specifying the crude internal details and structure of the end product. There are several ways of functionally describing a digital design, such as high level languages, natural languages, and dataflow graphs. In this thesis, we will assume that such a description is given as a dataflow graph where operations are the nodes and values are the arcs. Dataflow graphs are formally defined in Section 5.2.

- **The Structural Description** is the level at which the various logic components of the system are identified. The design logic can be specified in terms of *Register-Transfer Level* components such as adders and registers, which are themselves specified in lower level details in terms of *Boolean Gates* drawn from a *library* of pre-designed components. Structures fall into two broad categories, operators and storage elements. Even though these structures perform very well defined functions and operations, the topology of the interconnections is still not resolved and they are still abstractions of the actual silicon layout.

- **The Physical (layout) Description** is the level in which the various structures in the design have actually been mapped onto silicon. So, we need to describe the topology of the circuit as it would look on the actual chip.

1.1.2 The design process

Traditionally (Fig. 1.2), the design process starts with a functional description of the circuit. Once the designer has specified the design function as a dataflow graph, the next step is to generate a register-transfer (RT) level design implementation from the high level specs. This procedure is referred to

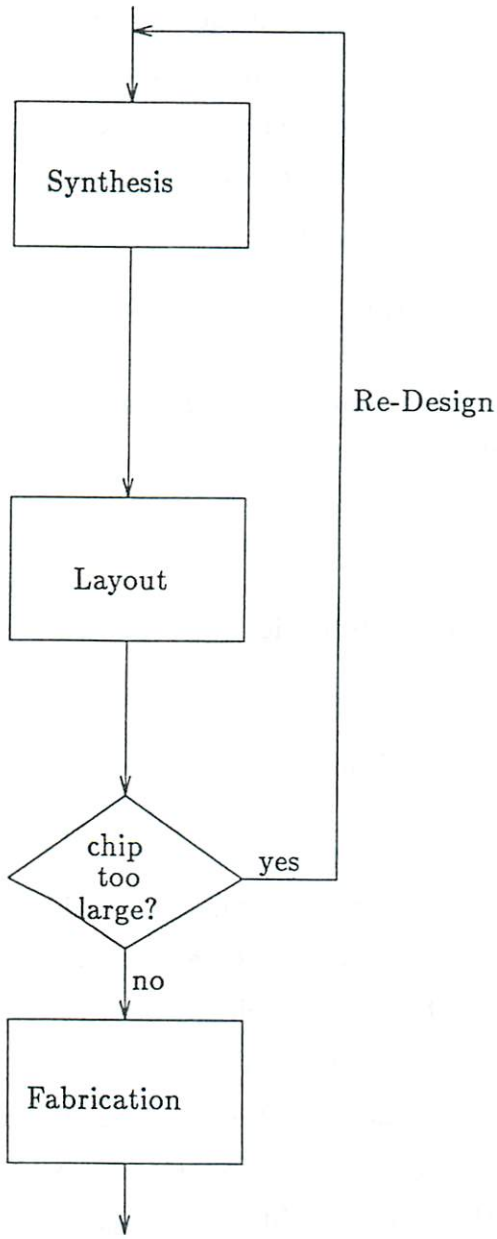


Figure 1.2: The design process

as *synthesis*. Synthesis is a process which takes as input a functional design description and constraints on various design aspects, such as area (cost), speed or power. The output of the synthesis process consists of an RT level design implementation which satisfies the specified constraints and is optimal, and a control graph specifying the temporal order of operations. In synthesis programs developed at USC and elsewhere, the synthesis process is divided into subtasks which, in general do not execute sequentially, nor do they execute only once. Major subtasks include

design style selection: e.g. pipelined, bus style, mux style,

clocking scheme synthesis: how many clock phases, and what clock cycle ;

scheduling: allocate operations to time control steps;

operator allocation: allocate to operations specific RT level modules, or *operators*; and

register allocation: allocate storage elements to values which need to be stored.

Intuitively, the number of possible choices and alternatives involved during this process, and the interdependence among the different subtasks contribute to boosting the time complexity of synthesis to combinatorial levels. Indeed, it has been shown that many of the synthesis subtasks which must be tackled in order to arrive at a solution are combinatorially explosive (or *NP-Complete*). Added to that complexity is the complexity introduced by the interdependence amongst the different subtasks, whereby one decision in a particular subtask may well affect decisions in other dependent subtasks.

Once a structural level design descriptions is synthesized, the next step is to generate a chip layout of the design. The layout process takes as input the a description of the logical structures of the circuit in terms of their gate level

components, and the interconnections between the various components. The layout task proceeds in two stages:

Placement: the various gate level components of the circuit are placed so that the ones which are highly connected are placed as close to each other as possible with the global aim of reducing the total layout area of the logic and the interconnect.

Routing: Once the placement is done, routing decides upon the topology of the interconnections. The main goal is to insure that all the interconnections are routed within minimal area.

A more detailed discussion of the layout tasks is presented in Section 2.2. The layout process can be very costly. As the circuit description evolves from abstract functional to structural logic, the amount of detail, or the number of components, increases thereby making the problem size larger. In addition, the placement and routing tasks have been shown to be *NP-Complete,* which means that the execution time for each task increases much faster than the problem size itself. Automated placement and routing schemes for commercial chips have been know to take days of computer time.

From this brief discussion, one can infer the following conclusions:

1. In general, design tasks are very costly since they take a long time to execute.

2. It is hard, and sometimes impossible to find the optimal solutions of each subtask, let alone that of the whole design process. Thus, it is difficult to guide the design process itself since there is no apriori knowledge of the outcome of each step of the design tasks.

3. Providing the designer with means of *evaluating* the design at hand before and during the design process would be quite useful, and may well reduce the design turnaround time.

The third conclusion is particularly true when one looks at the global design process, as shown in Figure 1.2, where one may process an abstract initial design specifications all the way to a silicon layout only to discover, after spending time and money, that the resulting design does not satisfy all the initial constraints, most important being speed and chip area. This means that the design process has to be restarted with different constraints. This cycle is repeated until the output is satisfactory. We propose a different scenario: instead of actually performing the design functions in sequence, one could initially bypass them by predicting the end result. Since such predictions are expected to take orders of magnitude less time than the design functions themselves, the time per cycle is reduced. Hence the total design turnaround time would be reduced.

1.2 The problem statement

The underlying philosophy in this thesis is that the design evaluation can be done at any point during the design process. Initially, there is a minimal amount of information in the design description about the final implementation and its details, so naturally the design evaluation cannot produce accurate estimates of the design size. As the design process progresses, more and more details are resolved and therefore the estimation procedures become more accurate. This idea is shown in Figure 1.3 which shows a design cycle modified from Figure 1.2. The goal of this research is to provide the designer with an aid to evaluate the size of design at hand. The targeted evaluations are to be done prior to each step in the design process and to predict the outcome of that step. As discussed in the previous section, there are two major steps in the design process, synthesis and layout. Therefore designs are to be evaluated first prior to the synthesis step, where the design is specified in an abstract functional description, and the designer needs to get some idea about the design size.

Once the synthesis step is done, it outputs a gate level design description. At this point, the evaluation is aimed at providing the designer with some concrete number on the amount of silicon area the layout will take.

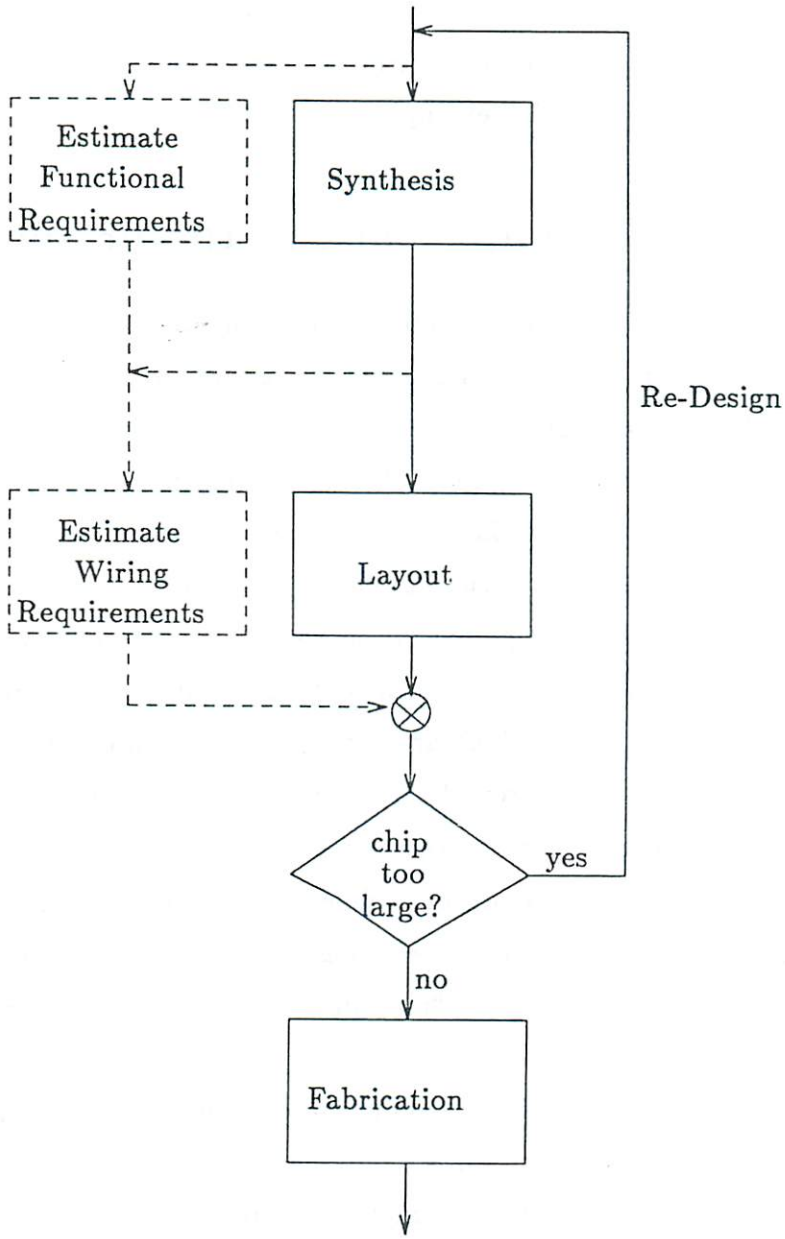


Figure 1.3: The modified design process

This thesis proposes and researches the task of evaluating the size of a VLSI chip design before each step of the design step by attempting to predict the outcome of that step. These evaluations are expected to take orders of magnitude less time than the actual design steps, therefore helping to reduce the design turnaround time. Finally these evaluations are verified and validated with respect to real life examples.

1.3 The approach to the problem

There are two major components which affect the area of a design, the *logic*, or *functional* area, and the *wiring* area. Logic area is the area occupied by the logic gates, and wiring area is the area occupied by the interconnections running between gates. Wiring is almost completely dependent on the design logic in the sense that the layout process which generates the wiring cannot start until the design logic has been generated and decided upon. Once the logical details of the design have been resolved and the gate level structure is known, the area of the logic can be easily determined if the gates area taken from a *cell library* which is composed of parameterized layouts of individual gates. What remains is to estimate the area of the wiring between the gates. Since the number of wires and the geometric features in the final layout are several orders of magnitude larger than the abstract problem size, the estimation scheme for wiring must be stochastic in nature. Once an estimation model is developed, it must be verified and validated against "real life" example layouts in order to establish accuracy measures.

Earlier in the design process, when the design is initially specified in a functional description, we only have an abstract idea about the logic of the design, and very little information about the detailed interconnect, or wiring, of the design. At this point, prior to the synthesis procedure, we need to have a better idea about the functional, or logic requirements of the design implementation. In order to estimate the functional requirements, one must make some assumptions about how the abstract design function is described. In this thesis, the design

function is assumed given as a *dataflow graph* where nodes are operations and edges are values. With this description as a starting point, the evaluation process must provide the designer with an estimate of the size of the design under consideration. As it has been observed when doing hardware synthesis, there is a large number of possible implementations of a design dataflow graph, sometimes referred to as the *design space*. Therefore the functional estimation scheme must characterize such a solution space for the design at hand and essentially give the designer bounds on the size of the possible implementations.

As discussed in Section 1.1.2, the synthesis procedure is composed of different tasks. First operations are allocated to operators, at the same time the control scheme is developed in detail. Operations are usually implemented by combinational logic, therefore this step can be though of as deciding upon the combinational logic part of the whole design logic. At this point, very little, if any, sequential logic has been allocated; this comprises the next task in synthesis, which is register allocation. Estimating the total logic requirements of the design at this point is reduced to estimating the sequential logic, or *storage* requirements.

Once the design logic has been estimated, the designer can use this information to guide the synthesis process, or if a more accurate estimate is desired, the wiring area can be estimated by using the wiring estimation model. The scheme in which these various estimation schemes interact is left to the designer.

The research of this thesis is part of the on going effort in building the ADAM system currently under development at USC [GKP84] [KGP83] [KP83]. Next we present a brief description of the ADAM system.

1.4 The ADAM system

The ADAM (Advanced Design AutoMation) system, currently being built at USC, is an integrated computer aided design system whose purpose is to aid the designer throughout the digital design process. Two views of the system is shown in Figures 1.4 and 1.5. The main features of the systems are the

design database, a knowledge based planner and an expert system which aids in the design of testable circuits [Aba85] [Zhu86]. The system components can be divided into three classes. They are

1. The *actors*, the procedures and subsystems that act on the database information. Some of these are

   - the planner, a knowledge based subsystem which acts as a 'monitor' during the design process, calling the design tools in the proper order;
   - the synthesis tools, include the RT-level allocator, the logic synthesizer, the pipeline synthesizer [PP86], the clocking scheme generator [PP85], and the silicon compiler; and
   - the analysis tools, including the area estimator, the critical path finder and the collision detector [Kna84].

2. The *objects*, including all the pieces of information related to the design process. The main objects are

   - the Design Data Structure (DDS) [KP83], a multilevel, multiview design representation,
   - the knowledge base, which contains information about the design process, and
   - the design plans. These are 'annotated tours through the knowledge base' [GKP84], i.e. a sequence of activities which would, hopefully, achieve the preset goal.

3. The *system interfaces*, including

   - A natural language interface, PHRAN-SPAN, which provides a natural language interactive media for specification [Gra86],
   - The 3DIS interface, which is an efficient method for browsing through the database [Afs85], and

- A proposed VHDL (VHSIC Hardware Description Language) interpreter tool, which translates an input description in a restricted set of VHDL [VHDL86] to the Design Data Structure.

1.5 Previous related work

1.5.1 Wiring area estimation

The wiring area estimation problem has received attention lately. There are mainly two approaches to the research on the problem: theoretical, which concentrates on wireability analysis and wiring space estimation, and experimental, which aims at developing area estimators for some specific layout systems based on previous experience with these systems.

Wireability analysis mainly deals with the problem of modeling the behavior of wires on a chip layout with the aim of predicting the amount of space to be allocated for wiring the interconnections. Almost all the wiring models proposed so far are *stochastic* in nature.

1.5.1.1 Wiring space estimation for PC boards

One of the first well-known works in this field is by Sutherland and Oestreicher [SO73] in which they develop 'a theory for choosing printed circuit board dimensions in order to avoid crowding of the printed wiring'. Components are assumed to be randomly placed on the board. By assuming also that a 'good' router is used for wiring, the expected number of wires crossing a cutline at the center of the board is found to be 1/4 the number of active pins on the circuit. It was also predicted that small boards are less efficient in area than large ones and that square boards are minimal in area. This work exhibits a good balance between theory and practice in the sense that each theoretical assumption and result is explained from a physical and intuitive point of view. The main drawback is that the assumption that components can be randomly

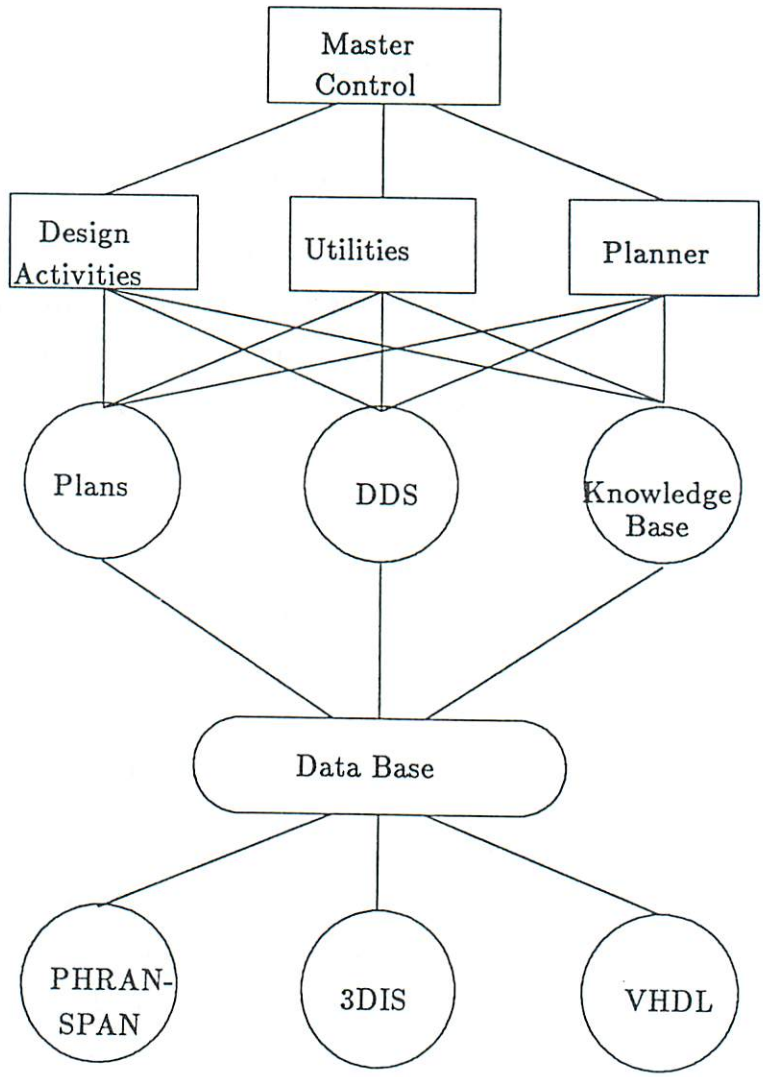


Figure 1.4: The ADAM system: A relational view

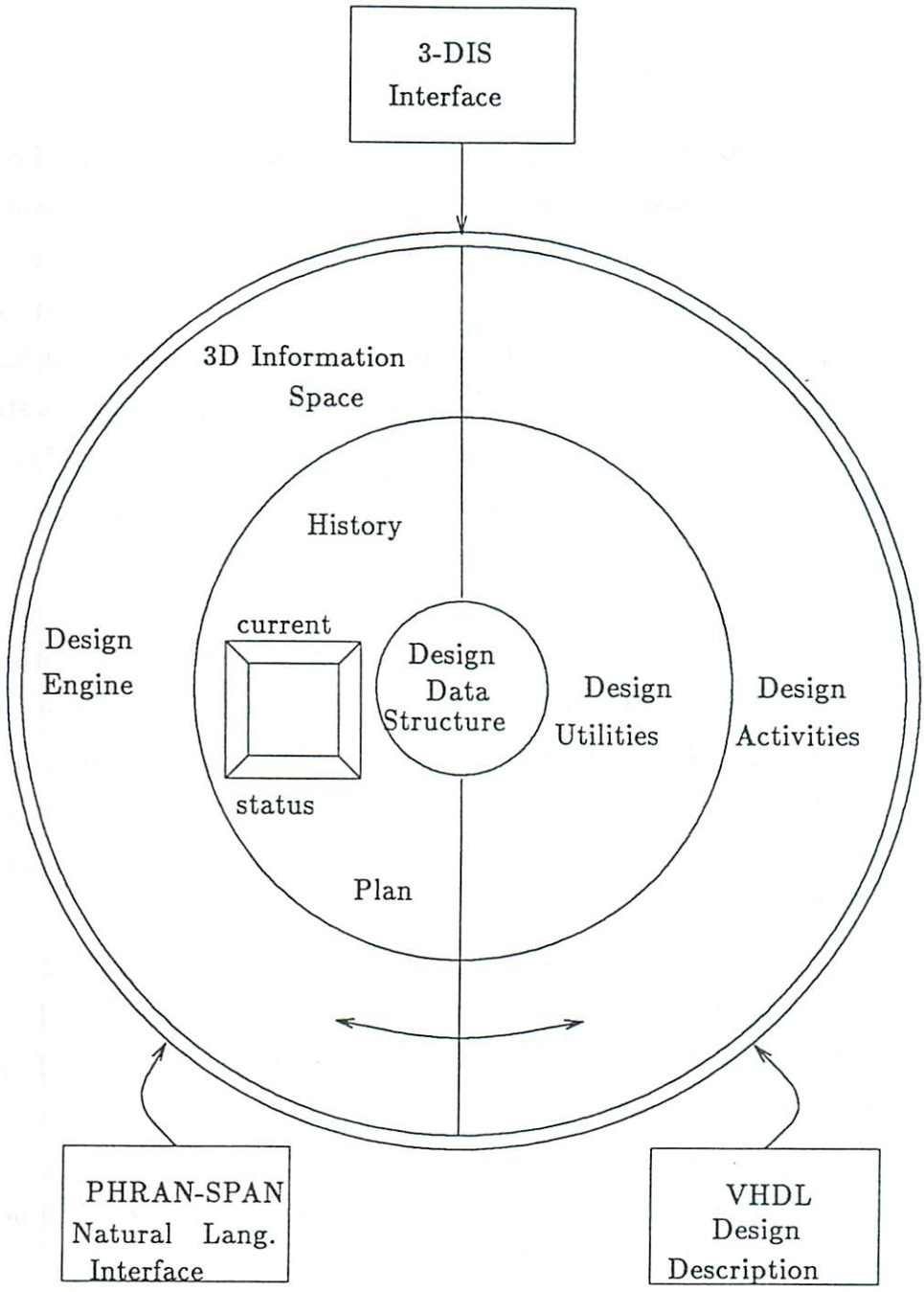


Figure 1.5: The ADAM system: A topological view

placed can not be extended to integrated circuits, since it doesn't take into consideration the connectivity information of the various subcircuits.

1.5.1.2 Wiring space estimation for gate arrays

One of the early attempts to model the placement of components was with the observation by E.F. Rent of IBM of an empirical relation existing in well placed and partitioned gate array designs between the number of components in a partitioned subcircuit and the number of its external connections to other subcircuits [LR71]. The relation (known since as Rent's rule) is further discussed in Section 4.3.1. Also at IBM, Heller et al [Hel77], developed a stochastic model for the prediction of the wiring space for a 1-D placement of cells. The model assumes that the average wire length is given and that the number of wires emerging from a cell can be modeled as a Poisson distributed random variable with parameter λ. Given a number of tracks allocated for wiring, the model predicts the probability of successfully routing the placement within the allocated space (in other words, the routability of the placement). The authors propose a heuristic extension to a 2-D placement by collapsing some of the rows and columns into a two row placement. This proposed extension has some weak points, especially in the calculation of the effective new model parameters after collapsing.

Feuer [Feu82] looked into the problem of predicting the wire length distribution and estimating the average wire length of IC's. The main result of that work is that if the partitioning of a logic graph exhibits Rent's rule, then the wire length distribution is expected to be of the form $q(r) \sim r^{-2(2-p)}$, where p is Rent's exponent and r is the wire length. One criticism of this model is that it assumes an 'infinite' chip. Thus, it may not correctly model the behavior of a real 'finite' chip near its edges.

In Heller's wiring model, a linear placement is assumed to be 'sliced' from an infinite width row. In [Yeh82], the model was modified to handle finite size placements by setting a limit on the maximum length of a wire, say L_{max}

ports. Hence a port can only be 'affected' by the Lmax ports to its left (assuming wires travel from left to right). The same model is used to handle 2-D cell arrays by mapping each of the main diagonals in the cell array into one 'effective' cell in the linear placement. The 1-D model is then used to estimate the routability of the array. This approach, however, gives the probability of successfully routing the array given the *sum* of the horizontal and vertical tracks at each cell, without looking at the distribution of these tracks in each direction. Hence, it might lead to results which may be overly optimistic. For example, the routing may result in insufficient tracks in, say, the vertical routing channel, but the sum of the tracks in the vertical *and* horizontal channels could still be predicted as 'sufficient' by the model if tracks were overallocated to the horizontal channel and underallocated to the vertical channel.

The model in [ElG81] is an extension to the one in [Hel77]. A two dimensional stochastic model for wiring is presented. A gate array chip is modeled as a two dimensional array of points with wires emerging form a point with a Poisson distribution of known parameter. A wire travels between its source and destination with a trajectory determined by flipping a fair coin. The distance a wire travels is a random variable drawn from a known distribution. The analysis indicated that the width of the widest channel grows as the average wire length. Also, from the model assumptions, the widths of the individual channels are found to be dependent, Poisson distributed random variables. Upper bounds on the means of these variables were estimated numerically. Using these estimates, the total wiring area on the chip was estimated. One of the problems with this model is that it is not clear how tight the upper bounds can be. This means that the channel width estimates can be much larger than the actual values and the designer might end up allocating much more wiring space than actually needed resulting in unnecessarily large chips. A more thorough critique of this work can be found in [Sas85].

In his thesis, Sastry [Sas85] addressed three problems related to wireability analysis in gate arrays. They are: *wiring space estimation, wire length distribution and average wire length*, and *routability*.

For the first problem, a gate array is modelled as a grid of *channel intersections* and the problem of wiring space estimation is reduced to estimating the dimensions of these intersections. This is done by classifying the wires at an intersection into six different types, each modelled as a Poisson distributed random variable. From this and the knowledge of the wire length distribution, the expected widths and heights of each intersection are obtained. Asymptotic results were also obtained regarding the widths of the intersections for 'large' chip, which were found to grow as the second moment of the wire length distribution. In treating the second problem, an equivalence relation was established between Rent's rule and the wire length distribution in the sense that Rent's rule (or any similar relation) does indeed provide all the necessary information about the wire length distribution (or vice versa). In the particular case of Rent's rule, it was found to correspond to wire length distributions of the Weibull family. This was experimentally validated with data from real layouts. The third problem of routability was treated as two overlapping problems, vertical and horizontal routabilities. vertical (horizontal) routability is defined as the probability of successfully routing a chip for a given number of vertical (horizontal) channels and *sufficient* horizontal (vertical) channels. The approach was to find the factor by which to multiply the previous estimates of channel intersection dimensions in order to achieve a given routability. This model, however, separately treats the routability problem for vertical and horizontal channels and does not look at the coupling of the effects of wiring in the two dimensions. Furthermore, a chip is modelled as a 'slice' of a doubly infinite array and hence, for real 'finite' chips, the estimation of channel intersection dimensions near the edges may not be as accurate as those for the intersections well inside the chip.

1.5.1.3 Wiring space estimation of custom layouts

The above works assumed that designs are laid out in the gate array design style. The problem of wiring space estimation of custom logic was addressed by Syed [Sye81]. He assumes that a placement of arbitrarily-sized rectangular

blocks is given, along with their interconnections. He constructs a *channel graph* for the placement, where edges are the routing channels between the blocks and vertices are the intersections of these channels. The widths of the channels (edges of the graph) are then estimated using a stochastic model for wiring in which pins are assumed to be generated along a channel with a Poisson distribution, and wires lengths are exponentially distributed. Once this is done, the initial placement of the blocks is then modified so as to accommodate the channel width estimates with minimal total displacement of blocks. Routing is performed in two phases. First topological routes are assigned to wires. In the second phase, tracks are assigned to wire segments. During this phase, the placement may be further modified if more space than predicted is needed in some channels. This process of track assignment and placement modification is repeated iteratively until complete routing is achieved. Syed's model concentrates on estimating the area needed for *global* routing between the major blocks of a chip and does not deal with estimating the local wiring area within blocks and hence, the area of the blocks themselves.

Another approach to custom layout area estimation is by Ngai [Nga83]. He assumes given a channel, a number of tracks allocated to it and a set of nets of three types: right and left nets entering the channel from right and left, respectively, and center nets which are born and die inside the channel. The problem is to estimate the routability of the channel over all possible pin permutations. The channel wiring is modelled as a Markovian stochastic process with state changes occurring at net terminals (pins). A state at a pin is defined as a quadruple of random variables representing the number of nets of different types up to that pin, the density at that pin being a simple function of these variables. The conditional transition probabilities are found and a recurrence relation is used to find the state occupancy probabilities from which the distribution of the density function is determined and the routability estimated. Since this model predicts routability over all possible pin permutations and not the subset corresponding to 'good' placements, the predicted routability figures may deviate from the actual ones.

1.5.2 General (Functional) area estimation

In the field of general area estimation, we note two works. The first is by Liu [Liu83] in which two models were developed : the point model for gate arrays and the rectangle model for general cells. In the point model, the well known separator theorem from VLSI complexity theory [Tho80] is used to prove some orders of growth of wiring area as a function of the number of cells. Besides being well known corollaries of the separator theorem, the results have little practical applicability to physical design since they are only expressed as orders and not as concrete numbers. For the rectangle model, the author develops an algorithm for testing the routability of a set of blocks given their relative placements and another algorithm to route the channels. Finally, the blocks in a chip are assumed to be implemented as PLA's and an algorithm is presented for partitioning a large PLA into several smaller PLA's.

The work by Ueda et al. [UKH85] is a more experimental approach to the area estimation problem. The authors describe a layout system, ALPHA, which uses the standard cell design style and in which an area estimator, CHAMP, is implemented. During the floor planning process, CHAMP estimates the areas of the different standard cell blocks by using empirical formulas obtained by running numerous layout experiments on several designs. The area estimation figures presented for some chips are within 10% of the actual area. The estimation formulas are, however, empirical in nature and no theoretical justification is provided. Hence it is doubtful whether such formulas are applicable in another system.

Estimating the amount of logic in a design implementation given its high level functional specification was researched by Leive [Lei81]. His research focused on developing a logic synthesis and module selection system. In order to evaluate the synthesis system, he established some *predictors* for area, delay, and power. The predictors are extracted from experience with previous designs and normalized with respect to some global design characteristics, such as number of nodes (or operations) in the design, major bit width, and control path length.

When a new design is to be evaluated the predictors are multiplied by the appropriate design parameters and an estimate of area, delay, or power is obtained. These estimation schemes are global in nature and pay minimal attention to the design details, therefore their use is limited to crude estimation. They are also dependent on the synthesis scheme, which means that new predictors have to be developed whenever the synthesis scheme is changed.

In his thesis, Park [Par85] established some bounds on the component count for pipelined design implementations. These bounds are used to initialize the pipelined synthesis algorithms embodied in Sehawa [PP86]. Later, in [JPP87], it was proved that an optimal design with maximum utilization of hardware would adhere to these bounds. An interesting issue is also discussed in this same paper regarding the relation between cost and delay of a design implementation, which was predicted to be of the form $AT^{\alpha} = k$ with A and T being the cost and delay of the implementation, α and k constants. The α exponent is conjectured to be related to the hardware utilization, so for maximum utilization, or optimal design, $\alpha = 1$ and Park's bounds apply. In Chapter 5, we see that a similar relation is empirically observed at a lower level when we analyze the tradeoffs of adder modules.

1.6 Thesis outline

Chapter 2 treats the problem of estimating the amount of wiring space in a chip layout. The details of the design logic are assumed known. Layouts are assumed done in the standard cell design style. A simple probabilistic model is developed for placement and routing of components on the chip. This model is used to estimate two parameters which reflect the amount of wiring space needed.

In Chapter 3, we verify the model of Chapter 2. First chip layouts are simulated and simulation results are compared to the model predictions. Second, real chips designs are laid out and the layout areas are compared to the model predictions. More examples were generated using the MP2D industrial layout system to further validate the model implementation, called PLEST (PLotting

ESTimator). The behavior of the estimates for different layout configurations is also studied.

Chapter 4 deals with studying the behavior of wires in chip layouts. The distribution of wire lengths in real chips is studied. We also investigate a well-known empirical relation of wiring space to circuit size called Rent's rule. Using this rule, we establish guidelines for estimating the average wire length of a circuit, an important input parameter for the model in Chapter 2.

Chapter 5 looks at the problem of estimating the amount of logic in a design. At the initial stages of the design process, only the abstract design function(s) is specified. Therefore a large number of register-transfer level implementations are possible. We establish some bounds on the functional design cost by partitioning logical components into three categories, namely operators, registers, and routers and bounding the costs of each category for a given design.

Chapter 6 is concerned with accurately estimating the amount of sequential logic (or registers) in a design, once the abstract operations have been allocated to operators. We present an algorithm borrowed from channel routing theory which is used to allocate the values in the design to registers. The basic algorithm is extended to account for conditional branches in the functional design specifications and to properly handle pipelined designs.

Finally, conclusions are drawn in Chapter 7 and future research topics are suggested and discussed.

Chapter 2

A model for wiring area estimation

Lest men suspect your tale untrue,
Keep probability in view.
—JOHN GAY (1727)

2.1 Introduction

The main goal of this chapter is to develop a probabilistic model that will be used to obtain estimates of the dimensions of a chip, once the gate level description of the circuit to be implemented is given. In this chapter and in Chapter 3, we will assume that the layout of a chip is done in the *standard cell* design style. This style is described in Section 2.2. This choice was influenced by the following arguments:

- The standard cell design style is widely used in the industry.
- It offers a compromise between the rigid regularity of gate arrays and the intractable randomness of full custom design style. The standardization of the basic component cells coupled with the flexibility of the placement (as compared to other semi-custom design styles, such as gate arrays) has

helped in automating the layout process. This, in turn, has helped reduce the design turnaround time for standard cell chips.

- While the standard cell design style is inefficient for some highly regular types of circuits, such as memory and PLAs, standard cell blocks can coexist with other types of structures on the same chip. Most other semi-custom design styles (e.g. gate arrays) do not usually support such integratability.

- Much research has been focused on gate array area and wiring space estimation [Sas85,HMD78]. Most of this research provides excellent guidelines for wiring space estimation of gate arrays, and the basic concepts and philosophies could be readily applied to standard cell type designs.

2.2 The standard cell design style

Figure 2.1 shows a typical standard cell chip. The standard cell layout scheme uses a library of pre-designed cells (usually of the SSI/MSI level complexity). These cells usually have the same height, but different widths depending on the complexity of the functions they are to implement. When placed together, the cells will abut vertically so that their power, ground (and sometimes global clock) lines will be connected automatically. Other signals are available on pins situated on the top and bottom of the cells. In many cases cells are designed so that any cell signal is available on equivalent pins on the top and bottom of the cell thereby giving the designer the freedom of connecting on the top or bottom. These cells are sometimes called *double entry* cells. The cells are arranged in rows of near equal sizes. The spaces between adjacent rows are called channels. Channels are used to route the signal wires between the cells. Routing between cells in non-adjacent rows can be achieved by using *feedthroughs*, which consist of cells whose input and output pins are electrically equivalent. Feedthroughs are inserted in a row to permit multirow signals to be routed across the row instead of going around it.

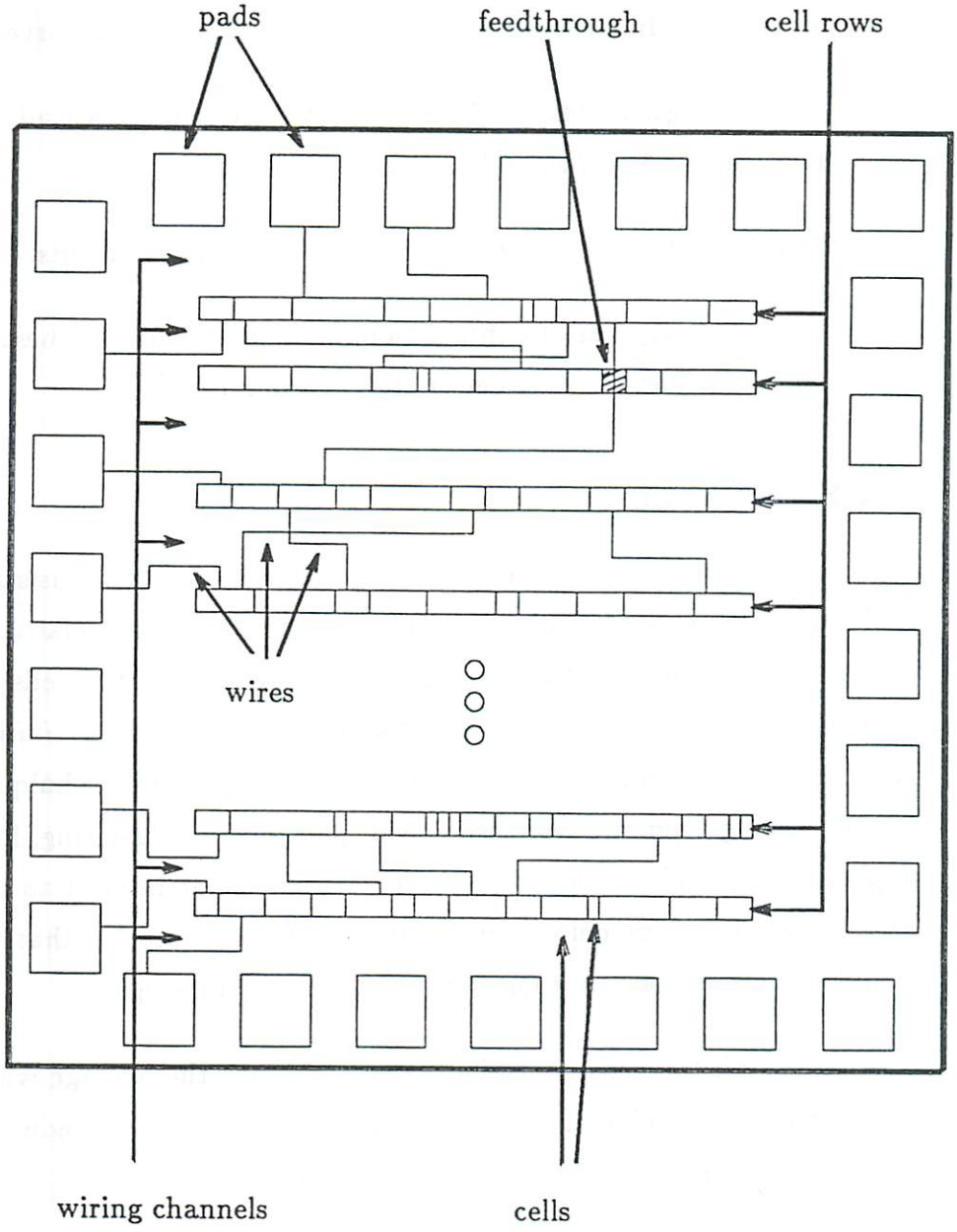


Figure 2.1: A typical standard cell chip

2.2.1 Complexity of standard cell layouts

Several observations concerning the complexity of standard cell layouts have been noted. In one reference [Hic83], the following is observed :

- As row length increases, the routing area grows quadratically with N (the number of cells) if the cells are *randomly* placed.
- This rate of growth becomes linear for *optimal* placements.
- Since it is very hard to obtain an optimal placement, the best that one can hope to achieve is an order of growth around $N^{1.5}$.

2.2.2 Placement

The placement procedure for standard cell layouts consists of assigning cells to rows (i.e. partitioning the design),and of assigning relative positions to cells within a row (i.e. finding a 'good' permutation of the cells on the row). There are two major phases in the placement procedure, the *initial placement* and the *iterative improvement* phases. There are several techniques for placement. Among them are row assignment, pair linking, clustering, force directed, bipartitioning and row folding [Ric84]. Of particular interest to us is the last technique. In our model, we chose to make the assumption that placement of standard cells is done by going through the following steps:

1. Place all the cells on a single row such that the average wire length (or some similar objective function) is minimized. This procedure is known as one-dimensional (1-D) or linear placement.
2. Fold the row into a number of smaller rows so that the resulting block layout satisfies some *a priori* set constraints on area and shape. Section 2.3.5 explains the folding process in further details.

We chose to make this assumption for the following reasons:

- 1-D placement and folding is a placement technique which has been successfully used for a long time in many layout systems (e.g.[Kan83] [UKH85] [SU72] [Sup83], [FN78]) and has been known to produce good placements.
- 1-D placement is much easier to model than 2-D placement (which is the case for most other placement techniques), hence it enables us to better analyze the placement problem and yield more confident area estimates.

2.2.3 Routing

After placement is finished, the routing of nets is to be done. This is usually done in two major phases, *feedthrough assignment* and *channel routing.*

2.2.3.1 Feedthrough assignment

Before carrying out the actual routing, one has to decide on the number and location of feedthroughs and nets which will be assigned to each feedthrough. Some cell libraries have two types of feedthroughs: *cell feedthroughs* which are cells inserted in rows during this phase and would permit the connection between two consecutive channels, and *terminal feedthroughs* which are parts of some cells. These are electrically equivalent terminals that run vertically through a cell which would otherwise be performing its ordinary function.

Several criteria are used to select feedthroughs such as: the number of cells to be inserted in a row, the maximum number of rows a net can feedthrough, the horizontal length a wire must travel before reaching the next feedthrough and whether to pass a wire through a feedthrough or to route around the row.

2.2.3.2 Channel routing

Once feedthroughs are assigned and the global route each net must take is known, the track assignment is to be done. A track is a 'clearance' which one wire can run through. The size of the track depends on the technology used and its associated design rules which specify the minimal spacing between wires.

The track assignment problem is known as the classical channel routing problem for which several algorithms and variations exist. Among the algorithms for channel routing we have the Lee [Lee61], Hightower [Hig69], Mattison [Mat72], Hashimoto and Stevens [HS71], Rivest and Fiduccia [Riv81] algorithms. In most technologies, vertical routes run on one layer, horizontal routes on another. Doglegging is used to break routing conflicts [Deu76]. Most good routers usually need only one more track in addition to the channel density for routing the channels. The *local density* at a certain ordinate is defined as the number of nets crossing a cutline at that ordinate. The maximum local density over the channel length is the *channel density.* The channel density represents a lower bound on the channel width. Local density is usually computed at grid points and is a discontinuous function.

2.3 A probabilistic model for standard cell area estimation

2.3.1 Estimating the area of standard cell chips

Figure 2.2 shows the three area components in a standard cell chip,

- The cell area is the area of the logic cells, sometimes referred to as the active area. This area is readily known from the input circuit description or from earlier estimation procedures.

- Wiring area is the area taken by the wires connecting various cells together. Estimating the wiring area is far from a trivial task, as we don't know *a priori* how the cell are placed, and thereby don't know the locations of the end points of the wires, and the routes the wires would take from source to destination(s). A major task of the model described in this section is to estimate the wiring area on a chip.

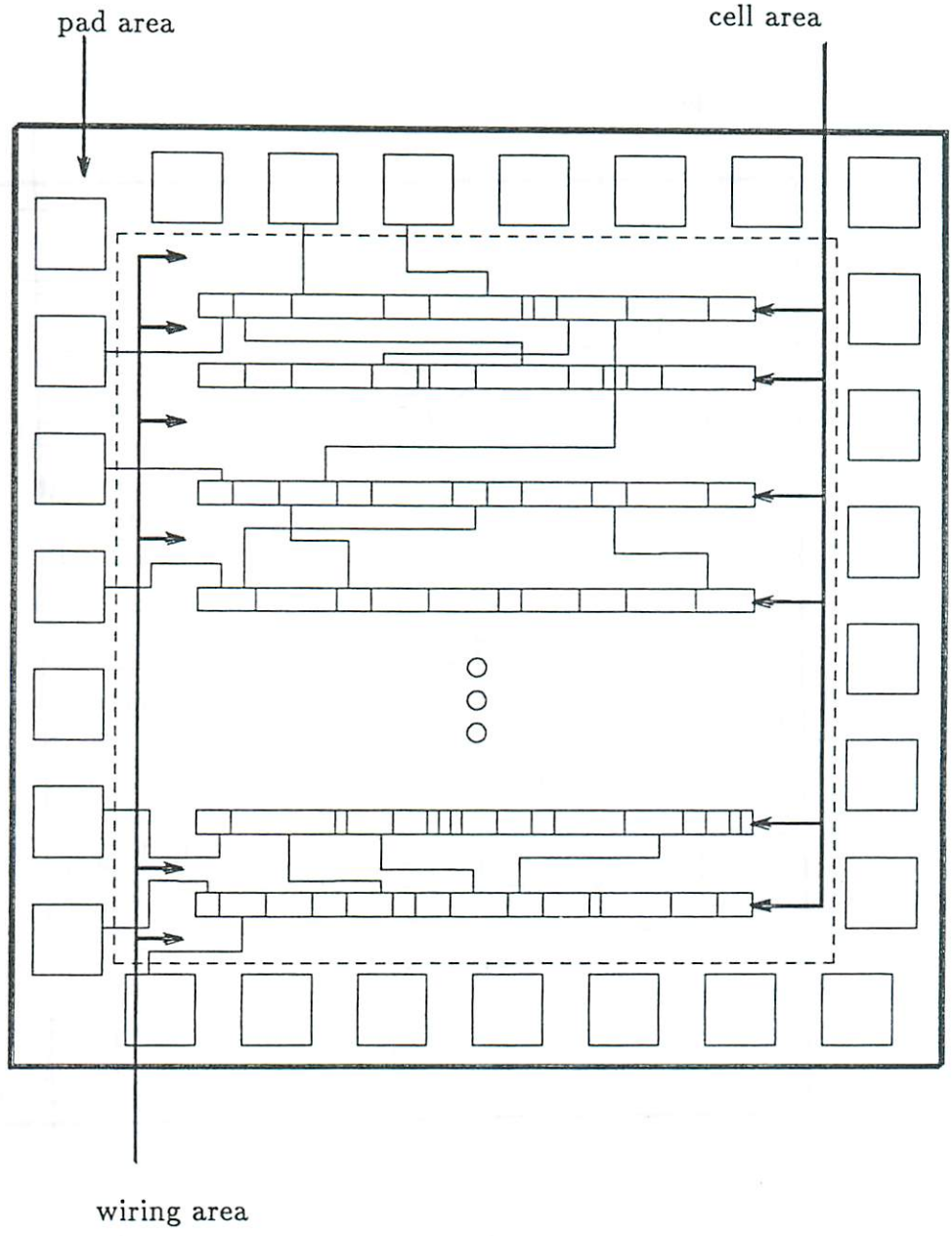


Figure 2.2: Area components in a standard cell chip

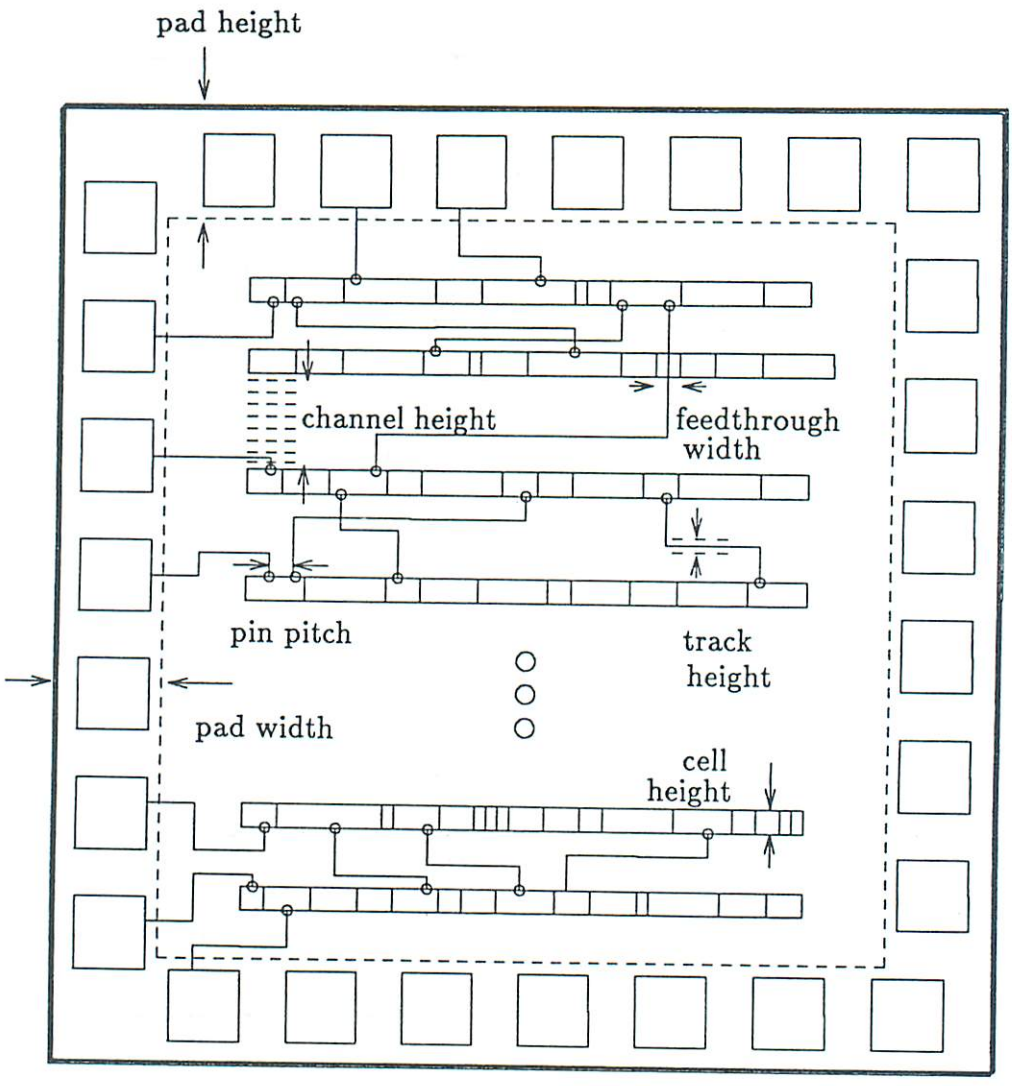


Figure 2.3: A standard cell chip parameters

- Pad area is the area taken by the pads placed around the periphery of the chip to provide connections to the outside world. Once the cell area and the wiring area are known (or estimated), it is almost trivial to estimate the pad area, since the number of inputs and outputs of the chip is known, and the dimensions of the individual pads are assumed to be given.

We chose to use probabilistic techniques in estimating the wiring area because of the following reasons:

- For a given gate level description of a design, the number of possible layouts that can be generated is huge. Attempting to characterize or predict each possible layout is an impossible task.

- Even when the solution space is restricted to include only the *good* layouts, i.e. the ones with the smaller areas, the variations are so numerous that a deterministic approach is not feasible.

- Probabilistic techniques have been used in the past to model placement and routing in more regular types of layouts, namely gate arrays and master slice type chips, and a relatively large amount of literature exists which helped to provide some guidelines for this work, as well as conceptual support of the ideas.

In Figure 2.3 we show the various parameters associated with the area of a standard cell chip:

- *cell height* is the height of the individual cells, constant for a given cell library,

- *number of rows,* r usually set so that the chip is of near square shape,

- *track height* is the minimum distance between two adjacent horizontal wires,

- *feedthrough width* is the width of the feedthrough cells described in the previous section, and

- *pad width and pad height* are the dimensions of the individual pads.

It is easy to see from Figure 2.3 that the dimensions of the standard cell chip are given by the simple relations shown in the figure. All the quantities in relations are known chip parameters except for two, the number of *horizontal wiring tracks*, and the number of *feedthroughs* in the cell rows. In the following sections, we present a probabilistic model for cell placement and interconnections which forms the basis for estimating these two quantities.

2.3.2 Assumptions

We make a number of assumptions about standard cell layouts. They are

- cells are arranged in rows of roughly equal sizes;
- cells are of the **double entry** type, i.e., equivalent pins are present on both top and bottom parts of each cell; this allows us to 'collapse' both pins into one equivalent pin, and hence the cell row is collapsed into a line;
- all cells have constant **pin pitch** which is defined as *the distance between two consecutive pins or pin slots* and cell width is an integer multiple of the pin pitch, thus, a cell row can be modeled as a row of *pin slots*;
- all nets are two point nets (i.e. connecting exactly two pins);
- all wires follow **minimal rectilinear paths**; no backward moves are allowed; and
- placement is done by first placing the cells on *one* row so that the total routing area in minimized (or nearly minimized), then *folding* the row into a number of rows so that a desired aspect ratio is achieved.

While this last assumption seems rigid, it allows us to analyze the problem in a much simpler fashion. The main idea can be expressed as follows: *Given the*

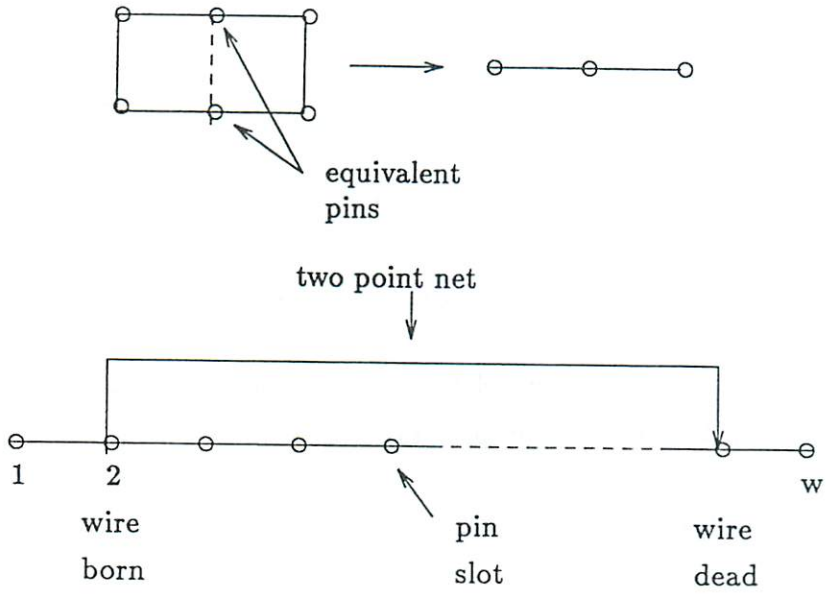


Figure 2.4: The row model

extreme complexity of modeling the problem as a two-dimensional one, assume that the placement is done by folding. Starting with the one row linear placement, which is relatively easier to model, predict the wiring space requirements after folding, given that pin positions are transformed in a predictable manner as explained in Section 2.3.5.

2.3.3 Description

An example of the row model is shown in Fig. 2.4. Some of the characteristics of that model are the following :

- Since cells are of the *double entry* type, the top and bottom equivalent pins in a cell are actually 'collapsed' into one 'equivalent' pin in the model.
- A cell row is modelled as a row of *pin slots* where the distance between two adjacent slots is equal to the pin pitch.
- A pin slot is occupied if a wire emerges from it.

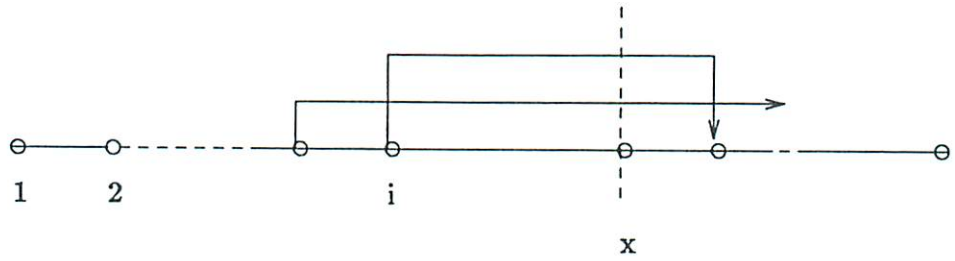


Figure 2.5: Density at a point x

- The total number of pins slots in the block (i.e. the sum of the widths of all the cells, in pin pitches) is w.
- The total number of nets in the block is N.

2.3.4 Single row case

We now look at the case of estimating the routing requirements of a single row configuration, that is, when the cells are all placed on one single row. This analysis is necessary for the more general multiple row case, which will be presented in the Section 2.3.5. We make the following additional assumptions :

- Pin slots are labelled $1, 2, \ldots, w$ from left to right.
- Wires travel from left to right. A wire starting at a pin slot is said to be *born* at that slot. The birth of a wire at a pin slot i is a random event with probability $p_B(i)$.
- A wire born at a pin slot i will *terminate* (or *die*) at a certain pin j to the right of i. This event is assumed to be dependent only on the distance between i and j (the length of the wire) and not on the individual values of i and j. In other words, the length of a wire does not depend on where it is born.
- The length of a wire is assumed to be a random variable, L, with a probability density function $p_L(l) = Pr\{L = l\}$.

We will define the density at point x, $d(x)$, as *the number of wires crossing a vertical cutline at* x as shown in Fig. 2.5. The objective is to estimate the average value of $d(x)$. To do that, we have to look at the following events:

- $A(i)$ = a wire is born at i, and
- $B(i,x)$ = a wire *crosses* x given that is is born at i.

The probability of the first event is given by $p_B(i)$. Now a wire born at i will cross x iff its length L is such that

$$x - i \leq L \leq w - i$$

So,

$$Pr\{B(i,x)\} = Pr\{x - i \leq L \leq w - i\} = \sum_{m=x-i}^{w-i} p_L(m)$$

Now, the probability of a wire being born at $i \leq x$ *and* crossing x is

$$Pr\{A(i)\}.Pr\{B(i,x)\}$$

So the average number of wires crossing x is given by

$$E\{d(x)\} = \sum_{i=1}^{x} p_B(i) \sum_{m=x-i}^{w-i} p_L(m) \tag{2.3.1}$$

However, the number of tracks needed is equal to the maximum density throughout the row. This means that, in order to estimate the number of tracks, one must estimate the expected value of the variable $d' = \max_{1\leq x\leq w} d(x)$. This, however, may be difficult, especially for arbitrary forms of $p_L(l)$ and $p_B(i)$. One way of approximating the expected value of d' is to use $\max_{1\leq x\leq w} E\{d(x)\}$. It was shown [Sas85] that

$$\max_{1\leq x\leq w} \{E\{d(x)\}\} \leq E\left\{\max_{1\leq x\leq w} \{d(x)\}\right\} \tag{2.3.2}$$

This means that the approximation is actually a lower bound on $E\{d'\}$. In section 3.2 we will show some simulation results which compare the two variables.

The next task is to make realistic assumptions about $p_B(i)$ and $p_L(l)$. The most general assumption for $p_B(i)$ is to assume that pins are **uniformly** distributed among the w slots. Hence, for N wires, the probability of a wire being born at i is independent of i and given by

$$p_B(i) = p_B = \frac{N}{w}$$

The choice of $p_L(l)$ has been discussed in the literature on wiring space estimation [Sas85] [Hel77] and many distributions were suggested, such as the Poisson, exponential and geometric distributions. In his thesis, Sastry [Sas85] proved that the exponential distribution is the 'ideal' distribution in the limiting case of optimal placements in the continuous domain. Bearing in mind the fact that the geometric distribution is the discrete counterpart of the exponential distribution, we assume in this model that $p_L(l)$ is *geometric*. So,

$$p_L(l) = pq^{l-1}$$

where $\frac{1}{p}$ = average wire length and $q = 1 - p$.

Given these two forms of $p_B(i)$ and $p_L(l)$, it is possible to obtain a closed form for $E\{d(x)\}$.

$$\mathrm{E}\{d(x)\} = \frac{N}{wpq}(1-q^x)(1-q^{w-x+1}) \tag{2.3.3}$$

Now, in order to find the maximum of $E\{d(x)\}$, we set

$$\frac{d\{\mathrm{E}\{d(x)\}\}}{dx} = 0$$

and solve for the root(s), so

$$\frac{N}{wpq}\left\{-q^x \log q(1-q^{w-x+1}) + q^{w-x+1}\log q(1-q^x)\right\} = 0$$

or,

$$-q^x + q^{w-x+1} = 0$$

So the maximum local density occurs at $x_{max} = \lceil\frac{w+1}{2}\rceil$ or $x_{max} = \lfloor\frac{w+1}{2}\rfloor$, and the estimate for the track requirements is $\mathrm{E}\{d(x_{max})\}$. [1]

[1] Even though the function differentiated is a *continuous* function of x, it can be easily shown that the function defined over the discrete subset of its domain is indeed maximum at x_{max}.

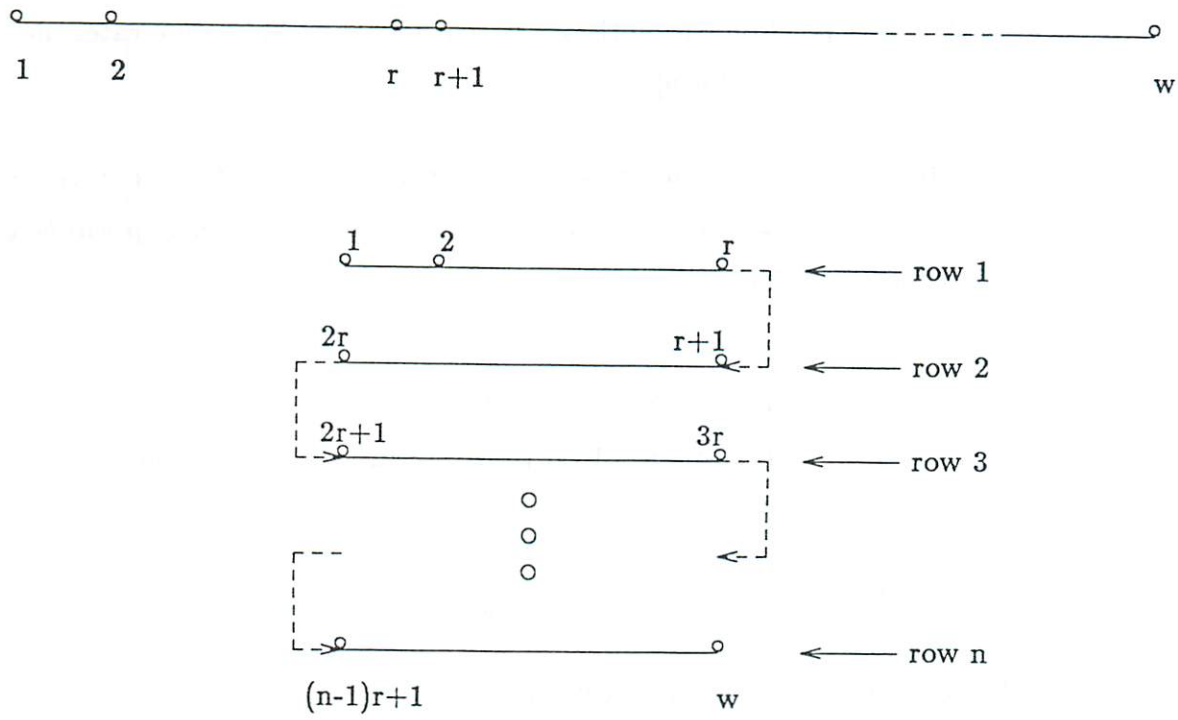


Figure 2.6: Folding of a row

2.3.5 The multiple row model

Before we extend the model in the previous section to the multiple row case, we should remind ourselves of the assumption that the placement of standard cells is done by first placing the cells in one single row so the average wire length is made as small as possible. The linear placement is then folded into n rows. The characteristics of the single row are discussed in the previous section. In this section, we focus on modeling the folding process, its effects on wiring, and how to estimate the wiring area after folding.

2.3.5.1 Folding

Once a placement is done on a single row, the folding is performed as shown in Fig. 2.6. The initial row is 'snaked' around itself into n rows. We start with a single row linear placement and consider a pin slot p on it. Now we fold the linear placement into n rows, each of width $r = \frac{w}{n}$. Let $y(p), 1 \leq y(p) \leq n$ denote the row in which the pin slot p will be located, and let $x(p), 1 \leq x(p) \leq r$

be its new x position within that row. The following lemma indicates the relation between $p, x(p)$, and $y(p)$:

Lemma 2.1 *Consider a pin slot p in a single row block of width w. After folding the row into n rows, each of width* $r = \frac{w}{n}$, *the new position of p will be given by* $x(p), y(p)$, *such that*

$$x(p) = \begin{cases} p - 2kr & \text{if } (p \, div \, r) = 2k \quad \text{odd rows} \\ 2kr - p + 1 & \text{if } (p \, div \, r) = 2k - 1 \quad \text{even rows} \end{cases} \tag{2.3.4}$$

$$y(p) = (pdivr) + 1 \tag{2.3.5}$$

where (pdivr) *denotes the integer division of p by r.*

Proof: The proof can be directly inferred from Fig. 2.6. □

As an example of Lemma 2.1, take $p = r + 5$, and assume $r > 5$. The new position of the pin slot will be in the second row ($(r + 5)$ div $r + 1 = 2$) and its position in that row is $2r - (r + 5) + 1 = r - 4$.

Since the cells are of the double entry type, wires can be routed from either the top or the bottom equivalent pins for any cell. When folding is performed, the wires must be re-routed and, in general, do not 'snake around' with the cell rows. Hence folding refers only to the placement of the cells and the new folded configuration of cells requires a new pass by the router.

2.3.5.2 Estimating channel density

We use the effect of folding on pin locations to estimate the wiring space after the folding of the one-row placement modelled in Section 2.3.4. We do so by predicting the channel densities of the placement. The **local channel density** at x in an n row block, $d_n(x)$, is defined as the total number of wires crossing a vertical cutline that passes through x, $1 \leq x \leq r$, and runs through all n rows of the block. Fig. 2.7 illustrates a cutline at x. Clearly, a wire will cross that cutline once iff it starts to the left (right) of x and terminates to the right (left)

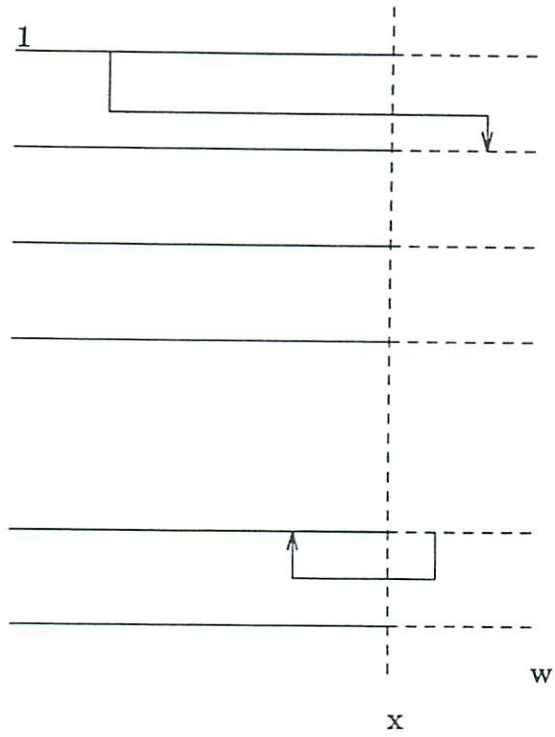


Figure 2.7: Cutline at x

of x. This is so because wires are assumed to follow minimal rectilinear paths with no backward moves.

Note that $d_n(x)$ does not change with the route a wire takes from source to destination. In Fig. 2.7, for example, a wire from row 1 to row 4 can either go vertically down to row 4 then horizontally, or alternatively horizontally through row 1 then down to row 4, or even 'zigzag' down through rows 1,2,3 and 4. In every case, that wire will contribute one track to the local density at x. In other words, *the local density is only dependent on the placement of the components and is independent of the routing*, assuming minimal rectilinear paths of wires.

Since linear placement has been fixed prior to folding, the aim is to predict the effects of folding on the linear placement and from that, the channel density. In Fig. 2.8, we describe the relation between the folded placement and the corresponding linear placement as follows :

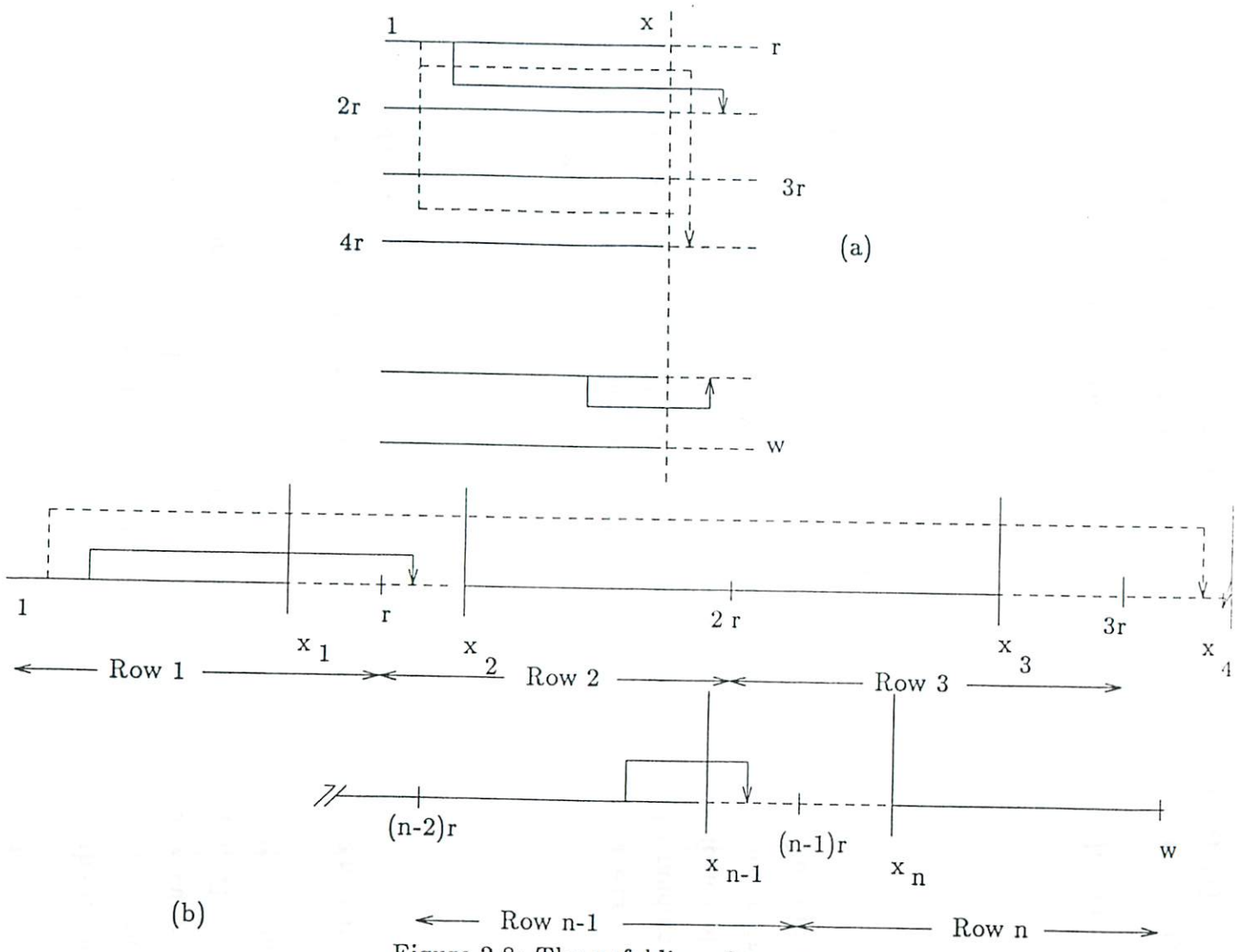


Figure 2.8: The unfolding of a row

- Part (a) of Fig. 2.8 shows the folded block and a cutline at x. Regions to the right of x are cross-hatched, those to the left are solid.

- Part (b) of Fig. 2.8 shows the unfolded row. Note that the left and right regions in (a) are now alternating in (b). Also note that, since the cutline at x crosses all the rows in (a), it will translate into n cutlines whose combined densities represent the local density at x, $d_n(x)$. The location of the n resulting cutlines, $x_1, x_2, \ldots, x_n$ is given by the following lemma.

Lemma 2.2 *A cutline at x in an n row block will correspond to n cutlines in the unfolded row, located at $x_1, x_2, \ldots, x_n$, given by*

$$x_i = \begin{cases} 2kr - x + 1 & i = 2k \\ 2kr + x & i = 2k+1 \end{cases} \tag{2.3.6}$$

Proof: This can be proved directly from Lemma 2.1. □

Having described the effects of folding on a linear placement, we look next at estimating the local density at x. Let $W_n(x)$ be the random variable denoting the number of wires crossing a cutline at x in the folded placement. In order to specify which wires will cross the cutline (and hence contribute to $W_n(x)$), let's define the intervals $I_1(x), I_2(x), \ldots, I_n(x), I_w(x)$ on the unfolded row, given by

$$\begin{aligned} I_i(x) &= [x_{i-1}, x_i], i = 2, \ldots, n \\ I_1(x) &= [1, x], \\ I_w(x) &= [x_n, w] \end{aligned}$$

The following theorem characterizes the wires crossing the cutline at x.

Theorem 2.1 *A wire born in $I_i(x)$ will contribute one to the cut density at x iff it terminates in $S_i(x)$, where $S_i(x) = \{I_{i+1}(x), I_{i+3}(x), \ldots, I_{i+2k+1}(x), \ldots\}$.*

Proof: The theorem can be proven by noting that the intervals in $S_i(x)$ correspond, after folding, to the regions on the other side of cutline x. In other words,

if $I_i(x)$ corresponds to a region on the left (right) of x, then the intervals in $S_i(x)$ correspond to the regions on the right (left) of x. □

By Theorem 2.1, the total number of wires crossing the cutline at x is the sum of all the wires starting in $I_1(x), I_2(x), \cdots, I_n(x)$, and terminating in $S_1(x), S_2(x), \cdots, S_n(x)$, respectively. Let $\mathrm{WI}_i(x)$ be the random variable denoting the number of wires starting in $I_i(x)$ and terminating in $S_i(x)$. $\mathrm{W}_n(x)$ can now be written as

$$\mathrm{W}_n(x) = \sum_{i=1}^{n} \mathrm{WI}_i(x) \tag{2.3.7}$$

$\mathrm{WI}_i(x)$ represents the *contribution* of the wires starting in interval $I_i(x)$ to the local density at x. Clearly, $\mathrm{WI}_i(x)$ is the sum of the wires starting in $I_i(x)$ and terminating in each of the intervals $I_j(x)$ such that $I_j(x) \in S_i(x)$. Now let $\mathrm{WI}_{i,j}(x)$ be the random variable denoting the number of wires born in an interval $I_i(x)$ and terminating in $I_j(x)$, where $i < j$, then $\mathrm{WI}_i(x)$ can be written as

$$\mathrm{WI}_i(x) = \sum_{j | I_j \in S_i(x)} \mathrm{WI}_{i,j}(x) \tag{2.3.8}$$

We must next find an estimate for $\mathrm{WI}_{i,j}(x)$. To do so, let's take a wire born at point t inside $I_i(x)$, having a length L [Fig. 2.9]. This wire will terminate in $I_j(x)$ if its length is such that $x_{j-1} - t \leq L \leq x_j - t$. This corresponds to the event $C(t, j, x)$

$$C(t,j,x) = \{\ x_{j-1} - t \leq L \leq x_j - t \quad \text{given that the wire of length } L \text{ is born at } t\}$$

which has a probability

$$\Pr\{C(t,j,x)\} = \sum_{m=x_{j-1}-t}^{x_j-t} p_L(m)$$

The average value of $\mathrm{WI}_{i,j}(x)$ is then given by

$$\begin{aligned} E\{\mathrm{WI}_{i,j}(x)\} &= \sum_{t=x_{i-1}}^{x_i} p_B(t)\ \Pr\{C(t,j,x)\} \\ &= \sum_{t=x_{i-1}}^{x_i} p_B(t) \sum_{m=x_{j-1}-t}^{x_j-t} p_L(m) \end{aligned} \tag{2.3.9}$$

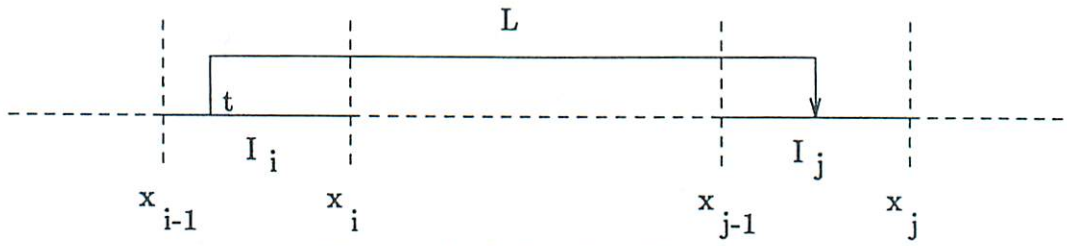


Figure 2.9: Estimating $\mathrm{WI}_{i,j}(x)$

Let's use $\mathrm{E}\{\mathrm{WI}_{i,j}(x)\}$ as an estimate of $\mathrm{WI}_{i,j}(x)$. We can then write the average contribution of $I_i(x)$ to $\mathrm{W}_n(x)$ as the expected value of $\mathrm{WI}_i(x)$, given by

$$\mathrm{E}\{\mathrm{WI}_i(x)\} = \sum_{j|I_j \in S_i(x)} \sum_{t=x_{i-1}}^{x_i} p_B(t) \sum_{m=x_{j-1}-t}^{x_j-t} p_L(m) \qquad (2.3.10)$$

We classify the WI_i's into two groups, $\mathrm{WI}_{2k}(x)$ and $\mathrm{WI}_{2k+1}(x)$, depending on i being even or odd (The contributions of the intervals at the edges, $I_1(x)$ and $I_w(x)$ are treated separately). This is necessary because $I_{2k}(x)$ and $I_{2k+1}(x)$ (and also their corresponding S_i's) have different expressions for the interval bounds (and hence different summation limits). So for an *even* interval

$$I_{2k}(x) = [x_{2k-1}, x_{2k}] = [2(k-1)r + x, 2kr - x + 1]$$

and for an *odd* interval

$$I_{2k+1}(x) = [x_{2k}, x_{2k+1}] = [2kr - x + 1, 2kr + x]$$

So now we can write the average value of the contributions of $\mathrm{WI}_{2k}(x)$ and $\mathrm{WI}_{2k+1}(x)$ as follows :

$$\mathrm{E}\{\mathrm{WI}_{2k}(x)\} = \begin{cases} \sum_{l=k}^{\frac{n}{2}-1} \mathrm{E}\{\mathrm{WI}_{2k,2l+1}(x)\} & n \text{ even} \\ \sum_{l=k}^{\frac{n-1}{2}} \mathrm{E}\{\mathrm{WI}_{2k,2l+1}(x)\} & n \text{ odd} \end{cases} \qquad (2.3.11)$$

and,

$$\mathrm{E}\{\mathrm{WI}_{2k+1}(x)\} = \begin{cases} \sum_{l=k+1}^{\frac{n}{2}} \mathrm{E}\{\mathrm{WI}_{2k+1,2l}(x)\} & n \text{ even} \\ \sum_{l=k+1}^{\frac{n-1}{2}} \mathrm{E}\{\mathrm{WI}_{2k+1,2l}(x)\} & n \text{ odd} \end{cases} \tag{2.3.12}$$

From these equations we can get the average total contribution of the odd and even intervals as follows :

$$\mathrm{E}\{\mathrm{WI}_{even}(x)\} = \begin{cases} \sum_{k=1}^{\frac{n}{2}-1} \mathrm{E}\{\mathrm{WI}_{2k}(x)\} & n \text{ even} \\ \sum_{k=1}^{\frac{n-1}{2}} \mathrm{E}\{\mathrm{WI}_{2k}(x)\} & n \text{ odd} \end{cases} \tag{2.3.13}$$

and that of the odd intervals

$$\mathrm{E}\{\mathrm{WI}_{odd}(x)\} = \begin{cases} \sum_{k=1}^{\frac{n}{2}-1} \mathrm{E}\{\mathrm{WI}_{2k+1}(x)\} & n \text{ even} \\ \sum_{k=1}^{\frac{n-3}{2}} \mathrm{E}\{\mathrm{WI}_{2k+1}(x)\} & n \text{ odd} \end{cases} \tag{2.3.14}$$

We must also consider the effects of the first and last intervals, namely,

$$I_1(x) = [1, x_1 = x]$$

and

$$I_w(x) = [x_n, w]$$

For $I_1(x)$,

$$\mathrm{E}\{\mathrm{WI}_{1,2k}(x)\} = \sum_{t=1}^{x} p_B(t) \sum_{m=x_{2k-1}-t}^{x_{2k}-t} p_L(m)$$

and the total contribution of $I_1(x)$ is

$$\mathrm{E}\{\mathrm{WI}_1(x)\} = \begin{cases} \sum_{k=1}^{\frac{n}{2}} \mathrm{E}\{\mathrm{WI}_{1,2k}(x)\} & n \text{ even} \\ \sum_{k=1}^{\frac{n-1}{2}} \mathrm{E}\{\mathrm{WI}_{1,2k}(x)\} & n \text{ odd} \end{cases} \tag{2.3.15}$$

Finally, for the last interval, $I_w(x)$, two cases appear

(1) n even,

$$\mathrm{E}\{\mathrm{WI}_{2k,w}(x)\} = \sum_{t=x_{2k-1}}^{x_{2k}} p_B(t) \sum_{m=x_n-t}^{w-t} p_L(m)$$

and the contribution of $I_w(x)$ is

$$\mathrm{E}\{\mathrm{WI}_w(x)\} = \sum_{k=1}^{\frac{n}{2}} \mathrm{E}\{\mathrm{WI}_{2k,w}(x)\} \tag{2.3.16}$$

(2) n odd,

$$\mathrm{E}\{\mathrm{WI}_{2k+1,w}(x)\} = \sum_{t=x_{2k}}^{x_{2k+1}} p_B(t) \sum_{m=x_n-t}^{w-t} p_L(m)$$

and the contribution of $I_w(x)$ is

$$\begin{aligned} \mathrm{E}\{\mathrm{WI}_w(x)\} &= \sum_{k=1}^{\frac{n-1}{2}} \mathrm{E}\{\mathrm{WI}_{2k+1,w}(x)\} + \mathrm{E}\{\mathrm{WI}_{1,w}(x)\} \\ &= \sum_{k=1}^{\frac{n-1}{2}} \mathrm{E}\{\mathrm{WI}_{2k+1,w}(x)\} + \sum_{t=1}^{x} p_B(t) \sum_{m=x_n-t}^{w-t} p_L(m) \end{aligned} \tag{2.3.17}$$

Finally the average value of $\mathrm{W}_n(x)$ can be written as

$$E\{W_n(x)\} = E\{WI_{even}(x)\} + E\{WI_{odd}(x)\} + E\{WI_w(x)\} + E\{WI_1(x)\} \tag{2.3.18}$$

Equation 2.3.18 gives an average value of $\mathrm{W}_n(x)$ as a function of $p_B(t)$ and $p_L(m)$. In order for the estimate to be of practical value, it is important to know these two distributions. As in the previous section, we will assume that

(a) pins are uniformly distributed along cell rows, so

$$p_B(t) = \frac{N}{w}$$

(b) wire lengths are geometrically distributed, so

$$p_L(m) = pq^{m-1}$$

where $\frac{1}{p}$ is the average wire length and $q = 1 - p$.

Under these assumptions, it is possible to find closed form expressions for $E\{W_{even}(x)\}$, $E\{W_{odd}(x)\}$, $E\{W_1(x)\}$ and $E\{W_w(x)\}$. Hence $E\{W_n(x)\}$ can be expressed as

$$E\{W_n(x)\} = \frac{N}{w} \times \frac{1}{pq(1-q^{2r})} \left\{ 2(1-q^w)(1-q^{2r-2x+2})(1-q^x) + (1-q^{2x})(1-q^{2r-2x+2}) \left[n-2-2q^{2r}\frac{1-q^{w-2r}}{1-q^{2r}} \right] \right\} \tag{2.3.19}$$

for n even, and

$$E\{W_n(x)\} = \frac{N}{w} \times \frac{1}{pq(1-q^{2r})} \left\{ 2(1-q^{w-r})(1-q^{r-x+1})(1-q^x)(2+q^x+q^{r-x+1}) + (1-q^{2x})(1-q^{2r-2x+2}) \left[n-2-2q^{2r}\frac{2-q^{w-r}-q^{w-3r}}{1-q^{2r}} \right] \right\} \tag{2.3.20}$$

for n odd, $n \geq 3$.

Equations 2.3.19 and 2.3.20 give an estimate for $d_n(x)$, the local density of a cutline at x. One must, however, allocate at least a number of tracks equal to the *maximum* density across any cutline parallel to x, or CH_n. As in the single row case, the track requirement is *approximated* by

$$\max_{1 \leq x \leq r} \{E\{W_n(x)\}\}$$

Note that, as in the single row model, this represents a lower bound on the actual track requirements.

Another approximation is also introduced by the fact that we are estimating, at each x, the *sum of the local densities* of the individual routing channels instead of looking at each channel separately from the others, as shown in Fig. 2.10. Suppose that $C_1(x), C_2(x), \ldots, C_n(x)$, are the local densities at x of channels $1, 2, \ldots, n$, respectively, then

$$d_n(x) = \sum_{i=1}^{n} C_i(x)$$

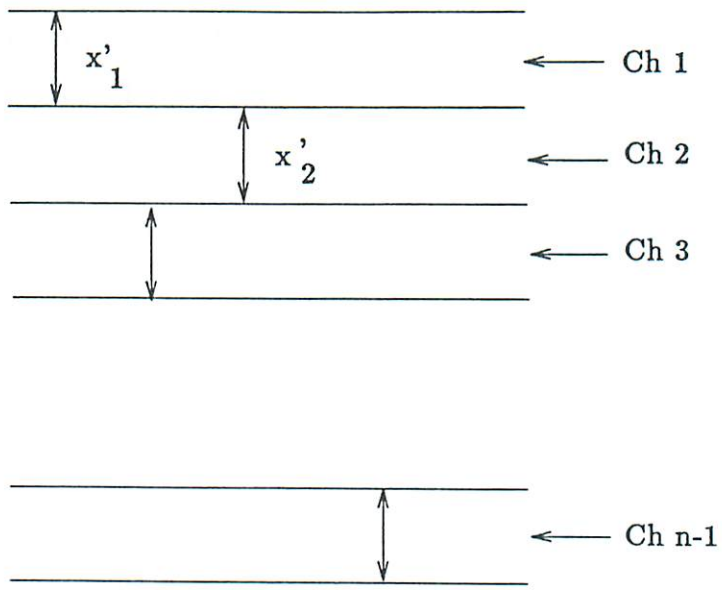


Figure 2.10: Maximum density calculation

and

$$\mathrm{CH}_n = \max_{1 \le x \le r} \{d_n(x)\} = \max_{1 \le x \le r} \left\{ \sum_{i=1}^{n} C_i(x) \right\} \tag{2.3.21}$$

This is not, in fact, the actual track requirement estimate since the functions $C_1(x), C_2(x), \ldots, C_n(x)$ may be maximum at different values of x, say, $x'_1, x'_2, \ldots, x'_n$, respectively and the actual track requirement will be

$$\mathrm{CH}'_n = \sum_{i=1}^{n} C_i(x'_i) = \sum_{i=1}^{n} \max_{1 \le x \le r} \{C_i(x)\}. \tag{2.3.22}$$

The following theorem establishes a relation between CH_n and CH'_n.

Theorem 2.2 $\mathrm{CH}_n \le \mathrm{CH}'_n$ *with equality iff* $\forall i, x'_i = x'_0$, where x'_0 *is the point for which* $d_n(x'_0) = \mathrm{CH}_n$

Proof: The expression for CH_n in 2.3.21 can be re-written, assuming that maximum channel density occurs at x'_0, as

$$\mathrm{CH}_n = \sum_{i=1}^{n} \mathrm{C}_i(x'_0) \tag{2.3.23}$$

Note that each C_i is maximum at x'_i, hence

$$\forall i,\ C_i(x'_0) \leq C_i(x'_i)$$

and the inequality between Equations 2.3.23 and 2.3.22 follows. □

This approximation is, however, not a severe one due to two reasons :

1. The channel densities are, in general maximum near the centers of the channels. Hence the points $x'_1, x'_2, \ldots, x'_n$ are, in practice, close to each other.

2. We have observed that local densities do not, in general, vary a lot around maximum density points.

Under these considerations, it is safe to assume, in general, that $x'_1, x'_2, \ldots, x'_n$ occur in the neighborhood of x'_0 and that $CH_n \approx CH'_n$.

As in the single row case, estimating x'_0 is done by setting the derivative of $E\{W_n(x)\}$ to zero and solving for x. It may not be possible, however, to obtain an analytical solution for x'_0, in which case numerical methods must be used to find an approximate solution. In the particular case of Equations 2.3.19 and 2.3.20, the derivative of $E\{W_n(x)\}$ is a fourth degree polynomial in q^x whose root(s) can be found numerically using the Newton-Raphson or any similar method.

2.3.6 Estimating the feedthroughs

Now that we have estimated the wiring space requirements in the vertical direction (i.e. the channel densities), the next step is to estimate the amount of space needed in the horizontal dimension. We make the assumption that vertical wires which run in more than one row are routed using feedthroughs [Fig. 2.11]. Estimating the space required by the vertical wires requires estimating the number of feedthroughs in the cell rows. Feedthroughs are used to route wires that connect cells which are placed more than one row apart. They have the advantage of avoiding the routing of wires 'around' the cell rows, thereby avoiding the need for possible extra tracks. Feedthrough cells must, however, be

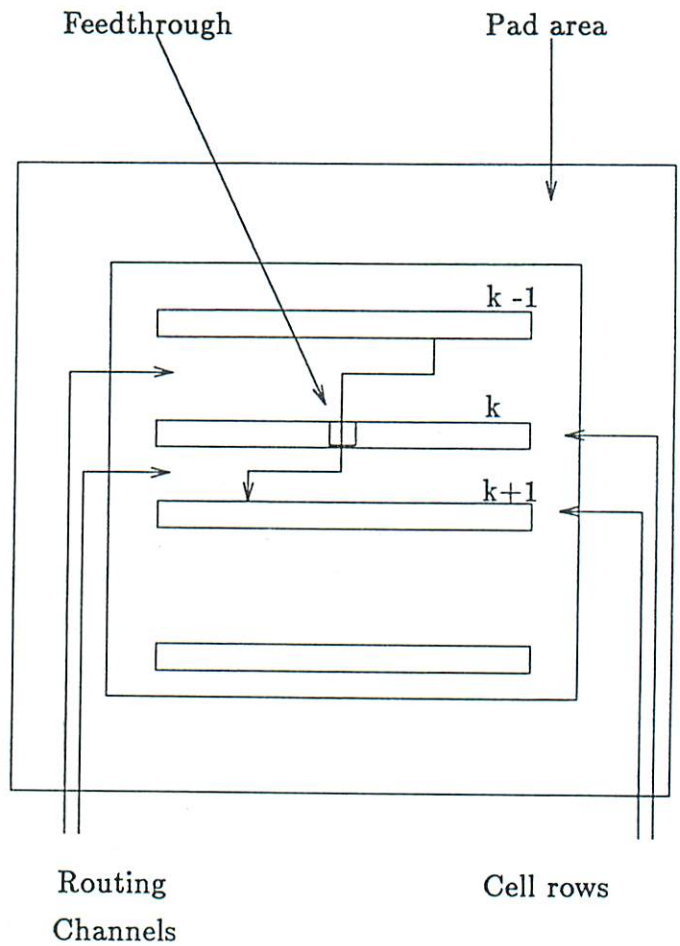


Figure 2.11: Feedthroughs in a chip

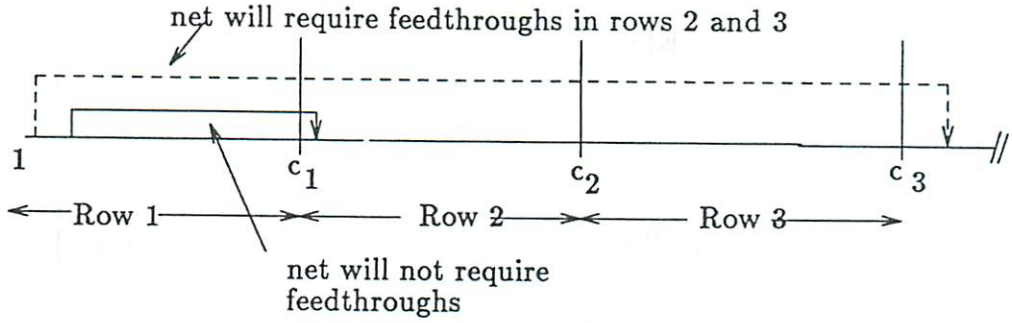


Figure 2.12: Unfolded row

inserted in cell rows[2], hence the necessity of adjusting the cell placement during the routing phase. Feedthroughs also increase the chip width since, by inserting them in cell rows, they increase the sizes of these rows. So it is necessary to have an estimate for the increase in row sizes due to feedthroughs for an area estimate of the chip to be accurate.

The model described in the previous section can be used to estimate the number of feedthroughs in cell rows. Let FT_i be the random variable denoting the number of feedthroughs in row i. A wire will need a feedthrough if it connects cells which are more than one cell row apart. For example, in Fig. 2.11, a wire born in row *k-1* and dead in row *k+1* will require a feedthrough in row k. Conversely, if we look at row k , all wires born in row *1* or *2* or ... *k-1*, *and* dead in rows *k+1* or *k+2* or ...or n will require one feedthrough each in row k. Hence, the task of estimating the number of feedthroughs in row k is equivalent to estimating the number of such wires.

Let's look again at the unfolded single row placement shown in Fig. 2.12. Let $c_1, c_2, \ldots, c_{n-1}$ be the pin slots at which the linear row is folded. Since the resulting cell rows are assumed to be of equal sizes, then $c_1 = r, c_2 = 2r, \ldots, c_{n-1} = (n-1)r$ and a row k can be looked at as an interval $[c_{k-1}+1, c_k]$.

Now let $WR_{i,j}$ be the random variable denoting the number of wires born in row i and dead in row j. As in the previous section, we can express $E\{WR_{i,j}\}$ as

[2]This is the only type of feedthrough allowed in the model.

$$\mathrm{E}\{\mathrm{WR}_{i,j}\} = \sum_{t=c_{i-1}+1}^{c_i} p_B(t) \sum_{m=c_{j-1}+1-t}^{c_j-t} p_L(m)$$

or,

$$\mathrm{E}\{\mathrm{WR}_{i,j}\} = \sum_{t=(i-1)r+1}^{ir} p_B(t) \sum_{m=(j-1)r+1-t}^{jr-t} p_L(m) \qquad (2.3.24)$$

and the estimate for the number of feedthroughs in row k is equal to the average number of wires born in rows 1 or 2 ... *or* $k-1$ and dead in *k+1* or *k+2* ... or n, given by

$$\mathrm{E}\{\mathrm{FT}_k\} = \sum_{i=1}^{k-1} \sum_{j=k+1}^{n} \mathrm{E}\{\mathrm{WR}_{i,j}\} \quad k = 2, \ldots, n-1 \qquad (2.3.25)$$

The chip width (without I/O) must be at least as wide as the widest cell row, since the rows are assumed to be of equal sizes initially. The widest cell row is the one that has the maximum number of feedthroughs, so we can write the width of the widest row as

$$r + \max_{1 \le k \le n} \{\mathrm{E}\{\mathrm{FT}_k\}\} \qquad (2.3.26)$$

Now if we assume, as in the previous sections, that the pins are *uniformly* distributed along slots, and wire lengths are *geometrically* distributed, it is possible to obtain a closed form expression for $\mathrm{E}\{\mathrm{FT}_k\}$:

$$\mathrm{E}\{\mathrm{FT}_k\} = \frac{N}{w}\frac{q^r}{p}\left(1 - q^{(n-k)r}\right)\left(1 - q^{(k-1)r}\right) \qquad (2.3.27)$$

Setting

$$\frac{\mathrm{d}E\{\mathrm{FT}_k\}}{\mathrm{d}k} = 0$$

and solving for k, the maximum number of feedthroughs occurs in row k_{max}, where

$$k_{max} = \lceil \frac{n+1}{2} \rceil$$

Using Equation 2.3.26, we can find the width of the widest row.

2.4 Summary

This chapter presented a probabilistic model for estimating the values of two important area parameters in a chip. The model attempts to characterize placement of cells on a chip and the behavior of interconnections between the cells. In the next chapter, we validate the model and in Chapter 4 explore the various characteristics of placement and wiring in standard cell chips.

Chapter 3

Verifying the area estimation model

There was things which he stretched,
but mainly he told the truth.
– MARK TWAIN, *Huckleberry Finn* (1884)

3.1 Introduction

In Chapter 2, we presented a model for estimating the area of a chip, given its gate level description. This chapter aims at verifying and validating the model. The first step in this process consists of verifying the derivations of the model by simulating chips where wires and cells behave as assumed by the model and comparing the resulting estimates to the predictions of the formulae. The next step is to verify the initial model assumptions by comparing the estimates to 'real world' chips in order to verify their accuracy.

3.2 Simulation

It is important to verify the correctness (and accuracy) of the derivation of the proposed formulas for chip area parameters. A first step in this validation is simulation of real chip layouts.

3.2.1 Description

The inputs to the simulation program are the total cell width, w, the number of nets, N, the number of rows, n, and the average wire length. The program then simulates a single row of width w. N simulated wires are born from N randomly selected pins, and have lengths drawn from a geometric distribution with parameter equal to the inverse of the average wire length. The single row is then folded into n rows and the local densities are computed. This process is repeated for s samples and the average densities are computed. Two quantities are also computed, $AvgMax$, which is the average value of the maximum density in each sample over all samples, and $MaxAvg$, which is the maximum value of the average densities. More formally, if $d_i(j)$ is the local density at pin slot j in the i^{th} sample and r is the row width, then

$$AvgMax = \frac{1}{s}\sum_{i=1}^{s} \max_{1\le j\le r} d_i(j) \tag{3.2.1}$$

$$MaxAvg = \max_{1\le j\le r}\left\{\frac{1}{s}\sum_{i=1}^{s} d_i(j)\right\} \tag{3.2.2}$$

The number of samples, s, was set to 100. Table 3.1 shows some results of these simulations. The track estimates obtained using Equations 2.3.19 and 2.3.20 are in the last column. The input parameters in the first four columns were chosen as 'reasonable' values compared to real life layouts. Some of the criteria for the selection of the data points are the following :

- The average wire lengths were chosen as 'reasonable' estimates, based on some values from the real data described in the next section.
- In each case, the number of rows was chosen so as to make the chip, according to the estimates, as square as possible (which is the criterion for selecting the number of rows in the real world.)

w	n	N	1/p	AvgMax	MaxAvg	Est
500	8	150	30	60.6	56.7	55.7
600	8	200	32	77.0	72.6	69.8
800	8	300	35	101.4	95.6	99.0
1000	9	400	38	134.3	127.8	122.5
1500	10	600	50	190.3	182.6	179.3
2000	10	800	65	248.1	238.2	233.0

Table 3.1: Simulation results

3.2.2 Observations

The following was observed :

- In all cases, *AvgMax* is within 6% of *MaxAvg*, so the approximation of $E\{\max_x d(x)\}$ with $\max_x E\{d(x)\}$ seems to be a valid one.
- Our estimation formulas for tracks requirements are, in most cases, more optimistic than both *AvgMax* and *MaxAvg*. This may be due to the fact that we neglected, in our estimates, the wires that 'spill over' the edge at w. The difference between the estimate and the actual value (*MaxAvg*) is, however, within 4%.

3.3 Real data validation and implementation

Once the formula derivations have been validated, we applied the model to real test chips, using the PLEST program. The next logical step in validating our probabilistic model of Chapter 2 is to apply PLEST to 'real world' cases and compare the model predictions to the actual values in these cases.

3.3.1 PLEST

PLEST is a software package which performs random logic area estimation. The basis for PLEST is the probabilistic model presented in Chapter 2 for estimating the area of random logic blocks, given a cell level description

Chip num	Width w	Rows n	2-point nets N	Avg wire Len 1/p
1	533	7	161	31.0
2	648	7	242	31.7
3	670	8	248	24.0
4	788	9	307	24.2
5	783	9	307	24.9
6	949	10	382	21.0

Table 3.2: Chip characteristics

in which cells are assumed to be laid out in the standard cell layout method. PLEST's chip estimation parameters are currently 'tuned' for the MP2D chip layout system parameters.[1] The present implementation of PLEST is written in Berkeley Pascal on a VAX-11/750 running the 4.2 bsd UNIX operating system. It prompts the user for design parameters, such as total cell width, number of nets, and average wire length. For a given row configuration, the program estimates the number of tracks needed for routing by using Equations 2.3.19 or 2.3.20 at the point of maximum density. The number of feedthroughs needed to wire multi-row nets is then estimated using Equation 2.3.6. Using these estimates, PLEST produces an estimate for the total chip area. This process is repeated for a range of possible row configurations specified by the user, the idea being for the designer to choose the aspect ratio that best fits his/her purposes. The program also produces plots of different estimates for chip parameters such as number of rows, height, width, aspect ratio, and area (hence the name PLotting ESTimator). Figure 3.1 shows an example run of the program. It is interesting to note, for this particular run, that the width estimates for the chip for 14 and 15 rows configurations are the same, one explanation for this is that the width reduction due to folding in 15 versus 14 rows is compensated for by an additional number of feedthroughs that must be inserted in the cell rows to accomodate an increase in the number of wire travelling vertically between the rows.

[1] MP2D will be detailed in Section 3.3.4.

```
poisson-figs{1} plest
do you want a plot? [y] y
enter plot file name: chip1.plt
enter x plot variable [rows, height, width, aspect, area]: rows
enter y plot variable [rows, height, width, aspect, area]: area
enter min number of rows: 1
enter max number of rows: 15
enter total width [in pin pitches]: 533
enter number of nets [two point]: 161
enter avg wire length [in pin pitches]: 31.0
```

# rows	tracks	height	width	aspect-ratio	area
1	10	43	511	0.08	21657
2	19	54	271	0.20	14557
3	28	65	191	0.34	12328
4	36	75	151	0.50	11386
5	43	85	129	0.66	10901
6	49	94	115	0.82	10716
7	54	103	105	0.98	10686
8	58	111	98	1.13	10766
9	62	119	92	1.29	10935
10	64	125	88	1.42	11149
11	67	133	85	1.56	11431
12	69	140	84	1.67	11723
13	71	146	82	1.78	12076
14	72	153	81	1.89	12413
15	73	159	81	1.96	12822

Figure 3.1: Example run of PLEST

3.3.2 Test cases

Our data consisted of a set of six different register-transfer (RT) level implementations of the same dataflow specification. The RT level designs were automatically synthesized using mixed integer linear programming techniques [HP83]. By varying the performance requirements (reflected by varying the cost function), it was possible to obtain six different RT level implementations, reflecting the cost-delay tradeoffs. In a previous experiment [GP82], Granacki and Parker constructed actual layouts of the designs using the standard cell design methodology and the **MP2D** layout system [FN78]. The overall characteristics of the designs are summarized in Table 3.2.

In order to evaluate the predictions of the proposed model and compare them to the actual values, we applied PLEST to the set of six designs described above. For a given row configuration of a design, the program uses the channel density estimation Formulae 2.3.19 and 2.3.20 to generate estimates of the number of tracks needed for routing by setting its derivative to zero, and solves for the point of maximum average density using the Newton-Raphson method. The average density at that point is taken as the number of tracks needed for routing. It then uses the feedthrough estimation formula 2.3.26 to estimate the maximum number of feedthroughs in the cell rows, which occurs at the center rows. The width of the center rows plus the feedthroughs is taken as the width of the standard cell block (i.e. chip width minus frame width).

3.3.3 Estimation results

PLEST was applied to the set of six chip layouts. Table 3.3 shows the estimation results, along with the real values of the parameters as extracted from the layouts and the estimation time quoted in seconds. The area figures are in square mils.

Tracks (Actual)	Tracks (Est)	Area (Actual)	Area (Est)	%err (Area)	Time (Secs)
63	54	10593	10686	.8	0.9
65	73	12584	13388	6.4	1.1
64	68	12535	12533	.0	1.1
92	82	14375	14469	.7	1.4
89	85	14625	14571	.4	1.4
96	93	16864	16161	4.8	1.1

Table 3.3: Estimation results

3.3.4 Using MP2D to generate more examples

In order to further validate and verify our model, it was necessary to produce more example layouts. After several attempts at analyzing layouts generated elsewhere, we concluded that the best approach would be to generate the examples locally. To do so in a relatively short amount of time, we had to have access to an automated layout system. Negotiations with RCA concluded in the company's agreement to provide the MP2D automated layout system for the use of the design automation and testing groups at USC. Initially, the software was running under the VAX VMS operating system; later, it was ported to run under UNIX. The port is now near completion. The structure of the MP2D layout system is shown in Figure 3.2. A circuit is described using components from a CMOS 3μ cell library provided with the program, along with standard and user-defined macros. While MP2D assumes as input a *flat* description of the design, a pre-processing module accepts a hierarchical description and flattens it. MP2D allows the designer to specify several dozens of parameters controlling both the process and the layout, (global and detailed,) such as the number of rows, location of pads and the like. The user can also specify partially placed components and indicate critical paths. Coupled with MP2D is a pre-processing module, DPLAC, which partitions the circuit and pre-places parts of it with the aim of further optimizing MP2D's outputs and execution time. Globally, there are two major modes with which MP2D can be run: with or without using DPLAC (DOMAIN PARTITIONING AND PLACEMENT). If DPLAC is used,

more compact layouts would, in general, be generated. For an average design of around 500 gates, MP2D is expected to run around 5-6 hours on a 'free' CPU of the VAX-11/750 class. Running DPLAC would roughly take around half an hour but the MP2D runtime would be reduced by about 20%. The runtime increase with circuit size is faster than linear. Circuit designers at RCA mentioned that their designs of 2000+ gates could take MP2D more than 20-30 hours of CPU time to generate the layouts.

The MP2D program itself runs in four phases: input, placement, routing and artwork. The first phase initializes the data base and checks for consistency of inputs. The placement phase has three different placement algorithms running in sequence. Routing is then performed first on the center (channel) section of the chip, then on the sides. Once routing is done, MP2D extracts many different statistics on the generated layout, routes the power line and produces an artwork plot of the chip, along with a PG (Pattern Generator) description of the chip.

3.3.5 More results

To further validate and test PLEST, MP2D was used to generate six more chip layouts. Resulting layouts were analyzed. Layout parameters were extracted from the layout and fed into PLEST. PLEST output was then compared to the actual values from the layouts. Table 3.4 describes the circuit of each one of the six example chips. Table 3.5 depicts the layout parameters of each chip. Table 3.6 shows the estimation results versus the actual values. These results offer further validation for PLEST. The area figures are now in square millimeters since the new version of MP2D has converted all measurements into the metric system. For chips 5 and 6, the chip area was *I/O bound.* This means that there were too many pads around the chips to fit right next to the cell area (or active area as it is called in MP2D). Figure 3.3 depicts such a situation. For such chips, it is enough to know the number and size of the I/O pads in order to get an almost exact estimate of the chip area. In this case, we decided to compare the *active* areas of the chips versus the estimated ones, since this is what PLEST

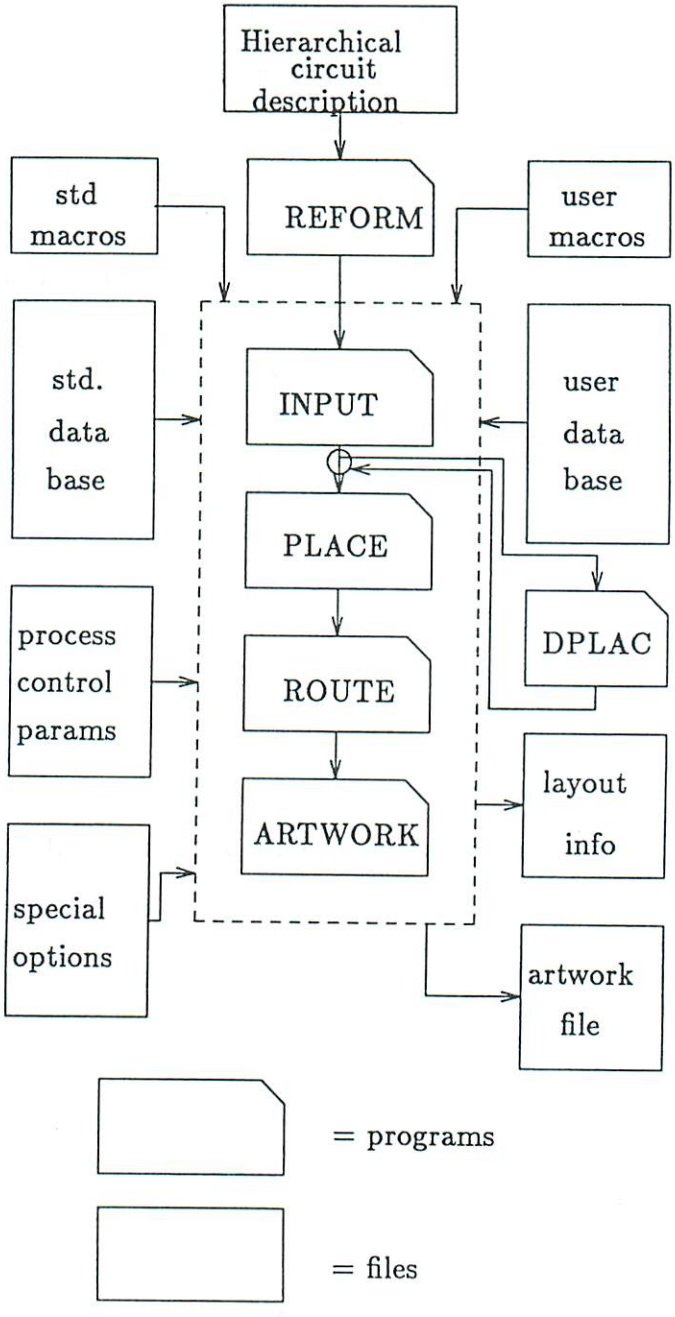


Figure 3.2: MP2D block diagram

Chip #	Circuit Type
1	8 bit array multiplier
2	16 bit 3 level CLA multiplier
3	16 bit multiplier with 8 bit CLA
4	8 bit multiplier with 8 bit CLA
5	32 bit adder with 4 bit CLA
6	16 bit adder with 8 bit CLA

Table 3.4: Chip types

Chip #	Width w	Rows n	2-point nets N	Avg wire Len 1/p
1	3009	8	1365	38.9
2	2892	12	1134	34.5
3	2937	12	1202	41.19
4	1574	9	532	44.5
5	1832	12	884	21.4
6	1301	10	590	26.6

Table 3.5: More chip characteristics

actually evaluates. Chip area estimates are within 10.6 % of the actual values. In some cases, the area estimate is less than 1% away from the actual area. As promising as these results may look, more tests must be run on larger examples in order to offer even more validation for PLEST.

3.3.6 Estimator behavior

We have also studied the variations in area with different row configurations. Fig. 3.4 shows a typical curve of the area vs. the number of rows for the layouts. The area estimate is, in most cases, minimal at or near the row configurations which were selected by MP2D. Fig. 3.5 shows a typical area estimate vs. aspect ratio curve for a chip. The area estimate seems to be minimal for aspect ratios around 1 (i.e. square chips). Also note that the area estimate does not vary a lot in that region. Furthermore, it is a well known 'rule of thumb' in chip

Chip #	Tracks (Actual)	Tracks (Est)	Area (Actual)	Area (Est)	%err (Area)
1	144	145	8.848	9.369	5.8
2	175	166	7.703	7.747	0.6
3	201	205	8.576	8.841	3.1
4	127	130	4.210	4.493	6.7
5	136	129	3.935*	3.939*	0.1
6	113	122	2.702*	2.990*	10.6
* Active area					

Table 3.6: More estimation results

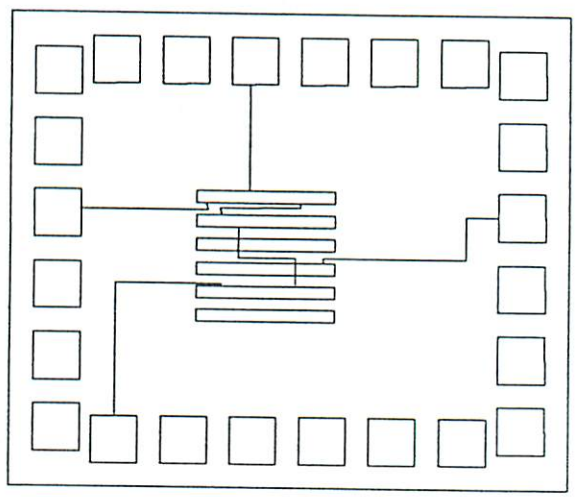


Figure 3.3: An I/O bound chip

design that chips that are as close to square as possible tend to have minimal areas. The same phenomenon was observed in our model.

3.4 Summary

This chapter concentrated on experimental validation of the model presented in Chapter 2. First, we simulated chip layouts and compared their parameters to the predictions of the model. Next, the model estimates were compared to real chips. The area estimates were within 10% of the actual values. We used the MP2D layout system to generate more example layouts. PLEST's estimates for these chips were still within the same accuracy. Finally the behavior of the estimates for different chip row configurations was studied.

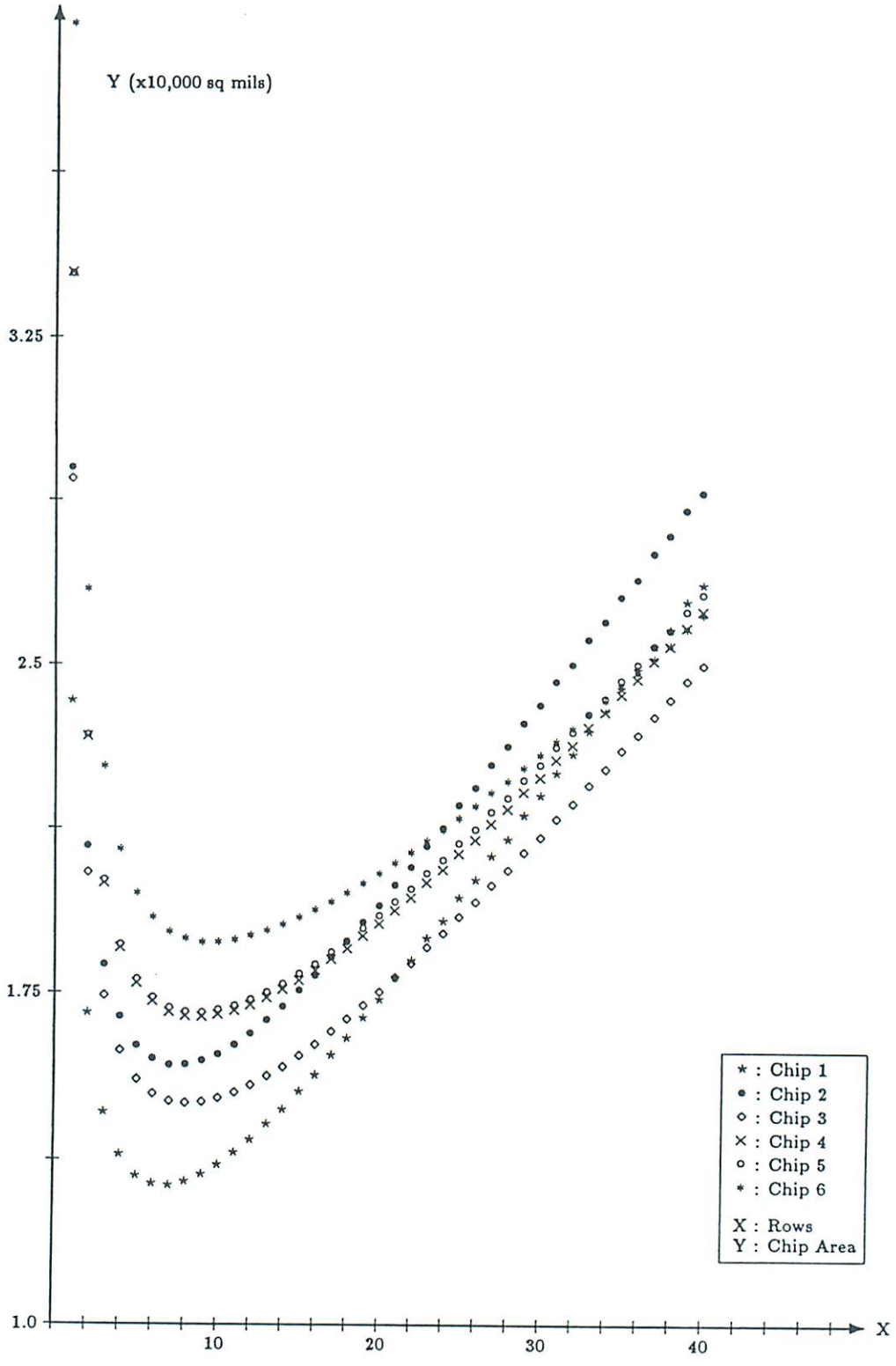


Figure 3.4: Chip area estimates vs. number of rows

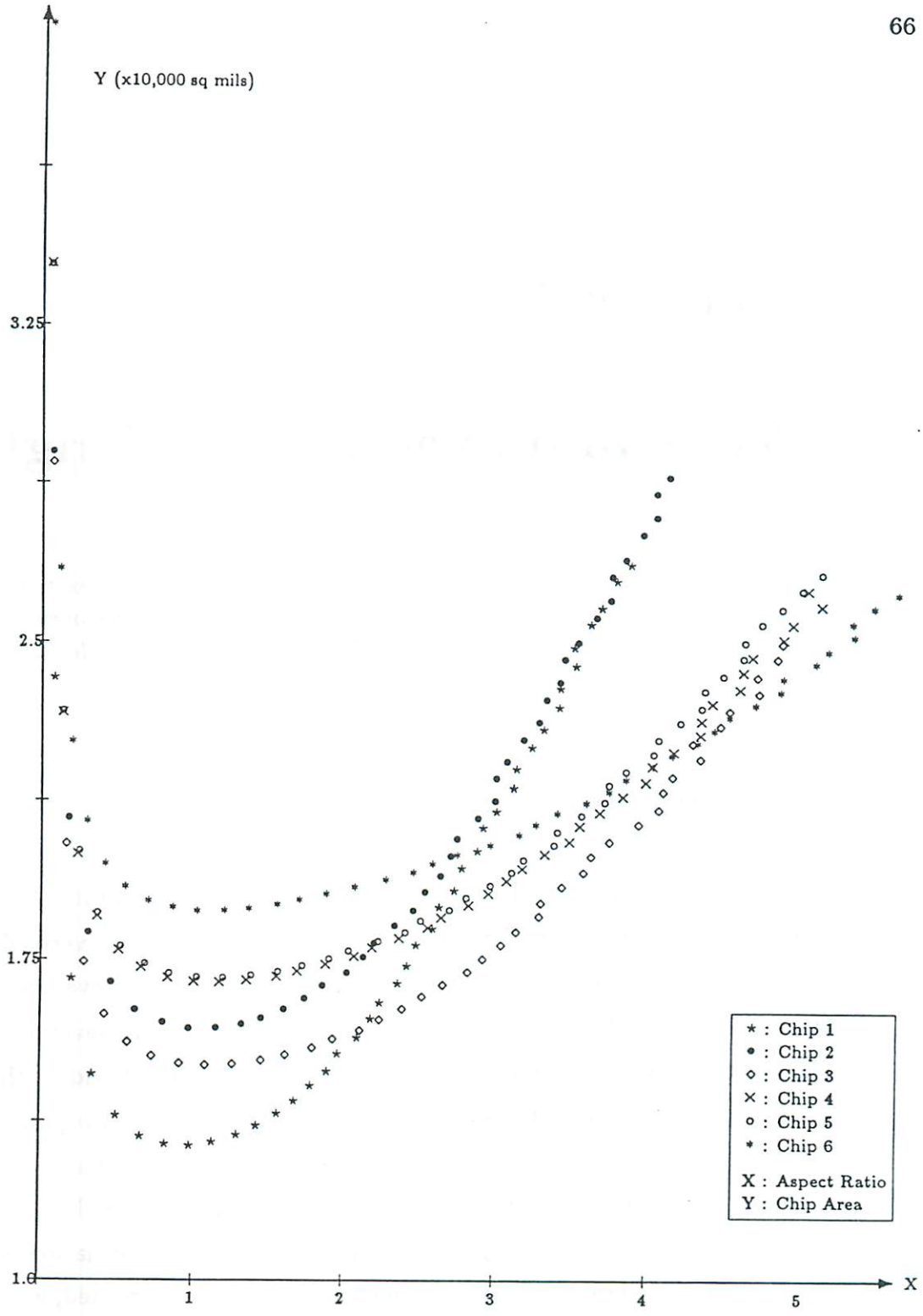


Figure 3.5: Variations of chip area estimates with aspect ratio

Chapter 4

Wires and average wire length

It is a capital mistake to theorize
before one has data.
– SIR ARTHUR CONAN DOYLE, *The Adventures of Sherlock Holmes* (1891)

4.1 Introduction

In the previous chapter, we focused on the implementation of the wiring space estimation model. The software package, PLEST, was verified and validated against data from real layouts. While the results prove that PLEST's estimates are quite accurate, in order to run the program, several parameters must be available to the user. These include the total cell width, the number of two point nets, and the average wire length. The first two parameters are almost trivial to obtain once the cell and net list description of a circuit is given. Computing the average wire length is generally a much less trivial task. Getting an *exact* value for the average wire length assumes that information is available on the details of how the cells are placed and the wires are routed, which defeats the purpose of an area estimator. In this chapter, we attempt to analyze the behavior of wires in real chips with the aim of providing the user with some guidelines on how to estimate the average wire length.

4.2 Wire length distribution

One of important steps in the process of validating the gate level area estimation model from Chapter 2 is to make sure that the assumption made regarding the wire length distribution is indeed a realistic one. In the model, we made the assumption that wire lengths are drawn from a geometric distribution. While the reasons for such a choice of distribution are presented in Chapter 2, these reasons do not constitute a validation of the geometric distribution assumption of wires. Therefore it was necessary to compare the wire length distributions in actual chips with the proposed distribution.

The first step in the validation was to extract statistics on wire lengths from the chips that were laid out using MP2D. Figure 4.1 shows a typical histogram for wire lengths in one chip. The second step was to compare the resulting distributions to the geometric form. To do so, we used the *Kolmogorov-Smirnoff* goodness of fit test which indicated that the geometric distribution is indeed a good representation of the wire length distribution within high confidence levels. The choice of the geometric distribution is therefore justified.

4.3 Rent's rule and average wire length

In order to accurately use the gate level area estimation model, one must characterize the wire length distribution associated with the specific case to which the model is to be applied. The geometric distribution is characterized by one parameter, its average. In other words, knowing the average of the distribution means we know everything about the distribution itself, including all the other moments. Therefore, estimating the average of the distribution, or the average wire length is an important task in the area estimation process.

Estimating the average wire length has been studied by several researchers. Most notably Donath [Don79], Feuer [Feu82] and Sastry [SP84] each derived relationships for average wire length estimation based on different models of placement and routing. Two observations are common to these works: one,

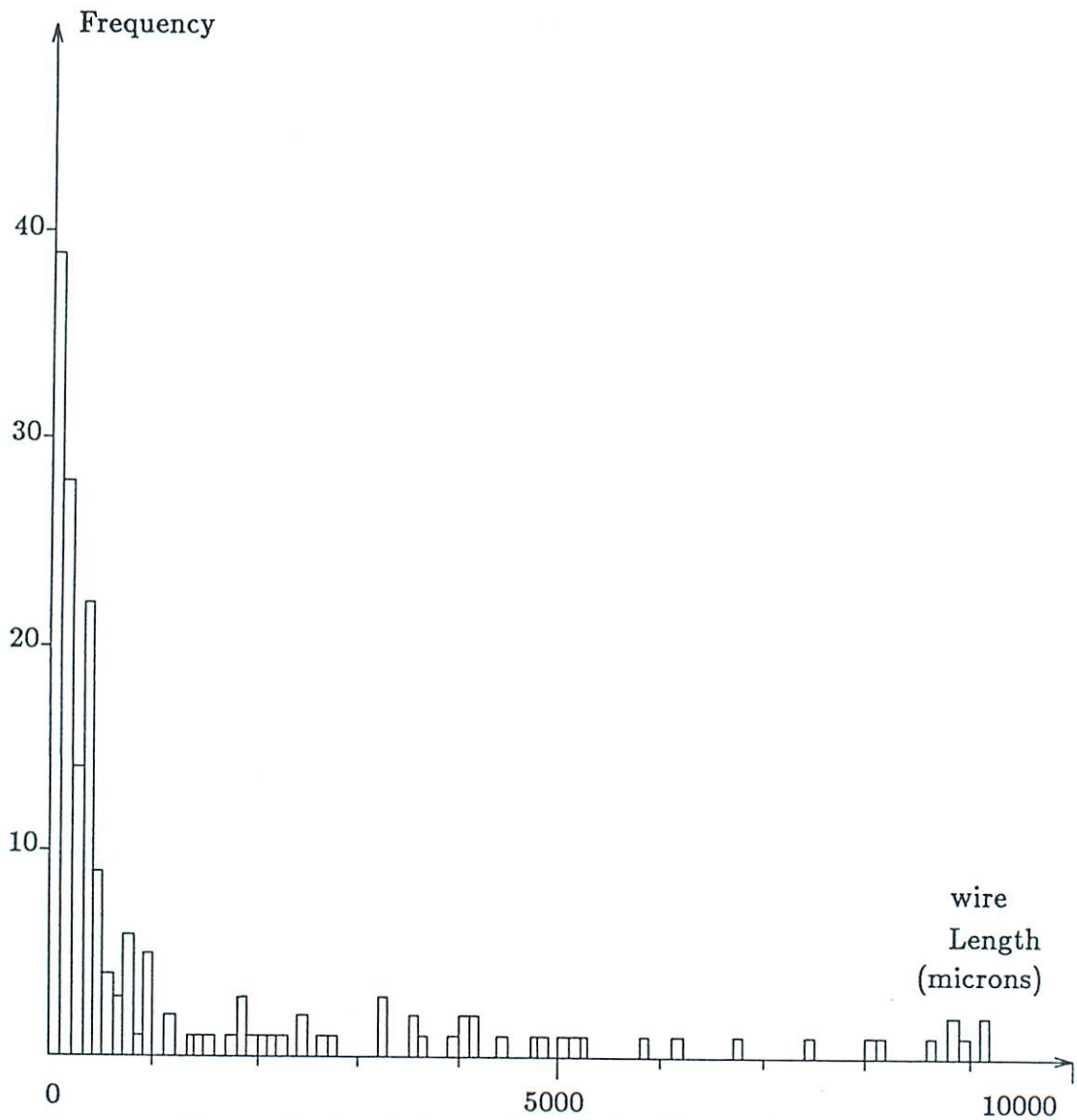


Figure 4.1: Typical wire length distribution histogram

they are all models for *gate array* type chips, and hence do make some assumptions about the regularity of the chips to be modeled. Two, these models are based on a well known empirical relationship called Rent's rule [LR71] which will be discussed in the following section.

4.3.1 Rent's rule

One observed characteristic of some 'good' placements is the empirical relationship known as **Rent's rule**. Rent's rule is an empirical relation which has been observed in well-partitioned and placed circuits and chips between the number of components in a subcircuit and the number of external terminals of this subcircuit. Qualitatively, it says that the number of external terminals of a subcircuit tends to grow slower than the number of components in that subcircuit. The intuitive explanation of this behavior is as follows. As the number of components in a partition increases, a proportional increase in the number of pins would take place. Not all these pins, however, would be expected to be external pins. This is because components in the same partition in a well partitioned circuit will have, on the average, higher connectivity to components inside the partition than to ones outside the partition. Hence many of the pins will actually be 'consumed' inside the partition and a relatively small part of the pins will actually connect to components outside the partition. Quantitatively, this rule exhibits itself as a power law relation of the form

$$T = AC^p \tag{4.3.1}$$

- T = number of external terminals (pins) in a subcircuit or partition
- A = average number of terminals (pins) per component
- C = number of components in the subcircuit (or partition)
- p = Rent's exponent, $0 \leq p \leq 1$

This relation was first observed by E. F. Rent of IBM in the late 1960s, and independently, by several others. Donath [Don69] of IBM derived the same

relation from a stochastic model which assumes a hierarchical design process. In 1971 Landman and Russo published an extensive study of the relation [LR71] where they carried out numerous experiments on partitioning large 'real-life' circuits. Their main conclusions were

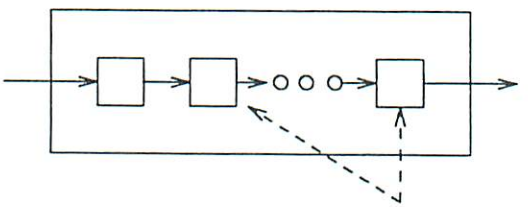
1. Rent's rule is actually a two region relationship, where region I is the power law relation of Equation 4.3.1 and region II is governed by a more complex relation.

2. Rent's exponent, p, lies, in practice, between 0.47 and 0.75 and depends upon the structure of the circuit and the partitioning algorithm. In a subsequent paper, Russo concluded that, for the same partitioning algorithm, p tends to be high for high performance circuits, and low for low performance circuits. This can be visualized by the simple example in Fig. 4.2.

In his famous book on *fractal geometry* [Man81], Mandelbrot presents Rent's rule as a *dimensionality ratio* between gates and external pins.[1] In simpler words, one can think of Rent's rule as an *area-to-perimeter* relation. Figure 4.3 illustrates the case where the subcircuit can be laid out so that all the external pins lie on the periphery. The number of these pins is hence proportional to the perimeter of the circuit, the number of gates being proportional to the area. A Rent's exponent of 1/2 results (1 being the dimension of the perimeter, 2 that of the area). Examples of such circuits include 2-D memory arrays, PLAs and the like. If circuits are more complex than that, there may not be enough 'space' on the perimeter of the circuit to place all the external pins, and one would expect a larger value for Rent's exponent. Figure 4.4 illustrates a 3-dimensional cubic layout of a subcircuit in which all the external pins lie on the *surface* of the cube. The number of gate is now proportional to the volume (dimension 3), the number of external pins is proportional to the surface (dimension 2), and the resulting Rent's exponent is 2/3, a value frequently observed in the data analyzed for Rent's rule.

[1] A similar argument is also presented in [Key81].

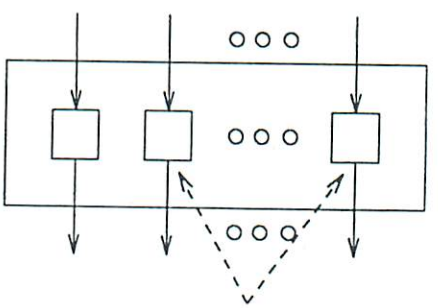
serial shift register



$T=2=2*C^0$ flip flops

$p = 0$

parallel register



$T = 2* C$ flip flop

$p = 1$

Figure 4.2: Extreme cases for Rent's rule

One can make two interesting related remarks here. First, since 3-D type layouts are still, at best, laboratory experiments, circuits with Rent's exponent greater than 1/2 will result in longer wires as the circuit size increases. This is due to the fact that not all external connections can be made from the perimeter and more wires have to be routed from 'inside' the circuits, resulting in increasingly longer connections. The second observation is that one would expect an appreciable decrease in routing complexity of circuits when 3-D VLSI chips become a wide spread reality, since more and more pins can be placed on the circuit periphery (surface) thereby providing some amount of abuttment of subcircuits. Finally, in Mandelbrot's theory on fractal geometry, non-integer dimensions are allowed, and therefore values of Rent's exponent other than 1/2 and 2/3 are quite feasible.

One underlying assumption in all the discussions above is that the circuits are partitioned and placed using *optimal* algorithms. In real life this is not the case and one has to settle for suboptimal output. As a result of this, Rent's parameters may deviate from the values that they would have taken under optimal conditions. The amount of deviation is usually dependent on the layout algorithms used, particularly the partitioning algorithms. In a way, this means that Rent's parameters do also characterize the quality of the particular placement and partitioning algorithms used for a specific system.

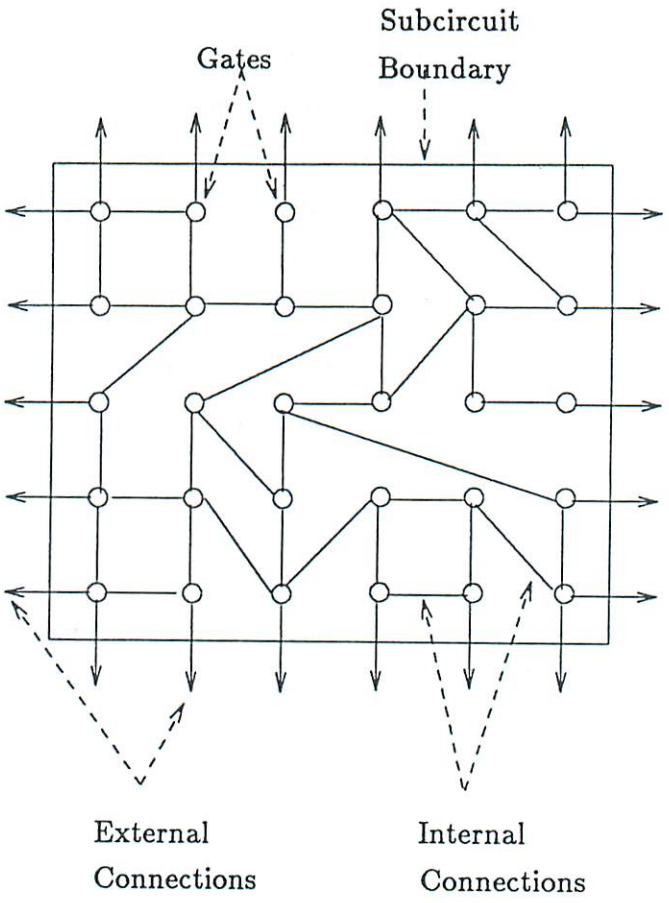


Figure 4.3: Rent's exponent = 1/2

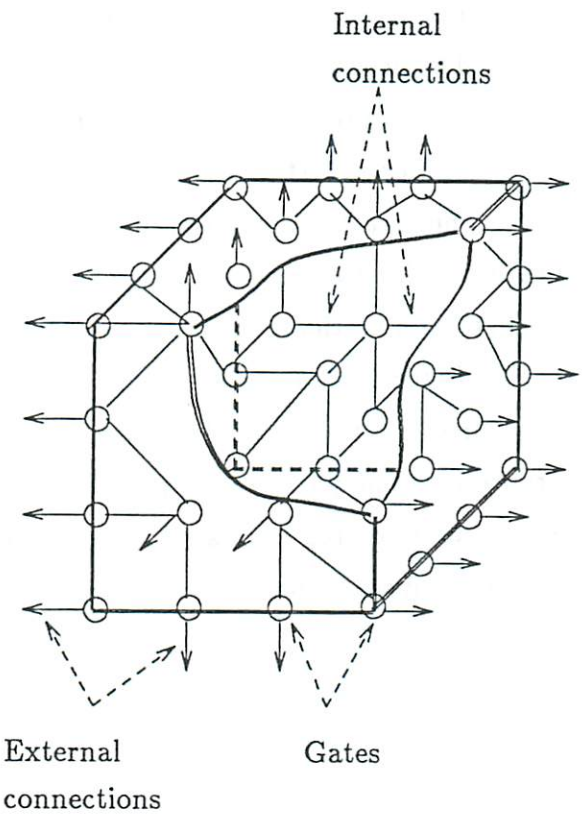


Figure 4.4: Rent's exponent = 2/3

4.3.2 Rent's rule in standard cell designs

The extensive amount of data that has been investigated successfully for its adherence to Rent's rule suggests that it tends to apply to well-placed circuits. The layouts which were studied, however, were almost exclusively gate array designs. To investigate the applicability of Rent's rule to standard cell designs, we carried out experiments on several chip designs and studied the relationship of partition size to partition connectivity. The experimental procedure consisted of running circuit descriptions through the MP2D partitioner, DPLAC, with a specified number of partitions. We then computed the average partition size and the average number of external pins per partition. This procedure was repeated several times, each time specifying a different number of partitions. This procedure is similar to the one carried out by Landman and Russo and described in [LR71] to investigate Rent's rule. The experiment was carried out on several circuits. Figures 4.5 and 4.6 show the results for two circuits : an 8-bit array multiplier and an 8-bit serial (add-shift) multiplier, respectively. Each point on the graph represents one partitioning procedure (i.e. one run of DPLAC). In order to correlate the predicted Rent's relation to the experimental values, data was plotted on a log-log scale. So the power law relationship

$$T = AC^p$$

is plotted as

$$\log T = \log A + p \log C$$

and the plotted relationship is now a straight line equation.

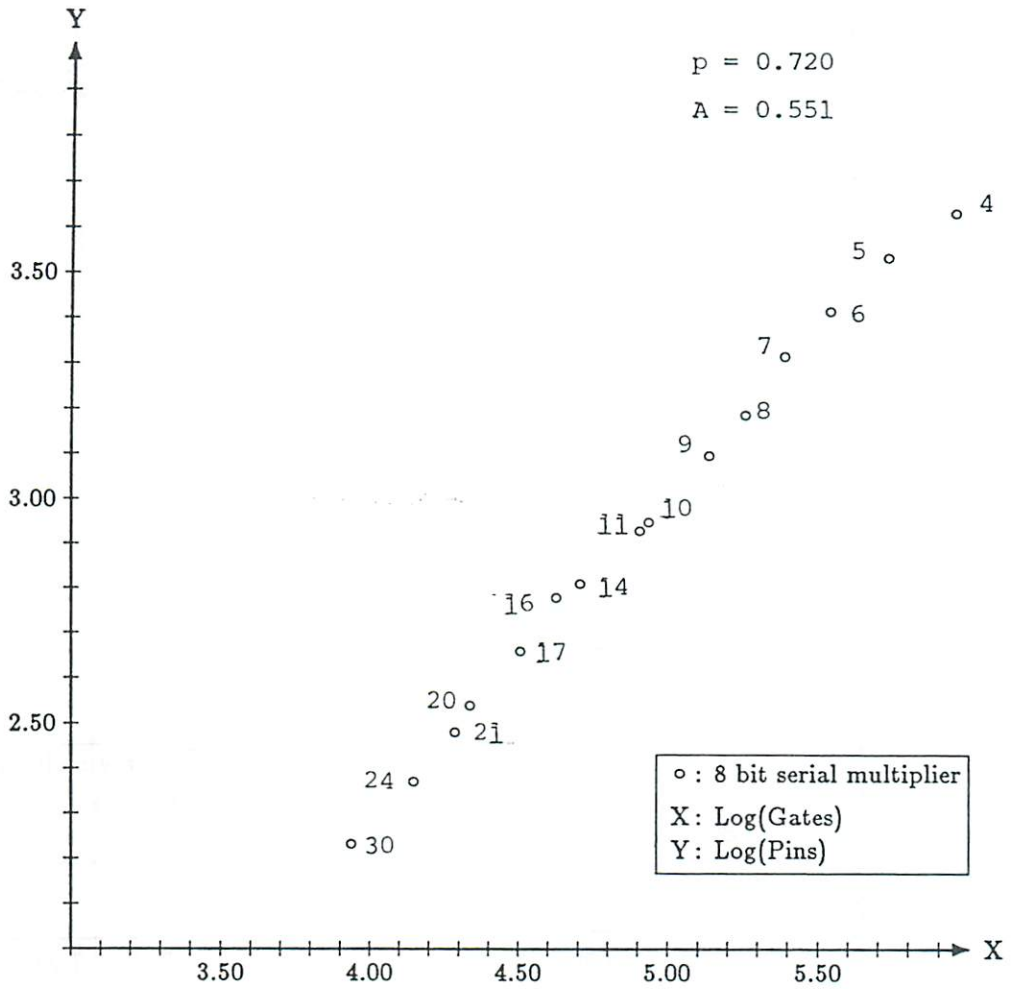


Figure 4.5: Rent's rule fit for the serial multiplier

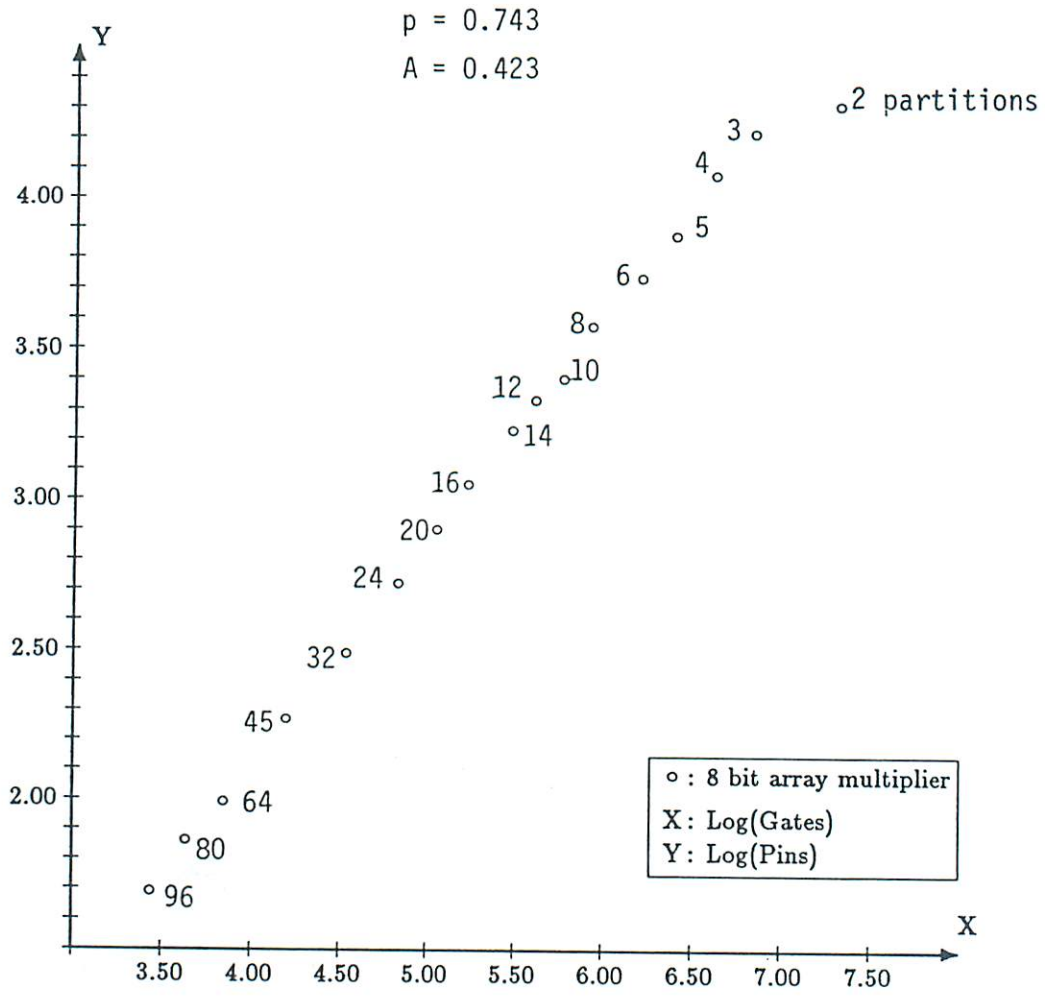


Figure 4.6: Rent's rule fit for the array multiplier

In both cases, the data points appear to have an excellent fit to the predicted straight line. Similar fits were observed in all the other circuits. Rent's rule parameters can be estimated by regression analysis on the data points and are shown in the graph. The slope is Rent's exponent p and the intersect is the scaling constant A. We observe that Rent's exponent in the case of the array multiplier (0.74) is higher than that of the serial multiplier (0.72). This confirms the statement in the original paper that highly parallel components tend to have higher Rent's exponents. Also note that, in the case of the array multiplier in Figure 4.6, Rent's relationship tends to break down when the number of partitions is 2. The same effect was observed in the original experiment by Landman and Russo. It can be concluded from the results of the experiments that Rent's rule does indeed apply to standard cell designs.

4.3.3 Relation of Rent's rule to average wire length

Having investigated and confirmed the adherence to Rent's rule of standard cell designs, we now look at using the rule to estimate the average wire length. As mentioned in Section 4.2, such a relation was studied by three researchers, Donath, Feuer and Sastry, each of which derived a relationship of average wire length to Rent's parameters based on a different model.

The relationship proposed by Donath [Don79] is derived from a model which assumes a hierarchical partitioning and placement of logic on a square array. Using Rent's rule to estimate the number of wires connecting the partitions at each level of the hierarchy, an expression for the average wire length, $\bar{R}$, is derived. Donath found that, for a circuit of C gates and Rent's exponent p, $\bar{R}$ varies in the following way

$$\begin{aligned} \bar{R} &\sim C^{p-1/2}, \quad p > \tfrac{1}{2} \\ \bar{R} &\sim \log C, \quad p = \tfrac{1}{2} \\ \bar{R} &\sim f(p), \quad p < \tfrac{1}{2} \end{aligned}$$

$f(p)$ being independent of the number of gates, C. Two observations need to be made here. First, the 'critical' value of $p = 1/2$ for Rent's exponent seems

to agree nicely with the intuitive arguments presented in Section 4.3.1. Second, the experimental results presented in Donath's paper show a large difference between theory and real values of $\bar{R}$, even though the variations in $\bar{R}$ seem to agree with what the theory predicted. This makes the relationships less useful from a practical point of view.

Feuer's model [Feu82] is aimed at relating Rent's rule to the distribution of wire lengths on a chip. Using a continuous model, he defines a partitioning function, $I(R)$, as the number of connections born inside and terminating outside a 'circle'[2] of radius R. Clearly, this partitioning function is another name for Rent's rule. He then proceeds to derive the wire length distribution from $I(R)$ by integrating an infinitesimal strip around the circle over the whole plane. The wire length distribution is found to be of the form

$$q(r) \sim r^{-2(2-p)} \tag{4.3.2}$$

and the average interconnection length is found to be

$$\frac{\bar{r}}{\delta} = \sqrt{2}\frac{2p(3+2p)}{(1+2p)(2+2p)}\frac{C^{p-1/2}}{(1+C^{p-1})} \tag{4.3.3}$$

where $\frac{\bar{r}}{\delta}$ is the average wire length in *gate length* units, δ. The predictions of Equation 4.3.3 are compared to the experimental data on real chips, assuming a constant value of $p = 2/3$, and the results show a close agreement. This indicates that the formula may be useful in practice. Even though the derived wire length distribution is different from the geometric distribution assumed in our model, the two distributions, however, approximate each other over a large range of values. This means that the average value found from 4.3.3 can be used as an estimate of the average of the geometric distribution. One drawback of this model is that it does not allow for values of $p \leq 1/2$. Such values of p were not, however, observed in the layouts we studied.

In his thesis [Sas85], Sastry assumes a continuous model of logic. With this in mind, he uses an analysis borrowed from the theory of reliability to derive

[2]Feuer assumes a 'Manhattan' type model, where only two directions, x and y are permitted, so a circle is really a diamond of diagonal $2R$

a relationship between Rent's rule and the wire length distribution, showing that, if Rent's rule applies, then the wire length distribution is expected to be of the *Weibull* family of distributions. A relationship is derived between Rent's parameters and the Weibull parameters. The Wiebull distribution very closely fits the frequency histograms of wire lengths taken from real layout data. The average of the Weibull distribution is then computed from Rent's parameters and used as an estimate of the average wire length, and is given by

$$\bar{R} = \frac{1}{\beta}\left(\frac{1}{\alpha}\right)^{\frac{1}{\beta}}\Gamma\left(\frac{1}{\beta}\right) \tag{4.3.4}$$

where α and β are Rent's exponent and scaling constant, respectively, and Γ is the factorial function. The estimates from Equation 4.3.4 agree closely with the actual values. While the model seems to closely approximate the behavior of wires in gate arrays, on which the investigated circuits were built, we note that the average wire length as predicted in Equation 4.3.4 does not vary with the gate count of a circuit. In the next section, we present some experimental results which show that this may not be the case in standard cell chips. Furthermore, Equations 4.3.4 and 4.3.3 were both used to estimate the average wire lengths in some of the chips presented in Sections 3.3.2 and 3.3.5. The predictions from Feuer's formula seem to model standard cell layouts better than Sastry's.

4.4 Average wire length variations with chip size

In the works of Feuer and Donath, the average wire length in a circuit is shown to be determined solely by the gate count and Rent's parameters of that circuit. Although such a statement may be too strong to be general, it seems intuitive that the lengths of the wires are determined both by the *type* of the circuit and its *size*. Rent's parameters seem to give a good indication of the type of the circuit. Highly parallel circuits are expected to have a larger Rent's

exponent than highly serial ones, for example. The size of the circuit is generally expressed as its gate count.

In attempting to investigate the behavior of the average wire length and to compare it to what was predicted by Feuer and Donath, the following experiment was carried out. We first designed a four-bit multiplier as shown in Figure 4.7. The multiplier was laid out using the MP2D system. Next we increased the bit width to eight and did another layout. The experiment continued by designing the same multiplier with bit widths from 4 to 32 in four bit increments. For each one of the 8 layouts, we extracted the average wire length as well as the gate count. The experimental data of average wire length versus gate count was plotted on a *log-log* scale. The resulting graph is shown in Figure 4.8. These multipliers represent the same circuit type, and thereby their Rent's parameters are expected to be very close, if not identical. In Section 4.3.2, we analyzed the partitioning data from the 8 bit add-shift multiplier layout data and obtained its Rent's parameters. If these parameters have the same value for all the add-shift multipliers of bit widths 8 to 32, then, by Feuer's and Donath's predictions, the average wire length is expected to increase as $C^{p-1/2}$. With $p = .72$, then the average wire length is expected to increase as $C^{.22}$. On a log-log scale, such a dependency would appear as a fit of the experimental points to a straight line of slope around .22. While the fit of the points in the graph of Figure 4.8 to a straight line is not as good as the case of the Rent's rule graphs of Figures 4.5 and 4.6, there is enough correlation of the data to claim, with reasonable confidence, that the average wire length tends to grow as C^{α}, α being the slope of the regression line. By regression analysis, we found $\alpha = .28$ which is not too far away from the predicted slope of .22.

The main conclusion that one can infer from this experiment is that there is a strong dependency between the average wire length and the circuit size and that this dependency seems to be more or less characterized by Rent's exponent. This allows us to predict the average wire length using Rent's rule.

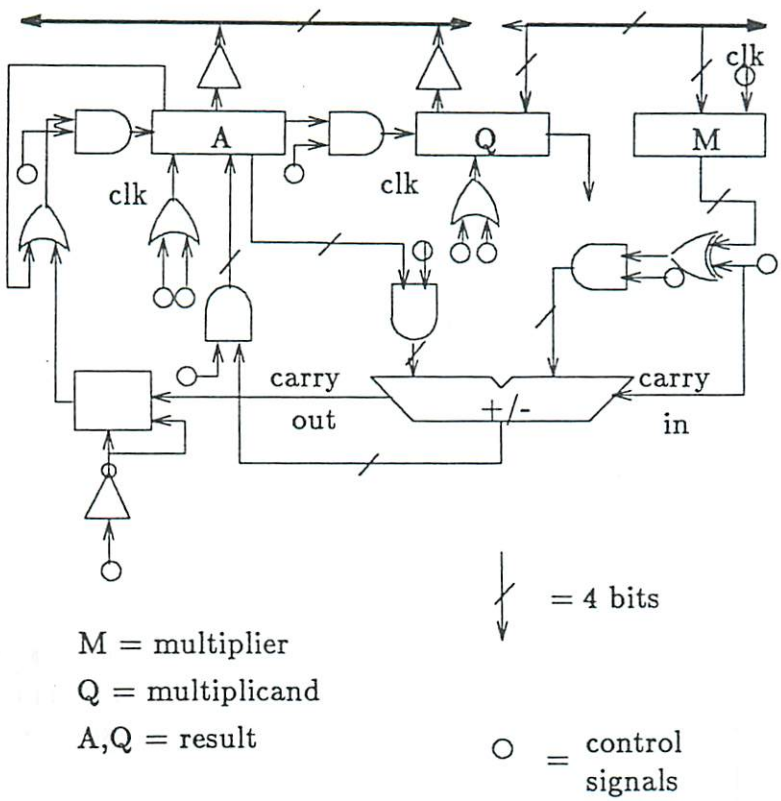


Figure 4.7: Four bit serial multiplier design

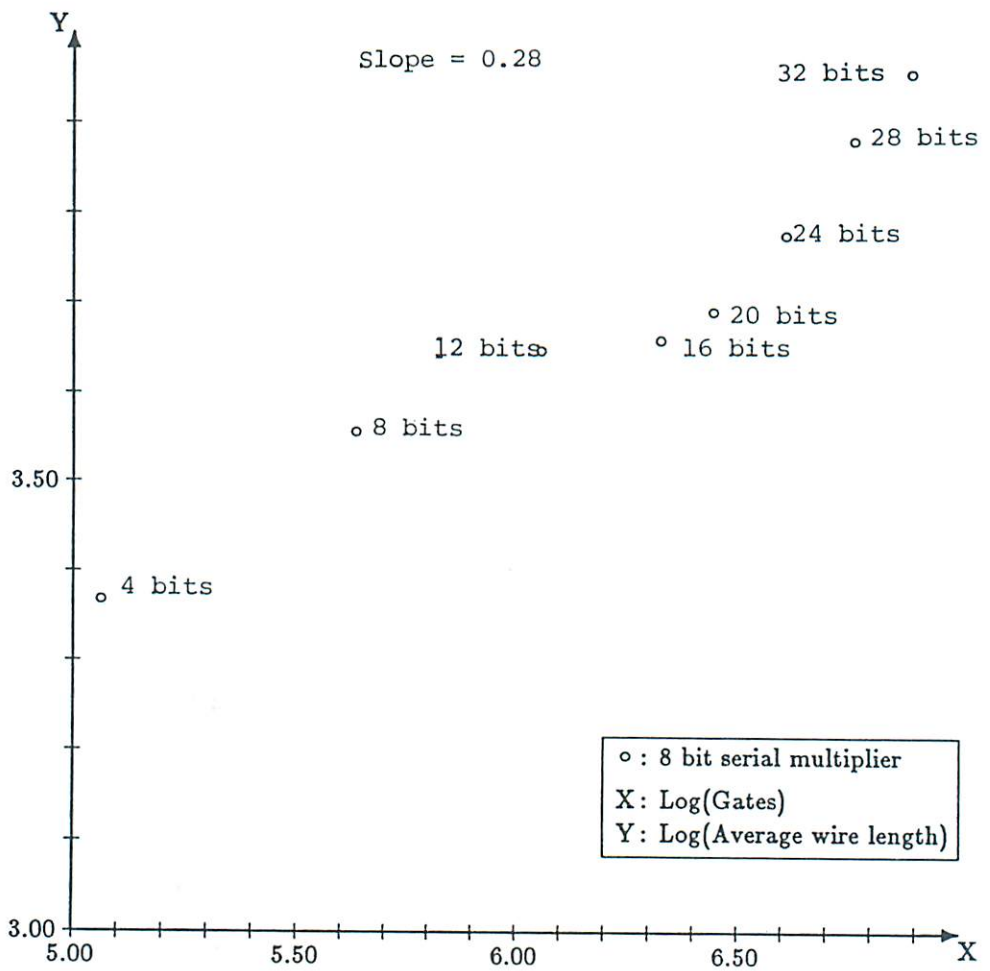


Figure 4.8: Average wire length variations with gate count

4.5 Guidelines for average wire length estimation

The main purpose of studying the behavior of wires, Rent's rule and the average wire length is to provide some means for estimating the latter quantity. Estimating the average wire length on a chip is, as we have seen, a difficult task, since different placement and routing algorithms may produce layouts in which wiring behavior is different for the same input circuit. In order for the estimation task to be accurate, we must have sufficient amount of information about the layout system itself, in addition to the information about the circuit under consideration. In Section 4.4 it was discussed that Rent's parameters do indeed characterize the circuit as well as the placement and partitioning algorithms. Another way to characterize a system is to gather statistics from runs on similar previous circuits and use these statistics to help predict the behavior of the system on future input.

4.5.1 Rent's parameters

In Section 4.4 we presented the works previously done on estimating the average wire length from Rent's parameters. It was concluded in Section 4.3.3 that Feuer's prediction formula 4.3.3 was the most suited for use to estimate the average wire length in standard cell designs. Using Equation 4.3.3, however, assumes that we know the value of Rent's exponent. In order to obtain estimates of Rent's exponent, we must follow the procedure used in Section 4.3.2 which consists of successively partitioning the input circuit, each time with a different number of partitions. The resulting data is plotted on a log-log scale, using the regression line slope as an estimate of Rent's exponent. The partitioning process is usually much cheaper than the whole layout process. In the case of MP2D, running the partitioning phase only took about 2 to 5 minutes of CPU time for each partitioning point in Figures 4.5 and 4.6 as opposed to more than 10 hours for the whole layout process in each case. Furthermore, the excellent fit of the

data points to Rent's rule predictions indicates that a relatively small number of partitioning runs is necessary to provide an accurate regression estimate of Rent's parameters. In his paper, Feuer uses a 'default' value of 2/3 for Rent's exponent. Hence even if no partitioning runs are performed, it is still realistic to assume a default value for Rent's exponent. The value of such a default value is a characteristic of the specific placement and partitioning algorithms used. While Feuer's default value of 2/3 for the exponent is probably derived from the fact that such value is typical for gate array partitioning and placement algorithms, the experience gained from studying standard cell systems such as MP2D indicates that a default value of Rent's exponent of around 0.7 is better representative of the placement and partitioning algorithms for standard cell layouts.

To illustrate the use of Equation 4.3.3 to estimate the average wire length, let's take as an example the 8 bit array multiplier from Table 3.5. Rent's exponent was extracted from Figure 4.6 by regression and found to be $p = 0.743$. The gate count, C for the circuit is 708. Feuer's formula 4.3.3 gives an estimate for the average wire length

$$\frac{\bar{r}}{\delta} = 6.098 \text{ gate lengths}$$

Using the cell library data, the average number of devices (transistors) per gate is found to be 7.55, the average number of devices per pin pitch is computed to be 1.58, and hence δ is 4.78, and the average wire length estimate is found to be 30 pin pitches. While this estimate is about 23% off the actual value (38.9), when fed to PLEST as input, the resulting area estimate was found to be 8.124 mm^2, only 8% different from the actual area (9.36 mm^2). This indicates that relatively large errors in average wire length estimation do not largely penalize the accuracy of the area estimation provided by PLEST. To further justify this point, we studied the sensitivity of the area estimates to the average wire length variations. Figures 4.9 and 4.10 show the variation of PLEST's area estimates for the total chip area and the active areas (respectively) of the 8 bit array multiplier with the average wire length. It can be seen that the area estimates

show relatively little variation with the average wire length. The data indicated that sensitivity of the total area estimates to the average wire length is around 0.58 (total area) and 0.61 (active area) for this circuit, which means that a relative error of x in the average wire length estimation is expected to introduce a $0.58x$ (total) and $0.61x$ (active) relative error to PLEST's area estimate.

4.5.2 Statistical analysis

Another approach to estimating the average wire length of a circuit layout is to analyze the layouts obtained from previous runs of the system on other circuits, gather statistics on the average wire length and use these statistics to estimate this quantity in future chip layouts. As discussed earlier, the major factors affecting the average wire length in a circuit are the circuit size, the type of circuit, and the layout system itself. The first quantity can be expressed as the gate count in a circuit. The second and third quantities, as argued in the previous section, can be characterized by Rent's parameters. Therefore, in order to encompass these first order effects in statistics on the average wire length, we must study the average wire length variations with respect to gate count and Rent's parameters (mainly Rent's exponent). An example of such statistics is presented in Section 4.4, where average wire length variations with gate count were observed, fitted to a line, and plotted in Figure 4.8 for a specific value of Rent's exponent. By clustering the available data points into subsets having different values of Rent's exponent and fitting them to straight lines, we obtain a set of regression lines, one for each value of Rent's exponent. How many such lines are needed and how separated must be the successive values of Rent's exponents is difficult to ascertain and is heavily dependent on the layout system itself and the type of circuit being laid out.

To show how to use the average wire length statistics to estimate the average wire length and thereby the area of a chip, let's take again the 8 bit array multiplier from Table 3.5. The value of Rent's exponent in this case is obtained from the regression line in Figure 4.6, which is $p = 0.74$. This value is very near

the one for the serial multiplier obtained from the regression line in Figure 4.5, which is 0.72. Therefore we can use the statistics obtained on the average wire lenth variations for the serial multipliers in Figure 4.8 to estimate the average wire length for the array multiplier. The regression line has a slope of 0.286 and an intersect of 1.91. The gate count in the array multiplier is 708 gates, which results in an estimate for the average wire length of 44.06 pin pitches. This value is within 13 % of the actual average wire length (38.9). When fed into PLEST, the area estimated was 10.069 mm^2 which is only 7 % away from the actual area (9.369 mm^2).

One final note here is that, although the average wire length estimation techniques discussed resulted in area estimates within acceptable accuracy, the statistics used were obtained from a relatively small number of previous layouts. Clearly, more experience must be gained in the layout system being used through gathering and analyzing more data, before a more assertive generalization is to be made on the behavior of wires and the average wire length.

4.6 Summary

The goal of this chapter was to conduct comprehensive and experimental studies on chip layouts in layouts with the aim of better understanding the behavior of wires and its relation to placement and partitioning. We used the MP2D layout system to run numerous experiments on layouts with the aim of studying the various layout tasks. Rent's rule [LR71] was found to apply to the resulting layouts. This was used to characterize the placement an partitioning procedures and to provide guidelines to estimate the average wire length of a circuit layout.

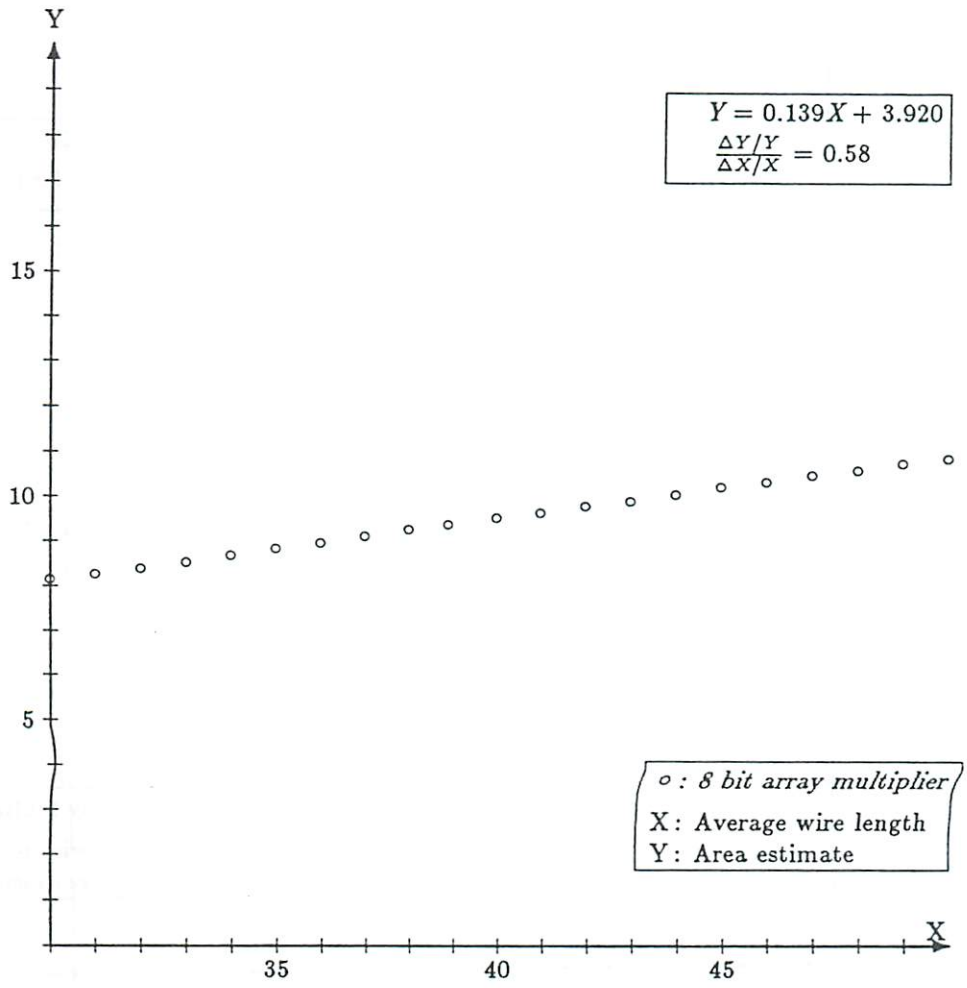


Figure 4.9: Area estimate variations with average wire length for the array multiplier

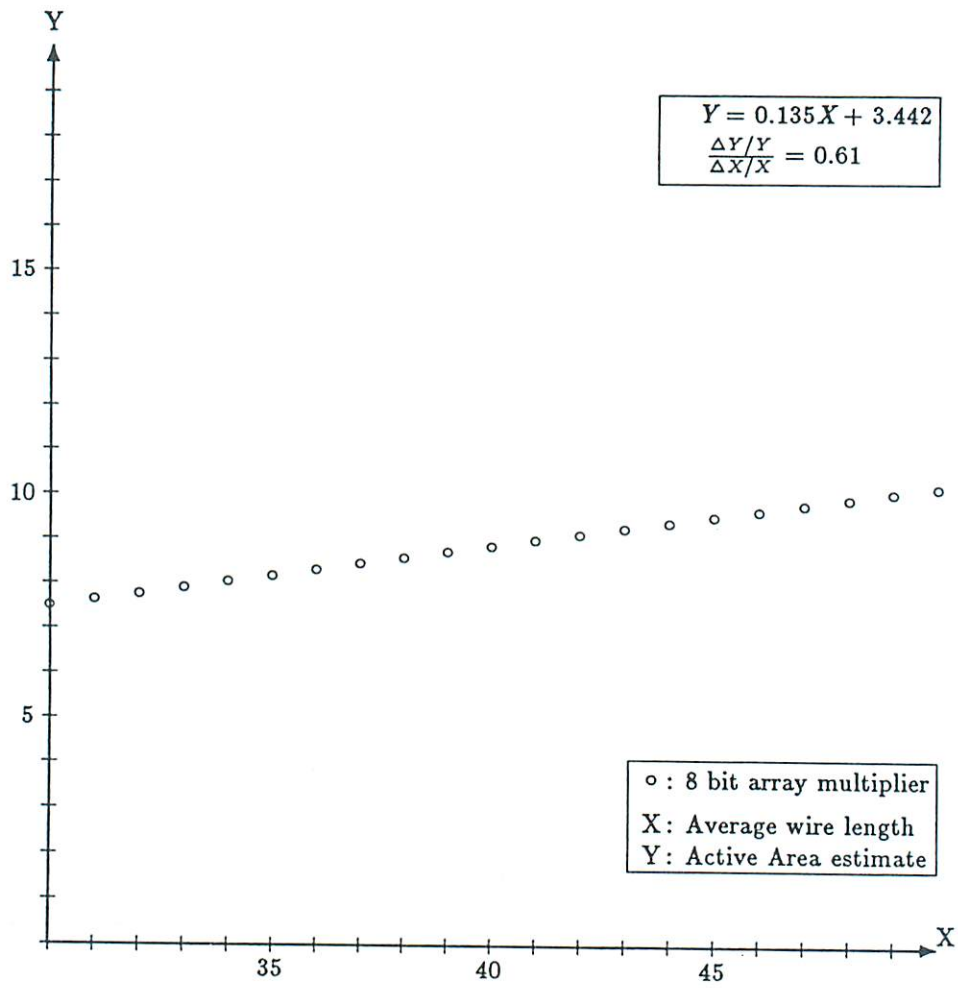


Figure 4.10: Active area estimate vs. average wire length for the array multiplier

Chapter 5

Estimating functional requirements

Beings low in the scale of nature are
more variable than those which are higher.
– CHARLES DARWIN, *On the Origin of Species* (1859)

5.1 Introduction

In previous chapters, we made the assumption that the design logic has been specified in detail, and therefore, the cell (or functional) area is known. In general, however, the designer would like to start by specifying the abstract functionality of the end product, while the decisions involving the implementation details are left to the automatic *synthesis* programs or to the designer. In this chapter, we look at the task of establishing some bounds on the size and cost of the functional (or logic) parts of a VLSI design, given the functional specifications. We begin by presenting the means by which we describe the design initially. Then, we look at tradeoffs in functional module design by examining adders and multipliers. Next, we look at the problem of bounding the cost of the cheapest design implementation and the fastest implementation as a step towards bounding the whole design space.

5.2 The high level design description

Initially, the designer describes a circuit in terms of *what* it is supposed to do, or its *function*. Such a description is usually subject to constraints such as speed of execution, cost of implementation, or power consumption. Ideally, the designer need not have any knowledge as to *how* the design will be implemented, (i.e. its *structure*).

There are several means of describing (or specifying) the function of a design. They are broadly categorized as follows:

- Hardware Description Languages (ISPS [BS77], SLIDE [PW81]),
- Formal models (Behavior expressions [McF81])
- graphical representations (Value trace [McF78], dataflow), and
- natural language (PHRAN-SPAN [Gra86])

In his thesis, [Gra86], Granacki presents an excellent discussion of the merits and penalties of each category. For the remainder of this thesis, we shall assume that the function of a design is specified as a dataflow graph. An example dataflow graph is shown in Figure 5.1. We formally define a dataflow graph, in the following manner:

Definition 5.1 *A* **Dataflow** *graph is a directed graph, $G = (V, E)$, such that*

1. *Each node $v \in V$ is called an* operation, *associated with each operation is a* operation type o_v, $o_v \in O$, *where O is the set of possible types of operations that a node can perform.*
2. *Each edge $e = (v_1, v_2) \in E$ is called a* value, *it is an* input *to operation v_2 and an* output *of operation v_1*
3. *There exist two special nodes, s and t, called* source *and* sink, *respectively, such that s has no inputs, and t has no outputs.*
4. *Every edge (value) is on some directed path from s to t.*

5. *An operation type can be a* distribute (D), *a* join (J), *or a* calculation *(e.g. add, subtract, compare, etc..)*

6. *A* DJ block *is a subgraph* $H = (U, F) \subset G$, *with the following properties,*

   - H *contains a distribute node* D_H, *and a join node* J_H,
   - *Every edge output to a node in* $U - J_H$ *is on some directed path from* D_H *to* J_H

5.3 Design tradeoffs and the design space

Predicting the functional requirements of an RT level design is usually a more complex task than wiring area estimation. The main reason for this is that the solution to the synthesis problem is not unique. In other words, given a functional design specification, one can synthesize not only one, but a whole range of possible RT level implementations satisfying the specs, and performing the required functions. Different designs would correspond to different area-speed-power performances (among others). So one can have fast designs which take up large areas on silicon, and would probably consume a lot of power, or serial designs which are compact in area and power consumption. Hence it would be more appropriate to talk about a *design space* which embodies all the possible implementations for a given design specifications.

Since the synthesis process does not have a unique solution, a single prediction may not be useful. A much more useful task would be to explore the design space and thereby give the designer an *overview* of the possibilities at hand involving the various tradeoffs in design characteristics.

5.3.1 Design tradeoffs for adders and multipliers

In order to better visualize and understand the solution space associated with synthesized designs, we conducted two experiments which aimed at

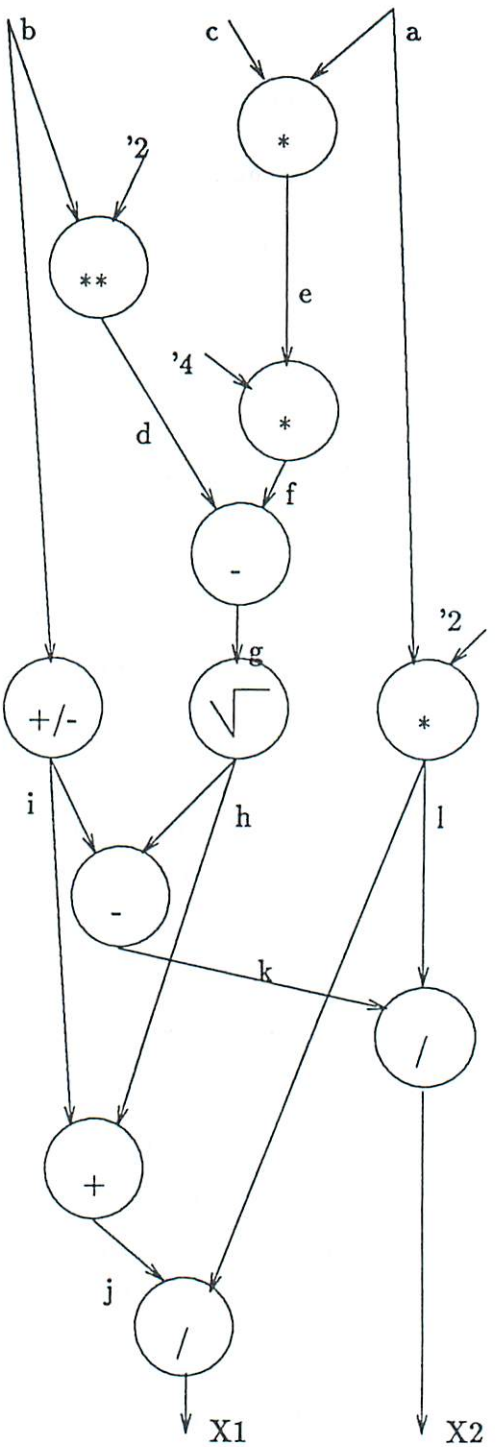


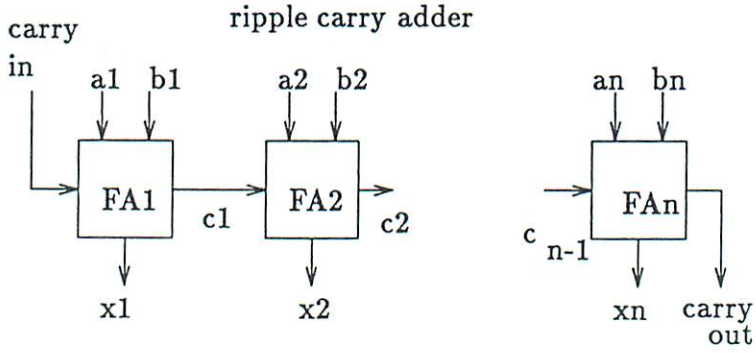
Figure 5.1: An example dataflow graph

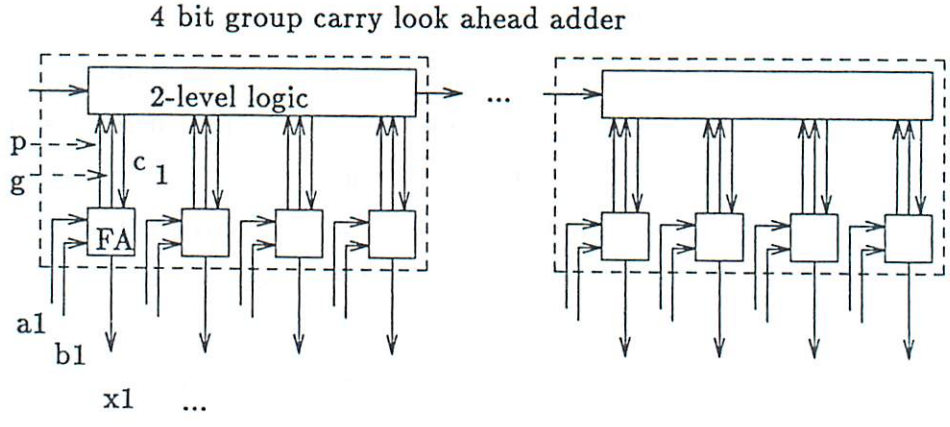
exploring the design tradeoffs and the design space of two type of arithmetic circuits, adders and multipliers. The experiments consisted of synthesizing several different implementations of adders and multipliers and looking at the *cost-speed* tradeoffs associated with each type of circuit.

5.3.1.1 Tradeoffs for adder circuits

Figure 5.2 shows different ways of implementing an n bit adder. The ripple-carry adder is the most compact in area, but it is also the slowest, and the worst case delay increases linearly with the bit width. Adders could be implemented with carry-look-ahead capability, which introduced an appreciable decrease in delay, but at a high cost in gate count and area. Because of fan-in restriction and gate complexity, carry-look-ahead circuits become impractical for adders more than 8 bits wide. In such cases, group carry-look-ahead circuits are used, where the n bits are divided into groups of 4 or 8 bits wide, each with a carry-look-ahead block. These groups are connected in a ripple-carry fashion. Group carry-look-ahead implementations are slower than their full carry-look-ahead counterparts but the reductions in gate cost and area are appreciable.

Figure 5.3 shows the area-delay tradeoffs for adders of 8,16, and 32 bit widths. For each bit width group, three RT level designs were implemented and laid out using MP2D; ripple-carry, 4-bit group carry-look-ahead, and 8 bits group carry-look-ahead. One interesting result observed is that the various adder implementations, for a fixed bit width n, tend to fall on a curve of the form $AT^{\alpha_n} = k_n$ where α_n and k_n are parameters depending on n. Also note that α_n decreases with n. This observation may be important, for if such a relation is proven to be applicable in the general case, it may indicate that the design space can be further bounded by the curve and thereby provide means of a priori prediction of design functional size under the designer imposed constraints of cost and/or speed.





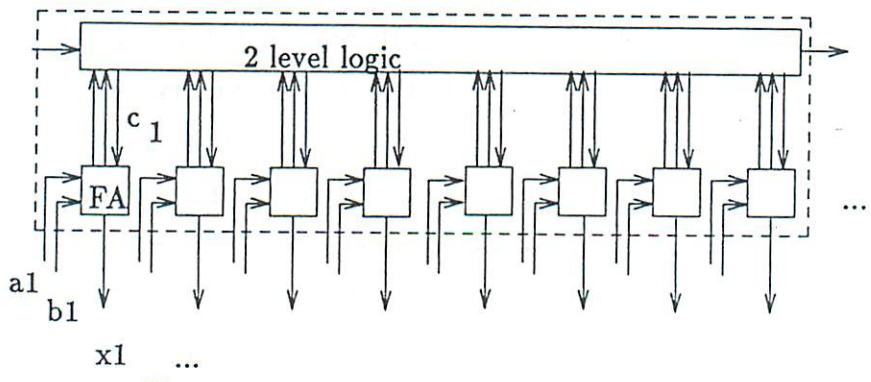


Figure 5.2: Different implementations of an adder

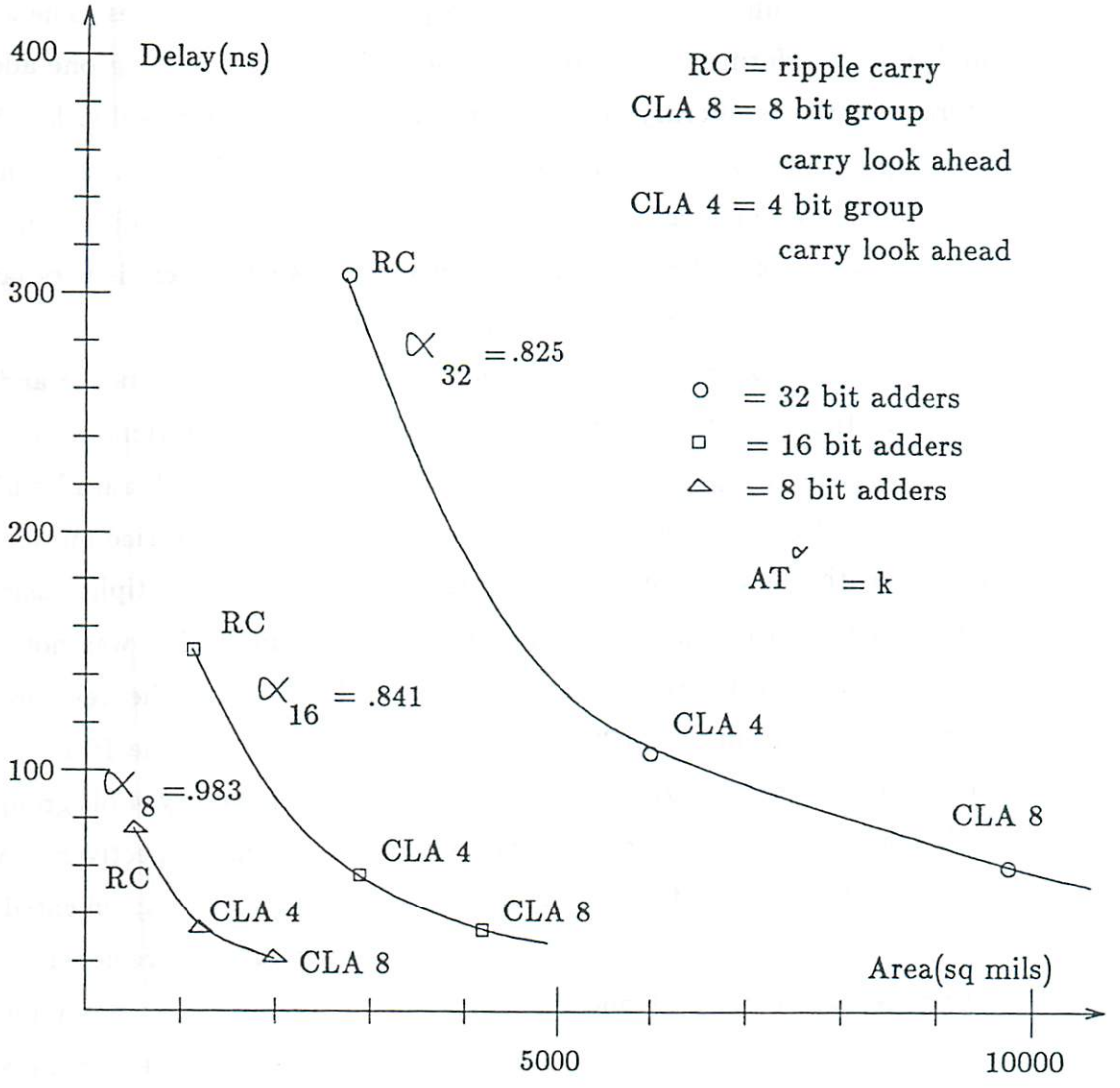


Figure 5.3: Adder tradeoffs

5.3.1.2 Tradeoffs for multipliers

A similar experiment was conducted for multipliers. Figures 4.7, 5.4, and 5.5 depicts different ways of implementing multipliers. There are two broad categories of multipliers, the add shift types and the array types. The add shift multipliers perform multiplication by successive additions using one adder and storage of the temporary results. Several variations are possible by changing the implementation of the adder as presented in Section 5.3.1.1. The array multipliers are much faster since they perform most of the additions in parallel using an array of adders. The area complexity of such adders is very large and increases very rapidly with bit width.

Figure 5.6 shows the area delay tradeoffs for multipliers 4, 8 and 16 bits wide. In the case of the 4x4 multipliers, three implementations were laid out, add shift using a ripple-carry adder, a 4 bit group carry-look-ahead adder, and an array multiplier. The same set of implementations was carried out for 8x8 bit widths, with one additional implementation of the array multiplier using carry save adders as the final stage. The 8x8 bit array multiplier was not actually laid out. Instead its area was estimated using PLEST and the delay estimated using the data from the MP2D 3μ CMOS cell library. For the 16 bit family of multipliers, we synthesized add shift circuits with ripple-carry, 4 bit group carry-look-ahead , and 8 bits group carry-look-ahead adders, respectively. Another variation of the add shift types was to have the 16 bit adder implemented in four groups of 4 bit carry-look-ahead blocks whose carry outputs are generated using a third level of carry-look-ahead circuit. Finally, the 16x16 bit array multiplier was not laid out; the estimates were obtained in the same way as for the 8x8 bits array multiplier.

5.3.2 Exploring the RT design space

Exploring the whole RT design space is a difficult task. Given one possible RT-level implementation of a set of specifications, one can generate a whole array of other implementations by incrementally changing or reconfigur-

Figure 5.4: Four bit array multiplier implementation

Figure 5.5: Eight bit multiplier implementation using four bit array multipliers

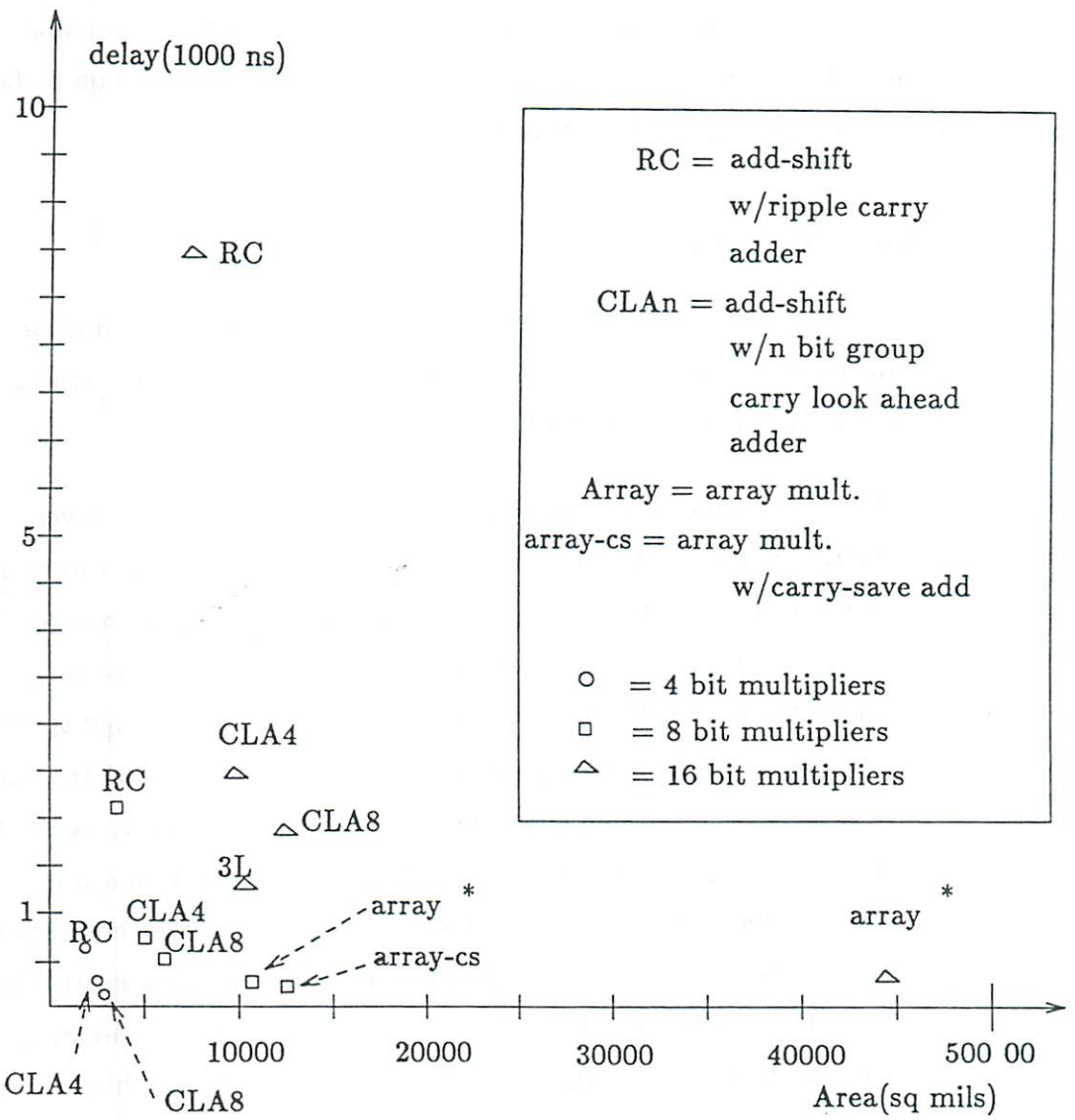


Figure 5.6: Multiplier tradeoffs

ing any part of the design. Compound changes may even make the number of variations combinatorially explosive. An example of compound changes was presented in the previous section, where the different implementations of an adder do compound those of the add-shift multiplier. Attempting to find a general characterization of the design tradeoffs in hindered by the fact that in general the designs to be synthesized tend not to have much in common as far as their structure and target functionalities are concerned.

5.3.3 Assumptions

Faced with this difficult task, there are several assumptions that one can make which would make the problem at hand more or less tractable. Some of these assumptions are the following:

Design space dimensions: The design tradeoffs involve several *figures of merit*, such as cost, speed, and power. These translate to dimensions when one is examining the design space. The more dimensions one considers, the worst the problem complexity will be. To reduce the problem complexity, we only consider two dimensions in dealing with the design space, *cost* (or *area*) and *speed* (or *delay*). In fact, if we look carefully at the three major dimensions in a design space, i.e. cost, speed, and power, we see that they are not really orthogonal. In general, the faster a design implementation is, the more power it is expected to consume. Conversely, serial designs, which are slower and more compact in size, tend to consume less power than their faster and larger counterparts consume. However, by reducing the design space to only two design variables, it is possible to characterize the design tradeoffs in a more or less complete fashion.

Inferior designs: When looking at possible design implementations, one would observe that there is a fraction of these implementations which are non-optimal, or *inferior* [HP83]. A design implementation is inferior when there exists at least another implementation which performs better in one or

more figure of merit, all other figures of merit being equal. So, for example, take two designs which have the same speed and power consumption. If one is costlier (in area) than the other, it is considered inferior, since common sense always dictates choosing the cheaper design. Therefore, only non-inferior designs will be considered when exploring the design space.

Design space boundaries: Even with the two assumptions discussed above, the design space exploration is still a formidable task. Exploring the design tradeoffs for all implementation possibilities requires the user (or the automatic design process) to tackle a multitude of details in every implementation. Such exploration will undoubtedly take a lot of time and computation which may well defeat the purpose of design prediction and evaluation. Instead of exploring the whole design space, we propose to explore only the design space *boundaries*. In other words, we look at bounds on design performance, in terms of both time and area. This, in fact, reduces the design space exploration task to evaluating two design implementations: the cheapest and the fastest implementations. This is depicted in Figure 5.7. Since we are looking only at the cost-speed tradeoffs, we need only look at lower bounds on these two dimensions. Furthermore, because we are discarding inferior designs, we need only look at the cheapest and the fastest implementations. Any implementation slower than the cheapest one will be as costly or costlier and hence inferior to it. In the case of the fastest design, one may have to consider more than one implementation, depending on whether the implementation is pipelined or not, since time performance measures are different for the two cases. The time performance of a pipelined design is usually in terms of its data throughput: either the number of initiations per unit time, or its inverse, the average time between two consecutive initiations. In the case of non overlapped designs, a more conventional unit of measurement is the average total delay time through the hardware. Hence the predictions for the fastest design

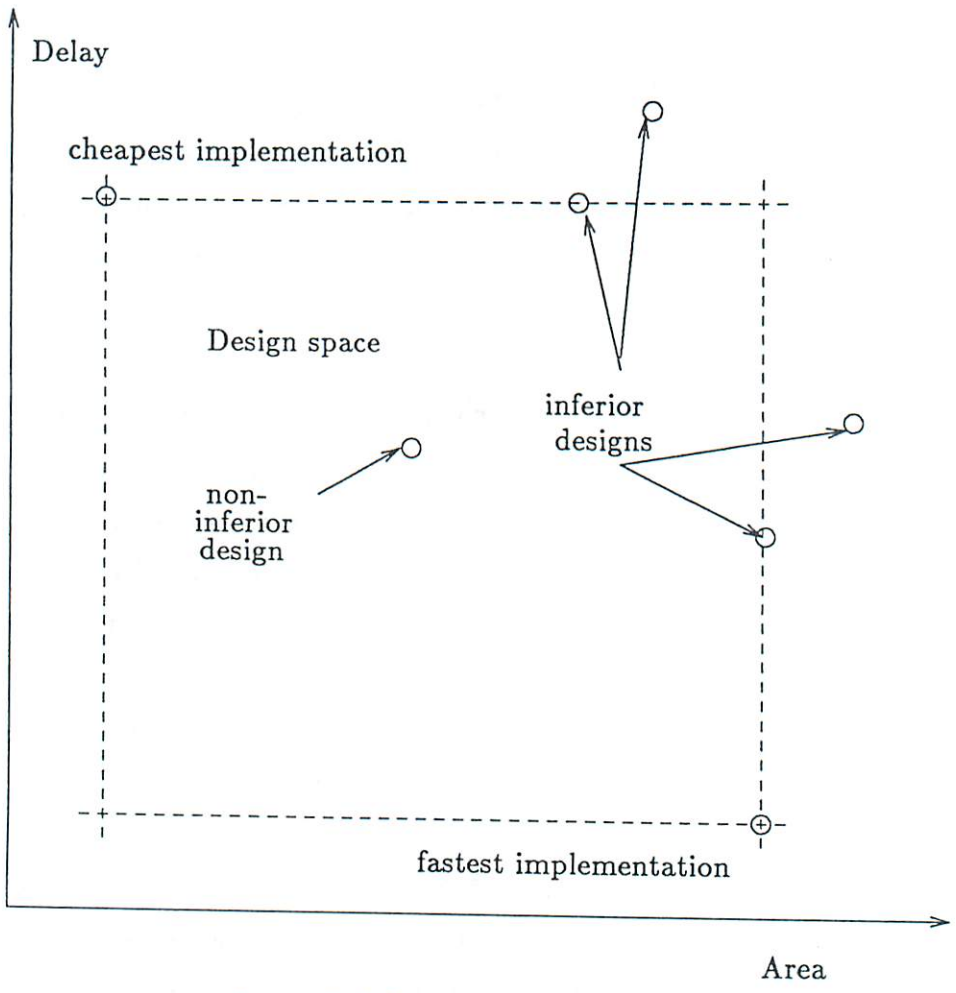


Figure 5.7: Design space boundaries

may be different depending on the design style, a decision which must be taken before the prediction task is to be initiated.

5.4 Estimating the functional requirements of RT-level designs

In register transfer designs, the main components that are used can be divided into three categories:

Operators: These include all the modules that operate on data values. They are generally combinational circuits, but may also be sequential. Examples of these operators include adders, comparators, and multipliers.

Registers: These include all the modules that are used to store data values. They are sequential circuits. Registers, static and dynamic, and flip-flops fall into this category.

Routers: These include all modules that are used to route data values from one place to another. They are usually combinational circuits. Examples include multiplexers, demultiplexers, and buses (if supported by the design style).

When looking at RT level designs, one observes that operators are usually the most influential items in the total cost estimate because, in addition to contributing a large portion to the total cost, they have a direct effect on the routing, register, and router costs. However, this does, by no means, imply that the register and router costs are negligible. It may only indicate that, when attempting to optimize design cost and performance, it is, in general, more advantageous to focus on optimizing the operator cost since it is expected to represent a first order effect on the total cost.

5.5 Lower bounds on functional cost of a design

In this section, we will look at the problem of finding some lower bounds on the cost of a design. When trying to embed as much functionality as possible in a predetermined silicon area, the designer usually attempts to reduce the operator count, since, as discussed in the previous section, this is the first order effect on cost. Reducing operator cost can be achieved by sharing one operator by several operations. So, if a dataflow graph contains two add operations, both can use the same physical adder (at different times, of course), and hence the operator cost is cut by half. This is known as *operator sharing*. Figure 5.8 depicts

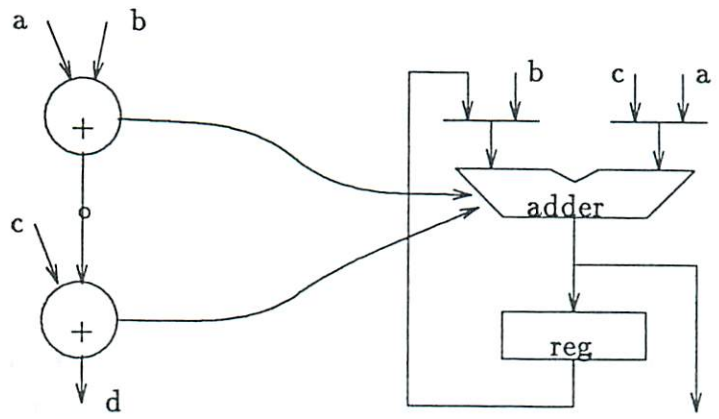


Figure 5.8: An example of operator sharing

an adder shared by two add operations. There are disadvantages (or tradeoffs) associated with sharing operators: first, the time performance (or speed) of the implementation is usually reduced because the design is being serialized (i.e. the two add operations could have occurred simultaneously, had we had two adders, and hence the speed would have roughly doubled). Second, sharing may require adding some extra logic to store the temporary values, in addition to multiplexing the inputs of the operations sharing the operator and distributing their outputs. These tradeoffs are depicted in Figure 5.9. Since trading off operators for registers and routers is, as discussed earlier, advantageous, the only remaining disadvantage is speed reduction. One can recognize some examples of highly serial designs in the microprocessors available currently. Here the operator cost is minimized, usually down to one ALU, there are a large number of registers for storage of temporary variables, and there are many routers (buses or muxltiplexers). One reason for serializing such a design is for it to fit on one chip[1].

From the brief discussions above, one can conclude the following

- Implementations with minimal operator cost tend, in general, to have minimal total cost,

[1] Ironically, having the design on one chip may indeed help increase the time performance, since it avoids distributing the logic on several chips and hence avoiding performance degradation due to inter-chip communications. This is usually a second-order effect when compared to serial versus parallel operations.

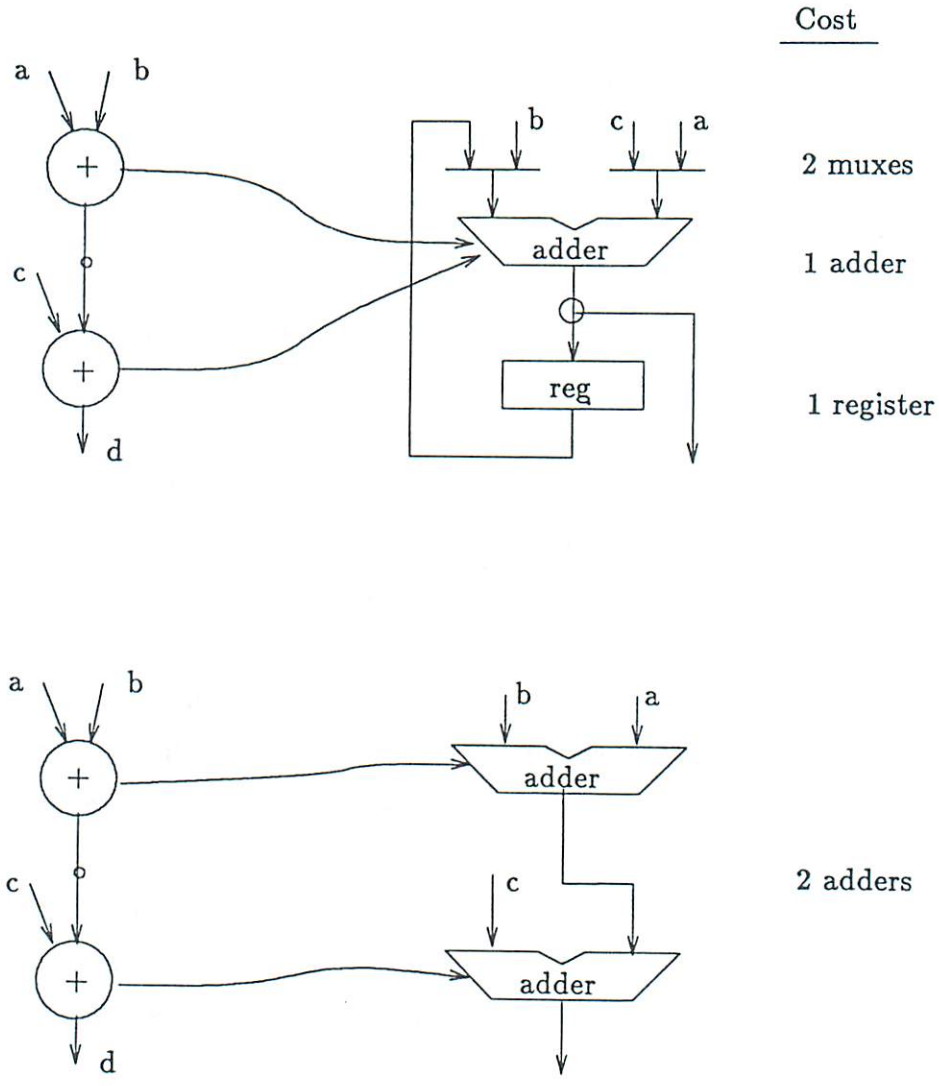


Figure 5.9: Storage of temporary values

- Minimizing operator cost is usually associated with increasing the number of registers and routers.

In the following sections, we will discuss the tasks of estimating bounds on costs of operators, registers and routers for highly serial designs.

5.5.1 Lower bounds on operator cost

In order to minimize operator cost, one would attempt to have the dataflow operations share as few operators as possible. This, as discussed above, is achieved by having several operations share an operator. Obviously, several operations of the same type can always share an operator which is capable of implementing that operation. In some cases, however, more sharing is possible by using operators which can implement several types of operations. An example of this kind of operators is the ALU, which can perform most of the basic arithmetic operations. Generally speaking, the cost of such operators is less than the sum of costs of the individual operators of each type of operation it can perform. If we take, for example, an n-bit adder and add n XOR gates at one of its inputs, we can make it perform both add and subtract operations at a cost less than that of an adder *and* a subtractor. Since, in a minimal cost operator set, we only need to implement each operation type once, the general problem of minimizing operator cost can be stated as follows:

Definition 5.2 (Minimizing operator cost) *Given a set of operation types,*

$$\mathcal{A} = \{a_1 \ldots a_n\}$$

and a set of operator modules,

$$\mathcal{O} = \{o_1 \ldots o_m\}$$

associated with each o_i are a type set $P_i \subseteq \mathcal{A}$ and a cost C_i, find a subset $\mathcal{Q} \subseteq \mathcal{O}$ such that

$$\bigcup_{o_i | o_i \in \mathcal{Q}} = \mathcal{A}$$

and,

$$C = \sum_{o_i | o_i \in \mathcal{Q}} C_i \quad \text{is minimized}$$

Stated as such, the problem is identical to the switching theory problem of finding a minimal complete cover of prime implicants for a Boolean function, a problem which has been shown to be *NP-complete.* In fact, even if the cost minimization requirement is removed, the problem is reduced to the set covering problem, which is also *NP-complete.* This means that, in the general case, the solution of such a problem is time-expensive, since the solution space is expected to grow exponentially with the problem size. There are several factors, however, that make the problem tractable:

- There are a number of excellent heuristics that were developed for the identical minimal complete cover problem by switching theorists. Such heuristics are usually time efficient, and get very close to, if not exactly, the optimal solutions. An excellent presentation and comprehensive discussion of many of these algorithms and heuristics is found in [Bre72].

- The set of all solutions is of size 2^m (set of all possible subsets of $\mathcal{O}$), m being the set of candidate modules . In real designs, m is small compared to the size of the set of operations. This means that bounded exhaustive search over the solution space may well be feasible and, of course, would yield an optimal solution.

- In many cases, some operations can be implemented by only one module; this make that module an *essential module* (borrowing the switching theory terminology), since it *must* appear in the solution. This helps to reduce the search space. Let $\mathcal{E}$ be the set of essential modules, then the set of modules to be searched is reduced to $\mathcal{S} = \{s \mid s$ does not implement any operation in $\mathcal{E}$ *and* s implements some operation in $\mathcal{A}\}$. In the extreme case, where all modules are essential modules, the solution reduces to a linear selection of n modules implementing the n operations.

Thus, the problem of operator cost minimization would, in most cases turn out to be a tractable task. As the operator cost is decreased, the register cost is increased. In the next section, we look at the problem of estimating register cost in highly serial designs.

5.5.2 Bounds on register cost

The operator-register tradeoff can be explained as an intuitive notion. Borrowing some terminology from queueing theory, as more operators, or servers are shared by operations, the values associated with these operations (or customers) cannot be processed (serviced) right away and hence must be stored (queued). To insure that no value is lost, there must be, at any time, as many registers available as there are values which must be stored. The number of registers needed is determined by the maximum over time, of the number of values to be stored. The smaller the number of operators (servers), the larger the number of values waiting to be processed would be, and hence the larger the number of registers needed. So one would expect the number of registers needed to be largest when the number of operators is smallest. Thus, associated with a lower bound on operator cost, one must estimate an upper bound on register cost. In the extreme case, where the design is maximally serial, a good approximation to reality would be to assume that *all* values will be stored at some point (or points) in time (this is particularly true in the case of microprocessors, where every intermediate value will be stored either in registers or in memory). Given a dataflow graph representation of a design, the following definition and theorem provide an upper bound of register cost for any implementation.

Definition 5.3 *Given a dataflow graph, $G = (V, E)$, a cut set S is a subset of edges (or values) $S \subseteq E$ such that*

- *no two edges in S lie on the same directed path from source to sink, and*
- *when removed from E, the graph is disconnected into two components.*

Theorem 5.1 *Given a dataflow graph representation of a design, the maximum number of registers needed for storage is given by the value of the maximum cut in the graph.*

Proof: Let C be the value of the maximum cut set in the graph, and let R be the maximum number of values which need to be stored at any given time. Two values which are on the same path are dependent and therefore cannot be stored simultaneously. Thus the set of values to be stored must form or be part of a cut set. Since the value of any cut set cannot exceed C, then we must have $R = C$ □

In the theorem above, we assumed that all values (arcs) have the same bit width. Actually, the theorem still holds if we associate with each value a bit width. The cut set becomes the sum of the bit width of its values. The quantity that is bounded becomes the total bit width of the registers, in other words, the total number of flip-flops.

Now that we have established a bound on register cost, the next step is to obtain a value for such a bound. The maximum cut problem is usually encountered in network flow problems. In the general case, finding a maximum cut in a graph is known to be an *NP-complete* problem [GJ79], so one would expect such a task to be of exponential complexity. Fortunately, however, if the graph under consideration is a PERT[2] type graph [Eve79], then the problem complexity becomes polynomial in time. PERT graphs are network type graphs in which every edge is on some directed path from the source to the sink. This is precisely the definition of dataflow graphs as we presented it in Section 5.2, where no loops are allowed in the graph. This means that we can indeed find a maximum cut in a dataflow graph in a tractable amount of time. The following algorithm presented in [Eve79] is borrowed from network theory and slightly modified to fit our purposes. The given graph is modeled as a *network* where each edge e is assigned a *flow* and a lower bound on the flow, $b(e)$, such that $b(e) \leq f(e) \leq \infty$. By corollary of the basic theorem in network flow theory, the *maximum cut* in the graph is the cut set for which the *flow is minimum*,

[2] Project Evaluation and Review Technique

i.e. gives us a lower bound of flow in the graph. Finding the minimum flow in a network is achieved by applying any of the algorithms used for solving the well known *maximal-flow, minimal-cut* problem, this time attempting to find a maximum flow from the *sink, t* to the *source, s* instead of the reverse, as in the normal case.

Algorithm 5.1 (Max Cut)

Input : *Dataflow graph* $G = (V, E)$

Output : *Maximum cut in graph* G

Step 1:*(Find legal initial flow)*

for each edge $e \in E$
{
$f(e) \leftarrow 0;$
$b(e) \leftarrow 1;$
}
for each edge $e \in E$
{
trace a directed path P_e *from s to t passing through e;*
for each edge $e_i \in P_e$
$f(e_i) \leftarrow f(e_i) + 1;$
}

Step 2:

Find Maximum flow *from t to s; the resulting cut is* maximal

Even estimates the complexity of this algorithm in the following way. Step 1 can be performed in time $O(|V|.|E|)$. For Step 2, we can use Dinic's Algorithm [Din70] which has a time complexity of $O(|V|^3)$. This yields a total complexity of $O(|V|^3)$ assuming that $|E| \leq |V|^2$.

As presented, the algorithm assumes that all edges have equal capacities, or bit widths in our case. In order to handle edges (values) of different bit widths, we associate with each edge e_i a positive integer $bw(e_i)$ representing the bit width of value e_i. Thus, the algorithm is slightly modified. In specific, Step 1 is modified in the following manner:

Step 1 (modified):

for each edge $e \in E$
{
  trace a directed path P_e from s to t passing through e
  $P_e = e_1 e_2 \ldots e \ldots e_{n_e}$;
  for each edge $e_i \in P_e$
    $b(e_i) \leftarrow \max_{e_i \in P_e}\{bw(e_i)\}$;
    $f(e_i) \leftarrow f(e_i) + b(e_i)$;
}

This modification makes the flow in each path lower bounded by the maximum bit width over all its edges and thereby making the resulting maximum cut a weighted sum of the capacities of its edges.

Figure 5.10 shows an example dataflow graph. The edge labels represent the bit widths. Figure 5.11 shown the paths and the network modeling of the graph, along with the maximum cut. Figure 5.12 shown a scheduling of the graph which results in register requirements equaling the predicted maximum.

5.5.3 Bounds on router cost

The amount of data routers is expected to increase as more operator modules are shared: more values have to be multiplexed at the inputs to operator modules, and more output values have to be routed to storage registers. As in the case of registers, we are interested in finding an upper bound on the number of routers required for data routing. In the following, we shall assume that all

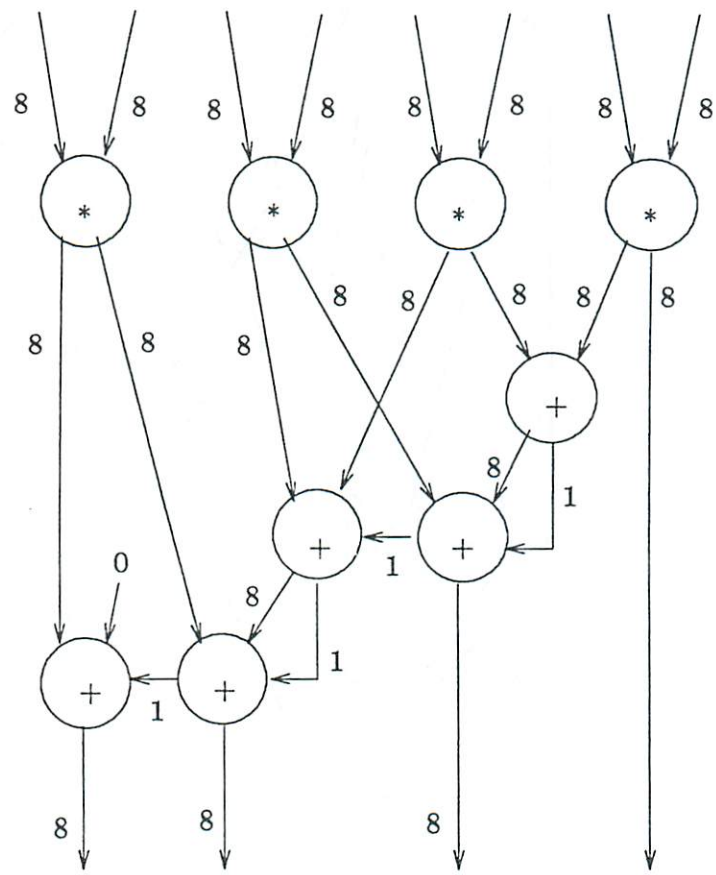
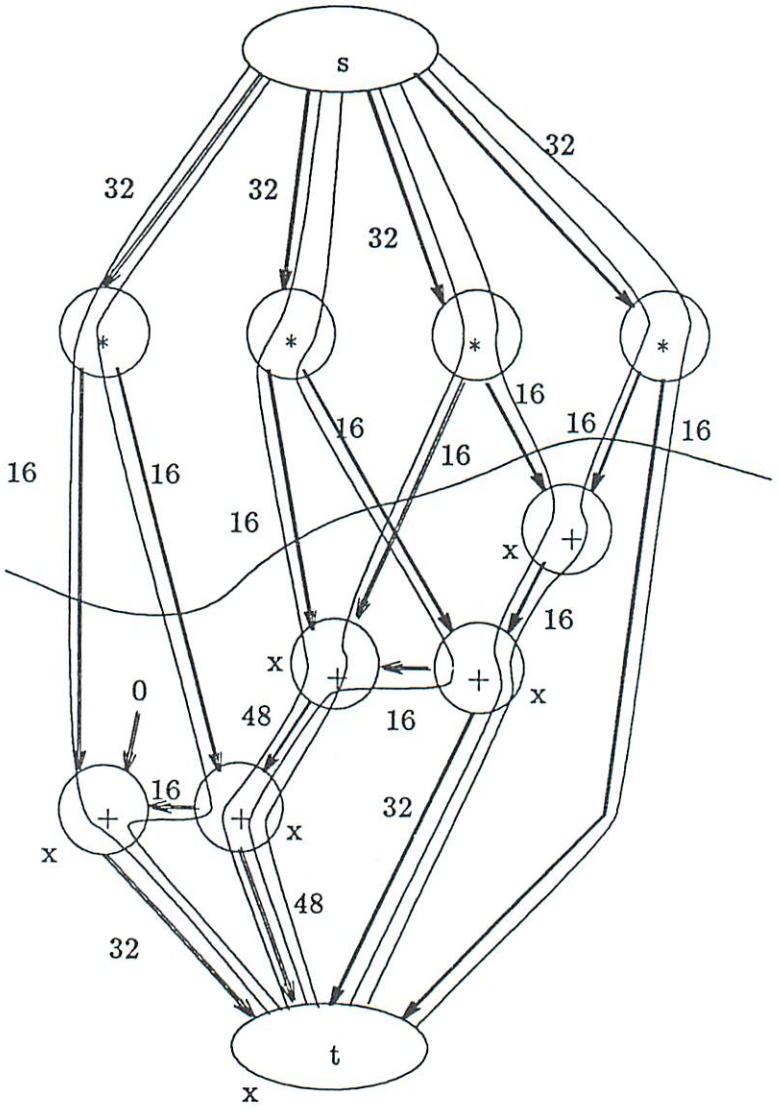


Figure 5.10: Data flow example

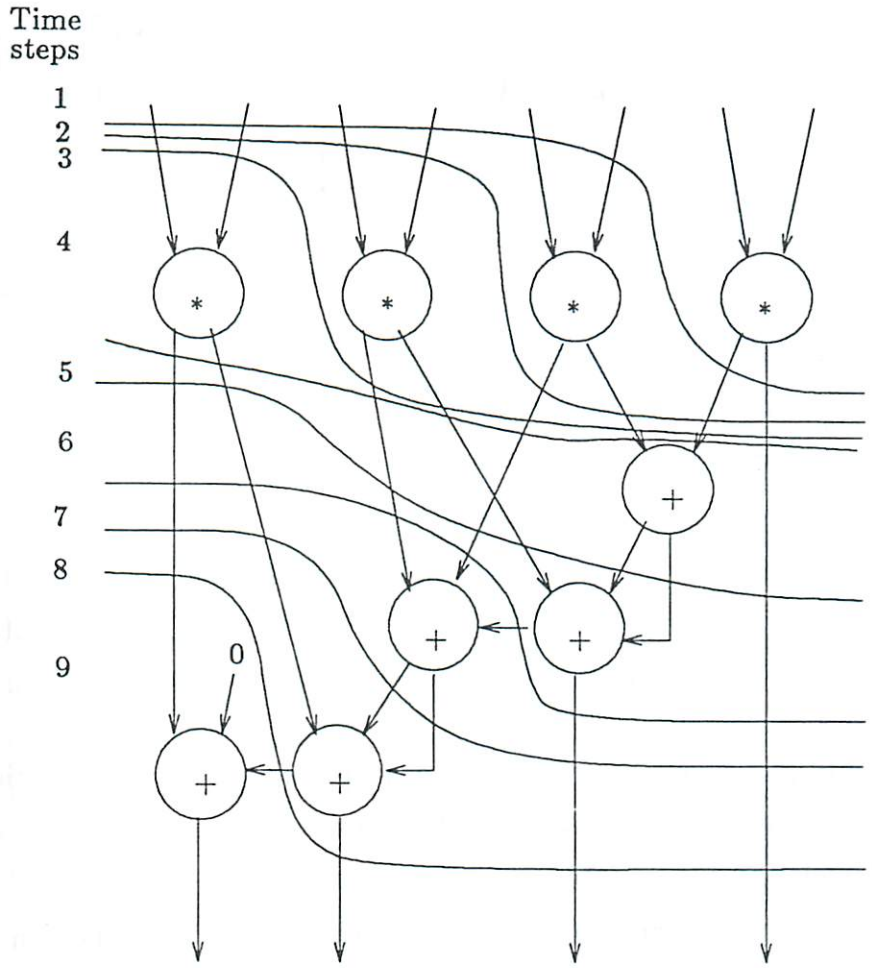


Min flow in all arcs = 16

x = labelled nodes

Max Cut = 8 regs

Figure 5.11: Maximum cut through the graph



Hardware available = 1 ALU

Max # regsisters = 8 at
Time steps 2,3,4,5

Figure 5.12: Scheduling with the maximum number of registers

Figure 5.13: Multiplexors and wired fanouts

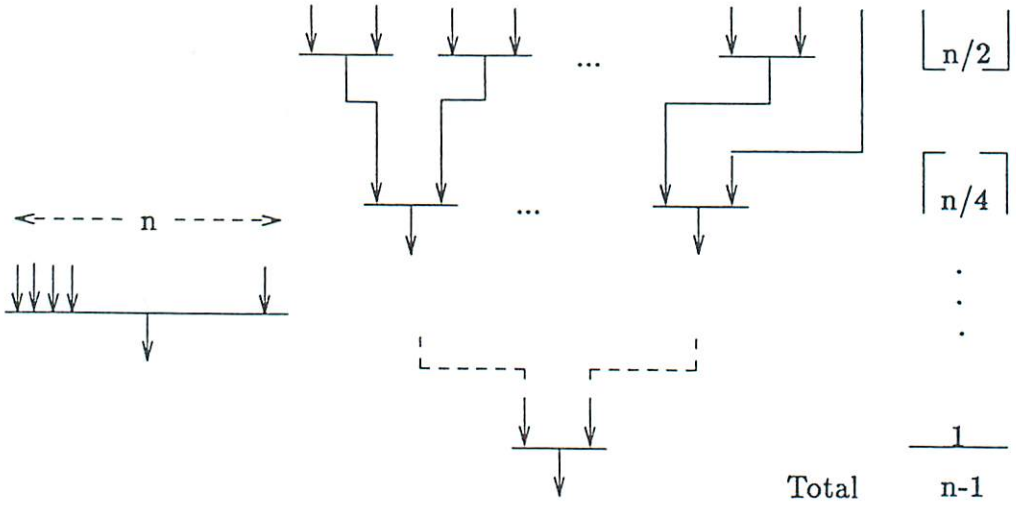


Figure 5.14: Using 2 to 1 muxes to build n to 1 muxes

routing of values at inputs is done using *multiplexors*. Distribution of outputs to various places is done using simple *wired fanout* circuits. These assumptions are depicted in Figure 5.13.

In estimating the routing cost, we shall only be concerned with the cost of multiplexors (or muxes, for short). A multiplexor is characterized by the number of inputs it multiplexes, hence we talk about an *n to 1* multiplexor as being a router of n values to one destination (such as an input to an operator). The bit width of the data values determines how many *n to 1* multiplexors of one bit each are needed. Using *2 to 1* muxes as basic building blocks, a one bit *n to 1* mux can be constructed with *n-1* such blocks as shown in Figure 5.14. So a p bit *n to 1* mux can be constructed using $p(n-1)$ *2 to 1* muxes of one bit each. Thus, the cost of muxes can be normalized to the number of *2 to 1* basic muxes it contains. In an RT-level circuit, there are usually two types of multiplexing: at the input of operator modules being shared by several operations, or operator

muxes, and at the input of a storage element being shared by several values, or register muxes. These two types are shown in Figure 5.15.

Once the minimal operator cover for a set of operations in a dataflow graph has been obtained, the number of operator muxes is estimated by simple counting. The following theorem gives an upper bound on the operator multiplexing cost of a dataflow graph assuming a minimal cost operator allocation.

Theorem 5.2 *Given a dataflow graph $G = (V, E)$, associated with each edge e is a bit width $b(e)$, a set of operators S, and an operator cover $c : V \Longrightarrow S$, then the total number of 2 to 1 multiplexors for data is upper bounded by $U - B$, where*

$$U = \sum_{e \in E} b(e)$$

$$B = \sum_{i \in S} w_i$$

where w_i is the sum of the bit widths of the inputs to module i.

Proof: Let V_i be the set of operations sharing module i. We will need at most $|V_i| - 1$ multiplexers at each input to module i (assuming that none of the values multiplexed are identical). The upper bound on the muxes cost can be estimated by summing the multiplexing cost over each input of each operator, and the above expression results. □

An upper bound on the multiplexing cost of the register muxes can be estimated using the following theorem.

Theorem 5.3 *Given a set of n registers of p bits each, and a set of m values of p bits each, the total register multiplexing cost is upper bounded by $p(m - n)$.*

Proof: Let $a_1, a_2, \ldots, a_n$ be the number of values sharing registers $1, \ldots, n$, respectively. The cost of register multiplexing for register i is at most (assuming none of the values sharing it are identical) $p(a_i - 1)$. Thus, the total cost of register multiplexing is given by

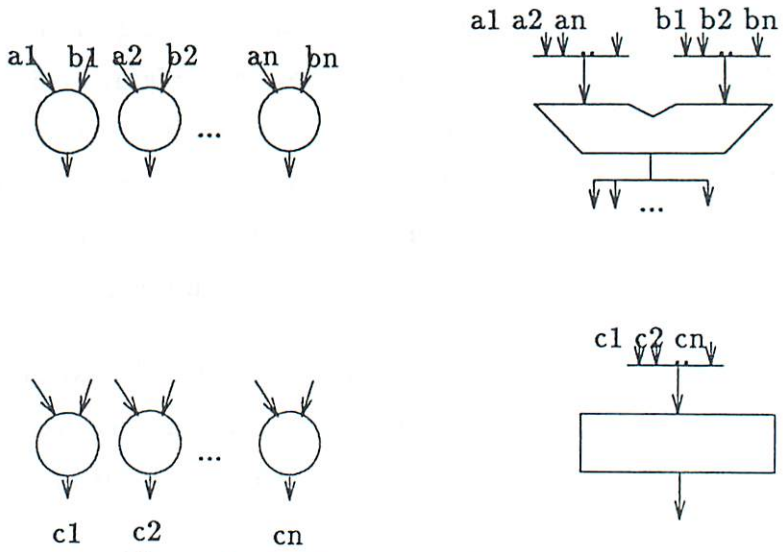


Figure 5.15: Two types of multiplexing

$$RM = \sum_{i=1}^{n} p(a_i - 1)$$

the sum of the a_i's is actually m and hence the bound follows. □

Having obtained estimates for operator, register and routing costs, the total functional cost estimate for the cheapest design is obtained by summing these costs. Next we turn our attention to the other end of the design space and look at estimating the cost of the fastest design.

5.6 Upper bounds on design cost

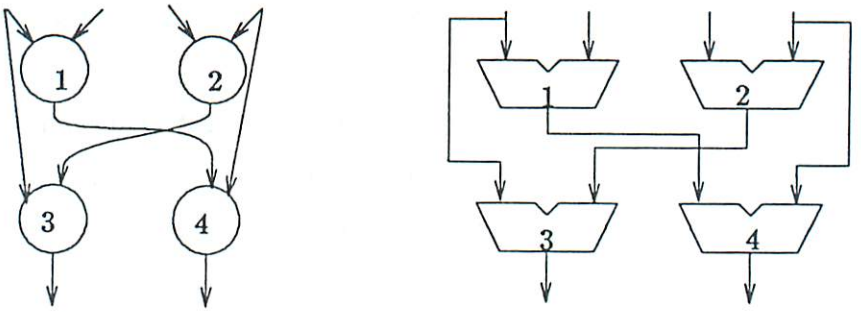
Since we are focusing our attention on non-inferior designs, the cost of the fastest design implementation actually represents an upper bound on the design cost. Estimating the cost of such a design implementation turns out to be a much easier task than that of estimating the cost of the cheapest design. As discussed in Section 5.3.3, there can be two implementations referred to as the fastest, depending on the design style. If the design style is non pipelined, then the fastest implementation is the one with the smallest propagation delay. To minimize propagation delay, every operation is performed by a separate operator.

In other words, none of the operators are shared, in which case no registers are needed to store the values generated. Thus the cost of the design is simply the sum of the operator cost.

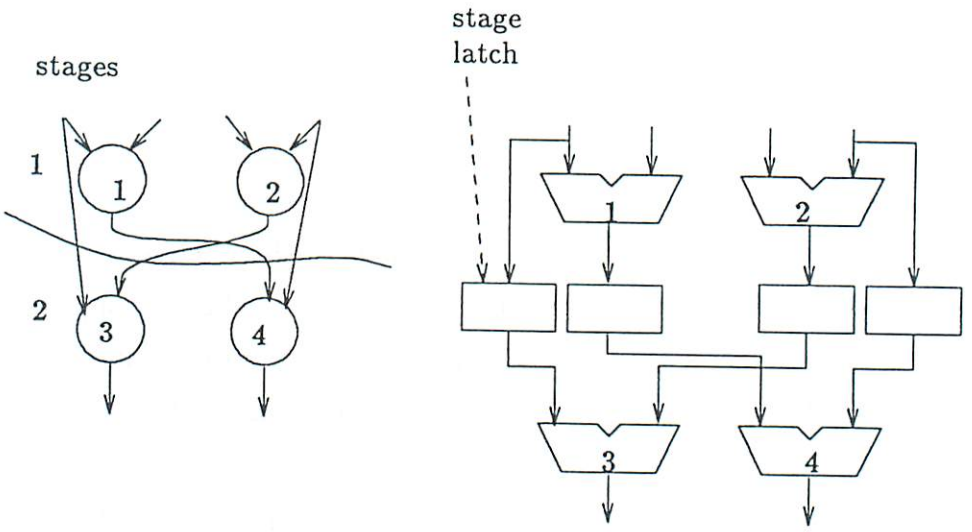
In the case of pipelined designs, the fastest design in the one with the highest throughput, or number of initiations per unit time. The highest initiation rate corresponds to pipelines with a latency of one, i.e. one initiation every clock cycle. In his thesis [PP86], Park provides estimates of operator cost for pipelined designs for any latency L. In fact one can also intuitively reason that, when new data sets are input at each clock cycle, all operations in the dataflow graph must be active all the time, and hence no sharing can be possible. This is a similar situation to the fastest non-pipelined implementation with the difference that we now need latches to hold data between pipeline stages. The latch cost for each stage is equal to the total bit width of the values output to the next stage. The total latching cost is the sum of latching costs of all the stages. Figure 5.16 shows fastest design implementations of a dataflow graph, both pipelined and non-pipelined.

5.7 Summary

This chapter dealt with the task of predicting the cost of a design implementation. The design is specified in a high level (behavioral) desciption in the form of a dataflow graph. Generally, there is not jut one solution, but a *solution space* consisting of the possible implementations of the design specification. Such a solution space is usually bounded by the cheapest (most serial) design and the fastest (most parallel) design. The cost of an implementation is composed of the costs of the operators, the registers and the routers. The problem of finding the lowest operator cost is identified and shown to be identical to a well known problem in switching theory. As the number of operators is minimized more registers and routers (muxes) are needed. An algorithm for finding an upper bound on the number of registers is proposed along with an example. Upper bounds on router cost are also presented. The fastest design implementation would result when



fastest non-pipelined implementation



fastest pipelined implementation

Figure 5.16: Pipelined and non-pipelined fastest designs

every operation in the dataflow graph is implemented by a separate operator. No registers or routers are needed as none of the operators are shared. If the design style is pipelined, the one must add the cost of pipeline stage latches to the operator cost of the fastest design implementation.

Later in the design process, once the scheduling of operations and their allocation to operators is performed by the automatic synthesis programs, the cost of operators is known. What remains is to estimate the cost of registers. This problem is the subject of the next chapter.

Chapter 6

Estimating storage requirements

All perception of truth
is the detection of an analogy.
– THOREAU, *Journal*, Sept. 5, 1851

6.1 Introduction

In the previous chapter we discussed the problem of bounding the cost of RT level designs given their high level specifications. As is the case in almost any process, the more information one gathers as the process proceeds with its task, the more accurate one can expect an estimate of the process output to be. Put in other terms, once the synthesis process has been partially finished, more decisions involving the implementation details have been resolved, and hence more information about the final implementation is available. Thus, an estimation procedure which makes use of such information is expected to be more accurate. In this chapter, we look at estimating the RT level design implementation cost *during* the synthesis procedure. In particular, we address the task of estimating the storage requirements, once the global timing of the design has been decided upon. As we will discover, the problem of storage allocation, in most cases, turns out to be a tractable one, and it is possible to obtain a quick estimate by *construction* (i.e. by actually *performing* the task).

6.1.1 Progression of synthesis tasks

As discussed in the previous chapter, synthesis of RT level implementations from high level specifications is divided into the set of tasks presented in Section 1.1.2. The first task, design style selection, is really a 'strategy' decision and is generally decided upon by the designer before the automated synthesis procedure is started. The reason for this is that design style is dependent partially on factors external to the design specifications. As an example, the decision to use buses or multiplexors in an implementation may depend on the type of interfaces envisioned for the resulting implementation. Buses may limit speed but would certainly increase interface and expansion capabilities. As a result of this situation, the RT datapath automated synthesis tasks can now be broken down into the tasks presented in Section 1.1.2;

1. clocking scheme synthesis,
2. scheduling of operations,
3. operator allocation,
4. mux and bus allocation, and
5. register allocation.

The first task involves deciding upon an optimal clock period and number of phases. Tasks 2 and 3 deal with allocating operations to specific clock cycles and hardware modules to individual operations. They are usually performed together as they exhibit a high degree of interdependence; for example, the number of operations that can be scheduled at any given clock cycle is constrained by the number of hardware modules available and how the operations are allocated to these modules. The same can be said about task 4, whose outcome is dependednt upon the way operators are shared and operations are scheduled. Finally, the last task involves allocating registers to store temporary values. In the high level synthesis tools at USC, MAHA and Sehwa, the first three tasks are automated.

Once a design has been processed by one of these programs, the resulting output is a *partial design* in the form of hardware operators implementing the dataflow operations and a time schedule of these operations. Given a partial design as such, the information about the operator cost is readily available, along with the cost of operator multiplexing. Evaluating such a partial design means predicting the outcome of the last step in the RT synthesis process, i.e. the register allocation. In the following, we look at the problem of register allocation and provide some means for predicting the number of registers needed for a partial design output from the automated synthesis programs.

6.2 The Problem

6.2.1 The Problem domain

Figure 6.1 depicts an example partial design output from a synthesis program. Since the operations have been scheduled in time and allocated to operators, we readily know the times at which each value is created and when it is consumed. The span of time between the creation of a variable and its consumption is referred to as the *lifetime* of the value. Values which are consumed right after they are created need not be stored, except when they are created and consumed by two operations sharing the same operator. Such a situation is shown in Figure 6.2. A value which is not consumed right after creation must be stored in registers or else it will be lost if the operator that generated the value is used during the lifetime of the latter. Two (or more) values which have disjoint lifetimes can be stored in the same register. Deciding which values are stored in which registers is the purpose of register allocation. The problem will be stated in Section 6.2.3; later sections will discuss the problem and outline an approach to solving it.

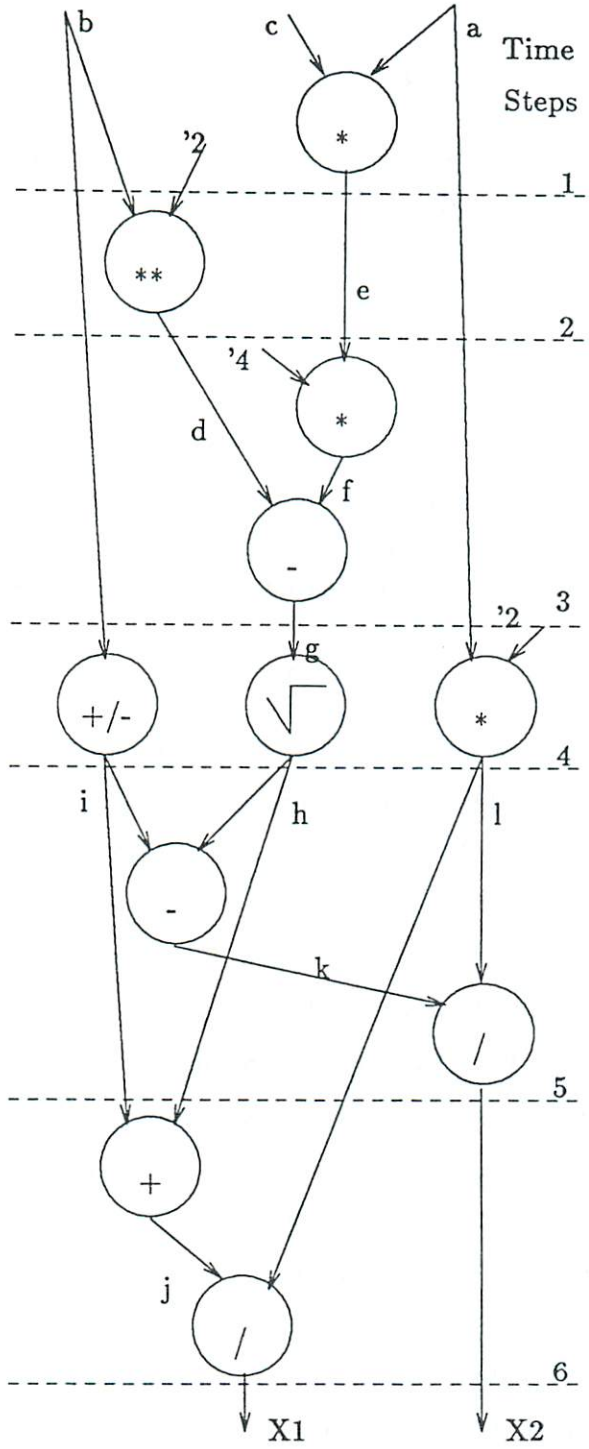
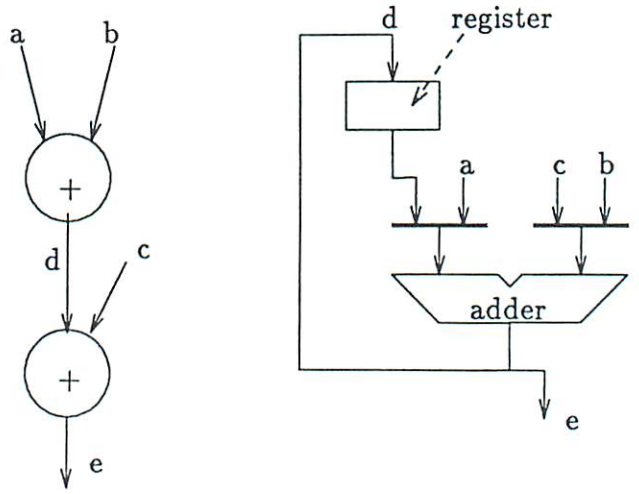


Figure 6.1: A partial design



data flow graph

RT level design

Figure 6.2: Value created and consumed by the same operator

6.2.2 Related research

Register allocation has been performed by many synthesis programs, starting with Hafer [HP78], who attempted to share only temporary registers using heuristics. EMUCS [Hit83] uses a MINIMAX cost criterion and employs an algorithm invented by McFarland. Here, scheduling of operations into time steps is done in advance, and register allocation is performed along with operator allocation. Registers and operators are allocated by choosing the values and operations with the lowest implementation costs, and looking ahead to determine the impact on future costs if allocation is postponed. Sharing across conditional branches is supported, but the algorithm has not been implemented on pipelined designs. No discussion of algorithm performance has been published. ELF [GK84] uses the same method.

A formal approach using clique partitioning has been implemented by Tseng [TS86]. Values become nodes in a graph, and arcs connect values which are never alive during the same timeslot. Values sharing a register will hence belong to the same clique in the graph. Maximal clique partitioning will therefore yield an optimal register allocation. Clique partitioning has been shown to be *NP*-

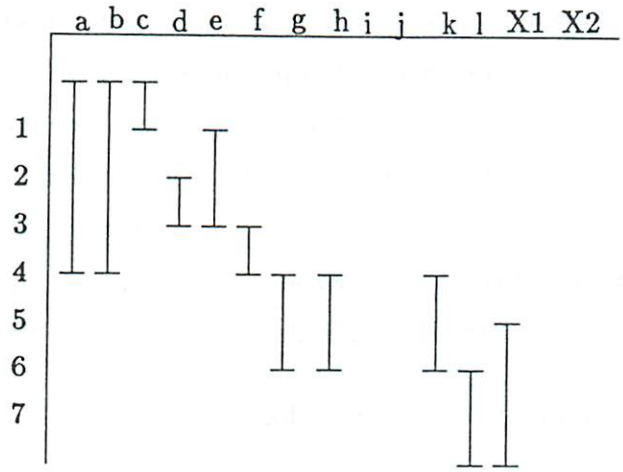


Figure 6.3: Lifetime table for example in Figure 6.1

Complete [GJ79], and thus Tseng's formulation of the problem is *NP-Complete*. However, Tseng implements a polynomial-time solution to the problem which is not guaranteed to be optimal. While the problem in its general form may well be *NP-Complete*, we believe that the specific problem that is treated in this chapter is not *NP-Complete* and can indeed be solved optimally in polynomial time.

6.2.3 The problem statement

Now that the problem domain has been presented and previous related work has been discussed, we turn our attention to presenting the problem itself. We first start by making some definitions in addition to the ones in Section 6.2.1. A value output from an operation is *created* when the operator implementing that operation is finished executing the operation. A value can be either stored for usage at a later time, or can be used in operations immediately following its creation. The last time a value is used as input to an operation is defined as the time of *consumption* of the value. The time span between the creation of a value and its consumption is defined as the *lifetime* of the value. Values are *overlapping* if their lifetimes overlap (i.e. if they can be present at the same time), otherwise they are *disjoint*. Values can be *conditional* or *unconditional* depending on whether or not they are inside a DJ (if-then-else) node. Finally,

the set of lifetimes of all the values in a dataflow graph whose operations have been scheduled and allocated to operators is defined as the *lifetime table.* Figure 6.3 shows a lifetime table for the example dataflow graph from Figure 6.1.

With these definitions in mind, the problem of register allocation can be stated as follows: given a lifetime table generated from a dataflow graph whose operations have been scheduled and allocated to operators, allocate the values in the lifetable to registers such that

1. overlapping values cannot share a register if they are unconditional,
2. overlapping conditional values can share registers under certain conditions (to be defined later),
3. pipelined designs are handled properly, and
4. the number of registers needed is minimized.

In the following section, we present an approach to solving the problem pertaining to dataflow graphs without applying conditions 2 and 3 above. Later we present extensions to handle these two conditions,

6.3 Approach to the problem

The approach to solving the problem hinges around the similarity of the register allocation problem to another problem whose solution has been known and proven to be optimal. In the following section we present the analogy between the two problems. Section 6.3.2 presents an algorithm to solve the register allocation problem based on the analogy.

6.3.1 Analogy

If conditions 2 and 3 are temporarily removed from the problem statement in Section 6.2.3, the resulting statement bears a striking resemblance to a

well known problem in physical layout, which is referred to as the *track assignment* problem. A typical instance of the track assignment problem is shown in Figure 6.4, taken from [HS71].

The goal of track assignment is to allocate the wire segments to tracks so as to minimize the total number of needed tracks. Wire segments can share a track if they do not overlap. For dataflow graphs (dfg's) with no loops or conditional branches and non-overlapping scheduling, the register allocation problem is identical to the track assignment problem as described above. Figure 6.5 depicts this analogy.

Given a lifetime table for a dfg, the goal is to assign values(wires) to registers(tracks) so as to minimize the total number of registers(tracks) needed to store the values. Two values cannot share a register if they overlap in time, whereas two wire segments cannot share a track if they overlap in space. With this in mind, we can solve the register allocation problem by solving the similar track assignment problem. An optimal solution is outlined in the following section.

6.3.2 The left edge algorithm

Essentially, the track assignment problem is solved using the following algorithm, often referred to as the Left Edge algorithm, first proposed by Hashimoto and Stevens [HS71].

Algorithm 6.1 (Left Edge)
Input : *set of wire segments*
Output:*assignment of wire segments to minimal number of tracks*

sort the list of wire segments in increasing order of their left edges
Assign the first segment (the leftmost edge) to the first track
delete the first segment from the list of wires
set current track to the first track
while there are still wire segments to be assigned to tracks

{

repeat

{

Find the first wire whose left edge is to the right of the last selected wire and assign it to the current track.

Delete the wire segment from the list of unassigned segments

}

until no more wires can be assigned to the current track,

start a new track and begin again.

}

It is surprising to notice that the left edge algorithm, in spite of taking a greedy approach, does indeed give optimal results. The proof of optimality is presented in the original paper [HS71]. The analogy of the track assignment problem to the restricted register allocation problem (without conditions 2 and 3 in Section 6.2.3) implies that the Left Edge algorithm can always find an optimal allocation of registers. One note here is that the algorithm was given the name Left Edge because the routing channel where the wire segments were to be assigned was 'drawn' horizontally, wires traveling from left to right. So the left edge of a wire in the algorithm really meant the 'beginning' of the wire. In our case, the left edge would mean the time the value was created. In Figure 6.6, we show an example optimal allocation of the values in Figure 6.3 to registers.

6.3.3 Estimation by construction

In light of the above discussion, it is clear that register allocation is a tractable problem. Furthermore, the problem complexity is low and experiments have shown that a program implementing the left edge algorithm would indeed run very fast. So, when presented with a partial design in the form of a dataflow graph, with operations allocated and scheduled, along with a lifetime table, the task of estimating the amount of storage needed can be accomplished by *actually* allocating the values to registers. This is what we mean by *estimation by*

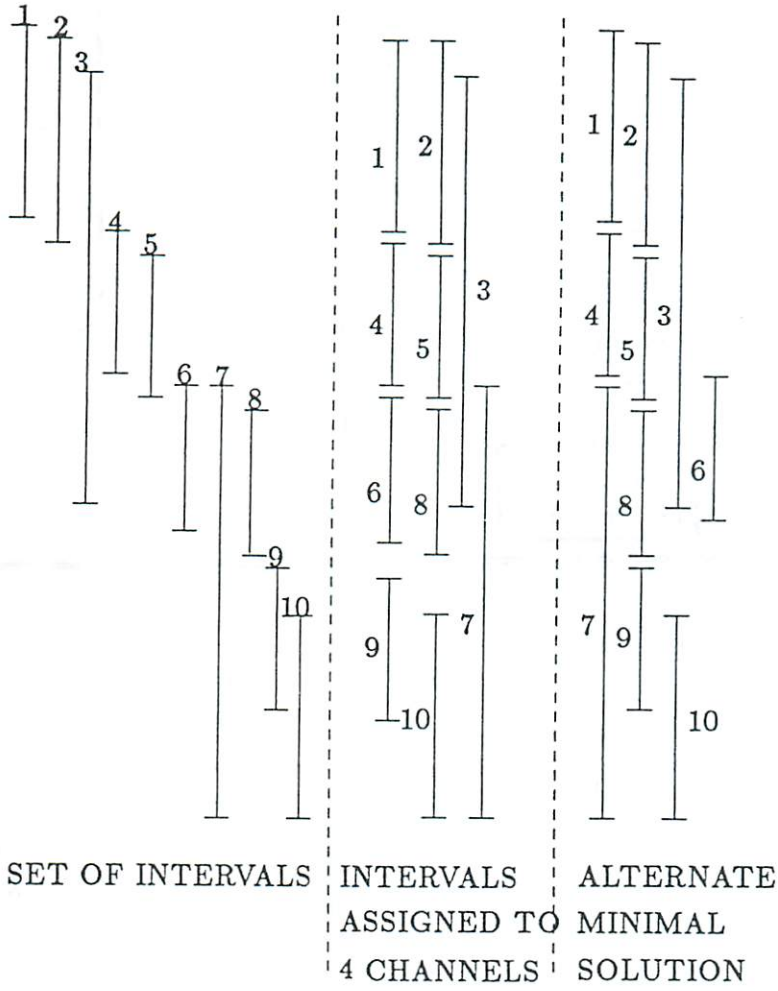


Figure 6.4: The left edge algorithm (from [HS71])

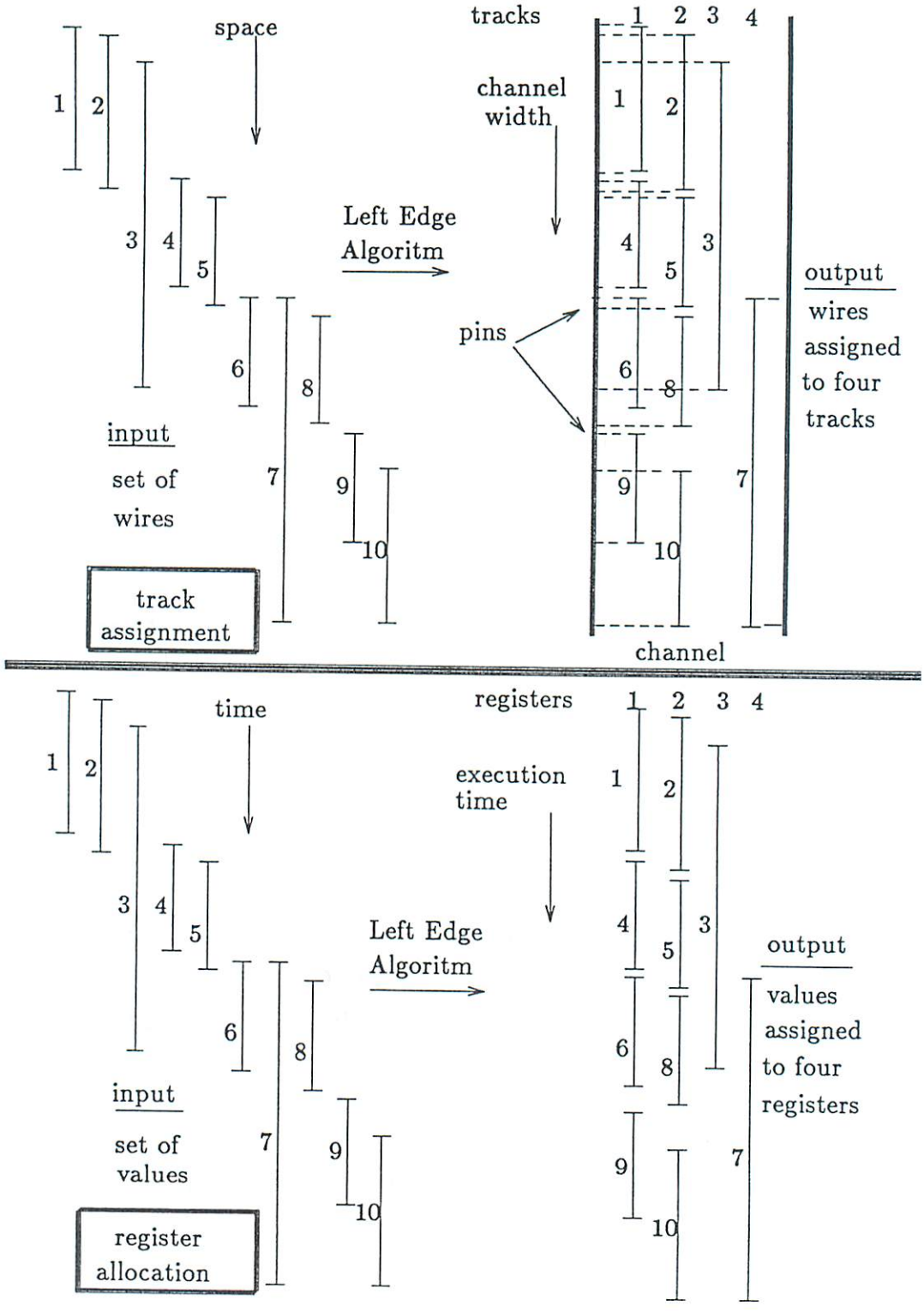


Figure 6.5: Analogy of track assignment to register allocation

```
Number of values = 11
Number of time steps = 8
Number of registers needed = 4
Allocation of values to registers
Register #: value1, value2,...

Register 1 : a ,h ,X1
Register 2 : b ,i
Register 3 : c ,e ,g ,l
Register 4 : d ,X2
```

Figure 6.6: Optimal allocation for lifetime table in Figure 6.3

construction. In Section 6.5 we present some examples which show the speed of the allocation algorithm.

As stated before, the problem being treated so far is a restricted version of the general problem. In the following sections, we extend the scope of the allocation process to cover variable sharing among conditional branches as well as to handle pipelined designs.

6.4 Extensions

6.4.1 Conditional branches

Conditional branching in our dataflow graph model is represented as Distribute-Join (DJ) blocks. The DJ blocks were defined in Section 5.2. Essentially, a DJ block represent a well formed IF-THEN-ELSE construct. An example DJ block is shown in Figure 6.7. One can clearly see that the two branches of the construct (i.e. the IF and the THEN branches) can never occur *simultaneously.* The same must be true for the values in both branches. This means that two values across conditional branches can share a register even if they overlap, since

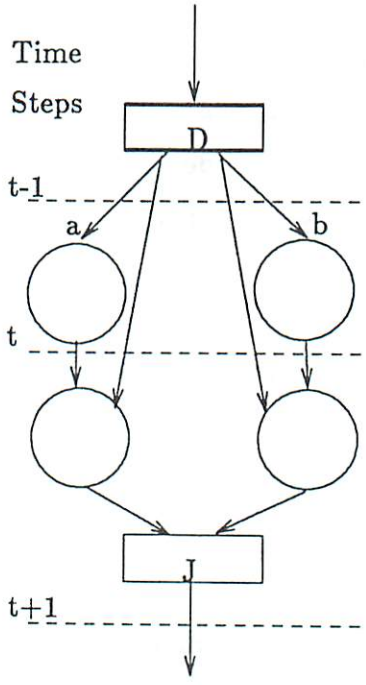


Figure 6.7: An example Distribute-Join (DJ) block

they can never exist at the same time. The problem of detecting which values bear this kind of relationship to other values may seem easy if we are talking about one DJ block in isolation. The problem become more difficult when we have multiple DJ blocks, and even more so when the blocks are nested. The next section outlines some terminology, Section 6.4.1.2 presents a coloring algorithm modified from [PP86]. Section 6.4.1.3 explains how to use this algorithm to share registers among conditional values.

6.4.1.1 Assumptions

Definition 6.1 *Two values are* mutually exclusive *if they can never exist at the same time.*

It follows from the definition that for two values to be mutually exclusive they must be on different paths in the same DJ block. In other words, one must be on the THEN part, the other on the ELSE part on some IF-THEN-ELSE (or DJ)

block. It follows from that two mutually exclusive values can share a register regardless of whether or not they overlap.

6.4.1.2 Value coloring for mutual exclusion

In his thesis [Par85] Park presented a node coloring algorithm which aims at identifying mutual exclusion between nodes (or operations) in a dataflow graph with the aim of overlapping operations which are mutually exclusive. We modified the algorithm to color values instead of nodes, and thereby to identify mutual exclusion among values. The algorithm is called the Modified Park Coloring, or MPC algorithm.

Algorithm 6.2 (MPC)
Input: *A data flow graph*
Output: *A coloring of values (edges) to detect mutual exclusion*

Traverse the graph from source to sink.
Values are colored according to the following rules:

> *1.All unconditional values (i.e. values not inside any DJ block) are assigned unique colors,* $0, 1, 2, etc....$
> *2.If a value is distributed (i.e. enters a DJ block), assign to its children (i.e. the distributed values) a color composed of the parent color affixed with a unique number, as shown in Figure 6.8.*

The following section describes the modification of the left edge algorithm to allow sharing of registers across conditional branches.

6.4.1.3 Handling conditional branches

Once the values of the dataflow graph have been colored, mutual exclusion between any two values can be checked by using an algorithm presented in Park's thesis [Par85]. Essentially, the algorithm checks the colors of the two

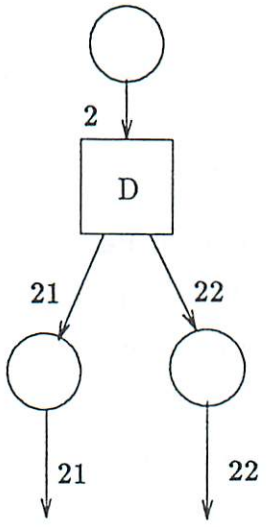


Figure 6.8: Color coding of conditional branches

values and determined whether or not they are mutually exclusive by applying a set of rules. With this in mind, the Left Edge algorithm is modified as follows, (with the values replacing wire segments, registers replacing tracks.)

Algorithm 6.3 (Modified Left Edge)
Input : *lifetime table of a colored dataflow graph*
Output: *assignment of values to minimal number of registers*

sort the value lifetimes in increasing order of their creation times;
Assign the first value (the earliest creation time) to the first register;
set current register to the first register;
while *there are still values to be assigned to registers*
{
  repeat
  {
A: **for***each value i not yet assigned to a register*
      if*the creation time of i >the consumption time of*
      the last value assigned to the current register **then**{
        assign value i to the current register;

```
        Delete value i from the list of unassigned values}
      else{
        foreach value j already assigned to current register
        and overlapping with i
          if i and j are not mutually exclusive (by Park's algorithm),
          (i cannot be assigned to the current register)
          then select the next unassigned value and goto A;
        (i is mutually exclusive with all the values j)
        assign i to current register;
      }
  }
  until no more values can be assigned to the current register,
  start a new register and begin again.
}
```

If we allow mutually exclusive values to share registers as shown in the algorithm, the proof of optimality for the original algorithm may no longer be valid. The problem may or may not become *NP-Complete*, but in all the examples we have tried (some of which are presented later), the algorithm did find the optimal solution, which indicates that even if the algorithm may not be optimal, it will still get very near, if not *to* the optimal solution.

6.4.2 Extensions to pipelined datapaths

Synthesis of pipelined datapaths is one of the main tasks that the ADAM system manages. The pipelined synthesis program, Sehwa, [PP86] takes as input a dataflow graph description, along with some constraints on cost and speed, it selects an optimal latency and produces a scheduling of operations and allocates them to operators. Figure 6.9 shows an example dataflow graph from [Par85]. Figure 6.10 shows a scheduling of operations with a latency of 3. Figure 6.11 shows the overlapping of the execution cycles in time. For example, time

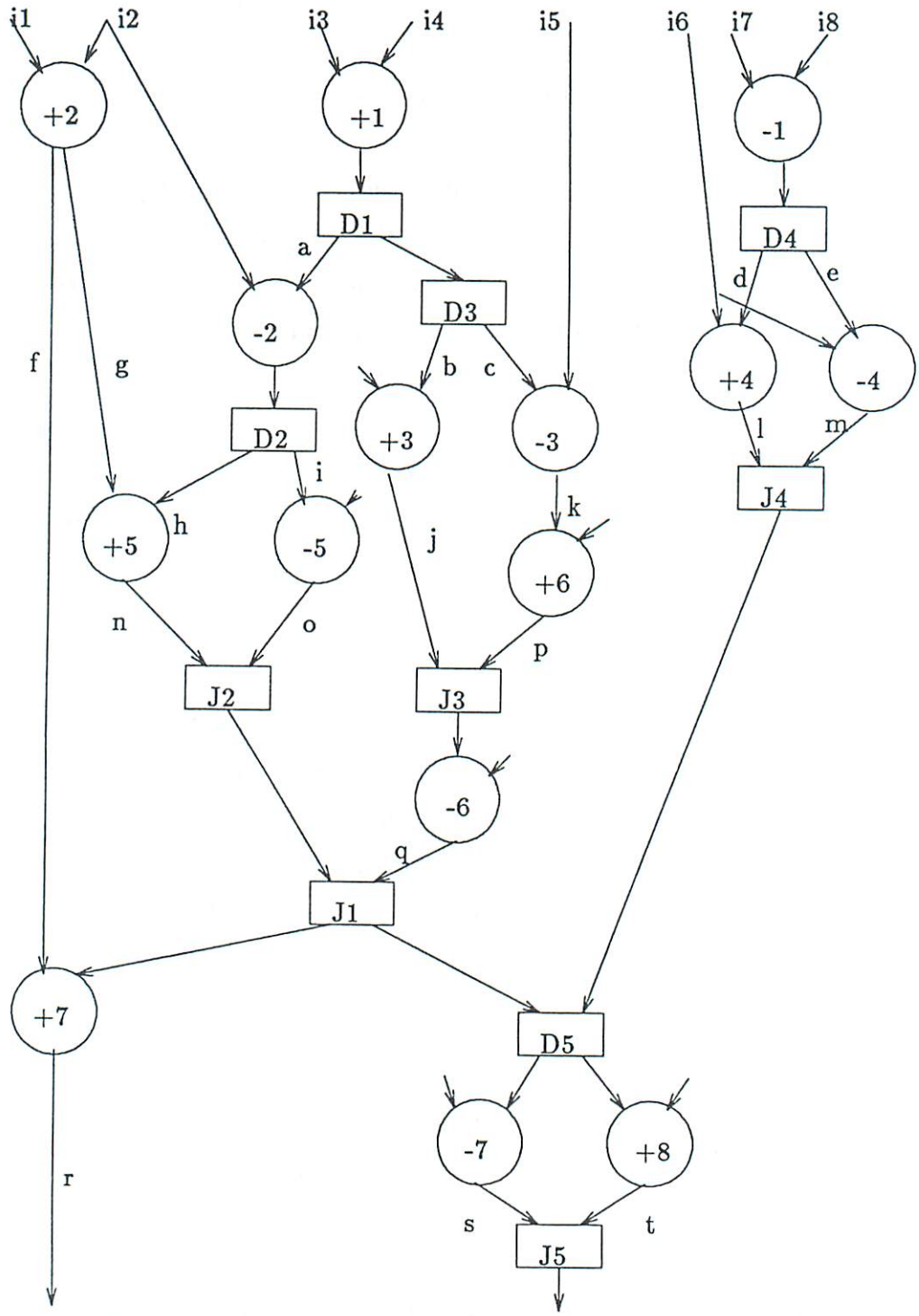


Figure 6.9: A data flow graph with conditional branches

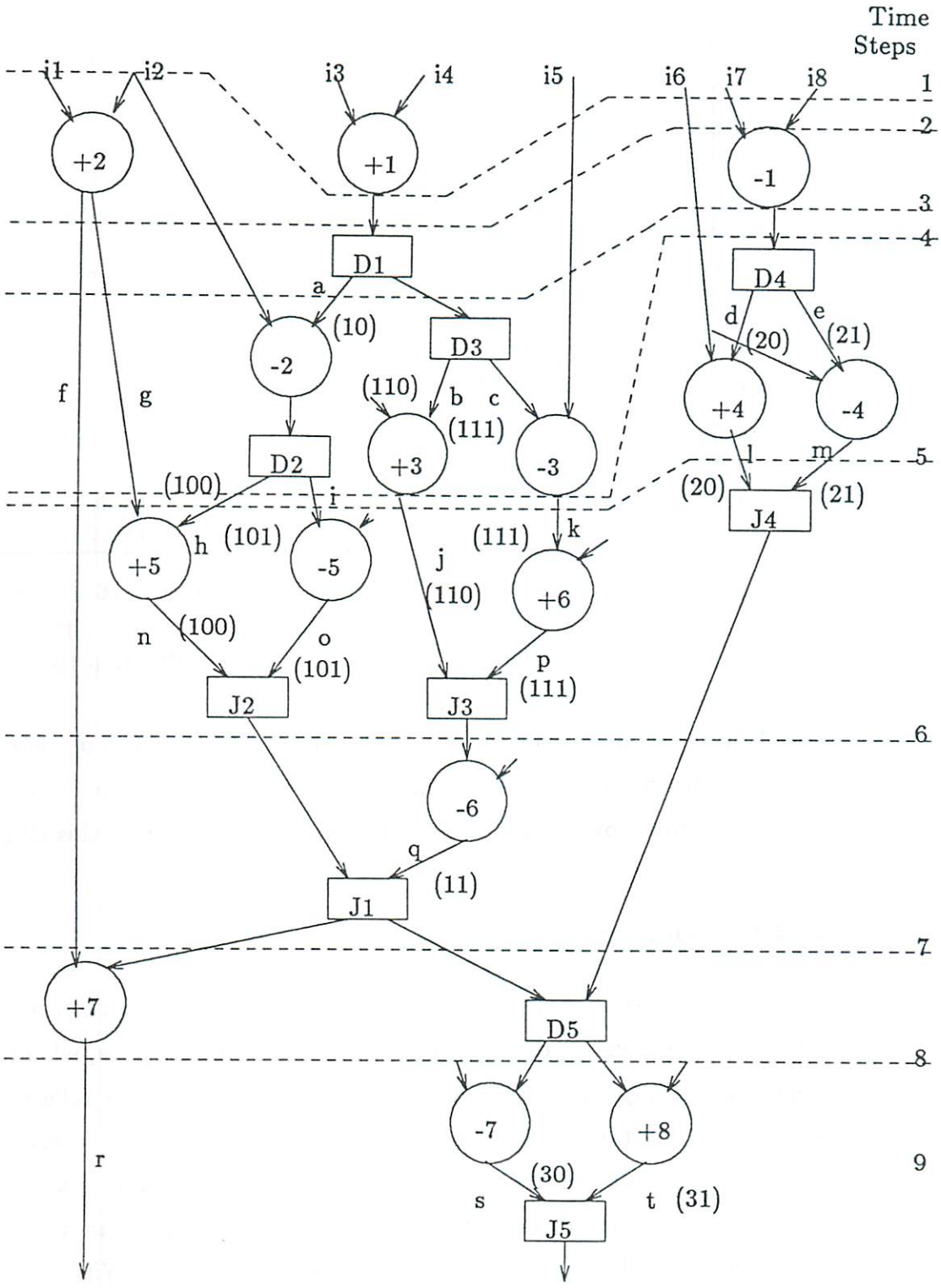


Figure 6.10: A pipelined scheduling of the dataflow graph with latency of 3

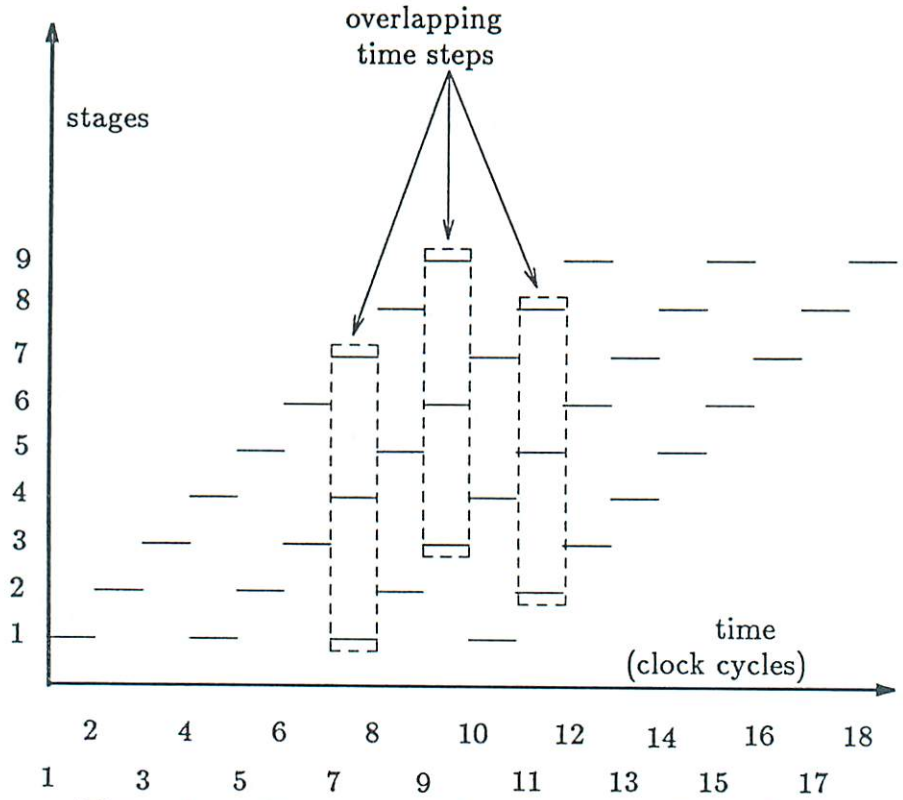


Figure 6.11: Time overlap of operations in Figure 6.10

steps 1, 4 and 7 are executing simultaneously on different sets of data. At any given time, the hardware is processing not one, but several sets of data. The next section describes how to modify the lifetime table to deal with this situation.

6.4.2.1 Modified lifetime table

If we take a 'snapshot' of the graph at any given time, there are *multiple instances* of each value, each separated by a period of L cycles, where L is the latency of the pipeline. This multiple instances of values is reflected in the *modified* lifetime table shown in Figure 6.12. A new instance of each value is generated every L cycles.It is interesting to note that if a value is alive for more than L timesteps, then more than one instance of that value may exist at the same time and will have to be stored in different registers. With the modified Lifetime table, one can obtain a register allocation using the Modified Left Edge

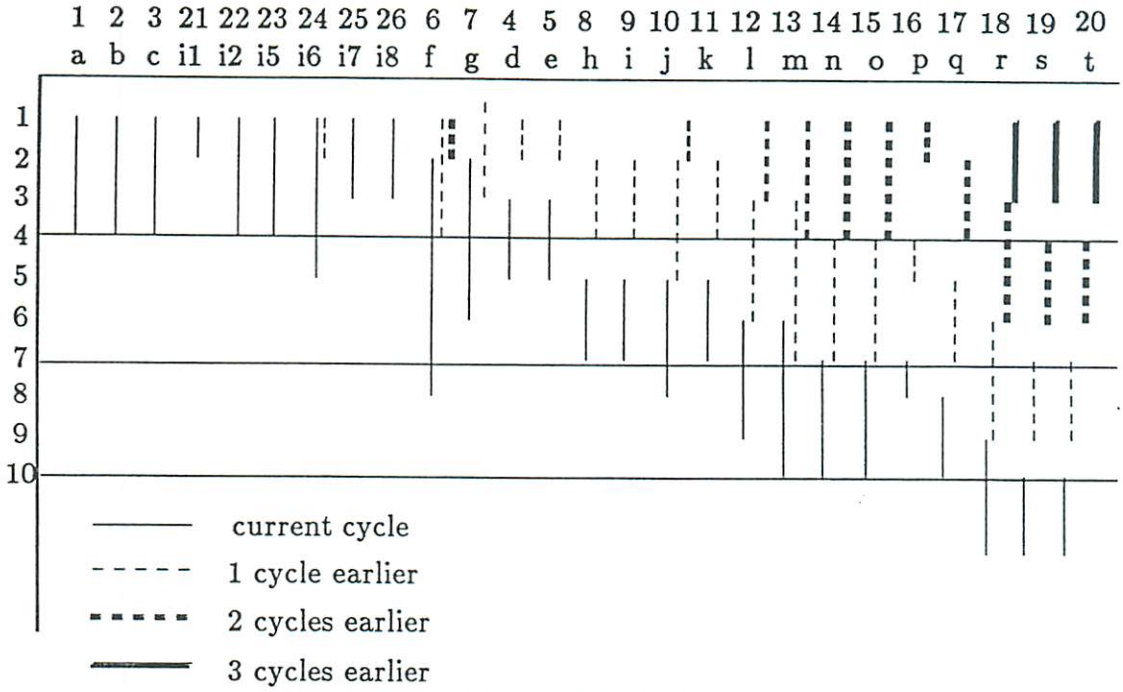


Figure 6.12: The modified lifetime table

algorithm. Note that the register allocation, when performed on the modified table, includes allocating the *stage latches* since it takes into consideration the multiple occurrence of values. We present a theorem which has the effect of reducing the size of the register allocation problem by reducing the problem to looking only at a part of the lifetime table.

Theorem 6.1 *In order to perform register allocation for a pipelined design with latency L, it is only necessary to look at any 'window' of L consecutive cycles of the modified lifetime table.*

Outline of Proof: Follows from the fact that the whole pipeline is periodic with period L. This means that values themselves are periodic with the same period. This can be better visualized by looking at Figure 6.12. Indeed the pattern of values is repeated every L cycles. Without any loss of generality, we choose the reduced lifetime table to comprise the first L cycles of the original table. □.

This is shown in Figure 6.13. If the input dataflow graph has N cycles per data set (i.e. the total time one data set spends in the system), then by using

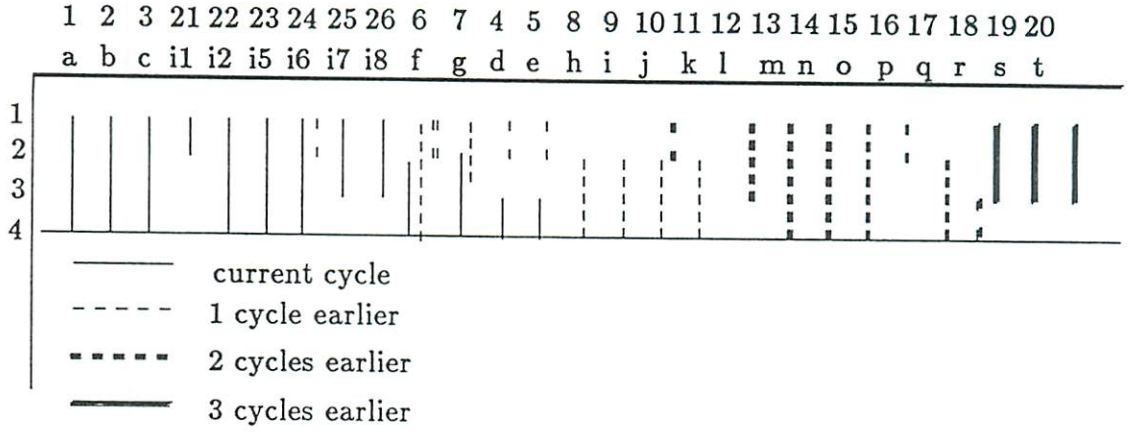


Figure 6.13: The reduced lifetime table

the reduced table instead of the original one, the problem size would be reduced by a factor of N/L.

6.4.2.2 Conditional branches in pipelines

Conditional branches in pipelined schedules require more attention. Two overlapping and mutually exclusive values can share a register *iff* they belong to the *same* data set, i.e. they must be in the *same* time step, not in overlapping time steps. This is due to the fact that the branching conditions may vary from one data set to another. Figure 6.14 depicts such a situation, Values x and y are mutually exclusive, since they belong to different branches. However, if the latency L is 2, then timesteps 2 and 4 will be overlapping, and x and y must be stored at the same time, but these two values cannot share a register because they belong to different data sets. This is so because the branching condition during the cycle when y was generated may be different than the one during the next cycle, when x was generated. This means that x and y can indeed exist at the same time and must be stored in separate registers.

To account for this peculiarity, each value in the lifetime table is assigned a time index indicating which cycle it belongs to. Overlapping values with different time indexes cannot share registers even if their colors indicate that they are mutually exclusive.

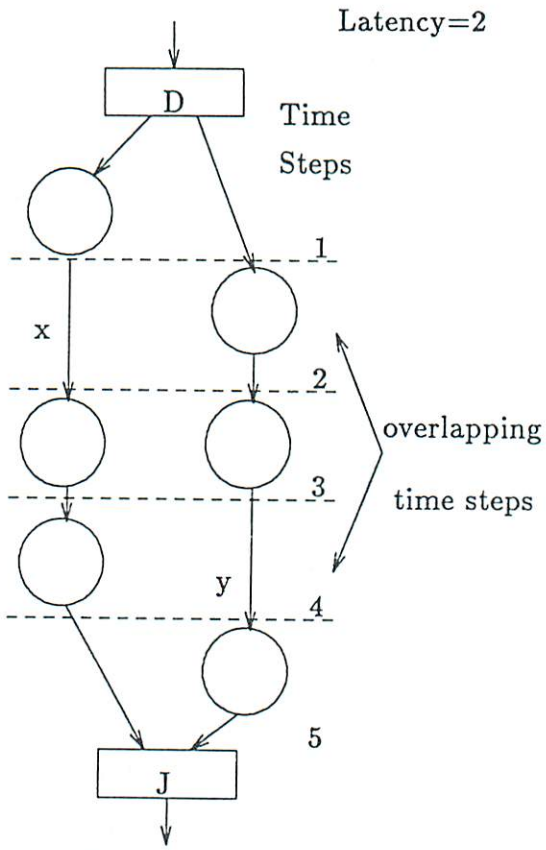


Figure 6.14: Conditional branches in pipelined dfgs

```
Number of values = 26
Number of time steps = 10
Number of registers needed = 8
Allocation of values to registers
Register #: value1, value2,...

Register 1 : 1 ,2 ,3 ,8 ,9 ,10 ,11 ,14 ,15 ,16 ,17 ,19 ,20
Register 2 : 21 ,6 ,18
Register 3 : 22 ,12 ,13
Register 4 : 23
Register 5 : 24
Register 6 : 25 ,4 ,5
Register 7 : 26
Register 8 : 7
```

Figure 6.15: REAL's allocation for the non-overlapped example in Fig. 6.9

6.5 Implementation and results

The Modified Left Edge algorithm was implemented in a program called REAL (or a REgister ALlocator). REAL is written in C and runs under UNIX 4.2 bsd on a SUN 3/160 workstation. Figure 6.15 shows the final allocation of values to registers for the example in Section 6.4.2, this time assuming that the design in non-overlapped. This allocation took less than a second of CPU time. While this allocation is not guaranteed to be optimal, all the examples we have run so far have produced optimal allocations. This is an indication that the algorithm may still be optimal in the existence of conditional branches, or at worst, would produce near optimal solutions.

Figure 6.16 shows the register allocation generated by REAL for the example in Figures 6.10 and 6.13. The resulting allocation took less than a second of cpu time for 36 values.

6.6 Summary

In this chapter, the task of estimating the register cost in a design implementation is examined. Once the dataflow graph design representation is processed by synthesis programs, such as MAHA or Sehwa, a partial design results, in which the lifetimes of values are known. The problem of allocating values to registers was identified and shown to be identical to the track assignment task in channel routing. The problem can be solved optimally under certain conditions. Estimating the register count is done by actually allocating the values to registers, since the allocation takes very little time to be produced. The solution scheme is extended for designs with conditional branches and to handle pipelines.

```
Number of values = 36
Number of time steps = 4
Number of registers needed = 17
Allocation of values to registers
Register #: value1:index1, value2:index2,...

Register 1 : 1:0, 2:0, 3:0,
Register 2 : 4:1, 5:1, 6:0,
Register 3 : 6:2, 7:0,
Register 4 : 6:1,
Register 5 : 7:1, 4:0, 5:0,
Register 6 : 10:2, 14:2, 15:2, 17:2,
Register 7 : 12:2, 13:2,
Register 8 : 16:3, 8:1, 9:1, 10:1, 11:1,
Register 9 : 18:3, 12:1, 13:1,
Register 10 : 19:3, 20:3, 18:2,
Register 11 : 21:0,
Register 12 : 22:0,
Register 13 : 23:0,
Register 14 : 24:1,
Register 15 : 24:0,
Register 16 : 25:0,
Register 17 : 26:0,
```

Figure 6.16: REAL's allocation for the pipelined lifetime table

Chapter 7

Conclusions and future research

If it [automation] keeps up, man will
atrophy all his limbs but the push button finger
–FRANK LLOYD WRIGHT, *The New York Times*, Nov. 27, 1955

7.1 Main contributions

The main goal of this research was to provide the designer (be it human or automated) with the capabilities of evaluating the implementation of the proposed design at various stages of the design process. Such evaluations are instrumental in shortening the design turnaround time because the outcomes of the actual design steps can be predicted and therefore bypassed. The primary result of this thesis is that predicting the area of a chip is indeed a feasible task. The models and data presented throughout the thesis provide means—direct in some cases, and guidelines in others—for predicting the size of the target design prior to the various design tasks.

The model presented in Chapter 2, and its implementation and validation in Chapter 3 provide the designer with accurate means for estimating the wiring area in a design, and therefore the total layout area once the configuration of the design logic has been determined. There are three features of this estimation scheme that make it valuable:

- The area estimation seems to be accurate to within 10%, which is a significant result considering that the underlying theory is independent of the particular layout system being used, which makes the model general.

- The model implementation, PLEST, can estimate the chip area in about a second of CPU time. This is at least three to four orders of magnitude faster than the layout task. Traditionally, the chip layout can be re-done several times until the target circuit can be fit on the chip. Introducing the area estimation in the design loop will reduce the number of layouts to one or two which means an appreciable decrease in design time and cost.

- PLEST provides the designer with not only one, but a whole range of estimates, one for each possible layout configuration. These layout estimates have different sizes and shapes. This is a useful feature when the circuit under consideration is a part of a *heterogeneous* chip containing other blocks such as memory or PLAs. By providing alternate layout possibilities to the designer, a more compact chip can be laid out.

In Chapter 4, the wire length distributions in real chips were studied and it was found that stochastic models of wires are appropriate and usually accurate. The data and results obtained from the partitioning experiments on chip layouts come to prove the validity of Rent's rule in Standard Cell layouts. There are two interesting implications of this result: first, that Standard Cell partitioning and placement schemes can be characterized by Rent's rule, which can be used to rank and compare various placement and partitioning algorithms for Standard Cells over various examples. Second, it is possible to use the work in gate array wiring space estimation—where the rule was first investigated—to estimate the average wire length using Rent's rule parameters.

The area versus delay data for adders and multipliers presented in Chapter 5 indicate that it is possible to quantize the tradeoffs of RT-level modules and therefore assist the designer in selecting the appropriate module for the specific design at hand. The bounding techniques presented in Sections 5.5 and 5.6

provide the user with some insight on the size of the logic in the circuit under consideration. These bounds seem to be appropriate at the dataflow graph representation level. More examples, however, are necessary in order to prove their validity at the gate level.

The approach taken in Chapter 6 to estimating the number of registers in a design is to actually allocate the values to be stored to registers. This seems to be an efficient approach in this case, since the problem was found to be tractable and the actual allocation taking seconds of CPU time for moderate size examples. The other novelty in this chapter is in exploring the analogy of a problem in a somewhat distant field, the track assignment in physical design, to the allocation problem. This is an indication that there are probably a multitude of problems at the high level design that may be analogous to some well known problems in other fields for which efficient solutions have been devised.

7.2 Future research

Clearly, no research can claim to be perfect and to completely cover and solve all aspects of the problems related to its topic. During the course of this research, numerous research problems were encountered. Some of these problems were addressed in this thesis while others were left as subjects of future research.

The estimates for track requirements in Chapter 2 were obtained by computing the expected density at every grid point and finding its maximum over the whole row width. As mentioned in the model analysis, the maximum of the expectation is a lower bound on the expectation of the maximum, which is the quantity that should be estimated. A possible extension to the model would be to estimate this quantity, either by analysis of by computational means. Most probably, this would require finding the *distribution* of the density at every point x. Finding the distribution of the density is also useful to establish some confidence measures in the estimates of the model.

Most of the examples in Chapter 3 were generated using the MP2D layout system. Although this is an excellent system in the quality of its layouts

and it has been widely used on an industrial basis, it is important to apply the model of Chapter 2 to other layout systems. This will present the model as a more general area estimation tool.

An essential parameter in our model is the average wire length. In estimating this quantity, we relied on results obtained by other researchers. These works were based on the assumption that Rent's rule holds for the layouts under consideration. The experimental results in Chapter 3 proved the adherence of the standard cell layouts to the rule. The models presented were however, based on gate array design style and, as seen, the average wire length estimates were not as accurate as one would like them to be. A model based on the standard cell design style may establish a more accurate relationship between Rent's rule and the average wire length of a layout. One possible direction for such a model is to use the fractal geometry interpretation of Rent's rule discussed in Chapter 3.

So far, we have been concerned with estimating the area of random logic (in the standard cell design style). In practice, a chip may also contain blocks of logic implemented in different design styles. This is because some parts of a digital design cannot be implemented efficiently in standard cells. Random Access Memories (RAMs), for example, would be wasteful in area and design effort if implemented with standard cells. Also, the designer may choose to implement some parts of a design in a different design style for simplicity and modularity of the design. Such may be the case, for example, in the design of the control part of a circuit using Programmable Logic Arrays (PLAs).

A global area estimator must be able to handle blocks of different design styles. Such an estimator can provide the designer with a set of possible, 'reasonable' configurations for each of the blocks that compose a chip, along with estimates of their physical dimensions. The designer can then select an appropriate configuration for each of the blocks so as to satisfy some preset constraints on the individual blocks or on the whole chip. Estimating the area taken by the global wiring is another important problem which must also be researched. One possible direction here is to investigate the model proposed by Syed [Sye81].

In Chapter 4 we derived upper bounds on register and routers costs for the cheapest(most serial) RT-level implementation of a design. Although one would expect the number of registers and multiplexors to increase as the design is serialized, in order to establish an absolute lower bound on design cost, one must find lower bounds on register and multiplexor costs for the cheapest design. Finding such bounds will most probably turn out to be *NP-complete* problems in which case approximations must be found.

Different RT level implementations of a dataflow graph can be obtained by varying the area (cost) or speed constraints. The bounds in Chapter 4 only apply to two possible designs, the cheapest and the fastest. One important extension would be to *characterize* the design space itself. Put in different words, given a constraint on speed, predict the cost of the cheapest (or smallest area) implementation which satisfies this constraint. The example implementations for adders and multipliers presented in Chapter 4 seem to follow relations of the type $AT^{\alpha} = k$ where A and T are the area and speed of an implementation, respectively, and α and k are constants which depend on the type of dataflow graph and the bit width of the adder or multiplier. In [JPP87] experiments on several dataflow graphs were run in which several implementations were synthesized for each graph and were found to fit the same relation as for the adder and multipliers with $\alpha = 1$. If such relationships can be proven and obtained for the general case, they may prove to be invaluable in exploring the design space and cutting down on the design turnaround time.

The Left Edge algorithm in Chapter 5 was proven to be optimal for register allocation in dataflow graphs with no conditional branches. One interesting extension would be to prove whether or not it remains optimal in the presence of conditional branches. Also, the algorithm as presented does not take into consideration the multiplexing cost associated with sharing registers. In many cases, this assumption is justified if the cost of multiplexors is small compared to that of the registers. A possible extension would be to do the register allocation with the dual goal of minimizing both the register cost and the multiplexing cost, or a weighted function of both.

One last note to be made at this point concerns integrating the high level prediction concepts and the gate level area estimation model. Clearly, there is a missing link [1] between the RT level description of a design and its gate level description. Such a link performs the module binding task which is synthesizing a gate level implementation of an RT level module. This is a difficult task as many possible gate level implementations may exist for a particular module, different gate sizes may be required depending on the fanout requirements, and some modules (such as multiplexors) are generated in a non trivial way. Initial work is being carried out at USC in building such a program which, when completed, would make possible the integration of the prediction system whereby the high level bounds of Chapters 4 and 5 can be translated to a gate level description which can be fed into the model of Chapter 2 to obtain accurate area estimates.

[1] with apologies to [BP81,Dar59].

Bibliography

[Aba85] M. Abadir. *A Knowledge Based System for Designing Testable VLSI Circuits.* PhD thesis, Department of Electrical Engineering-Systems, University of Southern California, December 1985.

[Afs85] H. Afsarmanesh. *The 3 Dimensional Information Space (3DIS), an Extensible Browsing-Oriented Framework for Database Systems.* PhD thesis, Department of Computer Science, University of Southern California, 1985.

[BS77] M. Barbacci and D. Siewiorek. An architectural research facility - ISP descriptions, simulation, data collection. *AFIPS Proceedings*, 46:161–173, 1977.

[BP81] M. A. Breuer and A. C. Parker. Digital system simulation: current status and future trends. In *Proceedings of the 18th Design Automation Conference*, pages 269–275, IEEE/ACM, June 1981.

[Bre72] M. A. Breuer. *Digital Systems Design Automation, vol. I: Theory and Techniques*, (M. A. Breuer, Ed.), chapter 2, pages 21–100. Prentice-Hall, Englewood Cliffs, N.J., 1972.

[Dar59] C. Darwin. *On the Origin of Species.* 1859.

[Deu76] D. N. Deutsch. A 'dogleg' channel router. In *Proceedings of the 13th Design Automation Workshop*, pages 425–433, IEEE, 1976.

[Din70] E. A. Dinic. Algorithm for solution of a problem of maximum flow in a network with power estimation. *Soviet Math. Dokl.*, 11:1277–1280, 1970.

[Don69] W. E. Donath. *Hierarchical Structure of Computers.* Technical Report RC 2392, IBM T. J. Watson Research Center, Yorktown Heights, N.Y., March 1969.

[Don79] W. E. Donath. Placement and average interconnection lengths of computer logic. *IEEE Transactions on Circuits and Systems*, CAS-26(4):272–277, April 1979.

[ElG81] A. A. El Gamal. Two-dimensional stochastic model for interconnections in master slice integrated circuits. *IEEE Transactions on Circuits and Systems*, CAS-28(2):127–138, February 1981.

[Eve79] S. Even. *Graph Algorithms. Computer software engineering series*, Computer Science Press, Rockville, MD, 1979.

[Feu82] M. Feuer. Connectivity of random logic. *IEEE Transactions on Computers*, C-31(1):29–33, January 1982.

[FN78] A. Feller and R. Noto. A speed oriented, fully-automatic layout program for random logic VLSI devices. In *AFIPS Conference Proceedings Vol.47, 1978 National Computer Conference*, pages 303–311, June 1978.

[GJ79] M. Gary and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, San Francisco, 1979.

[GK84] E. Girczyc and J. Knight. An ADA to standard cell hardware compiler based on graph grammars and scheduling. In *Proceedings, International Conference on Computer Design - ICCD*, pages 726–729, October 1984.

[GP82] J. J. Granacki and A. C. Parker. The effect of register-transfer design tradeoffs on chip area and performance. In *Proceedings of the 20th Design Automation Conference*, pages 419-424, IEEE/ACM, June 1982.

[GKP84] J. Granacki, D. Knapp and A. C. Parker. *The ADAM advanced design automation system: overview, planner, and natural language interface.* In *Proceedings of the 22nd Design Automation Conference*, pages 727–730, IEEE/ACM, June 1985.

[Gra86] J. J. Granacki. *Understanding Digital Systems Specifications Written in Natural Language.* PhD thesis, Department of Electrical Engineering-Systems, University of Southern California, November 1986.

[HP78] L. Hafer and A. C. Parker. Register-transfer level digital design automation: the allocation process. In *Proceedings of the 15th Design Automation Conference*, pages 213–219, IEEE/ACM, June 1978.

[HP83] L. Hafer and A. C. Parker. A formal method for the specification analysis, and design of register-transfer level digital logic. *IEEE Transactions on Computer-Aided Design*, CAD-2(1), January 1983.

[HS71] A. Hashimoto and J. Stevens. Wire routing by optimizing channel assignment within large apertures. In *Proceedings 8th Design Automation Workshop*, pages 155–169, IEEE, June 1971.

[HMD78] W. R. Heller, W. F. Mikhail and W. E. Donath. Prediction of wiring space requirements for LSI. *Journal of Design Automation and Fault Tolerant Computing*, Vol. 2, pages 117–144, May 1978.

[Hel77] W. R. Heller et al. Prediction of wiring space requirements for LSI. In *Proceedings of the 14th Design Automation Conference*, pages 20–22, IEEE/ACM, June 1977.

[Hic83] P. J. Hick, Ed. *Semi-custom IC design and VLSI.* IEE, 1983.

[Hig69] D. W. Hightower. A solution to line-routing problems on the continuous plane. In *Proceedings of the 6th Design Automation Workshop*, pages 1–24, IEEE, 1969.

[Hit83] C. Y. Hitchcock. *Automated Synthesis of Data Paths.* Master's thesis, Department of Electrical Engineering, Carnegie-Mellon University, 1983.

[JPP87] R. Jain, A. C. Parker and N. Park. Predicting the area-time tradeoffs in pipelined designs. In *Proceedings of the 24th Design Automation Conference*, pages 35–41, IEEE/ACM, July 1987. (to appear).

[Kan83] S. Kang. Linear ordering and application to placement. In *proceedings of the 20th Design Automation Conference*, pages 457–464, IEEE/ACM, 1983.

[Key81] R. W. Keyes. Fundemental limits in digital information processing. *Proceedings of the IEEE*, 69:267–278, February 1981.

[KGP83] D. Knapp, J. Granacki and A. C. Parker. An expert synthesis system. In *Proceedings of the International Conference on Computer-Aided Design(ICCAD)*, pages 419–424, ACM/IEEE, September 1983.

[Kna84] D. Knapp. The v collision detector. USC DA Group, available online, 1984.

[KP83] D. Knapp and A. Parker. *A Data Structure for VLSI Synthesis and Verification.* Technical Report, Digital Integrated Systems Center, Department of Electrical Engineering-Systems, University of Southern California, October 1983.

[LR71] B. Landman and R. Russo. On a pin versus block relationship for partition of logic graphs. *IEEE Transactions on Computers*, C-20:1469, 1971.

[Lee61] C. Y. Lee. An algorithm for path connections and its applications. *IRE Transactions on Electronic Computers*, EC-10:346–365, September 1961.

[Lei81] G. W. Leive. *The Design, Implementation, and Analysis of an Automated Logic Synthesis and Module Selection System.* PhD thesis, Department of Electrical Engineering, Carnegie-Mellon University, January 1981.

[Liu83] W.-T. Liu. *Techniques for Estimation of the Area of Integrated Digital Circuits.* PhD thesis, Department of Electrical Engineering, University of Michigan, Ann Arbor, 1983.

[Man81] B. B. Mandelbrot. *Fractals: Form, Chance, and Dimension.* W. H. Freeman, San Francisco, 1981.

[Mat72] R. L. Mattison. A high quality, low cost router for MOS/LSI. In *Proceedings of the 9th Design Automation Workshop*, pages 94–103, IEEE, 1972.

[McF78] M. McFarland. *The Value Trace: A Data Base for Automated Digital Design.* Master's thesis, Department of Electrical Engineering, Carnegie-Mellon University, December 1978.

[McF81] M. McFarland. *Mathematical Models for Verification in a Design Automation System.* PhD thesis, Department of Electrical Engineering, Carnegie-Mellon University, July 1981.

[Mur82] S. Muroga. *VLSI systems design.* Wiley-Interscience, 1982.

[Nga83] J. Y. Ngai. *Computational Stochastic Models for Channel Routing Track Demand.* Technical memo: 5094, Department of Computer Science, California Institute of Technology, 1983.

[Par85] N. Park. *Synthesis of High Speed Digital Systems.* PhD thesis, Department of Electrical Engineering-Systems, University of Southern California, August 1985.

[PP86] N. Park and A. C. Parker. Sehwa:A program for synthesis of pipelines. In *Proceedings of the 23rd Design Automation Conference*, pages 454–460, IEEE/ACM, July 1986.

[PP85] N. Park and A. C. Parker. Synthesis of optimal clocking schemes. In *Proceedings of the 22nd Design Automation Conference*, pages 489–495, IEEE/ACM, July 1985.

[PW81] A. Parker and J. Wallace. SLIDE: A hardware description language. *IEEE Transactions On Computers*, C-30(6):423–438, June 1981.

[Ric84] B. D. Richard. A standard cell initial placement strategy. In *Proceedings of the 21st Design Automation Conference*, pages 392–398, IEEE, ACM, June 1984.

[Riv81] R. L. Rivest et al. Provably good channel routing algorithms. In *Proceedings of the CMU Conference on Systems and Computations*, pages 153–159, October 1981.

[Sas85] S. Sastry. *Wireability Analysis of Integrated Circuits.* PhD thesis, Department of Electrical Engineering-Systems, University of Southern California, January 1985.

[SP84] S. Sastry and A. C. Parker. On the relation between wire length distributions and placement of logic on master slice ICs. In *Proceedings of the 21st Design Automation Conference*, pages 710–711, IEEE/ACM, June 1984.

[SU72] D. M. Schuler and E. G. Ulrich. Clustering and linear placement. In *Proceedings of the 9th Design Automation Workshop*, IEEE pages 50–56, 1972.

[SO73] I. E. Sutherland and D. Oestreicher. How big should a printed circuit board be? *IEEE Transactions on Computers*, C-22(5):537–542, May 1973.

[Sup83] K. J. Supowit. Reducing channel density in standard cell layout. In *Proceedings of the 20th Design Automation Conference*, pages 263–269, IEEE/ACM 1983.

[Sye81] Z. A. Syed. *On Routing for Custom Integrated Circuits.* PhD thesis, Department of Electrical Engineering, University of Southern California, July 1981.

[Tho80] C. D. Thompson. *A Complexity Theory for VLSI.* PhD thesis, Carnegie-Mellon University, 1980.

[TS86] C.-J. Tseng and D. P. Siewiorek. Automated synthesis of data paths in digital systems. *IEEE Transactions on Computer-Aided Design*, CAD-5(3):379–395, July 1986.

[UKH85] K. Ueda, H. Kitazawa and I. Harada. CHAMP: chip floor plan for hierarchical VLSI layout design. *IEEE Transactions on Computer-Aided Design*, CAD-4(1):12–22, January 1985.

[VHDL86] R. Lipsett, E. Marschner and M. Shahdad. VHDL – The Language. *IEEE Design and Test*, April 1986.

[Yeh82] C. Yeh. *The Prediction of the Requirement of Wiring Space for VLSI.* PhD thesis, Electrical Engineering Deparment, The University of Pennsylvania, 1982.

[Zhu86] X. Zhu. *A Knowledge-Based System for Testable Design Methodology Selection.* PhD thesis, Department of Electrical Engineering-Systems, University of Southern California, August 1986.