UC Irvine
ICS Technical Reports

Title
ChipEst-FPGA : a tool for chip level area and timing estimation of lookup table based FPGAs for high level applications

Permalink
https://escholarship.org/uc/item/39b894v6

Authors
Xu, Min
Kurdahi, Fadi J.

Publication Date
1995-10-04

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

SLBAR
Z
699
C3
no. 95-31

ChipEst-FPGA: A Tool for Chip Level Area and Timing Estimation of Lookup Table Based FPGAs for High Level Applications

Min Xu†
Fadi J. Kurdahi‡

Technical Report #95-31
Oct. 4, 1995

†Department of Information and Computer Science
‡Department of Electrical and Computer Engineering
University of California, Irvine
Irvine, CA 92717-3425
(714) 824-8104

mxu@ics.uci.edu
kurdahi@ece.uci.edu

Abstract

The importance of efficient area and timing estimation techniques for hierarchical design methodology is well-established in High-Level Synthesis (HLS), since the estimation allows more realistic exploration of the design space, and hierarchical design methodology matches well with HLS paradigm. In this paper, we present ChipEst-FPGA, a chip level estimator for designs implemented using a hierarchical design methodology for Lookup Table Based FPGAs. In FPGAs, the wire delay may contribute up to 60% of the overall design delay.ChipEst-FPGA uses a realistic model which takes the component area/delay as well as wiring effects into account.We tested our ChipEst-FPGA on several benchmarks and the results show that we can get accurate area and timing estimates efficiently.

Contents

1 **Introduction** 4

2 **Overview of Xilinx XC4000** 6

2.1 XC4000 Configurable Logic Blocks and Lookup Tables 7

2.2 XC4000 Programmable Interconnect Point and Routing Resources 8

2.3 Xilinx hierarchical design flow support 9

2.4 Wiring Delay in FPGAs 10

3 **Previous Work** 10

4 **Chip Estimation** 11

4.1 Problem Definition 11

4.2 Chip Area Estimation 12

4.2.1 Component Shape Function 12

4.2.2 Chip Level Area Model 13

4.3 Chip Level Timing Estimation 18

4.3.1 Predict the Pin Location on Each Leaf Block 18

4.3.2 Predict wiring delay 18

4.3.3 Predict Clock Cycle 23

5 **Experimental Results** 25

5.1 Experimental Procedure 25

5.2 Results 26

6 **Conclusion** 27

7 **References** 28

List of Figures

1 Flat HLS design flow vs. hierarchical HLS design flow targeted for FPGAs: (a) flat design flow (b) hierarchical design flow. 4
2 The importance of estimation in a typical hierarchical HLS design flow targeted for FPGAs. 5
3 XC4000 family members. 7
4 Xilinx XC4000 architecture (a) XC4000 (b) abbreviated CLB architecture (c) a LUT implementing $x = mn + \overline{n}p$. 7
5 One XC4000 CLB can not implement all the functions with less than 10 distinct inputs (a) 9 literals, 9 distinct inputs need 1 CLB (b) 9 literals, 6 distinct inputs need 2 CLBs. 8
6 Wiring architecture (a)single length lines (b)double length lines (c) the XC4000 switch metric, connections and PIP. 9
7 Net length does not necessarily correlate well with performance (a) same Manhattan distance connections with different delays; (b) different Manhattan distance connections with same delays. 10
8 An Example: (a) Topologies for 4X1 Mux, (b) Shape Function for 4X1 Mux 12
9 Constructive/analytical area estimation technique 13
10 An Example:(a) Netlist; (b) Slicing without the cutting edge threshold;(c) Slicing with the cutting edge threshold 14
11 Wire budget (a) with horizontal slice line (b) with vertical slice line. 15
12 Constructive prediction: (a) the slicing tree, (b) possible AB decomposition, and (c) predicted shape function of AB (points shown as block squares) . . 16
13 Approximate topology for HAL 17
14 Pin Location: (a) Determine source pin location; (b) Determine sink pin location 19
15 Point-to-point delay model and associated parameters. 20
16 Routing rule algorithm. 21
17 Delay Prediction Algorithm. 22
18 The parameters for the delay model and timing optimization. 22
19 Fanout Effects 23
20 Typical Timing Model for a Digital System 24
21 Determining the minimum clock cycle time 25

22 Experimental procedure 26
23 Experiment Results 27

1 Introduction

The ability to shorten development cycles has made Field Programmable Gate Arrays(FPGAs) an attractive alternative to standard cells and Mask Programmed Gate Arrays (MPGAs) for the realization of Application-Specific Integrated Circuits (ASICs). High Level Synthesis (HLS), on the other hand, is becoming the methodology of choice for shortening the design time by allowing the user to start from a behavioral specification. Thus, the marriage of these two concepts provides an ideal testbed for fast prototyping starting from an idea to a final product.

HLS generates an architecture from a behavioral specification subject to constraints on area and delay. Following that, the design process of FPGAs can be decomposed into three major steps as shown in Figure 1(a). First, **partitioning** (or **technology mapping**), which includes Lookup Table (LUT) mapping and Configurable Logic Block (CLB) construction, partitions the incoming logic into a netlist of CLBs. Following that, **placement** determines a good assignment for the CLBs in the FPGA array. Once the placement is known, **routing** decides the type of routing resources and route for each net. This is a flat design approach since the netlist fed into partitioning is a gate level netlist and the partitioning is done on the whole netlist (more detailed discussion can be found in [9]).

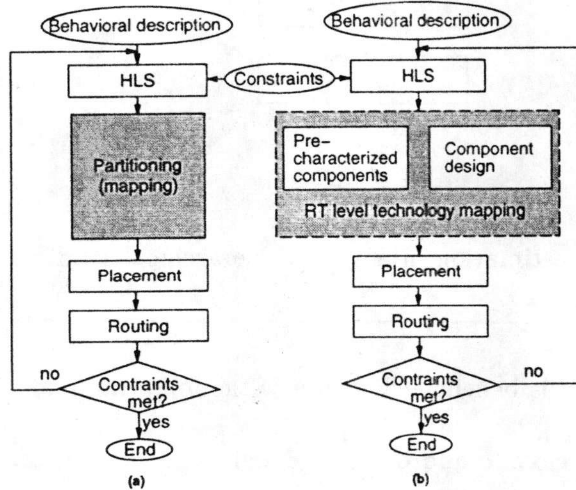

Figure 1: Flat HLS design flow vs. hierarchical HLS design flow targeted for FPGAs: (a) flat design flow (b) hierarchical design flow.

Contrast to this flat design flow, Figure 1(b) shows a hierarchical HLS design flow tar-

geted for FPGAs. It has an **RT level technology mapping** step which partitions the incoming netlist onto RT level components [1] and maps them into pre-characterized components or uses layout tools do the components layout. This way, the structural information is preserved in each component.

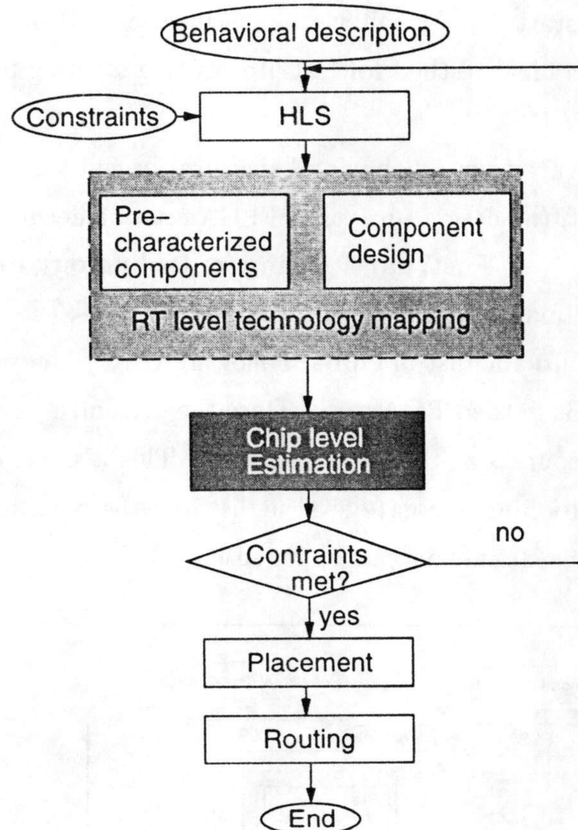

Figure 2: The importance of estimation in a typical hierarchical HLS design flow targeted for FPGAs.

Maintaining this hierarchy is beneficial because of the following reasons.

- It is easy to do debug, easy to add or change logic since design changes in one component can be made without affecting the placement and routing of the rest of the design.
- It is easy to adapt to different technology.

[1] we refer to individual registers, counters, adders, muxes, RAM arrays etc as RT level components.

- It is easy to improve the design routability by grouping and floorplanning the RT components according to the data flow. It is easy to improve the design's performance.
- It matches well with the HLS design paradigm since the hierarchy is maintained through out the design process.

In the hierarchical HLS design flow targeted for FPGAs, placement and routing make the design very unpredictable and the resultant design may violate the constraints. The reason is that in most FPGA designs, the wire delay, which is not considered in HLS, may contribute up to 60% of the overall design delay. The problem becomes especially acute when the design process starts at the behavioral level using HLS. In this case, a large number of candidate RTL designs are generated and must be evaluated to select the best design. Abstract cost measures which do not consider layout effects are likely to result in suboptimal designs. Thus, the design process may have to go through several iterations to reach an acceptable solution. Since placement and routing are usually quite time consuming, this may offset any turnaround time advantages of FPGAs and HLS. Indeed, such common situations have been reported in [1]. To avoid unnecessary iterations and shorten the design cycle, it is very helpful to have an estimator giving area and timing estimates quickly before actually going through the time consuming placement and routing phases as shown in Figure 2. It is very important that the estimator has a more realistic and accurate model which takes into account not only component area and timing, but also wiring effects.

One important class of FPGAs, implemented by Xilinx, uses LUTs to implement combinational logic and is called LUT based FPGAs. Xilinx has three logic cell array families of LUT based FPGAs including XC2000, XC3000 and XC4000. They share a common structure: an array of CLBs surrounded by configurable interconnect and they differ in details of the logic and interconnect structures. In this paper, we will concentrate on chip level area and timing estimation for LUT based FPGAs for designs implemented using a hierarchical design methodology. To be specific, we target the Xilinx XC4000 series because of their popularity. As stated before, our intended application domain is HLS since this is where fast and accurate estimation is most needed to support a high quality rapid prototyping environment.

2 Overview of Xilinx XC4000

Xilinx XC4000 consists of an array of CLBs embedded in a configurable interconnect structure and surrounded by configurable I/O blocks as shown in Figure 4(a). The Xilinx

Member	CLB Array Size	IOs	Gate Capacity	
			max	typical
XC4002A	8x8	64	3000	2000
XC4003A	10x10	80	4500	3000
XC4003H	10x10	160	4500	3000
XC4004A	12x12	96	6000	4000
XC4005/5A	14x14	112	7500	5000
XC4005H	14x14	192	7500	5000
XC4006	16x16	128	9000	6000
XC4008	18x18	144	12000	8000
XC4010	20x20	160	15000	10000
XC4013	24x24	192	20000	13000

Figure 3: XC4000 family members.

XC4000 family consists of ten members as shown in Figure 3. The family members differ in the number of CLBs, (ranging from 8x8 to 24x24), and I/O blocks, (ranging from 64 to 192). The typical gate capacity varies from 2000 to 13000.

2.1 XC4000 Configurable Logic Blocks and Lookup Tables

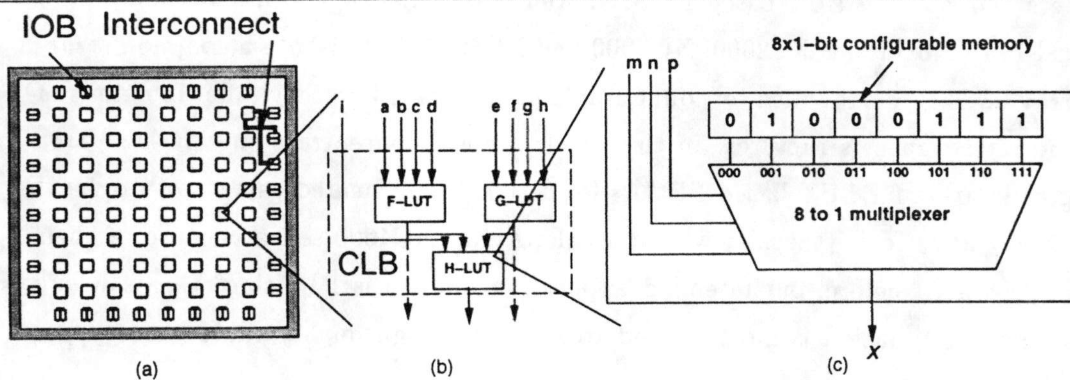

Figure 4: Xilinx XC4000 architecture (a) XC4000 (b) abbreviated CLB architecture (c) a LUT implementing $x = mn + \overline{n}p$.

Xilinx XC4000 CLBs mainly consist of two 4-input LUTs, which are called F-LUT and

G-LUT respectively, and one 3-input LUT, which is called H-LUT as shown in Figure 4(b). A K-input LUT is a memory that can implement any Boolean function of K variables. The K inputs are used to address a 2^Kx1-bit memory that stores the truth table of the Boolean function. For example, Figure 4(c) illustrates the structure of a 3-input LUT for implementing function $x = mn + \overline{n}p$. The truth table of the function is stored in an 8x1-bit memory, and an 8 to 1 multiplexor, controlled by the variables m, n. p, selects the output value x. All the CLB outputs can be either direct, inverted or registered.

Note that one XC4000 CLB, although it can accept 9 distinct inputs, is not equivalent to a 9-input LUT. A 9-input LUT can implement any functions of 9 distinct variables while one XC4000's CLB can implement any functions of 5 distinct inputs and some functions of 6 to 9 variables. For example, expressions in both (a) and (b) in Figure 5 have 9 literals, but (a) needs 1 CLB whereas (b) needs 2 CLBs.

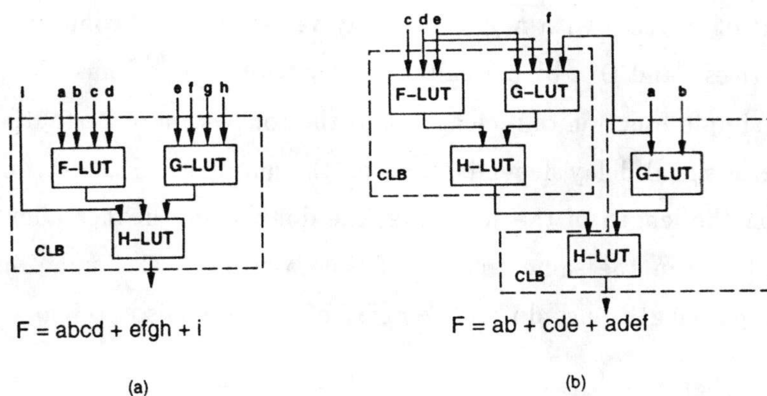

Figure 5: One XC4000 CLB can not implement all the functions with less than 10 distinct inputs (a) 9 literals, 9 distinct inputs need 1 CLB (b) 9 literals, 6 distinct inputs need 2 CLBs.

2.2 XC4000 Programmable Interconnect Point and Routing Resources

Xilinx XC4000 routing resources are connected by switch matrices. There are 8 (6 for smaller devices) intersections containing 6 programmable interconnect points (PIPs) each. The PIP, shown schematically in Figure 6(c), is a pass transistor controlled by a configuration memory cell.

XC4000 routing resources include *single-length (general-purpose) lines* (SLs), shown in Figure 6(a), *double-length lines* (DLs), shown in Figure 6(b) and *long lines* (LLs). LLs run

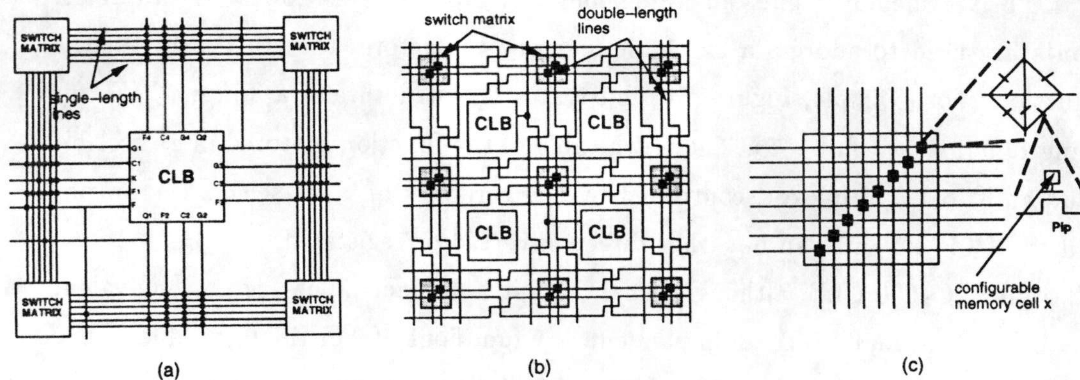

Figure 6: Wiring architecture (a)single length lines (b)double length lines (c) the XC4000 switch metric, connections and PIP.

the width or the height of the chip with negligible delay variations. SLs connect every pair of adjacent switch matrices [2] and DLs by-pass alternate switch boxes [3]. Thus, the wirability of a net is no longer a simple function of its length and the congestion of its routing region. On the other hand, since signal delay depends more on the number of PIPs through which a signal passes than on the length of the segments, the double-length lines allow a signal to travel twice the distance in the same amount of time, or to travel a same distance in half the time as the single length lines do [4]. The delay of a wire is also no longer a simple function of its length.

2.3 Xilinx hierarchical design flow support

In later versions of their CAD tools, Xilinx appears to be moving towards promoting hierarchical design flow by introducing **Hard Macros, hmgen,** and **RPMs.**

- Hard Macros are encoded files representing segments of XC4000 Logic Cell Array (LCA) logic that are mapped into LCA logic blocks, then placed and routed for a specific FPGA part.
- hmgen is the program to generate hard macro. It is often used to force a high degree of structure on a design since a well designed hard macro can boost design performance.

[2]The wire between two adjacent switch matrices is a SL segment.
[3]The wire connect every other switch matrices is a DL segment.
[4]Experiments show that SL segments and DL segments have approximately the same delay.

- RPMs are pre-designed netlist (soft macros) but no routing is accomplished until final chip assembly. They group logic into logic configurable array blocks where appropriate. RPMs enhance hard macros and achieve higher flexibility.

By using Hard Macros or RPMs, the component placement information is preserved in hard macros.

2.4 Wiring Delay in FPGAs

In Xilinx architecture, there are three types of wiring, i.e. single-length lines, double-length lines, and long lines. Thus, the delay of a net is mainly decided by the number of segments and the number of PIPs it goes through. As a result, net length does not necessarily correlate well with either routability or performance. For example, Figure 7(a) shows that nets with the same distance may have different delays, while Figure 7(b) shows that nets with different distances may have the same delays. Because of this, standard delay models cannot be directly applied to FPGAs.

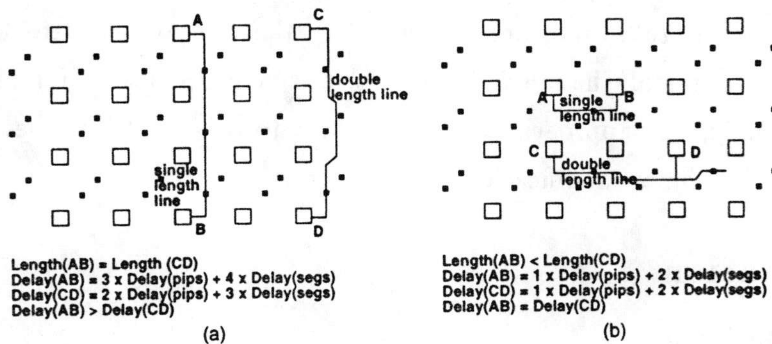

Figure 7: Net length does not necessarily correlate well with performance (a) same Manhattan distance connections with different delays; (b) different Manhattan distance connections with same delays.

3 Previous Work

Several fast mapping heuristics for LUT based FPGAs are surveyed in [6]. Such heuristics can be used to obtain estimation of CLB count. However, techniques for timing estimation haven't been proposed so far.

Xilinx's [3] Partitioning, Placement and Routing (PPR) software package has its own built-in estimation tool. This estimation is very accurate since it performs the actual mapping using Chortle [4], but the tool does not provide performance estimation.

Other than Xilinx, Synopsys [5] also provides accurate area estimation by doing actual mapping. Moreover, it can provide estimation of the number of logic levels for the design. Nevertheless, it doesn't take into account wiring delay.

The research presented in [2] empirically examines the performance of multi-level logic minimization tools for a LUT based FPGA technology and suggests that there is a linear relationship between the number of literals and the number of routed CLBs. It provides estimation for both area and timing but the work is only applicable to the XC3000 series.

CompEst-FPGA [9] presented an area and timing estimation for LUT based FPGAs approach. It takes into account gate area/delay as well as wiring effects. It can handle Xilinx XC4000 estimation.

All those approaches are suitable to estimate component level design and chip level design but with flat design methodology. None of them supports hierarchical design methodology.

The work presented here is the extension of our work presented in [9]. It has a realistic and accurate model since it takes into account not only the component area/delay but also the wiring effects. It mainly handles hierarchical design methodology for high level applications. Additionally, our approach is easy to adapt to other Xilinx series such as XC2000 and XC3000 with minor modifications.

4 Chip Estimation

4.1 Problem Definition

Given an RT level description, the goal of Chip Area Estimation is to predict the area of the chip in terms of number of CLBs as well as the most proper device it may fit by using the area information of all the RT level components.

Given an RT level description, the goal of Chip Timing Estimation is to estimate the performance of the chip in terms of clock cycles by using the delay information of all the RT level components along with the estimated topology information obtained from Chip Area Estimation.

4.2 Chip Area Estimation

Our chip level area model uses a slicing tree techniques derived from [8] for evaluating the area of designs implemented using RT level components.

4.2.1 Component Shape Function

To improve the density of the chip, designers may try different floorplans by varying the topological placements of each component. Component *shape function* represents the different topological placements in the actual layout and their corresponding delay information. For example, a 4X1 mux needs four CLBs. It can have three different topological placements in the actual layout Figure 8(a). These results in a shape function are shown in Figure 8(b).

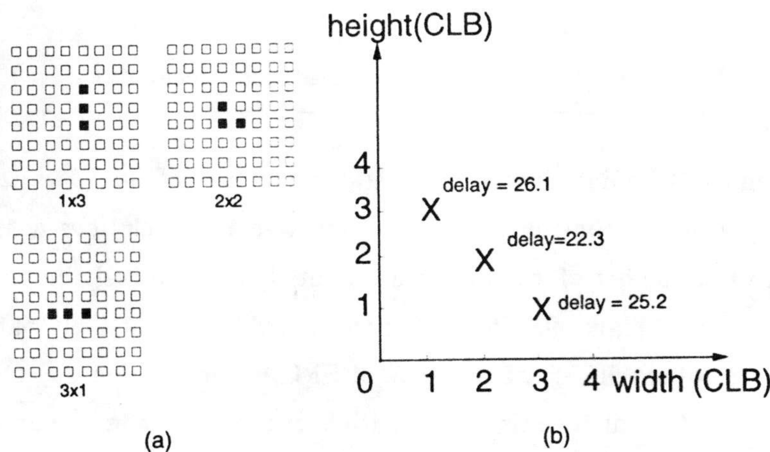

Figure 8: An Example: (a) Topologies for 4X1 Mux, (b) Shape Function for 4X1 Mux

At RT level, the shape function of some components can be obtained from our library which is a collection of hard macros with shape function and delay information. The collection of hard macros includes components those are frequently used in the design so we pre-characterized their shape function. Also, it includes vendor supplied pre-designed components such as hard macros in Xilinx library etc. For the components whose shape function is not known *a priori*, (controller for example), their shape function can be obtained by invoking the Component Estimator, CompEst-FPGA, described in [9].

4.2.2 Chip Level Area Model

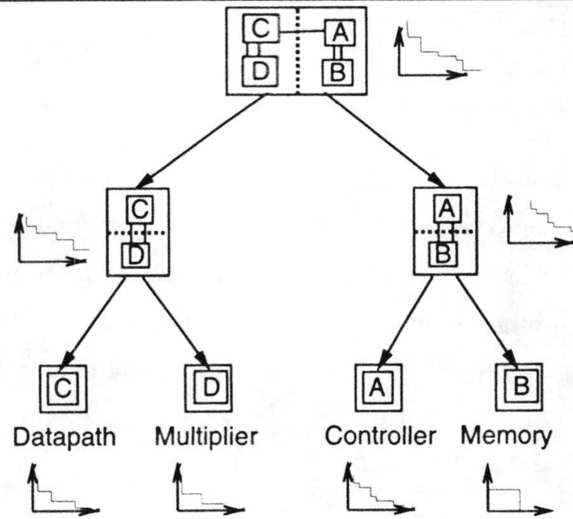

Figure 9: Constructive/analytical area estimation technique

The chip level slicing tree technique involves slicing down to the leaf blocks which consists of either RT level components or controller. This constructive approach does not consume excessive runtime since the number of leaf blocks are limited to a relatively small number. This technique is illustrated in Figure 9. The slicing tree is built by recursively partitioning the input design. Because of specific characteristics of FPGA, partitioning objectives have to be selected accordingly. One of them is minimization of routing resource consumption. It is mainly accomplished by devising data objects that will partition in such a way as to permit the greatest number of signals to traverse the shortest distances along the fewest routing channels with the least crossovers. This most often means placing interconnected objects adjacent to each other with related elements aligned to the routing axes.

Because of the granularity of FPGA (the area is in terms of CLBs rather than in terms of micron, e.g. the Xilinx XC4013 has 24 by 24 CLBs rather than thousands by thousands square microns in the custom design), reducing unused area is a very important objective. To achieve this, objects with similar sizes are placed adjacent to each other because this can minimize the wasted area. Sometimes, this will conflict with the objective to put strongly connected blocks adjacent to each other. We introduce a *cutting edge threshold* in our algorithm to trade off between area and performance. The cutting edge threshold actually is a parameter obtained by calculating the average size of all the blocks to be partitioned,

if some block's size exceeds the cutting edge threshold (that means it is far bigger than the rest of the blocks, it will be isolated from the rest of the blocks and be a sub-slice of the current slice. For example shown in Figure 10(a), the netlist contains 4 components, Mult needs 60 CLBs, two registers needs 16 CLBs each, one Mux needs 8 CLB. If we only consider the interconnection between them, we will end up with a 12x12 CLB device as shown in Figure 10(b) , if we consider the cutting edge threshold, the Mult will be isolated from the rest of the blocks and be one sub-slice for slice 1234. The result with cutting edge threshold is a 10x10 CLB device as shown in Figure 10(c), we can see that slicing with the cutting edge threshold produces more area-efficient result.

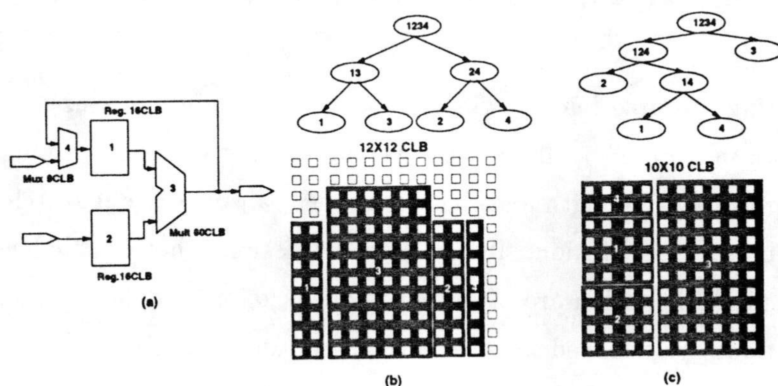

Figure 10: An Example:(a) Netlist; (b) Slicing without the cutting edge threshold;(c) Slicing with the cutting edge threshold

The shape function of the entire design is computed by constructively adding the shape function of these leaf blocks. In addition to the area of leaf blocks, the routing area used by the nets connecting these blocks also needs to be accounted for.

Because of the flexibility and symmetry of the CLB architecture, it facilitates the placement and routing. For leaf blocks, the inputs, the outputs and the function generators themselves can freely swap positions within CLBs of the components to avoid routing congestion. So, when we build up the shape function for the slice using shape function of two sibling slices at level i, our main concern about routing is the routing area between those two sibling slices. Routing budgets are given to each routing area, when the expected routing resources needed exceed the budget, the upper level slice will correspondingly be "dilated" so as to meet the need of routing. The expected routing resources can be obtained by estimating the interconnection count between two sibling slices, as described next.

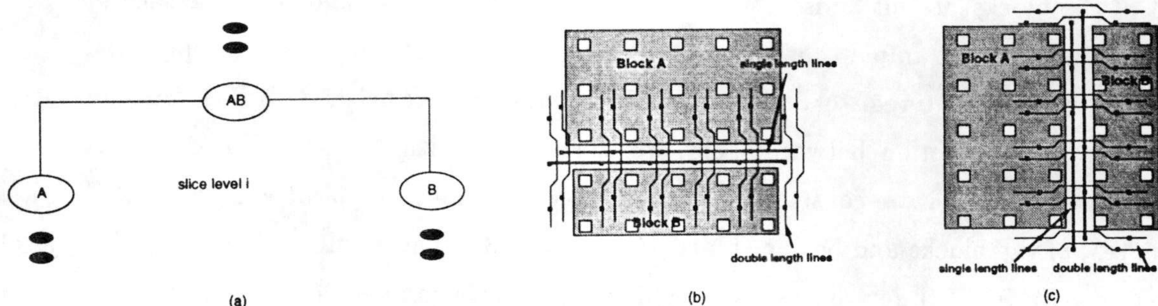

Figure 11: Wire budget (a) with horizontal slice line (b) with vertical slice line.

The available routing resource budget will depend on the shapes and sizes of the two sibling blocks as shown in Figure 11. In the intervening routing channel between the two sibling blocks, there are six single-length lines between every pair of adjacent switch matrices that are parallel to the slice orientation. In addition, we assume that double-length lines perpendicular to the slice orientation are also used in that channel, while the ones parallel to the slicing orientations are reserved for the parent level in the slicing tree. Let W_p be the width of the current parent slice, H_p be the height of the current parent slice, the total available routing budget can be calculated based on the size of the slicing cut (i.e. the length of the routing channel) between the two sibling blocks as follows:

$$total_routing_budget = \begin{cases} \alpha * W_p + 2 * (W_p + 1) & \text{for horizontal slicing} \\ \alpha * H_p + 2 * (H_p + 1) & \text{for vertical slicing} \end{cases}$$

where α is the single lines count: 6 for bigger devices and 4 for smaller devices.

Once we get the routing resources budget for the parent slice from the shape of two sibling slices, we can decide whether or not adjustment is needed according to whether or not the number of interconnection is exceeding the budget. If there is a need to adjust, (whether horizontal or vertical), the budget will then be appropriately increased by having more single length lines. Take a vertical slice as an example, whenever we increase one CLB in the horizontal direction, we will have two additional columns' worth of single length lines. Again, let W_p and H_p be the width and height of the parent slice, *num_con* be the expected number of interconnection between two sibling slices, we can estimate the increased area and the increased area can be added to the total estimated area of block AB by using the following equations as shown in Figure 12(b):

$$total_area = \begin{cases} W_p * (H_p + \triangle H_p) & \text{for horizontal slicing} \\ H_p * (W_p + \triangle W_p) & \text{for vertical slicing} \end{cases}$$

where:

$$\triangle W_p = Max\{0, (num_con - total_routing_budget)/2 * H_p * \alpha\};$$
$$\triangle H_p = Max\{0, (num_con - total_routing_budget)/2 * W_p * \alpha\};$$

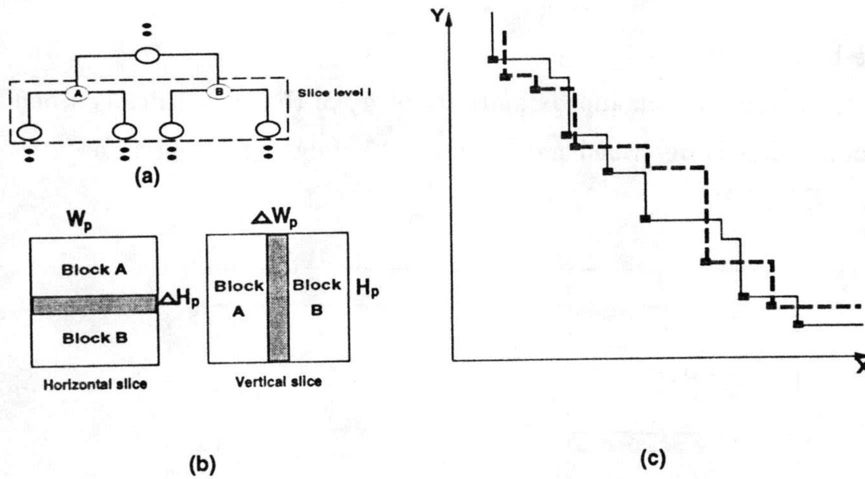

Figure 12: Constructive prediction: (a) the slicing tree, (b) possible AB decomposition, and (c) predicted shape function of AB (points shown as block squares)

This process of building the composite blocks is performed in a post-order manner from the leaves of the slicing tree towards the root. Thus determining the area of the entire design. For each two sibling blocks, there exists two possible ways of generating the parent block depending on the orientation of slice. Since only a prediction of the chip dimensions is desired, we need not perform an actual floorplan of the chip from the slicing tree. Therefore, we need not decide on the orientation of the slice line when traversing the slicing tree bottom up. For each two siblings, two shape functions of the parent block are generated: one assuming a horizontal slice and another assuming a vertical slice. The two curves are then superimposed and a "lower bound" curve is generated by keeping only the smaller of the two slice orientations at each x as shown in Figure 12(c). The resulting shape function is taken as the set of predicted dimension pairs for the optimal layout area of the parent block.

At the end of this phase, we can estimate the area of the overall chip, according to the number of I/O, we can predict whether the design can be fitted into one FPGA device or not, if it can be fitted, we can also predict the specific XC4000 device which will be the best choice. Let W, H, num_io be the estimated width , height, and number of IOs of the chip respectively, W_1, W_2, H_1, H_2, num_io_1, num_io_2 be the width, height and number of IOs of two consecutive devices: $device_1$, $device_2$ respectively.

if

$$((W_1 < W \leq W_2)\textbf{AND}(H_1 < H \leq H_2)\textbf{AND}(num_io_1 < num_io \leq num_io_2))$$

then $device_2$ is the best choice.

At this moment, we also have an approximate topology of the chip which can be used in the subsequent timing models described next. Figure 13 shows one approximate topology of our one example: HAL [10].

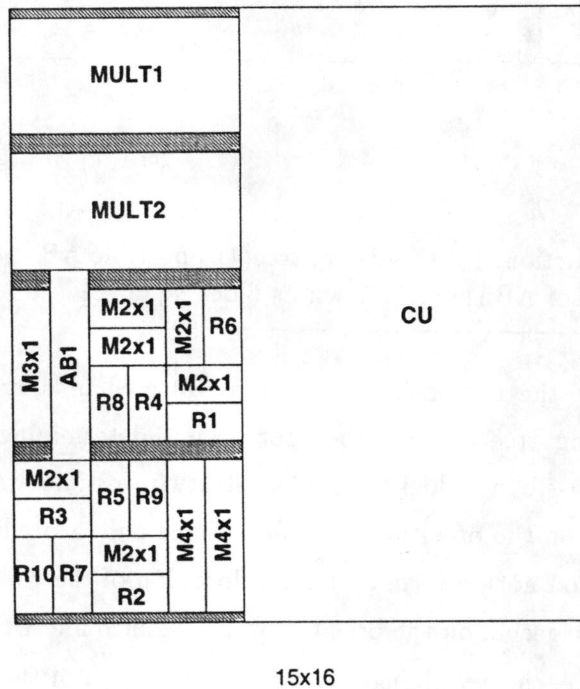

Figure 13: Approximate topology for HAL

4.3 Chip Level Timing Estimation

The Chip delay includes component delays, wire segment delay, and Programmable interconnection Point(Pip) delay. The Chip timing estimation model has the following phases:

- Predict the pin location on each leaf block
- Predict wiring delay
- Predict chip clock cycle

4.3.1 Predict the Pin Location on Each Leaf Block

Given an input RT level design, our chip level area model described in Section 4.2.2 outputs an approximate floorplan which provides estimates of the relative locations of the constituent blocks. To better estimate chip level timing, pin location must be either known or estimated. On those blocks which have been pre-designed, the pin location are known. For other components which have not been laid-out yet, we must estimate "preferred" location for each pin. location can be determined by evaluating the approximate topology of the design. Chip area estimation process determines the approximate locations of the blocks in the design taking routing area into account. For each net, first, we identify the source pin, then we identify load pins and their associated blocks. By evaluating the mean location of blocks associate with load pins, a "preferred" side location of each source pin is first determined. Then, by finding the shortest Manhattan distance between each pair of source and destination blocks, a preferred location of each sink pin can also be determined.

For example shown in Figure 14, the source pin of source block S have four destination blocks D1 through D4 as it's loads. By evaluating the mean location of D1 through D4, we can get the preferred source pin location shown in Figure 14(a). Then, by finding the shortest Manhattan distance between S and D1, S and D2, S and D3, S and D4, preferred location of each sink pin can also be determined shown by circles in Figure 14(b).

4.3.2 Predict wiring delay

To predict the delay between point A and B, $D(A, B)$, in Figure 15, the Manhattan distance x and y segments are first calculated using following formulas:

$$x = |A_x - B_x|$$

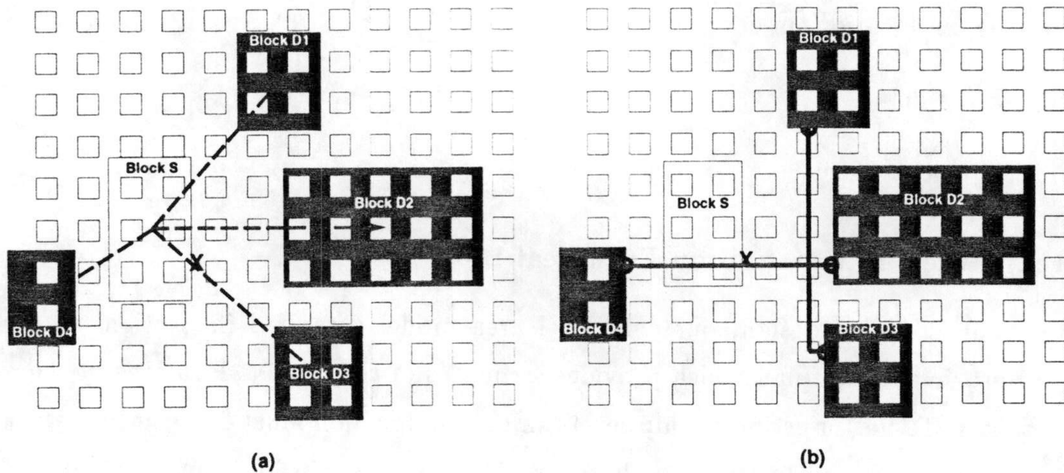

Figure 14: Pin Location: (a) Determine source pin location; (b) Determine sink pin location

and

$$y = |A_y - B_y|$$

Then, wire type (single-length line, double-length line and long line) is assigned to that wire as described in the following section. This decides the number of PIPs and number of segments between points A and B. Subsequently, the point-to-point delay (without fanout effects), $D_{pp}(A, B)$, can then be calculated. Finally, the delay with fanout effects, $D(A, B)$, can be obtained by adjusting the $D_{pp}(A, B)$ with a fanout factor as described below.

To predict the wire type, the algorithm mainly checks the interconnect wire length x and y respectively. First, long lines are assigned to all the wires which are longer than 8 CLBs in either direction. Then, single-length lines are assigned for all wires which are shorter than 2 CLBs. Note that single-length line can not be connected to double-length lines. Thus, if one segment of a wire is assigned to a single length line in one direction, then the other segment of the wire on the other direction is also assigned to a single length line if its length is between 2 and 8 CLBs. Finally, double-length lines are assigned to the rest of the interconnect wires.

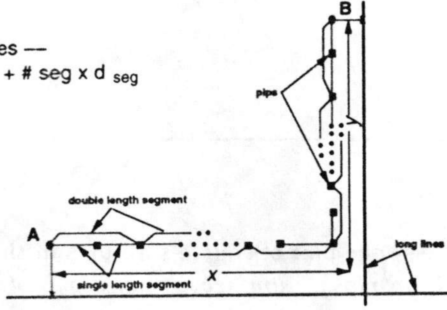

Figure 15: Point-to-point delay model and associated parameters.

The detailed algorithm is shown in Figure 16.

From Section 2.4, we know that net length does not necessarily correlate well with the actual delay. Therefore, we use an empirical model to characterize the delay-vs-wiring-type relationship. Our empirical model is based on a large number of observations obtained by using Xilinx's XDM layout tool to place and route a set of benchmarks and analyzing the delay of each point-to-point connection using *Xdelay*, the Xilinx timing analysis tool. We found that it is satisfactory to approximate the delay as a function of (1) the number of PIPs it goes through in both X and Y directions respectively, and (2) the corresponding segment delays. Let's denote the delay for each PIP in the programmable switch matrices as d_{pip}, and the delay for each segment as d_{seg}. Note that we use the same variable d_{seg} for both single-length and double-length segments since experiments show that their delays are approximately the same [5]. For a 2-point net (A, B), the point-to-point delay will be the summation of such delays in both X and Y directions. Let x and y be the Manhattan distances of (A, B) in X and Y direction respectively. If only single-length lines are used, they will pass through x and y PIPs, and through $x+1$ and $y+1$ segments in the X and Y directions respectively. Double-length lines need one PIP in every other CLB and, similarly, for segments on same distance as single-length line interconnection. Long lines with same length will not go through any PIPs and eventually the long line delay is approximated as being proportional to the wire length. Thus, the point-to-point delay will be:

$$D_{pp}(A,B) = \begin{cases} d_{pip} * x + d_{seg} * (x+1) + d_{pip} * y + d_{seg} * (y+1) & \text{for single length lines} \\ d_{pip} * \frac{x}{2} + d_{seg} * (\frac{x}{2}+1) + d_{pip} * \frac{y}{2} + d_{seg} * (\frac{y}{2}+1) & \text{for double length lines} \\ d_{ll} * (x+y) & \text{for long lines} \end{cases}$$

[5]The model can be easily modified to account for different delays of single-length and double-length segments, if needed.

Algorithm routing(x, y)

Inputs: Distance between one pair of source-sink pins in the approximate topologies: x, y.
Output: num_pips_x, num_pips_y, num_seg_x, num_seg_y, $total_num_pips$

begin Algorithm

```
if (x ≥ 8(CLBs)) then x use long line;
if (y ≥ 8(CLBs)) then y use long line;
if (x < 2(CLBs)) then x use single-length line;
    num_pips_x = x;
    num_seg_x = num_pips_x + 1;
    if (y < 8(CLBs)) then y use single-length lin;
        num_pips_y = y;
        num_seg_y = num_pips_y + 1;
    end if;
end if;
if (y < 2(CLBs)) then y use single-length line;
    num_pips_y = y;
    num_seg_y = num_pips_y + 1;
    if (x < 8(CLBs)) then x use single-length line;
        num_pips_x = x;
        num_seg_x = num_pips_x + 1;
    end if;
end if;
for the rest of x and y use double-length lines;
    num_pips_x = x/2;
    num_seg_x = num_pips_x + 1;
    num_pips_y = y/2;
    num_seg_y = num_pips_y + 1;
end for;
```

end Algorithm

Figure 16: Routing rule algorithm.

```
Algorithm point-to-point delay(x, y)

    Inputs: x, y,num_pips_x, num_pips_y, num_seg_x,num_seg_y .
    Output: point to point delay: delay_pp

begin Algorithm

    if ((x >= 8(CLBs)) AND (y >= 8(CLBs))) then /*assume long line*/
        delay_pp = α + p_ll * (n + y + 1);
    end if;
    if ((x >= 8(CLBs)) AND (y < 8(CLBs))) then /* assume x use long line */
        delay_pp = 2 * α + p_ll * x + d_seg * num_seg_y + d_pip * num_pips_y;
    end if;
    if ((y >= 8(CLBs)) AND (x < 8(CLBs))) then /* assume y use long line */
        delay_pp = 2 * α + p_ll * y + d_seg * num_seg_x + d_pip * num_pips_x;
    end if;
    if ((x < 8(CLBs)) AND (y < 8(CLBs))) then /*assume no long lines are used*/
        if((num_pips_x == 0)AND(num_pips_y == 0)) then
                /* assume direct interconnect without going through pips */
            delay_pp = β;
        else /* assume single/double-length interconnection */
            delay_pp = α + d_pip * (num_pips_x + num_pips_y) + d_seg * (num_segs_y + num_segs_x);
        end if;
    end if;

end Algorithm
```

Figure 17: Delay Prediction Algorithm.

ϵ	d_{pip}	d_{seg}	d_{ll}	α	β
2.5	0.3	0.3	0.18	0.6	1.3

Figure 18: The parameters for the delay model and timing optimization.

and the detailed algorithm is shown in Figure 17 and the associated parameters are listed in Figure 18.

When the number of fanout of a net is larger than one, say f, the delay on each sink pin j ($j = 1, ...f$) will be affected by the delay on the rest of sink pins k ($k = 1, ...f; k \neq j$) on the net. Let i be the source pin, for each sink pin j ($j = 1, ...f$). The point-to-point delay without fanout effect, $D_{pp}(i,j)$, is first computed. Afterwards, we denote $D(i,j)$ as the delay with fanout effects, and it can be obtained by adjusting the point-to-point delay without fanout effects, $D_{pp}(i,j)$, using the following formula:

$$D(i,j) = D_{pp}(i,j) + \frac{1}{\epsilon} \sum_{k=1,...f;k \neq j} D_{pp}(i,k)$$

Where, ϵ, a fanout adjustment factor, is experimentally obtained as 2.5. we can see that

the fanout delay effects at the chip level is quite big. This is because at the chip level, part of fanout effect could be masked by the components. For example in Figure 19, net n fans out form block A to two other blocks, B and C, so its RT level fanout is 2. However, the net actually feeds 5 CLBs when the design is flattened.

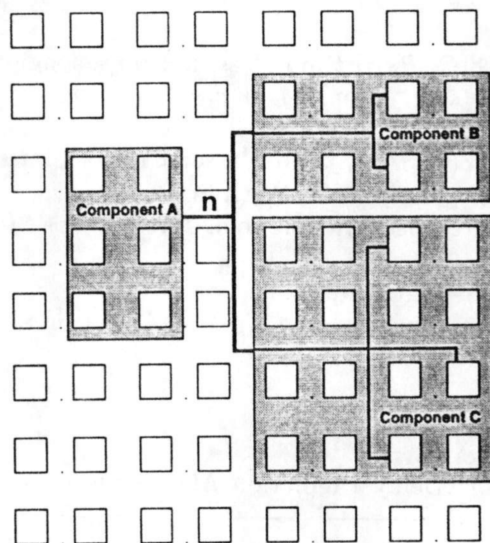

Figure 19: Fanout Effects

At the end of this step, we have a netlist which contains components' delay and the estimates of net delay.

4.3.3 Predict Clock Cycle

A typical timing model for digital systems is shown in Figure 20. The datapath part is composed of datapath logic blocks and the data registers. Data registers are used to store data inputs, outputs, and intermediate values in the data path. The controller can be implemented either as a Mealy or a Moore model. A Moore model is more widely used for high-speed synchronous systems and is also easier to synthesize automatically. Thus, our timing model assumes that the controller is implemented as a Moore Finite State Machine. A Moore controller consists of two combinational logic blocks: the next state logic and the output logic, one or more control registers store the current state information. The data path consists of combinational logic blocks (composed of functional units and muxes)

bounded by data registers [6].

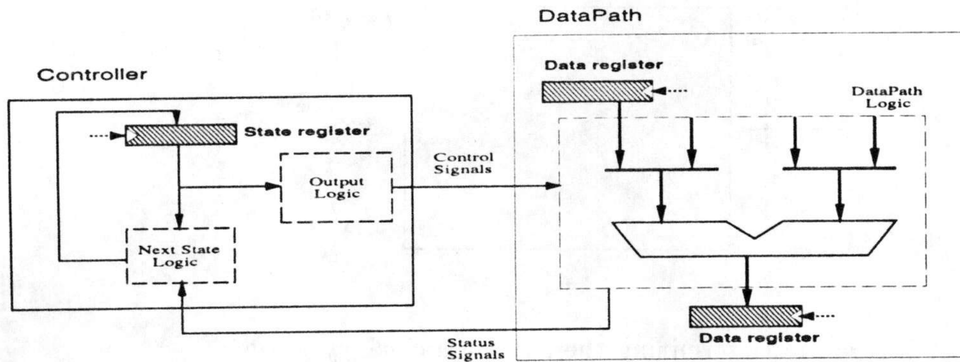

Figure 20: Typical Timing Model for a Digital System

Thus, the overall system can be modeled as a network of combinational logic blocks separated by registers. In this case, the worst case register-to- register delay is estimated and is output as a lower bound on the clock period for single phase clocking.

The total execution time of a design is given as the number of time steps times the clock period. The number of time steps is determined by scheduling and allocation and is known once the RT level design is generated. The minimum possible clock period is determined by the worst case register-to-register delay. Given the delay of a combinational block, we can determine the register-to-register delay between its input and output registers as shown in Figure 21. Let t_{cb} be the worst case delay through that block, and $t_{setup}(R_{in})$ be the propagation delay through the input register, and $t_{setup}(R_{out})$ be the setup time of the output register. The delay between R_{in} and R_{out} is estimated as

$$t_d(R_{in}, R_{out}) = t_p + t_{cb} + t_{setup}(R_{out}); \quad (1)$$

and the minimum possible clock period is estimated as:

$$t_{clock-min} = max_{\forall i,j|i \neq j} t_d(R_i, R_j); \quad (2)$$

Where R_i and R_j are assumed to be connected through a combinational logic block.

[6]This assumption, however, does not affect the validity of the overall approach since it is possible to substitute different timing models for other types of controllers should that be necessary.

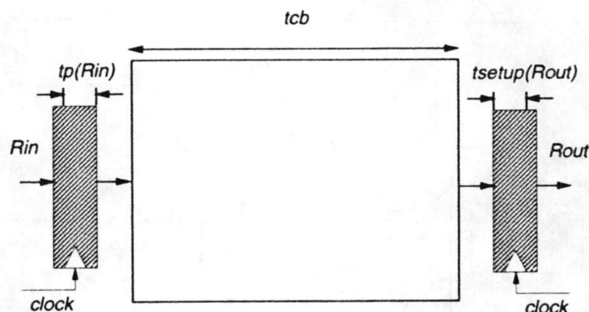

Figure 21: Determining the minimum clock cycle time

5 Experimental Results

5.1 Experimental Procedure

In order to benchmark the accuracy of our ChipEst-FPGA, we used six benchmark designs: (1) the AMD 2901 cpu with a bitwidth of 4, (2) RISC microprocessor Zot1—citeZot with 15 instructions and data path bitwidth is also 4, (3) The Differential Equation Example (HAL), (4) the Elliptic Filter[20] which with a bitwidth of 4 and 13 time steps. (5) and (6) are Fuzzy logic examples derived from [1]. Altogether, the RT-level implementations spanned a reasonably large set of design variation that are likely to be considered during high level design. The FPGA chips vary from XC4005 (with 12x12 CLBs) to XC4010 (with 20x20 CLBs).

All the RT-level implementations were written in VHDL. For components that can be pre-characterized, we can obtain their layout and timing information from the library. The layout and timing information for the remaining of components either by invoking our area and timing estimation described in [9] or by actually implementing the components. Since we are interested in benchmarking the chip level estimation procedure, we use component design whole procedure shown in [9] rather than run CompEst-FPGA to get the actual layout and timing information.

The components are implemented by different tools such as xilinx hard macro library, xblox and designware. Alternatively, use GENUS, a generic component generator to generate logic equation (EQN) according to desired functionality [11]. Then we use Synopsys do the optimization and synthesis. After synthesis, the component is translated to gate level xnf netlist and fed into xilinx ppr by giving different constraints to get different aspect

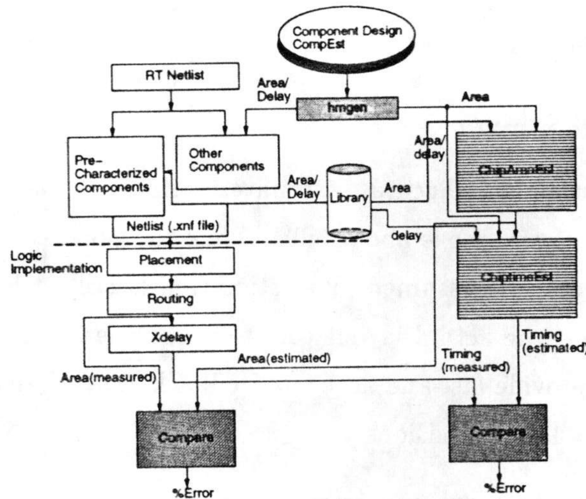

Figure 22: Experimental procedure

ratios. For each of those specific placement and routing, hmgen is invoked to generate a corresponding hard macro and followed by Xdelay to get their delay information.

The Chip level design is finished by instantiating components as hard macros with specific layout and timing information. Once we got the chip xnf files, again, they are fed into xilinx ppr and Xdelay are used to get the delay for the whole chip. Because of the non-deterministic nature of ppr, the designer tends to run ppr many times with different seeds and select the best one (in the experiments we ran, the worst delay varied from 4.4% to 20.9% percent off the best case in ten runs). To be fair, we also pick the layout with best performance to compare with our estimated results. In our experiments, ppr and Xdelay are run 10-20 times with different seeds and the best design is selected for comparison.

In order to assess the accuracy of our chip level estimation, we feed same RT level VHDL file into our ChipEst-FPGA to produce estimates of the chip area and delay using the models described in previous sections.

5.2 Results

The estimation results are shown in Figure 23. First, we note that our area estimates are very accurate. Our estimation accurately predicted the exact device type needed every time. For performance estimation, there was some differences between estimated and measured values. These differences can be attributed to the following factors:

- differences between estimated and final placements
- differences between routing rule assignment and final routing
- inaccuracies in the wiring delay model

Our ChipEst-FPGA can produce highly accurate estimates within very short run time. The average estimation error for performance is about 6.0%, while the worst case error is 18.7%. Even when one run of ppr/Xdelay is assumed, our estimation is still at least an order of magnitude faster to obtain that the actual layout process. This clearly indicates that our tool can be efficiently used to provide fast and accurate feedback to synthesis tools, allowing them to make better informed design decisions.

Benchmark design	Measured device (area CLB)	Estimated device	Measured IOs	Estimated IOs	Measured clock cycle (ns)	Estimated clock cycle (ns)	% error (cycle time)	Estimation run time(1) (s)	PPR/ Xdelay(2) (s)
AMD 2901	XC4006 (16x16)	XC4006 (16x16)	40	40	181.7	181.1	+0.4	1.8	8340
Zot1	XC4005 (14x14)	XC4005 (14x14)	51	51	99.5	118.1	+18.7	6	319
HAL	XC4006 (16x16)	XC4006 (16x16)	32	32	173.9	179.1	+3.0	3.8	327
EF19	XC4010 (20x20)	XC4010 (20x20)	66	66	489.6	482.2	-1.5	10.8	1835
Fuzzy1	XC4008 (18x18)	XC4008 (18x18)	76	76	286.6	272.1	-4.8	5.5	413
Fuzzy2	XC4006 (16x16)	XC4006 (16x16)	77	77	287.5	281.8	-2.0	5.2	566
Average % error							6.0		

(1) CPU run time.
(2) reported by ppr and xdelay.

Figure 23: Experiment Results

6 Conclusion

We presented a set of area and delay estimation techniques to support a hierarchical design model for Lookup Table Based FPGAs. The overall approach was benchmarked and found to be accurate. Future work will concentrate on linking the estimation model to synthesis so that better quality designs can be produced.

7 References

[1] D.D. Gajski, L. Ramachandran, P. Fung, S. Narayan and F. Vahid, "100-hour Design Cycle: A Test Case," *Proc. Euro DAC*, 1994

[2] M.D.F. Schlag, P.K. Chan, and J. Kong, "Empirical Evaluation of multilevel Logic Minimization Tools for a Field-Programmable Gate Array Technology", *Technical Report. University of California, Santa Cruz*, 1991.

[3] Xilinx, "XACT Development System: Libraries Guide," *Xilinx*, 1994.

[4] R.J. Francis, J. Rose, Z. Vranesic, "Chortle: A Technology Mapping Program for Lookup Table-Based Field-Programmable Gate Arrays," *Proc. 27th DAC*, June 1990.

[5] Xilinx, "XACT Xilinx Synopsys Interface FPGA User Guide," *Xilinx*, 1995.

[6] Robert J. Francis, "A Tutorial on Logic Synthesis for Lookup-Table Based FPGAs," *Proc. ICCAD 92*, 1992.

[7] D. Craig, M. Pontius, "The Zot1 Microprocessor implemented on an FPGA," *UCI course project report*, 1994.

[8] X. Chen and M. L. Bushnell, "A module are estimator for vlsi layouts," *Proc. 25th Design Automation Conf.*, pp. 54-59, IEEE/ACM, 1988.

[9] M. Xu, F.J. Kurdahi, "Area and Timing Estimation for Lookup Table Based FPGAs," *Technical Report #95-30, UCI*, Aug.1995.

[10] P.G. Paulin and J.P.Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASIC's," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 8, No. 6, pp.661-679, June 1989.

[11] P.K. Jha, T. Hadley and N.D. Dutt "The GENUS User Manual and C Programming Library," *Technical report.* No.93-32, April, 1993