Copyright © 1988, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

AUTOMATIC GENERATION OF CMOS DATAPATHS IN LAGER FRAMEWORK

by

Mani B. Srivastava

Memorandum No. UCB/ERL M88/40

3 June 1988

COPY

AUTOMATIC GENERATION OF CMOS DATAPATHS IN LAGER FRAMEWORK

by

Mani B. Srivastava

Memorandum No. UCB/ERL M88/40

3 June 1988

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

AUTOMATIC GENERATION OF CMOS DATAPATHS IN LAGER FRAMEWORK

by

Mani B. Srivastava

Memorandum No. UCB/ERL M88/40

3 June 1988

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Abstract

This report describes a system to generate CMOS data paths for a variety of applications demanding high performance, such as image and speech processing. The goal of the project was to arrive at a consistent and well defined methodology for creating data paths for many different applications. The major issue which was addressed was how to avoid the duplication of circuit design and layout effort for each new application while still meeting high performance requirements. To this end a large CMOS cell library was designed and tested, and a CAD tool, called the DPC (Data Path Compiler) was written to generate the layout from a structural description of the data path. This tool was integrated with the Design Manager, a silicon assembler. This report gives a detailed description of the motivation behind this system, description of the strategies used for placement and routing of bit slice data paths, the details of the software, the description of the various functional blocks currently provided in the system library and some examples where this system has been used. Also included are the user manuals for the tools described in this report.

This research was supported by Defense Advanced Research Projects Agency, Contract N00039-87-C-0182.

Contents

List of Figures

Chapter 1

Introduction

This report describes a system to design high performance data paths for custom VLSIs for a variety of applications like audio signal processing, image processing and robot control. High performance data path design is becoming an increasingly important problem in VLSI design particularly because of the increasing importance of Application Specific Integrated Circuits (ASICs) in the market place. ASICs aim at providing dedicated custom hardware based solutions for a variety of applications and high performance data paths play an important role in these applications. Further, the turn around time is of critical importance in the current industrial situation. Therefore, a system to systematically generate high performance dedicated data paths with a fast turn around time can play a very important role in decreasing the overall design time. This is true partly because unlike other circuit blocks in a typical VLSI, the high performance data paths are not implemented suitably by structured layout synthesis techniques like PLAs for which a variety of tools are available.

The system described in this report is basically a data path assembly system which generates the layout of a data path starting from a description of the data path as an interconnection of high level functional blocks. This tool is embedded in the Design Manager environment which is a system for hierarchical assembly of integrated circuits (see [3]). Design Manager provides extensive functional simulation support for the data paths designed using the system.

1.1 Drawbacks of Existing Approaches for Data Path Design

A major problem in the design of high performance data paths is the circuit design and layout effort which is required every time a new data path is designed. The reason for this is that conventional automated layout approaches, such as standard cells and conventional gate array layout often do not meet the performance and area requirements. This is because these techniques are general purpose techniques and do not try to exploit the special nature of signal flow in the data paths. As a result of the failure of these techniques in meeting the performance requirements, most designers of high performance data paths still use the approach of completely designing each new data path.

Obvious disadvantages of this approach are the inflexibility of the design to even minor alterations and the long turn around time. The second factor is particularly important in the current and future commercial scenarios with the increasing dominance of the ASICs. Further, the inflexibility of this approach forces the designer to decide on the functional and logic design before the layout can begin. Trying out various competing designs at the transistor layout level is not feasible. Another drawback is the duplication of design effort. Most of the data paths usually consist of the same basic blocks connected in different configurations. The conventional approach of designing from scratch does not take advantage of work already done by somebody else.

In view of these drawbacks, an approach which allows flexibility of design, fast turn around time and reusability of existing designs, but still meets the high performance requirements is desired. The design flexibility and fast turn around time requirement suggest a CAD tool based system. There exist layout synthesis tools which use a structured layout style such as PLA and Gate Matrix. The PLA style is good for combinational logic but quite unsuitable for data paths where we have a mix of latches and combinational logic. Further, despite techniques like PLA folding, these structured designs do not meet the high performance requirements. The Gate Matrix style allows mixing combinatorial logic and latches but still falls short on performance because of long polysilicon interconnects which result in high node capacitances. In short, structured layout strategies just do not seem to meet the high performance requirements. This points to a paradigm based on a well designed cell library where the individual cells are well designed and optimized. Such a cell library based approach naturally satisfies the requirement of reusability of existing designs.

Further, non-critical cells can be designed using a structured approach like Gate Matrix while the critical cells, like adders, shifters, can be designed and optimized manually.

Standard Cell based designs seem to meet the requirements. However, it is a general purpose technique and is unsuitable in the data path applications because it does not take advantage of the way the signals flow in most data paths. This makes this design style unsuitable in terms of area and to some extent performance. For example, in most data paths the directions of flow of the data signals and the control signals are mutually perpendicular. This orthogonal flow of signals, however, does not fit well with the Standard Cell style where all the terminals come out either at the top edge or the bottom edge of the cell.

1.2 Desired Features in the Data Path Design System

As discussed in the previous section, none of the current macrocell design techniques meet the special requirements for dedicated data paths. However, before designing a system to generate data paths, it is important to figure out the features desired in a good system. As already mentioned, this system was meant to be integrated with the silicon assembly environment provided by the Design Manager [3]. This placed an important compatibility requirement. However, more important was the fact that this system was meant to be used extensively within our research group for a variety of applications. This meant that the system had to use a strategy which satisfied most of the designers. In order to do this, manual design of a data path, specifically the Lager AUIOINC data path, was undertaken in order to study the various design issues. During the course of this manual design effort and by extensive consultation with other data path designers, a design strategy was arrived at which was reasonably close to the design practices prevalent among the designers in our research group and was automatable. Some important issues tackled at this stage were routing of power and clock signals and orientation of wells in the CMOS technology. Also, a suitable set of primitive functional blocks which met most of our requirements was determined.

One important consideration while deciding the overall strategy was that the system should be able to use the already existing leafcells with minimal changes and the cell design style used should be flexible enough so that the cell designers do not feel unduly constrained.

Another desired feature was parameterizability. This was considered important so as to avoid having to redesign similar data paths. One prime example of this is the width of the data path. To facilitate this paramterization all the data paths were designed in a bit sliced fashion. This also had the side effect of making the routing and placement problem easy.

A powerful user interface which provided features to support parameterizability and support flexible extension of the system library was also essential. Considerable time was devoted for this and the resulting syntax went on to become the core for the Structural Description Language used by the Design Manager.

1.3 Outline of the Report

This report is divided into 7 chapters. Chapter 2 gives a description of the system in terms of the overall strategy used. Chapter 3 deals with the algorithms used and the software details. Chapter 4 basically documents the current status of the system libraries. Chapter 5 describes the problem associated with generating the interface circuitry between the data path and the rest of the world (controller, i/o) and discusses a tool called eqn2sdl created for this purpose. Chapter 6 gives some example data paths with the input decks and the resulting layout along with the execution time. Finally, chapter 7 concludes the report by discussing the advantages and disadvantages of this system and suggestions for improvement. The software listing and the library documentation are included in the appendices.

Chapter 2

Description of the Data Path Generation System

This chapter describes the data path design system and the methodology used. The topics discussed include the floorplanning and routing strategy, the user input and the overall library organization. Also discussed are the various constraints placed on the leafcell designers.

As already mentioned in the previous chapter, the main goal behind the design of this system was to arrive at a systematic and automated technique for designing a variety of data paths in a flexible manner with maximum utilization of the design effort already put into previous designs. Further, parametrization was also a much desired goal so as to have a mechanism for customizing a data path for different situations. All this was to be done under the constraint that the leafcell designers continue to have as much design freedom as possible. These facts guided the various design choices made at the various stages of the system.

2.1 Minimization of Layout Effort

The first problem which was tackled was the minimization of layout effort. There are two possible ways for doing this, namely layout synthesis and using a cell library. The layout synthesis techniques work very well for structured layouts like PLAs. However, PLAs are meant for combinational logic only. Data paths have a variety of latches and flipflops which are not implementable in a PLA design style. However, in recent years general

purpose structured layout techniques, like Gate Matrix, have been proposed. These layout styles give fairly dense layouts and are not restricted to combinational logic alone. The second way to minimize the layout effort is by using a library of cells. In this case the layout effort has to be done once at the beginning and then the same cells can be used again later on.

The first approach (layout synthesis) would be the ideal one since there would be no layout effort required at all. Also, since the layout is being synthesized, transitions from one set of design rules to another would only require rewriting a technology file. Therefore, the first approach tried was the Gate Matrix style [10] which seemed to be the most suitable for our application. Unfortunately, there was no tool available to us which could synthesize such layouts. However, since our main interest was in making an evaluation of this design style, it was decided to use a symbolic input style which retained the design rule independence of this approach. Also, the layout was very structured so that the cell design time was less than that for random layout style. The generic LAGER data path was designed using this style as a test case. It was observed that high node capacitances are a problem with this layout style. The reason for this was that the gate matrix style prohibits usage of metal2 layer. Therefore, the two routing layers available to the user were metal1 and polysilicon. Unfortunately, polysilicon is not a good routing layer and running long lines in that layer severely affects the performance. Furthermore, the cell designers in our research group felt that the restriction placed on using metal2 was unreasonable. A more detailed description and evaluation of the various structured layout techniques can be found in [11].

As a result of the problems described in the previous paragraph, it was decided to use the second approach, namely a cell library based approach. This approach requires some initial cell design effort. Also, it has the drawback that a change in design rules will require redesigning the whole cell library, unless the technology itself is scalable, like the SCMOS technology from MOSIS (see [12]). However, it has the advantage that the cells can be optimized since the leafcell designer has full control over the layout.

Actually, the final approach which was taken was to have a library of cells, with the provision that any restrictions placed by the router on the leafcell layout should be such that they can easily be satisfied by structured layout styles like Gate Matrix. This is a compromise between the two approaches listed above and allows the users of this system to have a design rule independent cell library using the symbolic Gate Matrix style (described in [11]), if they so wish and provided the performance is satisfactory.

2.2 Bit Slice Data Paths

The second major issue was parameterization of the data paths. The users wanted to have the capability of customizing the data paths depending upon certain parameters provided at the layout time. One can essentially have two types of parameters. The first type are those which affect the net list of the data path. For example, a certain net may or may not exist depending upon a certain parameter. Such parameters, however, do not affect the floorplanning or the routing schemes. The second parameter is the capability of customizing the number of bits in the data path. If one is to follow a generalized macrocell placement type technique, this customization of the number of bits will not be a problem. However, we decided to take advantage of the customization of number of bits and use a specialized placement and routing scheme which is closer to the style used by most data path designers. Specifically, we adopted the bit slice strategy for laying out the data paths.

The bit slice strategy involves visualizing the data path to be made up of a number of slices where each slice corresponds to the data flow for one particular bit position. This however assumes that the data flow has the same width (number of bits) at all points. The advantage is that the data path can now be visualized as a linear array of blocks. However, one should note that one cannot express data paths with variable bit widths with this scheme. For example, if certain registers are say 12 bits wide and some others are 16 bits wide, then it is not possible to put them in a bit slice fashion. From the experience of the data path designers in our group it seemed that this is not a drawback and by suitably partitioning a variable word width data path into several data paths of constant word width (bit slice) data paths, one can handle any data path. Further, most of the data paths designed by the designers in the group were already in a bit slice fashion.

The major effect of adopting the bit slice data path strategy is in the floorplanning. The individual slices can be placed and routed independently and then one can abut them by using a one-dimensional tiler to get the complete data path. This tiling requires that the terminals at the edge where two slices abut should pitch match. Therefore, the leafcells need be designed keeping this on mind and the individual slices need be routed accordingly. Figure 2.1 shows the floorplan of a bit slice data path. As shown in the figure, the bit slice layout approach allows one to naturally exploit the orthogonality in the flow of the data signal and the control signals. What is meant by this is that the flow of the data signals is within a particular bit slice and is more or less replicated in the other bit positions. On the

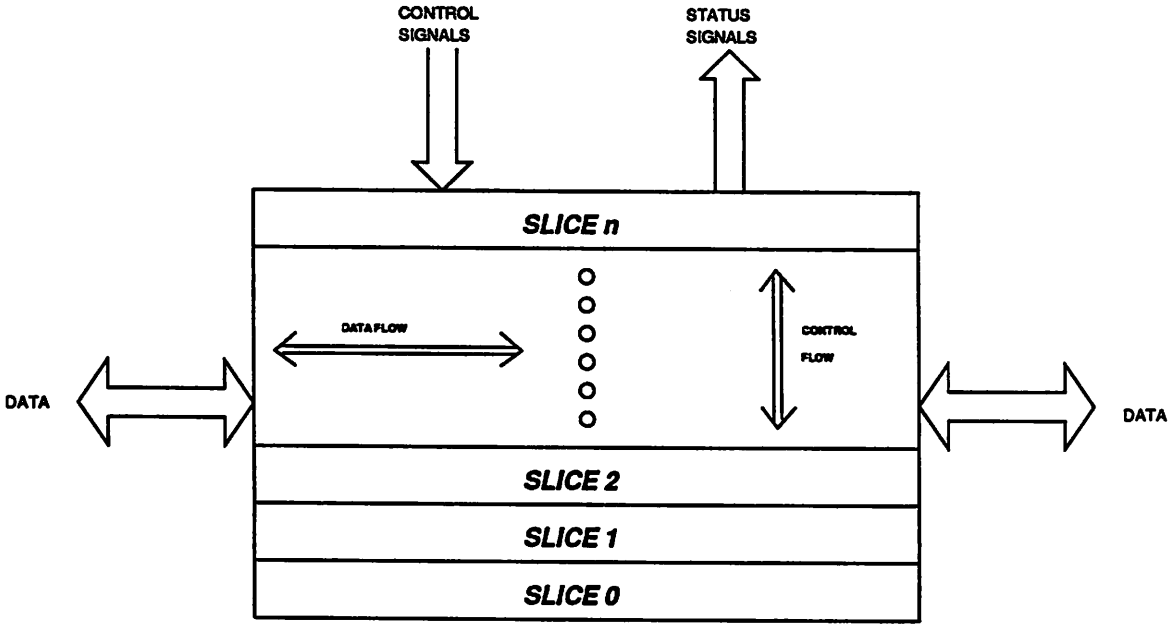


Figure 2.1: Floorplan of a Bit Slice Data Path

other hand, signals like the control lines, power and ground lines and clocks run across the bit slices and are common to all the cells at the same position. For example, the same control signal is used in a multiplexor at all bit positions. This orthogonality can be exploited at the layout level such that the control signals, power and ground lines and the clocks run orthogonal to the data flow direction. This style is the most common style used by the data path designers. When doing an automatic generation of the data paths, this style helps in simplifying the task. In particular, since the control signals, power and ground lines and clocks run across the slices, their routing is automatically taken care of by abutment. This requires that the leafcells be designed such that their control, power, ground and clock terminals abut with the corresponding terminals for the cell in the adjacent bit position. Throughout this report it is assumed that the data flow is in the horizontal direction and the control flow is in the vertical direction. Thus, the individual slices are formed by a linear placement and routing of leafcells in the horizontal direction. The slices are then abutted in the vertical direction.

2.3 Routing Strategy

As mentioned in the previous section, the bit slice approach reduces the problem to one of linear placement and routing. Linear placement is an easy task, although in the current system it has not been automated. The user has to specify the placement. The routing within the slice involves only the data signals. To ease the routing task, it was decided that the leafcells should be such that all the data signals come out on the left or the right side. The control, power, ground and clock terminals come out at the top and bottom and are extended to the top and the bottom edges of the slice so that they abut with the corresponding terminals in the adjacent slices.

The routing consists of two distinct type of nets. First are those which connect terminals of adjacent cells which are simple to handle. The second type are those nets which connect terminals on non-adjacent cells. This requires routing over the intermediate cells. To handle this routing, one approach is to make the cells transparent to a routing layer and then use that layer to route over a cell. The three routing layers available are polysilicon, metal1 and metal2. However, one cannot design cells without using polysilicon. Further, only metal1 can directly connect to polysilicon and diffusion. This precludes making cells transparent to metal1. Only metal2 is a possibility but this complicated the leafcell design.

The reason for this was that in the technology available to us (SCMOS from MOSIS [12]) the polysilicon resistance is a rather high 100 ohms per square. This renders polysilicon useless for routing except over very small distances. Therefore, metal2 was required in the leafcells because of the effect on performance.

In keeping with the philosophy of tailoring the system according to the design style used by the users, it was decided to use an alternative scheme for routing over the cells. Basically, it was decided that the leafcell designer should explicitly provide feedthroughs in the cells. A feedthrough is a pair of terminals, one on the left side and the other on the right side of the cell, which are electrically connected and are not used internally in the cells. They are provided for the express purpose of allowing the router to carry a net across the cell. A convention was adopted to have the names of these feedthrough terminals start with the *FEED* so that the router can recognize them. Further, the leafcell designers were encouraged to bring out the data terminals of the cells on both sides so that they too can be used as feedthroughs for nets connected to them. Again, a convention was adopted that terminals having the same name were considered electrically equivalent. A remaining issue in doing routing over the cell is how to handle the case when the router runs out of feedthroughs. To handle this, the router lays out a feedthrough in a global channel above the leafcell, essentially extending the height of the leafcell. Figure 2.2 shows an example of this.

The routing is now done by splitting the routing into a number of small channel routing problems. First, a global pass is made across the slice to assign the various nets to the feedthroughs or to create feedthroughs in the area above the leafcell if the leafcell does not have enough feedthroughs. This step together with the fact that the feedthrough terminals are just like the data terminals reduces the routing problem to the problem of connecting the terminals of two adjacent cells. This is then done by using a greedy channel routing algorithm, treating the space between two adjacent leafcells as a channel. The terminals are on the left and the right side of the channel. Some of the terminals would be the proper data terminals of the leafcell while the others would be feedthrough terminals for nets coming from some cell which is farther away.

The overall routing strategy can thus be described as one of divide and conquer where the total routing problem is reduced to a number of simple channel routing problems. There are many well established algorithms for channel routing which can then be used. In this system a greedy algorithm described in the next chapter was used. Figure 2.2 shows

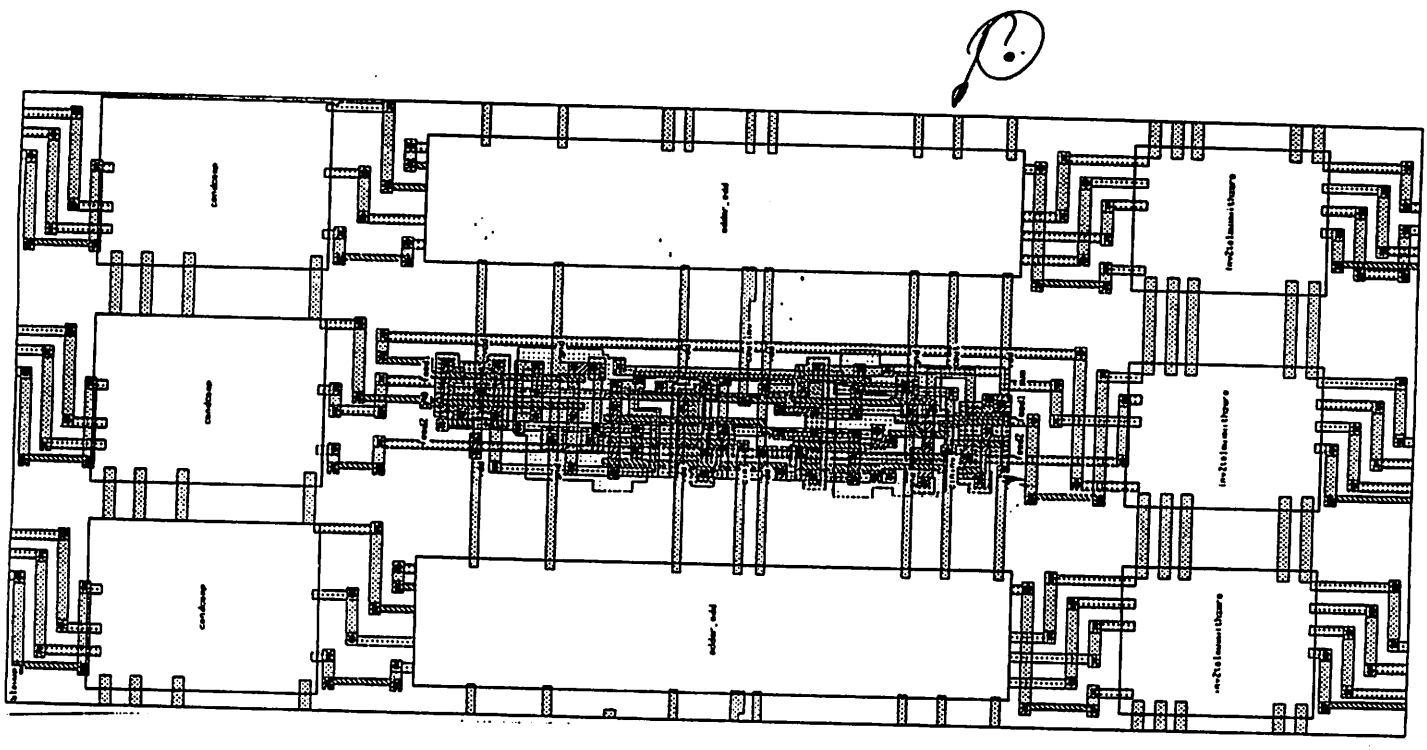


Figure 2.2: Example of channel routing

a channel routing while figure 2.3 shows the plot of a complete data path routed using the system.

2.4 Constraints on the leafcells

The routing strategy imposes certain constraints on the leafcell design. The constraints arise because of the choice of layers. Since polysilicon and metal2 cannot connect directly to each other and the data terminals can come out in polysilicon, it was decided to use metal1 in the vertical direction within the channels between the leafcells. The data terminals can then come out in either polysilicon or in metal2. Further, since the router may need to place metal2 feedthroughs above a cell, the control, power, ground and clock lines which run vertically cannot come out in metal2. Thus, the terminals at the top or the bottom are constrained to come out in metal1 or polysilicon. Figure 2.4 illustrates this clearly. Performance issues usually restrict the choice to metal1 only. Besides the constraints on the layers on which the terminals, there are some spacing constraints and some conventions adopted for our cell library. These constraints are listed below:

1. Although it is not necessary, one should try to keep the heights of all cells nearly equal. This results in better area efficiency.

2. The width of cells used to make up a functional block *must* be the same. For example, the cells at the even and the odd bit positions of an adder must have equal width.

3. Any layer, including the well can be used inside the cells.

4. All the data terminals *must* come out at the sides and should be in one of the following layers: M2, VIA, POLY, POLYCONTACT.

5. The data terminals should be "sufficiently" apart. This is rather vague since the terminal spacing requirement depends on the net connection to a certain extent. A safe rule of thumb is to have the terminals at $>= 4\lambda$ apart and the terminals themselves 4λ wide in the SCMOS technology. This will always work but leads to wasted area in many cases. A better spacing requirement is that the terminals be spaced such that one should be able to place minimum sized contacts to M1 next to them, such that all the contacts lie on the same vertical column. Such a situation arises in the left channel in figure 2.4.

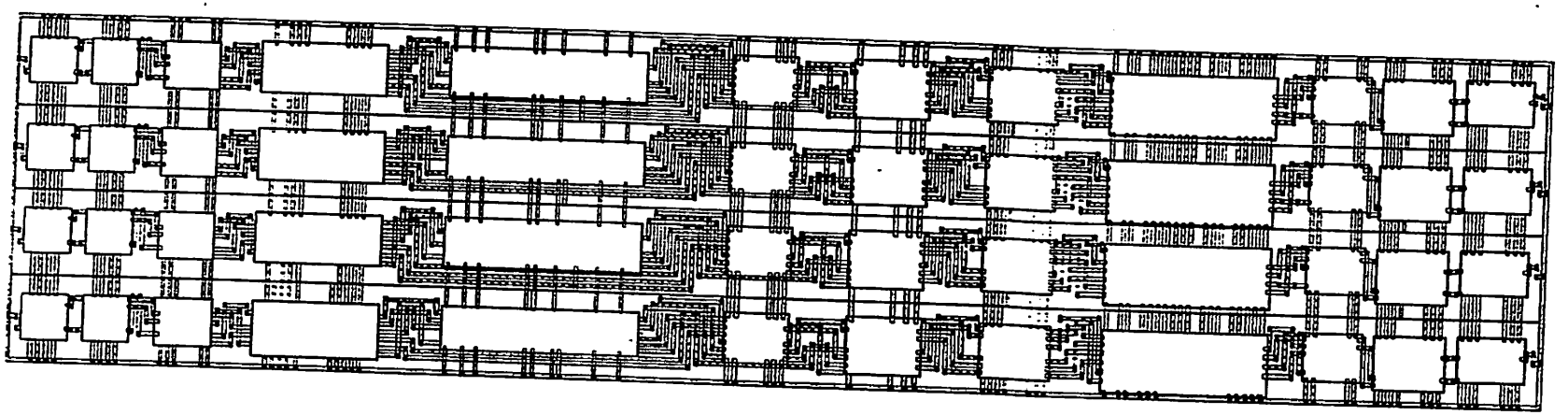


Figure 2.3: Example of a completely routed data path

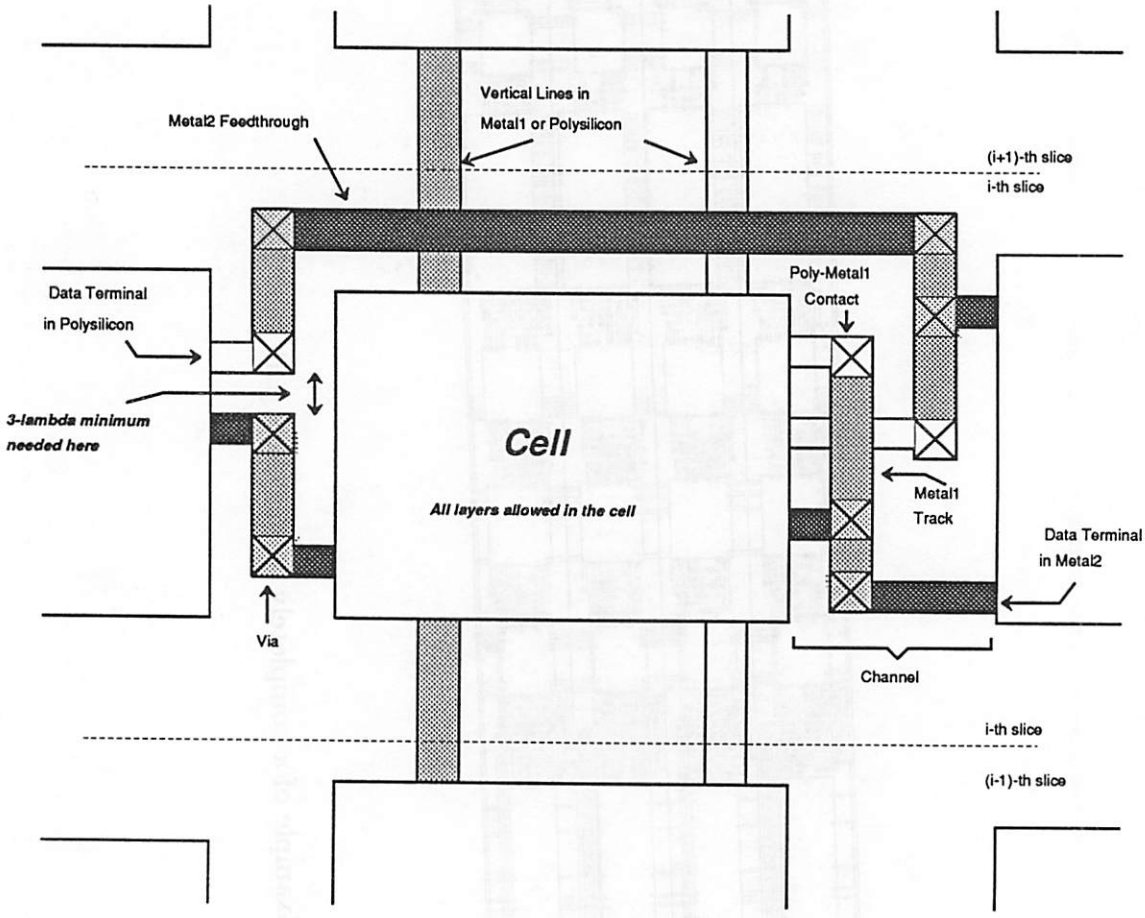


Figure 2.4: Detailed view of the channel routing

6. The control terminals, power, ground and clock lines of the cells should come out at the top and bottom edges in one of the following layers: M1, POLY and POLYCONTACT. Note that the program does not allow the top and bottom terminals to come out in DIFFUSION and DIFFUSIONCONTACT. These terminals are extended to meet the top or bottom edge of the slice. Further, the top terminals of the msb slice and the bottom terminals of the lsb slice are brought up to vias for compatibility with the macrocell router FLINT. This requires that the top and bottom terminals of the cells be sufficiently wide and distant from each other. As a rule of thumb, in MOSIS SCMOS technology, these terminals should be >= 4λ wide and >= 4λ apart, or 3λ wide and 5λ apart, provided no design rules are violated for the particular terminal layer.

7. To enable better routing performance, one should try to provide "feedthroughs" in the cells. These are pairs of terminals, one on the left side and the other on the right side of the cell, which are connected to each other and are NOT connected to anything inside the cell. A good strategy is to provide these in M2 in order to avoid high capacitance associated with POLY. The feedthrough pairs should be labelled FEEDn where n is a positive integer.

8. All the terminals should be labeled, the label being put on the layer on which the terminal is coming out. The box corresponding to the label denotes the location of the terminal. There should be no useless labels at the edge of the cells because all labels at the edge are treated as terminals.

2.5 User's View of the System

The system encourages the user to think about the data path as an interconnection of functional blocks, each of which perform some N bit operation. The construction of each of these functional blocks is hidden from a normal user. In other words, what cell to use at a particular bit position is hidden, unless the user wants to design a new functional block. The structure of these functional blocks is described procedurally in a separate file associated with the functional block. The procedure describes the generation of the functional block using the leafcells in the library. The generation can be guided by the values of certain parameters. This is described in detail in section 4.1.

The overall design cycle consists of making a block diagram of the data path using functional blocks from a system library. The input is then a net list description of this block diagram using the *sdl* syntax, which is a language used to describe the structure of macrocells in the silicon assembly environment used by the designers in our group. The details of the input syntax can be found in [3]. Several examples are given later in chapter 6 of this report.

The system is extensible in the sense that advanced users can design their own leafcells and functional blocks to meet their specific need. Chapter 4 describes in detail the library organization for using this tool.

Lastly, this tool is interfaced with *Design Manager* which is a supervisory program for the integrated CAD environment used in our group (see [3]). This allows this data path generator to make use of the simulation facilities provided by the *Design Manager*. In particular, the user can perform functional simulation of a data path interfaced with other macro cells in a chip using the same sdl description as is used for layout. Figure 2.5 shows the block diagram of the overall system interfaced with the Design Manager.

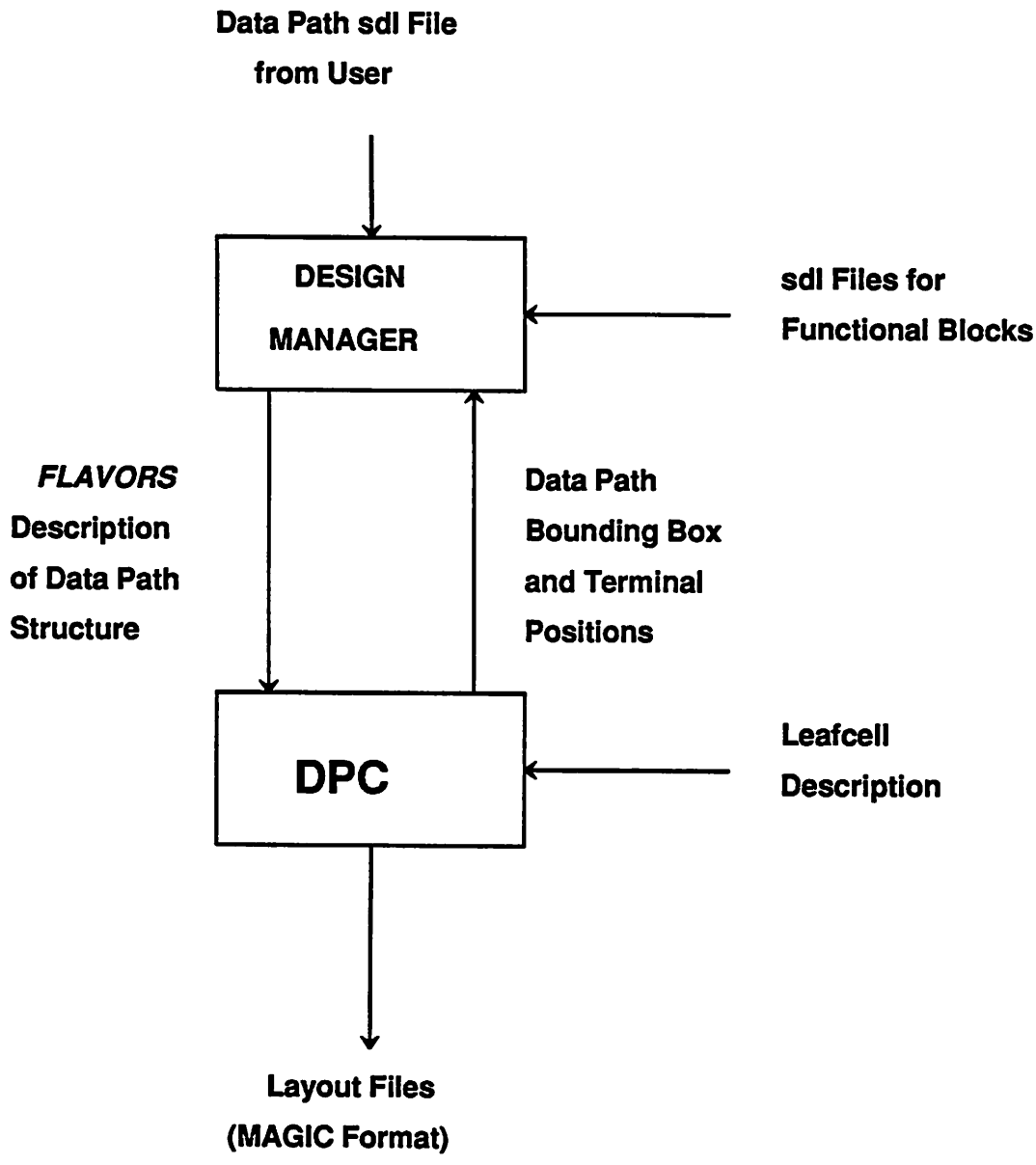


Figure 2.5: Overview of the Data Path Generation System

Chapter 3

Algorithms Used and Software Organization

This chapter describes the algorithms used in the various part of the program and the overall organization of the software. A description of the interface with the Design Manager is also given.

3.1 Choice of LISP and Object Oriented Programming

The program has been implemented using Franz Lisp with the Flavors system [1] which provides an object-oriented programming environment. This choice was motivated by several reasons, the foremost being to explore the suitability of Lisp and object-oriented programming for module generation tools. Conventionally, C has been the language of choice for such tools, primarily because of the speed advantage. However, as shown later in the report, the execution time of the program is acceptable for reasonably sized data paths when compiled Lisp is used. Any speed disadvantages are offset by the ease of code development offered by the LISP environment, in particular in the area of user interface and debugging.

The object-oriented programming style under the framework of Flavors provided an interesting programming paradigm with a much cleaner organization of software. The advantages of this style were particularly apparent when interfacing with the Design Manager.

3.2 Organization of the program

The program has been partitioned into three distinct portions corresponding to three distinct phases in the overall process of data path generation. This partitioning helps in treating the program as three independent pieces of code and thus results in easier code maintenance and debugging.

The first part consists of the code for interfacing with the Design Manager and for figuring out the way each individual functional block is made using the leafcells. This part basically handles the extraction of useful information from the flavor objects in the Design Manager and repacks them into structures used internally by the program. It sets up structures corresponding to each slice after figuring out which cell to use for a particular block at a certain bit position. Basically, it converts the netlist in terms of functional blocks into a set of netlists (one for each bit slice) in terms of leafcells. For doing this mapping, use is made of the cell mapping methods and terminal mapping methods associated with each functional block. These methods are defined by the block designer in the corresponding sdl files. Since the data path generator was written before the Design Manager, this interface is not the best possible since the internal data structures used by the program were designed independent of the Design Manager specifications. This part also handles the repackaging of the output information of the data path generator into the flavor objects of the Design Manager. This output information consists of all the interface information, such as the bounding box and the terminal locations, which are required by the tools operating at a higher level in the layout hierarchy. This portion of code resides in a single file *dpc.l* and consists of a single method declaration for the dpc-mixin flavor [2].

The second part of the code takes up the structures set up by the code in the first part and routes each unique bit slice. This is done by converting the slice routing problem into a set of channel routing problems which are then handled by the third part of the code. For doing this division of the overall problem, the code goes through two distinct phases. In the first phase, nets are allocated to the various terminals. The problem of carrying a net over a cell is also handled in this phase by allocating such a net to a feedthrough provided in the cell or laying out a feedthrough line at the top of the cell in a global channel. After this net allocation process, the problem basically reduces to routing each of the local channels between the adjacent cells. This is done by making repeated calls to a channel routing function which constitutes the third part of the code. After completing the

routing for all slices, this part of code creates the physical layout files in the Magic format [6] taking advantage of the hierarchical layout organization allowed by Magic. Magic format was chosen because of the wide usage of magic as the layout tool by most layout designers. However, the code is modular so that switching over to a new format will not be difficult. In fact, a new version of the tool which uses *Oct* [16] as the database has been developed.

The third part of the code handles the channel routing problem. This function basically takes a list of terminals along the two sides of a vertical channel and routes the channel. It makes the channel as wide and as tall as required to complete the routing. The routing is guaranteed to be completed as long as the cell terminals meet the spacing requirements and come out on the proper layers. The algorithm used is Rivest-Fidducia's Greedy Algorithm which, though not optimal, works adequately because of the small size of routing channels. The modularity of the code permits the current routing algorithm to be replaced easily by a better algorithm in future. This part of the code is spread over several files (*ch_route.l, ch_merge.l, ch_newtrack.l* and *ch_suitable.l*) and is organized as a function *route* which does the complete routing by making calls to functions *createtrack, try_to_merge, join_tracks* and *suitable* which handle specialized sub-jobs related to the routing process. The algorithm used for the channel routing is described in more detail in the next section.

Finally, the code makes use of flavor objects defined by the Design Manager and also uses the various utilities provided by the Design Manager environment. In other words, this program uses the Design Manager as a front-end and is meant to be used within the Design Manager framework.

3.3 Algorithm used for routing

The overall algorithm followed by the program consists of the following major steps:

1. Read in the input files in the sdl syntax (this is done by the Design Manager) and convert into the flavor objects according to the Design Manager convention. Send the layout generation message to the data path object at the appropriate time with the proper parameters

2. Using the block level netlist and the mappings associated with each block, determine the netlists in terms of the leafcells for each slice

3. Make a list of the unique slices
4. For each unique slice, first do the net allocation and then do all the channel routings
5. Output physical layout files corresponding to each unique slice
6. Generate geometry associated with the external terminals of the data path
7. Output the physical layout file for the complete data path using instances of the layouts of the slices
8. Put the information about the interface (bounding box, terminal locations) in the data path object and output the hdl file for use by module generators like Flint [4] and PadRouter [5]

The steps dealing with the front-end Design Manager and the output format are straightforward and not of much interest. The core of the whole process is the method used for placement and routing. Following is the algorithm used for this in a pseudo-C syntax:

```
CHANNEL_ROUTE(leftlist, rightlist)

{
commonlist=merge(leftlist,rightlist);

for each terminal in commonlist find the terminals which may
create vertical violations; /*at most two such terminals on the
opposite side*/

PREVTERM=nil;
EMPTYTRACKLIST=nil;
SPLITLIST=nil;
TRACKLIST=nil;
/*visualize the channel to be vertical*/
```

```
for (termptr=commonlist; termptr!=nil; termptr=termptr->next)

{
CURRTERM=termptr;
NEXTTERM=termptr->next;
CURRNET=net(CURRTERM);
CURRTRACKLIST=whichtracks(CURRNET);
CURRTRACK=nil;
SPLIT=no;
if (CURRTRACKLIST!=nil) /*there are tracks carrying the net*/
then
    {
    /*there are tracks carrying the net*/
    CURRTRACK=findtrack(CURRTERM,CURRTRACKLIST);
    if (CURRTRACK==nil) then SPLIT=yes;
    }
if (CURRTRACK==nil) /*there are no suitable tracks carrying the
                      net*/
then
    if (EMPTYTRACKLIST!=nil)
    then
        /*there are empty tracks*/
CURRTRACK=findtrack(CURRTERM,EMPTYTRACKLIST);
if (CURRTRACK==nil) /*no suitable empty tracks*/
CURRTRACK=newtrack(CURRTERM);
    else
        /*no empty tracks*/
CURRTRACK=newtrack(CURRTERM);
if (SPLIT) then addsplitlist(CURRNET,CURRTRACK);
mergesplitnets(); /*try merging some of the split nets and put
                    freed tracks in the EMPTYTRACKLIST
                   */
jognets();        /*try jogging some of the nets*/
```

```
freetracks();      /*free the tracks whose nets are done with and
    add those tracks to the EMPTYTRACKLIST
                    */
}
/*now we are done with channel route but there may be split nets
  remaining
*/
mergeallsplitnets();
}
```

As already mentioned, the greedy algorithm used for the channel routing is a very simple one and better results will be obtained by using a more sophisticated router. This is one possible area of improvement. Another drawback in the current algorithm is the lack of automatic placement.

3.4 Interface with Design Manager

This tool was interfaced with the Design Manager which is the supervisory program for the integrated CAD environment used by our research group. The Design Manager provides a clean interface for new module generators to be integrated into the CAD environment. It supports an object oriented paradigm based on Franz Lisp Flavors package. Basically, the Design Manager provides all the front end processing and passes all the information to the module generators in form of various objects. The tools can access the desired information by sending appropriate *messages* to the various objects.

Interfacing with Design Manager, however, requires that the module generators follow certain conventions. For easy interfacing as well as for fully exploiting the facilities provided by the Design Manager, it is better for the module generator itself to be in Lisp. This was a major reason for choosing Lisp as the language of implementation. Although it is definitely possible for a program in a language like C to be interfaced with the Design Manager, it is somewhat inefficient because of the complex foreign function support in Lisp.

A full documentation of the Design Manager interface can be found in the *Lager III Programmer's Manual* [2]. Basically, there are three types of objects seen by the tool:

cell, net and terminal. The tool itself is called by sending a layout generation message to the cell object corresponding to the data path (or, some other macrocell for other tools).

The various information about the various objects can be accessed by sending suitable messages. For example, by sending a message named :instance-net-list to the cell object, one can get a list of all the net objects attached to the cell.

The DPC has been fully interfaced with the Design Manager using the interface provided. This interfacing has resulted in several advantages. First, the tool is now part of the silicon assembly process supported by the Design Manager so that DPC can now be used to generate automatically various macrocells in a complete design hierarchy. This required the output layout to be compatible with the requirements of tools like *Flint* and Mosaico which are used for macrocell placement and routing. Secondly, the tool uses the Design Manager as the front end so that the user input is in the same syntax as used by the other tools in the design environment, namely the *sdl* syntax. Thirdly, and probably most important, the Design Manager provides extensive functional simulation support for the data paths designed using this system. There are functional simulation models associated with every functional block used in data paths. These models are basically lisp functions. The simulation works on the sdl input file so that the user need not generate the layout in order to be able to do the simulation.

All the software associated with this interface is in the file *dpc.l*. However, since DPC was written before the Design Manager interface came into existence, the internal data structures used in DPC are different from the objects provided by the Design Manager. As a result, translation is required in the interface software.

Chapter 4

Cell and Block Libraries

This chapter describes the leafcell and functional block libraries used by the tool. The main purpose is to document the current status of the libraries. A description of the various cells and blocks along with their intended use, the various associated files, the layout and circuit diagram and timing diagrams if any are given.

4.1 Block Library

As already mentioned in the earlier chapters, the basic entity used by the data path designer is a functional block. Although the user can use his own functional blocks, a system library of such functional blocks has been designed which can be used by the user directly. At the time of writing this report, the library had a large number of functional blocks which have been used in many varied applications including a DSP for a Robot Controller, an image processor for doing Radon Transform and in the generic Lager processor. The library has been designed through the joint effort of a number of people and appears to be sufficient for almost all the data path design needs encountered by the various users in our research group.

This library basically consists of a sdl file associated with each functional block. However, there is no netlist description in the sdl file. Instead the sdl file describes the block by specifying what leafcell is to be used at a particular bit position. The mapping from the block terminals to the leafcell terminals at each bit position is also specified in this file. These mappings are given as lisp functions and using the parameter mechanism of the sdl syntax and the control flow constructs provided by lisp, one can have fairly complex

programmability. To ease the job of writing these mappings, some lisp macros have been defined which should be used in order to have a friendly syntax.

An example sdl file of an ripple carry adder block (see figure 4.1) is given below:

```
(parent-cell adder (parameters version type))
(lisp-function

(deftermmap adder
(if (and (equal terminal "CIN") (= 0 i))
  then (useterm "cin")
elseif (and (equal terminal "CIN*") (= 0 i))
  then (useterm "cininv")
elseif (and (equal terminal "COUTn") (= i msb))
  then (useterm "cout")
elseif (and (equal terminal "COUTn*") (= i msb))
  then (useterm "coutinv")
elseif (and (equal terminal "COUTn-1*") (= i msb))
  then (useterm "cininv")
elseif (and (equal terminal "Ain") (oddp i))
  then (useterm "A1")
elseif (and (equal terminal "Ain") (evenp i))
  then (useterm "A0")
elseif (and (equal terminal "Bin") (oddp i))
  then (useterm "B1")
elseif (and (equal terminal "Bin") (evenp i))
  then (useterm "B0")
elseif (equal terminal "SUM")
  then (useterm "sum")
else (useterm terminal)
  )
)

(defcellmap adder
```

```
  (if (= msb i) then (usecell "adder_msb")
  elseif (evenp i) then (usecell "adder_even")
  else  (usecell "adder_odd"))
)
)
```

The first line in the above example defines the name of the parent cell and its parameters. In this case, the name is *adder* and there are two parameters, *version* and *type*. Next, the terminal mapping and the cell mapping are given. These two are given as arguments to a specially defined lisp function called *lisp-function*, which is a keyword in the sdl syntax. The terminal mapping is primarily used as a translation mechanism so that leafcells with different terminal names may be used within the same block. This mapping is defined using the special lisp macro *deftermmap*. This is to be used as a lisp function, the first argument to which is the name of the block and the second argument is a lisp expression which preforms the mapping. In the example, lisp control flow function *if ... then ... elseif ... then ... else...* has been used to define the mapping. The first clause in the statement says that if the block terminal name is **CIN** and the bit position is the least significant bit then the leafcell terminal **cin** is used. The remaining *else* clauses define the mapping for the other terminals. The more interesting mapping is the cell mapping which describes which leafcell is to be used at a particular bit position. In the example above we have a very simple cell mapping which says that the cell *adder_msb* is to be used in the most significant bit position, the cell *adder_even* is to be used in the even bit positions and the cell *adder_odd* is to be used in the odd bit positions. This mapping is done using the lisp macro *defcellmap*.

At the current time, the block library consists of the following functional blocks:

- domino dynamic latch with non-inverting output
- one-phase dynamic latch with inverting output
- two-phase dynamic latch with inverting output
- scanpath one phase dynamic latch with inverting and non-inverting outputs
- clocked inverter with non-inverting output

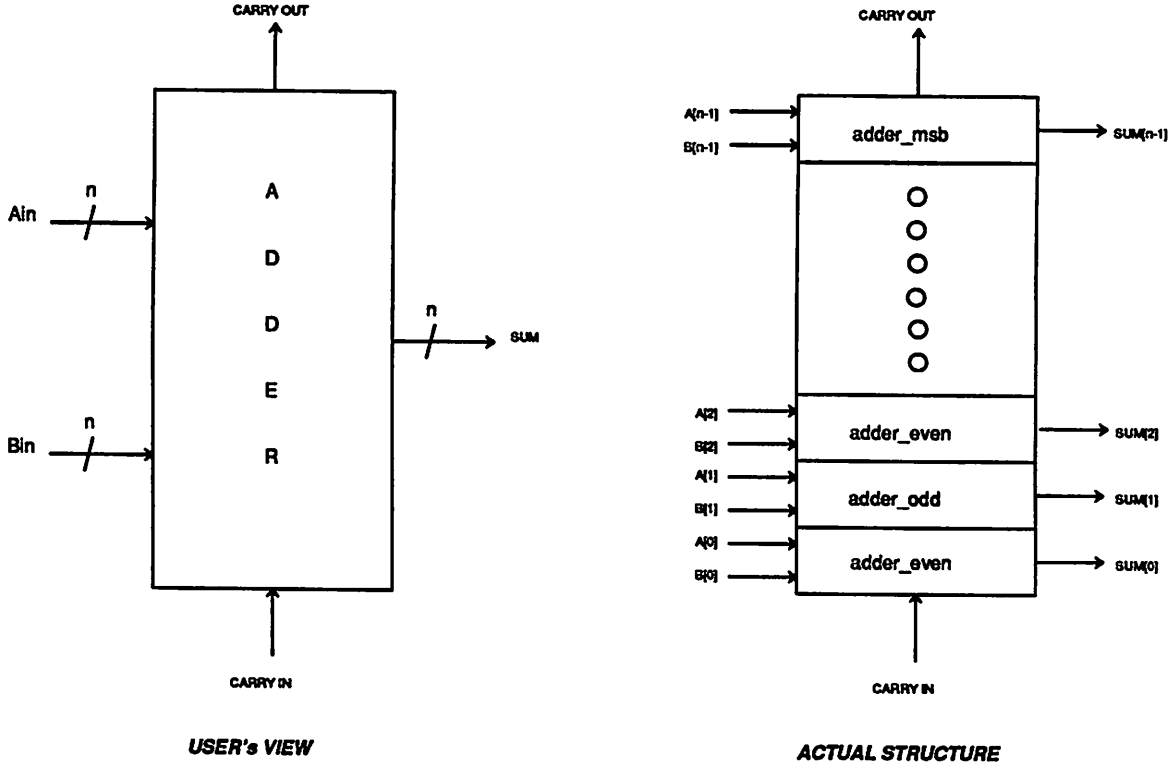


Figure 4.1: Structure of the Ripple Carry Adder

- clocked inverter with inverting output
- 2-to-1 multiplexor with inverting output
- 2-to-1 multiplexor with non-inverting output
- 2-to-1 multiplexor with invering output and control signal to force the output to be zero
- 4-to-1 multiplexor with inverting output
- constant register with value zero
- binary up-counter with parallel load
- ripple carry adder
- single-port static register
- single-port static register for use in scan path
- dual-port static register
- barrel shifter which shifts upto 6-bits one way and 1-bit the other way
- one-phase accumulator for Lager processor
- two-phase accumulator for Lager processor
- memory IO block for Lager processor
- miscellaneous random logic

The library has proven to be sufficient for many applications involving fixed-point arithmetic. It is however continuously being enhanced by the various users.

4.2 Leafcell Library

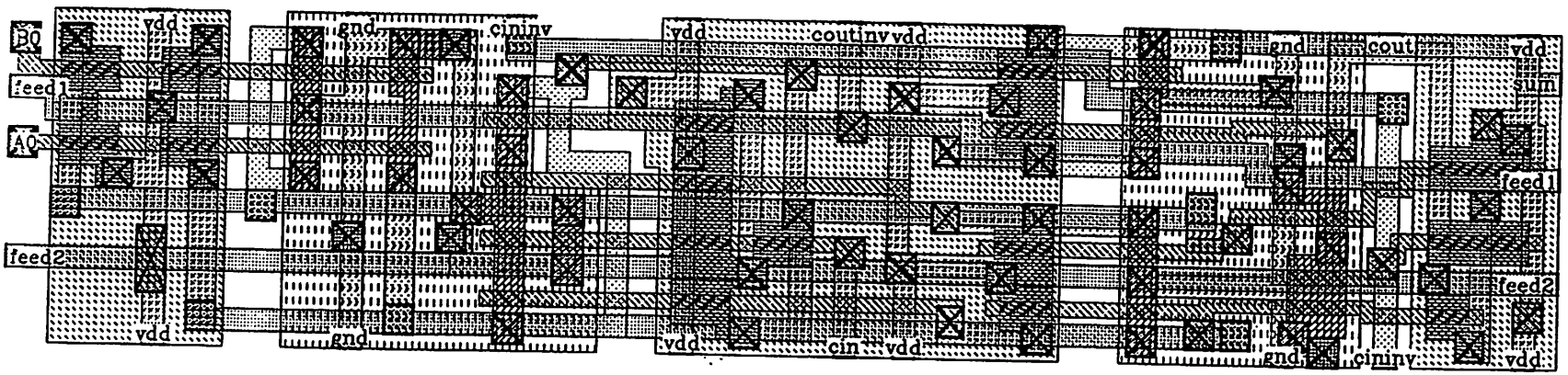
The leafcell library consists of all the leafcells required to support the functional blocks in the block library. Note that many functional blocks use more than one leafcells and that the same leafcell may be used in more than one block. The leafcells have to

follow certain layout restrictions which were outlined in section 2.4. At present the layout files are accepted only in the magic format (see [6]). Associated with each layout file is a *cell descriptor* file (known as the *cd* file) which gives the interface view of the cell, *i.e.* it gives the bounding box coordinates, the terminal positions and layer and the feedthroughs. This information is automatically extracted from the magic file by the tool and a *cd* file is created, so that the next time the cell is used the tool has to do less work. The tool compares the modification dates of the layout file and the cell descriptor file and a new cell descriptor file is created if the layout file had been recently modified. These cell descriptor files can however be created using the command *makecd* which can be used from Unix. These files also happen to form a good documentation of the cells. Following is an example cell descriptor file for a leafcell used in an adder:

```
cell adder_even -2 -63 210 -18
left A0 -38 -34 polycontact
left B0 -24 -20 polycontact
bot cin 111 114 metal1
equivalent (bot cininv 185 188 metal1) (top cininv 66 70 metal1)
top cout 185 188 metal1
top coutinv 111 114 metal1
feedthru (left feed1 -30 -27 metal2) (right feed1 -39 -36 metal2)
feedthru (left feed2 -53 -50 metal2) (right feed2 -53 -50 metal2)
equivalent (bot gnd 44 47 metal1) (top gnd 44 47 metal1)
(top gnd 171 174 metal1) (bot gnd 171 174 metal1)
right sum -26 -23 metal2
equivalent (top vdd 17 20 metal1) (bot vdd 17 20 metal1)
(bot vdd 89 92 metal1) (bot vdd 119 122 metal1) (top vdd 119 122 metal1)
(bot vdd 204 207 metal1) (top vdd 204 207 metal1) (top vdd 89 92 metal1)
```

In the example above, *cell*, *left*, *bot*, *top*, *right*, *equivalent* and *feedthru* are keywords which are part of the syntax of the cell descriptor files. The first line of the file should always begin with the keyword *cell* and defines the name of the cell and the coordinates of its bounding box. The remaining lines give the names, positions and the layers of the various terminals. The terminals which are electrically equivalent are grouped using the keyword *equivalent*. However, electrically equivalent pairs of terminals meant to be used as

feed-throughs are grouped using the keyword *feedthru*. Figure 4.2 shows the layout of the above cell.



adder_even $(45\lambda \times 212\lambda)$

Figure 4.2: Layout of the cell adder_even

Chapter 5

Control Interface Circuitry for Data Paths

In this chapter the problem of generating the interface logic between the data path and the rest of the chip is discussed. This interface logic typically deals with decoding the control microword (coming from the control ROM or PLA) into the signals needed by the data path control points. This decoding usually is not just plain combinational decoding but also involves qualification of the control signals by the various clock phases. This is needed in order to properly schedule the various suboperations in a clock cycle. Another use of this interface logic is in implementing conditional data operations wherein the signals from the controller are gated with some conditional flags coming out of the data path. Besides the pure logic operations, the interface block also carries out buffering operations. As we show in the following section, this interface logic does not fit into the bit slice scheme and therefore a separate methodology is required. A standard cell approach was used for this and a frontend tool called eqn2sdl was written which allows the specification of this interface circuitry as a set of lisp expressions. The interface circuit generated by this approach together with the bit slice macrocell generated by the data path generator forms the complete data path.

5.1 Unsuitability of bit slice strategy for control interface

Traditionally, the data path designers tended to include this interface circuitry as a slice in the bit slice configuration. Such a slice is usually referred to as the control slice

of the data path. However, we realized that this interface logic is very much application dependent. For example, the data processing part of the data path may be the same but the interface logic can be different depending on the way the microwords are encoded. Consequently, if we were to associate the interface logic with each of the functional blocks in our data path, we would need to have a very large number of functional blocks which have the same data processing function but different interface logic. To make matter worse, this interface logic is very much application dependent so that our philosophy of having a reusable set of functional blocks would be rendered useless. To study the implementation issues in detail, the interface logic for the Lager AUIOINC data path was manually designed as a control slice using a set of cells associated with the functional blocks. The disadvantages were apparent very quickly. Besides the issues raised above, it also became apparent that this strategy is not good in terms of area also. The reason is that there usually is very little correlation between the complexity of the data processing carried by a block and the complexity of its interface logic. Consequently, it turns out that some very simple functional blocks have quite complex interface logic associated with them. This results in empty space in the control slice. Note that in the traditional manual design, this problem does not occur because the whole control slice is designed as a single cell.

This unsuitability of the bit slice strategy for the interface logic prompted us to study an alternative strategy. Instead of considering the interface logic as part of the data path macrocell, it is treated as a separate macro cell and the complete data path consists of the two macrocells, one generated by the data path generator program discussed in the previous chapters and the other consisting of the interface logic. This partitioning simplifies the task and the functional blocks now consist of only the data processing functions which are application independent most of the time and can therefore be used as primitives for a variety of data paths. However, to ease the design of the macrocell for the interface logic, one needs some sort of automation of the task. The approach used for this is discussed in the next section.

5.2 Use of Standard Cells for control interface

The strategy chosen for the automated generation of the interface logic is based on the same idea that has been used for the data processing part. Basically, the interface logic is made up of reusable primitives. Since the interface logic is just a collection of miscellaneous

logic functions, latches and flip-flops with no particular regularity in the signal flow, it was decided to use a general purpose scheme like the standard cells. This approach has been used in a number of designs in the recent years and its main advantage lies in that it is equally easy to have both combinational logic and latches or flip-flops. This is unlike the array structures such as PLAs where it is rather difficult to incorporate random latches and flip-flops. The design style consists of using a cell library where all the cells have the same height. The cells are then placed in one or more horizontal rows and then routed using the channels between the rows. By having a well designed cell library, one can reuse it for a very large number of applications. These features plus the fact that a large number of very efficient design aids already exist for standard cell design style were the reasons behind choosing this style for implementing the interface logic. Another factor which was taken into consideration was the availability of a very extensive standard cell library. Further, the ability of the Design Manager to use different module generators at different points in the design hierarchy made this strategy viable without much effort.

5.3 Generation of macrocell using Design Manager and Wolfe

Design Managers ability to handle a variety of module generators using the same input syntax converts the problem of generating the interface logic macrocell to one of writing the sdl file with the appropriate layout generator which in this case is *stdcell*. Thus the complete data path consists of two macrocells, one of which is generated using the layout generator *dpc* and the other using *stdcell*. These two are then interconnected with each other and may be to other macrocells at a level above in the design hierarchy using a module generator which does macrocell placement and routing, for example *Flint* or *Mosaico*, both of which are now supported by the Design Manager.

The standard cell placement and routing are done by the Design Manager using a tool called *Wolfe* [13] which is a tool provided under the environment of the *Oct* database [16]. The Design Manager basically converts its internal netlist representation into the proper Oct representation or facet and then fires up Wolfe. Wolfe in turn uses a simulated-annealing based tool called *Timberwolf* [14]. The output of Wolfe, which is another Oct facet, is then read back by the Design Manager.

5.4 Structural description too cumbersome: need for a better way

The sdl input can be quite large for moderately complex blocks of logic. For example, the interface logic for the AUIOINC data path uses nearly fifty standard cells which results in a fairly large sdl file. Also, the net list description does not give a feel for the functionality of the logic. What is required is some sort of functional or behavioral description of the interface logic. Behavioral description in an absolutely general case is a very tough problem and is an area of current research. However, what is needed here is some behavioral representation which works for this restricted domain of expressing the data path interface logic and some tool to translate the description into a structural description in the sdl syntax. Although there exist powerful representations like bdsyn [15] which can be used to express combinational logic, they are not compatible with the Design Manager environment. Also, bdsyn can handle combinational logic only. It is not possible to specify latches, which are usually an essential part of interface logic. Therefore, it was decided to write a simple front-end tool which takes the description of the interface logic and converts it to a sdl description. This is a short-term solution only. A version of bdsyn which is compatible with the Design Manager environment and has appropriate enhancements to handle latches will be a better long-term solution. The tool is described in the following sections.

5.5 Tool for generating structural description of interface logic from its functional description

The tool is called *eqn2sdl*, which stands for equation to sdl translation. It takes in a description of the interface logic as a lisp program using some predefined functions and converts it to a sdl input file for the Design Manager. The input syntax for this tool is nothing but enhanced lisp. A set of predefined functions is provided to the user and the user writes lisp expressions using them. These predefined functions map to one or a group of standard cells which is hidden from the user. The user can define his own new functions based on these predefined functions or can define his own primitive function based on a new cell. The program treats the input as a program and generates the corresponding net list in the sdl syntax. The only optimization which is currently done is sub-expression

elimination and using multiple output cells. The program can handle recursive expressions also, allowing the user to express asynchronous or feedback logic.

All the lisp functions defined by the tool are distinguished by an exclamation sign (!) at the end. These functions are used to express the logic of the macrocell as a set of expressions. The arguments to these functions are other such expressions or variables declared by the user. The variables are of two main types: formal variables which correspond to the parent terminals in the sdl output and local variables which are meant for convenience in writing the expressions. The formal variables are declared using the *parent!* command and the local variables are declared using the *var!* command. These variables can either be scalar or one-dimensional vector. The expressions *(parent! x (y 10)) (var! (a 10) b)* declare a scalar formal variable x, a vector formal variable y of dimension 10, a vector local variable a of dimension 10 and a scalar local variable b. A third type of variable is the parameter variable which is declared using the *parameter!* command. These are used to provide parameterizability and are interactively input at the beginning. The function *set!* is provided to assign the value returned by an expression to a formal or local variable. The remaining functions provided by the tool are functions providing some logic or arithmetic functionality. Most commonly used functions like or, and, nand, nor, not, sum, carry, delay, multiplex, decode etc. have been provided. Many functions implementing complex logic expressions and flip-flops are also included in the package. Please refer to the manual page for the tool (see appendix B) for an exhaustive list of the functions.

The input is given as a lisp program. The control flow provided by lisp in conjunction with the parameter facility provides a mechanism for tailoring the specification of the interface logic according to some parameters. It is important to note that the lisp control flow functions should not be used to describe the control flow behavior of the interface logic. The logic is generated from the expressions using the logic functions provided by the tool. The description is treated as a program and therefore the order of expressions may matter.

Figure 5.1 shows the input file to the eqn2sdl tool describing the interface logic of the Lager AUIOINC data path and figure 5.2 gives the corresponding gate level description. As we see, the syntax allows the user to express the functionality of the logic without worrying about how it is implemented and is extremely straightforward to write given the logic diagram. Besides the convenience of description, it also separates the functionality from the implementation so that the same input file can be used if a different standard cell library or, for that matter, an entirely new implementation style is used.

```
;declare the variables corresponding to the external terminals
(parent! ZEROB* MEMB* ACCB* XMIT-ACC* XMIT-MOR* ZEROB MEMB
ACCB XMIT-ACC XMIT-MOR XMIT-ACCbar XMIT-MORbar WEN WEN.PHI1
WRITELATCH WLATCH.PHI2 WLATCH.PHI2bar SHIFT* SHIFT LOAD COEF1
ZEROA1 COEF2 ZEROA2 ZEROA3 QUOT ZEROA SOR* INV1 INV2 INV PHI1
PHI2 A1P A1P.PHI2 A1P.PHI2bar SIGN* NOFbar SIGN POF CO CIN*
ACC1IN ACC2IN S0 S1 S2 BS0 BS1 BS2 BS3 BS4 BS5 BS6 BS7)

;declare the local variables defined for ease of writing
(var! temp1 temp2 A4P A2P tempS0 tempS1 tempS2)

;declare some useful functions
(defun del1! (x) (del! x PHI1))
(defun del2! (x) (del! x PHI2))

;describe the logic
(set! ZEROB (not! (del2! ZEROB*)))
(set! MEMB  (not! (del2! MEMB* )))
(set! ACCB (not! (del2! ACCB* )))
(set! XMIT-ACC (not! (del2! XMIT-ACC*)))
(set! SHIFT (not! (del2! SHIFT*)))
(set! XMIT-ACCbar (not! XMIT-ACC))
(set! LOAD  (not! SHIFT))
(set! XMIT-MOR (not! (del2! XMIT-MOR*)))
(set! XMIT-MORbar (not! XMIT-MOR))
(set! WEN.PHI1 (and! PHI1 (del2! WEN)))
(set! WLATCH.PHI2bar (nand! WRITELATCH PHI2))
(set! WLATCH.PHI2 (not! WLATCH.PHI2bar))
(set! ZEROA (not! (del2! (xnor! ZEROA3
        (and/nor2_2! COEF1 ZEROA1 COEF2 ZEROA2)))))
(set! INV (not! (del2! (xnor! INV2 (or! SOR* INV1)))))
(set! A4P (not! (del2! A1P)))
(set! A2P (del1! A4P))
(set! A1P.PHI2bar (or/nand2_1! PHI2 A2P ACC1IN))
(set! A1P.PHI2 (not! A1P.PHI2bar))
(set! SIGN (not! SIGN*))
(set! temp1 (del! (not! (del2! SOR*)) (and! PHI1 A4P)))
(set! temp2 (nor! A2P ACC2IN))
(set! QUOT (xor! temp1 temp2))
(set! POF (nor! (del1! CO) (del1! CIN*)))
(set! NOFbar (nand! (del1! CO) (del1! CIN*)))
(set! tempS0 (not! (del2! S0)))
(set! tempS1 (not! (del2! S1)))
(set! tempS2 (not! (del2! S2)))
(vset! (BS0 BS1 BS2 BS3 BS4 BS5 BS6 BS7) (dec3to8! tempS0 tempS1 tempS2))
```

Figure 5.1: Input to *eqnsdl* for Interface Logic for AUIOINC Data Path

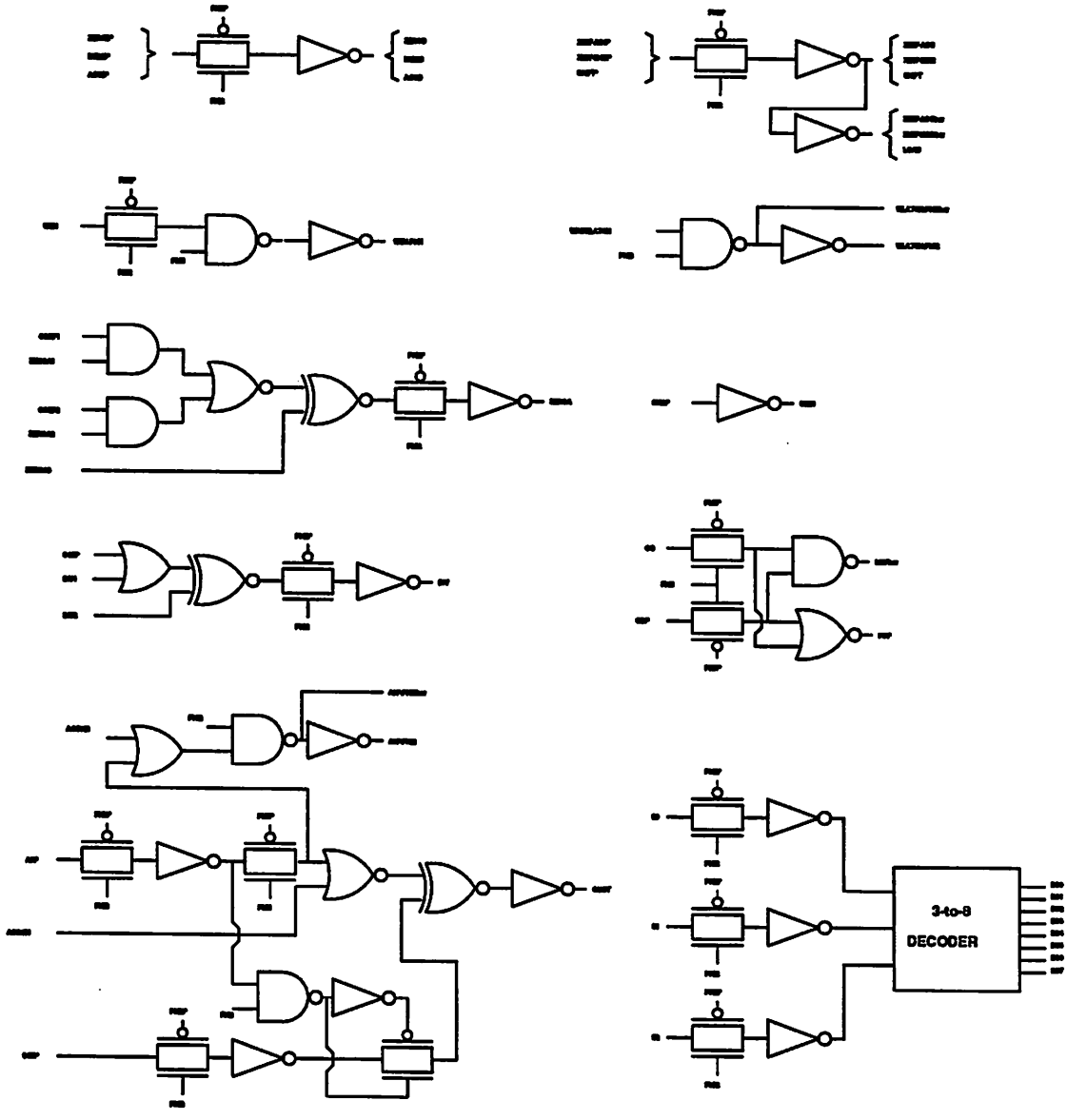


Figure 5.2: Gate Level Description of Interface Logic for AUIOINC Data Path

The input file is used to generate a *sdl* file using the *eqn2sdl* tool. The layout corresponding to the *sdl* file is generated using *Wolfe* and is shown in Figure 5.3.

5.6 Software organization of eqn2sdl

This tool is also written in *Franz Lisp* and utilizes the *Flavors* package. The code is partitioned into several functions spread over several files. The main philosophy of the tool is based on enhancing the Lisp by defining functions which the user can utilize to express the logic as a set of Lisp expressions. In order to avoid confusion with the standard lisp functions for doing logic, a convention was adopted to have an exclamation mark (!) at the end of the newly defined functions. Some of these functions are primitive functions in the sense that they directly map to a single standard cell. The other predefined functions are like macros defined in terms of the primitive functions. For example, the function or! which forms the logical-OR of any number of arguments is in fact defined in terms of primitive functions or2!, or3! and or4! which correspond to ORing of two, three and four arguments respectively. These primitive functions or2!, or3! and or4! are mapped to actual standard cells.

Although a fairly extensive set of functions has been provided which takes full advantage of all the standard cells existing in the MSU standard cell library, the tool has been designed to be extensible such that the user can define his own new primitive functions (which map to some standard cell he has designed) or new utility functions which act as macros. These can be defined either as part of the input file (which is just a Lisp program) or in a default file called *.eqn2sdl* in the current directory or the home directory. In the Lager example given before, the user has defined functions del1! and del2! which are used as macros. A special Lisp function called *newfun* has been provided which lets the user define a primitive function, the cell it maps to and the terminal information. For example, the commands

```
(newfun sum! 1850 (nonpermutable 1A 1B 2CIN) (SUM CO) 1)
(newfun carry! 1850 (nonpermutable 1A 1B 2CIN) (SUM CO) 2)
```

declare two functions, sum! and carry!, both of which take three arguments and both map to the same standard cell number 1850. The difference lies in that the function sum! utilizes the SUM output of the cell whereas the function carry! utilizes the CO

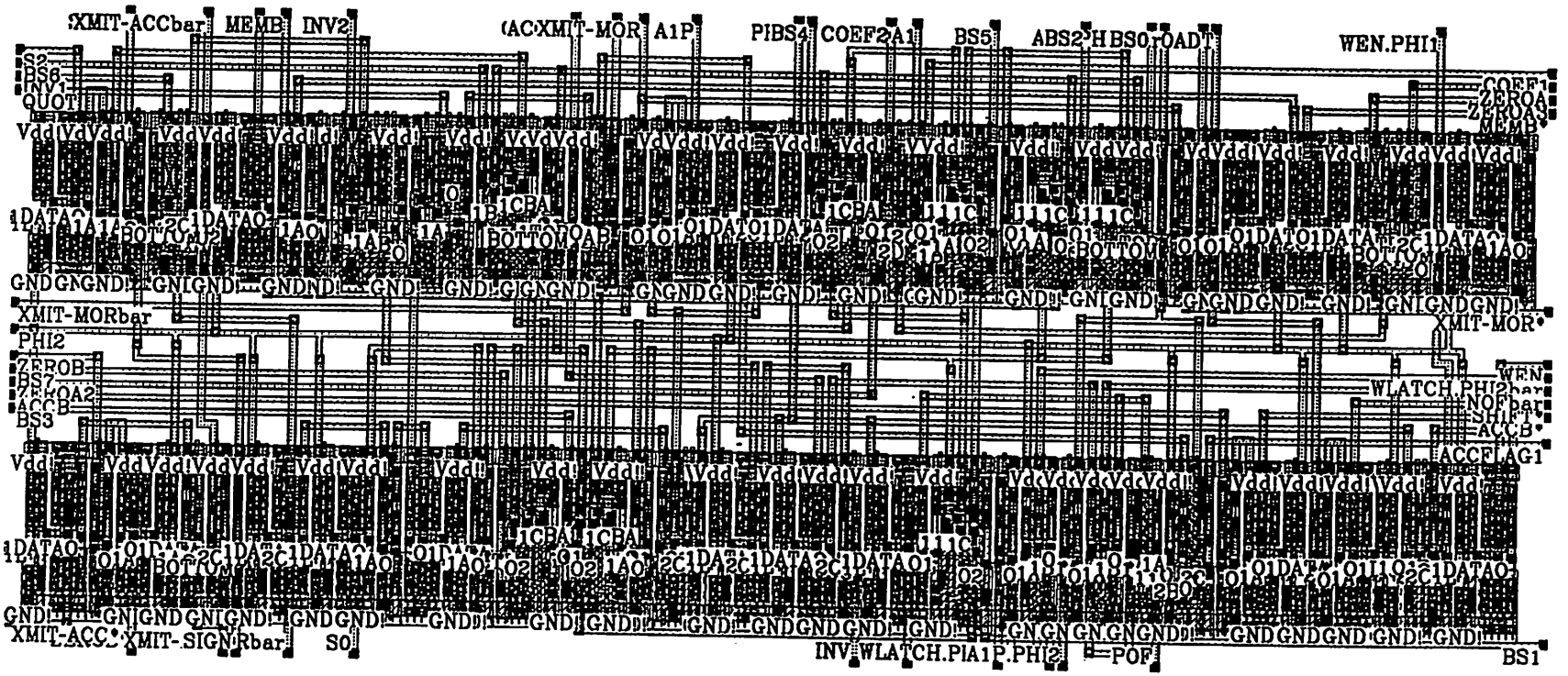


Figure 5.3: Interface Logic for AUIOINC Data Path

output of the cell. Further, the declaration says that for both of these functions the inputs are nonpermutable. Note that this is only partly true since the first two inputs are mutually permutable but this feature has not been implemented yet. At present, the tool can take advantage of permutable inputs only if all the inputs are permutable. The function *newfun* is in fact a Lisp macro which translates the given information into a large Lisp function for each such declaration. The Lisp function so created is then executed when the user uses it in the input file.

The tool works by first loading the internal (predefined) functions and then executing the user input file as a Lisp program. As the user input program is executed, a network of interconnected objects is created. There are two types of objects: *net* and *module*. Each module has several net objects attached as its inputs and one net as its output. While executing the user input, the tool automatically tries to do common subexpression elimination so that there are no redundant modules. During this elimination, the tool tries to take advantage of the permutability of the inputs of a function in order to minimize the number of modules. The sequence in which the user has given the expressions matters in logic having feedback. The basic philosophy adopted is that if any argument of a function has not yet been assigned a value then it is tagged as unassigned and as soon as a later step assigns a value to it, that value is used. This mechanism enables the user to give recursive definitions also, resulting in circuits having feedback. A special function *merge!* has been provided to merge two variables so that the outputs of two expressions can be forced to be identical to each other.

After the user program has been executed, a network of the two types of objects has been setup. Now, post processing is done on this network to eliminate redundant objects and then output the sdl file.

One advantage of the above scheme is that by appropriately redefining the functions, the user input file can be run in a loop as a Lisp program to provide logic simulation. This extension would require local memory to be associated with each object to take care of the latency aspects. However, this type of simulation is not very efficient as compared to the standard event driven simulators. A disadvantage of treating the user input as a Lisp program is that the tool loses the capability of doing extensive error diagnostics since the control is lost to the Lisp interpreter.

Some interesting extensions to this tool are also possible, including the extension to handle complete data path descriptions as a set of Lisp expressions so as to provide a

somewhat higher level interface for the data path description than is provided by the sdl syntax.

Chapter 6

Test Data Paths

This chapter contains some examples of data paths designed using the system described in the previous chapters. After that, some benchmark results about the execution time is given.

6.1 Example 1: A Simple Data Path to do Addition

This data path is a simple example chosen to illustrate the various steps involved in designing a data path using this tool. The goal was to design a data path which takes two inputs and outputs their sum. The data path is to be pipelined with latches present at the input and the output. The first step in the design is to construct a block diagram of the data path using the standard blocks available in the system library. In this case, we decide to use the blocks called *adder* and *latch_ph1*. The *adder* block is a ripple carry adder. It has two carry chains for enhanced speed and is implemented using three types of leafcells. However, the user need not worry about these details. The mechanism to generate an n-bit version of this adder is encoded in the file *adder.sdl* in the system library. This file also has information about the terminal names. Similarly, to implement the input and output latches of the data path it was decided to use the block called *latch_ph1* from the system library which is a dynamic, negative logic transparent latch, *i.e.* the output follows the negation of the input when the clock is high and retains the old value when the clock is low. Since it is dynamic, the output does not retain its value for long once the clock is low. The information about this block is encoded in the file *latch_ph1.sdl*. The sdl files for the two types of blocks used in this data path are shown in figure 6.1.

sdl FILE FOR ADDER BLOCK:

```
(parent-cell adder)
(lisp-function

;the terminal mapping
(deftermmap adder
(if (and (equal terminal "CIN") (= 0 i))
then (useterm "CIN")
elseif (and (equal terminal "CININV") (= 0 i))
then (useterm "CININV")
elseif (and (equal terminal "COUT") (= i msb))
then (useterm "COUT")
elseif (and (equal terminal "COUTINV") (= i msb))
then (useterm "COUTINV")
elseif (and (equal terminal "COUTN-1INV") (= i msb))
then (useterm "CININV")
elseif (and (equal terminal "IN1") (oddp i))
then (useterm "IN1")
elseif (and (equal terminal "IN1") (evenp i))
then (useterm "IN1")
elseif (and (equal terminal "IN2") (oddp i))
then (useterm "IN2")
elseif (and (equal terminal "IN2") (evenp i))
then (useterm "IN2")
elseif (equal terminal "OUT") then (useterm "OUT")
))

;the cell mapping
(defcellmap adder
(if (and (oddp i) (= i msb)) then (usecell "adder_odd_tapd")
elseif (evenp i) then (usecell "adder_even")
else  (usecell "adder_odd"))))
```

sdl FILE FOR latch_ph1 BLOCK:

```
(parent-cell latch_ph1)
(lisp-function

;the cell mapping
(defcellmap latch_ph1 (usecell "latch_ph1"))

;the terminal mapping
(deftermmap latch_ph1
(if (equal terminal "IN") then (useterm "IN")
elseif (equal terminal "OUTINV") then (useterm "OUTINV")
elseif (and (equal i msb) (equal terminal "PHIA")) then (useterm "PHIA")
elseif (and (equal i msb) (equal terminal "PHIAINV")) then (useterm "PHIAINV")
)))
```

Figure 6.1: sdl files for the blocks used in data path example 1

Figure 6.2 shows the block diagram of the data path.

The next step is to write the netlist description of the desired data path. This is done using the sdl syntax. Using the block diagram for guidance, the sdl description is written easily and is shown in figure 6.3. The description consists of three main things: the list of blocks or sub-cells used, the netlist and the constraints on terminal location. The data path is parameterized by the number of bits N and this is declared in the first line of file. The various sub-cells are given instance names. Thus, the block *adder* has the instance name *ADDER*. The two input latches are of the same block type but have different instance names, *INPUTREG1* and *INPUTREG2*. The net list consists of a list of nets and the terminals connected to each of them. There are two types of nets: the data nets and the control nets. Only the data nets are routed. The control nets are just extended to the top and bottom of the data path.

Finally, the Design Manager is run with the data path sdl file as input. The Design Manager calls the DPC which generates the data path. An 8-bit version of the data path was generated and had an area of 646λ by 516λ. The layout is shown in figure 6.4.

6.2 Example 2: Lager AUIOINC Data Path

This data path is the generic data path used in the processors designed using Lager Silicon Compiler [18]. The data path is a simple pipelined data path with an adder and a barrel shifter. It has three registers, two of which are pipeline registers. The data path has been used in a lot of applications involving multiply and accumulate type operations. The block diagram of the data path is given in figure 6.5. The corresponding floorplan in terms of the functional blocks existing in the system library is given in figure 6.6.

The sdl file input for the data path is shown in figure 6.7 and the final layout is given in figure 6.8. The logic diagrams for the the cells used in the design of this data path are shown in the figures 6.9, 6.10 and 6.11. The layouts of the cells used in this data path are given in the appendix.

The data path shown is 4 bits wide and has an area of 370λ by 1410λ. This data path has been submitted for fabrication to MOSIS. An earlier 20-bit hand-routed version of this data path using the same leafcells has been fabricated in a 3 micron CMOS technology and tested upto 8MHz approximately as part of a complete Lager processor. The layout area for the 20-bit hand-routed data path was 1260λ by 926λ, which corresponds to an area

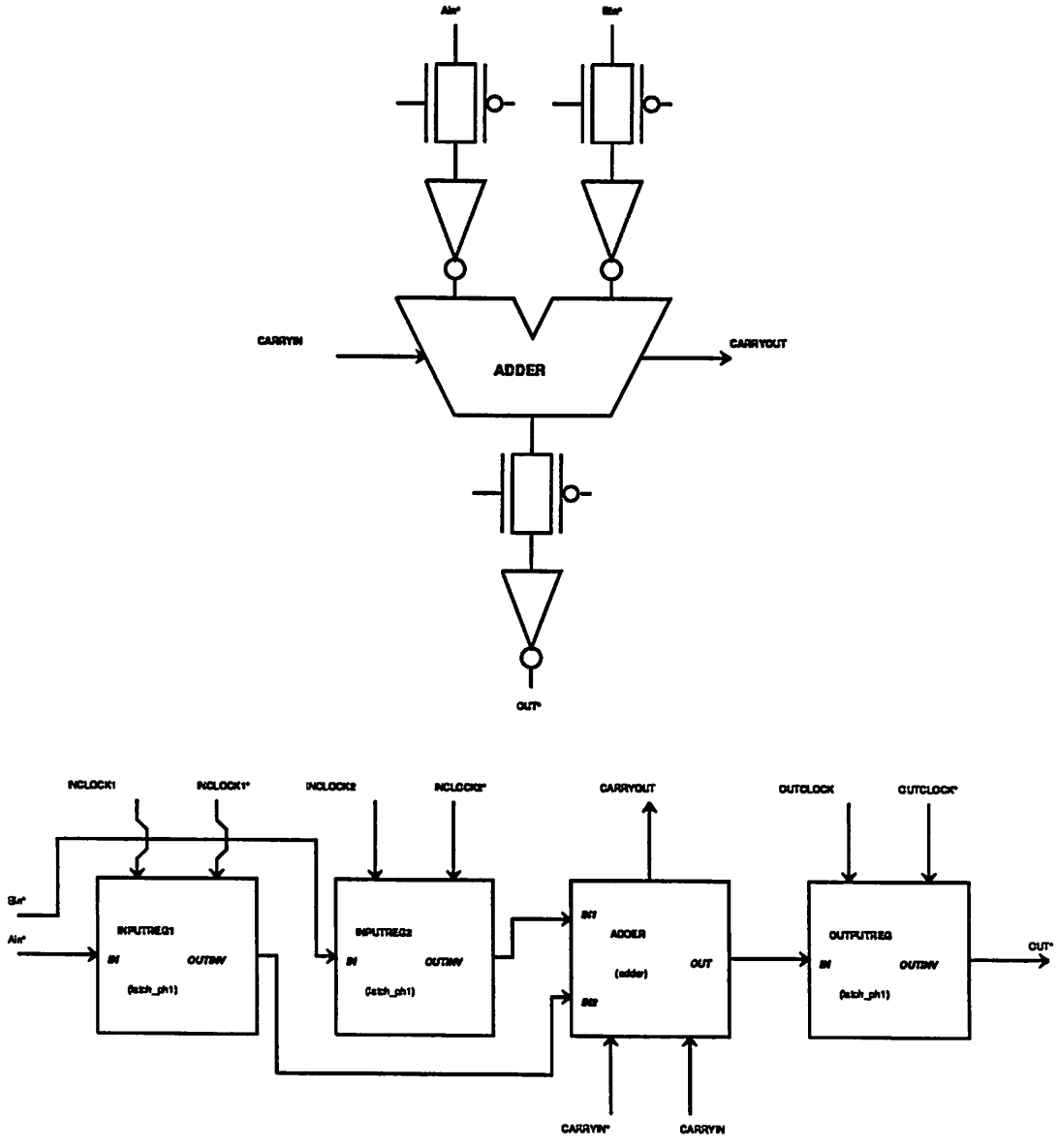


Figure 6.2: Block diagram of the data path in example 1

```
(layout-generator dpc)
(parent-cell exampledp (parameters N))
(sub-cells
    (adder ADDER (parameters (N N)))
        (latch_ph1 INPUTREG1 (parameters (N N)))
        (latch_ph1 INPUTREG2 (parameters (N N)))
        (latch_ph1 OUTPUTREG (parameters (N N)))
)

;data nets
(net DATA0  ((parent Ain*) (INPUTREG1 IN)))
(net DATA1  ((parent Bin*) (INPUTREG2 IN)))
(net DATA2  ((parent OUT*) (OUTPUTREG OUTINV)))
(net DATA3  ((INPUTREG1 OUTINV) (ADDER IN1)))
(net DATA4  ((INPUTREG2 OUTINV) (ADDER IN2)))
(net DATA5  ((ADDER OUT) (OUTPUTREG IN)))

;control nets
(net CNTL0  ((parent CARRYIN) (ADDER CIN)))
(net CNTL1  ((parent CARRYIN*) (ADDER CININV)))
(net CNTL2  ((parent INCLOCK1) (INPUTREG1 PHIA)))
(net CNTL3  ((parent INCLOCK1*) (INPUTREG1 PHIAINV)))
(net CNTL4  ((parent INCLOCK2) (INPUTREG2 PHIA)))
(net CNTL5  ((parent INCLOCK2*) (INPUTREG2 PHIAINV)))
(net CNTL6  ((parent OUTCLOCK) (OUTPUTREG PHIA)))
(net CNTL7  ((parent OUTCLOCK*) (OUTPUTREG PHIAINV)))

;constraints on parent terminal positions
(geometric-constraint-list
(terminal
  (Ain*    (side left))
  (Bin*    (side right))
  (OUT*    (side right))
  )
)
```

Figure 6.3: sdl file for data path in example 1

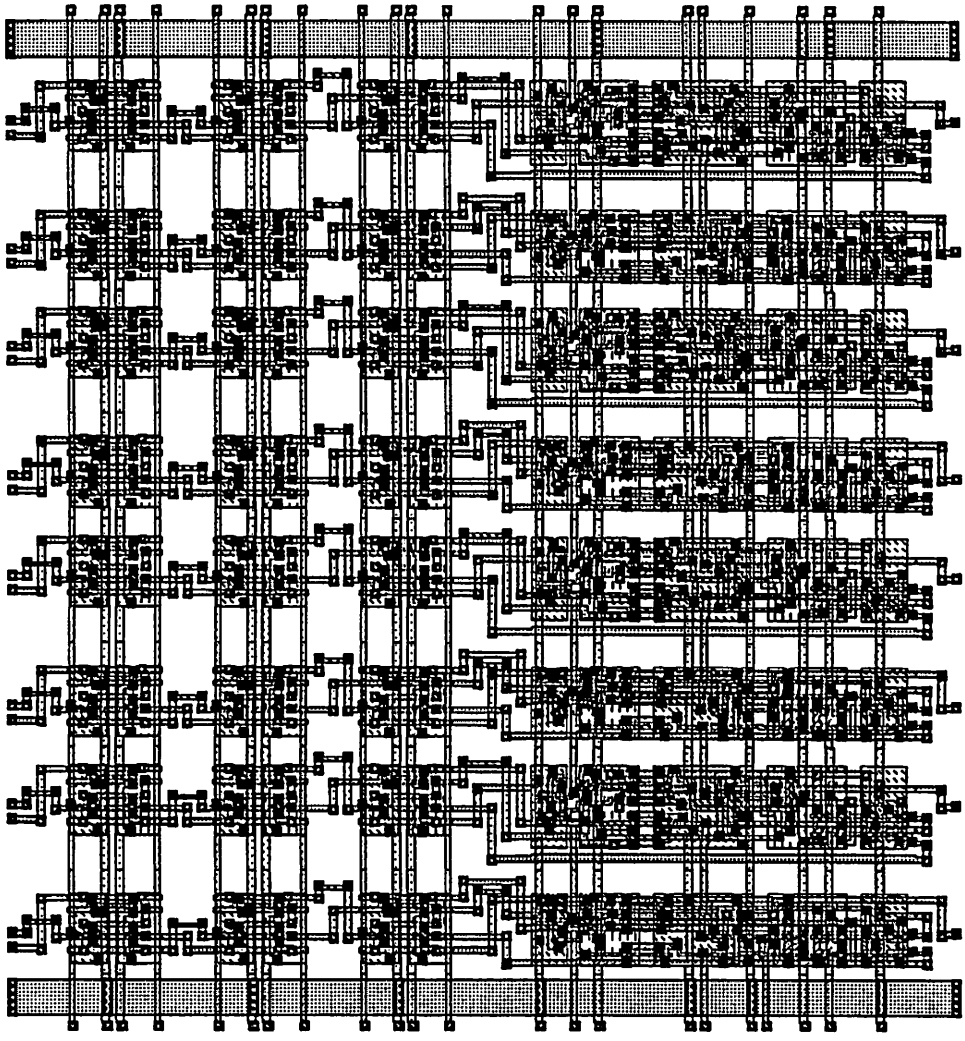


Figure 6.4: Layout of the data path in example 1

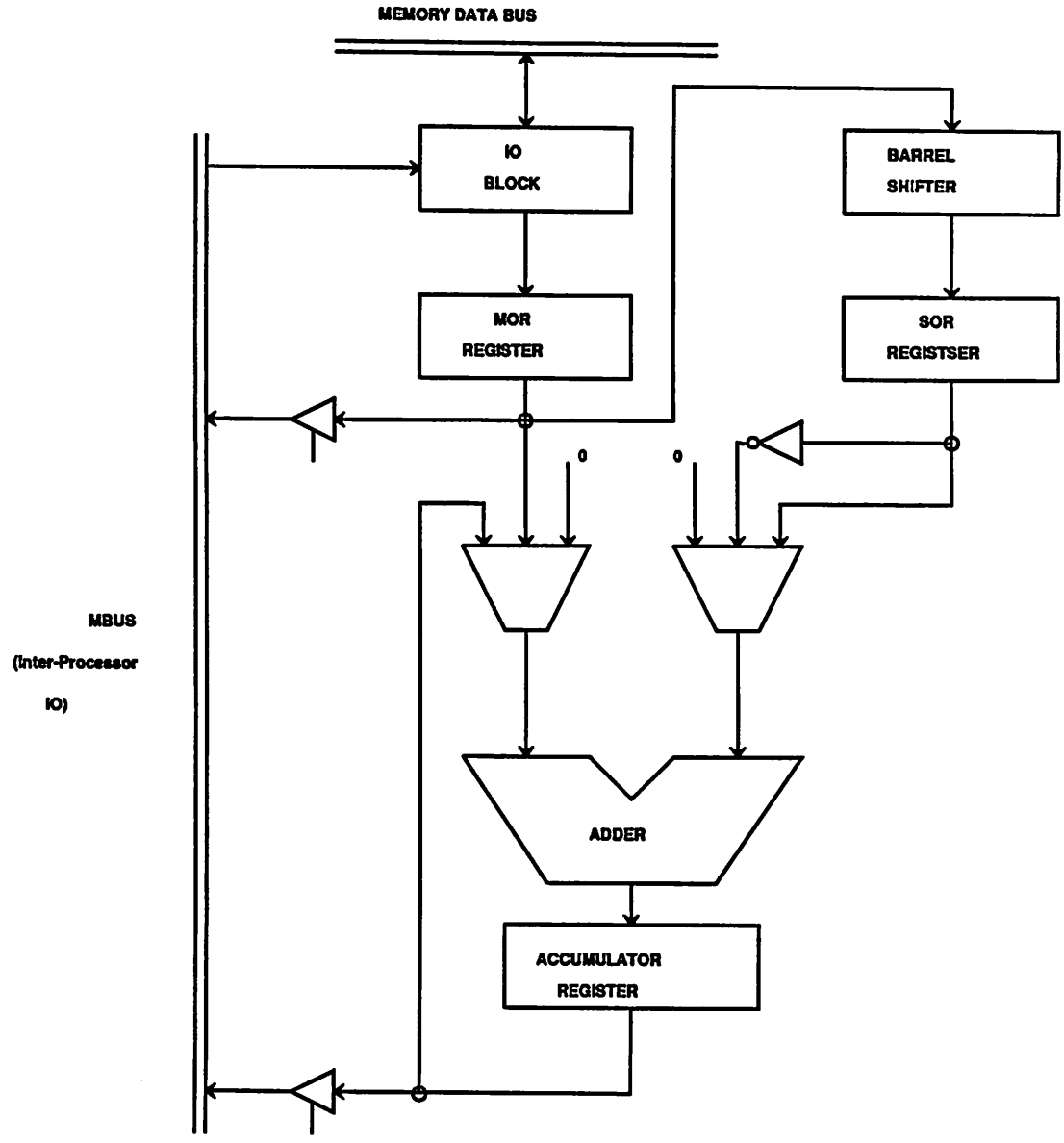


Figure 6.5: Block diagram of Lager data path AUIOINC

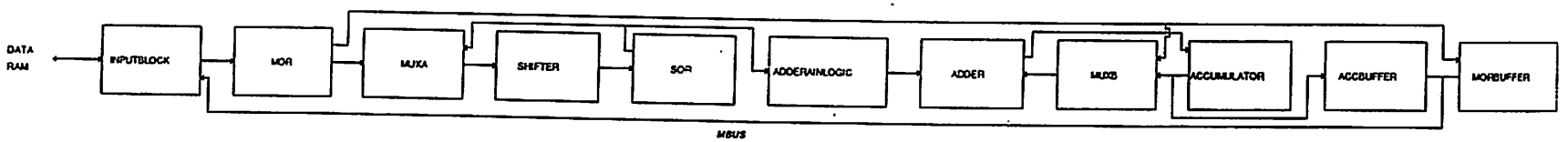


Figure 6.6: Floorplan of Lager data path AUIOINC

```
(layout-generator dpc)
(parent-cell lagerau (parameters N))
(sub-cells
        (LGinputstage INPUTBLOCK)
        (latch MOR (parameters (type dynamic) (logic negative)))
        (mux MUXA (parameters (type 2to1) (logic negative)))
        (barrelshifter SHIFTER (parameters (type R6L1)))
        (latch SOR (parameters (type dynamic) (logic negative)))
        (complementzero ADDERAINLOGIC)
        (adder ADDER)
        (mux MUXB (parameters (type 2to1zero) (logic negative)))
        (accumulator ACCUMULATOR (parameters (type LAGER)))
        (buffer ACCBUFFER (parameters (type tristate) (logic negative)))
        (buffer MORBUFFER (parameters (type tristate) (logic negative)))
)

;data nets
(net DATA00 ((parent RAMLINE) (INPUTBLOCK RAMIO) (MOR IN)))
(net DATA01 ((parent MBUS) (INPUTBLOCK MBUSIN) (ACCBUFFER OUT*)
(MORBUFFER OUT*)))
(net DATA02 ((MOR OUT*) (MUXA Ain) (MUXB Ain) (MORBUFFER IN)))
(net DATA03 ((MUXA OUT*) (SHIFTER IN)))
(net DATA04 ((SHIFTER OUT) (SOR IN)))
(net DATA05 ((SOR OUT*) (MUXA Bin) (ADDERAINLOGIC IN)))
(net DATA06 ((ADDERAINLOGIC OUT) (ADDER Ain)))
(net DATA07 ((MUXB OUT*) (ADDER Bin)))
(net DATA08 ((ADDER SUM) (ACCUMULATOR IN)))
(net DATA09 ((MUXB Bin) (ACCUMULATOR OUT) (ACCBUFFER IN)))

;control nets
(net CNTL00 ((parent WLATCH.PHI2) (INPUTBLOCK NEWMBUS )))
(net CNTL01 ((parent WLATCH.PHI2bar) (INPUTBLOCK NEWMBUS*)))
(net CNTL02 ((parent LOAD ) (MUXA SELA )))
(net CNTL03 ((parent SHIFT) (MUXA SELB )))
(net CNTL04 ((parent BS0) (SHIFTER SHFT-1 )))
(net CNTL05 ((parent BS1) (SHIFTER SHFT0)))
(net CNTL06 ((parent BS2) (SHIFTER SHFT1)))
(net CNTL07 ((parent BS3) (SHIFTER SHFT2)))
(net CNTL08 ((parent BS4) (SHIFTER SHFT3)))
(net CNTL09 ((parent BS5) (SHIFTER SHFT4)))
(net CNTL10 ((parent BS6) (SHIFTER SHFT5)))
(net CNTL11 ((parent BS7) (SHIFTER SHFT6)))
(net CNTL12 ((parent SORbar) (SOR INV_OUT)))
(net CNTL13 ((parent ZEROA) (ADDERAINLOGIC ZERO)))
(net CNTL14 ((parent INV) (ADDERAINLOGIC INV)))
(net CNTL15 ((parent MEMB) (MUXB SELA)))
(net CNTL16 ((parent ZEROB) (MUXB SELZERO)))
(net CNTL17 ((parent ACCB) (MUXB SELB)))
(net CNTL18 ((parent NOFbar) (ACCUMULATOR NOF*)))
(net CNTL19 ((parent POF) (ACCUMULATOR POF)))
(net CNTL20 ((parent A1P.PHI2) (ACCUMULATOR A1P.PHI2)))
(net CNTL21 ((parent A1P.PHI2bar) (ACCUMULATOR A1P.PHI2*)))
(net CNTL22 ((parent XMIT-ACC) (ACCBUFFER CNTL)))
(net CNTL23 ((parent XMIT-ACCbar) (ACCBUFFER CNTLinv)))
(net CNTL24 ((parent XMIT-MOR) (MOR CNTL)))
(net CNTL25 ((parent XMIT-MORbar) (MOR CNTLinv)))
(net CNTL26 ((parent WEN.PHI1) (INPUTBLOCK SELMBUS)))
(net CNTL27 ((parent PHI2INV) (INPUTBLOCK PRCHRG)))
(net CNTL28 ((parent PHI2INV) (SHIFTER PRCHRG)))
(net CNTL29 ((parent PHI1) (ACCUMULATOR CLK1)))
(net CNTL30 ((parent PHI1inv) (ACCUMULATOR CLK1inv)))
(net CNTL31 ((parent CIN) (ADDER CIN)))
(net CNTL32 ((parent CIN*) (ADDER CIN*)))
(net CNTL33 ((parent COn) (ADDER COUTn)))
(net CNTL34 ((parent COn*) (ADDER COUTn*)))
(net CNTL35 ((parent COn-1*) (ADDER COUTn-1*)))
(net CNTL36 ((parent BSDIN) (SHIFTER DIN)))
(net CNTL37 ((parent SIGN*) (ACCUMULATOR SIGN)))
(net CNTL38 ((parent FLAG1) (ACCUMULATOR OUTA)))
(net CNTL39 ((parent FLAG2) (ACCUMULATOR OUTB)))

;constraints on parent terminal positions
(geometric-constraint-list
(terminal
  (RAMLINE (side left))
  (MBUS    (side right))
  )
)
```

Figure 6.7: Input file for AUIOINC data path generated by DPC

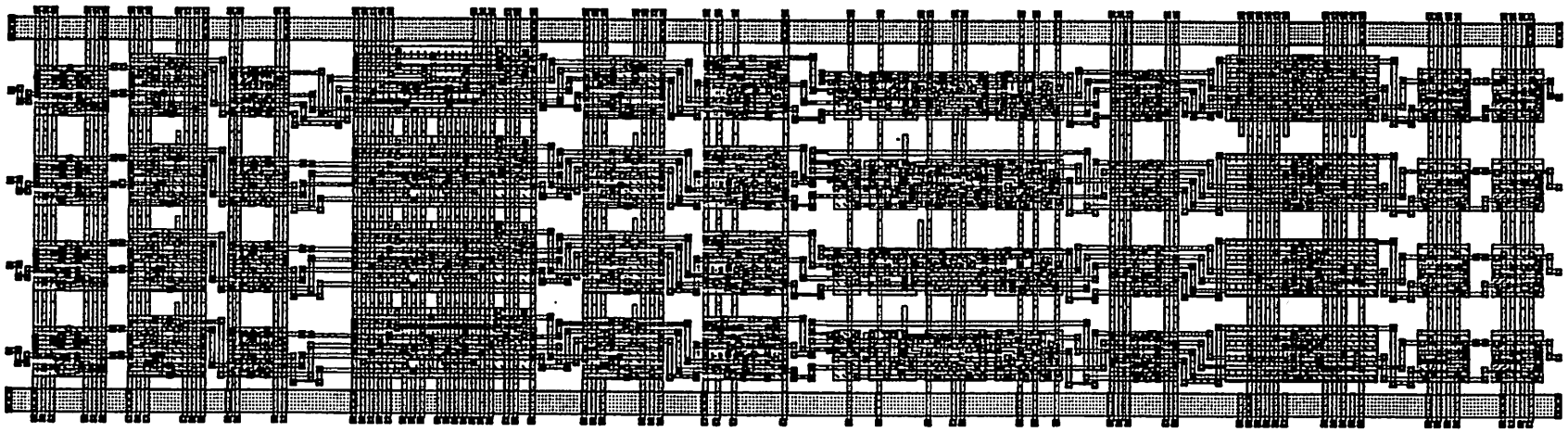


Figure 6.8: Layout of AUIOINC data path generated by DPC

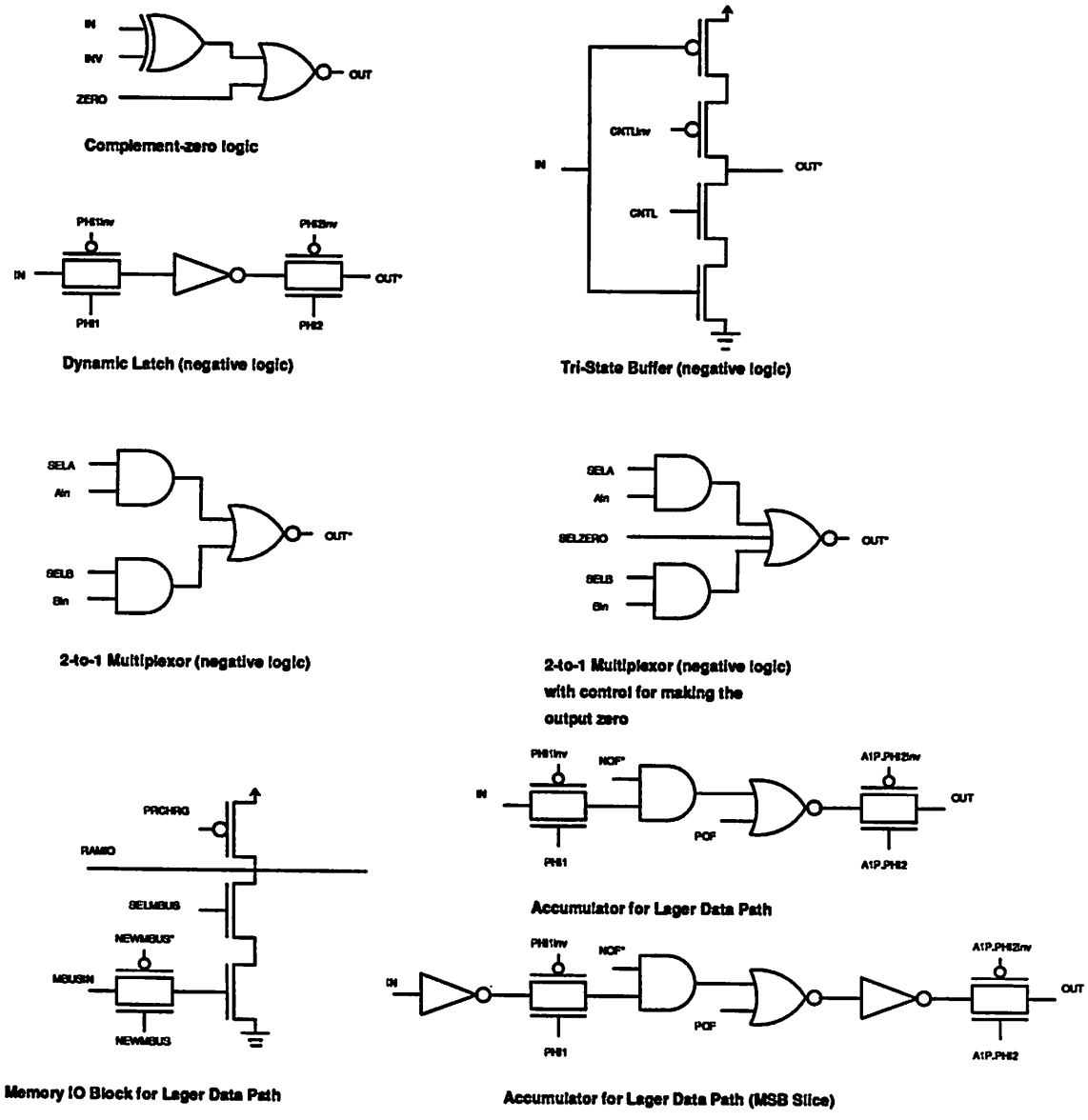


Figure 6.9: Circuits for some of the cells used in Lager AUIOINC data path

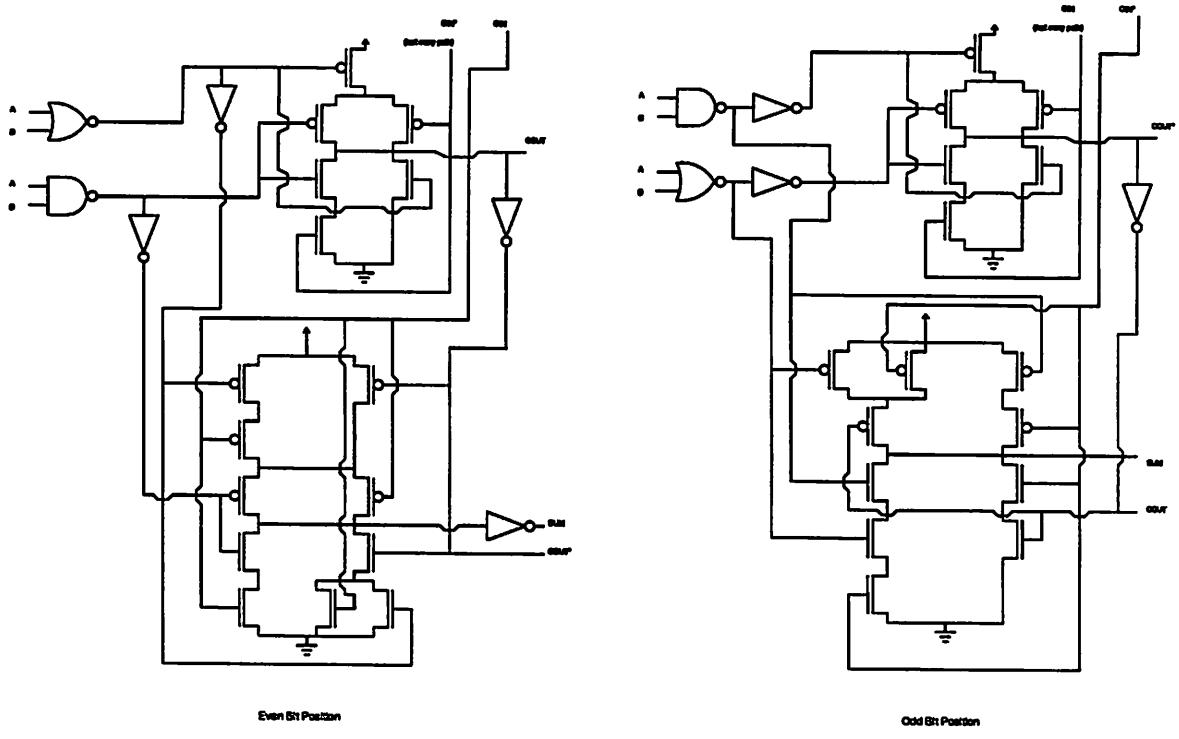


Figure 6.10: Circuit for the ripple carry adder used in Lager AUIOINC data path

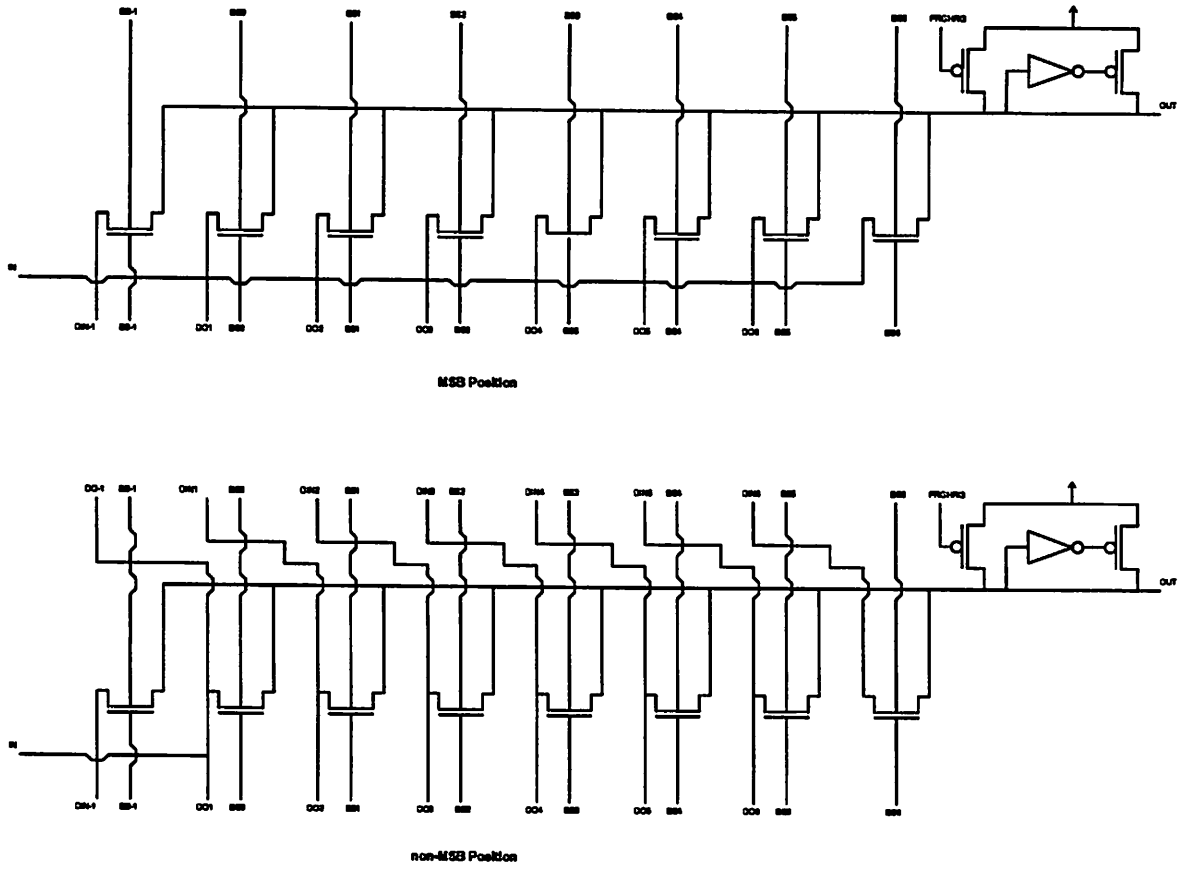


Figure 6.11: Circuit for the barrel shifter used in Lager AUIOINC data path

of 315λ by 926λ for 4-bits. The area of the hand-routed data path is nearly 45the tool. The critical path for the data path is the carry chain of the adder which is of the ripple carry type. The carry delay per stage has been found to be about 2.5ns by SPICE simulation and about the same delay values have been found on some test adders.

6.3 Example 3: Projection Collector Data Path

This data path is part of an image processing chip used for calculating the Radon Transform and the Inverse Radon Transform of an image. This data path is used for calculating the contour along which the projection is taken and is known as the Contour Image Generator data path. It basically calculates which contour does a particular pixel in an image lies on. For more details about this data path, please refer to [17]. The block diagram of this data path is shown in figure 6.12.

The final layout is given in figure 6.13. The data path shown is 10 bits wide and has an area of 758λ by 2520λ. This data path has been submitted for fabrication to MOSIS as part of a complete image processing chip.

6.4 Example 4: Data Path for a Robot Controller

This data path has been designed for use in a processor being designed for robot control application. The data path is a modified version of the AUIOINC data path described earlier. Many blocks used in the AUIOINC data path have also been used here and this demonstrates the flexibility allowed by the DPC in designing the data paths. A block diagram of the data path is shown in figure 6.14. The layout of a 4-bit version of this data path is given in figure 6.15 and the area of the data path generated is 370λ by 1694λ.

6.5 Benchmarking for execution time

One of the major concern while making the choice of LISP as the implementation language was the speed problem. Initially, interpreted LISP was used but resulted in hopelessly slow execution times. Later on majority of the code was compiled and that resulted in substantial improvements. Still, as shown in this section, speed is a problem. Also, since LISP takes a lot of core memory space, the system configuration can have a major effect

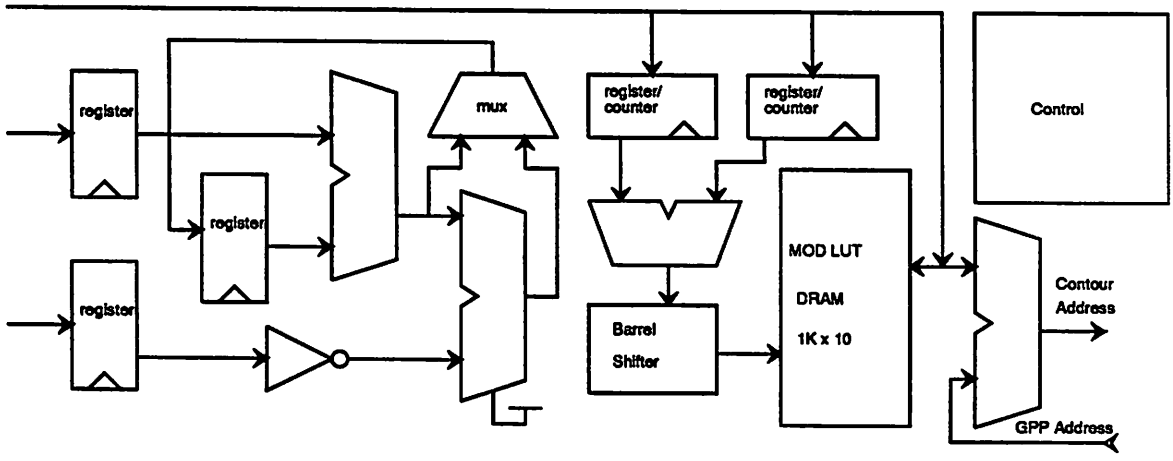


Figure 6.12: Block diagram and the floorplan of the Contour Image Generator Data Path

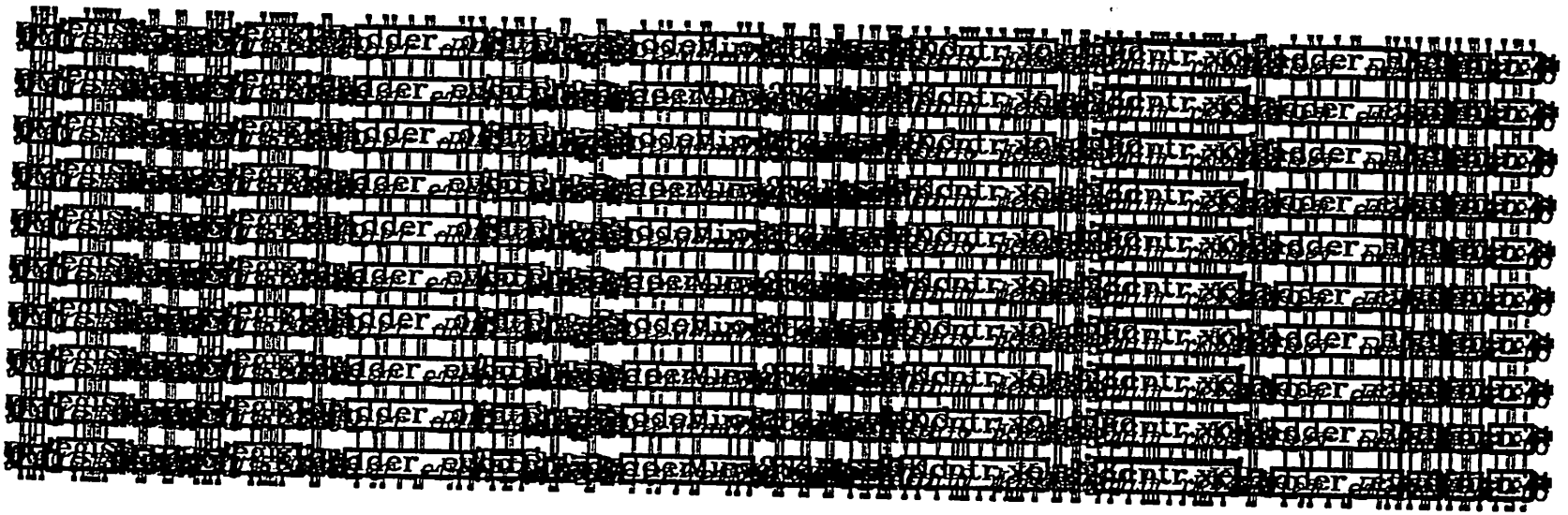


Figure 6.13: Layout of the Contour Image Generator Data Path generated by the DPC

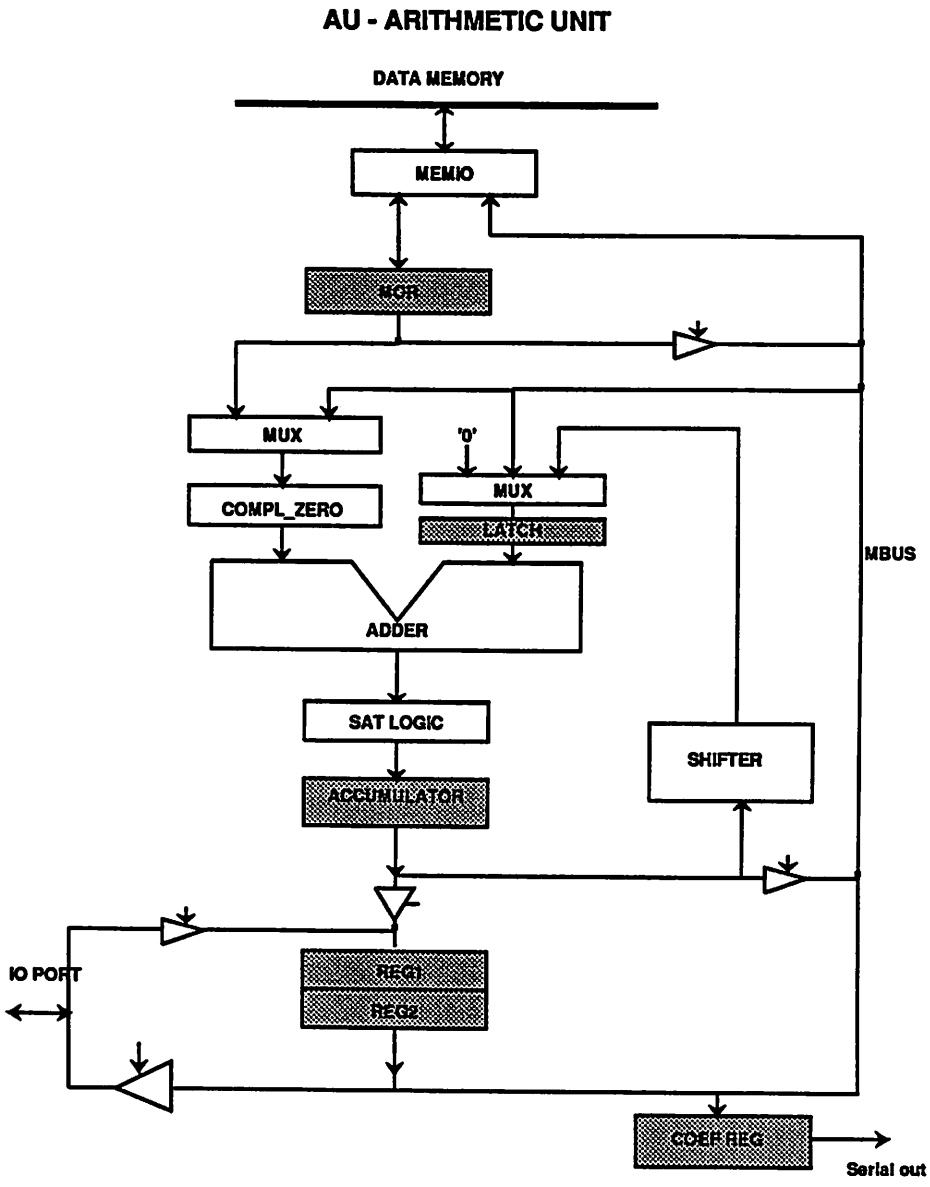


Figure 6.14: Block diagram of the Data Path for the Robot Controller

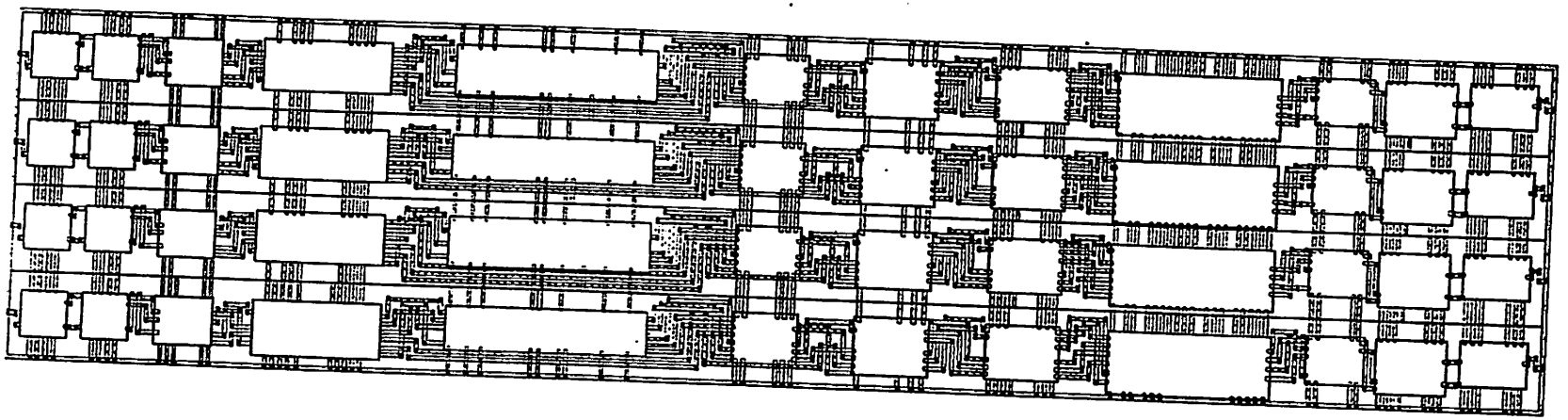


Figure 6.15: Layout of the Data Path for the Robot Controller

on the performance.

The data path chosen for the purpose of benchmarking the execution time was the AUIOINC data path described earlier in this chapter. A 4 bit version was chosen and the time spent by the program in the various phases of the software was measured. The time spent in the actual computation versus the time spent in doing the garbage collection (which is a function of the virtual address space size) was also measure. Finally, the benchmarking was done on two different machines, a SUN 3/160 workstation and a VAX 8800. The results obtained are as follow:

```
Test Data Path: AUIOinc, 4 bits

(I) SUN 3/160, 8 Megabyte Memory

    a. Total processor time spent in
       the frontend:                      32.9 seconds
    b. Total garbage collector time
       spent in the front end:            15.0 seconds
    c. Total processor time spent in
       placement and routing:            181.8 seconds
   (includes garbage collector time)
    d. Total garbage collector time
       spent in placement and routing:    89.6 seconds

(II) VAX 8800, dual CPU, 32 Megabyte Memory

    a. Total processor time spent in
       the frontend:                      11.7 seconds
    b. Total garbage collector time
       spent in the front end:             5.0 seconds
    c. Total processor time spent in
       placement and routing:             66.3 seconds
   (includes garbage collector time)
    d. Total garbage collector time
```

```
spent in placement and routing:        31.7 seconds
```

As is apparent, nearly half the total time is spent on doing just garbage collection. Further, VAX 8800 is nearly three times as fast as a SUN 3/160 for doing data path generation using this tool.

For a 4-bit data path with 11 functional blocks and 10 data nets, it takes nearly 3.5 minutes of CPU time. This is a bit high, although not unbearable. The culprit of course is LISP which results in very slow programs even when compiled.

Chapter 7

Conclusion

In the previous chapters a system to generate the layout of data paths starting from a structural description was given. This system is now fully functional and is part of the Lager silicon assembly system using the Design Manager as a front end. Several data paths have been designed and have been sent for fabrication. Some examples include the generic Lager data path, the new Lager data path which would be used in a robot controller and two data paths being used in a processor used for calculating the projection of images along an arbitrarily oriented straight line (radon transform).

After talking to some of the active users of this tool some problems have come to light. Foremost of these is an overall unsatisfactory quality of routing particularly in the case when there is a potential of doing river routing instead of channel routing. The inter-slice distance is also large. This problem is related to the well layout problem. The router does not layout the wells in the routing region and since the well layouts inside the cells are assumed to be arbitrary, the router is forced to leave a large spacing between adjacent cells. If this large spacing is not left then there are spurious design rule errors within Magic, spurious in the sense that if Magic were to expand the wells correctly while generating the cif files then we do not need to bother about these spurious errors. However, unfortunately, experience has shown that Magic's well expansion is rather error prone and that it is safer to leave enough distance between the cells. Substantial area may be saved by reducing these inter-cell gaps provided the user does a check for these spurious errors. An alternative solution is to enforce a fixed well layout pattern on the cell designers and then let the router do the well painting. However, this idea was rejected after talking to some of the users.

Besides the area loss resulting because of the wells, there is substantial scope of improvement in the routing itself. The router used is a very simple one and use of a better channel router needs to be looked into. The major drawback with the current router is that it is a greedy router and does not try to go back and improve the routing already done. Using good existing channel routers like YACR is a possibility. However, there are several issues which need to be looked into. The first requirement is that in our application we need a router capable of gridless routing in three layers. The requirement of gridless router arises because cases where a grid cannot be defined occur frequently. The capability of routing in three layers is needed in the current scheme where all the three layers, metal1, metal2 and polysilicon are used. However, by prohibiting the use of polysilicon terminals, the requirement of three level routing can be relaxed. The second issue is the mechanism of interfacing to the router. Using file communication for interfacing to an external router will result in too many system calls since the DPC requires a large number of simple channels to be routed and this will result in a large time overhead. A subroutine call based mechanism is highly desirable to avoid this overhead of system calls. Most of the existing routers (YACR, Mighty) are in C whereas this tool is in Lisp. Fortunately, Lisp provides a mechanism for calling C procedures from Lisp programs so that using these programs as subroutines is a definite possibility. Alternatively a channel router based on a better algorithm can be written in Lisp. Another feature which needs to be provided in the router is the capability of doing river routing so as to be able to take advantage of cases where river routing is the natural way to do the routing. A feature which is midway through implementation is automatic placement of the blocks. A scheme based on one dimensional mincut placement is being looked into.

As part of the new version of the Lager Silicon Assembly environment, an entirely new strategy for the data path generation is being explored. The new tool uses a tiling tool called TimLager and a macro-cell router called Flint to generate the data paths. There are many changes in the new environment. The foremost is the fact that now OCT is being used as the design data base. Also, we have moved away from Lisp as the programming language in order to avoid portability and speed problems associated with Franz Lisp. The new tool essentially uses TimLager to generate each individual n-bit wide blocks and then does global routing to prepare the input for Flint which then does the detailed channel routing. This tool is still in a development and testing phase but has already shown results better than or comparable to the older tool.

7.1 Suggestions for Future Work

There are several directions in which the current system may be extended. One thing which needs to be done is to allow the specification of the combined data path and the interface circuitry in form of expressions like the input description to the eqn2sdl tool. This would allow a structure independent description which is quite close to a behavioral description. Initially one might attempt just a one to one translation of this description to structure but at a later stage this may be extended by using optimization techniques. The advantage of such a description would be that the input would be close to the algorithm description itself. Another area which may be investigated is to try simple optimizations on the data path automatically, for example the number of registers. This might be a compromise from the full data path synthesis problem but will provide the user a means to easily explore the data path design space in order to arrive at a design suitable for the current application. Finally, one can try solving the complete synthesis problem in which the user specifies the algorithm in a high level language and the synthesis tool designs the optimal data path for implementing that algorithm under some area and delay constraints specified by the user. The control unit for the data path would be generated as a by-product of the data path synthesis. In this scenario, the current tool can be used to implement the data path architecture designed by the synthesis tool, which would generate a *sdl* description of the data path.

Bibliography

[1] Franz Lisp. Reference Manual, Opus 42. *Franz Incorporated, September 1985.*

[2] Chuen-Shen Shung and Rajeev Jain. Lager III Programmer's Manual. *U. C. Berkeley, 1987.*

[3] Chuen-Shen Shung and Rajeev Jain. Lager III User's Manual. *U. C. Berkeley, 1987.*

[4] FLINT User Documentation. *U. C. Berkeley.*

[5] Eric Lettang. Pad Router- User Documentation. *U. C. Berkeley, 1987.*

[6] G. Hamachi R. Mayo J. Ousterhout W. Scott and G. Taylor. Magic(1) and Magic(5) *Berkeley CAD Tools User's Manual, 1986.*

[7] J. Rowson and B. Walker. Inside a 2901 Datapath Compiler. *VLSI Systems Design, May 1986.*

[8] D. Bursky. Fast silicon compiler optimizes math blocks. *Electronic Design, April 1986*

[9] P. Ruetz. Data Path Generator. *Lager I Documents*

[10] A. D. Lopez and HF S. Law. A dense gate matrix layout method for MOS VLSI. *IEEE Transactions on Electron Devices, August 1980*

[11] Chi-Min Chu. Evaluation of Array Type Generation for Cell Library Layout. *M.S. Thesis, U.C. Berleley, 1987*

[12] MOSIS, Information Sciences Institute, USC. MOSIS scalable and generic CMOS design rules. *MOSIS report, revision 5, September 1986*

[13] R. Rudell. Wolfe - Oct Interface to the TimberWolf Standard Cell Placement Program. *Berkeley CAD Tools User's Manual, 1987*

[14] Carl Sechen and A. Sangiovanni-Vincentelli. The Timberwolf Placement and Routing Package. *IEEE Jornal of Solid-State Circuits, April 1985*

[15] R. Rudell and R. Segal. bdsyn – BDS subset translator for describing logic networks. *Berkeley CAD Tools User's Manual, 1987*

[16] Peter Moore, David S. Harrison, Rick L. Spickelmier and A. R. Newton. Data Management and Graphics Editing in the Berkeley Design Environment. *ICCAD, 1986*

[17] W. B. Baringer, B. C. Richards, R. W. Brodersen *et. al.* A VLSI Implementation of PPPE for Real-Time Image Processing in Radon Space. *Submitted to Workshop on Computer Architectures for Pattern Analysis and Machine Intelligence, 1987*

[18] Stephen P. Pope. Automatic Generation of Signal Processing Integrated Circuits. *Ph.D. Thesis, U. C. Berkeley, 1985*

Appendix A

Manual Page for DPC

NAME

DPC - bitslice datapath generator version 2

SYNOPSIS

DPC

DESCRIPTION

DPC is a tool to generate magic layouts of bit-sliced datapaths starting from a structural description of the datapath in terms of interconnection of datapath functional blocks. The width of the desired datapath (the number of bits) is fed by the user interactively. Associated with each functional block is a lisp function (written in terms of some pre-defined macro definitions) describing how the particular block is assembled from the leafcells in the cell library. A normal user need not concern himself with this and can use the pre-defined blocks only. However, by writing the appropriate lisp function, the user can define his own blocks, using leafcells already there in the system cell library, or using new leafcells. Asociated with each leafcell is a cell-descriptor file which contains information about the cell bounding-box and the terminal locations. A utility function "mag2cd" has been provided. If the cell-descriptor file is not given, the program can extract the information directly from the magic file of the leafcell.

The tool can be used from inside the Design Manager (see DM(1)) or can be used in a stand alone fashion. The syntax for the various files are identical in the two cases. The only difference is that when used from inside the Design Manager, it is expected that the datapath is part of a design hierarchy and is not the top level mdule. When used from inside the Design Manager, the program passes the relevant parameters to the Design Manager whereas when used as a stand alone tool, it outputs a file giving the various module parameters.

To use the tool to generate a datapath when generating layout using the Design Manager, please refer to the Design Manager manual for the details. In order to use it as a stand alone tool, the command to use is

DPC

The program then asks for the datapath name, the number of bits in the datapath and other parameters. A detailed description of how to use the program has been provided in the sections below.

DESIGN RESTRICTIONS ON THE CELLS

The tool imposes certain restrictions on the cell design style. The restrictions, however, do not seem to hinder the

design style too much. The restrictions and suggested design styles are:

1. Although it is not necessary, try to keep the heights of all your cells nearly equal. This results in better area efficiency.

2. The width of cells used to make up a functional block MUST be the same. For example, the cells at the even and the odd bit positions of an adder must have equal width.

3. Any layer, including the well can be used inside the cells.

4. All the data terminals MUST come out at the sides and should be in one of the following layers: M2, VIA, POLY, POLYCONTACT.

5. The data terminals should be "sufficiently" apart. This is rather vague since the terminal spacing requirement depends on the net connection to a certain extent. A safe thumb rule is to have the terminals at >=4 lambdas apart and the terminals themselves 4 lambdas wide in the SCMOS technology. This will ALWAYS work but leads to wasted area in many cases. A better spacing requirement is that the terminals be spaced such that one should be able to place minimum sized contacts to M1 next to them, such that all the contacts lie on the same vertical column.

6. The control terminals, power/ground and clock lines of the cells should come out at the top and bottom edges in one of the following layers: M1, POLY and POLYCONTACT. Please note that this version of the program does not allow the top and bottom terminals to come out in DIFFUSIOn and DIFFUSIONCONTACT. These terminals are extended to meet the top or bottom edge of the slice. Further, the top terminals of the msb slice and the bottom terminals of the lsb slice are brought up to vias for compatibility with the macrocell router FLINT (see FLINT(1)). This requires that the top and bottom terminals of the cells be sufficiently wide and distant from each other. As a thumb rule, in MOSIS SCMOS technology, these terminals should be >=4 lambda wide and >=4 lambda apart, or 3 lambda wide and 5 lambda apart, or 2 lambda wide and 6 lambda apart, provided no design rules are violated for the particular terminal layer.

7. Since FLINT gives special treatment to the power/ground and clock nets, a labelling scheme is

suggested for these terminals. The labelling scheme is: the power terminals should be named by the regular expression [vV][dD][dD]* (where, [vV] stands for v or V, and * stands for any string; thus vDD23, Vdd!, VDD are all valid names), the ground terminals as [gG][nN][dD]* and the various clock terminals as [pP][hH][iI]1[iI][nN][vV]*, [pP][hH][iI]2[iI][nN][vV]*, [pP][hH][iI]1* and [pP][hH][iI]2*. The program brings these terminals to via at the macrocell edge and labels them as VDD[i], GND[i], PHI1INV[i], PHI2INV[i], PHI1[i] and PHI2[i] where i is an integer >=0.

8. To enable better routing performance, try to provide "feedthroughs" in the cells. These are pairs of terminals, one on the left side and the other on the right side of the cell, which are connected to each other and are NOT connected to anything inside the cell. A good strategy is to provide these in M2 in order to avoid high capacitance associated with POLY. The feedthrough pairs should be labelled FEEDn where n is a positive ineteger.

9. All the terminals should be labelled and there should be no useless label at the edge of the cells .

USING THE PROGRAM AND AN EXAMPLE

This tool has been fully integrated with the Design Manager and the input is in the form of the .sdl files. Please refer to the Design Manager manual (also, see DM(1)) for the details of the syntax of .sdl files. Most of the features of the .sdl syntax are accepted. The program takes as input a .sdl file describing the structure of the datapath, a bunch of .sdl files for each of the blocks used by the datapath and .cd (cell descriptor) and/or .mag (magic) files corresponding to the leafcells used by the blocks.

The datapath is visualized as an interconnection of functional blocks (like adder, register, multiplexer) without worrying about the number of bits. This structural description is given in the .sdl file corresponding to the datapath. These functional blocks are referred to as blocks from now on. Each block has some terminals associated with it. For example, a n-bit adder has two n-bit inputs (say, A[0..n-1] and B[0..n-1]), one n-bit output (say, S[0..n-1]), a carry output (say, COUT) and a carry input (say, CIN). The block itself is made up of leafcells, with a certain leafcell being used at a particular bit position. Further, each terminal of the block corresponds to a certain terminal of a leafcell. For example, a n-bit adder can be made, say, by using the cell msbadder at the msb position, evenadder at the evn bit positions and oddadder at the odd bit positions.

The block terminals are mapped to some leaf cell terminal. For example, the block terminal COUT maps to the carry output of msbadder. Similarly, the block terminal S (a n-bit terminal) corresponds to the sum output of msbadder at the msb bit position, to the sum output of evenadder at even bit positions and to the sum output of oddadder at the odd bit positions.

The .sdl file corresponding to the datapath must have a parameter called N in it. A few other restrictions also apply. For example, the bus-width option of the sdl syntax should not be used. Also, the control nets should have names which start with the string CNTL. The control nets should have only two terminals, one of which should be a terminal of the datapath and the other should be a control terminal of a block. Lastly, the only geometric-constraints which are honoured by the program are the sides of the data terminals of the datapath. These sides should be either left or right or can be a parameter. Please refer to the Design Manager manual to know the details about the sdl syntax.

This description of how a block is made up of the leaf cells and the terminal mapping is encoded as a lisp function in the .sdl file associated with the block, using the lisp-function feature of the sdl syntax. Some lisp macros have been defined to make the syntax user friendly. Every block has a .sdl file associated with it. In most cases the .sdl for a block file is very simple, although potentially one can utilize the full control flow mechanism of lisp to describe fairly complex mappings. This encourages reusage of leafcells in many different blocks.

Besides the .sdl files associated with blocks, the program also needs the cell descriptor files (.cd files), which are files associated with each leafcell and describing its bounding box and terminal locations. These files can be manually written by the user for each leaf cell. Alternatively, the user can just give the .mag files associated with the leafcells and the program will automatically extract the relevant information from it. In fact, the way the program proceeds is as follows: The program first looks for both the .cd file and the .mag file. If both exist and the .cd file is newer, then the .cd file is used, else if the .cd is older, a new .cd is created and used. If only .cd file exists, it is used and a warning message is given. If only the .mag file exists, it is used. Finally, if neiether the .cd file nor the .mag file exists, an error message is given and the program exits. A note of caution: please ensure that the directory /tmp is writable by you.

In order to search for the .mag files, the following procedure is adopted: The program first looks for the file

according to the path given in the .lager file in the working directory under the heading DPC.mag (please see the Design Manager manual for details about the .lager file), then the program looks for the file according to the path given in ~/.lager under the heading DPC.mag. Next, the program looks for the file in the current directory and then in the directory ./MAGFILES. Finally, the program looks for the file according to the path given by the UNIX environment variable MAGPATH.

Similarly, in order to search for the .cd files, the following procedure is adopted: The program first looks for the file according to the path given in the .lager file in the working directory under the heading DPC.cd , then the program looks for the file according to the path given in ~/.lager under the heading DPC.cd. Next, the program looks for the file in the current directory and then in the directory ./CDFILES. Finally, the program looks for the file according to the path given by the UNIX environment variable CDPATH.

The output magic files are placed in the directory ./<datapath name>_layout where <datapath name. is the name of the datapath. For example, in the case of the example above, the output is put in ./lagerau_layout. In case of the stand alone version, a <datapath name>.hdl file is also created giving the bounding box and terminal information about the datapath.

FILES

./.lager
~/.lager
./*.sdl
./MAGFILES/*.mag
./CDFILES/*.cd
./<datapath>_layout/*.mag
./<datapath>.hdl

SEE ALSO

DM(1) Design Manager User's Manual

Following are files for various examples and are located on SUNs. It is highly recommended that you go through them to know the syntax of the various files.

Following are the files for Khalid's datapath:

~mbs/DM_v2/*.sdl

~mbs/DM_v2/CDFILES/*.cd

~mbs/DM_v2/MAGFILES/*.mag

AUTHOR

Mani B. Srivastava

DIAGNOSTICS

Provides some elementary error messages. Does not detect spelling mistakes. Needs lots of improvement in this area.

BUGS

Following are some bugs and limitations in the program:

1. This problem is related to wells. At present the program allows any type of well layout. As a consequence of this, the program leaves a worst case gap (9 lamdas in SCMOS) between cells to ensure that there are no design rule errors. This results in substantial area loss. One potential solution is to impose some discipline on the ways the well are placed.

3. Error diagnostic is not good, partly because the Design Manager itself does not do any error diagnostic.

Appendix B

Manual Page for eqn2sdl

NAME

eqn2sdl - equation to sdl converter

SYNOPSIS

eqn2sdl

DESCRIPTION

eqn2sdl is a tool to translate a description of logic given in terms of a high level lisp like syntax to the sdl syntax used by the Design Manager (see dm(1) and dm(5)). This output sdl file can then be used as an input to the Design Manager to generate the layout. At present the output sdl file is targeted at using the MSU Standard Cell library although the syntax is absolutely general and in future the tool may be enhanced to target at other styles of layout generation, for example the Data Path Compiler (see DPC(1)). Also, at present the only type of optimization done is common sub-expression elimination so that at present there is a fair amount of control over the final structure from the input. However, this may change when more optimization is done in the tool. Since the input syntax is a functional syntax, it is not limited to combinational logic only.

The input syntax is basically Franz Lisp enhanced by a set of functions implementing a lot of common functions used in describing logic. All these added functions are distinguished by an exclamation sign (!) at the end. A list of valid functions is provided later in this manual. These functions are used to express the logic as a set of expressions. The arguments to these functions are either another such expression or variables declared by the user. There are two types of variables, the first type are the formal variables of this block (which correspond to the parent terminals in the sdl output) and local variables which are meant for convenience in writing the input. Further, these variables can either be scalar or one-dimensional vectors. A word of caution however, the various functions impose restrictions on the type of their arguments and return a value of a certain type. It may not be possible to interchange a scalar and a vector variable. The formal variables are declared using the function parent! while the local variables are declared using the function var!. The variables can have names which are valid lisp atoms (i.e., any arbitrary string of printable characters which may need to be enclosed within vertical bars (|) if it is an evaluable lisp expression or number or contains a metacharacter). A vector variable x of dimension n is declared as (parent! (x n)).

A third type of variable is a parameter variable which are declared using the function parameter! and are scalar. These are meant to provide programability and are interactively input at the beginning and then used within the user input.

A simple input would consist of a set of expressions using the functions provided. However, advanced users can build their own functions based on these primitives and then use them. Also, the entire control flow mechanism provided by lisp can be used to express the input thus allowing a very flexible and powerful mechanism for programability depending on input parameters. Note that the logic behaviour itself is in a functional format and does not use the control mechanism of lisp. Thus, for example, the lisp function cond does not in anyway imply a multiplexer in the logic.

The logic description is treated as a program and therefore the order of expressions matters. Also, if any argument to an expression has not yet been assigned a value, then it is tagged and if at a later stage a value is assigned to it then the tagged item gets that value. This general scheme enables recursion and delayed assignments. This sequential description of the logic behaviour is very useful in many scenarios.

The basic function provided for assignment of variables is set!. It can handle both scalar and vector variables. In particular, if a vector is given a scalar value then all elements of the vector get that value. Also, if a vector is given a vector value, then the input value is truncated or sign-extended to match the length of the vector to which it is being assigned.

LIST OF FUNCTIONS

Following is list of functions currently avaulable for the tool (besides, of course the standard lisp functions which are used for programablity):

```
(parameter! par1 par2 par3 ....)
(var! var1 var2 var3 .....)
(parent! var1 var2 var3 .....)
(set! var1 arg2)
(vset! l_var1 l_arg2)
(or! arg1 arg2 agr3 .......)
(nor! arg1 arg2 agr3 .......)
(and! arg1 arg2 agr3 .......)
(nand! arg1 arg2 agr3 .......)
(xor! arg1 arg2 agr3 .......)
(xnor! arg1 arg2 agr3 .......)
(not! arg)
(eql! arg)
(sum! arg1 arg2 arg3)
(carry! arg1 arg2 arg3)
(del! arg1 cnt2)
(mux2to1 arg1 arg2 cnt3)
(zero!)
(one!)
```

```
(merge! arg1 arg2 arg3 .....)
(AorBinv! arg1 arg2)
(tristate! arg1 cnt2)
(dec3to8! arg1 arg2 arg3)

;O=(or! (and! 1A 1B) (and! 2C 2D) (and! 3E 3F))
(and/or3_2 1A 1B 2C 2D 3E 3F)

;O=(nand! (or! 1A 1B) (or! 2C 2D))
(or/nand2_2! 1A 1B 2C 2D)

;O=(nor! (and! 1A 1B) (and! 2C 2D))
(and/nor2_2! 1A 1B 2C 2D)

;O=(nor! 1A (and! 2B 2C))
(and/nor2_1! 1A 2B 2C)

;O=(nand! 1A (or! 2B 2C))
(or/nand2_1! 1A 2B 2C)

;O=(or! (and! 1A 1B) (and! 2C 2D) (and! 3E 3F) (and! 4G 4H))
(and/or4_2! 1A 1B 2C 2D 3E 3F 4G 4H)

;O=(or! (and! 1A 1B 1C) (and! 2D 2E 2F))
(and/or2_3! 1A 1B 1C 2D 2E 2F)

;O=(or! (and! 1A 1B) (and! 2C 2D))
(and/or2_2! 1A 1B 2C 2D)

;some functions involving flip-flop action

;transparent latch with reset
;works when clock low
;Q = [(Qn-1 * C) + (D * C')] * R
;QB = (not! Q)
(latchlevel! 1DATA 2CLK 3RST)
(latchlevel*! 1DATA 2CLK 3RST)

;nand latch
;Q = (Qn-1 * R * S) + S'
;QB = (Qn-1' * S * R) + R'
(nandlatch! 1RST 2SET)
(nandlatch*! 1RST 2SET)

;nor latch
;Q = (S * R') + (Qn-1 * S' * R')
;QB = (S' * R) + (Qn-1' * S' * R)
(norlatch! 1RST 2SET)
(norlatch*! 1RST 2SET)

;D flipflop with S and R
;works on high to low edge
```

```
;Q = [(Qn-1 * C) + (Dn-1 * C')] * R
;QB = (not! Q) + R'
(latchaedge! 1DATA 2CLK 3RST 4SET)
(latchaedge*! 1DATA 2CLK 3RST 4SET)

;D flipflop with asynchronous R
;works on high to low edge
;Q = [(Qn-1 * C) + (Dn-1 * C')] * R
(latchbedge! 1DATA 2CLK 3RST)

;D flipflop with asynchronous R
;works on high to low edge
;Q = [(Qn-1 * C) + (Dn-1 * C')]
;QB = (not! Q)
(latchcedge! 1DATA 2CLK 3RST)
(latchcedge*! 1DATA 2CLK 3RST)

;D flipflop
;works on high to low edge
;Q = [(Qn-1 * C) + (Dn-1 * C')]
;QB = (not! Q)
(latchdedge! 1DATA 2CLK)
(latchdedge*! 1DATA 2CLK)
```

FILES

```
./.eqn2sdl
~/.eqn2sdl
./input.eqn
./output.sdl
```

SEE ALSO

DM(1) Design Manager User's Manual

AUTHOR

Mani B. Srivastava

DIAGNOSTICS

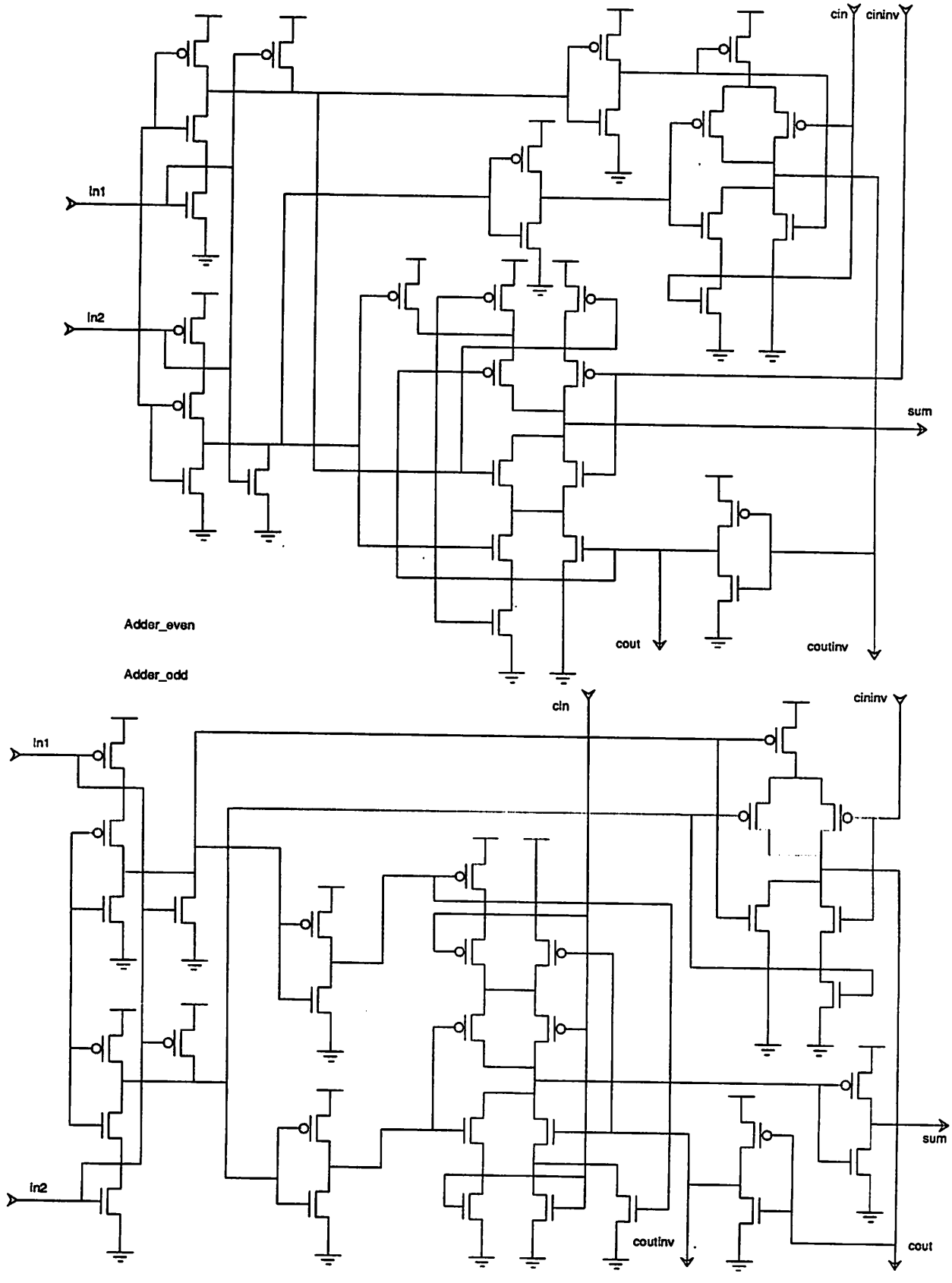
Provides reasonable error messages. Errors relating primarily to the lisp syntax are taken care of by the built-in lisp error handler.

Appendix C

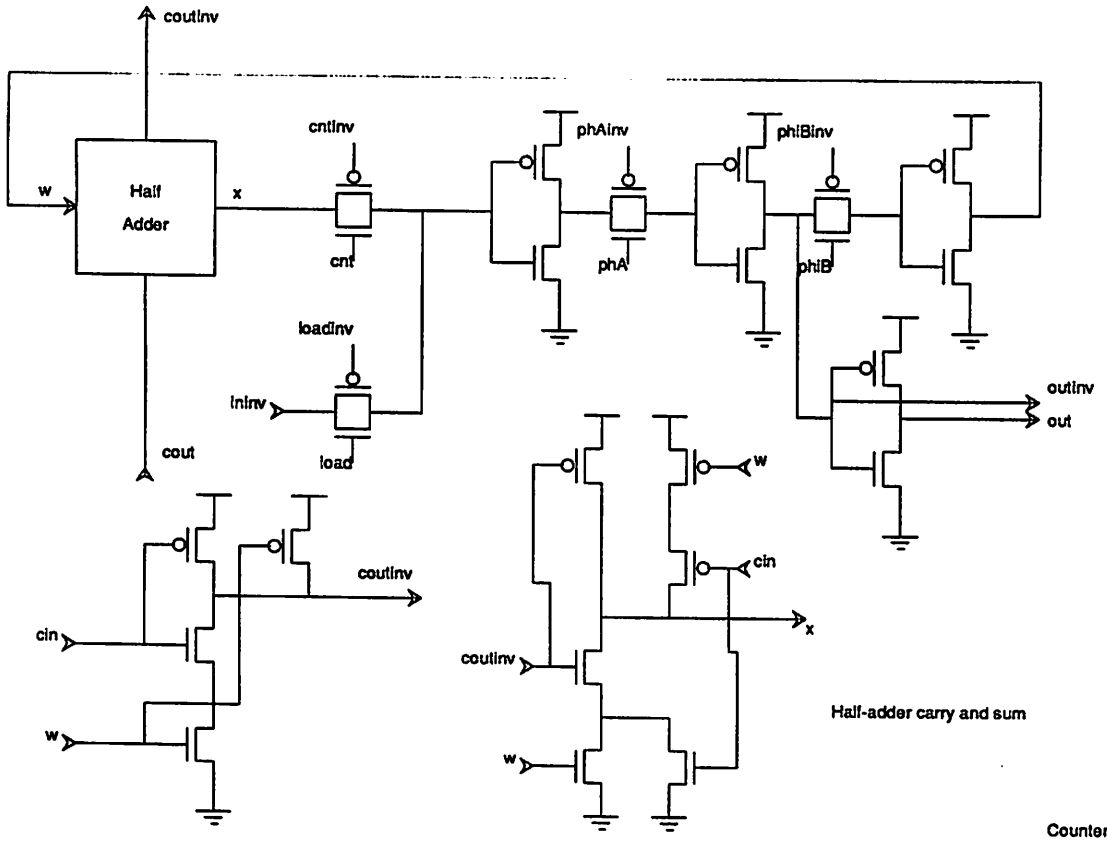
Circuit Diagrams of the Cells in the Library

List of Leafcells in the Library

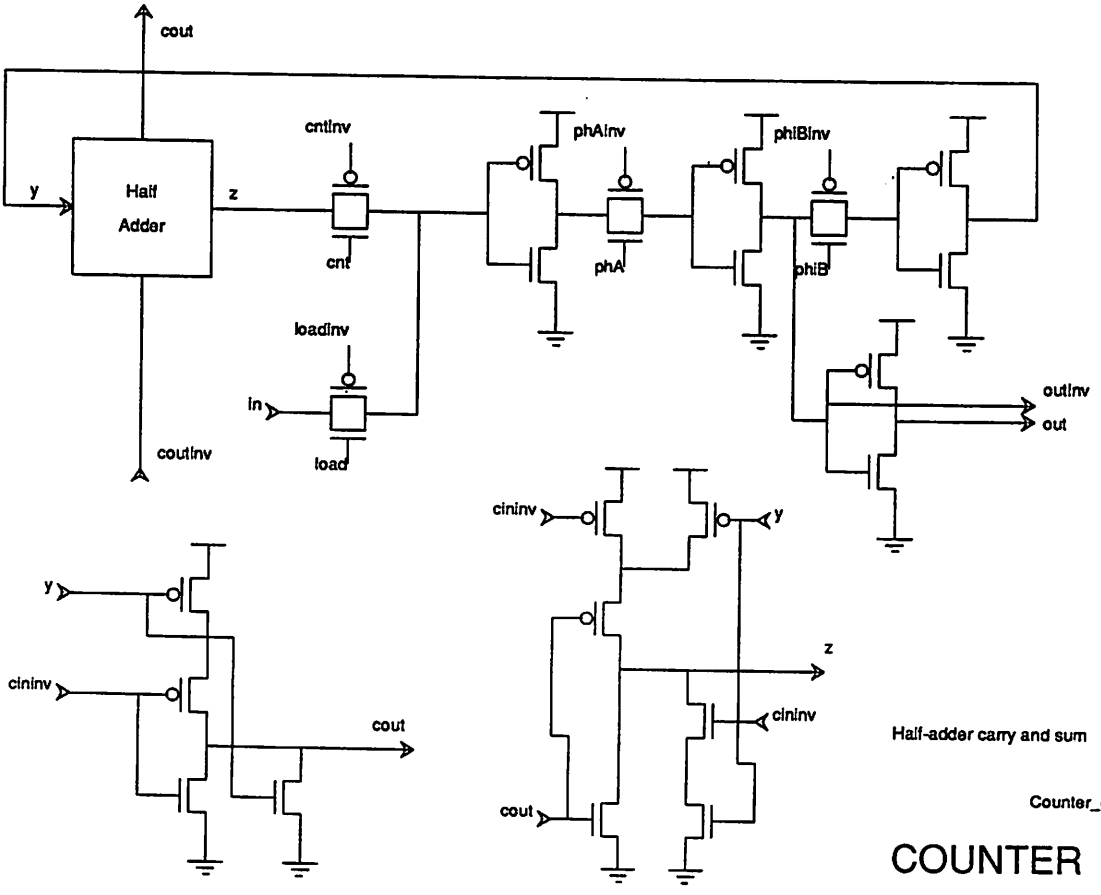
Name of the cell	size	comment
A. ADDER:		
adder_even	46 x 212	Ripple-carry adder even slice.
adder_odd	40 x 212	Ripple-carry adder odd slice.
adder_odd_tapd	40 x 212	Ripple-carry adder odd slice with CININV also out top.
B. COUNTER:		
counter_even	48 x 232	Loadable two-phase counter with count enable; no pass gate XOR in 1/2 adder.
counter_odd	48 x 232	Odd slice of above.
countersetgnd	48 x 11	Optional preset for setting IN on counter to gnd.
countersetvdd	48 x 11	Optional preset for setting IN on counter to vdd.
countersetx	48 x 11	Optional preset for setting IN on counter to pass.
C. LATCHES:		
prechrg_latch	44 x 50	Domino dynamic latch with non-inverting output.
latch_ph1	39 x 52	Single-phase dynamic latch with inverting output.
latch_ph2	55 x 71	Two-phase dynamic latch with inverting output.
scanlatch_ph1	53 x 100	Scanpath single phase dynamic latch with inverting and non inverting outputs. (Scan operation itself is two phase.)
clocked_buffer	38 x 48	Clocked inverter with non-inverting output.
clockedinverter	35 x 45	Clocked inverter with inverting output.
D. MULTIPLEXORS:		
inv2to1mux	42 x 53	Inverting 2-to-1 MUX.
2to1mux	43 x 69	Non-inverting 2-to-1 MUX.
inv2to1muxzero	44 x 61	Inverting 2-to-1 MUX with zero.
E. RANDOM LOGIC:		
inverter	33 x 31	Inverter
inverter4	46 x 38	Four inverters stacked on top of each other.
invertersense	33 x 31	Inverter with threshold shifted for sense-amp operation.
trist_buffer	34 x 48	Tristate buffer
nandnor	41 x 64	Outputs '(in1.in2) and also '(in1+in2)
andnorI	51 x 96	Outputs '(in.cntl1+cntl2)
bufferandnorI	51 x 96	Outputs ('in.cntl1+cntl2)
bufferbig	24 x 50	Buffer
buffersmall	24 x 46	Buffer
dual_buffer	51 x 45	Two buffers in one cell
xfer_gate	40 x 46	Transfer Gate
xornorI	56 x 75	Outputs '(cntl2+(cntl1 xor in))
F. REGISTER:		
register	46 x 105	Single-port static register.
scanreg1Port	50 x 147	Single-port static register with scan path.
scanreg1Portmx	50 x 147	Same as above but mirrored about the X axis.
scanreg2Port	58 x 147	Two-port static register with scan path.
scanreg2Portmx	58 x 147	Same as above but mirrored about the X-axis
G. ZEROS:		
zero	37 x 19	Single strong pull-down.
isozero	27 x 49	Isolated zero: pass-gate followed by pull-down.


cin
cininv
In1
In2
sum
Adder_even
Adder_odd
cout
coutinv
cin
cininv
In1
In2
sum
coutinv
cout

ADDER

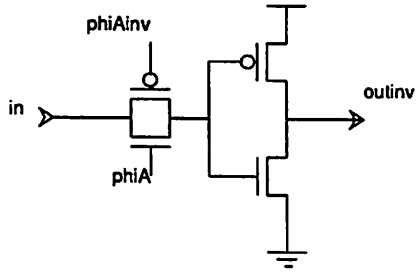


Counter_even

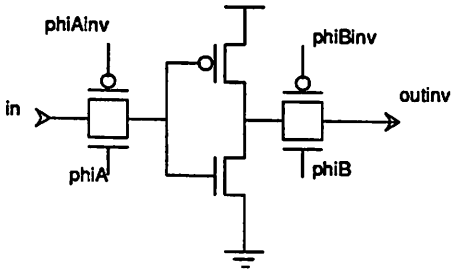


Counter_odd

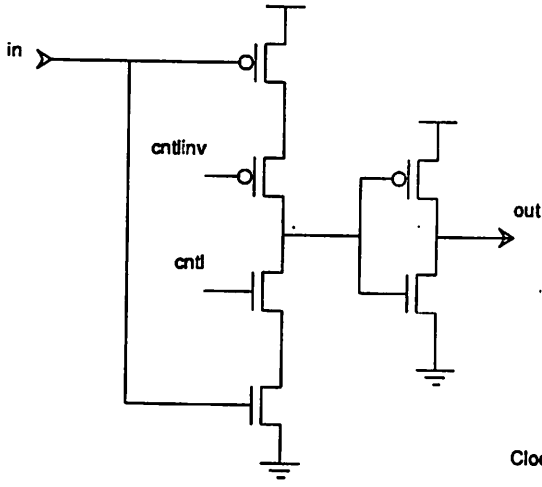
COUNTER



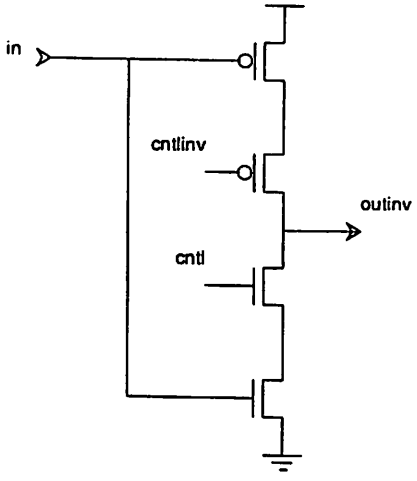
Latch_ph1



Latch_ph2

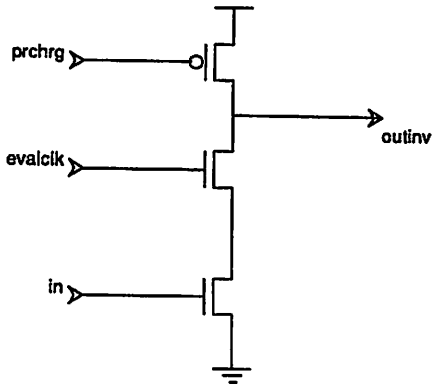


Clocked_buffer

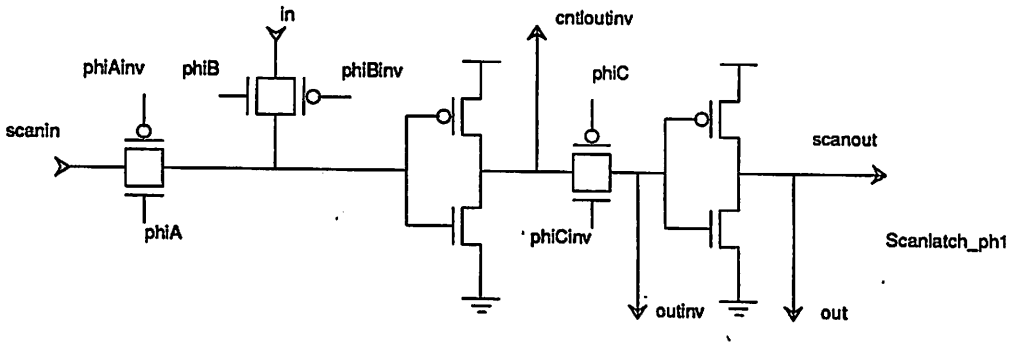


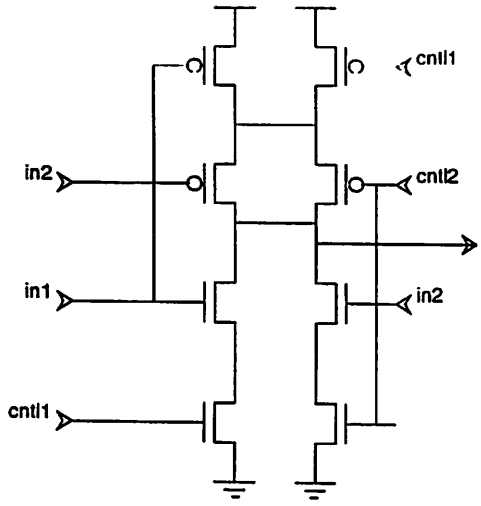
Clockedinverter

LATCHES

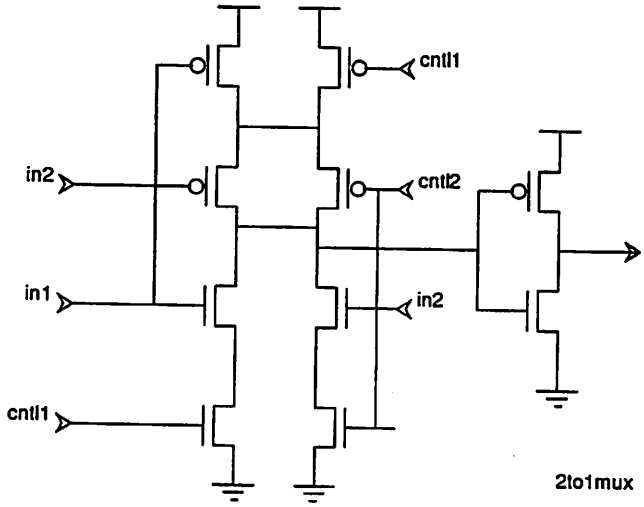


Prechrg_latch

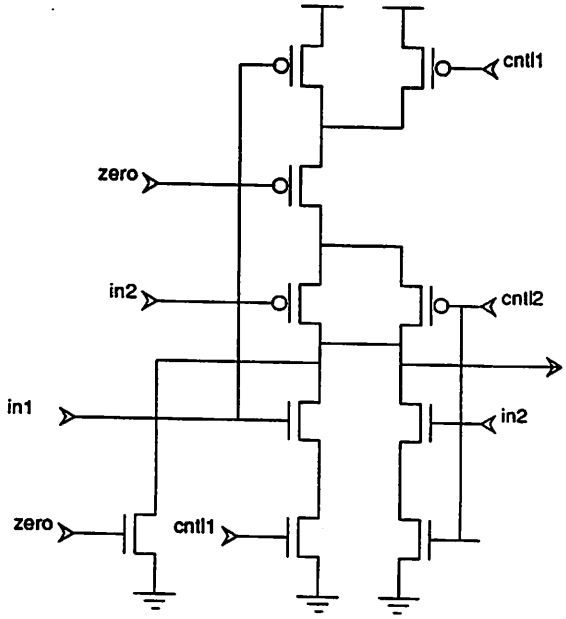




Inv2to1mux

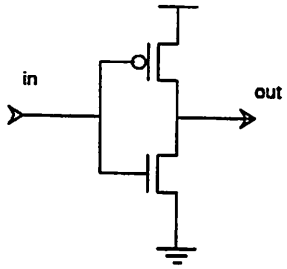


2to1mux

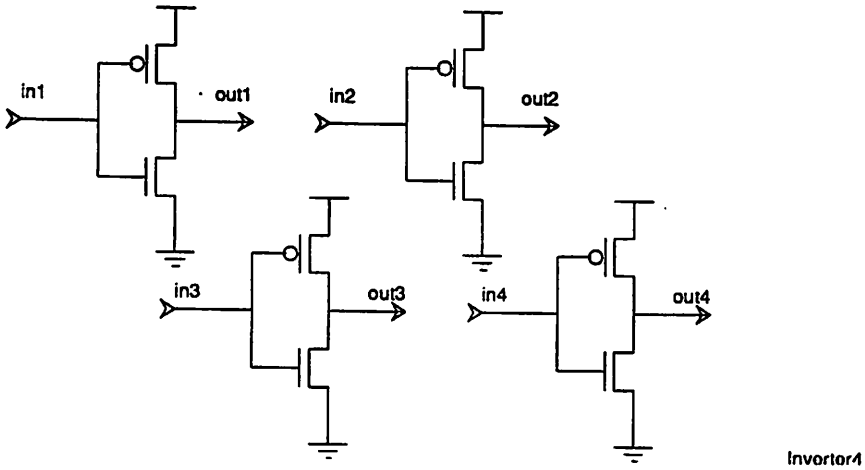


Inv2to1muxzero

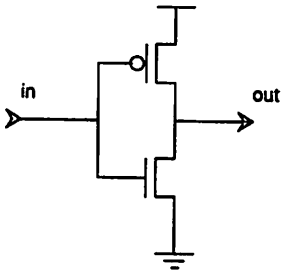
MUXES



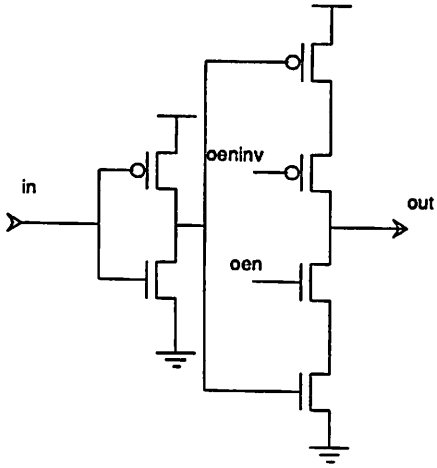
Inverter



Invertor4

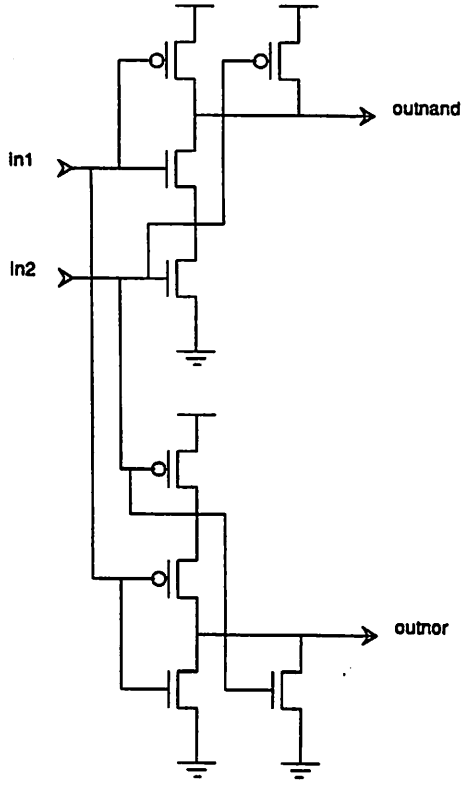


Invertersense

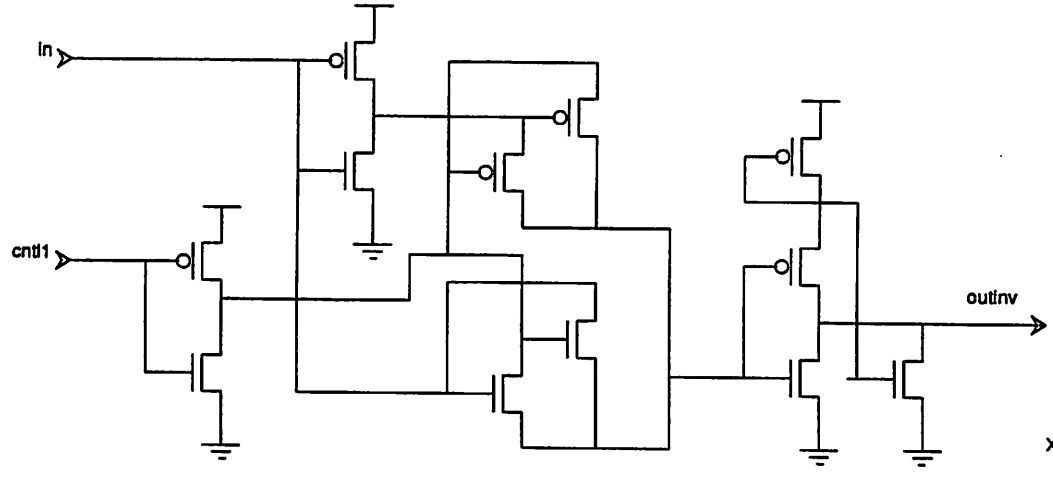


Trist_buffer

RANDOM1

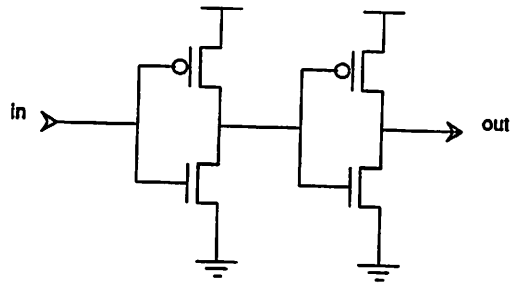


Nandnor

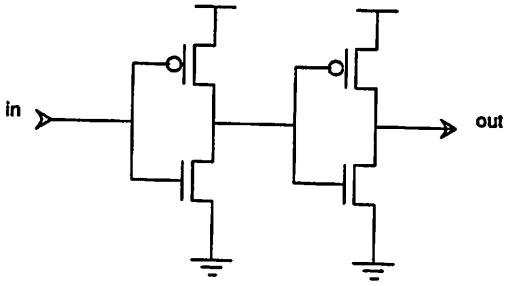


Xornorl

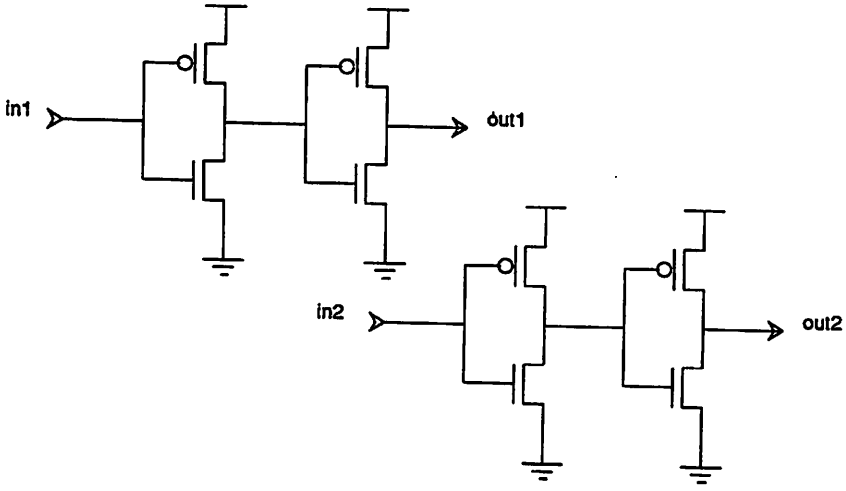
RANDOM2



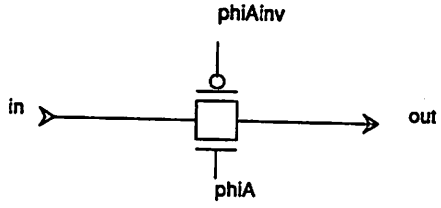
Bufferbig



Buffersmall

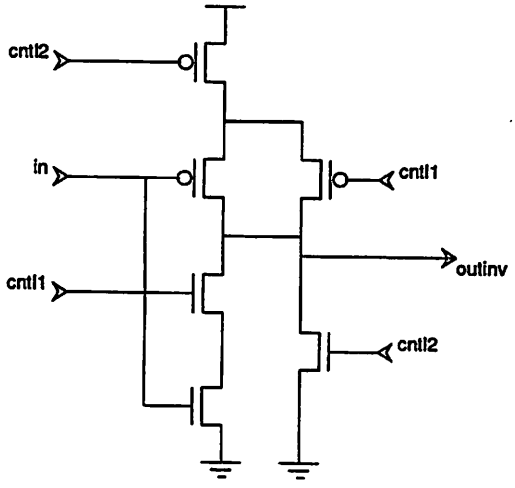


Dual_buffer

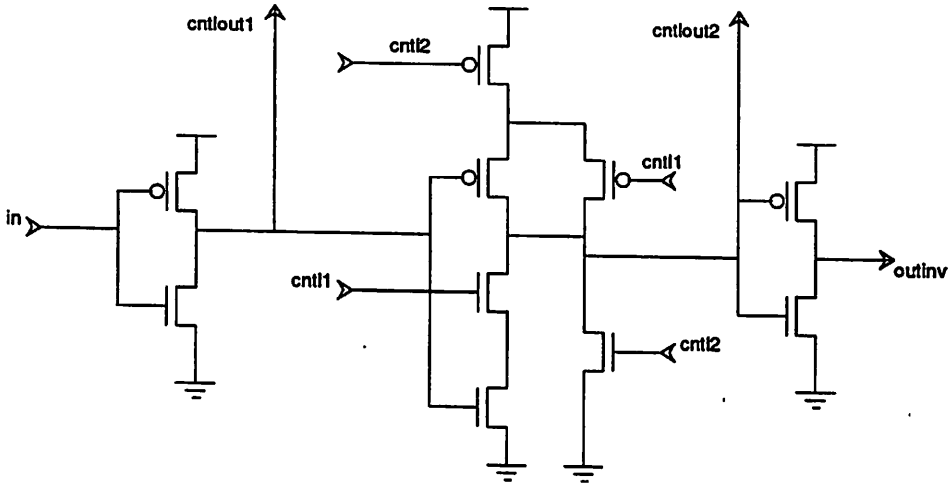


Xfer_gate

RANDOM3

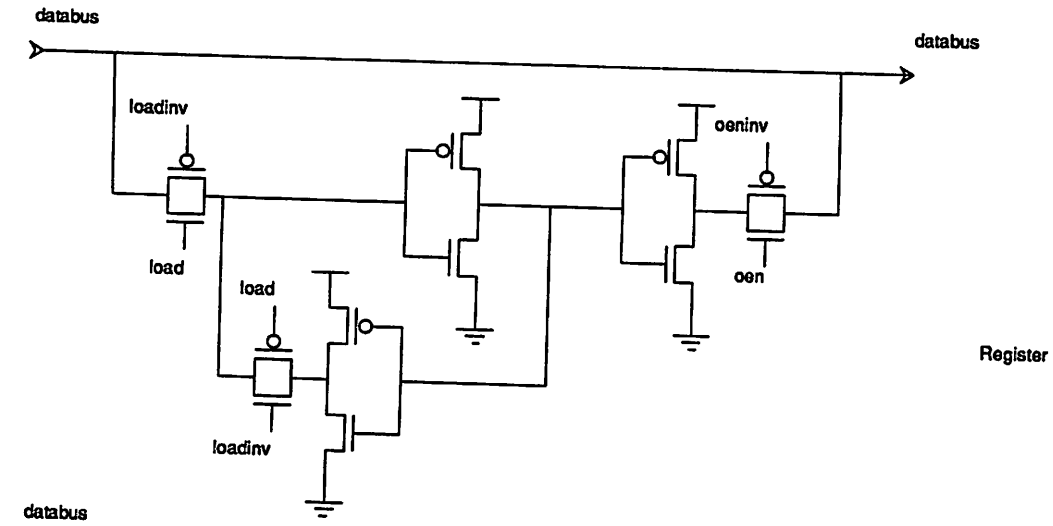


Andnorl

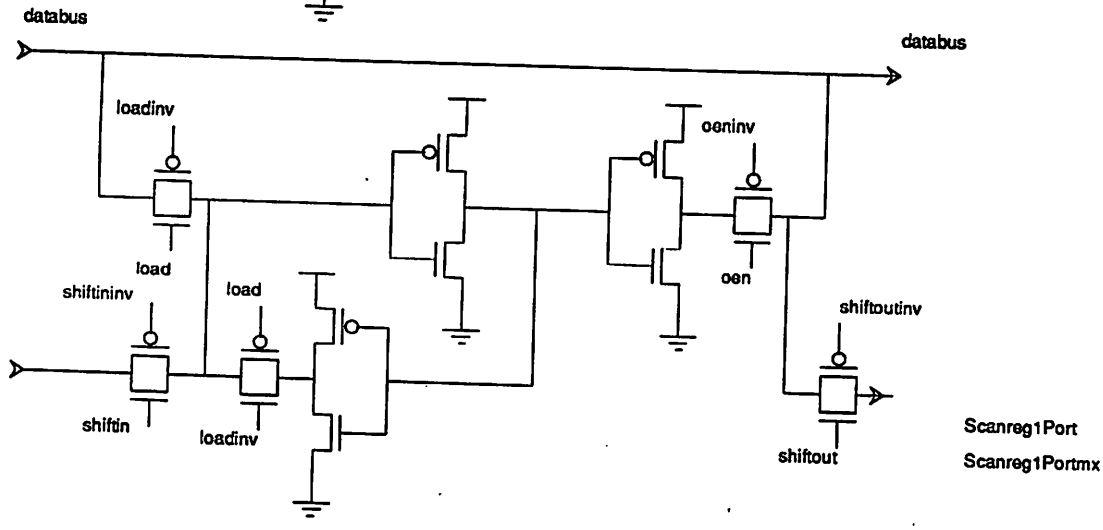


Bufferandnorl

RANDOM4

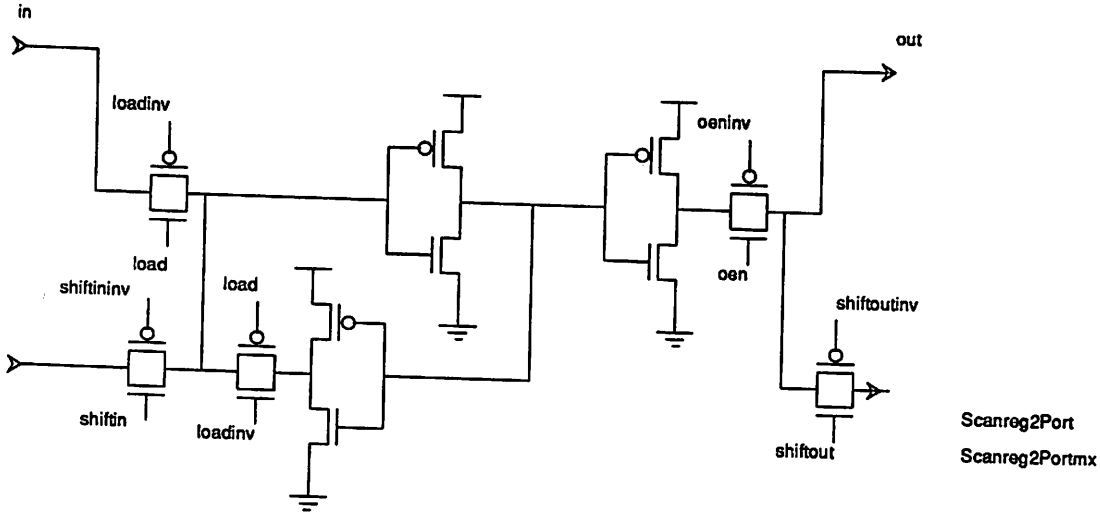


Register



Scanreg1Port

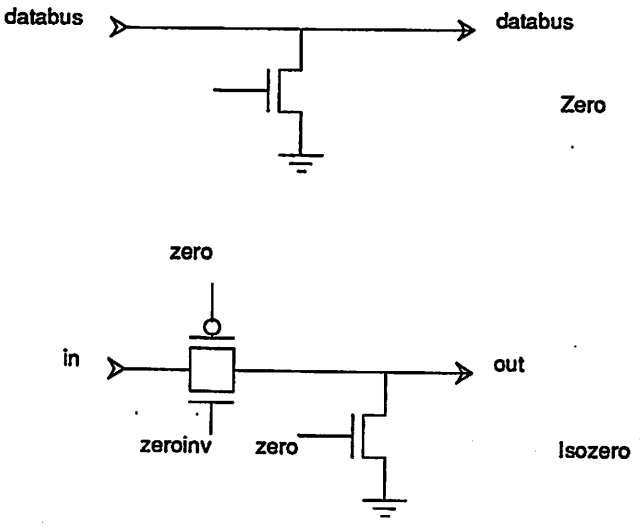
Scanreg1Portmx



Scanreg2Port

Scanreg2Portmx

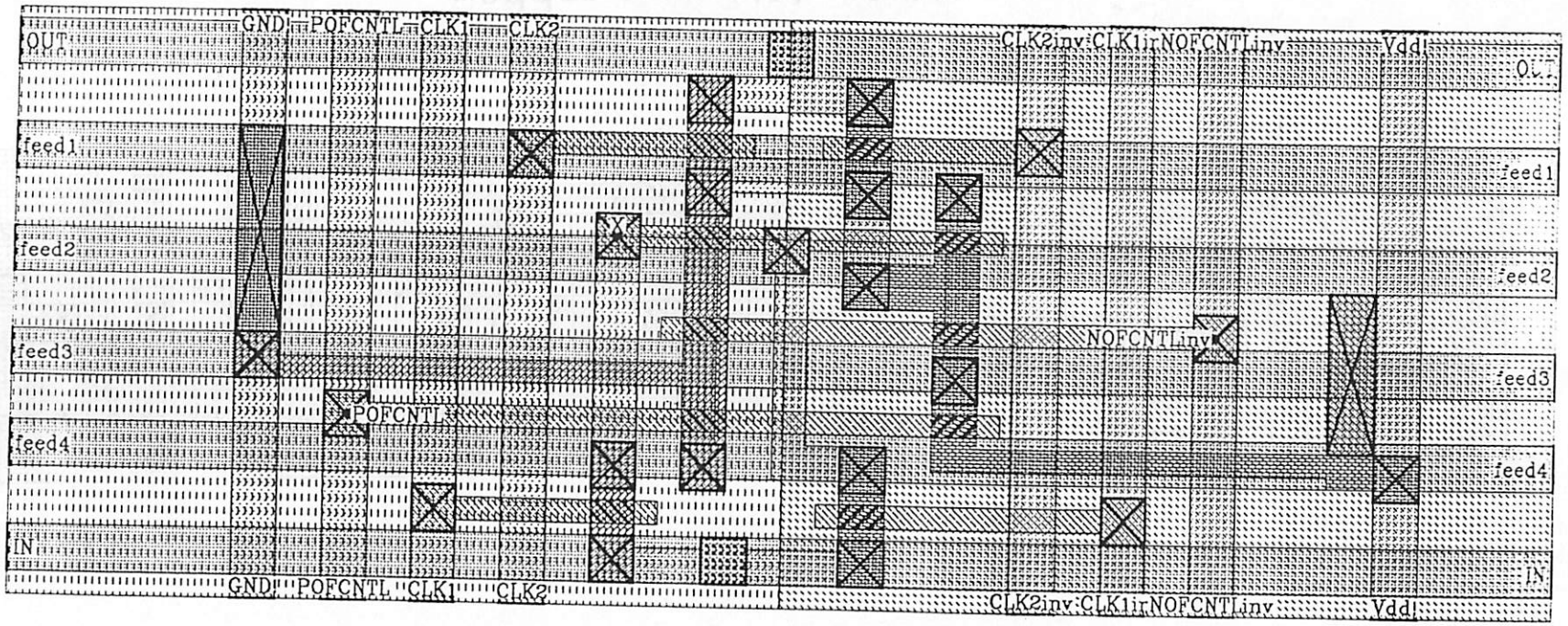
REGISTERS



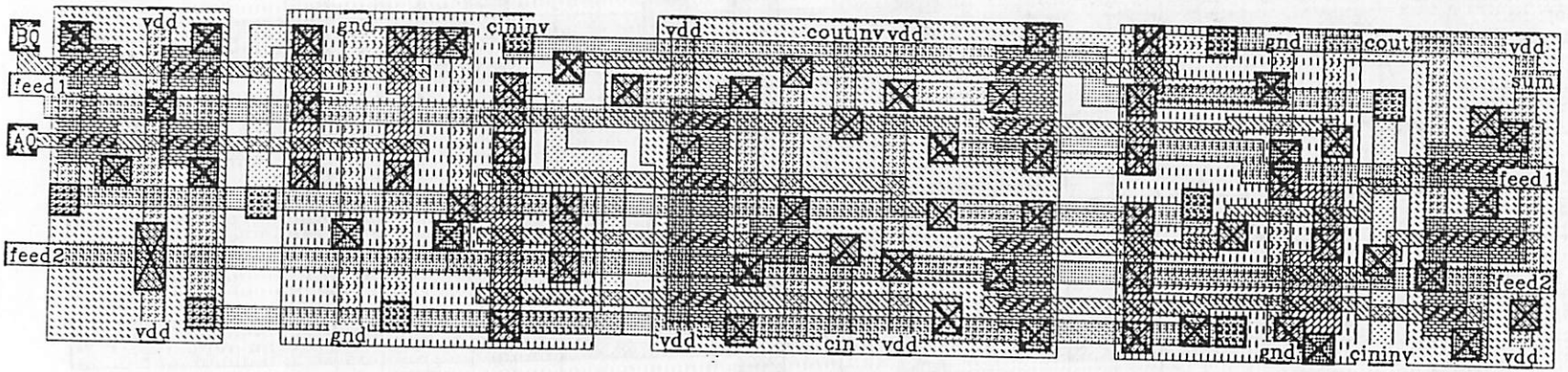
ZEROS

Appendix D

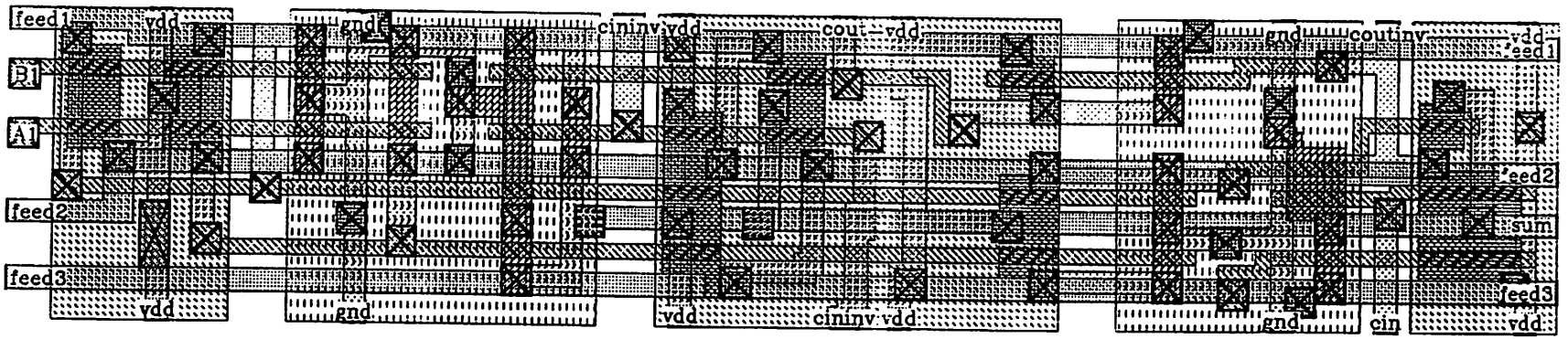
Magic Layouts of the Cells used in the Lager AUIO$_{INC}$ Data Path



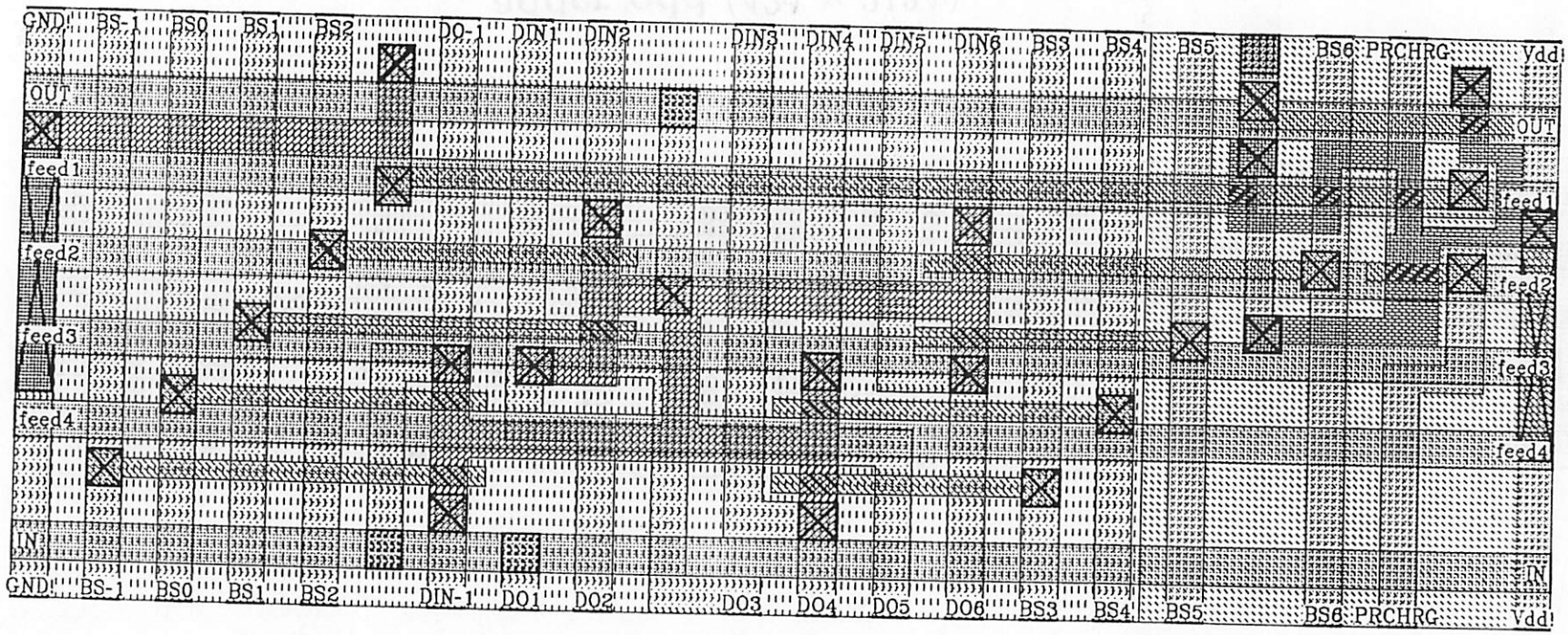
accumulator $(51\lambda \times 137\lambda)$



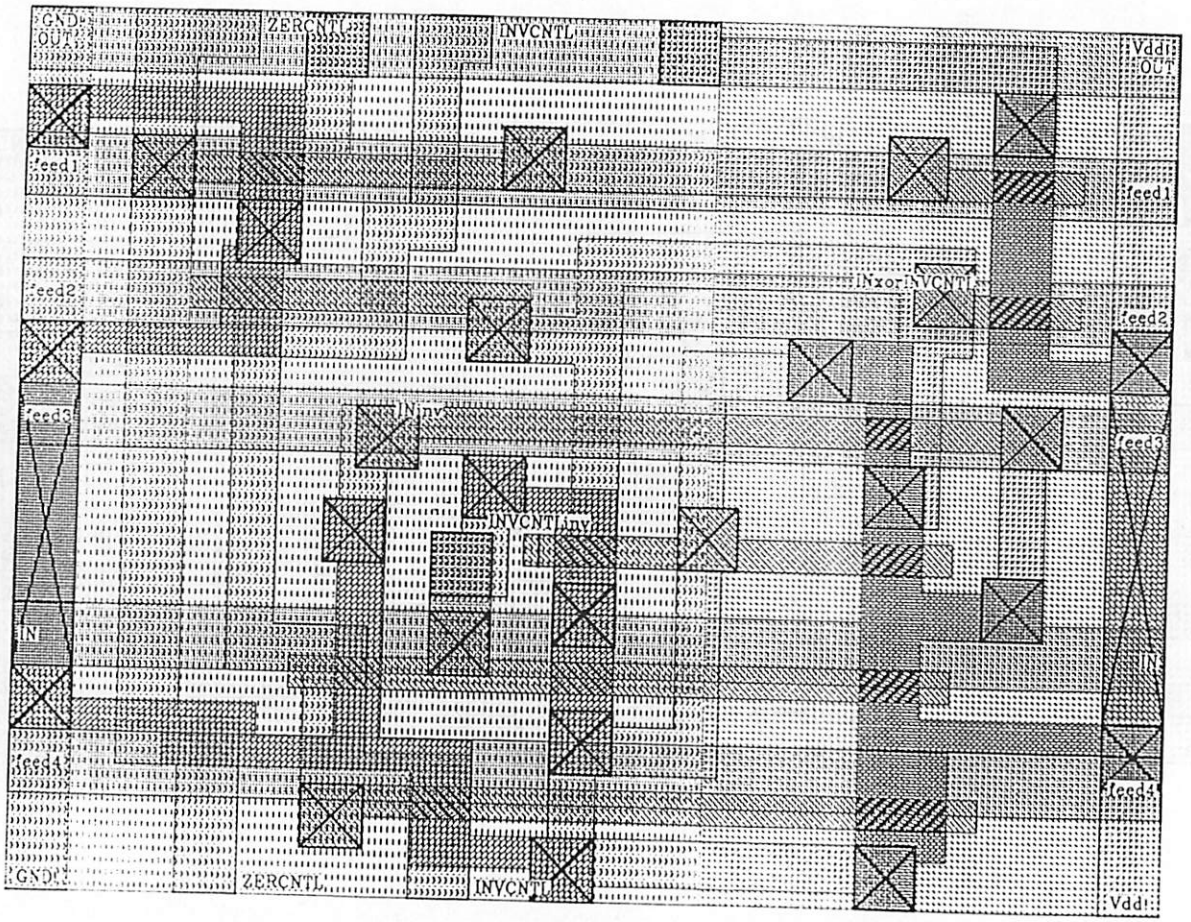
adder_even $(45\lambda \times 212\lambda)$



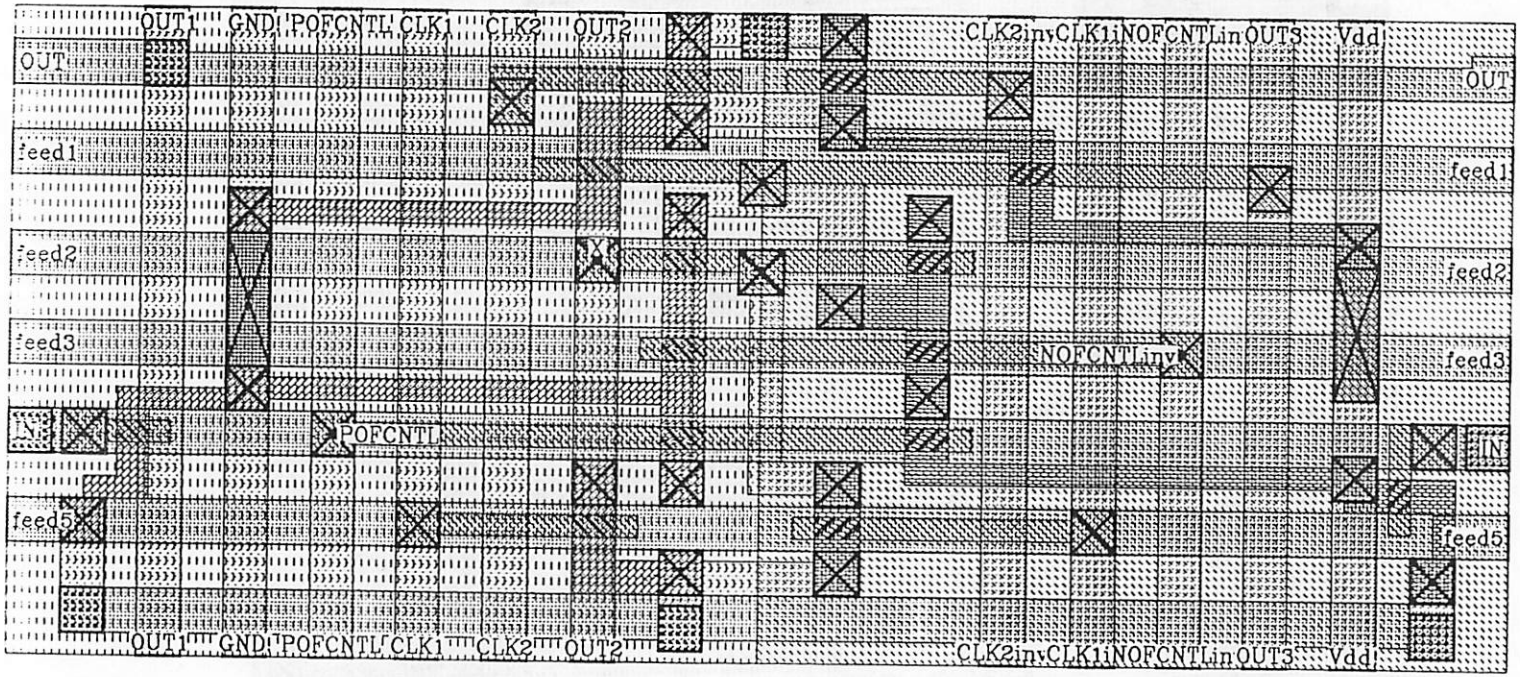
adder_odd $(42\lambda \times 212\lambda)$



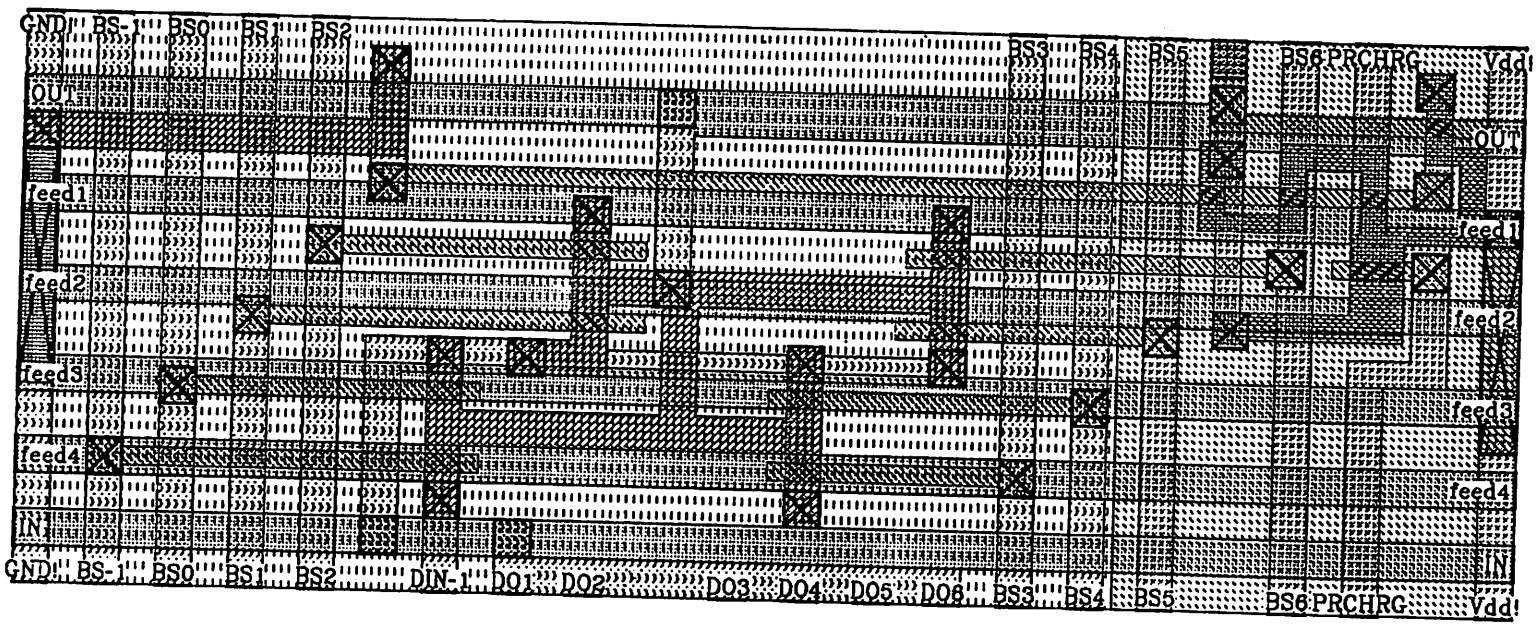
barrelshifter $(63\lambda \times 168\lambda)$



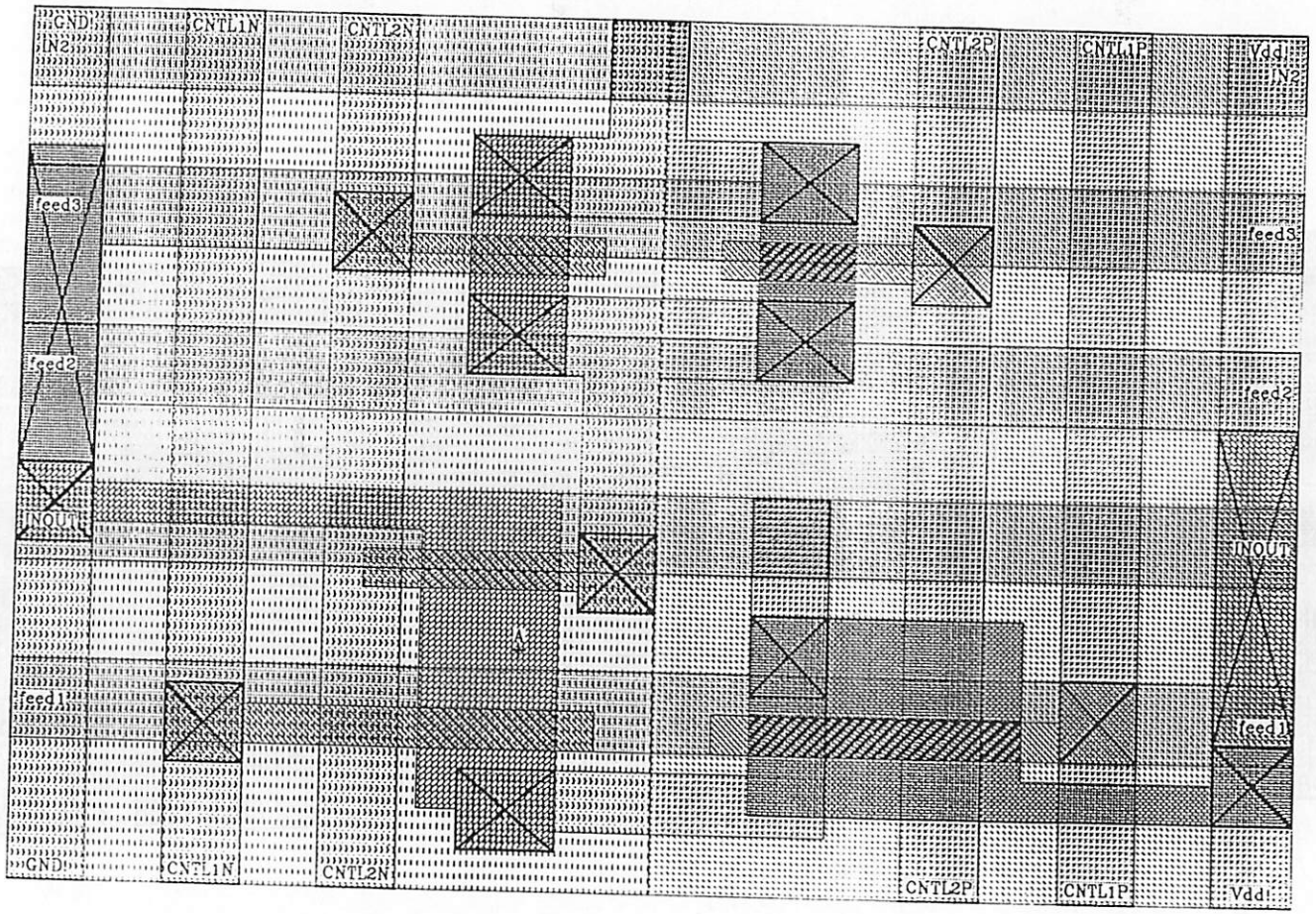
condcomp $(56\lambda \times 75\lambda)$



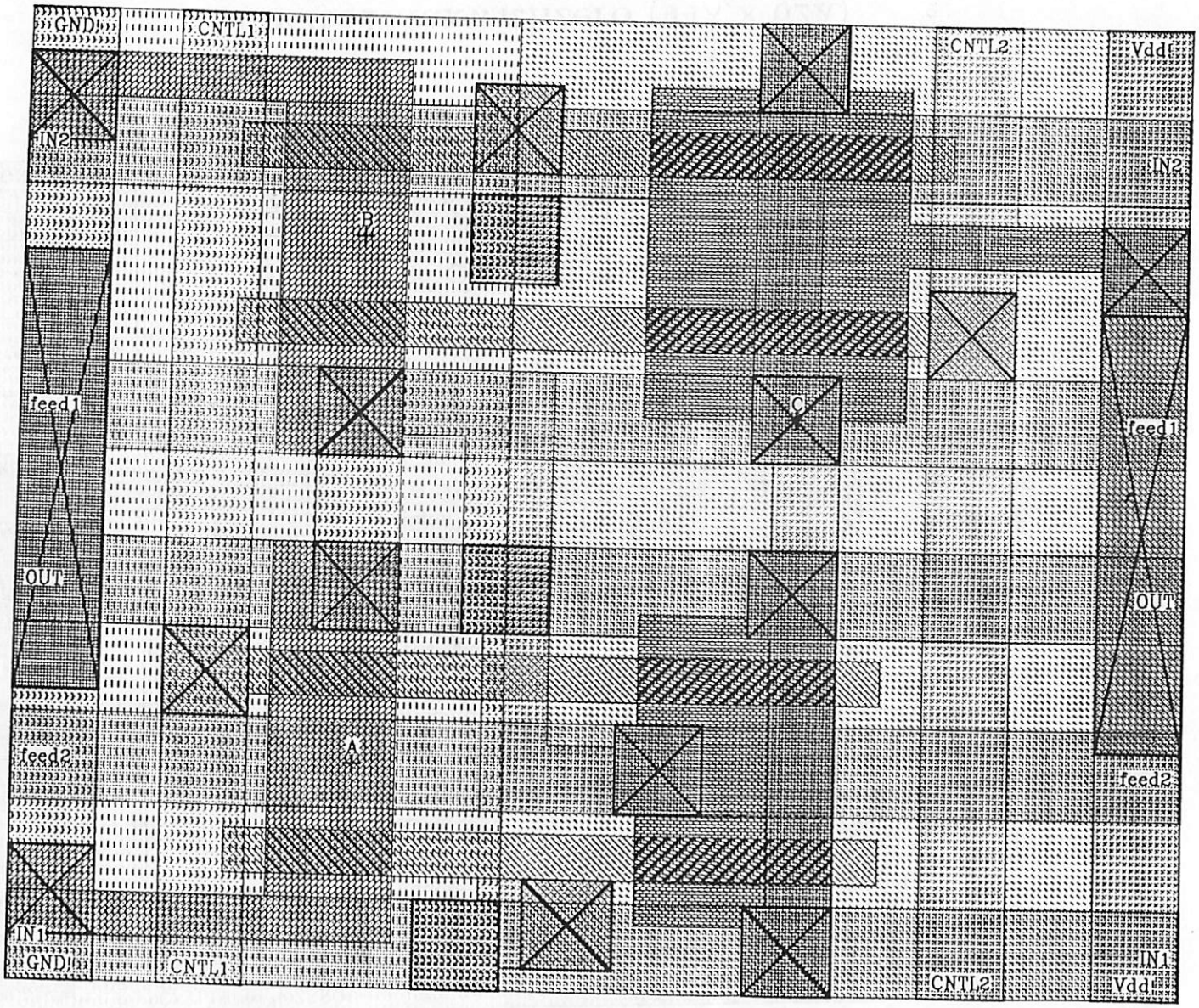
msb_acc $(58\lambda \times 137\lambda)$



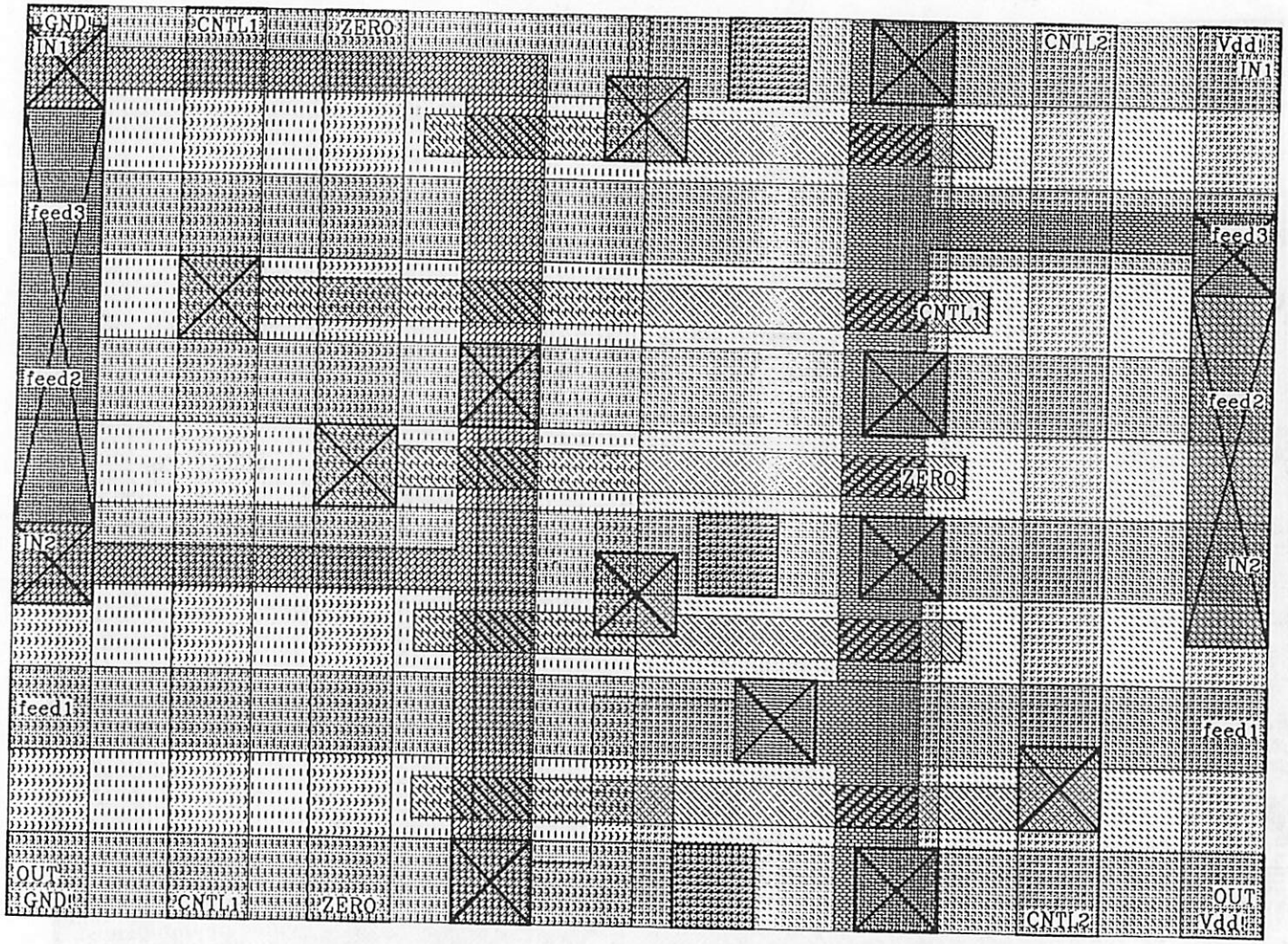
msb_barrel ($63\lambda \times 168\lambda$)



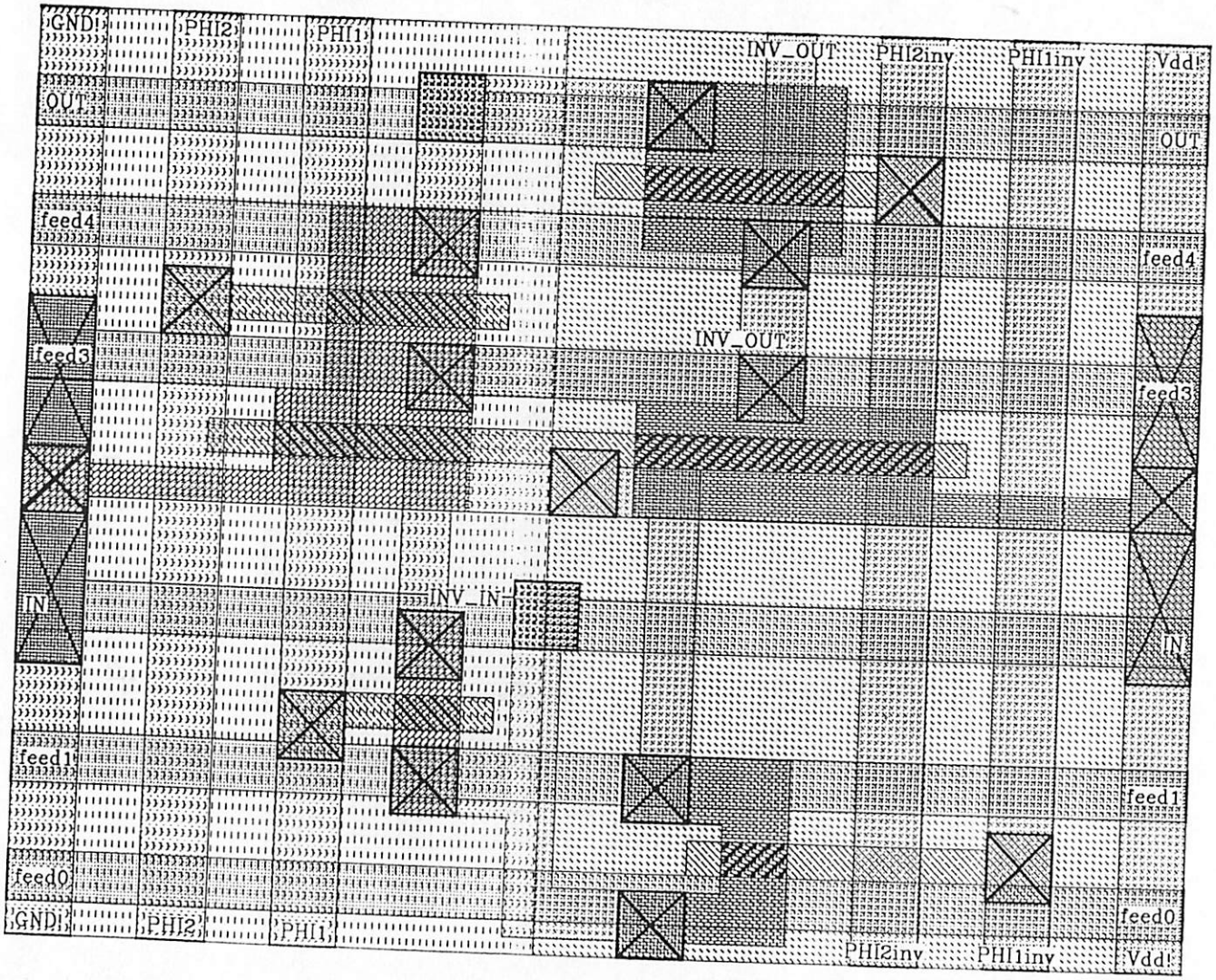
inputstage $(44\lambda \times 66\lambda)$



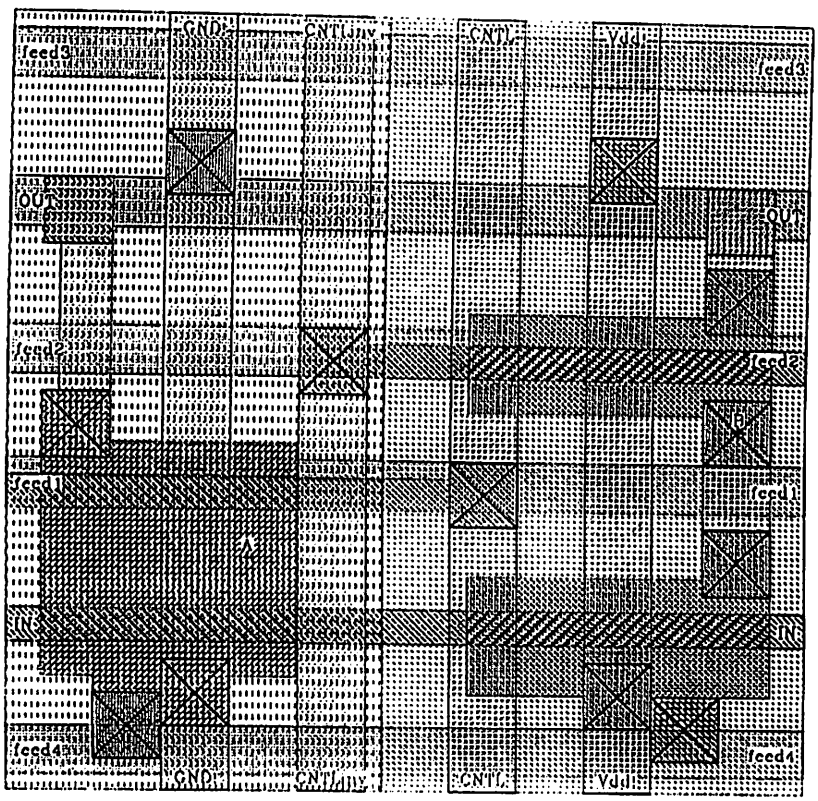
inv2to1mux $(44\lambda \times 54\lambda)$



inv2to1muxwithzero $(44\lambda \times 62\lambda)$



invlatch $(55\lambda \times 71\lambda)$



tristatebuffer $(47\lambda \times 47\lambda)$