EINDHOVEN
UNIVERSITY OF
TECHNOLOGY

Eindhoven University of Technology

MASTER

Architectures and placement algorithms for fine-grained reconfigurable computing

Poplavko, P.

Award date:
2001

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

TU/e technische universiteit eindhoven

Faculty of Electrical Engineering
Section Design Technology For Electronic Systems (ICS/ES)
ICS-ES 775

Master's Thesis

ARCHITECTURES AND PLACEMENT ALGORITHMS FOR FINE-GRAINED RECONFIGURABLE COMPUTING.

P. Poplavko

Coach: ir. K. Leijten-Nowak (Philips Research Laboratories)
Supervisor: prof.dr.ir. J.L. van Meerbergen
Date: June 2001

The Faculty of Electrical Engineering of the Eindhoven University of Technology does not accept any responsibility regarding the contents of Master's Theses

Abstract

This report is concerned with the development and modeling of fine-grained reconfigurable architectures. Under the 'fine-grained' architectures we actually understand the architectures similar to *Field-Programmable Gate Arrays* (FPGAs), e.g., to Xilinx or Altera FPGAs.

The reason of our interest in fine-grained architectures is that the computation density achieved by FPGA-based machines for some designs was proven to be one-two orders of magnitude greater than the computation density of programmable processors. Thus, fine-grained architectures can serve as acceleration units inside processors, i.e., they can serve for reconfigurable computing.

(Re-)configurable computing in general can be roughly defined as the use of acceleration units that offer a high degree of freedom in specification of the functions that are programmed in them, and use massive parallelism to speed-up the execution of those functions. In the context of reconfigurable computing, we pay most attention to the acceleration of digital signal processing (DSP) algorithms.

The method to explore the space of fine-grained architectures is based on running a set of benchmarks through the design flow based on VPR, the open-source tool developed at University of Toronto. That tool performs placement and routing targeting any architecture that fall into a parametrizable architecture template.

The contributions of this work are:

- extension of the routing architecture template by extra elements, namely, balanced connection box, half switch box and direct connections;
- modeling of a logic block developed by Ir. K. Leijten-Nowak;
- a framework for generating datapath-oriented netlists based on the new logic block;
- implementation of a bit-sliced placement algorithm;
- mapping a simple "application-specific unit" as a benchmark;
- comparison of random-logic-oriented and bit-sliced placement algorithms.

The work has been carried out at Philips Research Laboratories, Eindhoven, under the supervision of Prof. J. van Meerbergen and Ir. K. Leijten-Nowak.

Table of Contents

1. Introduction..1

2. Overview. 5

2.1 Introduction.. 5

2.2 Island-style architectures... 5

2.2.1 Island-style architecture.. 5

2.2.2 Configurable logic block... 7

2.2.3 Connection- and switch boxes. 9

2.3 Using VPR for Architecture Exploration 11

2.3.1 Introduction... 11

2.3.2 Layout design tools targeting FPGA................ 11

2.3.3 Routing architecture generation. 12

2.3.4 Cluster-based logic blocks.............................. 13

2.3.5 The routing-resource graph.............................. 14

2.3.6 Global architecture exploration........................ 15

2.3.7 Detailed routing architecture exploration.............. 17

2.4 Placement algorithms for FPGAs 18

2.4.1 Placement in FPGAs.. 18

2.4.2 Placement algorithms in general............................ 19

2.4.3 Placement algorithms for FPGAs........................... 19

2.4.4 Wirelength-based cost functions.......................... 20

2.4.5 Congestion-based cost functions........................... 21

2.4.6 Timing-based cost function.................................. 22

2.4.7 Placement heuristics... 22

2.4.8 Default placement algorithm in VPR...................... 23

2.5 Summary.. 25

2.6 Comments... 25

3. FPGA Architecture Exploration.. 26

3.1 Introduction.. 26

3.2 Modified connection box.. 26

3.2.1 Connection box generation........................ 26
3.2.2 Linear connection box generation.................... 27
3.2.3 Balanced connection box generation.................. 30
3.2.4 The comparison of two connection boxes.............. 32
3.3 'Half' switch box...................................... 32
3.4 Direct connections..................................... 34
3.5 New Logic Block.. 36
3.5.1 The structure of logic block 36
3.5.2 Logic block operation modes. 38
3.5.3 The routing models of *nlb*......................... 40
3.6 Summary.. 43
3.7 Comments... 44

4. Placement Algorithms. 45
4.1 Bit-sliced placement. 45
4.2 Related work... 46
4.3 Bipartitioning algorithm............................... 47
4.3.1 Bipartitioning for linear arrangement............... 47
4.3.2 Background.. 47
4.3.3 The specification of the algorithm.................. 49
4.4 Summary and cost function discussion 51
5. Benchmark and Results................................... 52
5.1 Direction detector benchmark........................... 52
5.2 Datapath mapping....................................... 55
5.3 Architecture specification............................. 55
5.4 Results.. 57
5.5 Summary.. 61
6. Conclusions... 62
6.1 Routing architecture elements.......................... 62
6.2 Modeling new logic block............................... 63
6.3 Datapath mapping and placement: 63

6.4 Proposals for the future work.. 63

Appendix 1. The Routing Area Estimation in VPR. 65

Appendix 2. Direct Connections for Random Logic. 66

Appendix 3. Architecture Specification 69

References 72

1. Introduction

As the title suggests, this Thesis is concerned with the development of fine-grained architectures and the placement of some components. Under the 'fine-grained' architectures we actually understand the architectures that are similar to the modern Field-Programmable Gate Arrays (FPGAs) and consist of the programmable elements that perform computation at the bit-level, e.g., Xilinx or Altera FPGA chips. In this work, we use the term 'FPGA' for any FPGA-like architectures, not only for commercial FPGAs.

The FPGA (Field-Programmable Gate Array) is an integrated circuit with a layout of the two-dimensional array of identical rectangular regions called *tiles*. Figure 1 shows a simplified view of an FPGA tile. The tile contains a *(configurable) logic block* (*clb*) and *(configurable) routing interconnect*. The logic block computes one or several bit-level logic functions on the data arriving at its pins and the configurable interconnect provides communication with other FPGA tiles. Different parts of the tile are joined by switches.

An FPGA is a single hardware platform for implementation of various logic circuits. The functions performed by the *clb*s and the routing paths through the interconnect are defined by the contents of the *configuration memory* (not shown in Figure 1). To be implemented in an FPGA, the logic circuit specification has to be

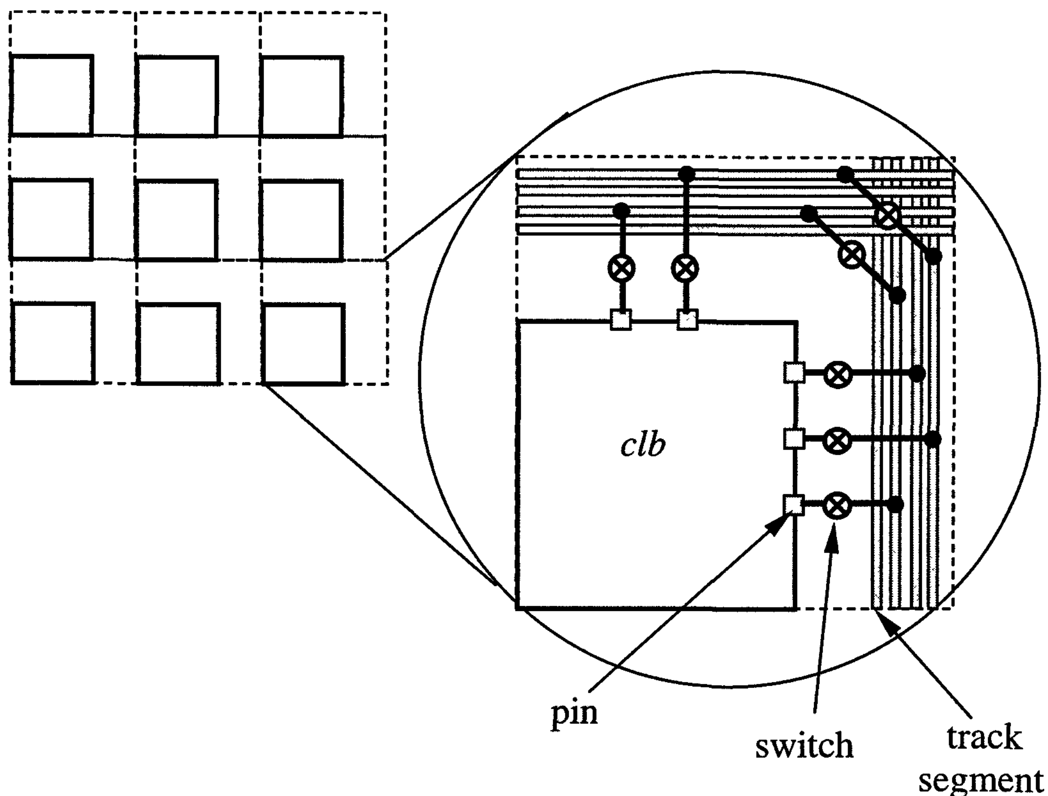


Figure 1 An FPGA tile

converted into a configuration bit stream. Different bits of the stream determine the state ('ON/OFF') of the routing switches. They also define the truth tables of the logic functions performed by *clb*s. Once the configuration stream is loaded to the configuration memory, FPGA has the functionality of the implemented logic circuit and this circuit is ready for operation.

Thus, FPGAs, being used for logic circuit implementation, are electrical circuits themselves. As a result, the terms '*circuit*' and '*circuit design*' in the context of FPGAs may cause ambiguity. From now on, we will use them only for the logic circuit being implemented, or mapped, in an FPGA. For FPGA itself, the terms '*architecture*', '*hardware*' and '*architecture development*' will be used.

Since their introduction, FPGAs have been used as hardware platforms for 'glue logic', e.g. circuits that perform some secondary intermediate function. Later, FPGAs have been used for more complex designs, such as controllers. In the cases mentioned above, circuits to be implemented are irregular and the netlist connections are apparently random. Such circuits are referred to as *random logic*. The design of random-logic circuits targeting FPGAs is a general and well-studied subject.

FPGAs have also been used for several years in rapid prototyping systems for verification of complex digital designs. In such systems, an FPGA chip is placed into the hardware environment and the digital design is mapped onto the chip for in-system functional verification [Stoh98].

Nowadays, FPGAs are considered to serve as an extension or alternative to programmable processors, because, just like processors, they allow to reuse the same piece of silicon for many applications. Note, that both the terms '*programmable*' and '*(re-)configurable*' can be applied in the domain of FPGAs, whereas when we speak about programmable processors only the term 'programmable' can be applied, referring to the execution of software by processors.

As it has been proven in practice, an FPGA is a more powerful implementation platform than a processor made in the same technology. This comes because the applications mapped to FPGAs can make use of multiple active computing elements, rather than sequentially reusing a small number of computing elements, as the programmable processors do. Moreover, when being mapped onto an FPGA, a digital circuit can use the on-chip hardware more effectively. For example, due to the bit-level programmability of FPGAs, the datapath operations can be performed on as many bits as it is really required by the design, whereas programmable processors have fixed datapath widths.

The performance achieved by configurable machines for some designs has been proven to be one-two orders of magnitude greater than the performance of processor [DeHon00]. E.g., the fastest RSA (Rivets-Shamir-Adelman) decryption rate of any machine has been achieved by an FPGA-like machine developed at Informatics and Automation Research Institute, Paris [DeHon00]. DeHon also shows that

computational density that an FPGA can achieve is one-two orders of magnitude higher than that of a programmable processor. He measured the computational density in the number of alu bit operations per λ^2·sec, where λ^2·sec is the minimum transistor area multiplied by second (area-delay product).

The case when an FPGA rivals or complements a programmable processor is referred to as *(re-)configurable computing*. We are interested, first of all, in the fact that they have proven to be good platforms for DSP applications. FPGA vendors pay much attention to that fact. Unlike glue-logic and controllers, DSP circuits possess much regularity, e.g., many of them are based on the full adders, that are basic elements for summation, multiplication, etc. We will call such circuits *datapath-oriented circuits*, as opposed to *random logic circuits*. FPGAs are already provided with additional features for high-performance implementation of the datapath-oriented circuits. FPGA vendors also provide automatic core generators and libraries for typical DSP components, like various shift registers, adders, multipliers or even whole FFT implementations [Xilinx]. However, the competitive implementation of arbitrary DSP algorithm on FPGA still requires that the designer be familiar with the architecture of FPGA and special coding styles of the synthesis tools.

This situation is very similar to the situation with mapping DSP applications to the programmable DSP processors. An important difference, however, is that, in the case of processors, the recent advances in the high-level synthesis and VLIW architectures have established a good foundation for effective and fast implementation of DSP algorithms starting from the high-level specification e.g. in "C". For the case of FPGA as the target, an effective "C"-compilation is an important and difficult research topic.

For example, in University of California, Berkley, an FPGA architecture aimed at datapath implementation was developed, called "Garp". Timothy Callahan has developed "C"-language compiler ("Gamma") for that architecture [Call99b]. The compiler mapped, placed and routed the datapaths, extracted from general-purpose "C"-language code [Call99a]. The benchmarks run by the UC-Berkeley Gamma/Garp project show that an FPGA-like architecture can extract more instruction-level parallelism out of some "C"-language applications than a VLIW or superscalar processor can exploit. Furthermore, in 2000, Synopsis Inc. has undertaken research effort aimed at "C"-language compilation for FPGA [Goer00a] [Goer00b]. That project has made extensive use of the work done by Callahan.

An advantage of programmable DSP processor is that it can run applications of virtually unlimited sizes and diversity. In FPGAs, an application can run only if it totally fits in FPGA. When the part of application present in FPGA has completed its work, the next part should be loaded. To reload an FPGA with new configuration would take a considerable time, which would likely lead to the violation of application's timing constraints. So, current FPGAs can serve to execute the parts of applications that are often reused (e.g. loop iteration), so that it is not needed to reconfigure FPGA at the run-time too often.

In processor technology, the parts of applications that are often reused are implemented in hardware as application-specific functional units on which some operations execute much faster than in software. A problem with such units is that they are not universal, because a different application is likely to require a different unit. FPGAs lend themselves to be serve as a remedy for this problem. A considerable amount of research is done in this direction. For example, the "Garp" architecture, already mentioned above, was designed as an acceleration unit of MIPS processor.

Our long-term goal is the development of a fine-grained reconfigurable acceleration unit for a programmable DSP processor. The framework that we use for the investigations in the architecture domain is based on running a set of benchmark circuits through a design flow targeting the FPGA architecture being evaluated. So, we can distinguish three main components required for the architecture development, such as: *the FPGA architectures*, *the design flow*, and *the benchmarks*. We have worked on all three components just mentioned, using a CAD tool, VPR, as a foundation. VPR version 4.3 has been developed by Betz et al at the University of Toronto. VPR stands for Versatile Placement and Routing. It is remarkable, that VPR is open-source and "retargetable", i.e., one can target various of FPGA architectures that fall into a certain architecture template.

The highest priority in our research has been assigned to the support of datapath-oriented circuits, because they represent the DSP applications. By default, VPR could only treat datapath-oriented circuits as if they were irregular circuits. For that reason, our main goal was to extend VPR's source code in such a way, that VPR would make use of the regularity of the datapaths and of the extra hardware included in FPGAs to support datapath operations.

The structure of this Thesis is as follows. In Chapter 2, we make an introduction into the architectures of FPGAs and the design flow targeting FPGAs. In the next three chapters we go through three main issues of architecture development already mentioned above. The first issue, *FPGA architecture,* is dealt with in Chapter 3. There, we introduce some interconnect elements into VPR's architecture model, because we believe they are of interest for the future fine-grained reconfigurable architectures. Also, we consider a particular *clb* which is supposed to be the computing element of such architectures. We discuss the question how the datapath-supporting features of that *clb* should be taken into account in VPR. The second issue, the *design flow*, is dealt with in Chapter 4, where we show how the regularity of datapaths can be exploited in the placement stage of the flow. The third issue, *benchmarks*, is considered in Chapter 5, where we introduce an initial framework for the implementation of datapath-oriented benchmarks using VPR. Having discussed all three components of the framework, we put the framework into work by performing several experiments, which we also present in Chapter 5. Chapter 6 contains the conclusions and suggestions for the future work.

2. Overview

2.1 Introduction

A complete overview of the research results in the domain of FPGAs would be rather a lengthy story. Two main domains would have to be considered, namely, FPGA architectures and FPGA design flow. Both of them include a great variety of subjects.

The main FPGA architectural issues are the type of routing interconnect (*island-style*, *row-based* and *hierarchical*), the architecture of the logic block, and embedded memories.

As for design flow issues, they include *logic synthesis* that translates the high-level specification into a structure based on logic primitives, *technology mapping* that packs the logic primitives into *clbs*-based building blocks, and *physical layout design.*

In this chapter, we consider both architectures and CAD. In this overview, we emphasize the sub-domain supported by VPR or the issues important for our research topic. As a result, in the domain of architectures, we look only at island-style architectures, and, in the domain of tools, we consider physical layout tools. To be more specific, we focus on *placement* tools, because placement is the first physical layout design stage, and, therefore, it has the greatest impact on the physical layout.

2.2 Island-style architectures

2.2.1 Island-style architecture

The island-style FPGAs architectures are the most widely-used architectures in FPGA industry. Perhaps, the most prominent examples of island-style architectures are Xilinx FPGAs, such as XC4000 and Virtex series. Other examples are Lucent, Atmel and Vantis FPGAs.

A fragment of an island-style FPGA architecture is depicted in Figure 2.1. This fragment is taken near the top edge of the array. The rest of the FPGA resources lies below, to the left and to the right of the fragment. In this example, we see four input/output pads (*I/O pads*), two per column, that provide communication of the array with the external world. Just below the pads, the horizontal *I/O channel* is located. I/O pads are located at all four edges of the FPGA. Four I/O channels constitute an I/O channel ring around the array and the pads use them to communicate to the internal parts of the FPGA.

Configurable logic blocks are separated by *horizontal* and *vertical routing channels*. A routing channel consists of *wire segments*. The wire segments are aligned to *tracks*, i.e., to the parallel geometrical lines running along the channels. The number of tracks in a routing channel is called *channel width*, denoted by symbol W. In the presented example, all channels have W=4.

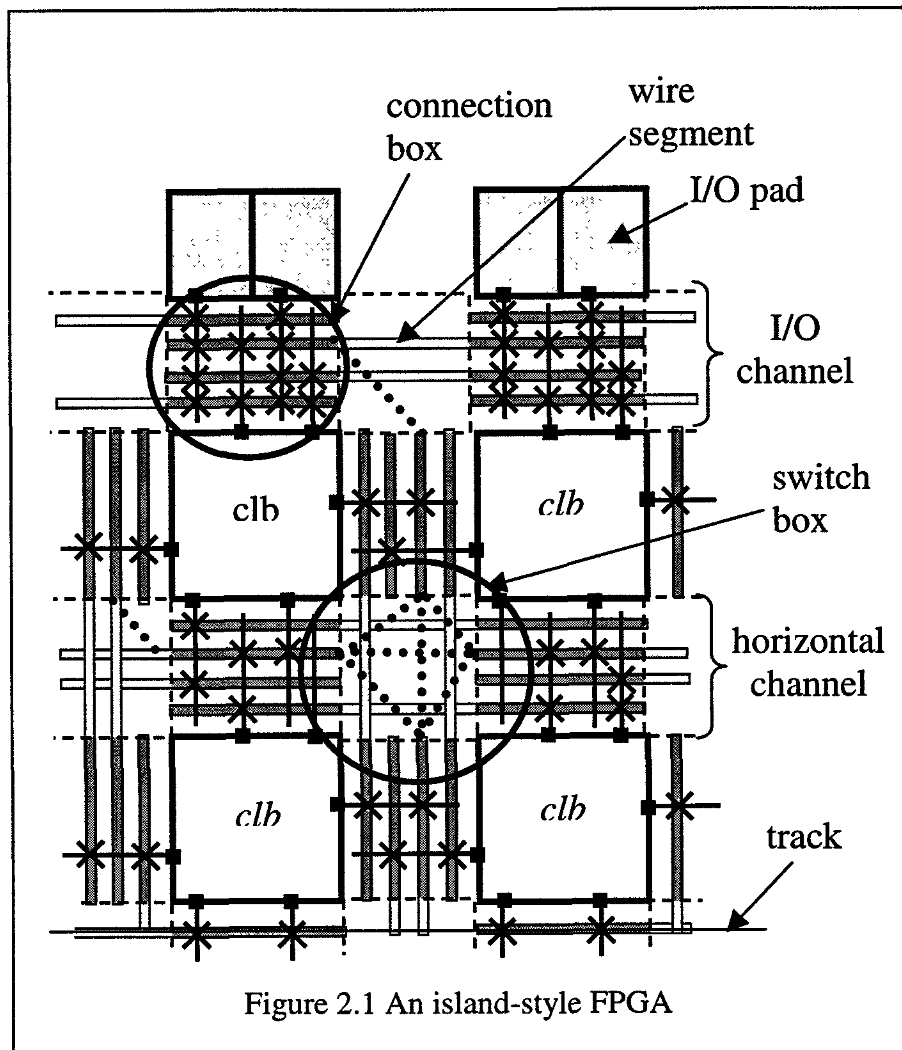


Figure 2.1 An island-style FPGA

Wire segments have limited *lengths*. The length of a wire segment is the number *clb*s that that segment spans. The *nominal length* of every wire segment in this example is 2, but some vertical segments that lie close to the top edge are truncated to the length 1 because they do not fit into the FPGA.

Configurable logic blocks and I/O pads are connected to the channels via *connection boxes* that join logic block *pin*s to the wire segments. A pin can have only one direction: *input* or *output*. Connection boxes consist of *connection switches*, shown in Figure 2.1 with 'crosses'. The connection switches are *programmable*, i.e., they can be configured to active state (when they pass signals) or passive state (when they do not pass the signal).

At the crossings of vertical and horizontal channels the *switch boxes* are located. A switch block consists of *routing switches*. These switches are programmable too. They join wire segments of horizontal and vertical channels, enabling bends in the routing. A wire segment may stop at the switch block or cross it. If it stops there, it can be connected to wire segments of the same channel at the other side of the switch block. This allows making connections longer than the length of the wire segments.

To simplify Figure 2.1, we have shown only a few routing switches. They are depicted with thick dotted lines.

The FPGA architecture in general can be divided into two main parts: configurable logic block architecture and routing architecture. In the following subsections we first consider logic block architecture and then examine routing architecture elements, namely, connection and switch blocks, and routing channels.

2.2.2 Configurable logic block

In application-specific (and, thus, non-programmable) integrated circuits (ASICs) the logic primitives are usually implemented on silicon with *logic cells*, the small pieces of hardware with fixed functionality, like 'AND', 'OR', 'D flip-flop', etc. These cells are organized in a library and they are different from one another. The equivalent of a logic cell in FPGA is a *configurable logic block* (*clb*). In contrast to logic cells, *clb*s are identical to one another. Many elements of ASIC logic cell libraries can be implemented using one or several *clb*s.

Configurable logic block has input and output pins and normally contains look-up tables (LUTs), flip-flops, local programmable interconnect (based on multiplexers, buffers and SRAM cells) and dedicated logic for arithmetic operations.

The most important basic element of *clb* is a *look-up table* (LUT). An N-input look-up table is a configurable element that is used to implement any logic function of N inputs. The particular function implemented is defined by a truth table stored in LUT's memory. Thus, it can be an arbitrary function.

An example of simple *clb* structure is shown in Figure 2.2a. It contains a 4-input LUT, a D-flip-flop (shown with black rectangle) and a multiplexer. The input pins are shown with gray boxes and the output pin is shown with a triangle. The multiplexer can be configured to pass the direct or buffered output of a LUT. The SRAM cell that determines the input selection is not shown. Note, that, from now on, a multiplexer shown in our figures without selection port will denote a multiplexer driven by SRAM cells.

A circuit containing a LUT, multiplexer and D-flip-flop connected that way is called *basic logic element* (BLE), [Betz99 p. 38]. Thus, the *clb* architecture shown in Figure 2.2a is one-BLE-based.

The greater the number of inputs a LUT has, the greater the combinational part of logic circuit that can be implemented in one LUT, and, as a consequence, the smaller the number of *clb*s required to implement a logic circuit. This means that less FPGA tiles are used. However, on the other hand, the area of LUT increases, roughly, by a factor of two with every new additional input port because of the growth of the truth table. Also the channel width, W, must be increased because the number of the logic block pins increases and more signals have to be routed. Rose et al [Rose90] has

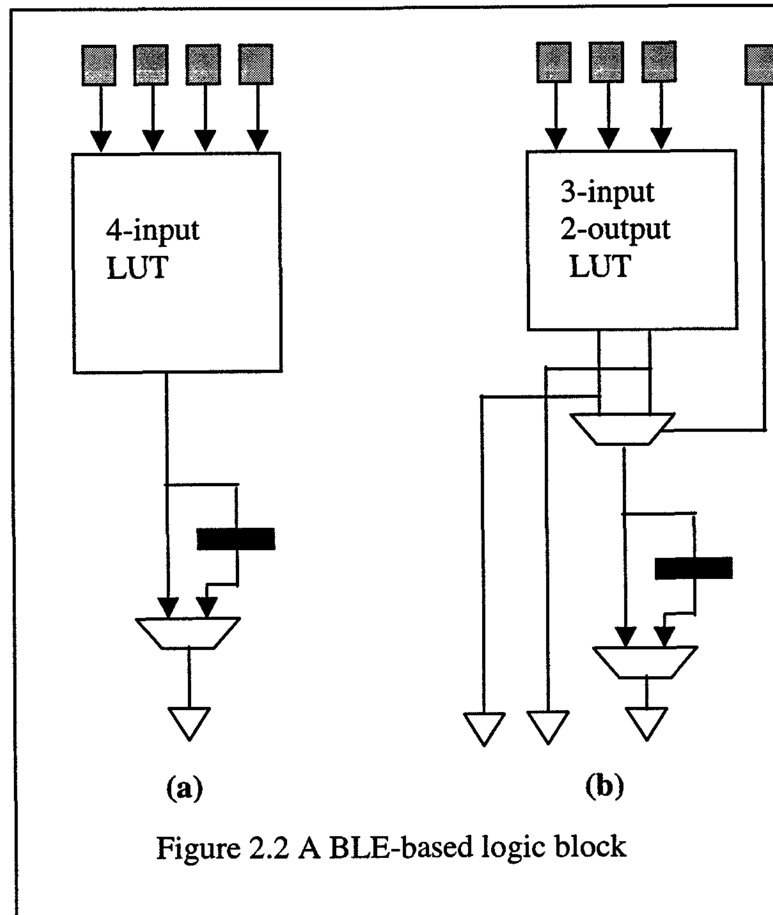


Figure 2.2 A BLE-based logic block

considered the factors mentioned above and shown by experiments on several random logic benchmarks that the use of 3- or 4-input LUTs leads to the greatest area-efficiency. This LUT types are commonly used in the commercial FPGAs.

Let us define an *N*-input *K*-output LUT as a unit that contains *K* LUTs with common inputs. One important property that is exploited in FPGAs is the possibility to combine two LUTs with common inputs into a larger LUT. This property is a direct consequence of Shannon expansion property of Boolean functions [DeMicheli].

'*N*+1' LUT Property. An *N*-input 2-output LUT and a 2:1 multiplexer can be used to implement an *N*+1-input LUT. To do that, 2 outputs of *N*-input LUT should be fed as two multiplexer inputs, and the extra input should go to the multiplexer's selection port.

Figure 2.2b shows how the 4-input LUT of the BLE can be implemented using a 2-output 3-input LUT. The outputs of the LUT can go to separate output pins. The advantage of this implementation is that one BLE can implement in this case *two* Boolean functions, instead of one, if the inputs are shared. This architecture supports the implementation of the full adder, three shared inputs being used for the operands and the carry-in, and two outputs being used for the sum and the carry-out. Actually, Figure 2.2b shows a simplified logic block used in Atmel AT40K architecture.

In most commercial FPGAs, several LUTs and local interconnect between them are included in one *clb*. For example, in Virtex FPGAs proposed by Xilinx, one logic block contains four 4-input LUTs. Betz et al [Betz99] used VPR to investigate the best sizes for so-called *cluster-based logic blocks*, measured in the number of BLEs they contain. In that research, they used the detailed timing and area models built into VPR, and they took into account the adjustment of transistor sizing to different sizes of the logic block. They have shown that the best area and delay efficiency is obtained with 4 BLEs per block. The cluster logic block will be considered in more detail later in this chapter.

2.2.3 Connection- and switch boxes.

Both connection and switch boxes are collections of programmable switches available in the routing architecture and activated when they lie on the paths routed from a driver output pin of some logic block to the input pins of the receiver logic blocks. First, let us consider switch boxes.

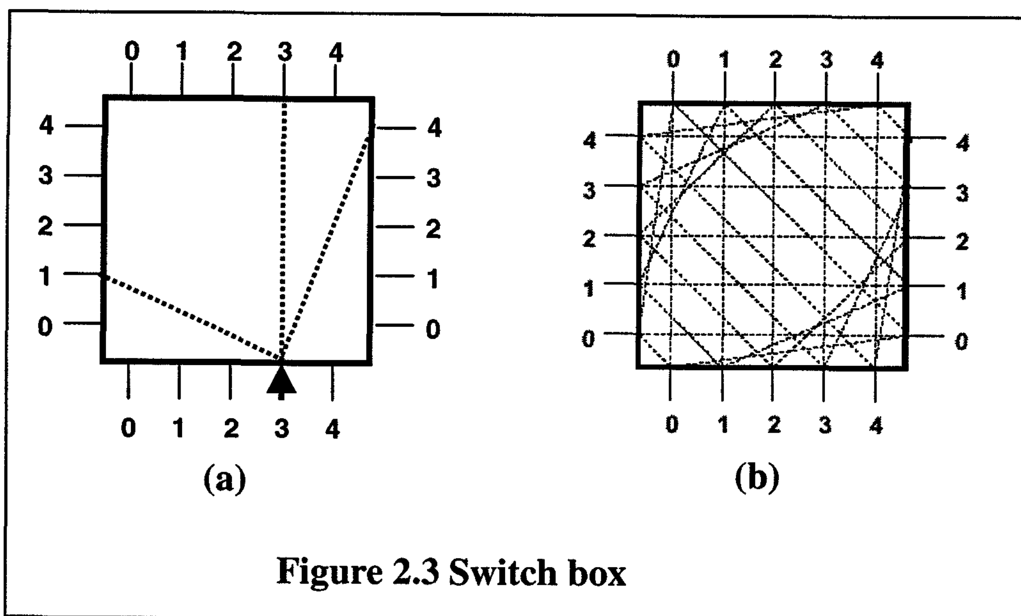


Figure 2.3 Switch box

Consider the switch box shown in Figure 2.3a. Recall that switch boxes lie on the crossings of horizontal and vertical channels. The points where channel tracks enter the switch box (called switch box pins, not to be confused with the logic block pins) are marked in Figure 2.3 with the *track numbers*. The dotted lines show the *topology of the switch box* with respect to track 3, bottom side. The topology defines the *potential connections* of the switch box pins. A potential connection indicates the pair of pins to be connected by a routing switch, if a wire segment that crosses the switchbox does not already join those two points. For example (see Figure 2.3a), if at track 3 a vertical wire segment crosses the switchbox, no vertical switch is needed.

As it can be seen in Figure 2.3a, the potential connections of the pin at track 3, bottom side, do not join that pin to every other pin, but only to three of them. The

number of potential connections of a switch box pin is called *flexibility* (denoted as F_S) of the switch box, so, in that example, $F_S = 3$. The previous research results have shown that the option $F_S = 3$ for every track on every side is a good choice from the area-efficiency point of view [Betz99, p.15]. It is also the flexibility of the switch box used in the Xilinx' XC4000 architecture and some other architectures.

Several switch block topologies with $F_S = 3$ have been proposed (see [Imran99] and [Betz99, p.16]). In this work, we use only the *disjoint switch box.* An example is shown in Figure 2.3b. This topology connects all tracks with the same track number. This means that the tracks of the routing channels are *disjoint*, i.e., once routed to a particular track, a signal will follow tracks of that particular number in every channel where it goes.

Now let us consider connection box. Such a box lies on a routing channel between two adjacent logic blocks or a logic block and an I/O pad (see Figure 2.1). The connection switches join the logic block and I/O pad pins to the routing channel. A part of the connection box that belongs to one *clb* pin is shown in Figure 2.4a. In this example the channel is horizontal, and the switches are present only on some tracks, on the other tracks they are missing. The topology of the connection box, or the *(connection) switch pattern*, specifies for every pin the tracks to which the pin is connected. The same switch pattern is replicated for the given pin at every logic block and I/O pad.

Let us define the *(relative) flexibility* of the connection box as the number of switches that connect one pin to the tracks divided by the number of tracks, *W*. Flexibility is denoted by F_C. For example, in Figure 2.4a, $F_C = 0.5$.

In Figure 2.4b, the simplest implementation of the pin switch pattern is shown. Buffers that amplify the signals are not shown. It requires 4 SRAM cells to control the state of the switches. If any two switches are active, a so-called *dogleg* joins two wire segments. An opportunity to use doglegs (when the pin is not used or carries the same signal) would increase the flexibility of routing.

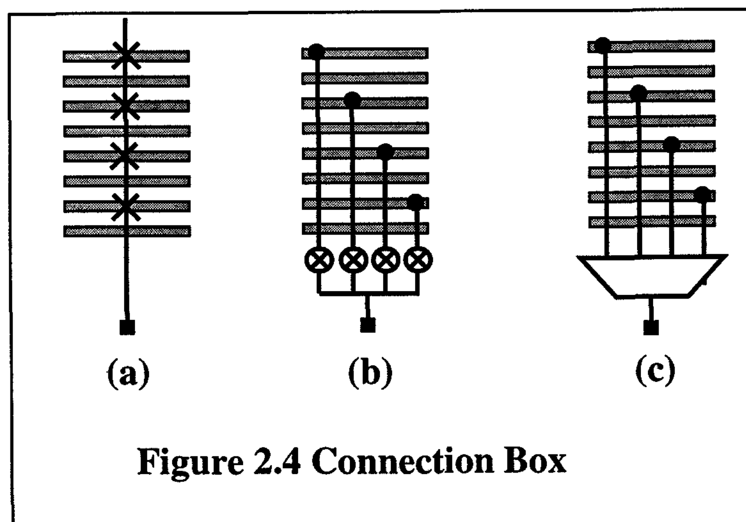


Figure 2.4 Connection Box

Actually, the 'dogleg-style' is used in VPR only for output pins. It allows the output signal to be sent to several tracks simultaneously. For input pin the 'dogleg-style' is not used. One of the reasons, perhaps, is because that style would require more SRAM cells then the 'multiplexer-based' implementation, illustrated in Figure 2.4c. Only 2 SRAM cells are needed in this example for the connection to the pin, because two bits suffice to encode one single connection of the four possible. For more details about connection box implementation, see [Chow99] and [Betz99, Appendix B].

2.3 Using VPR for Architecture Exploration

2.3.1 Introduction

In this section, we consider the architecture exploration framework used in the previous advanced research of FPGAs conducted by Betz et al using VPR [Betz99]. To explore the space of architectures, one has to use some criteria to evaluate the decisions made about architectural parameters. Betz et al evaluated the architectures based on average results experiments with 20 MCNC benchmarks. The largest one consists of approximately 8300 LUTs.

Given an architecture that we want to evaluate and which falls into VPR's architecture template, VPR can perform physical layout design for every benchmark, and report the minimal channel width which the architecture should have to successfully place and route the circuit. Also, the switch area and critical path are reported. The area is measured in minimum-size transistors and the critical path (Elmore) delay in nanoseconds. The physical parameters of the routing switches and wiring segments are provided as an input to VPR. Betz et al optimized them for 0.35μm process, and provided a sample file containing those parameters, so that other researchers could use them in their work.

We start with a brief introduction to the physical design flow, to give a feeling what tools have been used to map benchmarks to architectures during the architecture exploration. In the rest of the section, we concentrate on architecture development issues.

2.3.2 Layout Design Tools Targeting FPGA

The layout design of digital circuits targeting FPGAs as implementation platform consists, in general, of the same stages as in the classical design flow for VLSI. The layout design is the last stage of circuit design. It starts after the logic synthesis and technology mapping are completed.

The technology-mapping tool provides circuit's logic-block-level *netlist*, which is the input for the layout design tools. The basic elements of a netlist are *netlist blocks*

and *nets*. Netlist blocks are parts of the circuit mapped to *clb*s and the ports of the circuit mapped to I/O pads.

The first stage of the physical design is *placement*. In this stage, a particular physical location *(x,y)* for every netlist block in the array is selected. We come back to placement in Section 2.4

The placement is followed by *global routing* (see Figure 2.1 again). At that stage, routing paths along the wire segments and programmable switches are found for every net. These paths together constitute a tree-like structure. They start from the output pin of the block (or I/O pad) that sends the signal and finish at input pins of the receivers. Note, that the pins used by the router can be other than defined in the netlist, because *clb* architecture allows some groups of pins (*equivalent pins*) to exchange their roles. The global router makes use of that feature to route nets to the most convenient pin in terms of the physical location, sparing the extra tracks required to route them to their precise destinations. The paths of use certain vertical and horizontal channels. It makes no difference, at this stage, which particular wire segments and switches are used, only which particular channels, switchboxes and connection boxes lie in the paths. The attention is paid that at every channel location no more tracks are used than are available in the channel.

To complete the layout, one has to define the details of the routing paths, taking into account the fine-grained structure of the channels and switchboxes. This stage is called *detailed routing*. It is the last design stage. When it is completed, the final versions of truth tables and the required states of every programmable switch and programmable multiplexer is known; so, the configuration string can be generated.

VPR can perform both placement and routing. Two types of routing are implemented in VPR, namely, the global routing and combined global-detailed routing.

2.3.3 Routing Architecture Generation.

No doubt, that, in order to obtain a robust and competitive FPGA architecture, the FPGA tile details should, at the final stage, be carefully tuned and perfected by the humans. However, at the initial stage of architecture development, an enormously large space of architectures has to be explored to figure out the optimal basic characteristics the architecture should possess. So, at this stage, scrupulous architectural elaboration is not justified. For that reason, Betz et al have developed a framework for architecture exploration and a key part of that framework is the automatic *routing architecture generator* built in VPR [Betz99, p.70].

The routing architecture generator takes as an input a set of parameters listed in *architecture specification file*. These parameters are easy to understand and they directly influence the quality of the architecture. For example, the length of the wire

segments is such a parameter. It directly influences the delay of the routing interconnect. Taking the segment length as an input, the routing architecture generator must produce a pattern of wire segments that yields good routing channels. Look at the I/O channel in Figure 2.1. The segments on different tracks in that channel start at different points, they are not all aligned to one another. Intuitively, such a location of wire segments is good for routing.

The routing architecture has to match the architecture of a logic block. The parameters of logic block pins are listed in VPR's architecture specification file. Although the greatest part of the delay and area in FPGAs is determined by the general routing interconnect[1], the logic block architecture also considerably influences the area and performance of an FPGA as a whole. As we will see later in this section, Betz et al has proven by experiments that the right balance should be found between logic block and routing interconnect in terms of the area and delay contribution.

2.3.4 Cluster-based logic blocks

The cluster-based logic blocks, or logic clusters, were introduced for the VPR-based architecture exploration. The structure of such a block is shown in Figure 2.5. The elements of the block are N basic logic elements (see Figure 2.2), a so-called *interconnect matrix*, I input pins and N output pins [Betz99 p.38]. The number of logic elements, N, is called the *size* of the logic cluster.

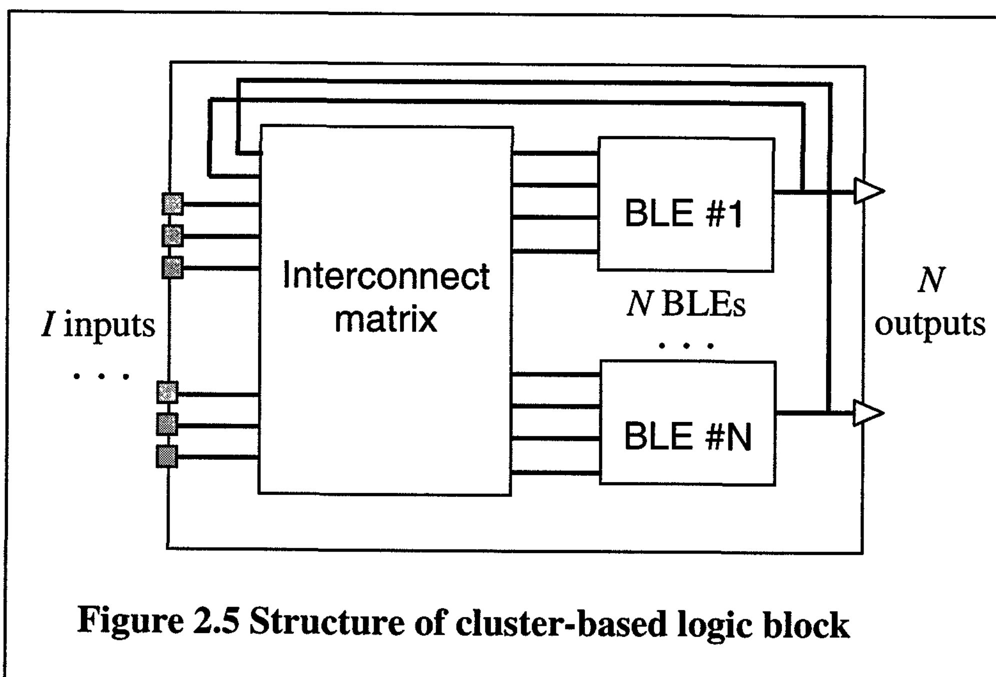


Figure 2.5 Structure of cluster-based logic block

The interconnect matrix can be seen as another level of routing architecture, namely, the *local routing architecture*. The input of the matrix is the block's input

pins and N feedback loops from the output pins. This matrix can connect any of its $N + I$ inputs to any BLE input, so, it is *fully connected.*

The full connectivity simplifies the task of the routing algorithms. This is because it provides the routing tool the greatest freedom for the assignment of the signals to logic block pins. All input pins of the logic cluster are *equivalent* and the same holds for the output pins.

The price paid for this flexibility is an extra area and delay of a logic cluster. The greater the cluster's size, the greater is the area and delay of the local routing. On the other hand, the more connections can be routed locally reducing their delay and the demand for the external routing.

Imran Masud [Imran_MSc] has shown that modified logic clusters with a reduced local interconnect can lead to a significant speed-up (up to 30%) and area reduction (up to 56%) of the logic block. Taking into account that the number of logic blocks required to map the circuit remained almost the same, the overall area reduction of the logic-block portion of the routing architecture was also around 50%. However, because the pin equivalence has been sacrificed, no efficient routing model could be built for VPR. With inefficient routing, the penalty of the routing delay and routing switch area has been more considerable than the profit from a modified logic block.

Since the new logic block we investigate in Chapter 3 also allows only limited pin equivalence, this is somewhat related to our work.

2.3.5 The routing-resource graph

Internally, FPGA architecture is represented by a directed graph, called the routing-resource graph [Betz99 p.68]. It is actually the output produced by the routing architecture generator.

The vertices of that graph represent, first of all, the elements that always pass the signal, namely: wire segments and logic block pins. The edges of the graph represent, first of all, the programmable switches. In addition to pin nodes, logic blocks are represented in the routing graph by special vertices called *sources* and *sinks*. Sources represent the initial points of the routing paths in the graph. Special edges go out from the sources to the output pin nodes. If several output pins are equivalent, the special edges join the same source to them. Sinks are the final points where the routing paths terminate. Special edges come into the sinks from the input pin nodes. If several input pins are equivalent, the special edges join them the same sink.

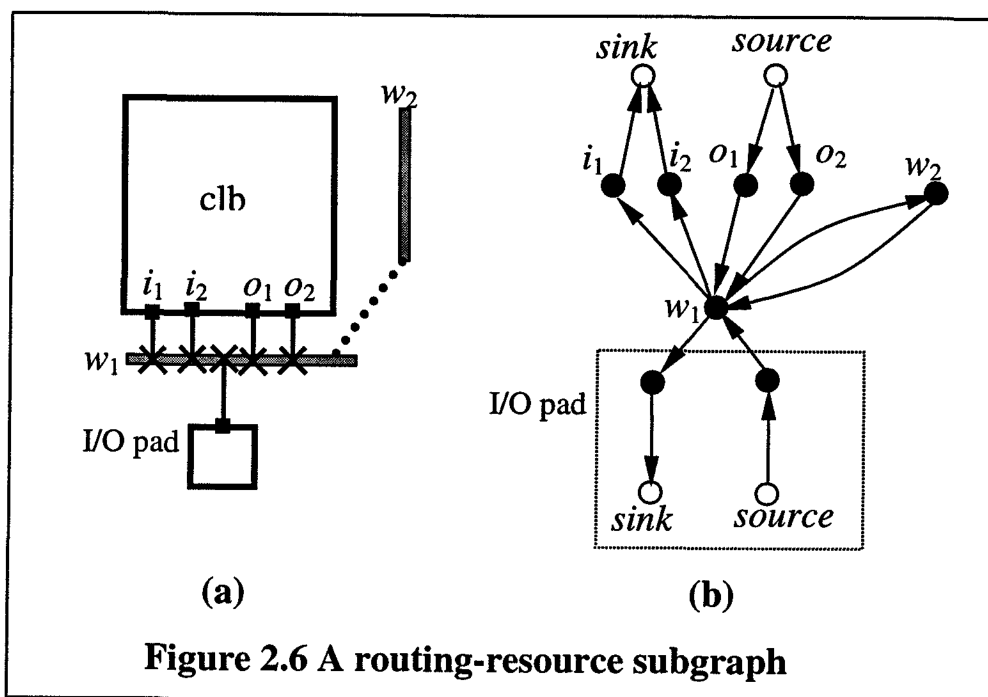


Figure 2.6 A routing-resource subgraph

Figure 2.6a shows a simple architecture fragment. It consists of a logic block, an I/O pad, two wire segments and several switches. The corresponding sub-graph is shown in Figure 2.6b. The input pins i_1 and i_2 are equivalent. So, they are joined to the same sink. To arrive to that sink, a net can select any of those pins. The output pins o_1 and o_2 are equivalent too.

The routing in VPR is based on the Dijkstra shortest-path algorithm. Every net is assigned a source and several sinks, and this assignment is determined by the placement of the net's terminals in the FPGA array. Every vertex has a weight, called *capacity*. It specifies the number of routing paths that can share that vertex. Normally, only a sink or a source can have capacity more than one, because they gather all input or output nets of the block.

2.3.6 Global architecture exploration

Recall, that VPR can perform either global or combined global-detailed routing. Some architecture parameters, called global architecture parameters, are required only for the global routing. Several basic global architecture parameters are variable. They can be specified separately or VPR will find the minimum values for them. These parameters are: the dimensions of FPGA, n_X and n_Y, and *the (reference) channel width W*. The reference channel width is a channel width value to which the widths of all channels in the architecture are proportional. To find the minimal (reference) channel width, W_{min}, a binary search procedure is run that performs multiple routings at different channel widths until the minimum is found.

The other *global architecture parameters* are:

1) Logic block pin specification.

2) *I/O-pad ratio* (number of I/O pads per row/column);

3) The *distribution of channel widths* in FPGA

Let us consider *clb* pin specification. Every pin has a direction (in/out), and is located on one or more *clb* sides. An important parameter is the class number of the pin. All pins are divided into equivalence classes and the class number defines to which class the pin belongs. As for the logic cluster, its pins are divided into two classes: one for the input pins and the other one for the output pins. The number of output pins is equal to the number of LUTs per block (cluster size, N). The question is, what N leads to the best architectures?

The increase of logic block size leads to the increase of the complexity of the interconnect matrix (local routing), so the contribution of logic block to the area and delay increases. More connections are routed via (a faster) interconnect matrix and less connections use the routing interconnect resources, so the delay and the required routing area decreases. However, at some point of logic block growth, the routing area will stop to decrease, because the tools will no longer be able to turn more and more inter-block connections into intra-block connections [Betz99 p.146], probably because the remaining inter-block connections will become too long. Moreover, because of the increase of physical lengths of wires, one has to scale-up the sizes if transistors in routing switches, as the size of the logic block increases. Using VPR and MCNC benchmarks, Betz et al has come to the conclusion that the size $N = 4$ appears to yield FPGAs with reasonable performance and area compared to the other sizes, and it has been chosen as a logic block for the architecture research. Thus, the number of output pins is 4. The total number of BLE input ports inside that block is 16, but, because many logic functions share inputs, one can save on 6 inputs and use just $I = 10$ logic block input pins.

Every pin can be located on one or more sides ('top', 'left', 'right' and 'bottom'). One can distribute the logic block pins between the sides uniformly, which yields 1 output and 2-3 input pins per side. From now on, we assume, by default, that the pins are distributed uniformly. One can also put more pins to, e.g., horizontal sides, increasing the utilization of horizontal channels w.r.t. vertical channels.

In contrast to logic block pins, the horizontal and vertical densities of I/O pads have been restricted to be equal. In VPR, the number I/O pads per column and per row is the same. It is called the *I/O ratio*. An important case is when it is roughly the same as the number of *clb* pins per side. For the logic block described above, *I/O ratio* = 4 has been chosen. An advantage is that the number of pins using normal routing channels and I/O channels is roughly the same, so that setting the I/O channel width to the same value as other channels will be a good choice.

The channel width distribution parameters are meant for the case when one wants to examine architectures with different channel widths. These parameters are the ratios of I/O, horizontal and vertical channel widths to the reference channel width. For convenience, when we consider the width of some channel with respect to the width of some other channel, we will use the notation R_h to denote the ratio of the width of the wider channel to the width of the narrower channel. VPR's routing architecture generator cannot generate detailed switch boxes if there exists a pair of channels for which $R_h \neq 1$. Thus, to explore different R_h, one can use only global routing, which yields only a rough routing estimation.

For the global architecture exploration, Betz et al used an area-metric based on the total length of all tracks in the architecture with the minimum reference channel width required to route the benchmark. For various global architecture parameter sets, all 20 benchmarks have been placed and the global router has estimated the minimum reference channel width for all of them. From the results, Betz et al has drawn, among other, the following conclusions:

1) Uniform logic pin location, aspect ratio 1 (square), and the same channel widths for all channels lead to the most area-efficient architectures.

2) If, in case 1), we change the *clb* pin distribution such that the pins are distributed between two *clb* sides (either horizontal or vertical), the width of the channels lying in the direction of these sides must be increased, roughly, by a factor of $R_h = 2$. Such an architecture is only 8% less area-effective than in case 1).

3) If, in case 1), the aspect ratio of the array is increased from 1 to 3, the area increases by 18%. However, if the width of the channels lying along the longer edge of the array is optimally increased, this penalty is reduced to 5%. The optimal channel width ratio is $R_h = 1.6$ in this case.

From now on, unless stated otherwise, we assume that the following default parameters: the logic cluster has parameters $I = 10$ and $N = 4$, *I/O ratio* = 4, uniform channel width distribution and uniform logic block pin distribution. However, when we will consider architectures based on the new logic block other parameters may be used.

2.3.7 Detailed routing architecture exploration

The parameters of the detailed architecture are:

- Connection block flexibility values, Fc_{OUT}, Fc_{IN}, and Fc_{IO}, for logic block output pins, input pins and I/O pads correspondingly;
- Type of the switch box; we consider only disjoint switchbox here;

- The length (L) of the wire segments, their physical parameters (capacitance and resistance per unit of length), the type of the routing switches used to drive them;
- The physical parameters of the routing switches.

By exploring various tradeoffs, Betz et al has discovered several 'optimal' sets of architecture parameters. Given that the default global architecture parameters are as mentioned above, he has found that the following parameter values are reasonable:

$Fc_{OUT} = 0.25, Fc_{IN} = 0.5,\ Fc_{IO} = 1$, and $L = 4$ or 8.

2.4 Placement algorithms for FPGAs

2.4.1 Placement in FPGAs

As it was mentioned when we discussed physical layout design flow, the input provided to placement tool is the netlist. Two basic elements of the netlist are *blocks* and *nets*. Recall, that the task of the layout tools is to place the blocks on the two-dimensional FPGA array and find the routing paths for every net.

Although nets actually join logic block pins, at the level of placement it is assumed that they join blocks. So, the netlist can be described as a multigraph $G_N(V_B, E_N)$ with block as vertices and nets as edges. The nets are actually multi-edges, i.e., they can connect more than two blocks. A net is defined by its *sender block* and a non-empty set of the *receiver blocks*. The netlist blocks connected by a net are called *terminals* of the net.

The placement tool has to find a solution of the placement problem. That solution is also called placement and can be defined as a function:

$$p: V_B \rightarrow X \times Y,$$

where $X = \{0,1,2,\ldots,(n_X + 1)\}$ and $Y = \{0,1,2,\ldots,(n_Y + 1)\}$; n_X and n_Y are FPGA dimensions; extreme column and row values like 0 and n_Y+1 are for I/O pads at the edges of FPGA.

In important case is when the placement must satisfy so-called *relative location constraints* [Xilinx]. These constraints keep groups of blocks that belong to the same module of the circuit next to each other on the array at some specified relative locations inside the rectangle bounding that module. Such rectangular groups of blocks are called *macros*. Macros play important role in the placement algorithms that we want introduce later in this Thesis. Originally, VPR did not recognize any relative location constraints.

2.4.2 Placement Algorithms in General

In VLSI design, placement can be decomposed into two problems: *packing*, on the one side and *connection cost minimization*, on the other side. The class of placement algorithms that deal with *both* problems is called *floorplanning*. The class of algorithms that does not consider packing problem and focuses only on connection cost minimization is called *homogeneous placement* [Len90].

Packing problem is relevant when the rectangular blocks to be placed have different sizes and aspect ratios. The purpose is to arrange, orient and shape a set of rectangular blocks in such a way, that the resulting placement fits into the rectangular chip area of the limited size or to minimize the chip area. The solution of this problem is called *floorplan*. Many efficient floorplanning algorithms have been developed for VLSI layout. One can distinguish several main approaches to floorplanning, namely, oriented mincut method (OM), unoriented mincut (UM), multiway partitioning, and clustering [Len90].

The objective of *connection cost minimization* is to obtain placement such that wiring connections between the blocks would not occupy too much space and the propagation delays would not be too long. To solve connection cost minimization problem, one usually has to find the optimal location of all the blocks relative to one another.

When all blocks have the same shape, size and orientation, packing problem is irrelevant, and one can focus only on the connection cost minimization. That is the task of homogeneous placement. It considers a set of locations at which any of the blocks can be placed. So, it assumes that the blocks have essentially the same size.

2.4.3 Placement Algorithms for FPGAs

Since all FPGA blocks have the same size, floorplanning might seem to be irrelevant for FPGA placement. The default placement routine in VPR is a homogeneous two-dimensional placement algorithm, that places the netlist blocks on *clb* locations. However, floorplanning *is* relevant when it is not just blocks, but rectangular *macros* that have to be placed. Rectangular macros can have different sizes (measured in logic blocks). They can have fixed or floating shapes.

At the first glance, floorplanning seems to be the most appropriate for our purposes (the placement of datapaths for reconfigurable computing), because datapath consist of macros of different sizes.

Floorplanning is a known topic for FPGAs. FPGA vendors, like Xininx, provide floorplanning tools for their FPGA chips. Many floorplanning algorithms have been studied by researchers. Oriented mincut floorplanning for FPGAs has been studied by Emmert et al [Emm98]. Clustering approaches for FPGA are reported by Emmert et al [Emm99] and Russel Tessier [Tess99]. Examining these papers, we found that the

main motivation for FPGA floorplanning was the reduction of computation time for mapping of large circuits (like CPUs). Because now we do not consider mapping large circuits to FPGAs, we preferred to avoid the packing problem. We use an alternative approach, based on homogeneous placement. It is introduced and described in Chapter 4.

Almost every placement problem is a hard combinatorial optimization problem, no polynomial-time algorithms is known that can find exact solutions for it. The optimization cost in VPR consists of two components: the wiring component and timing component. The wiring component is either a wirelength-based or a congestion based cost function. The timing component is a timing-based cost function. Almost every such cost function can be represented as a summation of the *net costs* taken over all nets in the netlist. So, we will define every cost function by the net cost.

Having considered the cost functions, we proceed with an overview of heuristic optimization algorithms used in homogeneous placement and finish with a description of 'T-VPlace algorithm', built in VPR.

2.4.4 Wirelength-based cost functions

A wirelength-based cost function is an estimation of the total length of wiring required to route the design with the given placement. The length is measured in terms of the number of gate-array blocks spanned by the connections. We consider here only the functions that are linearly proportional to the length estimations, although some placement algorithms use quadratic wirelength functions [Len90]. The simplest wirelength cost function is the half-perimeter of the *bounding box.* The net cost in this case is defined by equation:

$$bb_cost(i) = q(\#trm(i)) \cdot [bb_X(i) + bb_Y(i)] \qquad (2.1)$$

where $bb_X(i)$ and $bb_Y(i)$ are horizontal and vertical dimensions of the bounding box, i.e., the smallest rectangle that covers all terminals of net i. So, inside the square brakes in equation (2.1) the half-perimeter of the bounding box is taken. The half-perimeter yields the precise wirelength only for 2 and 3-terminal nets (if their connections form the minimum spanning tree on the terminals). For more terminals, the correction coefficient q depending on the number of net's terminals allows to get a more realistic estimation of the wirelength.

Other wirelength-based cost functions known in the literature split the net into a set of two-terminal sub-nets and apply bounding box measure to every sub-net. The examples are rectilinear minimum spanning tree and star graph-based cost [Len90].

2.4.5 Congestion-based cost functions

A connection cost objective, specific for FPGAs, is that the placement has to be *routable*. Every FPGA channel has a fixed width, that cannot be increased at the design time. The nets most often use the channels that lie between location of the source and destinations. Having placed the blocks, one can estimate the number of nets that will present at the given place of the array after the routing, i.e., one can estimate connection density. If it is too large at some locations, it has to be spread to other places. The situation when the router is not able to route the circuit is called congestion. A congestion-based cost function tries to avoid congestion in routing.

The wirelength-based cost functions focus on the reduction of the number of wiring segments required for routing but they do not try to spread the connection density across the available area. In fact, they tend to locate interconnected blocks as close to each other as possible, which can lead to congestion that otherwise could be avoided.

Betz et al have proposed generalization of bounding box cost that improves the circuit routability on the architectures where channel widths change from channel to channel. That cost function is called *linear congestion cost*:

$$linear_congestion(i) = q(\#trm(i)) \cdot \left[q_X(i) \cdot bb_X(i) + q_Y(i) \cdot bb_Y(i)\right] \quad (2.2)$$

The correction coefficients at $q_X(i)$ and $q_Y(i)$ depend on the average channel width of the channels that cross the bounding box [Betz99, p.56]. That cost function is the default *wiring cost component* in the placement cost function used in VPR. In fact, this function can hardly be called a 'true' congestion-based function, because it is still based on the wirelength.

A simple example of a 'true' congestion-based cost function is the maximum *cut-size* across the channels with *cut-lines* crossing the channels at multiple locations in the array. The cutsize is the number of nets that have terminals at both sides of the cutline [Len90].

In VPR placement, a congestion based cost function defined in [Cheng94] is used, called *nonlinear congestion* function [Betz99, p.57]. The use of that function leads in average to up to 6% reduction of the required minimum channel width to route the circuit compared to the linear congestion cost. Unfortunately, it also leads to an enormous increase of the computation time.

Another cost function that estimates the number of wire segments used at different locations of the array is the alignment-based cost, used in the Xilinx placement tools [Trim94, p. 60]. Alignment enables a more efficient use of long lines at the subsequent routing stage. It keeps track of how many long segments in the channel are used by nets. The placer passes its routing expectation to the router as a

guide. Betz et al also states that VPR placement tool must be improved by taking into account the long segments [Betz99, p.196].

2.4.6 Timing-based cost function

Timing-based cost functions rely on the *timing analysis* of the placement. The timing analysis procedure determines for every source-sink connection in every net its *criticality* value lying in the closed interval [0, 1]. The connections that lie on the critical path have criticality equal to 1; the other connections have smaller criticality values. The timing-based net cost in VPR is defined by the equation [Marq00]:

$$timing_cost(i) = \sum_{j=1}^{\#sinks(i)} delay(i,j) \cdot \left(prev_criticality(i,j)\right)^{crit_exp} \qquad (2.3)$$

where *delay* is the delay of the signal propagation from the source pin of net *i* to the sink of the net that has index *j*.

prev_criticality – the criticality value calculated during the previous timing analysis;

crit_exp – a positive constant showing the importance of critical nets in the cost.

2.4.7 Placement Heuristics

Placement algorithms can be divided into two main classes: constructive and iterative. *Constructive* algorithms try to find the optimum solution by making multiple decisions and thereby gradually shrinking the subspace of the search space where the optimum is sought until it reduces to one element. *Iterative* algorithms make steps in the search space from one point to another, gradually, in many iterations, approaching a local optimum. They start from a random or pre-optimized point of the search space, called the *initial solution*.

The important constructive algorithms for placement are partitioning-based algorithms. They recursively divide the placement region into several sub-regions by *cut-lines* and distribute the netlist blocks among the sub-regions. The case when the number of regions is two is called *bipartitioning*, and the cost function used is *cut-size*. An original cut-size minimization algorithm is Kernighan-Lin heuristic [Len90]. Because we use it in our work, we consider it in Chapter 4. Some partitioning-based algorithms also perform global routing simultaneously with placement, like the one proposed by Huijbregts [Huij96] or the one developed specifically for FPGAs by Togawa et al [Tog94].

As for the iterative placement algorithms, the most important class is *simulated annealing* (SA). VPR uses an SA algorithm. An advantage of iterative placement in general is that it can use virtually arbitrary placement cost function. Another

advantage, more specific for the SA, is as follows. If the proper *adaptive schedule* (a part of the algorithm) is used, the quality of the result is not very sensitive to the initial solution.

A disadvantage of simulated annealing is the long computation time. One of the ways to tackle the problem is packing the blocks in clusters and placing the clusters instead of blocks, and Sankar et al has proposed such an algorithm for FPGAs [Sank99]. Another way is to use a heuristic that converges faster to the optimum, such as 'tabu search' [Emm99] and 'generalized force relaxation' [Goto81].

2.4.8 Default placement algorithm in VPR

By default, VPR places the circuit using the T-VPlace timing-driven algorithm using the simulated annealing heuristic [Marq00]. The cost function minimized is defined by equation:

$$Cost(p) = (1-\lambda)\cdot\eta_W\cdot Wire_Cost + \lambda\cdot\eta_T\cdot Timing_Cost \ , \quad (2.4)$$

where λ is the coefficient for the tradeoff between wiring and timing; η_W, η_T are adaptive normalization coefficients based on the previous values of the correspondent components. The components themselves are calculated as the sum of all net costs defined by (2.2) and (2.3) correspondingly.

Simulated annealing algorithms are based on steps in the search space from the current point to a neighbor point. These steps are called *moves*. A move in the FPGA placement consists of selection of two locations in the FPGA array and swapping the netlist blocks assigned to these locations. Simplified pseudo-code for T-VPlace algorithm is shown below.

Algorithm. Default simulated annealing placement in VPR

```
1:     computeDelayMatrix();
2:     S = RandomPlacement();
3:     T = InitialTemperature();
4:     R = InitialR();
5:     while  (T > threshold()){
6:                 TimingAnalyse();
7:                 while (InnerLoopCriterion()){
8:                         Snew = GenerateMove (S,R);
9:                         ΔC =  C(Snew) - C(S);
10:                        r = random(0,1);
11:                        if (r < exp( -ΔC/T)) S = Snew;
12:                }
```

```
13:                 T = UpdateTemperature();
14:                 R = UpdateR();
15:     }
```

In line 1, the delay look-up matrix is computed. It is indexed by the possible distance values between two blocks $(\Delta x, \Delta y)$ and contains the delay of the fastest connection between such blocks in the given FPGA architecture. Elmore delay formula is used for the delay estimations.

In lines 2-3, a random initial placement is generated, and, based on the standard deviation of the cost function in the neighborhood of the initial solution, the highest 'temperature' of the schedule is computed. It is the initial temperature of the simulated annealing. In line 4, the move radius is assigned, which the maximum distance allowed between the blocks being swapped.

One temperature update is done per iteration of the main loop (lines 5-15). The temperature gradually decreases and when it drops below a certain threshold the loop finishes. Every iteration starts with the timing analysis that has to be regularly repeated because the current placement, `S`, changes. Actually, to keep the criticality values precise, one has to repeat the timing analysis after every move. However, this would considerably slow-down the algorithm and also lead to rapid changes in the timing component of the cost function, and they would lead to deterioration of the search procedure. So, the solution is to perform timing analysis once per temperature update, which usually means every several thousands of moves.

The timing analysis is followed by the inner loop, in which the moves are actually tried. First (line 8), a random move within the given maximum distance (R) is selected. Then (line 9), the increase of the cost (ΔC) that results from the move is calculated. The decision whether the move will be accepted is random (lines 10,11). If the move leads to cost decrease ($\Delta C<0$), it is always accepted. If the move is bad ($\Delta C \geq 0$), the probability that it will be accepted is `exp(-ΔC/T)`.

At the beginning of the annealing, the temperature and the radius are high, so, many bad-quality and long-distance moves are accepted to explore a large part of the search space and avoid trapping into the local minima. The lower the temperature gets the more the search procedure dedicates inself to converging to the minimum in the neighborhood of the current solution. Both explorative and converging behaviors are important, and one has to strike a proper balance between them. They depend on the adaptive schedule, embedded in the routines called by the algorithm. Betz et al has built in VPR an adaptive schedule that is favorable for FPGA exploration, because, while the algorithm adapts to the problem at hand, the computation time is still well-predictable [Betz99, p.52].

2.5 Summary

In this chapter, two main FPGA issues have been considered: the architectures and design tools (focusing on placement). VPR is a complex environment to conduct research on both issues and open for extensions. That is what we are going to do in the following chapters.

2.6 Comments

[1] Some commercial architectures rich in routing resources (such as Virtex architectures) represent an exception from this rule in terms of delay. For Virtex, a '50/50 rule of the thumb' is practical when one wants to estimate the routing delay given the logic block delay [Xilinx].

3. FPGA Architecture Exploration

3.1 Introduction

In this chapter, we present the results of our investigations directed to the extension of VPR's architecture template. As explained in Chapter 2, the architecture generator produces the routing-resource graph based on the architecture specification file. The motivation of our architecture investigations is the development of a new fine-grained architecture for reconfigurable computing.

At first, we introduce, one-by-one, some extra options for the routing interconnect. Although the interconnect topology that VPR already could produce has proven to be effective for random logic benchmarks, we wanted to extend VPR's arsenal for several reasons that will be explained later. The extra options are: a modified connection box, a switchbox with a halved number of switches, and the nearest-neighbor direct connections between logic blocks. For convenience, we refer to the architecture generation means that VPR has already possessed as *original architecture generator*, and the extra options are referred to as *modified architecture generator*. To get an idea about the usefulness of the proposed options, we have evaluated them by using VPR's default architecture exploration framework, described in Chapter 2.

At second, we consider the other important part of the FPGA architecture, namely, the logic block architecture. A new logic block (*nlb*) previously introduced by Leijten-Nowak [Now01a] is briefly described and the appropriate routing models are introduced. The main motivation for introduction of *nlb* is to use it for datapath-oriented circuits. The experimental results with that logic block are postponed until Chapter 5, where we introduce a datapath-oriented benchmark for the new logic block.

3.2 Modified connection box

3.2.1 Connection box generation

Connection box generation is a part of the architecture generation which builds the switch patterns of the connection boxes, given the flexibility (F_C) values for different pins. If the relative flexibility F_C is smaller 100% (a '*sparse*' connection box), it is not trivial to distribute the connection switches among the tracks in such a way that the best routability is obtained.

Figure 3.1 illustrates a connection box generation problem[1] [Betz99 p.71]. One has to generate connection switch patterns for all 4 sides of a logic block. The problem instance contains a set of logic block pins, channel width (*W*), and flexibility values Fc_{IN} and Fc_{OUT} for input and output pins. For every logic block pin, it is

known whether it is input or output, and on which *clb* side(s) it is located. To which equivalence class every pin belongs is important, but obsolete for the original architecture generator, since in a logic cluster all input pins and output pins are equivalent.

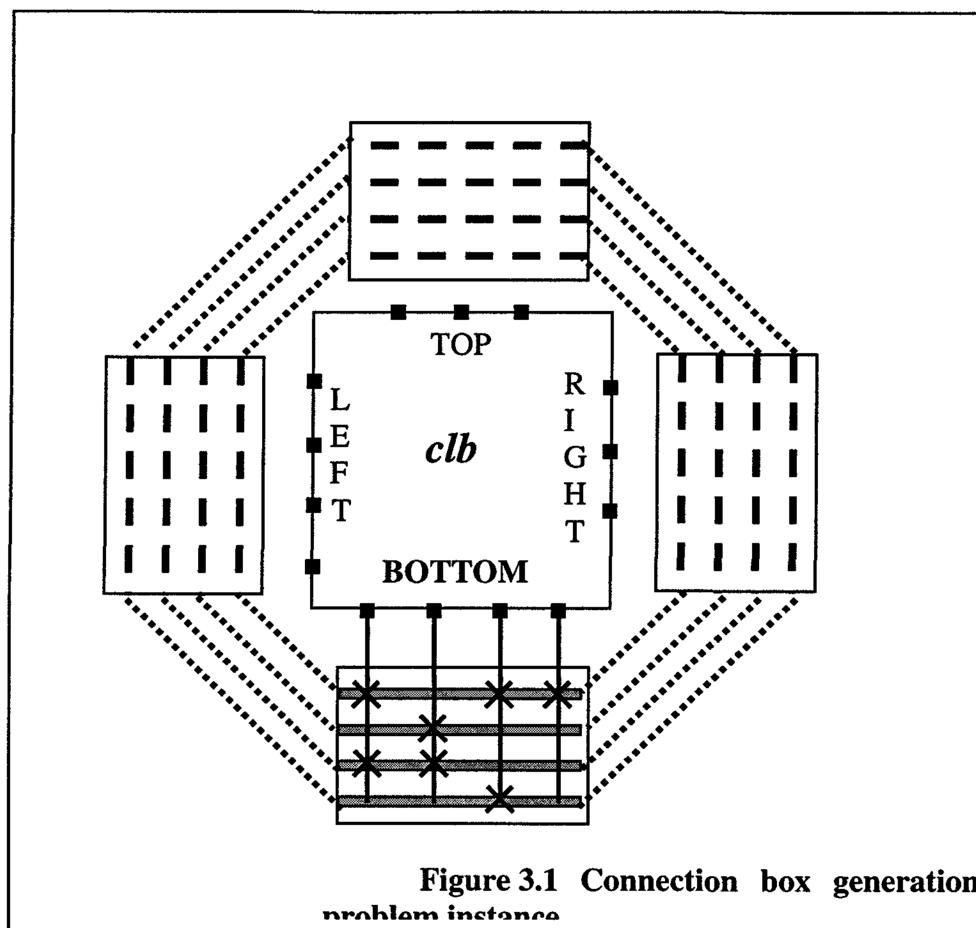


Figure 3.1 Connection box generation problem instance

For convenience, we will refer to the connection box generation algorithm built in the original generator as 'linear connection box generation'. Betz et al has not described that algorithm in the literature, so we had to extract it from the source code. The algorithm is described in Subsection 3.2.2. We have to stress that algorithms we consider in this section are conceptually different from placement and routing algorithms, because they deal with the development of the architecture itself, not the implementation of some circuit on a FPGA architecture.

As it will be shown later in this chapter, we are interested in a logic block whose pins are dedicated for special purposes. So, a desirable feature for the connection box is the possibility to assign different flexibility values to different input or output pins. Examining the linear connection box algorithm, we have seen that it does not support such a feature. So, we have developed an alternative algorithm, called 'balanced connection box generation'. We introduce the algorithm in Subsection 3.2.3.

Note, that in the following two subsections we will talk about the *placement* of switches, but it has nothing to do with the *placement algorithms*, whose task is the placement of netlist blocks.

3.2.2 Linear connection box generation

The connection box is generated in two phases: output and input switch pattern generation. The following discussion holds for any of those phases, except for a small difference, explained later.

Assume that the number of pins to be connected is *K*. Let *M* be the number of switches per pin [2]. To construct a well-routable connection box, the following intuitive hints were used:

1. Uniform switch distribution.
2. Switches must cover as many tracks as possible on every *clb* side.
3. Every input should be connected to every output.

We explain these hints in parallel with the explanation how they were realized in VPR, using Figure 3.2a. Two axes are shown there, the *virtual* channel axis *X* and the *real* channel axis *Y*. The channel region on the real axis is divided into *W* slots representing the tracks of the channel. The channel region on the virtual axis is divided into *K**M* slots. For every pin, the switches are actually placed in the virtual axis' slots and the corresponding track number is found via a simple linear mapping from the horizontal axis to the vertical (see the figure).

The first hint, uniformity, is achieved by the division of the virtual channel region into *M* sub-regions, every sub-region containing *K* slots. Every pin will have a switch in exactly one slot per sub-region. As a result, the density of the switches in the channel will remain roughly the same across the channel.

We will call the slots of the first sub-region the *starting slots*. To every starting slot a pin is assigned. The pins are assigned to the starting slots in some *pin order.* Every pin is connected to the channel by placing a switch at the slot to which it is assigned and then proceeding with equal steps from one sub-region to another (see the figure).

The second hint, i.e., the track cover of the logic block sides, is satisfied simply by choosing such a pin order that the corresponding side order are interleaved, e.g., {top, left, bottom, right, top, left, bottom, right,...}. With such an order, all switches that belong to the same side are evenly distributed along the channel region.

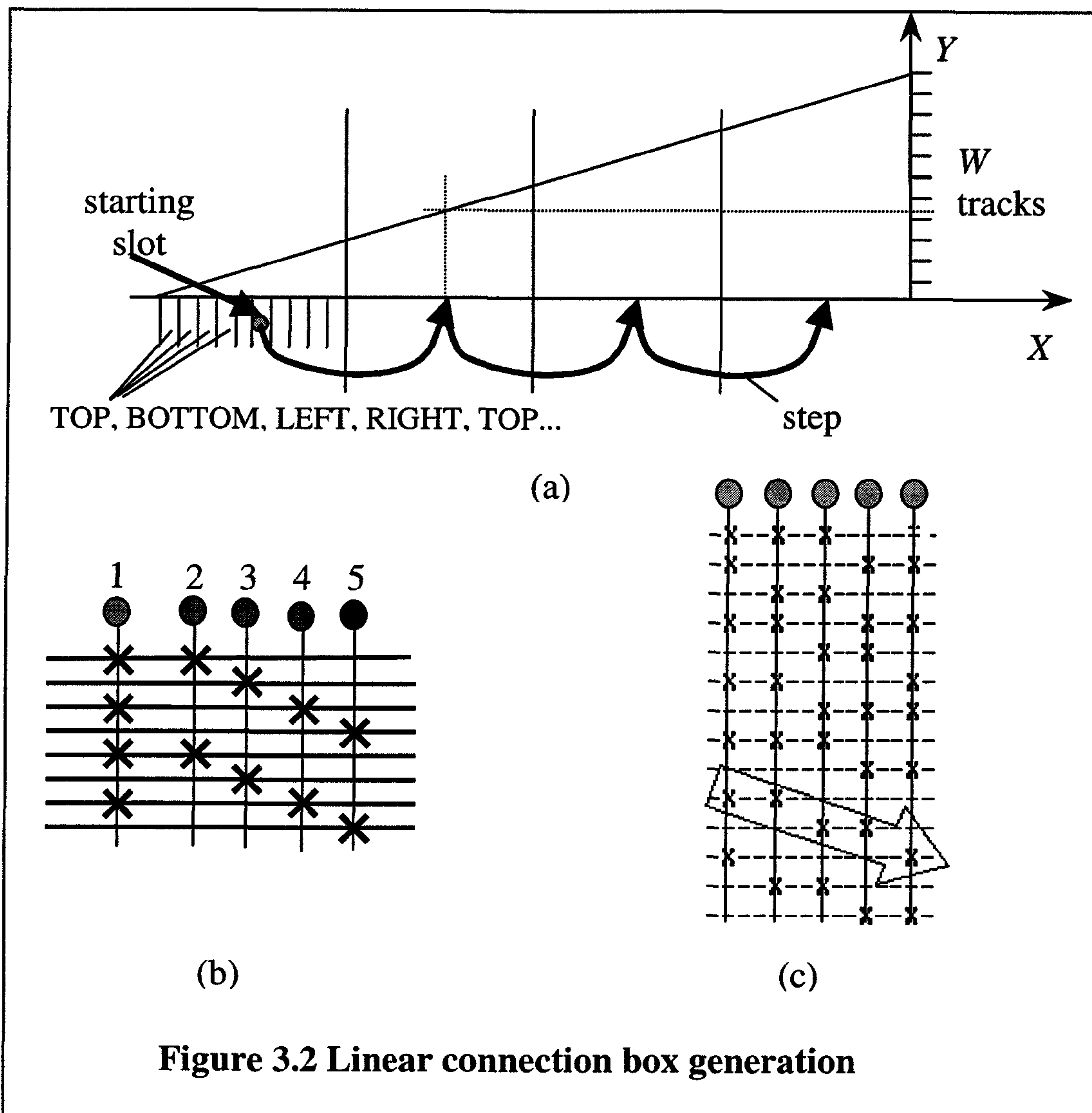


Figure 3.2 Linear connection box generation

The third hint uses the fact that, with the disjoint switchbox topology, all tracks of the same number can be connected to each other, but completely isolated from the other tracks (see Figure 3.1 again). So, if an input has a switch on the track where an output has a switch, they are *connected*, otherwise they are isolated. Isolation should be avoided, but, if no precautions were taken, the isolation would be possible in this algorithm. It is illustrated in Figure 3.2b. Assume that '1' is an input pin and '2', '3', '4', '5' are output pins. In this example, input '1' is connected to outputs '2' and '4', but isolated from '3' and '5'. This situation could be repaired by replacing two switches of the input pin. Note, that this example shows what can happen if the flexibility values are $Fc_{IN} = 0.5$ and $Fc_{OUT} = 0.25$, as in the 'optimal' case (see Subsection 2.3.7).

When such a situation is detected, the following measures are taken. In the input-pin-generation phase, instead of using one linear slope for mapping, a piecewise-linear mapping is used with two slightly different slopes. As a result, in one half of the channel the switches are located more densely than in the other. Which particular half it is (upper/lower) is interleaved, making the connection box, as a whole, uniform. For

example, Figure 3.2c shows a fragment of the connection box switch pattern for the size 4 clustered logic block. It shows the switch patterns of the input pins whose switches are located more densely in the upper part. These pins have been sorted according to the pin order and an arrow stretching over the pattern highlights the switches lying in one of the sub-regions.

3.2.3 Balanced connection box generation

In the proposed pattern generation algorithm, the output connection phase must precede the input connection phase. The output phase can be seen as a simplified version of the input phase, so we consider the input phase first, and then explain the output phase.

The 'balanced box' algorithm uses the same three hints as mentioned in the previous section. For the uniformity of the switch placement, also the channel sub-regions of the real channel region (not the virtual one) are used. The main idea of the algorithm is to use special *intermediate arrays* to satisfy the other two hints. To cover all tracks at every side, we introduce the array `P[t][s]` that counters the number of switches already placed at track `t` on side `s`. To connect every input pin to all output pins, we keep the array of 'output pin importance' `Wi[j]`, and the *importance* of connection of output pin *j* to input pin *i* decreases every time when these pins are connected. The algorithm tries to keep all elements of both arrays equal, that is why it is called 'balanced'.

The following is the list of *input data structures*:

- `num_inputs` - the number of input pins;
- `<i_or[l], s_or[l], m_or[l]>` - the tuple of arrays specifying the pin number, the *clb* side where it lies, and the number of switches that have to be placed; so, each pin can have a different number of switches, which was the purpose of introduction of this algorithm;
- `O[t]` - the set of output pin numbers connected to track `t`, for all tracks.

The *output data structure* of the algorithm is the set of tracks `T[i]` connected to pin *i*, for all pins. The pseudo-code of the algorithm is shown below:

Algorithm. Balanced connection box generation: input phase

```
1:     clear(P);
2:     set_random_seed(1);
3:     for (l∈[0..(num_inputs-1)]) {
4:            <i, s, m> = < i_or[l], s_or[l], m_or[l]>;
5:            set(Wi);
6:            T[i] = Ø;
7:            for (k∈[0..(m-1)]) {
8:                   C_I = Chan_sub_region[k];
9:                   C_II = Arg min_{t∈C_I} (P[t][s]);
10:                  C_III = Arg min_{t∈C_II} ( Σ_{j∈O[t]} Wi[j] )
11:                  t = random_element(C_III);
12:                  T[i] = T[i] ∪ t;
13:                  P[t][s]++;
14:                  for (j∈O(t)) Wi[j] = 0.5*Wi[j];
15:           }
16:    }
```

The algorithm works as follows. First, the switch counters are reset (line 1). Then, the random number generator seed is set to 1, so that this algorithm, being random, could always produce the same result for the given problem instance.

In line 3, the basic loop starts. In every iteration of the loop (lines 4-15), a pin is connected to the channel. First (line 4), the pin's parameters are extracted. Then (lines 5,6), the output pin importance coefficients Wi are all set to 1, and the set of tracks for the given pin is emptied. After that, the inner loop follows.

In every iteration of the inner loop, one sub-region of the channel is considered and a switch is placed there. Originally (line 8), the set of candidate tracks are all tracks lying in the sub-region (C_I). Then (line 9), the candidate set is reduced to the tracks with the minimum number of switches at the given side (C_{II}). Finally (line 10), the set is reduced to the tracks having the maximum importance (the sum of importance values of the output pins connected to the track). A random selection among the remaining candidate tracks is made (line 11). The selected track is appended to the set of tracks for pin i. The switch counter for that track is incremented, and the importance values of the output pins connected to that track are reduced.

So, we have considered the input switch placement. The output switch placement phase precedes the input pin switch phase and uses the same algorithm with two differences:

- The importance values are not used; so, line 5 and line 10 are removed;
- Instead of using counters `P[t][s]`, counters `P[t]` are used, which ignore the side of the logic block; this is done so, because only a few output pins per side is available (only 1 in the logic block considered), so they cannot cover all tracks on every side, they just have to cover all tracks.

3.2.4 The comparison of two connection boxes

To assess the routability of the balanced connection block, 10 MCNC benchmarks have been placed and routed on architectures containing linear connection box and balanced connection box. We used the sample architecture specification file (provided with VPR) with near-optimal parameters (logic cluster size 4, wire segment length 4, etc.). The detailed table with the results are shown in Table 3.1.

Table 3.1 Comparison of connection boxes

Benchmark	Array size	W_{MIN}, linear	W_{MIN}, balanced
alu4	20x20	33	33
apex2	23x23	43	42
diffeq	20x20	27	28
elliptic	31x31	40	40
s1423	8x8	15	14
apex4	19x19	41	41
des	32x32	24	23
dsip	27x27	25	24
ex1010	35x35	45	44
bigkey	27x27	24	23

In 6 cases out of 10, the use of the balanced connection box has lead to a decrease of minimal channel width by 1 track; in 3 cases it remained the same and in 1 case it increased by one track. From this, one can conclude that, for the given architectures, the balanced connection box has a slightly better routability than the linear connection box.

3.3 'Half' switch box

A fragment of the topology of the disjoint switch box is shown in Figure 3.3a. We call it here the *basic fragment* of the switch box. In that fragment, four wire segments coming from the different sides are joined by 6 switches. They can connect any pair (or pairs) of 4 segments. Not only disjoint switch box, but also the other topologies supported by VPR's architecture generator have such a basic fragment, the difference resting on the choice of the track number of the vertical segments and horizontal segments being joined.

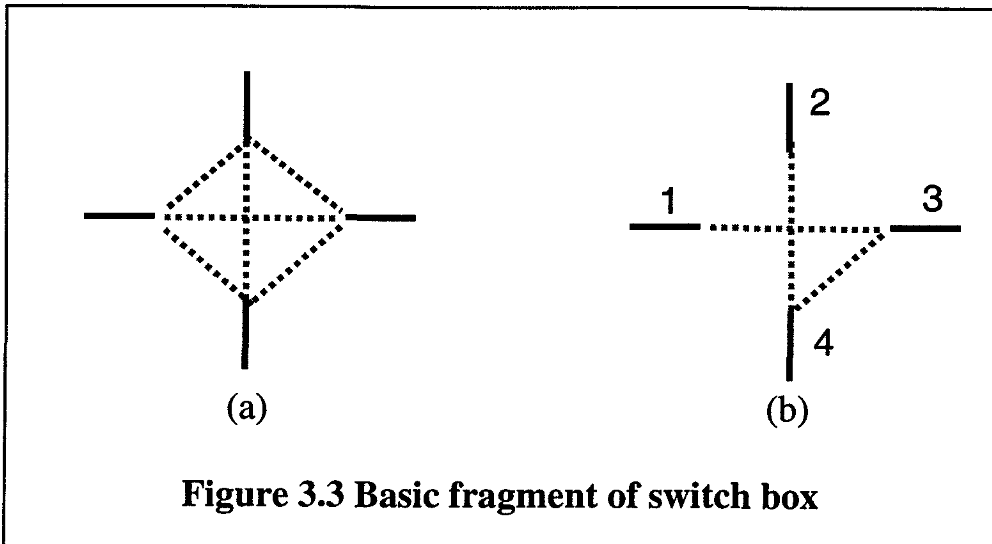


Figure 3.3 Basic fragment of switch box

We have considered an alternative basic fragment for switchbox topologies. The number of switches in the new basic fragment is only 3 (see Figure 3.3b). By reducing the number of switches, we reduce the capacitive load attached to the wire segments, so, the interconnect delay can be reduced. Given that the channel width remains the same, the routing area also reduces.

Lets call the original case *full* switch box and the new topology *half* switch box.

Half switch box can also connect any pair of 4 segments. However, to connect, for example, segment 4 to segment 1 (see Figure 3.3b), one has to use two switches instead of one as in the full switch box, and also to include segment 3 in the path. This leads to increase of interconnect delay. One can also expect that the channel width will have to be increased to keep the architecture routable, because the new topology is less flexible. In the example mentioned above, segment 3 has been used, while it would not be necessary with the full switch box. Note, that the longer wire segments are used the greater is the penalty paid for such an overuse of wire segments.

So, the new basic fragment has both advantages and disadvantages in terms of the interconnect area and delay. Experiments with two benchmarks have been conducted for architectures based on size 4 logic clusters and different wire lengths (L = 1, 2, and 4). The results are shown in Table 3.2.

Table 3.2 Comparison of switch boxes

Benchmark		Full		Half	
	L	W_{MIN}	Area, λ^2	W_{MIN}	Area, λ^2
ALU4 20x20 *clbs*	1	27	8375	28	7425
	2	27	7275	29	7125
	4	30	7425	32	7500
APEX4 19x19 *clbs*	1	32	9917	33	8809
	2	34	9197	37	9114
	4	36	8947	39	9197

For all cases, the area of FPGA tile on minimum channel width has been evaluated by VPR as described in Appendix 1.

The results show that:

1) For wire lengths $L = 1$ and $L = 2$, an area reduction has been obtained due to half switch box, in spite of the increase of channel width. For $L = 4$, the area has increased making the resulting architecture even not worth consideration;
2) Architecture with $L = 1$ and full switch box is the least area-efficient among others;
3) The half-switch-box-based architectures at $L = 1$ and $L = 2$ rival the architecture with $L = 4$ and full switch box, which was one of the best discovered in the previous research.

From these results, one can conclude that half switch box can be effective, if used for short wire segments.

3.4 Direct connections

A *direct connection* is a programmable switch that joins a *clb* output pin to an input pin of a neighbor *clb*. They are additional elements of many island-style architectures, like Xilinx XC4000 or Atmel AT40K. We consider here only the *nearest-neighbor* direct connections, i.e., those joining *clb* at location (x, y) to 8 *nearest clb*s, namely, to 2 *clb*s on the *east* and *west* at $(x \pm 1, y)$, to 2 *clb*s on the *north* and *south* at $(x, y \pm 1)$, and 4 *clb*s on the diagonal directions at $(x \pm 1, y \pm 1)$. A remarkable architecture with all 8 connections is AT40K [Atmel99].

To our opinion, the primary role of direct connections is being a dedicated routing resource for effective implementation of data-path circuits. They are used for carry chain propagation in adders, and, moreover, for partial sum propagation in multipliers [Atmel99]. The secondary role, once they have been introduced, is using them in common routing.

At the time of doing this work, VPR did not support direct connections. To introduce them, we had to extend the format of the architecture specification file. The extended pin specification now supports the assignment of any number of direct connections in any of 8 directions. An important question immediately arises: how these connections are represented internally in VPR, and what timing and routing area models are provided.

Figure 3.4 (a) shows an architecture fragment with a direct connection and (b) the corresponding routing-resource subgraph. Two adjacent blocks, *A* and *B*, are joined by an *east* direct connection from pin o_A to pin i_{B1}. Another way to connect these two pins is to connect them to the same track w by connection switches. To model the

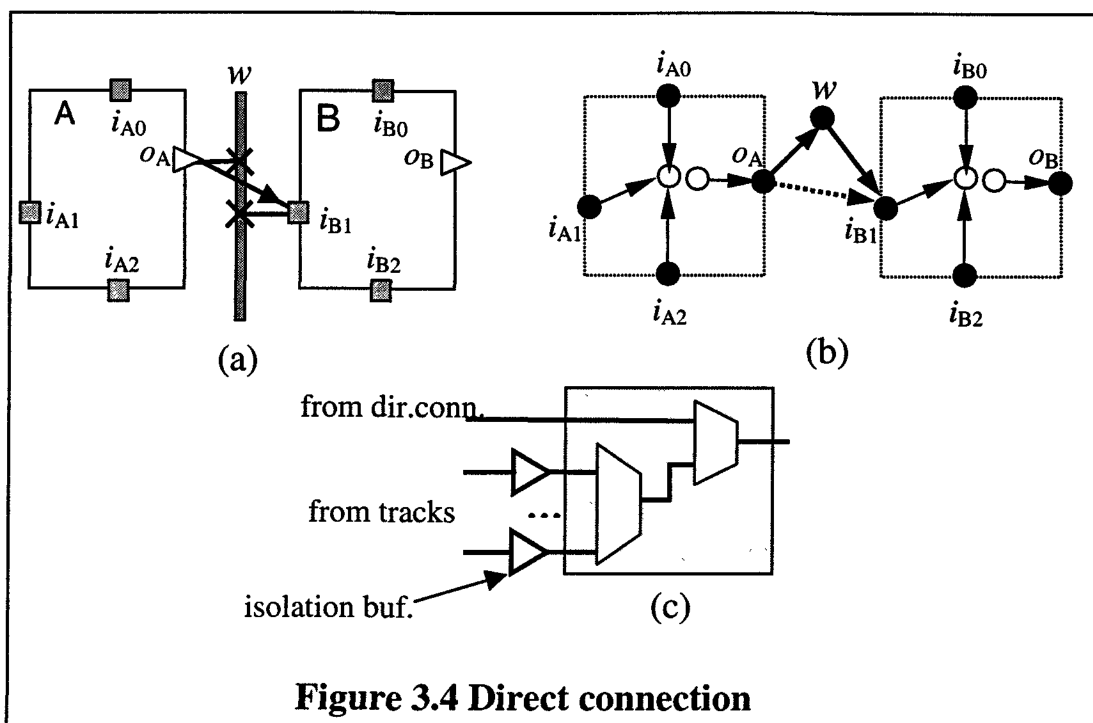


Figure 3.4 Direct connection

direct connection in the routing-resource graph, one just has to join the corresponding pins with an extra edge (shown as a dotted arrow).

For simplicity of VPR source code modification, the timing model that we use now assigns to direct connections the delay value equal to track-to-input-pin delay. This, of course, is an overestimated delay, because it includes the delay of the *track isolation buffer*, lying between the track segment and the connection box multiplexer. Direct connections do not need such a buffer. Moreover, according to Betz et al [Betz99 p.198] and to Atmel documentation [Atmel99], the direct connections should use unbalanced connection box multiplexer path, like shown in Figure 3.4c, to make direct connections faster. However, they would become faster at the expense of *making general connections slower*, but, perhaps, this does not matter much for general connections, because it seems that the isolation buffer delay is much longer than the connection box multiplexer delay. However, our current timing model still assumes that the delay from the tracks and the delay from direct connections are equal, making direct connections less beneficial.

The area model currently used is also imprecise, because it accounts the direct connection just as an extra connection switch. However, the contribution of direct connection transistor area is small compared to the total FPGA tile area, so the impreciseness that direct connection introduce in the estimations of the FPGA tile area is negligible (less than 1%).

To evaluate the benefit of direct connections for general routing (not dedicated one, as in the carry-propagation case), several experiments have been conducted based on MCNC benchmarks and size 4 logic cluster.

A detailed description of these experiments can be found in Appendix 2. Here, we just suggest one remark. The case considered in Appendix 2 is when a direct connection serves only for a limited purpose, namely, to connect only adjacent blocks and only in the given direction. How often such connections are needed in a 'random logic' circuit is a difficult question, and the answer also depends on the placement tool used, but, surely, that number is much less than the number of longer connections. Direct connections can provide fast links from *output pins to input pins*; if also routing links from *input pins to output pins* would exist (routing *through* the logic block), longer connections would also be able to use the direct connections. This opportunity is used, e.g., in Atmel's AT40K series FPGAs. So, it would be an interesting subject for future work.

3.5 New Logic Block

The configurable logic block being developed by Ir. K. Leijten-Nowak at Philips Research is called here *new logic block* (*nlb*) [Now01a]. Our assignment was to model it in VPR.

3.5.1 The Structure of Logic Block .

The global structure of the *nlb* is shown in Figure 3.5. The *nlb* has 22 pins: 16 input pins and 6 output pins. The clock input pin is not shown. The other input pins are shown with gray squares and output pins are shown with triangles. Input and output pins are connected to routing channels. Input pins can be also connected to constant values '0' or '1' and some output pins can be unused.

Two essential parts of the *nlb* are a pair of identical slices: `slice A` and `slice B`. In the discussions that follow we always consider `slice A` as an example, while `slice B` has identical properties. So, consider `slice A`. It is driven by the *major input pins* A0-A3 and two *extension input pins* `EA0` and `EA1` and has two *major output* ports. The data that arrives via the major input pins can be either *preprocessed* by so-called `AND-PLANE` using the extension input pins or just *passed-through* to the four major input ports of `slice A`. Two major output ports can be configured to produce two independent arbitrary Boolean functions of up to four major inputs. Thus, `slice A` can be seen as a 4-input 2-output LUT [Now01b]. However, it contains some extra elements: a carry-input and a carry-output port and carry logic to facilitate effective implementation of arithmetic operations.

Using the *N*+1 LUT property (see Subsection 2.2.2), one can make a 5-input 1-output LUT using the multiplexer at the slice's outputs. For `slice A`, the 5-th input pin is `ICA`, for `slice B` it is `ICB`. Moreover, on the pre-condition that the same set of 5 input nets is routed to the two 5-input LUTs, one more multiplexer can be used to obtain a 6-input LUT with pin `IC` as the 6-th input. The pins `ICA`, `ICB` and `IC` are called *control input* pins.

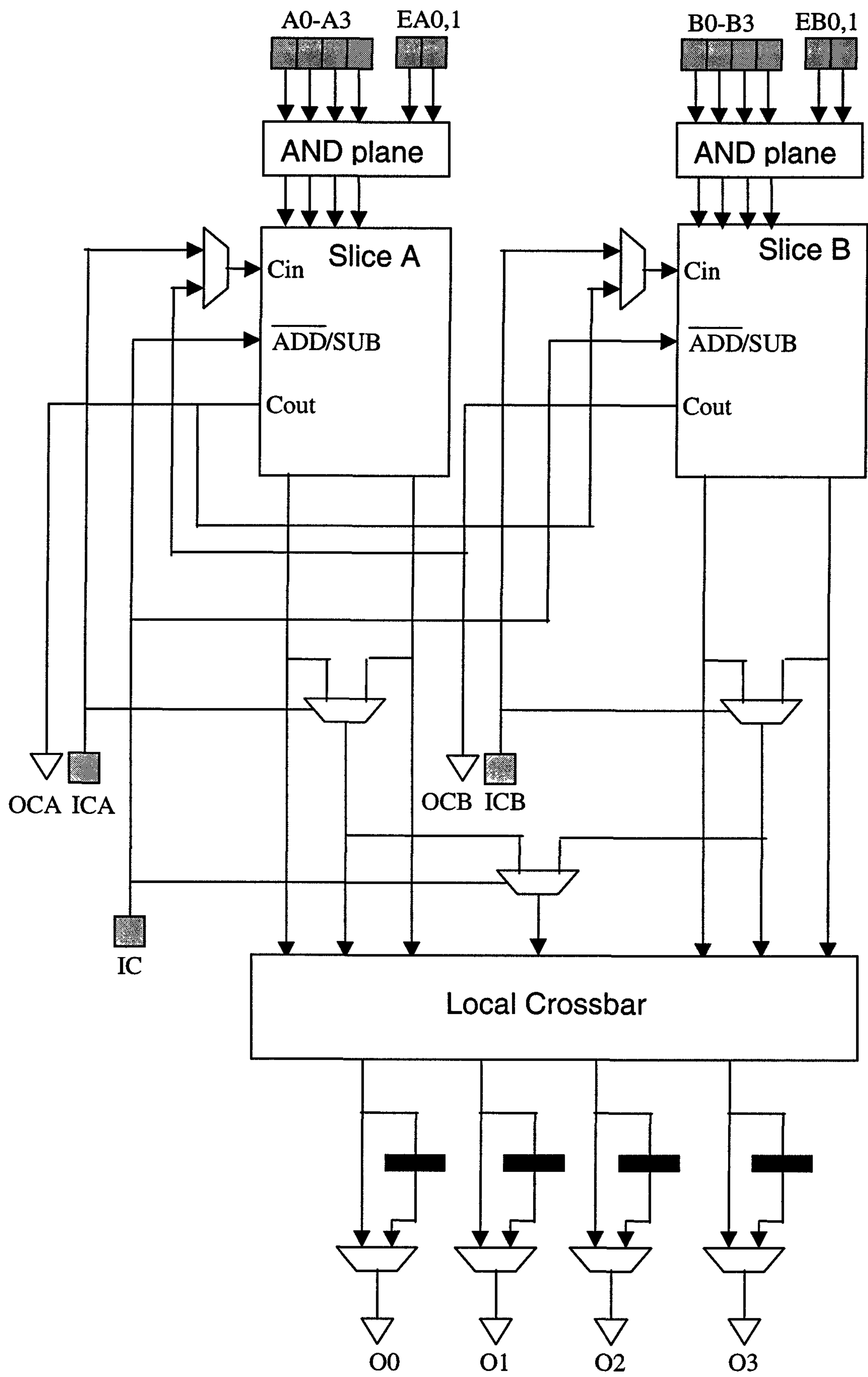


Figure 3.5 Structure of new logic block

Every slice can implement an adder of two 2-bit operands. Another function of control input pins is the external interface for carry-in and $\overline{\text{ADD}}$/SUB signal. The meaning of the last mentioned signal will be explained soon. Carry-out signals use pins OCA and OCB dedicated for that purpose.

In the context of datapath, one logic block can implement a 4-bit arithmetic operation with two inputs and one output.

The outputs of logic functions go to a *local crossbar*, that can be configured to direct them to four *major output* pins O0, O1, O2 and O3. The local crossbar provides equivalence of the major outputs. The major outputs can be configured to be buffered in flip-flops.

3.5.2 Logic Block Operation Modes.

The logic or arithmetic function implemented by a logic block is determined by the configuration loaded into the configuration memory of the block. The set of all possible configurations can be divided into several classes, called *modes*. The configurations of the logic block as a whole form *logic block modes*. The configurations of a slice form *slice modes*.

The slice/*nlb* modes are considered here one by one.

LUT mode. In this mode, *arbitrary* logic functions of major input pins and control input pins are implemented. The logic block can implement two arbitrary logic functions of the inputs A0-A3, two arbitrary functions of B0-B3, two arbitrary functions of 5 inputs, ICA and ICB providing the fifth input, and one arbitrary function of 6 inputs.

ADD mode (x+y). In this mode, one slice implements one full adder of 2-bit operands, x and y. The input operands for slice A are coming from the input pins A0-A3.

Pin assignment:

A0	$x[0]$	A2	$y[0]$
A1	$x[1]$	A3	$y[1]$

Pin equivalence. Pins A0 and A2 are equivalent, because full adder inputs are commutative. Pins A1 and A3 are equivalent for the same reason. Carry-in and carry-out pins provide carry-propagation interface via control pins and/or slice B ports.

SUB mode ($x \pm y$). This mode is similar to ADD mode, but selection signal $\overline{\text{ADD}}$/SUB can be forced to constant '1' or driven by control input pin IC. When $\overline{\text{ADD}}$/SUB gets value '1', operand y is inverted prior to the summation to facilitate the implementation of subtraction as a summation using two's compliment arithmetic.

Pin equivalence Because subtraction is not commutative, none of the input pins is equivalent to another one.

MUX mode, *mux(sel,x,y)*. In this mode, a slice can implement a 2:1 multiplexer with one control operand *sel* and two 2-bit data operands, *x* and *y*. Pin assignment can be seen in Figure 3.6.

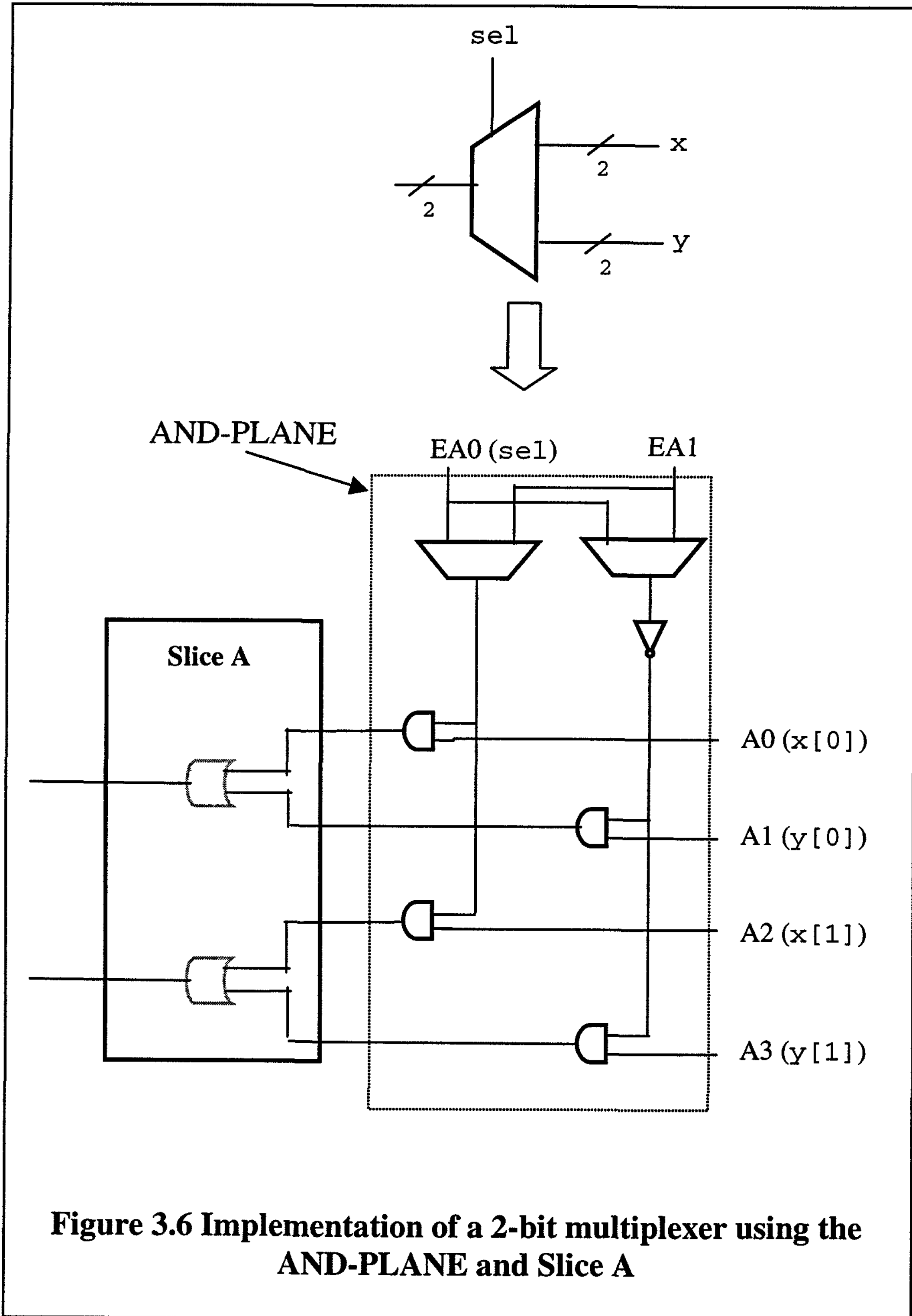


Figure 3.6 Implementation of a 2-bit multiplexer using the AND-PLANE and Slice A

In this mode, the AND-PLANE is used to pass one operand and to put the other one to zero. The selection signal comes from one of the extension pins. Figure 3.6 shows the AND-PLANE schematics and how it is used together with the slice to implement a multiplexer. In this example, `slice A` is configured to produce two ORs of operands {A0, A1} and {A2, A3}. Because `slice A` supports arbitrary functions of major ports, it could be configured for two ORs of {A0, A3} and {A2, A1}, also yielding a 2-bit multiplexing, but with signals on A1 and A3 swapped.

Pin equivalence. As explained above, pins A1 and A3 are equivalent, and the same holds for pins A0 and A2.

3.5.3 The Routing Models of *nlb*

In this subsection we consider the routing-resource subgraph that corresponds to a single *nlb*, or its *routing model.*

Following the analogy with cluster logic blocks, whose pins are equivalent, we could sketch a model for *nlb* as shown in Figure 3.7. The major input pins of every slice would be modeled with four nodes with outgoing edges directed to the same sink.

The problem with the model shown in Figure 3.7 is that it is correct only when every slice is used in the LUT mode to implement up to two functions of its four major inputs. Extension pins, AND-PLANE, control pins and carry propagation ports are ignored. The presence of such extra hardware leads to the fact that, for every logic mode, the appropriate model is different and can be much more complex than the model in Figure 3.7. Note, that the part of that model that corresponds to the major output pins is correct in every mode due to the local crossbar, so we skip major output

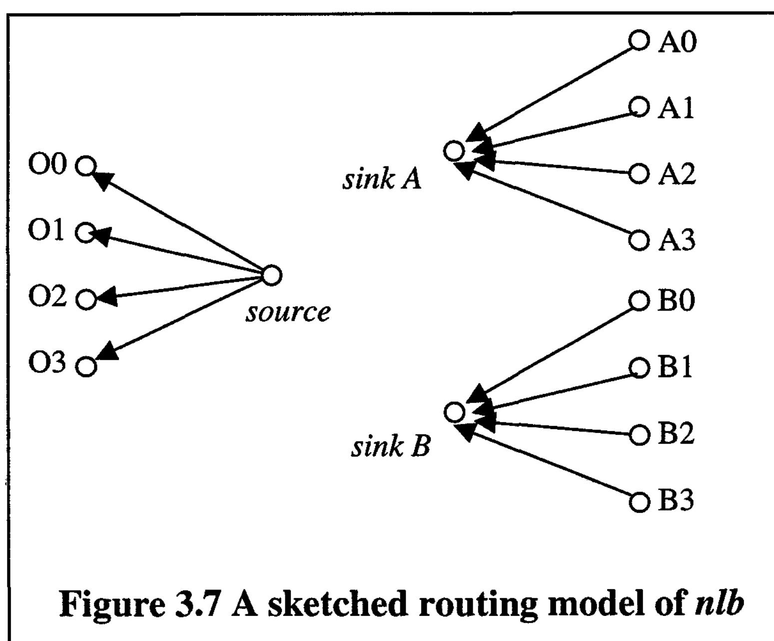


Figure 3.7 A sketched routing model of *nlb*

pins from the further discussion.

We are going to consider `slice A` and *nlb* routing models for every operation mode mentioned above. Note, that the architecture generator currently supports only *one single routing model* for all *clb*s, so, among the models described below we have to select only one. It will be the model of SUB mode. In that mode, no major input pins are equivalent (see the previous Subsection), so, its model is very inflexible and leads to routing of inferior quality. Anyway, we would not have been able to use a more flexible model, because it would be incorrect for the blocks using SUB mode.

To make a full use of the models described below, for every *clb* location, the architecture generator would have to refer to the placement to see in which mode the given location is used and to generate the appropriate subgraph for that location.

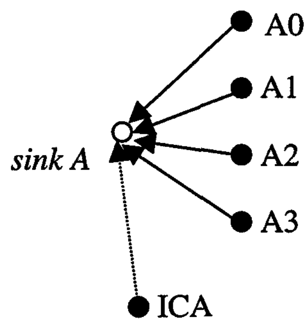
LUT mode. For the case that the slice is used to implement *two* logic functions, the model like in Figure 3.7 is the most appropriate. That model is also repeated (just for one slice) in Figure 3.8a.

In the case when only *one* logic function is mapped per slice (a special case is a 5-input function), also `ICA` input pin can be joined to the sink, which corresponds to 5-input LUT. In the architecture specification file this would correspond to assigning those 5 pins to the same class of equivalence.

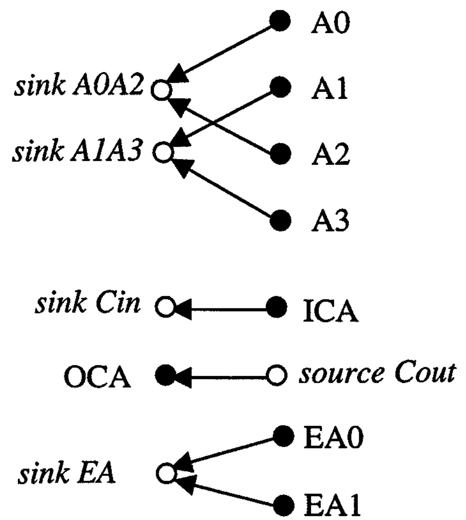
ADD and MUX modes. These two modes have essentially the same routing model. It is shown in Figure 3.8b. Obviously, it is less flexible then in the LUT mode. *nlb* hardware does not allow all four major inputs to be swapped in this case. There are two pairs of equivalent pins {`A0,A2`} and {`A1,A3`} (see the descriptions of those modes for explanation).

Carry propagation pins `ICA` and `OCA` are joined to separate source and sink. Note, that to get an even more flexible model, we could have joint `ICA` to `sinkA0A2`, because carry-in is equivalent to the least significant addition operand, but it might be not advisable, because pin `ICA` has a direct connection for carry propagation between adjacent blocks.

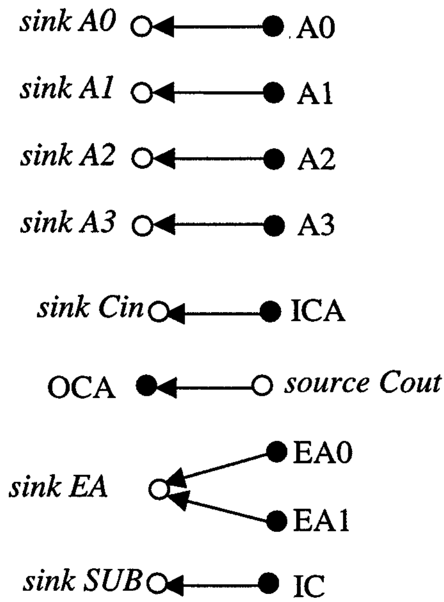
SUB mode. In this mode, no two major inputs can be swapped, because internal hardware would not be able to account for it. The correspondent model is shown in Figure 3.8c. As mentioned above, this model is the model that was used in the architecture specification file for this block, since it is also correct in the other modes.



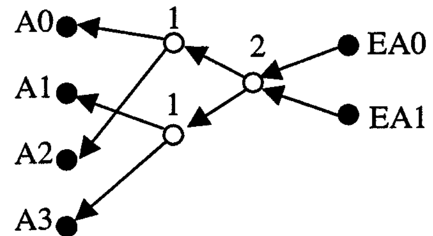
(a) LUT mode



(b) ADD and MUX mode



(c) SUB mode



(d) model extension

Figure 3.8 Routing models for slice A

An extension of the model. Any of three models in Figure 3.8 (a), (b), and (c) can be specified in the VPR architecture file, because it just partitions the pins into equivalence classes. Here we want to suggest an extension part for those models that could make them somewhat more flexible, but cannot be described in the current specification format. The idea is to use extension pins in the modes in which they are not supposed to be used (in LUT, ADD and SUB modes). The extension pins can be regarded as *partially equivalent* to the major pins. This is possible if major pin is forced to logic 1, prior to the AND gate of the AND-PLANE (see Figure 3.6). In Figure 3.8d, the appropriate extra special edges and nodes are shown that would enable the router to make use of this possibility. The special node marked with their capacity values ('1' and '2'; these numbers show in how many routing paths these nodes can be shared).

This extension reflects the following properties of the AND-PLANE:

1) Instead of reaching major input pins via their explicit connections to the track segments, the net can reach them via extension pins.
2) Two different nets can reach major input pins in that way.
3) If a net reaches `A0` or `A2` via an extension pin, no other net can reach the other pin in this pair via another extension pin.
4) Property 3) holds also for pins `A1 and A3.`

3.6 Summary

In this chapter, several FPGA routing architecture elements have been described, namely, balanced connection box, half switch box and direct connections. VPR routing architecture generator has been modified to enable modeling of these elements. They have been evaluated in the context of random logic benchmarks and the architectures developed in the previous research. Based on the experimental results, they were found worth further consideration. Because direct connection was a completely new element for VPR, the current timing model and area models still needs improvement.

Also, a new logic block has been considered in the context of routing. It has been shown that, to model the new logic block effectively, the architecture generator has to be further modified. It must generate different subgraphs of *clb*s depending on the *placement.*

It must be mentioned here that we have not said anything about the timing models, because the precise timing figures for that block are still not known.

3.7 Comments

[1] Following Betz et al, we assume that the same connection switch pattern is replicated in all FPGA tiles. However, in real architectures, one would actually rather expect that it is the switch pattern *layout* that is replicated in every tile. The difference is essential, because in the switch pattern the tracks remain in the same position from tile to tile, and in real layouts the track position may shift from tile to tile [Betz99 p.74].

4. Placement Algorithms.

4.1 Bit-sliced Placement.

In this chapter, we consider datapath-oriented placement algorithms that we introduced in VPR. As it was mentioned in Section 2.4, datapath-oriented circuits consist of regular modules, represented in the netlist as groups of blocks called *macros*. One of the ways to place the macros is to use *bit-sliced* placement, as we do in this chapter.

Let's define the bit-sliced placement in the context of FPGA. *Bit-sliced* approach is the convention that logic blocks that produce the bits of the same significance are bound to a particular column of FPGA. The modules that process data words occupy the whole row or several rows of FPGA. Currently, we support only single-row macros.

Consider Figure 4.1. It illustrates bit-sliced placement assuming that the new logic block (*nlb*) is used. Every *nlb* processes four bits of data. So, four bits are assigned to one column. Bits 0-3 are processed in column 1. Bits 4-7 are processed in column 2 etc. Every logic block performs a unary or binary operator on data words. The *nlb*s belonging to the same operator (such as ADD and SUB) are organized in rows. The *nlb*s actually used by macros are shown with gray squares. In the example that we considered here, macro ADD is a 16-bit adder (it uses 4 blocks), and macro SUB is a 12-bit subtractor (it uses 3 blocks).

The placement of macros using this approach is defined by the sequence in which macros are stacked in the layout on top of each other. In the terminology used in placement algorithms, this problem is called *linear arrangement* [Len90].

At the time of doing this work, VPR did not support macro-based placement. A new architecture parameter was introduced in VPR: number of bits per column (*bpc*). For the new logic block, *bpc* = 4. A macro-oriented netlist is the input of the bit-sliced placement. Blocks are still specified at the bit level, but they get the additional attributes, such as: the identifier of the macro to which block belongs and the relative location in the macro. We had, first of all, to extend VPR placement procedure with the subroutine that reads the additional attributes from the block specification in the netlist. According to the macro name, the initial row for the block is chosen, the bit number is divided by *bpc* to find the column where the block must be placed.

When the initial bit-sliced placement is ready, one of the two optimization algorithms that we use in VPR finds an optimal linear arrangement of macros. One of the algorithms is a constrained default placement algorithm of VPR (T-VPlace). T-VPlace algorithm has been described in Section 2.4. The difference of the constrained algorithm is that, instead of swapping logic blocks, it swaps complete rows of logic blocks. The timing and wiring cost of the nets is calculated for every bit-level net, just as if the bit-level placement was done.

The second algorithm that we use is a successive bipartitioning heuristic. It works with the macro-level netlist. After a brief overview of the related work, we proceed with the description of that algorithm.

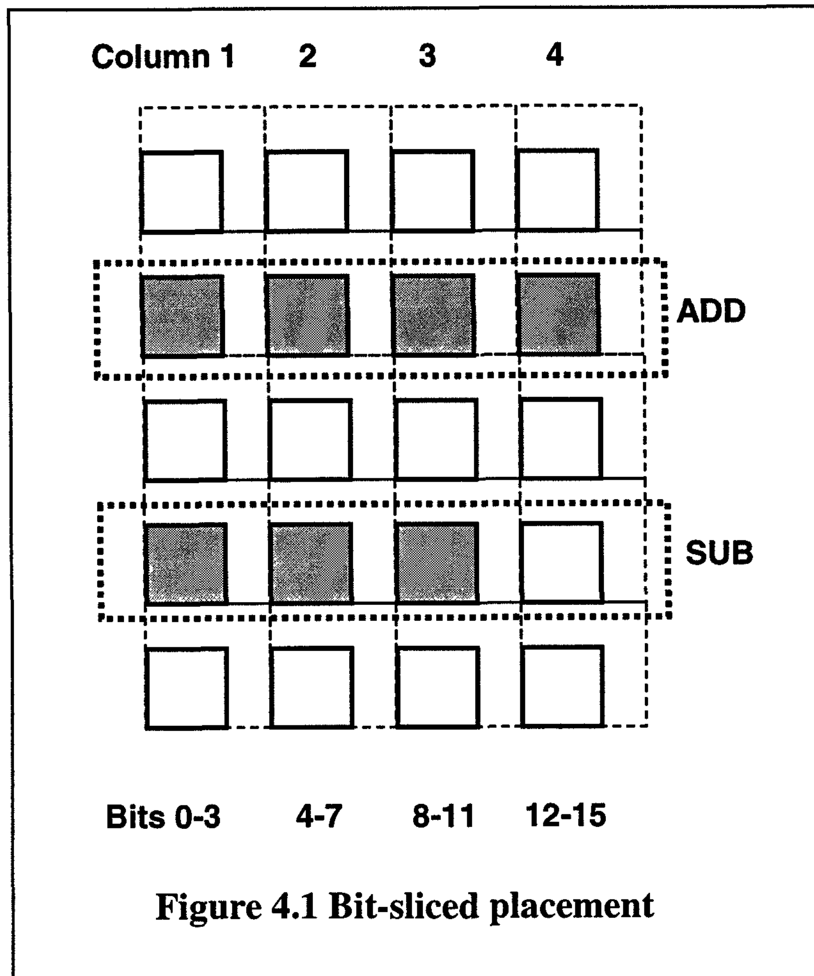


Figure 4.1 Bit-sliced placement

4.2 Related work

The bit-sliced placement in FPGAs has been studied, among others, by Koch at University of Braunschweig and Callahan at University of Berkeley.

Koch has developed so-called Structured Design Implementation (SDI) suite of specialized tools, which aims at efficient mapping of regular datapaths on FPGAs [Koch96]. In all steps, regularity is preserved whenever possible, or restored after the necessary disruptive operations. The result is sophisticated mapping and placement with a regular bit-sliced layout. The optimal linear arrangement is found by a genetic algorithm.

The work of Callahan has been mentioned in the introduction. He has developed a C compiler that extracts some software loops from the C code and implements them of a fine-grained coprocessor architecture. The word-level operations are placed using the bit-sliced style. Optimal linear arrangements are constructed using a dynamic programming algorithm.

4.3 Bipartitioning Algorithm

4.3.1 Bipartitioning for Linear Arrangement

The task of bipartitioning algorithms is the solution of the mincut problem. Assume that we have a netlist specified by an undirected graph and the vertices of the graph have to be placed in the area, called *room*. The room is divided into two (upper and lower) rooms by a horizontal cut-line, and half of the vertices is assigned to the upper room, the other half to the lower room. The problem is to partition the vertices into two sets so that the least number of edges join the vertices of different sets. In other words, the minimum number of connections must cross the *cut-line* that divides the room into two. That number is called *cut-size* and it is the cost function of the algorithm.

Figure 4.2 shows the idea of application of bipartitioning for the optimization of linear arrangement. The stack of macros is first partitioned into two sub-stacks. Then, every sub-stack is partitioned into two, etc.

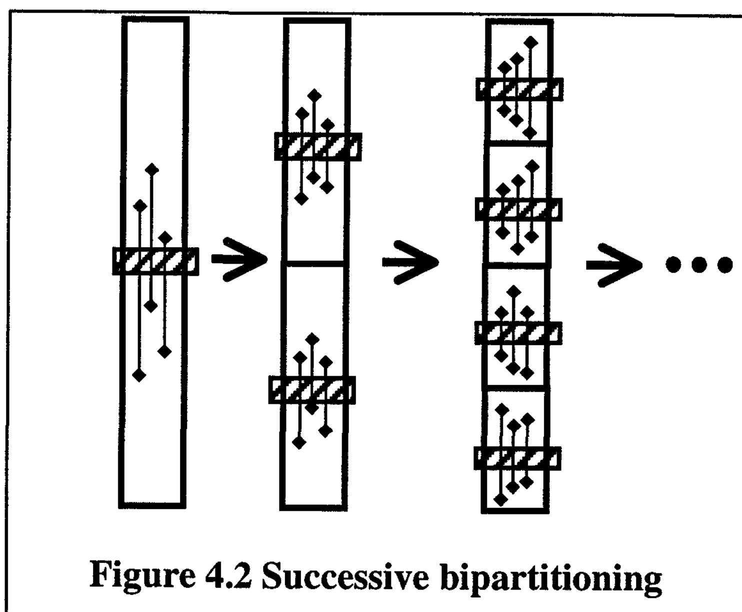


Figure 4.2 Successive bipartitioning

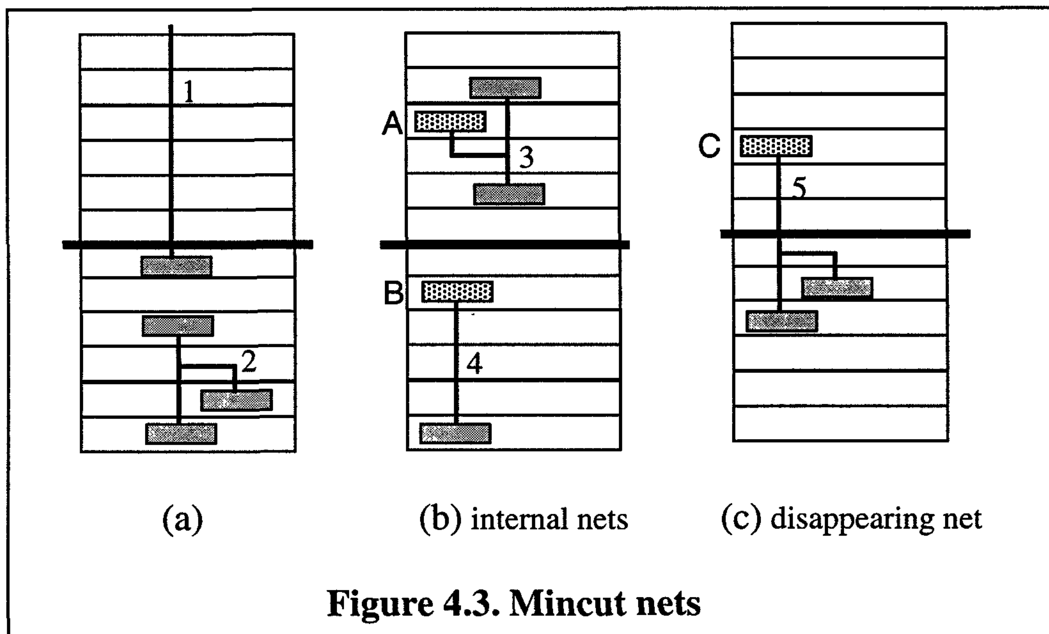
Mincut problem is NP-complete [Len90]. To solve it, we use Kernighan-Lin heuristic [Len90]. It has been adapted for linear arrangement and multiterminal nets (Lengauer defines that heuristic only for two-terminal nets). It is known that another algorithm, Fedducia-Mattheysus bipartitioning, is a more robust heuristic, but we currently implemented only the Kernighan-Lin algorithm.

4.3.2 Background

In this bipartitioning algorithm, we consider not the bit-level netlist of logic blocks, but the word-level netlist of macros. Bit-level netlist can be translated into a word-level netlist by a trivial algorithm that scans all nets and blocks of the bit-level netlist

and makes use of the macro information. *Macro net* (or, in this section, just *net*) actually represents a *bunch* of bit nets that carries one word or one bit when it is a control net. Every net has one source macro, called the *ancestor* of all sink macros. The sink macros are called its *successors*. An ancestor and all its successors are called *terminals* of the net.

Assume that the room is a linear arrangement of n_Y slots, and every slot can contain one macro. In the examples shown in Figure 4.3, $n_Y = 12$. A *mincut net* is a net that has at least one terminal in the room. For example, net 1 that connects a macro present in a room to other macros above the room is a mincut net (see Figure 4.3a). Another mincut net is net 2, whose all three terminals are in the room.



(a) (b) internal nets (c) disappearing net

Figure 4.3. Mincut nets

Kernighan-Lin algorithm is based on the swaps between slots in the upper and lower halves of the room. A mincut net that has a macro that has to be swapped among its terminals is called an *affected net*. Assume that macros *A* and *B* have to be swapped (Figure 4.3b), then nets 3 and 4 are affected nets.

We will define the *cut-size* here as the number of mincut nets that have terminals on both sides of the cutline. For example, net 1 contributes to the cut-size, because one of its terminals is in the lower room and the rest is above the room. The mincut nets that do not contribute to the cut-size are called *internal* nets (like nets 2,3,4). Internal nets, when affected, will contribute to the *increase of cut-size* (like nets 3 and 4 if A and B are swapped). Affected nets that lead to the decrease of cut-size are *disappearing* nets, like net 5 in Figure 4.3c. Disappearing nets are the mincut nets that have only one terminal on upper/lower side, and that terminal has to be moved to the other side due to a swap, like macro C in Figure 4.3c. The other condition for disappearing is that the terminal being moved is not swapped with another terminal of the disappearing net.

The algorithm is based on finding the best swap among all possible, i.e., the move that leads to the smallest increase of the cut-size. As described above, the increase can be calculated based on the affected nets. Even if the increase is positive, it is temporarily accepted and algorithm again proceeds with the search for the best move. This helps the algorithm to find the improvements of the cut-size that require not just one but several swaps to be performed.

4.3.3 The Specification of The Algorithm

Let's consider the pseudocode of our variant of Kernighan-Lin heuristic.

Algorithm. `Kernighan_Lin_Pass(y_bot, y_top, y_cut, max_s) {`

```
1:      compute_mincut_nets( y_bot, y_top, y_cut );
2:      unlock_locations( y_bot, y_top );
3:
4:      best_cost = get_cutsize();
5:      swap_cost[0] = best_cost;
6:      best_s = 0;
7:
8:      for (s=1; s<=max_s; s++ ) {
9:         best_delta_cost = +∞;
10:
11:        for (y_hi = y_cut; y_hi<=y_top; y_hi++ ) {
12:
13:          if (locked[y_hi]) continue;
14:          imacro_hi = macro_at_location[y_hi];
15:
16:          for (y_lo = y_bot; y_lo < y_cut; y_lo++ ) {
17:
18:              if (arr_location_locked[y_lo]) continue;
19:              imacro_lo = arr_macro_at_location[y_lo];
20:
21:              get_affected_nets(imacro_hi, imacro_lo);
22:
23:              delta_cost = balance_affected_nets();
24:
25:              if (delta_cost< best_delta_cost) {
26:                 best_delta_cost = delta_cost;
27:                 best_swap_pair   = <y_lo, y_hi>;
28:                 save_affected_nets();
29:              }
30:          } /* END for y_lo */
31:       } /* END for y_hi */
32:
33:       <y_lo, y_hi> = best_swap_pair;
34:
35:       locked[y_lo] = TRUE;
36:       locked[y_hi] = TRUE;
37:
38:       update_affected_nets();
39:
40:       swap_cost[s]= swap_cost[s-1] + best_delta_cost;
41:       swap_pair[s]= best_pair;
42:
43:       if (swap_cost[s] < best_cost) {
44:         best_cost = swap_cost[s];
45:         best_s = s;
```

```
46:        }
47:     } /* END for s */
48:}
49:   for (s=1; s<=best_s; s++ ) perform_swap(swap_pair[s]);
```

Usually, one pass of the algorithm is enough to find a solution that could not be improved by one more pass [Len90]. The arguments of the algorithm are the room boundaries (top and bottom), the position of cut-line, and the number of slots in the smallest half of the room (`max_s` = $\lfloor n_Y / 2 \rfloor$). That number is, at the same time, the number of swaps performed by the algorithm before it stops, and it is enough to swap the two halves of the room.

First (line 1), a subroutine computes two arrays of counters, that, for every mincut net, contains the number of terminals above and below the cut-line. Lets call them `upper_terminal` and `lower_terminal` arrays. In line 2, all slots in the room are unlocked (which allows to consider macro swaps at all locations). In lines 3-6, the initial situation is declared to be the best known. Array `swap_cost` is used to store the cut-size values after every accepted swap. Variable `best_s` points to the position in that array which contains the best cost. Also, array `swap_pair` will be later used to store the swapped locations.

Line 8 is the begin of three main nested loops of the algorithm. In every iteration of the first (outer) loop, the best swap among all possible and not yet locked swaps is found, and the correspondent slots are locked. In the calculations of cut-size it is assumed that swaps found at the previous iterations have been accepted. Variable `best_delta_cost` indicates the cost increase that the best known in the current iteration candidate-swap would bring.

The second and third nested loops provide that variables `y_high` and `y_low` iterate through all unlocked locations in the upper and lower half correspondingly. The swap of two macros, identified with indices `imacro_hi` and `imacro_lo`, is considered as a candidate. In line 21, a subroutine is called that builds the list of affected nets, i.e., all nets that are connected with these macros. In line 23, a function call analyses affected nets and returns the difference between the number of internal affected nets and disappearing nets (the increase in cost). If that value is the best among considered candidates, the swap locations and affected net list is saved.

After the second loop, the locations of the best swap are locked and the list of affected nets of the best swap is retrieved, and the `upper_terminal` and `lower_terminal` counters for the affected nets are updated (lines 33-38). The swap cost and locations are stored in the array (lines 40, 41), and a check is performed if that swap leads to a better cost than encountered so far (lines 43-45).

After the pass is finished, variable `best_s` points to the tail of the sequence of swaps that leads to the improvement in cost. That sequence just has to be executed (line 49).

4.4 Summary and Cost Function Discussion

Two macro-oriented placement algorithms have been implemented in VPR. Both assume bit-sliced placement convention. In that case, the optimal linear arrangement of macros in the stack of macros has to be found. In one case, it is done by an adjusted default T-VPlace algorithm, and, in the other case, by a min-cut algorithm.

One can show that the complexity of the complete placement using successive bipartitioning and `Kernighan_Lin_Pass` algorithm is $O(N_M^3)$, where N_M is the number of macros. Although it is greater than the $O(N_M^{4/3})$ complexity of the T-VPlace algorithm [Marq00], in our experiments the min-cut algorithm run much faster. The reason for that is that the cost calculated in the constrained T-VPlace is still performed at the bit level, whereas bipartitioning works at the macro level with a simple cost function.

Having mentioned the difference in objectives of those two algorithms, lets try to compare them in terms of cut-sizes at all cutline positions.

T-VPlace minimized the bounding box of the nets. The bb_Y component of that cost is the sum of contributions of the cut-sizes at all horizontal cut-lines:

$$bb_Y = \sum_y cutsize(y)$$

Successive bipartitioning gives the cut-size at the cut-line in the middle of the stack the greatest priority, the next priority goes to the cut-lines in the middle of upper and lower halves, the next one goes to the middle of quarter parts etc:

$$bipartitioning_cost = \sum_y priority(y) \cdot cutsize(y)$$

The cost function that would be directly related to the channel width (a congestion-based cost function) is to a mini-max cost [Len90]:

$$congestion_based = \max_y ((cutsize(y))$$

Maximum cut-size is a minimal bound for channel width. From this we see, that neither of the algorithms has tried to minimize that bound directly. In the future, one could try to improve them in that direction.

5. Benchmark and Results

In this chapter, we present the results of implementation of a datapath-oriented benchmark on fined-grained architectures based on the new logic block.

5.1 Direction Detector Benchmark

The benchmark we consider here is derived from a video-processing unit for Progressive Scan Conversion Algorithm. Assume we have three pixels in row “a” and row “b” (see Figure 5.1). Consider three vectors A, B and C, each one having a two-bit code. Lets call the absolute difference of the 8-bit pixel values at the ends of the vector the *gradient* of the vector.

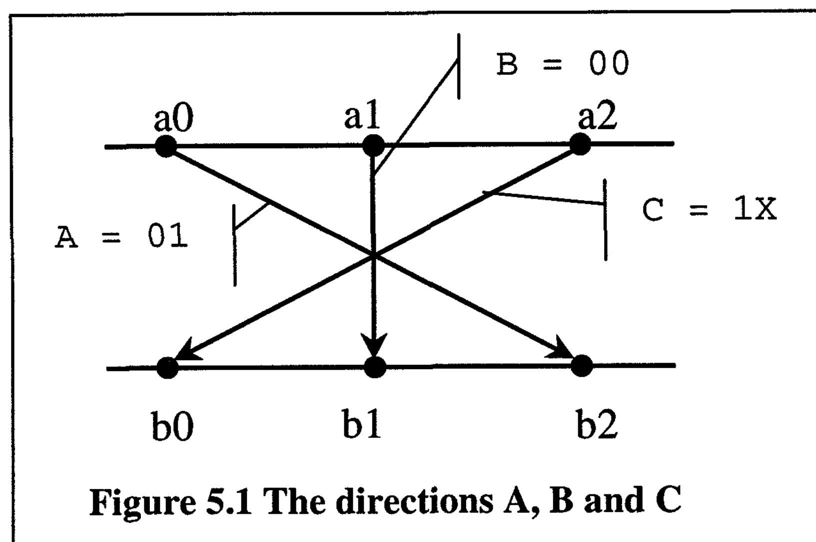


Figure 5.1 The directions A, B and C

The direction detector calculates the gradients and finds out which vector has the greatest gradient and the smallest gradient. If the difference between them is greater than some threshold value, it sends the code of the vector with the greatest gradient to the output. Otherwise, it outputs the code “00”, which corresponds to the default direction B.

***Structural* Specification in C++ for 8-bit Direction Detector:**

```
1   Integer a[3], b[3]; // a[0] and b[0] are inputs
2
3   a[1] = reg(a[0]); a[2] = reg(a[1]);
4   b[1] = reg(b[0]); b[2] = reg(b[1]);
5
6   Integer A = abs(a[0] - b[2]);
7   Integer B = abs(a[1] - b[1]);
8   Integer C = abs(a[2] - b[0]);
9
10  Bit A_gt_B = A>B;
11  Bit A_gt_C = A>C;
```

```
12  Bit B_gt_C = B>C;
13
14  Integer max_AB = mux(A_gt_B,  A, B );
15  Integer min_AB = mux(A_gt_B,  B, A );
16
17  Control C_is_min  =  A_gt_C & B_gt_C;
18  Control C_is_max  = (!A_gt_C) & (!B_gt_C);
19
20  Integer max_ABC  = mux(C_is_max, C, max_AB );
21  Integer min_ABC  = mux(C_is_min, C, min_AB );
22
23  Integer MM   = abs(max_ABC-min_ABC);
24
25  Constant threshold;
26
27  Bit MM_gt_thre = MM > threshold;
28
29  Control direction_bit0 = MM_gt_thre & A_gt_B;
30  Control direction_bit1 = MM_gt_thre & C_is_max;
31
32  Output(direction_bit0);
33  Output(direction_bit1);
```

In lines 3-4, flip-flop registers are declared that represent one clock-cycle-delay. In lines 6-8, gradients are calculated. In lines 14-21, the minimum and maximum gradient are found. In lines 23-27, the difference between the gradients is found and compared to the constant threshold. In lines 29-33, the direction code is constructed and sent to the output.

The design flow for this benchmark is shown in Figure 5.2. First, the specification is compiled and linked with a C++ netlist-expansion library, containing classes, functions and operators used in the specification.

The results of the compilation and execution are:

1) a bit-level netlist of the datapath (word-processing part) in VPR format;

2) bit-level equations for control part of the circuit.

The bit-level equations for this benchmark are just the replication the four control bit operations present the C++ code. They are supposed to be an input of the random logic mapping tool, but currently they have been mapped manually. The resulting netlist is fed as an input to VPR together with the architecture specification file and the layout is generated.

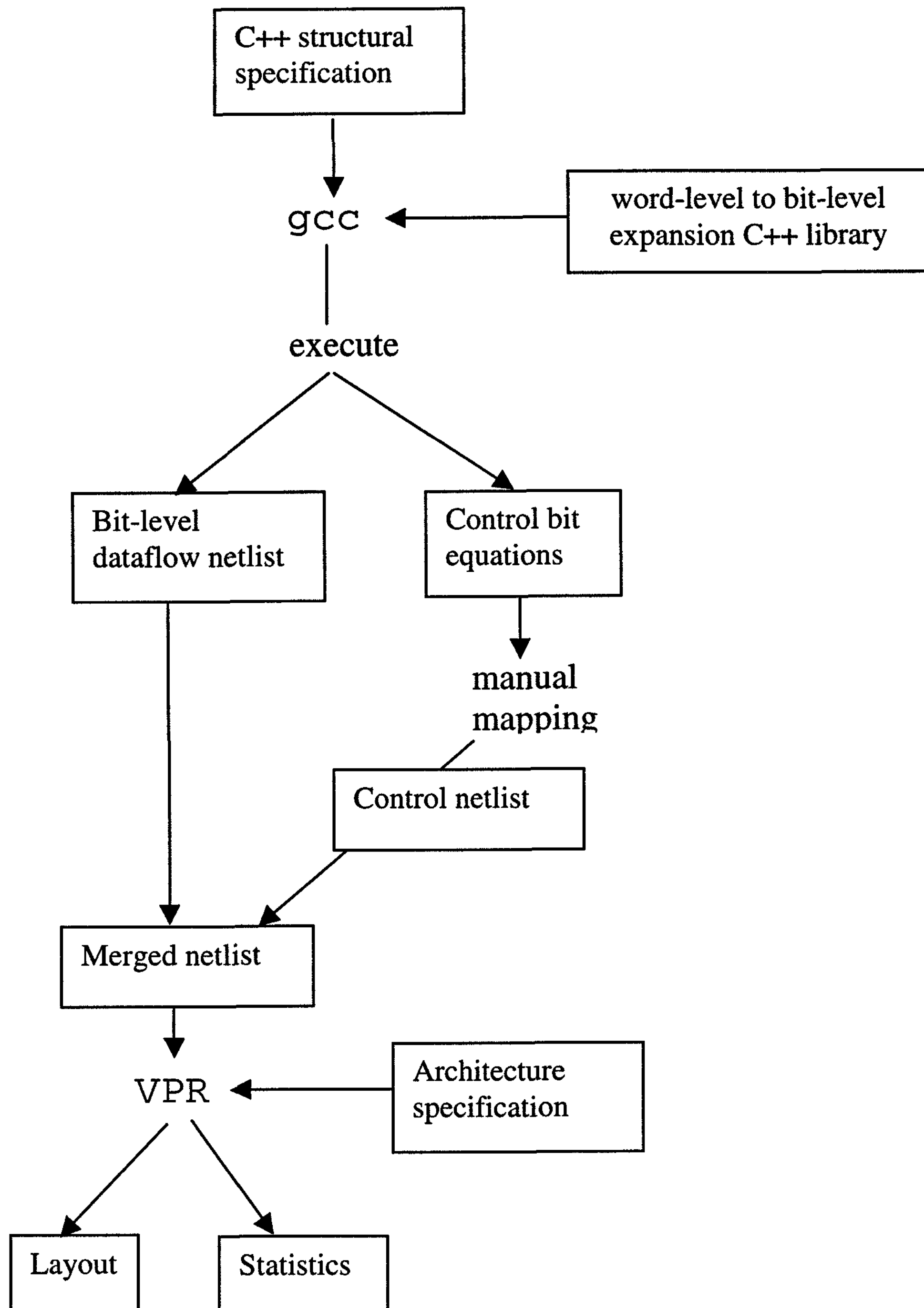


Figure 5.2 Direction Detector Design flow

5.2 Datapath Mapping

To map datapath operations of the direction detector, the following word-level operations had to be translated to the bit-level netlist: '–', 'abs', '>', 'mux'. To do this translation, the specification is compiled, linked with the netlist-expansion library and executed. That library is one of the contributions of our work.

During the execution of the compiled and linked specification source code, every datapath operation function call adds a *macro* to the list of macros in the memory. In this way, the specification is translated to a list of macros. At the end, the list of macros is scanned and, for every macro, a *chain* of *nlb*s is written to the bit-level netlist file. Every *nlb* in the chain is responsible for processing of 4 bits. The number of *nlb*s in the chain is, roughly speaking, the bitwidth of operands divided by 4.

Operations '–', 'abs', '>' are implemented with what we call *carry chain*. *nlb*s in that chain are assumed to be configured to SUB mode. 'mux' operation is implemented on *multiplexer chain*, where every block is configured to MUX mode.

The set of chains written to the netlist file can be seen as the top level of the netlist hierarchy. We have implicitly introduced that level into the netlist file format of VPR. The next level specifies the interconnections between netlist blocks, that we call nets. The lowest level of hierarchy is the structure of the netlist block. Let us spend some words on that level.

According to VPR's netlist format, for every netlist block, there is a 'sub-netlist' of *subblocks*. Every subblock has a delay listed in the architecture specification file. The purpose of the 'sub-netlist' is using them for the timing analysis. Based on the subblock delays and interconnection delays, VPR builds a timing graph and finds the critical path in it. It is required for critical path minimization during placement and routing. We have adopted some subblock netlist model for the new logic block, but it is not presented in this Thesis.

5.3 Architecture Specification

The file with FPGA architecture specification for placement and routing of direction detector was composed based on a modification of a sample architecture file ('4x4lut_sanitized.arch') provided with VPR. This architecture has parameters that are close to the best architecture parameters discovered with VPR (See Section 2.3). It is based on size 4 cluster logic block.

We have modified that sample architecture. First of all, we changed the logic block pin specification and subblock delays, so that we can model *nlb* as efficiently as we can with the current version of the tools. We list the parameters here one by one. The listing of the final architecture specification file can be found in Appendix 5. The description of the file format can be found in VPR User Manual.

Global parameters. A new architectural parameter that is needed for bit-sliced placement is number of bits per column, *bpc* = 4. Because direction detector has 2 inputs, we need *I/O ratio* = 2×4 = 8 input pads per column. The channel width distribution is uniform.

Pin parameters. Figure 5.3 shows an FPGA tile using *nlb*. As one can see, a uniform pin distribution was used. As for equivalence classes, they have been assigned according to the routing model of SUB mode, as described in Chapter 3.

Direct connections. In this benchmark, a single east-direction direct connections is used for fast carry propagation in ripple-carry modules, namely, in subtractor, comparator and absolute value calculator. That connection joins carry-out pin OCB to carry-in pin ICA (see Figure 5.3).

Connection box. Because using sparse connection boxes makes sense only if several input pins are equivalent, and because for SUB mode it is not the case, *fully* connected connection box has been used ($Fc_{IN} = 1$ and $Fc_{OUT} = 1$), not like it is shown in Figure 5.3.

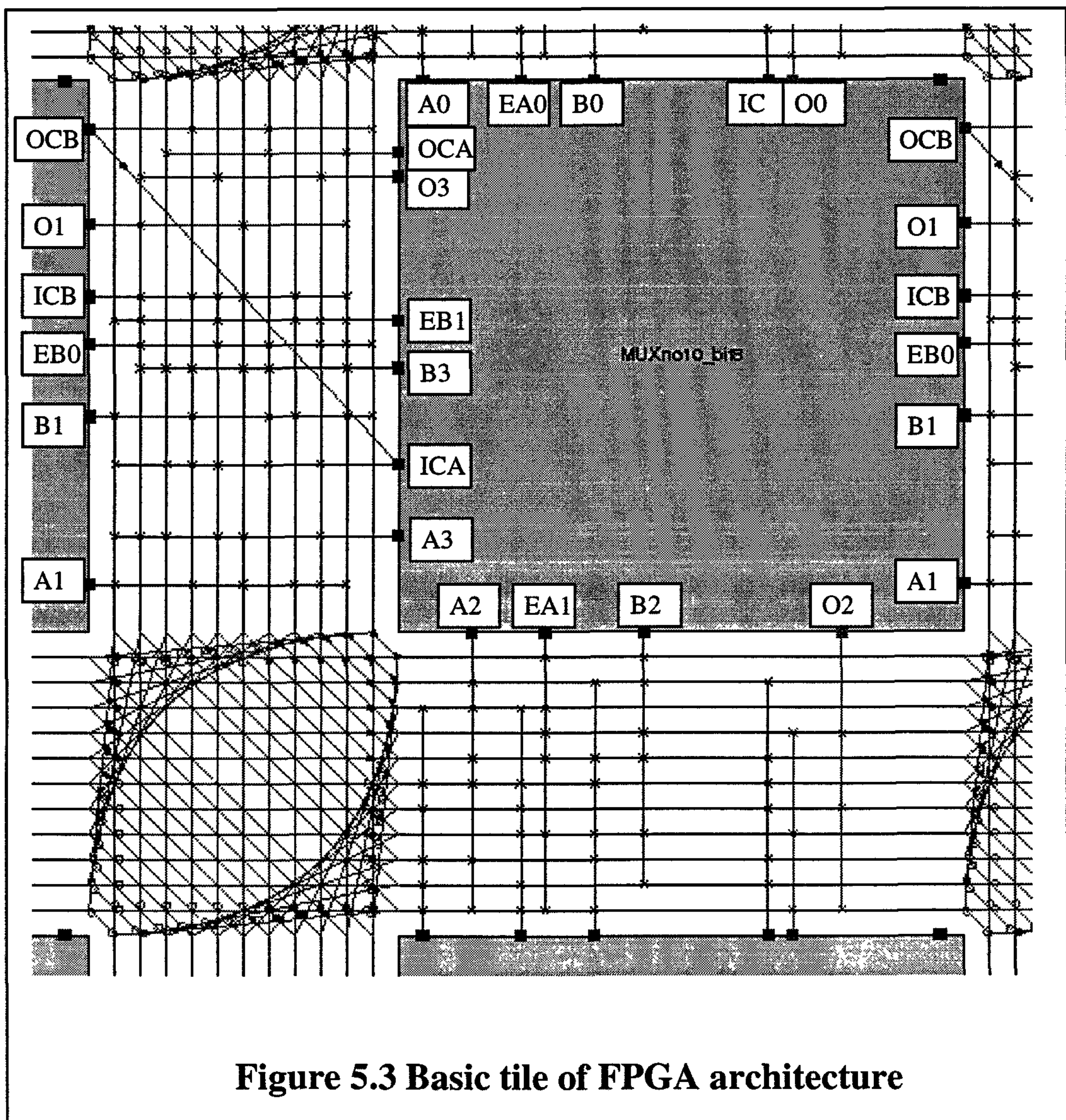


Figure 5.3 Basic tile of FPGA architecture

Physical parameters. These parameters do not influence the topology of interconnections, but they are important for delay and transistor area estimations done by VPR. Some of them are listed below.

Table 5.1. Detailed routing architecture parameters.

Wire segments	50% are pass-transistor switched 50% are tristate-buffer switched
Routing switches	**1) pass-transistor switch strength = 10** 2) buffer switch strength = 5 3) pass-switched track segments are driven by buffers of strength 10

In the architecture specification, resistance and capacitance values of the switches and wire segments are listed. The numbers were originally based on analog-simulation experiments with 0.35μ TSMC's process [Betz99 p.154]. However, in the publicly open sample file, these parameters have been changed to protect the foundry. The authors declared them to be still reasonable enough to allow FPGA experimentation.

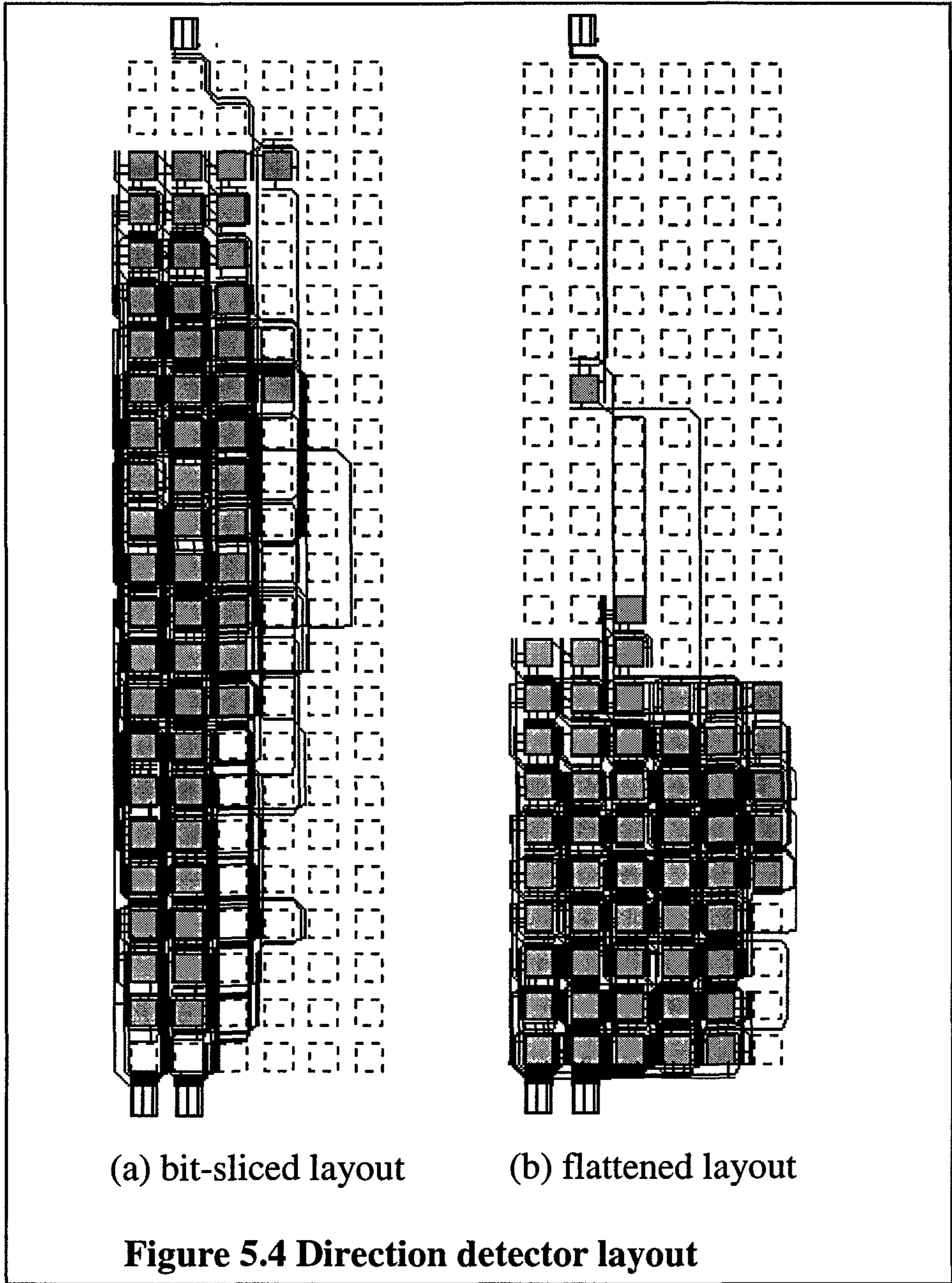
5.4 Results

The netlist file of the direction detector benchmark contains 55 netlist blocks, 16 data inputs (for two 8 bit operands) and two outputs (for the direction encoding). The netlist blocks are organized in 20 chains, corresponding to 20 macros of the circuit, namely: 4 flip-flop registers, 4 subtractors, 4 multiplexers, 4 comparators, and 4 absolute value calculators. It has been placed and routed on a 6x23 FPGA array.

Figure 5.4 shows examples of bit-sliced and '*flattened*' layout for the direction detector benchmark. We call the layout *flattened* if it allows any block to be placed at any location. The flattened layout in Figure 5.4 has been produced by the default T-VPlace algorithm of VPR. The bit-sliced layout example has been produced by the constrained T-VPlace algorithm.

The inputs have been placed on the bottom I/O pads. In accordance with bit-sliced placement conventions, 8-bit inputs are locked to I/O pads in the first two columns. The I/O pad locking models the situation when the fine-grained architecture is a ASU unit inside the fixed layout of a processor, so the placer algorithm cannot select the location of I/O pads. Not only constrained, but also the default algorithm placed the netlist with the locked I/O pads.

The outputs have been placed on the top I/O pads. In this benchmark, we have two control bits at the output, they have been locked to some arbitrary I/O pads on the top.



(a) bit-sliced layout (b) flattened layout

Figure 5.4 Direction detector layout

Table 5.2 shows the results of several runs of placement and routing.

Table 5.2 Default vs. constrained placement

	Default			Constrained		
	W_{MIN}	τ_{CRIT}	Area	W_{MIN}	τ_{CRIT}	area
$L = 1$	13	74	9452	12	75	8749
$L = 2$	14	79	9227	14	68	9227
$L = 4$	23	88	14191	19	69	11902

Three parameters are shown for each run, namely: minimum channel width, critical path delay (in nanoseconds), and the FPGA tile area measured in minimal transistor areas. Three versions of the architecture have been considered with different wire segment lengths: $L = 1, 2,$ and 4. The total number of block moves performed by simulated annealing in both cases was 200 - 300 thousand.

Comparing channel widths and critical path values for different placement algorithms, we see that, for this benchmark, the constrained algorithm performed better, especially in the case when segments were longer. This can be explained by the fact that the bit-sliced layout makes use of the great number of long vertical connections that join macros lying at different rows.

The effect of the longer wire segments on the transistor area is twofold. On one side, it reduces the number of switches in the switch boxes (because many segments just cross the switch box eliminating the need on the switches that bridge the tracks over the switch boxes). On the other side, the flexibility of the routing decreases. From Table 5.2, one can see that, depending on the placement algorithms, FPGA tile with the smallest area can be used if $L = 1$ or 2.

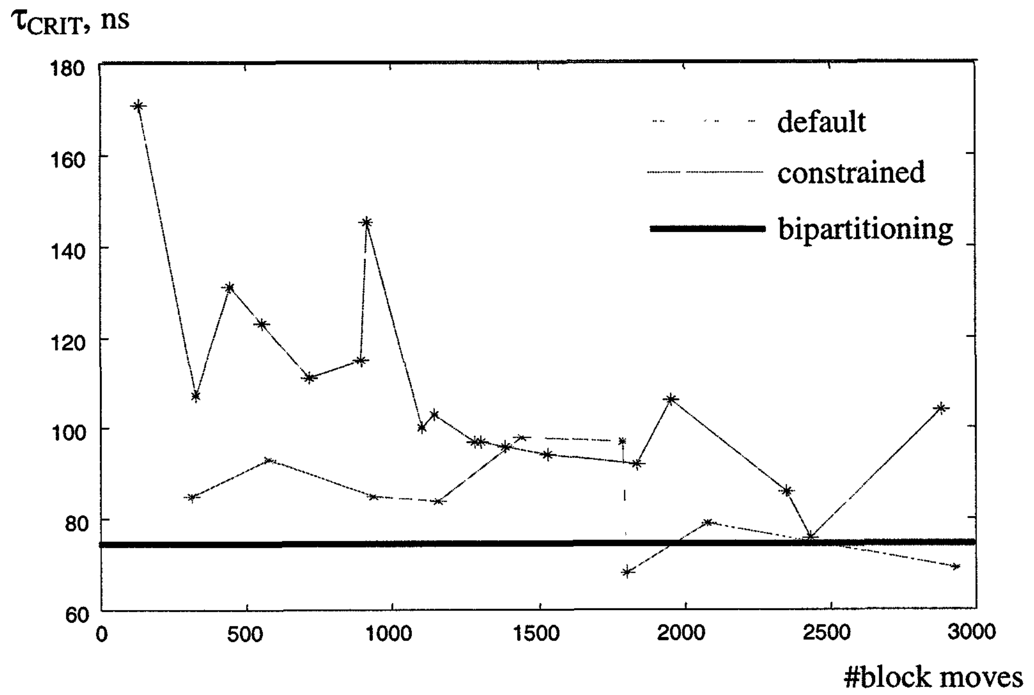
Table 5.3 repeats the results obtained with the default placement algorithm and shows the results of the bipartitioning placement.

Table 5.3 Default vs. Bipartitioning placement

	Default			Bipartitioning		
	W_{MIN}	τ_{CRIT}	Area	W_{MIN}	τ_{CRIT}	Area
$L = 1$	13	74	9452	13	99	9452
$L = 2$	14	79	9227	14	75	9227
$L = 4$	23	88	14191	18	76	11240

Bipartitioning placement has produced layouts with the critical paths inferior to the constrained algorithm, because it is not a timing-driven algorithm. However, it also appears to perform better than the default placement with the longer wire segments.

Because simulated annealing algorithms are iterative optimization algorithms, to make a fair comparison between the default and constrained default algorithm, one should compare their *convergence* properties.



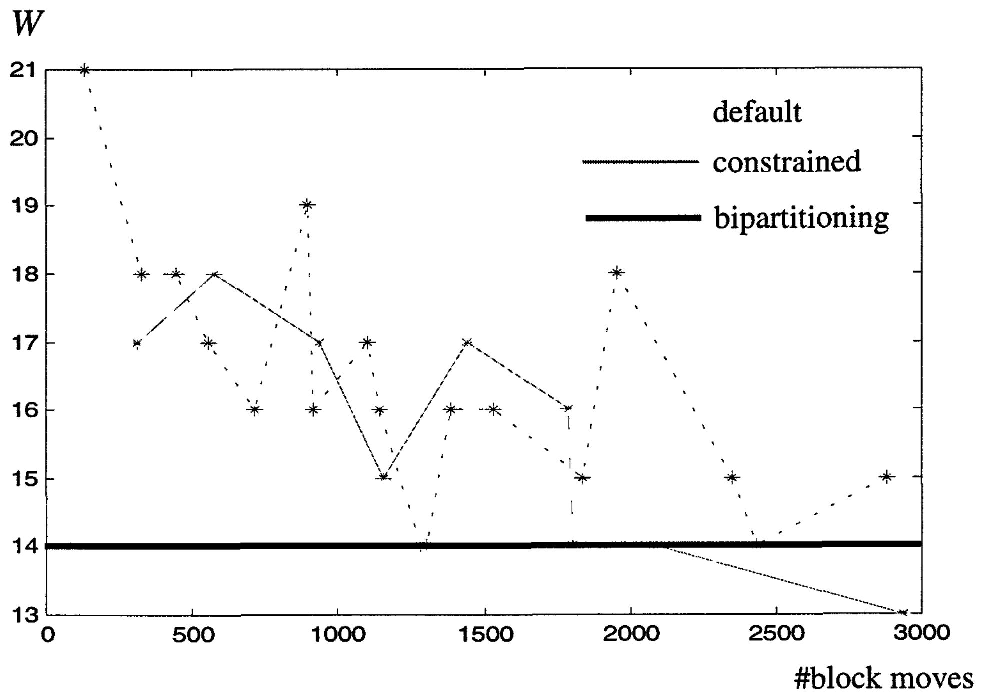


Figure 5.4 Placement algorithm convergence (*L*=2)

Using a command line option of VPR, one can regulate the number of algorithm's inner loop iterations per temperature, in other words, one can set *optimization effort*. Figure 5.5 shows the graphs for critical paths and channel widths obtained by default and constrained simulated annealing algorithms with different optimization effort settings. The experiments have been performed with the wire segment length $L = 2$. The effort is measured by the total number of *block moves*. For the default algorithm, it is the same as the number of iterations; for the constrained algorithm it is the number of iterations multiplied by the average number of blocks per macro. The results obtained by bipartitioning algorithm are shown with the horizontal bold line. The runtime of bipartitioning was faster than that of the simulated annealing algorithms with the smallest optimization effort.

From the graphs for critical path, one can see that the constrained algorithm could obtain a good layout in terms of critical path, already at the small levels of effort. As for channel widths, it is hard to say whether one algorithm is better than another. From both graphs, one can conclude that bipartitioning algorithm, although having spent much less time on computations, have produced the results of reasonable quality.

So, for this benchmark, the algorithms that we have introduced in VPR (constrained and bipartitioning) have advantages over the default placement algorithm in terms of quality and computation time.

5.5 Summary

In this chapter, we have introduced a datapath-oriented benchmark by a specification in high-level language. Using our C++ library, we have converted it to a VPR-format netlist, oriented to new logic block. Using VPR, we placed and routed that benchmark several times using different placement algorithms. From these experiments we have seen that the bit-sliced placement algorithms discussed in Chapter 4 have advantage over the default placement in terms of quality and computation time.

6. Conclusions

The work presented in this Thesis was motivated by the importance of the fine-grained reconfigurable computing architectures, that can considerably accelerate the execution of some applications by programmable processors if used as processors' data-processing units. The recipe for architecture development contains three main components, namely: the *benchmark applications*, the *architecture template* and the *design tools*.

Fine-grained reconfigurable architectures are inspired by the ideas used in field-programmable gate arrays (FPGA), so we used the term FPGA as a synonym for fine-grained architectures. A framework for FPGA development, including all three components, has been elaborated and opened for the public use by Betz et al at the University of Toronto. The central place in that framework is occupied by the design tool called VPR.

In our view, the VPR-based framework was too general-purpose, and we decided to optimize it for a particular application domain we were interested in, namely, for digital signal processing (DSP). DSP applications make use of regular datapaths, so, our efforts have been mainly spent on adding special support for regular datapath circuits for all three framework components mentioned above.

The contributions of this Thesis can be divided into several main topics, discussed in three following sections.

6.1 Routing Architecture Elements

1. Configurable logic blocks connect to the routing channels by patterns of switches, called connection boxes. To save area, *sparse* switch patterns can be used. We proposed an algorithm that generates switch patterns of slightly higher quality than VPR's default switch patterns.
2. Channel tracks are joined with one another by means of switch boxes. A *switch box with half number of switches* of the usual switchbox has been considered. We have extended VPR's source code to include this option. It has been shown that the 'half switch box' is area-efficient for short wire segments (length 1 and 2).
3. *Direct connections* join adjacent logic blocks to make local connection faster and to save routing tracks. The source code of VPR has been extended to include direct connections. It has been shown that a single direct connection leads to a slight reduction of the required number of tracks to route circuits. We have also used direct connections for fast carry propagation in the implementations of arithmetic circuits.

6.2 Modeling New Logic Block

The new logic block proposed by Ir. K.Leijten-Nowak for fine-grained reconfigurable computing has been modeled in VPR, so that benchmark circuits can be placed and routed on the architectures based on that block.

6.3 Datapath mapping and placement:

1. An initial framework has been established for *mapping the datapath-based circuits* to the desired FPGA architecture, starting from structural specification in a high-level language.
2. Using that framework, a datapath-oriented *benchmark circuit*, called 'direction detector', has been mapped on an FPGA architecture based on the new logic block.
3. To obtain good layout for datapath in less computation time, one can make use of the *regularity of the datapath.* To preserve regularity, blocks must be arranged in macros, or, in other words, relative location constraints must be applied.
4. Bit-sliced placement has been introduced into VPR. It is a placement approach that makes use of the regularity of datapath. The corresponding optimization problem is finding an optimal *linear arrangement* of datapath macros placed in the array's rows. Bit-sliced placement has been implemented using two algorithms: a 'constrained' default algorithm of VPR and successive bipartitioning, a totally new algorithm for VPR. Using 'direction-detector' benchmark these algorithms have been examined in comparison to each other and to default placement algorithm.
5. Bit-sliced placement is an example of using the knowledge about the designed application at the design time. Compared to the general-purpose design approach, e.g., VPR's default placement algorithm, the dedicated-purpose design approach can yield better solutions at the same computation time or the same quality in less computation time. The results have shown that this holds also for the bit-sliced placement algorithms.

6.4 Proposals for the Future Work

1. The proposed algorithm for connection box generation is random. A 'smarter' connection box generator that would exclude random choice can be expected to produce connection boxes of a higher quality.
2. The architecture generator of VPR has to be improved to support logic block modes. It has to generate the logic-block subgraphs of routing resource graph depending on the mode of the netlist block placed at the given location.

3. Only one benchmark has been mapped to new-logic-block-based architectures, so, based on our datapath-mapping framework, other benchmarks should be mapped.

4. Combined *slice-and-nlb placement* is required for efficient placement of random-logic on the new logic block architecture. VPR supports only swapping of entire *nlb*s.

5. Currently implemented bit-sliced-placement algorithm considers control blocks as part of macros they control. Control should be treated as random logic and a combined *datapath/random logic placement* should be developed.

6. Direction detector benchmark is based on a *delay line*, as many other DSP benchmarks can be expected to be. Assuming that the delay line is outside FPGA increases the number of the required input pads. It also leads to the requirement for the peripheral I/O-pad routing channels on the edges and wider vertical routing channels that have to bring the data from I/O channels inside FPGA. On the other hand, implementing the delay line on FPGA will require the use of extra logic blocks. The question which option is better seems to be important for DSP applications and still has to be investigated.

7. Because new logic block is rich in flip-flops, the placement algorithm should be enhanced with the ability to replace flip-flops to improve throughput (*retiming*). That idea has been used in previous research [Call98b].

8. Bit-sliced placement leads to non-square layouts; it uses much more rows than columns of FPGA. So it may be expected that such placer can run out of the rows when there is still free space available on FPGAs. So, techniques should be considered how to place more than one macro per row.

Appendix 1. The routing area estimation in VPR.

The area reported in the tables is the area measured in minimum-width transistors, sometimes denoted as λ^2. The total routing area divided by the number of logic blocks is reported, so it specifies roughly the area of one FPGA tile. The total routing area is calculated by scanning every edge of the routing graph that represents a switch in a switchbox and accumulating the switch area. The other part of the routing area is the area of multiplexers and buffers that implement input connection boxes. Because VPR assumes the implementation of tristate buffer as a chain of inverters followed by a pass transistor, two values are actually reported by VPR: the pessimistic and optimistic area. The first one assumes that every switch has its own inverter chain and the second one assumes that e.g. three tristate-buffer switches that go from the same track segment in the switch box share their inverter chains. In the tables only the results based on pessimistic area estimation are reported. Optimistic results are around 1,5 times smaller (this ratio varies from architecture to architecture). The calculation of every single switch area is based on the relation of the switch resistance to the n-MOS and p-MOS resistance values taken from the architecture specification file. The complementary transistors in the inverter chains are assumed to be sized so that they have equal resistance when open. The smallest inverter in the chain is assumed to have the minimal-sized n-MOS transistor. The stage ratio of 4 is assumed. Note, *logic block* area is not included in the routing area. Betz et al has estimated the area of a size 4 logic cluster as 1678 min-width transistors [Betz99 p.219].

Appendix 2. Direct connections for random logic.

The experiments have been conducted on architecture based on size 4 logic cluster, wire segment length $L = 4$ and the sparse connection box. 3 MCNC benchmarks have been placed and routed on that architecture to compare the results in the '*normal*' case (without direct connections) and with direct connections.

1. Single direct connection along columns

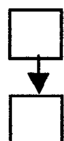


Figure 1a
"SOUTH1"

Figure 1b
"SOUTH1B"

Figure 1 (a) and (b) shows two cases considered for direct connections: "SOUTH1" when a single direct connection goes from an output at the bottom to an input at the top and "SOUTH1B" when it goes from an output at the top to an input on the bottom. Table 1 shows the results for the "SOUTH1" case.

Table1. Results: SOUTH1.

	Global routing		Detailed routing		
	Normal	Direct	Normal	Direct	Direct at W_{NORMAL}
Alu4 20x20	W=17	W=18	W=30 Time: 48 Area: 4397 λ^2	W=28 Time: 65 Area: 4157 λ^2	W=30 Time: 45 Area: 4403
Apex4 19x19	W=21	W=21	W=36 Time: 53 Area: 5258 λ^2	W=35 Time: 73 Area: 5141 λ^2	W=36 Time: 58 * **Area: 5265**
Clma 47x47	W=25	W=24	W=46 Time: 210 Area: 6157 λ^2	W=44 Time: 238 Area: 5933 λ^2	W=46 Time: 126** Area: 6164

*, ** in the table indicate unexpected results.

The table shows the channel width, the critical path delay, and the transistor areas. The left part shows that the global routing does not indicate any profit from direct connections. When doing global routing, the router has the greatest routing freedom, so, the decrease in track demand due to the direct connection does not influence the result much. In the case of length 4 wire segments, short connections between blocks without direct connections must use long segments, so, the improvement of routability is to be seen (compare the columns 'normal' and 'direct'). To make critical path comparisons fair, additional experiment with the availability of direct connections has been done, namely, routing on the wider channel width, equal to the minimum width obtainable without direct connections (see the last column). In two cases, surprising results have been obtained, which indicate the problems that the router has with minimizing the timing cost of the circuit. In first case, the critical path has increased with direct connections, and in the second case it has decreased very much. It can be explained by the fact that VPR's router stops looking for alternative routing paths as soon as he has found a feasible routing. It tries to minimize the critical path only as long as it looks for a feasible routing; once a feasible routing has been found it stops immediately without continuing the attempts to optimize the critical paths.

Table 2, when compared to Table 1, shows that the case 'B' is not so favorable as the previous case.

Table2. SOUTH1-B (detailed routing)

	Normal	Direct
Alu4	W=30 Time: 48	W=29 Time: 59
Apex4	W=36 Time: 53	W=36 Time: 58
Clma	W=46 Time: 210	W=45 Time: 226

2. Mutiple direct connections.

One more direct connection, in the east direction has been added. Table 3 shows the results:

Table3. Results: South-and-east connections

	Normal	Direct: P&R	Normal: P; Direct: R
Alu4	W=30	W=31	W=29
Apex4	W=36	W=38	W=36

The column 'Direct: P&R' shows the case when the architecture with direct connections has been used for both placement and routing, like in the previous tables. We see that the introduction of second direct connection has not lead to improvement, but to the worse figures. It was very surprising to see the worse results for an architecture with extra connections, so we explained it by unfavorable placement. So, we used the placement with the normal architecture and routing has been performed with the direct connections (see the last column), and obtained the results that are at least not worse than in the normal case.

Although experiments with more benchmarks have to be performed for more convincing results, we think that direct connections can strongly influence the quality of the default placement algorithm and either improve it or deteriorate it.

Appendix 3. Architecture Specification Used in Experiments with Direction Detector Benchmark

```
# NB:  The timing numbers in this architecture file have been
modified
# to comply with our NDA with the foundry providing us with process
# information.  The critical path delay output by VPR WILL NOT be
accurate,
# as we have intentionally altered the delays to introduce
inaccuracy.
# The numbers are still reasonable enough to allow CAD
experimentation,
# though.  If you want real timing numbers, you'll have to insert
your own
# process data for the various, R, C, and Tdel entries.

# Architecture with two types of routing segment.  The routing is
# fully-populated.  One length of segment is buffered, the other uses
pass
# transistors.

# Uniform channels.  Each pin appears on only one side.
io_rat 24
chan_width_io 1
chan_width_x uniform 1
chan_width_y uniform 1

# New logic block
inpin  class: 0  top    #A0
inpin  class: 1  right  #A1
inpin  class: 2  bottom #A2
inpin  class: 3  left   #A3
inpin  class: 4  top    #EA0
inpin  class: 4  bottom #EA1
inpin  class: 5  left   east #ICA

inpin  class: 6  top    #B0
inpin  class: 7  right  #B1
inpin  class: 8  bottom #B2
inpin  class: 9  left   #B3
inpin  class: 10 right  #EB0
inpin  class: 10 left   #EB1
inpin  class: 11 right  #ICB
inpin  class: 12 top    #IC

outpin class: 13 top    #O0
outpin class: 13 right  #O1
outpin class: 13 bottom #O2
outpin class: 13 left   #O4

outpin class: 14 left   #OCA
outpin class: 15 right  east #OCB

inpin  class: 16 global top #CLK


# Class 0-12 -> logic inputs, Class 13-15 -> Outputs, Class 16 ->
clock.
```

```
subblocks_per_clb 15
subblock_lut_size 6

#parameters needed only for detailed routing.
switch_block_type subset
Fc_type fractional
Fc_output 1
Fc_input 1
Fc_pad 1

# All comments about metal spacing, etc. assumed have been deleted
# to protect our foundry.  Again, the R, C and Tdel values have been
# altered from their real values.

segment frequency: 0.5 length: 1 wire_switch: 0 opin_switch: 1
Frac_cb: 1. \
         Frac_sb: 1. Rmetal: 8.316 Cmetal: 162e-15

segment frequency: 0.5 length: 1 wire_switch: 2 opin_switch: 2
Frac_cb: 1. \
         Frac_sb: 1. Rmetal: 8.316 Cmetal: 162e-15

# Pass transistor switch.

switch 0  buffered: no  R: 196.728 Cin: 20.574e-15  Cout: 20.574e-15
Tdel: 0

# Logic block output buffer used to drive pass transistor switched
wires.

switch 1  buffered: yes  R: 393.47 Cin: 7.512e-15  Cout: 20.574e-15 \
       Tdel: 524e-12

# Switch used as a tri-state buffer within the routing, and also as
the
# output buffer used to drive tri-state buffer switched wires.

switch 2  buffered: yes  R: 786.9 Cin: 7.512e-15  Cout: 10.762e-15 \
       Tdel: 456e-12

# Used only by the area model.

R_minW_nmos 1967
R_minW_pmos 3738

# Timing info below.  See manual for details.

C_ipin_cblock 7.512e-15

T_ipin_cblock 1.5e-9

T_ipad 478e-12    # clk_to_Q + 2:1 mux
T_opad 295e-12    # Tsetup

T_sblk_opin_to_sblk_ipin 0
T_clb_ipin_to_sblk_ipin  0
T_sblk_opin_to_clb_opin  0

# Delays for each of the 15 sequential and combinational elements

# "nADDXOR"
```

```
T_subblock T_comb: 80e-12 T_seq_in: 100 T_seq_out: 100    # gigantic
value for register, because there are none

################### Slice A model subblocks
############################
#  "A_MXLUT0", "A_MXLUT1"
T_subblock T_comb: 550e-12 T_seq_in: 845e-12 T_seq_out: 478e-12
T_subblock T_comb: 550e-12 T_seq_in: 845e-12 T_seq_out: 478e-12

# "A_CARRY0", "A_CARRY1",
T_subblock T_comb: 200e-12 T_seq_in: 100 T_seq_out: 100
T_subblock T_comb: 300e-12 T_seq_in: 100 T_seq_out: 100

#  "A_ANMXLUT0", "A_ANMXLUT1"
T_subblock T_comb: 700e-12 T_seq_in: 845e-12 T_seq_out: 478e-12
T_subblock T_comb: 700e-12 T_seq_in: 845e-12 T_seq_out: 478e-12

# "A_MUXN"
T_subblock T_comb: 120e-12 T_seq_in: 100 T_seq_out: 100    # gigantic
value for register, because there is none

################### Slice B model subblocks
############################
#  "B_MXLUT0", "B_MXLUT1"
T_subblock T_comb: 550e-12 T_seq_in: 845e-12 T_seq_out: 478e-12
T_subblock T_comb: 550e-12 T_seq_in: 845e-12 T_seq_out: 478e-12

# "B_CARRY0", "B_CARRY1",
T_subblock T_comb: 200e-12 T_seq_in: 100 T_seq_out: 100
T_subblock T_comb: 300e-12 T_seq_in: 100 T_seq_out: 100

#  "B_ANMXLUT0", "B_ANMXLUT1"
T_subblock T_comb: 700e-12 T_seq_in: 845e-12 T_seq_out: 478e-12
T_subblock T_comb: 700e-12 T_seq_in: 845e-12 T_seq_out: 478e-12

# "B_MUXN"
T_subblock T_comb: 120e-12 T_seq_in: 100 T_seq_out: 100    # gigantic
value for register, because there is none
```

References

[Atmel99] *AT40K Series Reconfigurable DSP Co-Processor FPGAs with FreeRAM™*, Atmel's Figaro™ tool, help file.

[Betz99] V. Betz, J. Rose, and A. Marquardt, *Architecture and CAD for Deep-Submicron FPGAs*, Kluwer Academic Publishers, February 1999.

[Call98a] Timothy J. Callahan and John Wawrzynek, *Instruction-level Parallelism for Reconfigurable Computing*. Proc. 8th *International Workshop on Field Programmable Logic and Applications* FPL'98, Springer-Verlag, Tallinn, Estonia, Aug-Sep 1998.

[Call98b] Timothy J. Callahan, Philip Chong, André DeHon, and John Wawrzynek, *Fast Module Mapping and Placement for Datapaths in FPGAs*. Proc. ACM/SIGDA International Symposium on Field Programmable Gate Arrays FPGA'98, ACM, Monterey, CA, February 1998, pp. 123-132.

[Cheng94] C. Cheng, RISA: Accurate and Efficient Placement Routability Modeling. Proc. ICCAD, 1994, pp. 690-695.

[Chow99] P. Chow, S. Seo, J. Rose, K. Chung, I Rahardja, and G. Paez, *The Design of an SRAM-Based Field-Programmable Gate Array: Part II: Circuit Design and Layout*. IEEE Transactions on VLSI, Vol. 7 No. 3, Sept. 1999, pp. 321-330.

[DeHon00] A. DeHon, *The density advantage of configurable computing*. IEEE Computer, Volume: 33 Issue: 4, April 2000, pp. 41-49.

[Emm98] John M. Emmert, Akash Randhar and Dinesh Bhatia, *Fast floorplanning for FPGAs*. Proc. 8th International Workshop on Field Programmable Logic and Applications FPL'98, Springer-Verlag, Tallinn, Estonia, Aug-Sep 1998, pp. 129-138.

[Emm99] John M. Emmert and Dinesh Bhatia, *A Methodology for Fast FPGA Floorplanning*. Proc. ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'99), ACM, Monterey, CA, February 1999, pp. 47-56.

[Goer00a] Richard Goering, *Compiler project marks Synopsys' step into post-ASIC world*. EE Times, 28 August 2000. http://www.eet.com/story/OEG20000828S0020

[Goer00b] Richard Goering, *EDA may fill software gap*. EE Times. 11 Sep 2000. http://www.eet.com/story/OEG20000911S0011

[Goto81] Satoshi Goto, *An efficient algorithm for the two-dimensional placement.* IEEE Transactions on Circuits and Systems, Vol. CAS-28, No. 1, January 1981.

[Huij96] Ed Huijbregts, *Complete Design Path of Flexible Macros.* Ph.D. Thesis, Eindhoven University of Technology, 1996.

[Imran99] M. Imran Masud, Steven J.E. Wilton. *A New Switch Block for Segmented FPGAs.* Proc. 9th International Workshop on Field Programmable Logic and Applications FPL'99, Springer-Verlag, Glasgow, Scotland, Aug. 1999.

[Imran_MSc] M. Imran Masud, *FPGA Routing Structures: A Novel Switch Block and Depopulated Interconnect Matrix Architecture.* M.A.Sc. Thesis, The University of British Columbia, December 1999.

[Koch96] Andreas Koch, *Structured Design Implementation - A Strategy for Implementing Regular Datapaths on FPGAs.* Proc. ACM/SIGDA International Symposium on Field Programmable Gate Arrays FPGA'96, ACM, Monterey, CA, February 1996.

[Lem00] Guy Lemieux, Paul Leventis, David Lewis, *Generating Highly-Routable Sparse Crossbars for PLDs.* Proc. FPGA-2000, pp. 155-164.

[Len90] Thomas Lengauer, Combinatorial algorithms for integrated circuit layout. Wiley, 1990.

[Marq00] A. Marquardt, V. Betz, and J. Rose, Timing-Driven Placement for FPGAs. In *Proc. ACM/SIGDA International Symposium on Field Programmable Gate Arrays FPGA'2000*, ACM, Monterey, CA, February 2000, pp. 203-213

[Now01a] K. Leijten-Nowak. (Internal communication; the conference paper in preparation).

[Now01b] K. Leijten-Nowak, J. van Meerbergen, *Applying the Adder Inverting Property in the Design of Cost-efficient Reconfigurable Logic.* Accepted for IEEE 44th Midwest Symposium on Circuits and Systems, August 2001, USA.

[Rose90] J.S. Rose, R.J. Francis, D. Lewis, and P. Chow, *Architecture of Field-Programmable Gate Arrays: The Effect of Logic Block Functionality on Area Efficiency.* IEEE J. Solid-State Circuits, Vol. 25 No. 5, October 1990, pp. 1217-1225.

[Sank99] Yaska Sankar and Jonthan Rose. *Trading Quality for Compile Time: Ultra-Fast Placement for FPGAs.* Proc. ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'99), ACM, Monterey, CA, February 1999.

[Stoh98] Jörn Stohmann, Klaus Harbich, Markus Olbrich, Erich Barke. *An Optimized Design Flow for Fast FPGA-Based Rapid Prototyping.* Proc. 8th *International Workshop on Field Programmable Logic and Applications* FPL'98, Springer-Verlag, Tallinn, Estonia, Aug-Sep 1998, pp. 79-88.

[Tess99] Russel Tessier, *Frontier: A fast placement system for FPGAs.* Proc. 10th IFIP Conference on VLSI, MIT, 1999.

[Tog94] Nozomu Togawa, Masao Sato, Tatsuo Ohtsuki. *A Simultaneous Technology Mapping, Placement and Global Routing Algorithm for FPGAs.* Proc. ISCAS'94, pp. 483-486, 1994.

[Trim94] *Field programmable gate array technology* ed. by Stephen M. Trimberger. Kluwer Academic, 1994.

[Xilinx] Xilinx Alliance Tools 'Watch' Tutorial, www.xilinx.com

[VPR00] Vaughn Betz. *VPR user manual.* March, 2000.