ELSEVIER

INTEGRATION, the VLSI journal 17 (1994) 191-240

INTEGRATION the VLSI journal

INTEGRATION Report

A survey of DA techniques for PLD and FPGA based systems

R. Venkateswaran, P. Mazumder*

Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109-2122, USA

Received 10 January 1994

Abstract

Programmable logic devices (PLDs) are gaining in acceptance, of late, for designing systems of all complexities ranging from glue logic to special purpose parallel machines. Higher densities and integration levels are made possible by the new breed of complex PLDs and FPGAs. The added complexities of these devices make automatic computer aided tools indispensable for achieving good performance and a high usable gate-count. In this article, we attempt to present in an unified manner, the different tools and their underlying algorithms using an example of a vending machine controller as an illustrative example. Topics covered include logic synthesis for PLDs and FPGAs along with an in-depth survey of important technology mapping, partitioning and place and route algorithms for different FPGA architectures.

Key words: Programmable logic devices, FPGAs, Logic synthesis, State assignment, Technology mapping, Routing

Contents:

* Corresponding author.

1. Introduction

The original goal of programmable logic devices was to allow the designer to implement more complex logic and state machines with dramatically fewer components than is possible with discrete *fixed-function* logic such as the 7400-series; simultaneously retaining all the advantages of the latter such as short design cycles, low development costs and less reliance on specialized skills. Despite the numerous competing alternatives, most of the earlier PLDs are based on an AND-OR plane architecture. The inputs to the AND/OR planes are driven directly by dedicated input pins of the device and some user-selectable input/output pins or feedback paths. The outputs are driven directly from sum-of-product logic outputs or from flip-flops. This general arrangement leads to efficient realizations of sum-of-product Boolean equations. For every input variable in the Boolean equations, there is an input signal to the AND array and for every output there exists a signal emanating from the OR array. Depending on the flexibility of the AND/OR arrays, these simple PLDs have been classified as either programmable read-only memories (PROMs), programmable array logic (PAL) or programmable logic array (PLA) devices, see Table 1.

The primary limitation of the above architecture is the restricted nature of the AND/OR plane and the dedicated nature of the interconnections. Gate utilization seldom exceeds 15% and a practical limit on the number of usable gates is in the low hundreds. Performance is also fixed for each level

Table 1

Class	AND array	OR array	Best suited for
PROMs	Fixed	Programmable	Dense functions e.g. code converters
PAL's	Programmable	Fixed	Sparse functions with little or no common product terms. Small and fast.
PLA's	Programmable	Programmable	Sparse high-fanin functions with potentially lots of shareable common product terms. Compact and quite general.

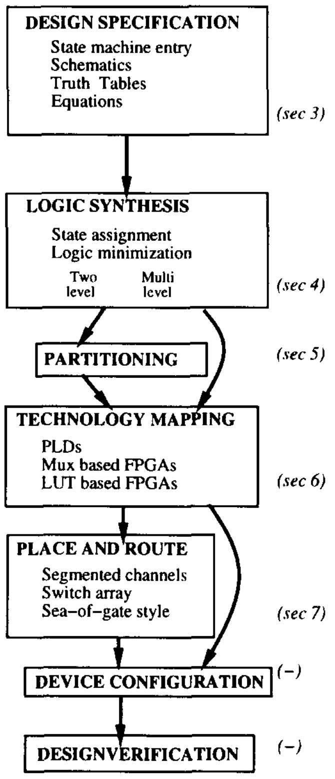


Fig. 1. Major steps in design of PLDs.

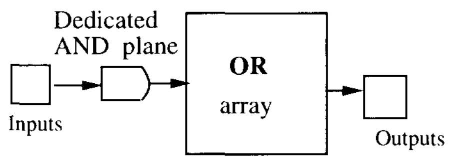
of logic. Each path through the AND/OR plane takes about 25–45 ns. Typically, these devices are used for realization of two-level logic such as glue logic, interface logic, decoders and so on.

However, with advances in integration levels, a new breed of PLDs have emerged. These are of two main types (i) Complex PLDs incorporating multiple two-level PLDs on a single device with some sort of either hardwired or programmable switching matrix interconnecting them. (ii) FPGAs (field programmable gate arrays) which are devices whose cores are populated with an array of logic structures of varying granularity and programmable interconnect provided to connect them in a variety of different ways. For instance, the logic blocks can be SRAM-based look-up tables or even multiplexers with or without registers while special purpose routing switchboxes or segmented channels can constitute the programmable interconnect. The main advantages of FPGAs are their high densities (approaching 10 000 usable gates per device), lower quantity production costs, faster turnaround design time, design security, and multiple usage possibilities for the same device in field through reprogramming. Consequently, they have started making serious inroads into the higher-density semi-custom ASIC markets which is currently dominated by mask-programmed gate arrays. Unlike simple PLDS, FPGA design requires the availability of powerful logic synthesis software capable of supporting multi-level synthesis; mapping algorithms to configure the logic blocks appropriately; and good place and route software to make the interconnections efficiently both in terms of wiring length as well as timing considerations.

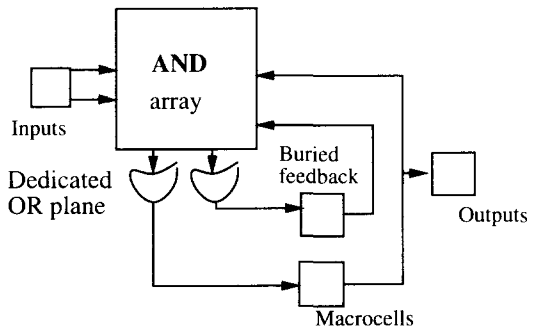
The purpose of this paper is to review some of the major concepts in system design using programmable devices, roughly reflecting the state of the art in the early 1990's. More importantly, we identify in an unified manner the pertinent issues of this design process, the tools available, the nature of the algorithms and their application. A roadmap of the organization of the rest of the paper is shown in Fig. 1 where the numbers besides the boxes indicate the section numbers where that topic is discussed in this paper. Our self-imposed focus throughout this paper will be on the design issues and algorithms. Consequently, we cover technological details only to the extent that they affect the algorithms. Recent books on programmable devices [1-4] provide more information on implementation details of specific devices. We also do not cover architectural studies of FPGAs in this paper (see [5,6]). Also, we will not be covering design verification issues.

2. Review of PLD architectures

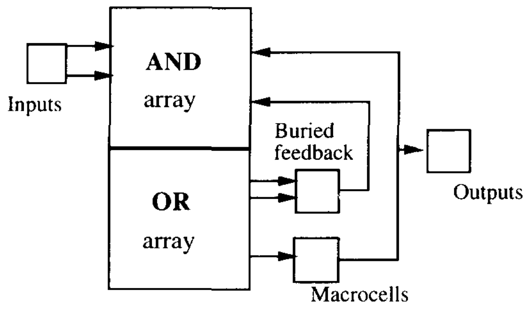
Before we take a look at the algorithms themselves, it will be useful to make a brief review of the PLD architectures. The architectures of different PLDs (and there are more than 300 of these



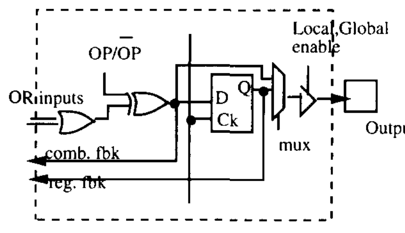
(a) ROM model with programmable OR array



(b) PAL model with dedicated OR plane. Each OR gate typically has between six and eight inputs.



(c) PLA model with programmable AND and OR arrays.



(d) Sample macrocell organization showing combinational as well as registered outputs and feedback, outputs in true or complemented form, output enable etc.

Fig. 2. PLD classification.

in existence) can directly affect their suitability for a particular application. The basic distinguishing feature is the *size* which is a function of the number of input and output pins, the number of product terms and so on. However, other features of the PLDs can also affect the complexity considerably. Figure 2 depicts the various types of basic PLD structures that were elided to in the introduction.

Historically speaking, the first PLDs were TTL-based PALs such as the 16L8 from MMI (now part of AMD). These provided only combinational outputs. Later registered outputs, the ability to control the output polarity, feedback structures, etc were added. Yet, a particular pin was restricted to playing a single role. The next type of PLDs, referred to as Generic PALs, typified by the 22V10 device from AMD and the GAL (generic array logic) from Lattice attempted to eliminate this hardwired asymmetricity that existed between combinational and registered outputs by placing a common logic block, made up of small standard cells such as flipflops, buffers, XOR gates, etc., between the I/O pins and the AND/OR array. This structure, called a *macrocell* could be programmed by the user to equip a particular pin with the appropriate functionality. The next advancement was in the availability of reprogrammable PLDs beginning with the PEEL devices. These employed Fowler–Nordheim tunneling techniques to trap charges onto a floating gate through a thin oxide insulator. The charges remain trapped even after power is removed allowing non-volatility of programmed data, but could be removed by electrically erasing the device. Once fully erased, the device could then be reprogrammed into a new configuration. This was followed by the advent of CMOS technology such as the CPL devices from Samsung. The advantages of CMOS include low power requirements, low propagation delay, greater noise margins, and also greater integration possibilities. Most CMOS PLAs use a NAND/NAND array instead of the traditional AND/OR array since NAND gates place the faster n-channel devices in series whereas the NOR form would result in having the slower p-channel devices in series. From Boolean theory, it is easy to see that a two level AND-OR design is the same as a two-level NAND-NAND structure. Hence in CMOS PLAs, the Boolean equations are first converted to their equivalent NAND forms. Recently, some PLDs based on GaAs are also emerging which are targeted towards faster clock rates.

2.1. Architectural variations for simple PLDs

2.1.1. Multi-level structures

AND/OR logic is good for two-level designs. Sometimes for high fanins, it may be required to decompose a function into a multi-level form. Since, any external feedback uses precious pads, a different type of device architecture called PML (Programmable Macro Logic) was introduced by Signetics. PML devices are unique in that they contain only a single-level of NAND gates. The output of every NAND term feeds back into the inputs of the NAND array. Therefore, every NAND array is connected to the outputs of all other NAND terms through programmable elements. The core of PML can thus be looked upon as a programmable interconnect which is a NAND array that has a large number of NAND terms. Figure 3 shows an SR latch and the programmable NAND array. The main advantages of this architecture are

- It allows for product terms to be shared.
- It overcomes the limitations of traditional PALs to preassign a macrocell to a particular OR gate. Instead, the PML structure is much more efficient in terms of utilizing the resources provided within the macros.

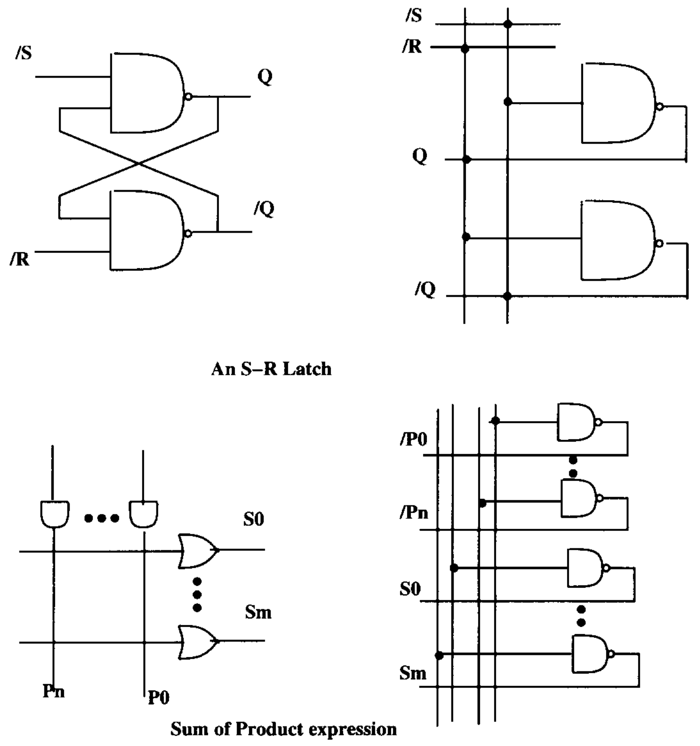


Fig. 3. Folded NAND arrays as exemplified by the PML devices of Signetics.

- It allows single-level logic functions to be realized efficiently.
- The feedback provide by the NANDs enable the user to implement complex multi-level functions without using valuable I/O pins or macro-cell circuitry.

2.1.2. Macrocells

The macrocell structures vary widely among different devices. Some of the major functions performed by macrocells are the ability to

- *generate complemented outputs*. This can provide a savings in the number of product terms if the number of 1s in a function is much more or less than the number of 0s.
- *register outputs* by including flip/flops at the output pins. These are useful to implement state machines such as sequencers. The type of flip/flops also can can vary and this can affect the number of product terms needed as well. For example, registers are best realized with D flip-flops, while counters are more efficiently realized using either T or J-K flipflops.
- *feedback* some outputs (both combinational and registered) internally as additional inputs to generate other product terms. This feature makes it possible to implement multilevel logic, sequential logic, amongst others.
- *to preset all flipflops* so as to be able to bring the system up in a known state.
- *selectively enable* output pins based on either local or global control. These are important in implementing multilevel PLD networks.
- *output polarity* that can be used to generate positive or negative logic signals.

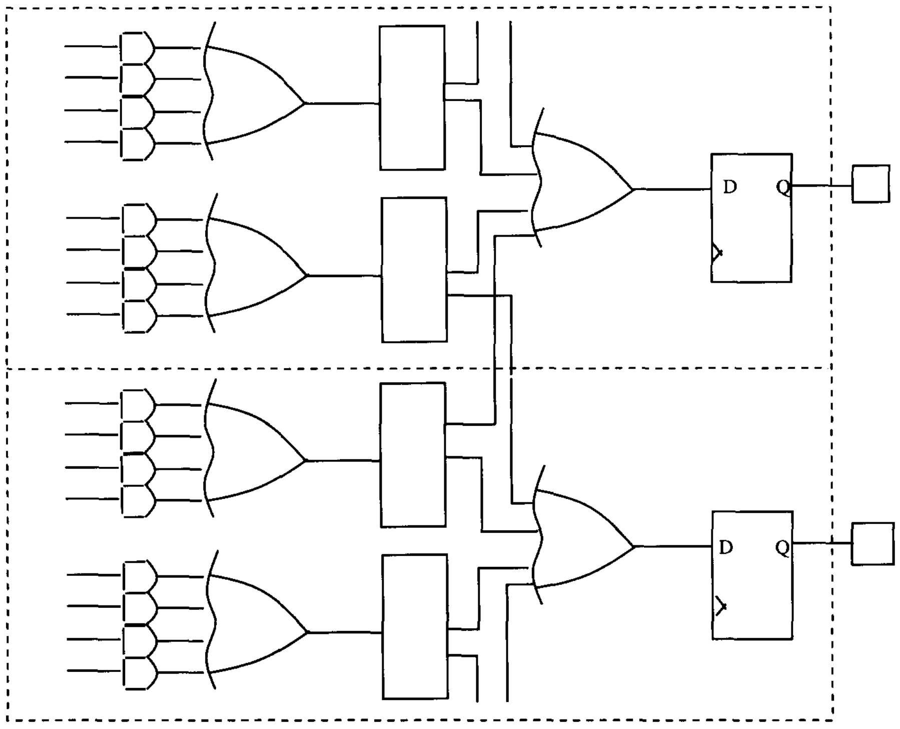


Fig. 4. Product term steering through reallocation.

- *product steering in PALs*: The macrocells in some complex PALs often provide *product-steering* features, i.e, special ways of assigning product terms to a block. This enables the PLD to be used for implementing larger fan-in circuits. Three main techniques for doing so are called *allocation, joining* and *expander logic*.
  (1) In allocation, a particular macrocell can have more product term inputs by taking logic from an adjacent macrocell. This helps for instance in generating wider OR gates than are directly possible. In the Intel device shown in Fig. 4, each output normally can generate an OR of upto 4 terms. But for larger fan-in requirements, the product terms of neighboring macrocells can also be used up to a maximum of 16 inputs.
  (2) Joining refers to the ability to share inputs to a macrocell between its combinational and sequential parts as required. It can be done without incurring any timing penalty.
  (3) Finally, expander logic techniques provide additional product terms that can be assigned to any macrocell needing it. Thus expander logic can be viewed as providing PLA product term sharing functionality in a PAL device at the cost of an extra level of delay. Figure 5 shows the expander logic implemented in the Altera macrocell.

2.1.3. More complex PLDs

A generic block diagram of a complex PLD, typifying the MAX (multiple array matrix) family from Altera [7] is shown in Fig. 6. Other commercial offerings of such complex PLD devices with gate-counts of between 1500 and 10 000 include devices from AMD Mach and Plus Logic. For instance, the EPM5128 device from Altera has eight dedicated inputs including the clock, 64

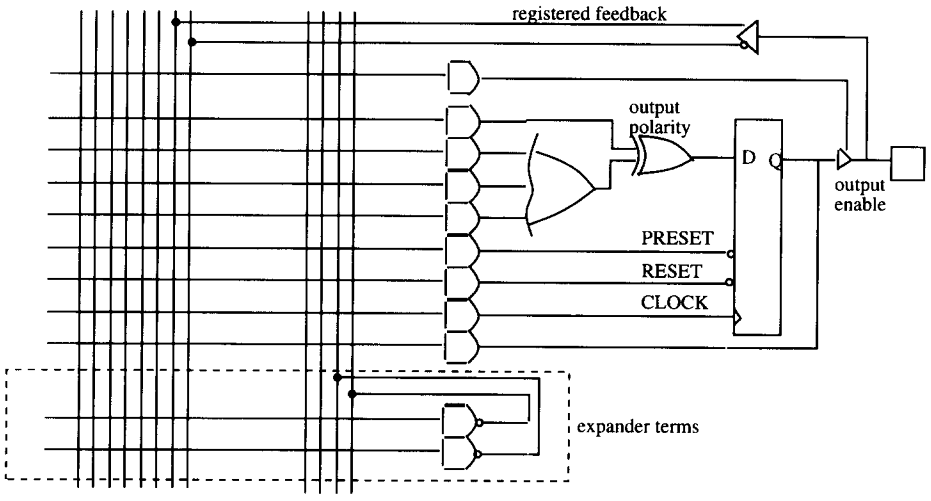


Fig. 5. Product term steering using expander terms.

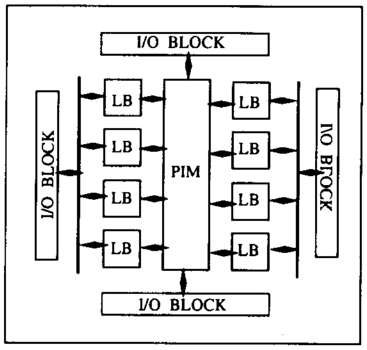
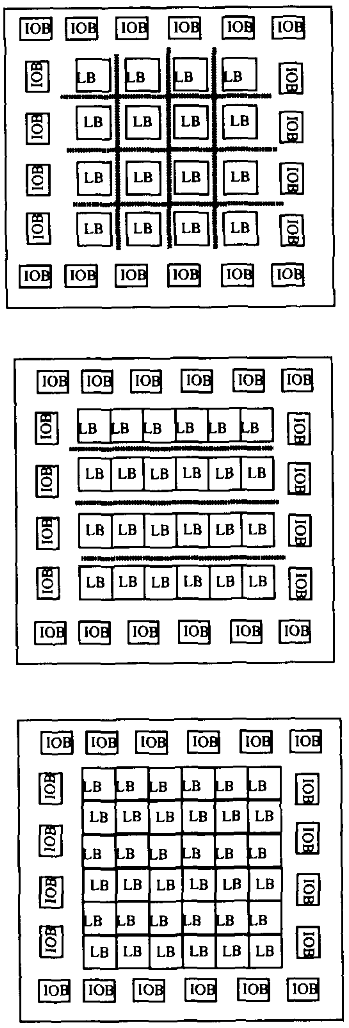


Fig. 6. A block diagram of a typical complex PLD

programmable I/O pins, eight logic blocks (LBs), 16 macrocells and 32 product term expanders per LB. Each expander term (see Fig. 5) makes it possible to implement a function with upto 35 product terms inside a single macrocell which can be contrasted to only eight terms per function for most PAL families. The dedicated input pins come in along the top and are distributed to each of the eight LBs. The PIM (programmable interconnect matrix) routes global signals. All on-chip signals also have a path to the PIM if so needed.

2.2. FPGAs

Instead of programmable AND or OR arrays, FPGAs have an array of programmable logic cells interconnected by a programmable wiring matrix. These logic cells have limited fanin and fanout and so placement and routing of the logic cells can have a significant impact on the timing performance of a device. FPGAs place the most demand on design tools that are fully automated and time efficient and so we shall review algorithms pertaining to FPGA design in greater detail in this paper. In a sense, FPGAs can be viewed as an extension of PLDs that account for the need to incorporate more flip-flops and random logic along with fast local interconnections.



ARRAY STYLE
(eg Xilinx series)

LB: Lookup table based; complex;
larger fanin (9 for the 4000 series)
Routing resources: Horizontal/vertical
wiring channels. Programmable switchboxes
at channel intersection and for connecting
cell inputs/outputs to adjacent channels.

ROW STYLE
(eg Actel series)

LB: Mux based; simple;
smaller fanin; non-registered
Routing resources: Segmented channels
between cell rows. Programming through
antifuses- hence faster

SEA-of-GATE STYLE
(eg Labyrinth series)

LB : Simple; can be configured either as a
logic block or as a routing unit.
Routing resources: Very
limited special purpose routing resources
are provided.

IOB refers to programmable Input/Output Block.
They enable configuring pins as input,/output or
bidiretional ports.

Fig. 7. FPGA architectural styles.

FPGA architectures can be characterized based on:

- *Array style*: FPGAs from Xilinx [8] and Actel[9,10] use a *channelled* architecture with distinct logic blocks separated by routing channels. The Xilinx model resembles the traditional island-style of gate-array architecture with logic blocks separated by channels in both directions; while the Actel resembles the standard-cell format with logic blocks in rows separated by routing channels. There are even sea-of-gate array offerings with mainly local interconnections such as Algotronix, Concurrent Logic and GEC-Plassey. These are shown in Fig. 7.
- *Logic block*: Arrays differ in the logic block functionality as well. Xilinx uses a fairly large logic block with table-lookup functionality and two D flip-flops. Actel logic blocks, on the other hand are very small and are multiplexer based. Altera logic blocks directly support multi-level combinational logic (AND-OR-EXOR).

- *Routing resources*: Xilinx arrays provide specialized routing blocks to enable interconnection of a subset of input pins to one another. Actel arrays are based on the use of segmented channels with antifuses. Algotronix and CAL reuse some of the logic cells themselves to act as routing resources. This will be elaborated in the routing subsections since they play a key role in the algorithms. In most implementations, some long-range global busses are also provided to improve the routability of the design. Special clock schemes are also directly provided which minimizes the clock skew problem that can otherwise occur.
- *Programmability*: Programming of a PLD or FPGA is accomplished by enabling or disabling interconnections in the device's programmable array so as to obtain a certain logic function. This process is referred to as array *personalization*. There are two types of programmable FPGAs:
  - *One-time programmable*: The best example are the Act series of devices from Actel which antifuse as the programming elements. If a high current is passed through the antifuse, it shorts and forma a low-resistance connection. One-time programmable arrays tend to be faster and hence are preferable when speed is important. Also, since an antifuse is relatively small, many more of them can be incorporated within the same silicon area.
  - *Many time programmable*: The best example here are the LCA family of devices from Xilinx which uses pass transistors controlled by SRAM cells to implement the programmable points. The advantage is the same hardware can be reprogrammed and reused several times. The disadvantage is that the on-resistance is usually 2–3 times more than that of the antifuse technology and hence leads to more switching delays. All the memory cells are linked into one huge shift register. Reprogramming the device can thus be achieved by raising the voltage on a certain programming pin on the device and shifting in the appropriate configuration data.

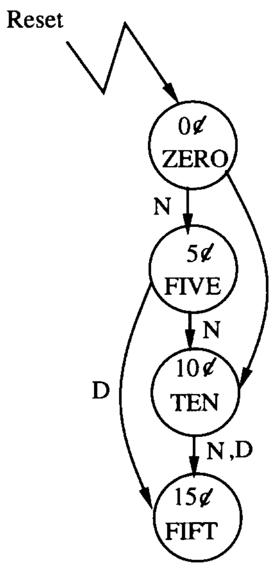
3. The design process

3.1. Description of the main steps

The main steps in the design process, as outlined in Fig. 1, are

(1) *Design entry*: using software such as ABEL from Data I/O, CUPL, AMAZE, PLAN to name a few. These programs are much more flexible and offer the user a range of options such as high-level logic equations, truth tables, state diagrams or structurally, in the form of schematics, as compared to initial offerings such as PALASM from MMI (now part of AMD). The latter were extremely restricted in the sense that the circuit had to be specified in terms of Boolean equations only. Having a range of options is helpful because different types of circuits are most easily described in different ways. Counters and multiplexers are best expressed as Boolean equations; display drivers and code converters in the form of truth tables; state machines as state diagrams; circuits that are mainly datapaths such as memory or arithmetic slices are easier to express in structural terms. In fact, the forms can often be mixed and this adds to the ease of the software.

As an illustration, we shall consider the design of a simple controller for a coin-operated vending machine that is supposed to deliver a package of gum after it has received 15 cents in either dimes or nickels, one coin at a time. A mechanical sensor is assumed to indicate to the



State diagram

Present State		Inputs		Next State		Output
Q1	Q0	D	N	D1	D0	Z (open
0	0	0	0	0	0	0
		0	1	0	1	0
		1	0	1	0	0
		1	1	x	x	x
0	1	0	0	0	1	0
		0	1	1	0	0
		1	0	1	1	0
		1	1	x	x	x
1	0	0	0	1	0	0
		0	1	1	1	0
		1	0	1	1	0
		1	1	x	x	x
1	1	0	0	1	1	1
		0	1	1	1	1
		1	0	1	1	1
		1	1	x	x	x

State Transition Table

Fig. 8. Specification of the vending machine controller.

control which coin has been inserted by asserting either the N (nickel) signal or the D (dime) input. The controller's output, denoted Z, has to be asserted once the required amount has been deposited, to dispense a packet of gum. Note, we shall assume that an external reset signal is activated each time by the dispensing mechanism and also that the machine does not give any change. So a customer who pays two dimes is out five cents. Figure 8 shows the state-diagram and state-table representation for this problem based on a simple state-encoding.

The same description can be also entered in a hardware language such as ABEL. The advantages of using ABEL is that it understands many low-level details of different PLDs including polarities and macrocell configurations. Also it can accept both state-tables as well as logic equations. However, the designer still has to be aware of some of the low-level details such as the number of pins, or size of the product terms and so on. If the logic has to be partitioned to fit a particular device, then it has to be done by the user. ABEL cannot handle this. Figure 9 gives an ABEL input specification for the controller. Line 3 specifies the particular device which in this case happens to be the P22V10 PAL which has 12 inputs, 10 outputs, and embedded flip-flops associated with the outputs. Line 5 specifies the output macrocell associated with pin 21 is to be registered with positive polarity. Line 7 is an example of entry using equations while line 8 is a word description of the state diagram. ABEL also allows the designer to specify test vectors.

(2) *Logic optimization*: Once an initial set of equations and state transitions have been specified for a given design, they should next be optimized for the most cost-effective implementation. Two steps are typically involved:
   - *Logic reduction* algorithms which minimize the size of the circuit by determining and retaining only essential terms depending on the type of circuit desired (viz. either using two-levels of logic or multiple levels). In the traditional minimization model, two optimization

```
1. module  VendMachineController

2. title   'Controller to dispense gum for a vending machine
           Created by Venky Ramachandran'

3. u1      device 'p22v10';"Device specified as a P22V10 PAL

4. "Input Pins
      Clk, Reset, N, D pin 1,2,3,4;

5. "Output Pins
      Z  pin 21;
      Z  istype 'pos,reg';

6. "State registers with simple-encoding
   SREG  = [S1, S0];
   ZERO  = [0, 0];  "nothing deposited yet
   FIVE  = [0, 1];  "five cents
   TEN   = [1, 0];  "ten cents
   FIFT  = [1, 1];  "give him the gum

7. equations
     [S1.ar, S0.ar] = Reset; "Reset to state ZERO
     Z   =  S1 & S0;

8. state_diagram SREG
   state ZERO: if Reset then ZERO
               else if N then FIVE
               else if D then TEN
               else ZERO
   state FIVE: if Reset then ZERO
               else if N then TEN
               else if D then FIFT
               else FIVE
   state TEN:  if Reset then ZERO
               else if (N # D) then FIFT
               else TEN
   state FIFT: if Reset then ZERO
               else FIFT

9. test_vectors
                  .
                  .
                  .
10. end VendMachineController
```

Fig. 9. ABEL input file for vending machine controller.

criteria have been used: minterm or product term minimization and literal minimization. In PLA design, the first factor is important in reducing the size of the PLA while the second is usually irrelevant as each input is available to all product terms anyway. The reverse is true for FPGAs where literal minimization is often more important. The main reason is that the latter leads to smaller fanin circuits and places less strain on the routing fabric available.

- *State assignment and encoding* which is especially important for finite state machine (FSM) designs such as the vending machine controller. A good assignment can considerably simplify the complexity of the next state and output equations. This step traditionally consists of the following four steps:

(a) *State identification*: Determine an appropriate set of states that capture the required design and generate a state transition table.

(b) *State reduction*: Identify and merge states which have identical transition and output functions on the same inputs. State minimization techniques, however, have limited use in large designs. This is because for a flip-flop to be saved, up to half the existing number of states may have to be eliminated.

(c) *Register synthesis*: Identify the proper types of flip-flops to be incorporated which will lead to a minimal design. Proper register synthesis can lead to a considerable saving in the number of product terms needed to generate the excitations of the state registers. For example, counters are usually designed using JK or T flipflops instead of D flipflops. This is because a transition term is always needed for a D flip-flop whenever it has to be set 1 irrespective of whether the flip-flop is currently holding a 1 or not. For JK and T flip-flops, such hold states do not need any connection; hence a product term can be saved. For simple pad-restricted PLAs with no internal feedback, D or T flip-flops may be preferred since they require only one pad.

(d) *State assignment*: Assign codes to the various states and derive the required excitation functions.

Logic synthesis tools are discussed in Section 4.

(3) *Partitioning*: Partitioning can be defined as the process of breaking a design into pieces for some purpose. A circuit may be partitioned in order to assign parts of it to different chips, to reduce the running time of a placement or routing algorithm, or for other reasons. Partitioning for PLDs is usually done using function decomposition methods whereas those for FPGAs use more sophisticated algorithms that have been successfully applied to custom VLSI design. Pin assignment is also performed based on the partition generated. Partitioning methods are discussed in Section 5.

(4) *Technology mapping*: Technology mapping can be defined as the problem of transforming a set of logic equations into an interconnection of parts that are instances of available elements in a given library. In case of PLDs, the problem of mapping the equations to the array (product terms) can be achieved in a straightforward fashion except for the more complex devices. In case of FPGAs, the problem is much more complex for two reasons: (i) Complexity of the basic circuit elements (lookup-tables or multiplexers in lieu of product terms) which can implement a variety of logic functions other than simple NAND/NOR like functions, (ii) Generality of the interconnection scheme which can easily support multi-level logic implementations. Good schemes are available, that are fast and efficient, and are covered in Section 6.

(5) *Place and route*: This is of consequence only for FPGAs. Placement algorithms such as the min-cut approach or standard-cell placement techniques are used to generate an initial placement so as to maximize the chances of the router. The routing problem differs from routing in custom design styles because the wiring segments that are available for routing are preplaced and can be connected together in only specific patterns. The router has to also resolve conflicts arising due to assignment of wiring segments to different nets. Some timing features are also needed to minimize the number of programmable elements within the path so as to minimize the total delay. Earlier, we had classified FPGAs into row-based, matrix-based or plane-based (sea-of-gate) style architectures. The routing algorithms differ for each of them as it is directly related to the underlying resources available.

(6) *Device fitting*: Device fitting for most PLDs [11] deals only with pin assignment, node assignment, macrocell configuration and control term configuration. Pin assignment refers to the assignment of an output to a particular pin, similarly node assignment refers to the selection of the appropriate node or internal point. These are non-trivial problems if, for example, there exist different types of macrocells. Macrocell configuration takes into account the programmable resources in a device's macrocell in transforming the minimized equations onto the device. The

various handles that can be used include choice of feedback, output polarity and so on. Control configuration involves choosing between options for register preset, output enable and so forth. These are highly architecture dependent and general techniques have not been developed. Usually the fitter will have some device specific code that will accommodate the unique features of that device. The final output is typically a JEDEC-style fuse map or, in the case of FPGAs, a configuration bitmap that has been derived from the routed netlist. These are then used to program the device using a piece of custom hardware called a device programmer that is attached to a host computer containing the configuration data. In some cases, the same hardware can be repeatedly personalized and this feature becomes extremely useful in correcting design errors or in accommodating changes in the initial design specifications.

3.2. Example

We illustrate the above steps by way of the custom XACT design software [12] for use with the Xilinx LCA (logic cell array) family of FPGA devices.

(1) *Entry*: The design is first specified, say using ABEL, and translated into an *external netlist format* (XNF) file. The XNF format is primarily provided to provide a uniform interface to the various design-entry tools. One or more XNF files can be created per design. These files have a one-to-one correspondence to the design but are not LCA-specific.

(2) *Optimization and partitioning*: The XNF files are manipulated in several ways to create an optimized circuit targeted specifically to a particular LCA. Multiple XNF files can be merged together into a single circuit and the Xilinx *circuit optimizer* (XNFOPT) program can be invoked to better fit the circuit to the particular LCA's constrained structure. For example, if the circuit uses very wide gates, then it can be decomposed into a set of four or five input gates that are amenable to single CLB implementations. In practice, XNF files are first converted to an intermediate form called a *MAP* file, which represents a partitioned circuit. Each of the circuit partitions in the map file corresponds to one CLB or IOB in the target device. After the MAP files have been generated, they are then converted to an *LCA* design file using the program *Map2LCA*. The LCA file is actually a command script that controls the XACT software during the placement and routing portion of the implementation phase. The entire design can be merged and converted into a single LCA design file or the various design segments can be maintained as separate LCA files and merged later. This decision depends on the size and complexity of the circuit being implemented.

(3) *Implementation*: The LCA file created above is then fed to the *automatic place and route* (APR) program which results in the generation of a new LCA design file that includes the calculated logic block placements and signal routings. These can also be manually inspected using the XACT design editor and modified if so desired. Once an acceptable solution is achieved, then the final step is to convert it to a binary bit pattern using a program called *Makebits*. For prototyping, the bit pattern can be downloaded from the host computer using a special download cable. However, this method is vulnerable to power failures. A more permanent solution is to store the bit-pattern in a PROM that is installed on the board along with the LCA with provisions to boot the LCA from the PROM on power-up.

4. Logic synthesis algorithms

4.1. Two-level synthesis techniques

As discussed in the previous section, when applied to PLAs, the primary goal is to minimize the total number of product terms.

4.1.1. Logic minimization

For the two-level minimization problem, the following steps are typically performed for each function:

(1) Exhaustively generate a set of prime implicants from the sum of products expressions.
(2) Choose a minimum set of the prime implicants generated so that all the minterms of the function are covered. Each of these implicants represent a product term when mapped onto the PLD.

Exact methods: For small problems, Step 1 can be done using Karnaugh maps or a standard algorithm such as the Quine–McCluskey tabular method, while a minimum cover can be obtained using Petrick's method. Typically, several rules regarding essential rows and columns, dominated rows and dominating columns can be applied to simplify the prime implicant table prior to applying Petrick's algorithm. While this procedure is sufficient for PALs, it may not yield optimal results for PLAs. This is because in PLAs, product terms can be shared between different functions (*product-term sharing*) leading to a substantial decrease in the overall number of product terms needed. For PLAs, therefore, the multiple-output variant of QM method is used wherein we try to optimize the total number of implicants for a group of functions taken together. These are the so-called multiple-output prime implicants which in fact do not need to be prime in the individual functions.

Heuristic approaches: Unfortunately, these methods are NP-complete because the number of minterms for a function with n variables can be as high as 2^{n-1} (parity function) and the number of implicants can be as high as 3^n. So, an exhaustive generation method such as QM is not practical for large circuits. Consequently, several heuristic algorithms such as Mini, Espresso [13], etc. have been designed. The idea is to derive a minimum cover with high probability without first generating all prime implicants. Efficient computational techniques based on unate functions have been developed for Espresso and fairly good solutions are typically produced. Typically, Espresso performs three steps iteratively:

(1) The first step is called *expansion* wherein each implicant is expanded as much as possible; covered implicants are dropped from further consideration while essential implicants are converted to don't-cares.
(2) The next step is *reduction* wherein the primes generated in expansion are trimmed to the smallest size without losing any minterm coverage.
(3) The final step is sometimes referred to as *reshaping* wherein the best irredundant cover is generated.

Figure 10 shows the input and output PLA personalization files produced by Espresso using the state assignment shown in Fig. 8. The first two lines show the number of inputs and outputs while the next two associate symbolic names to them. This is followed by the PLA personalization matrix with standard notations (a 1 or a 0 in the AND plane denotes a variable in true or complemented form while a – indicates a don't-care. Similarly a 1 in the OR plane denotes a product term that is to

```
.i 4
.o 3
.ilb q1 q0 d n
.ob d1 d0 z
.p 11
0000 000
0001 010
0010 100
0100 010
0101 100
0110 110
1000 100
1001 110
1010 110
11-- 111
--11 ---
.e

Espresso input  file
```

```
.i 4
.o 3
.ilb q1 q0 d n
.ob d1 d0 z
.p 7
-0-1 010
-1-1 100
-1-0 010
1-1- 010
11-- 011
--1- 100
1--- 100
.e

Espresso output  file
```

Fig. 10. Espresso files for vending machine design.

be included in the SOP expression for the corresponding output). Thus, from this particular matrix, we have

$$\begin{aligned} d1 &= q1 + D + q0 \cdot N, \\ d0 &= q0' \cdot N + q0 \cdot N' + q1 \cdot D + q1 \cdot q0, \\ Z &= q1 \cdot q0. \end{aligned} \tag{1}$$

Architecture-based minimizations: Another minimization technique that can be useful for PLA designs is that of *input reduction*. This involves the detection and elimination of logically redundant inputs and can be achieved using a tabular approach similar to the implicant covering procedure. Yet another optimization technique is made possible in some PLAs (e.g. the Signetics series) using a *complementary array* connection. This is simply an OR term which can have connections to all product terms; the complement of this OR is then made available as a new input to the AND array. This feature can be made use to generate default conditions (for example, in a BCD counter, this can be used to detect if the input is a non-BCD number and take appropriate action). It can be also profitably used in sequencer designs to test for illegal states and make the transition to a valid state. Often the alternative approach of specifying the default conditions in terms of Boolean equations can lead to many more product terms and is more complex.

4.1.2. State-assignment methods

A crude estimate of an upper bound on the total number of product terms needed is given by the sum of the number of distinct destination states for each input in the state transition table. However, with a good assignment this number can be significantly reduced. Choosing the best assignment is NP-hard since the maximum number of distinct state assignments for N states using q variables is $2^q - 1!/2^q - N!q!$. Further constraints are imposed on state assignment due to static and dynamic hazard considerations. For example, to avoid a static hazard, adjacent states should vary by just 1 bit position. The same holds when state variables are used as outputs as in Moore machines. One of the CAD tools available for state encoding is called *nova* [14] from the University of Berkeley. Intuitively, nova clusters states that are mapped into the same next state by some input and those that assert the same output into separate groups. In the terminology of state assignment, these are called *input constraints*. Nova attempts to assign adjacent encodings within the smallest Boolean cube to states in the same group. A related concept is *output constraints*. States that are next states of a common predecessor state are given adjacent assignments.

Nova implements a wide range of state-encoding strategies any of which can be selected while running the program. Some of these strategies are as follows:

(1) *Greedy*: prioritizes satisfying input constraints and only looks at new state assignments that show a strict improvement over those already examined.
(2) *Hybrid*: also prioritizes satisfying input constraints but also considers some state assignments that are worse-off in the short run but can potentially lead to better results. This minimizes the possibility of getting trapped in a local optimum.
(3) *Annealing*: same as hybrid but uses a more sophisticated annealing schedule for improving state assignment.
(4) *I/O hybrid*: same as hybrid except it also tries to satisfy output constraints at the same time.
(5) *One-hot*: uses a one-hot encoding which rarely minimizes the number of product terms but can significantly reduce the number of literals. Hence, this technique is more useful for FPGAs than it is for PLAs.
(6) *Random*: As the name indicates, assignments are tried out randomly and the best found reported.
(7) *Exact*: obtains the best encoding satisfying input constraints. Still may not be the best overall due to output constraints.

Example 4.1. For the vending machine example, the various encoding algorithms of nova yield the following assignments:

	ZERO	FIVE	TEN	FIFT	Number of product terms	Number of literals	PLA area
Greedy	01	10	00	11	6	21	66
Annealing	10	01	00	11	6	20	66
Hybrid	10	01	00	11	6	20	66
IO hybrid	00	11	10	01	8	23	88
Random	01	10	11	00	6	23	66
One-hot	1000	0100	0010	0001	9	21	153

The input file to nova is shown in Fig. 11. Each line has the following syntax:

$InputsPresent - NextState - StateOutputs$

Some of the resulting PLA personalization files generated using Espresso on these state assignments are also shown. Not in each case the order of the inputs and outputs are $D, N, q1, q0$ and $d1, d0, Z$ respectively. For the vending machine example the IO-hybrid strategy does poorly but the rest excepting the one-hot method give more or less the same area. The PLA area is computed to a first approximation by the formula $Area = (2i + o)p$ where i, o, p are the number of input variables, output variables and product terms used. The equations for the best case (annealing) which has 6 distinct product terms is given below:

$$d1 = N' \cdot D' \cdot q1 + N \cdot q1' \cdot q0' + q1' \cdot D + q1 \cdot q0,$$
$$d0 = q0' \cdot N + q0 \cdot N' + q1' \cdot D + q1 \cdot q0,$$
$$Z = q1 \cdot q0.$$

```
                                                        .i 6
                                                        .o 5
                                                        .p 9
                    .i 4            .i 4                1-1--- 00100
00 ZERO ZERO 0      .o 3            .o 3                -11--- 01000
01 ZERO FIVE 0      .p 6            .p 6                -1-1-- 00100
10 ZERO TEN  0      00-1 010        001- 100            -1--1- 00010
00 FIVE FIVE 0      -100 110        -100 100            001--- 10000
01 FIVE TEN  0      -01- 100        -1-0 010            00-1-- 01000
10 FIVE FIFT 0      -1-1 100        -0-1 010            00--1- 00100
00 TEN  TEN  0      --11 011        1-0- 110            -----1 00011
01 TEN  FIFT 0      1--0 110        --11 111            1-0--- 00010
10 TEN  FIFT 0      .e              .e                  .e
-- FIFT FIFT 1
11  -    -   -      Greedy          Annealing           One-hot

Nova input file   Espresso outputs generated based on nova state-assignments
```

Fig. 11. Nova files for vending machine design.

In contrast the equations for the one-hot encoding style are

$$d3 = N' \cdot D' \cdot q3, \qquad d2 = N \cdot q3 + q2' \cdot D' \cdot N', \qquad Z = q0,$$
$$d1 = D \cdot q3 + N \cdot q2 + D' \cdot N' \cdot q1, \qquad d0 = q1 \cdot N + q0 + q3' \cdot D.$$

The one-hot encoding needs the most product terms; however an inspection of the next-state and output equations reveals that only 21 literals and 11 gates with fanin of 3 or less (5 2-input ANDs, 3 3-input ANDs, 1 2-input OR and 3 3-input ORs) are required. Smaller fanin gates are one reason why this technique is popular for FPGA designs that are heavily fanin constrained.

4.2. Multi-level logic synthesis

Unlike simple PLDs, the multi-level logic synthesis [15] is the logic minimization type of choice for FPGAs. This is because the logic blocks in FPGA have very limited fanin and fanout capabilities. The more powerful routing structures also support the multi-level implementations. Note that the use of expander terms in complex PLDs is also an instance of a multi-level logic structure and so much of the following discussion is also applicable for them.

4.2.1. Multi-level logic minimization

The primary goal of multi-level logic minimization is to minimize the total number of literals in a Boolean equation. This generally leads to fewer resource requirements but also often leads to more logic levels and hence increased delay in logic paths. Such delays, if found unacceptable, can lead to further constraints. The main idea is to compute the factors and subfactors of a design's Boolean equations; these determine the number of logic levels possible. Each factor or subfactor can be treated as an internal input that can be fanned out to multiple outputs or other logic levels. A popular tool for multi-level synthesis is the *misII* [16] program developed at the University of Berkeley.

To begin with the Boolean expressions are placed in a "tree-structure" or are manipulated algebraically. The former usually is a *directed acyclic graph* (DAG). Each node in the DAG defines a function represented by a variable associated with the node. The root node defines the entire Boolean function. The following type of operations are then performed on this data structure.

(1) *Decomposition*: This step takes a single Boolean expression and replaces it by a collection of new expressions.

(2) *Factoring*: This step takes an expression in two-level form and re-expresses it as a multi-level function. For example the function $F = AD + BCD + E$ can be factored as

$$F = GD + E, \quad G = A + BC.$$

(3) *Extraction*: This operates on multiple, possibly factored, functions and identifies common subexpressions amongst them. It is by far the most complex of the operations to implement. The general strategy is to rewrite the expression for each function, say F, in a polynomial form in terms of subexpressions P, Q and R which represent the divisor, quotient and remainder respectively. The divisors represent the required set of possible common factors. Finding divisors is hard because of the complications resulting from application of the rules of Boolean algebra. For example, under algebraic laws the function $H = A + B$ does not divide F. However, using the distributive rule, viz. $X + Y \cdot Z = (X + Y)(X + Z)$ we can rewrite F as

$$F = (A + BC)D + E = (A + B)(A + C)D + E.$$

Clearly, with this representation it is easy to see that $(A+B)$ is indeed a divisor with $(A+C)D$ as the product and E as a remainder.

(4) *Substitution*: Substituting a function G into a function F reexpresses F in terms of G. For example, substituting H in the above example leads to F being expressed as $F = H(A+C)D+E$.

(5) *Collapsing*: This is the reverse of substitution and is done to reduce the number of levels so as to meet a timing constraint.

4.2.2. State assignment for multi-level designs

The goal of state assignment here is to minimize the number of literals that are required for the next-state and output functions. Two tools, *mustang* [17] and *jedi* [18] developed at Berkeley do state assignments for multi-level designs. Like *nova*, *mustang* uses several alternative strategies for state assignment such as *random, sequential, one-hot, fan-in* and *fan-out*. Fan-in and fan-out strategies work by creating partitions among states based on input and output constraints, and the state assignment that maximizes common subexpressions among partitions is chosen. *Jedi* is very similar to mustang except it can find encodings for outputs as well as the states. It uses other solution strategies such as input dominant, output dominant, modified output dominant and input/output combination algorithms. Once, these encodings are made, the resulting state-table or expressions can be input to misII for obtaining an optimized set of logic equations.

Example 4.2. The state assignments made by mustang for the same vending machine controller are shown below.

	ZERO	FIVE	TEN	FIFT	Number of product terms	Number of literals
Random	10	00	01	11	8	24
Sequential	01	10	11	00	8	24
Fanin	11	10	00	01	7	18
Fanout	10	00	01	11	7	20

The number of product terms and literals in the resulting expressions fed to misII is also reported. It can be noted that the fanin heuristic yields the least number of literals; but the random and sequential

```
#state assignment based on fanin heuristic of mustang
.i 4
.o 3
.ilb d n q1 q0
.ob d1 d0 z
.p 11
0011 110
0111 100
1011 000
0010 100
0110 000
1010 010
0000 000
0100 010
1000 010
--01 011
11-- ---
.e

%misII -f lib/script -t pla fin.mustang
 .model fin.mustang
 .inputs d n q1 q0
 .outputs d1 d0 z
 .names q1 q0 z
 01 1
 .names d n q1 q0 z d1
 001-- 1
 0--10 1
 .names d n q1 q0 z d1 d0
 ----1- 1
 010--- 1
 10-0-- 1
 -0-1-1 1
 .end
```

Fig. 12. misII output based on fanout-based state encoding of mustang

versions do poorly and are even worse than those got from *nova*. The misII output script based on the state encodings with the fanin heuristic are shown in Fig. 12. Each of the **names** section gives the PLA matrix for the variable appearing last on the associated name list. For instance, the multi-level (here 3-level) equations computed are:

$$\begin{aligned} z &= q1' \cdot q0, \\ d1 &= D' \cdot N' \cdot q1 + D' \cdot q0 \cdot z', \\ d0 &= z + D' \cdot N \cdot q1' + D \cdot N' \cdot q0 + N' \cdot q0 \cdot d1. \end{aligned} \tag{2}$$

5. Partitioning methods

5.1. Partitioning techniques for simple PLDs

Sometimes, a single PLD is insufficient to realize a given set of functions. This could be for lack of either sufficient product terms, input pins or output pins. One simple method to alleviate the product term crunch is to employ a technique called *product-term sharing*. The approach is to choose a variable to segment the given PLA table; one table comprises the rows of the original table wherein the variable appears in either 0 or don't-care form while the other comprises of the rows with the variable in 1 or don't care form. Since, don't care entries are repeated, one heuristic is to choose variables having fewer don't care entries for segmentation purposes. The two tables can then be independently optimized by the standard techniques. The final output can then be formed by ORing the corresponding outputs of the two PLDs.

Input and *output* encoding schemes have also been proposed when the number of inputs and outputs of a given PLD are at a premium. Input encoding works by identifying a subset, m, of variables that cause the output to be 1 in only say c combinations of these variables. For this technique to be effective, we must have $c \ll 2^m$. Thus we can use another PLD which will identify these c patterns and encode it into $c' = \lceil \log(c) \rceil$ bits which can be then fed into a second PLD. Thus the input requirements for the two PLDs are now m and $n - (m - c')$ respectively. Output encoding techniques can be similarly devised by identifying sets of outputs which are active only in a limited number of combinations. Thus these outputs can be encoded using fewer bits and then fed into a subsequent PLD decoder to generate the original outputs.

A whole theory of *function decomposition* is also available in the literature which generalizes the above simple strategies. Input encoding is an instance of what is known as *simple disjunctive decomposition*. Other forms such as multiple, iterative and complex disjunctive decompositions have also been studied. It should also be noted that such techniques can also be applied for complex PLDS with multiple partitions.

5.2. *Partitioning for FPGAs*

More sophisticated algorithms are used to partition the larger systems designed using FPGAs. These are influenced by several aspects of the design such as:

- The delay introduced into the circuit path is related not only to the length of the signal but also to the number of chip and board crossings. Interchip delays for Xilinx parts can range from 8 to 24 ns.
- Congestion in interchip routing. Since the number of device pins are limited, it is necessary to partition in such a way that all the resulting chip-crossings can be accommodated.
- Physical limits on the logic on each chip. The Xilinx chips, for instance, can accommodate from 2000 to about 10 000 gates theoretically. However, the inefficiency of logic utilization, due to the general purpose design tools, can significantly degrade the usable gate count. Thus the partitioning program has also to ensure that none of the partitions is severely overloaded.
- Complexity of intra-chip routing affects the logic utilization and can cause greater delays.

There have been a number of techniques used for partitioning circuits and graphs such as (a) gradient descent, (b) graph partitioning methods such as Kernighan-Lin (K-L) and Fiduccia-Matheyses, and (c) force-directed methods. Such algorithms have been extensively studied in the literature, for instance, in Chapters 3 and 4 of [19]. It is the intent of this section to only comment upon how these methods can be useful for PLD partitioning problems. We assume the following steps will be performed in order:

(1) Read in the input specification and hardware description of both the logic blocks as well as the routing structure.
(2) Do a critical-path analysis step.
(3) Perform an intial partitioning and pin-assignment step.
(4) Evaluate the solution and generate module moves/swaps.
(5) Repeat steps 3 and 4 till an acceptable solution is produced.

In the original K-L model, the partitioning problem is formulated as a simple graph-problem of dividing a set of nodes connected by a set of edges into two or more subsets so that the number of edges between any two subsets (or bins) is minimized. Furthermore, the problem can be generalized by assigning a cost to each edge and minimizing the sum of the weights of the edges crossing the bins instead. The K-L model considers swaps between pairs of bins at a time. When there are more than two bins, the K-L algorithm can be augmented with the F-M approach which considers only single-module moves at a time.

The force directed approach models a network in terms of springs and spring forces between modules and tries to minimize the total force exerted on the system. When applied to FPGAs, care has to be made to prevent individual devices from overflowing. This can be done using an additional repulsion cost that tends to push modules into different devices. The new force cost becomes $F = Kx + RN$ where R is the repulsion coefficient and N is the number of logic blocks that are already occupied in the same device as the one for which the force is being calculated. Typically, the model has to be accompanied by some sort of gradient-descent approach that can transform the system from one state to another. For example in the Generalized Force Directed Relaxation method suggested by Goto, first single-module moves are attempted. If no such move yields an improved result, then the best move is used and the process repeated, with the proviso that the module must come from the same PLD as the one to which the last module was moved to. If a move at the second level is accepted, it actually corresponds to a module swap via two single-module moves. The approach can be extended to as many higher levels as desired without having to pay the full price of an exhaustive search.

5.3. Pin assignment

In addition to assigning modules to the different devices, it is also necessary to address the problem of assigning pins to all the connections that need to go between the devices. Typical pin assignment algorithms work by attempting to minimize weighted net lengths, where the length of a potential network can be defined as the number of chip boundary traversals that it makes. The weight of the network can be taken as

(1) A constant, in which case the effect is to minimize the total network length, or
(2) An exponential function such as $K \times e^{-(\text{path slack})}$, where the path slack is the difference between the actual delay and the required delay and K is any large constant. The idea is to increase the weights for time-critical nets so that they get assigned shorter routes. This step can also be viewed as a slack-minimization method. Since, this calls for path analysis to obtain the slack delay values, it is much slower than using the network oriented constant cost function

Armed with the data on net weights and location of modules and the FPGA interconnectivity, the pin assignment algorithm tries to find a feasible assignment that minimizes the objective. Note that if the partitioning algorithm had not taken congestion into account, such an assignment may not be feasible because there are fewer pins than are needed to make all interchip connections. In that case, we can use the KL-FM or other algorithms to redo the partition paying special emphasis on the offending modules and go over the entire process again.

6. Technology mapping problem

In this section, we look into the algorithms pertaining to the technology mapping problem for FPGAs. We shall assume that the given network is purely combinational. Networks containing memory elements can also be handled, but it would first be broken into a set of combinational functions at flip-flop boundaries. The network will also be assumed to be represented as a DAG similar to the one used for multi-level optimization programs where the intermediate nodes represented factors of the function. For FPGA implementations, each intermediate node is assumed to be capable of being implemented using a single logic module. The logic modules for most FPGAs come in two flavors:

- Look-up tables (LUTs): These are primarily static RAMs. A LUT with k inputs can support any Boolean function of upto k variables. The Xilinx family of FPGAs is LUT based.
- Multiplexers: The Actel Act1 and Act2 series of FPGAs use multiplexers as the basic logic blocks.

6.1. Generic library mapping methods

The pioneering work in technology mapping algorithms is based on the formalization proposed in the *dagon* [20] and *mis* [16] systems. The overall task can be split into three components:

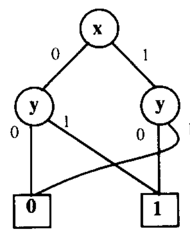
- *Partitioning*: The given network is partitioned into a collection of multiple-input single-output combinational sub-networks.
- *Decomposition*: Each sub-network is then decomposed into two-input functions, to increase the network granularity, and
- *Covering*: Each decomposed sub-network is then covered by available circuit elements of a library so that either area or time is optimized.

Each decomposed sub-network is defined by an appropriate DAG. The DAG is then converted into a tree by creating a unique instance of every input node for each of its multiple fanout edges. The optimal circuit implementing each tree is then constructed using a dynamic programming method that proceeds from the leaf nodes to the root node. For every node in the tree the optimal circuit implementing the subtree extending from the node to the leaf nodes is constructed. This circuit consists of a library element that implements the function specified by the node and the previously constructed circuits implementing its inputs. The cost function can incorporate two measures: (i) an area measure which is simply the sum of the areas of the modules used to construct the DAG, and (ii) a delay measure which is the maximum delay for any leaf node where the delay of a leaf node is defined as the sum of the delays of all modules lying on its path to the root.

To find the lowest cost circuit, typically all library elements that can serve as candidates for implementing the subfunction at any node are considered and the cheapest retained. The set of candidate elements are found using a version of a tree-matching algorithm. There are two main drawbacks associated with these techniques:

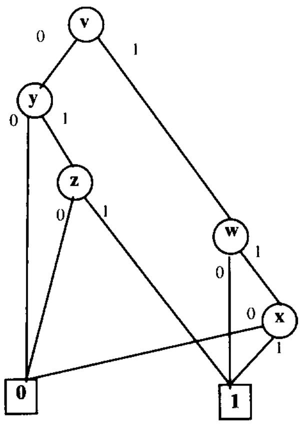
- Requirement for explicit library enumeration: In programs such as misII and Ceres, the library of cells that can be derived from the uncommitted modules needs to be captured explicitly. Such an enumeration can be very large for most functions. [1] Consequently, any practical

[1] Remember there are 2^{2^k} possible k-input functions.



BDD for the EXOR function

xy'+x'y



BDD for the function

v'yz + vw' + vx

Fig. 13. Example module-BDDs.

implementation can hope to explore only a subset of all possible combinations which comes at the expense of quality.

- Retargettability: As a way of getting around the combinatorial problems associated with a very large library, programs such as mis-pga have chosen to target very specific architectures such as the Act-1 series. As a result, the entire library can be modeled using a few primitive building blocks. The resulting programs are fast and yield good solutions; but the drawback is that they cannot be readily extended to other implementations.

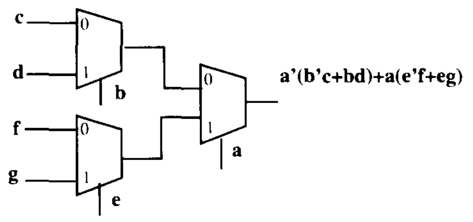
6.2. *Technology mapping for multiplexer-based FPGAs*

Next, we describe a matching algorithm [21] for multiplexer based systems such as the Act-1 family. This algorithm has been implemented as part of the Proserpine system and avoids the pitfalls of the generic method. The main idea behind the matching algorithm is to model the personalization of the multiplexers as *stuck-at-0*, *stuck-at-1* and *bridging* faults on a *Binary Decision Diagram* (BDD) data structure representing the given function.

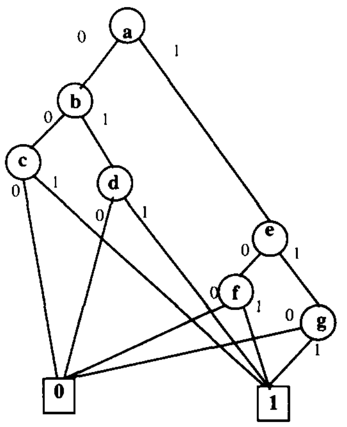
6.2.1. *Binary decision diagrams*

A BDD is a two-terminal leveled DAG with a single root node. The terminal nodes represent the values 0 and 1 respectively. Each level of the BDD is associated with a variable. Every intermediate node is also associated with a function and has two outgoing edges that are labeled 0 and 1. The root is associated with the Boolean function that the BDD represents, while an internal node is associated with the sum of cofactors with respect to the variables on the paths to the root.

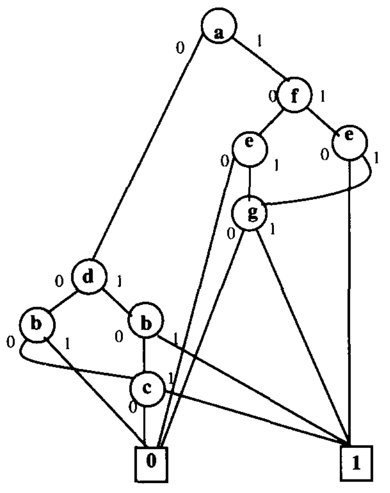
For instance, Fig. 13(a) shows the BDD for the exclusive OR function that could possibly arise as part of a parity circuit or for the SUM function of a half-adder. Figure 13(b) shows the BDD for a more complex function. We will refer to BDDs representing circuit functions to be implemented as "function" BDDs. This is to differentiate them from the "logic-block" BDDs that represent the function that a given logic block represents. For instance, Fig. 14 pertains to the Actel series multiplexer based



The Actel Act-1 multiplexer based logic block



Input order: (a b c d e f g)



Input order: (a f e g d b c)

Two BDDs for the Act 1 block

Fig. 14. Act-1 logic block and two of its BDD representations.

logic block. The figure also is useful in noting the fact that a function need not have a unique BDD. In fact, the structure depends on the order of the variables used to levelize the graph. This fact is important from the point of view of the matching algorithm to be described shortly. A lot of unnecessary search can be avoided based on the fact that for an input ordering, the reduced BDD (i.e. a BDD such that no two sub-BDDs are isomorphic) is a canonical form. Hence, it is possible to construct a more general structure, also called a *Global Binary Decision Diagram*, that represents the logic block structure for all possible input orderings. The main advantage of the GBDD is that it contains no subgraphs that are isomorphic to each other.

6.2.2. Matching algorithm

One part of the algorithm is fairly obvious by now. The goal is to detect a subgraph within the logic-block BDD that is isomorphic to the function BDD. A few observations are in order. First, it is sufficient to search for subgraphs of the same height as the function BDD. Secondly, it is necessary

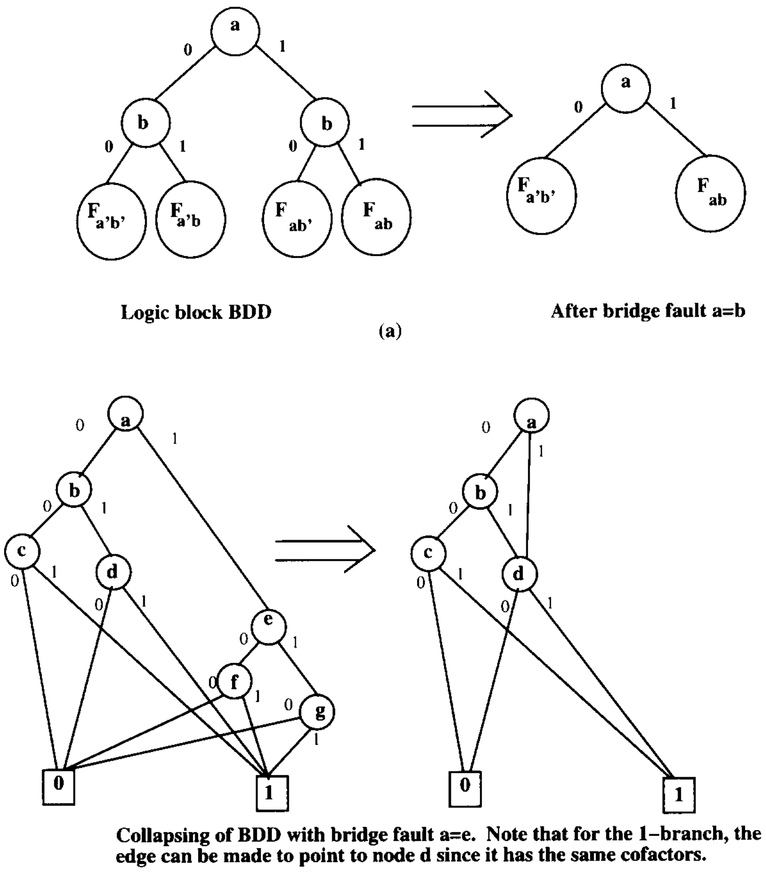

Logic block BDD
After bridge fault a=b
(a)
Collapsing of BDD with bridge fault a=e. Note that for the 1–branch, the edge can be made to point to node d since it has the same cofactors.
(b)

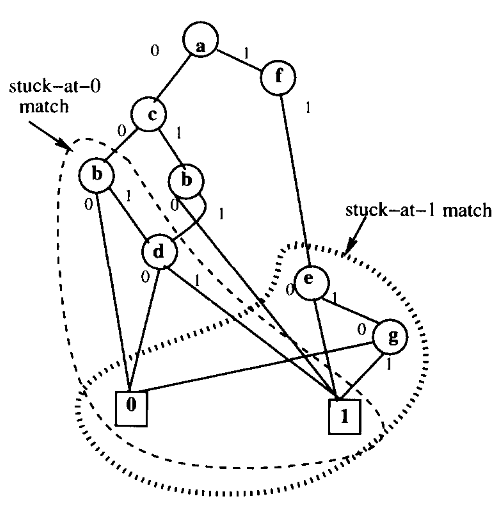
Fig. 15. Effect of bridging on BDD structure.

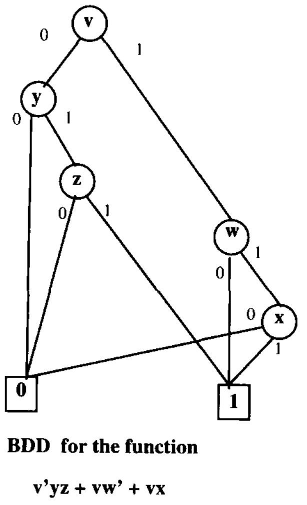
to consider all possible input orderings in the process. However, the number of such searches can be reduced by only considering non-isomorphic cases. The GBDD structure is quite useful for this purpose.

The second part of the algorithm is executed if a match is found and pertains to "personalizing" the logic block by connecting input variables or constants 0 or 1 to each of the inputs of the logic block. This is done as follows:

- The variables on the path leading from the root node of the module BDD to the root of the matched subgraph will be assigned constant 0 or 1 values depending on the corresponding edge labels. These variables can be thought of as stuck-at faults.
- The mapping between the nodes in the matched subgraph and the circuit BDD specifies the input assignment to be made.

Even when the matching algorithm fails to find a set of stuck-at faults that personalizes the logic block to the cluster function, it may still be possible to do so by *bridging* together certain inputs. The idea is as follows: Suppose in an input ordering variables a and b are adjacent to one another. Now if they were to be bridged, i.e. have the same logic value, then it is clear that of the four possible cofactors $F_{a'b'}, F_{a'b}, F_{ab'}, F_{ab}$, the only two still possible are $F_{a'b'}$ and F_{ab}. What this implies in term





Input order: (a f c b d e g)
Bridge fault: a = f

Fig. 16. Matching illustration using 1-bridge.

of the logic BDD is that the subgraphs corresponding to the other two cofactors, viz., $F_{a'b}$ and $F_{ab'}$ can be deleted. Also the node corresponding to the variable b can be dropped. Figure 15(a) shows the general case of bridging; while Fig. 15(b) shows the effect of bridging variables a and e for the Act-1 logic block under the first input ordering of Fig. 14. Often times, the resulting BDD is structurally different from the ones constructed purely on the basis of input orderings. Hence, the matching process can be continued on the new BDDs. Note that bridging can in principle be carried out for any number of variables. However, even for the simplest case of a 1-bridge fault, wherein only two variables are considered, the possibilities are astronomical. Clearly for each of the input orderings, a different pair of input variables can be considered.

In practice, though, the one-bridge algorithm considers each of the function variables in order and constructs a function BDD with that variable as the first one in the BDD ordering. For each of the function BDDs, all possible input orderings are considered for the logic-block BDD. It then searches for subgraphs within this BDD where bridging the first two variables of the subgraph (say a and b) to the first variable of the function BDD (say x) results in a match being found. In actual practice, this determination can be done in place rather than create a new BDD as suggested by Fig. 15(b). The key point is to note that the effect is the same as checking for match between the subgraph of the function BDD specified by the stuck-at-fault $x = 0$ and that in the logic-block BDD specified by the pair of faults $a = 0$ and $b = 0$. Similarly, a match is checked between the subgraph of the function BDD specified by the stuck-at-fault $x = 1$ and that in the logic-block BDD specified by the pair of faults $a = 1$ and $b = 1$.

Figure 16 shows the logic-block BDD based on the input ordering $(a\ f\ c\ b\ d\ e\ g)$. We consider the 1-bridge of the first two variables a and f and check for isomorphic subgraphs based on the stuck-at notion developed above. The two subgraphs for the logic-block BDD corresponding to the appropriate stuck-at BDDs of the function module are also marked in the figure. From this it can be determined that the required personalization is $a = v, b = y, c = 0, d = z, e = w, f = v, g = x$. Note how the input variable v is simultaneously applied to both inputs a and f which represent the bridge set in this case.

Example 6.1. Figure 17 shows the implementation of the vending machine design using Eq. (2) and the Actel mux-based architecture shown in Fig. 14 with b, c, d, e, f, g replaced by SA, $A1$, $A0$, SB, $B1$, $B0$ and signal a formed by the OR of $S1$ and $S0$. As can be seen, this is a three-level implementation and needs only 7 basic blocks and so is quite compact. The lines beginning with an asterisk have been added by us to clarify the function being implemented by that logic block. On the other hand, a two-level implementation would require upto 16 basic blocks.

```
MODEL "act.in";

TECHNOLOGY scmos;
VIEWTYPE SYMBOLIC;
EDITSTYLE SYMBOLIC;

INPUT  "d" : "d";
INPUT  "n" : "n";
INPUT  "q1" : "q1";
INPUT  "q0" : "q0";

OUTPUT  "d1" : "[26]";
OUTPUT  "d0" : "[28]";
OUTPUT  "z" : "[24]";


 INSTANCE "BASIC_BLOCK":physical NAME = INST0;
        "A0"           :          "GND";
        "A1"           :          "q0";
        "SA"           :          "Vdd";
        "B0"           :          "GND";
        "B1"           :          "GND";
        "SB"           :          "Vdd";
        "S0"           :          "q1";
        "S1"           :          "GND";
        "out"          :           "[24]";
******=> out = q1'q0

 INSTANCE "BASIC_BLOCK":physical NAME = INST1;
        "A0"           :          "GND";
        "A1"           :          "q0";
        "SA"           :          "Vdd";
        "B0"           :          "GND";
        "B1"           :          "GND";
        "SB"           :          "Vdd";
        "S0"           :          "[24]";
        "S1"           :          "GND";
        "out"           :          "[36]";
******=> out = [24]'q0

 INSTANCE "BASIC_BLOCK":physical NAME = INST2;
        "A0"           :          "GND";
        "A1"           :          "[36]";
        "SA"           :          "Vdd";
        "B0"           :          "Vdd";
        "B1"           :          "[36]";
        "SB"           :          "n";
        "S0"           :          "q1";
        "S1"           :          "GND";
        "out"           :          "[37]";
******=> out = q1 N' + [36]
```

```
 INSTANCE "BASIC_BLOCK":physical NAME = INST3
        "A0"          :          "GND";
        "A1"          :          "[37]";
        "SA"          :          "Vdd";
        "B0"          :          "GND";
        "B1"          :          "GND";
        "SB"          :          "Vdd";
        "S0"          :          "d";
        "S1"          :          "GND";
        "out"         :           "[26]";
******=> out = D'[37]

 INSTANCE "BASIC_BLOCK":physical NAME = INST4
        "A0"          :          "[24]";
        "A1"          :          "Vdd";
        "SA"          :          "d";
        "B0"          :          "[26]";
        "B1"          :          "Vdd";
        "SB"          :          "[24]";
        "S0"          :          "q0";
        "S1"          :          "GND";
        "out"          :          "[38]";
******=> out = [24] + D q0' + [26]q0
 INSTANCE "BASIC_BLOCK":physical NAME = INST5
        "A0"          :          "Vdd";
        "A1"          :          "GND";
        "SA"          :          "d";
        "B0"          :          "GND";
        "B1"          :          "GND";
        "SB"          :          "Vdd";
        "S0"          :          "q1";
        "S1"          :          "GND";
        "out"          :          "[39]";
******=> out = q1' D'
 INSTANCE "BASIC_BLOCK":physical NAME = INST6
        "A0"          :          "GND";
        "A1"          :          "[38]";
        "SA"          :          "Vdd";
        "B0"          :          "[39]";
        "B1"          :          "Vdd";
        "SB"          :          "[24]";
        "S0"          :          "n";
        "S1"          :          "GND";
        "out"         :           "[28]";
******=> out = [38]' N' + [24]N + [39]N

ENDMODEL;
```

Fig. 17. Actel netlist for vending machine design.

6.3. Technology mapping for LUT-based FPGAs

A K-input LUT is a digital memory that can implement any Boolean function of K variables. The K inputs can be thought of as addresses to a $2^K \times 1$ memory that stores the truth table of the Boolean function. We again assume that the overall functions to be implemented have been optimized and specified in the form of DAGs. The technology mapping problem can be understood as mapping sub-networks of the DAG that can be implemented by the available circuit elements. In our case, we will restrict ourselves to K-input LUTs. Consequently, the only restriction on any sub-network is that it has no more than K inputs. Two optimization goals for LUT synthesis can be identified: (i) minimization of the total number of LUTs, and (ii) minimizing the number of levels of LUTs. The former helps reduce the total area while the latter addresses the performance of the circuit in terms of programmable routing and logic block delays along the longest path. These two goals are often in conflict and hence algorithms primarily focus on one and modify the intermediate or final solution to account for the other as well.

6.4. Discussion

The key to all approaches in LUT-synthesis is the ability of the K-LUT to implement all functions of up to K variables. Because of this *completeness* property, it is no longer necessary to go through a costly library matching procedure; it is sufficient to count the number of inputs to a sub-network and verify that the number of inputs do not exceed the constraint K.

The first step is to make the given DAG K-feasible. By this we mean that each node which has greater than K inputs is recursively decomposed into sub-functions that use fewer inputs until all resulting nodes are feasible. This phase is also referred to as *decomposition*. Note that decomposition can be carried out for feasible nodes as well which can in practice lead to superior circuits. Figure 18 illustrates this by considering the implementation of an arbitrary 9 input function expressed in AND-OR form (with $K = 5$). Note, how decomposing the OR gate at the output into two OR gates enables one to pack the same function into only 2 LUTS resulting in a saving of 50%. In principle, we can even reduce the given Boolean network to a two-input network, the reason being that if we view the mapping process as one of packing gates into K-LUTs, then the smaller the gates, the more easy it is to pack them with less wasted space in each K-LUT.

The next step is known as *covering* and it helps to identify sub-networks that can be assigned to the same LUT. While efficient algorithms exist when the DAG to be synthesized is fanout-free, the problem is more complex in the presence of reconvergent paths and fanout nodes.

6.5. Decomposition techniques for infeasible nodes

Four methods have been proposed for the decomposition of infeasible nodes, i.e. those having greater than K inputs into a set of feasible nodes implementing the same function: (i) *disjoint decomposition*, (ii) *algebraic decomposition*, (iii) *AND-OR decomposition* and *(iv) Shannon cofactoring*.

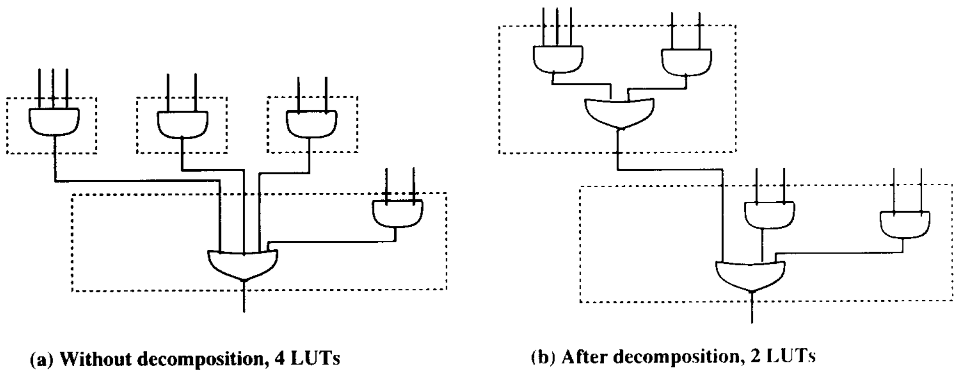

(a) Without decomposition, 4 LUTs
(b) After decomposition, 2 LUTs

Fig. 18. Decomposing feasible nodes can reduce LUT count as well.

6.5.1. Disjoint decomposition

This is based on a partition of the inputs into two disjoint sets referred to as the *bound* and the *free* set. Each of these sets, by definition, have fewer variables than the original. This is similar to the functional decomposition techniques mentioned for PLD synthesis. Here, we review one well known method based on *residues*. For any given partition, a residue function is obtained by replacing the variables in the free set by constant values. If the set of all possible residue functions for a given partition consists of the constants 0 or 1, or a single function h of the bound variables or its inverse h', then the partition is a disjoint decomposition with one extracted function. One has to be careful using this method for the search for feasible partitions can become exponential in nature. Moreover, it is insufficient in many cases such as for the majority function $s = ab + bc + ca$ that is part of a full-adder circuit.

6.5.2. Algebraic decomposition

This resembles some of the same methods used for multi-level synthesis. The idea is to identify algebraic factors in the expression of the given function that can be factored out thereby reducing the number of inputs needed at the cost of more LUTs.

6.5.3. AND-OR decomposition

Since both the AND and OR operators are associative and commutative, each such node can be divided into smaller nodes of the same type using any partition of the inputs. The approach works for any gate that represents a commutative and associative operator and is mainly used to reduce larger infeasible nodes to smaller infeasible nodes that are more practical for exhaustive search methods. The problem with this method is that it may use up a lot more LUTs than really required because of the manner of partitioning.

6.5.4. Shannon cofactoring

This method is based on the identity

$$f(x_1 \ldots x_j \ldots x_n) = x_j f(x_1 \ldots 1 \ldots x_n) + x_j' f(x_1 \ldots 0 \ldots x_n).$$

Thus, after cofactoring the function f depends on only 3 inputs, viz., x_j and the two cofactors. The latter depend on at most $n - 1$ variables and so the process can be recursively applied on them till all functions are feasible.

6.5.5. Other schemes

Some other schemes exist such as the Roth–Karp decomposition strategy or the Huffman-coding tree method to reduce each multiple input simple gate into a tree of two-input simple gates.

6.6. Bin-packing method for function covering

This method has been implemented in the Chortle-crf technology mapper [23]. It combines the AND-OR decomposition strategy with a covering algorithm similar to the library-based approach. The original network is assumed to be K-feasible and to consist of AND and OR nodes and is traversed from primary inputs to the primary outputs. A circuit implementing each node is constructed from the circuit implementing its immediate *fanin* nodes as follows:

- The first step is to try and combine as many fanin LUTs into a single one as possible. To determine if a group of fanin LUTs can be packed into one LUT, it is sufficient to count the total number of LUTs that are used by this group. Thus, the minimization of the LUTs can be viewed upon as a *bin-packing* problem. Here the boxes are the fanin LUTS, the number of inputs to that LUT denoting its size; and the bins are the new LUTs into which they are to be packed. The size of each bin is of course the feasibility constraint K. The First Fit Decreasing (FFD) algorithm is known to be optimal for boxes and bins with integer sizes less than or equal to 6. Hence, for smaller LUT implementations, this is the algorithm of choice. It begins with an empty list of bins and a sorted list of boxes, beginning with the largest box. Each box in the list is then placed in the first available bin into which it fits. If no such bin is available then a new bin is created and added to the end of the bin-list and the box is placed into the newly created bin. Packing boxes into the same bin implies the decomposition of the node being covered.
- The next step is to connect the LUTs defined by the packed bins. One method is to sort the bins by filled capacity and then connect the output of one bin to an unused input of one of the following bins. If no such bin exists a new LUT is created and is added to the root of the circuit. An advantage of this technique is that it minimizes the number of inputs used by the root LUT of the circuit. Since, this LUT becomes the box for the following node, a smaller size can result in fewer bins being needed for the bin-packing step of that node leading to an overall superior circuit.

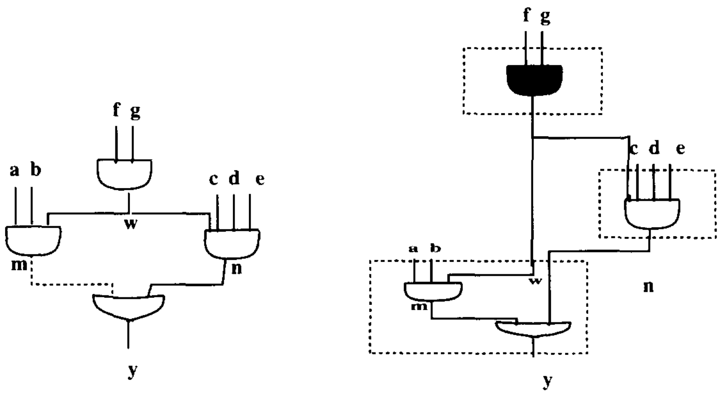
This method, though simple to understand, has the following drawbacks:

- It gives optimal results only for LUTs of size ≤ 6 (due to the FFD algorithm).
- It works only for fanout-free circuits. More general networks are handled by partitioning the network at fanout nodes into a forest of trees each of which is separately handled as above.
- The LUT connection scheme can often lead to taller trees which has an adverse effect on performance.

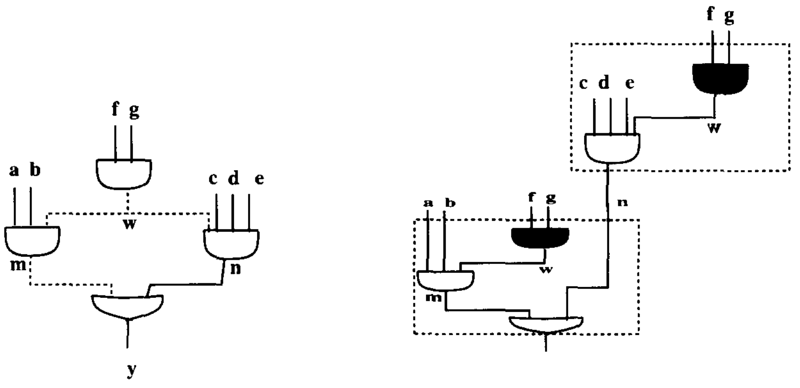
6.6.1. Covering in the presence of fanout

The mapping algorithm above can be improved in the presence of fanout by augmenting it with an *edge-visibility* technique. This was first implemented in the VISMAP technology mapper [24]. The method is as follows. Given a subnetwork, assign a label to each edge in the subnetwork. Labels can either mark an edge as *visible* in which case it is to be driven as the output of an LUT or *invisible* in which case the edge is to be realized entirely within a LUT. The actual label-assignment phase can be optimized using a pruning scheme. Certain combinations of edge-labels can be eliminated if it can be ascertained that they correspond to infeasible nodes.

Figure 19 shows a circuit and the LUT representation denoted by two different edge-labellings. In case (a), the logic for the fanout node w is not replicated and this requires 3 LUTs. In case (b), however, the edge-labellings are such that only 2 LUTs are needed. In this case the logic for the fanout node is replicated. However, note that if this logic needs more inputs, then replicating logic becomes less attractive. In practice, therefore, it is not possible to state whether replication or covering reconvergent paths together result in fewer LUTs in all cases. Hence, one needs to exhaustively check all edge-labellings and retain the one that leads to a circuit needing the fewest LUTs.



(a) Logic for fanout node is not replicated. 3–LUT solution.



(b) Logic for fanout node is replicated. 2–LUT solution.

Fig. 19. Mapping of fanout nodes using edge-visibility techniques. Dotted edges are invisible while other edges are visible.

6.7. *Network-flow approach to LUT synthesis*

One of the problems mentioned in the bin-packing scheme is the fact that it may lead to taller trees. An alternative scheme which uses network-flow techniques has also been suggested which directly tries to compute a minimum-height K-feasible solution to the problem. The first part of the algorithm proceeds from the inputs towards the outputs in a topological order, and assigns a label to each node in the DAG. In the final representation each label represents the depth of the optimal K-LUT mapping solution for that node. Thus, the goal of the labeling phase is to minimize the maximum label number used while maintaining feasibility requirements. The second step then works from the primary outputs upward towards the primary inputs generating a network of K-LUTs which is logically equivalent to the original network.

6.7.1. *Labeling phase*

Initially all primary inputs are assigned a level $l = 0$. The algorithm labels each node in a topological fashion, i.e. when a node v is considered, all its predecessor nodes, denoted as N_v will already have been assigned labels. Recall that the label given to a node indicates the height of a minimum LUT-tree realizing that node's function. Also note that in the labeling phase, the authors consider each node in isolation, i.e. it does not matter if the LUT-scheme for one node conflicts with the one used for labeling another node. This is accounted for in the mapping phase.

Suppose p is the maximum label assigned to any node in N_v. Then, it is clear that the label of node v cannot be less than p, since otherwise we could use the same scheme to generate a LUT-tree of height less than p for that node. In fact, it can be shown that v will either receive a label p or a label $p + 1$. The latter follows from the fact that the original network is K-feasible and the fact that all inputs to v can be realized using LUT-trees of height less than or equal to p. The only question that remains is to determine if a p-deep LUT tree exists that can implement v.

This is where the network-flow techniques are useful. From the given network, the authors first create an alternative network by collapsing all nodes in the original network, say N_1, with label $\geq p$ into a single node. Then they replace each node w by a pair w_1, w_2 and add a directed edge $w_1 \longrightarrow w_2$ with unit capacity. Each edge in N_1 connecting nodes m and n is replaced by a directed edge from m_2 to n_1 with infinite capacity. A dummy source (target) node can be added and connected to all primary inputs (output) again with an edge of infinite cost. Let the resulting network be called N_2.

Now let F be the maximum feasible flow in N_2. This can be determined in polynomial time using the method of augmenting paths. If $F > K$, then it implies by the max-flow min-cut theorem and the above transformation that the minimum cut X_v, X_v' separating the source from the target includes more than K nodes on the source side in N_1 that are connected to the target side. Hence, it is not possible to realize the target node v using only K-LUTs without increasing the height by 1. Thus, the labeling rule can be stated succinctly as follows: "If the maximum flow in the transformed network is less than or equal to K, then label the node as p; else label it as $p + 1$, where p is the maximum label assigned to any predecessor node".

It can be noted that in general the minimum-height K-feasible cut is not unique. Intuitively one wants as large an X_v' (nodes on the target side of the cut) as possible. This is because in the mapping phase all these nodes get assigned to the same LUT.

6.7.2. Mapping phase

Let L denote the set containing all outputs to be implemented using K-LUTs. Initially L contains all the primary output nodes. For each node v in L, let X_v, X'_v denote the minimum-height K-feasible cut that had been generated for v during the labeling phase. We then introduce a K-LUT for v using as inputs those crossing the cut. Note that this process subsumes all other nodes in X'_v. This can be likened to the covering of reconvergent paths within the same LUT as was done before using visibility methods. The process is then repeated by replacing L with the set $L-\{v\}+input(v)$ where $input(v)$ refers to the set of nodes that correspond to the input signals of the cut. Again, it may be possible that the same node occurs in X'_v and X'_u for two (or more) nodes in L in which case it gets replicated automatically.

7. Routing algorithms for FPGA designs

7.1. Row-based FPGA model

Such models are exemplified by the Actel family of FPGAs[9,10]. We review the basic architecture of the array and then discuss the routing issues. Finally a segmented channel router is presented.

7.1.1. The Actel FPGA architecture

Wiring resources Unlike traditional gate arrays, the routing channels are not empty areas where customized metallization can be performed. Instead, they contain predefined wiring segments of various lengths. These wiring segments are interconnected using a two terminal programmable element called *antifuse*. [2] The antifuse plays the role of a via in this FPGA.

Two categories of antifuses can be identified depending on where they occur:

- A horizontal fuse or *hfuse* links two adjacent horizontal segments within a channel. This enables longer segments to be realized.
- A *cross* fuse links a horizontal segment with an intersecting vertical segment.

Each of the cell outputs and inputs are connected to a dedicated vertical segment which is used to connect them to a horizontal track in an adjacent channel. A vertical segment typically spans only 1 row of cells for inputs and about 4–5 rows for outputs; horizontal segments span two or more modules within a row. Thus inputs can only be connected within the channel directly above or below the cell. Furthermore, for some nets, it may not be possible to allocate cells such that all its terminals lie within the span of the vertical output segment. To accommodate such nets, certain long vertical segments are also provided in the array. The horizontal routing for such nets should preferably avoid concatenated wire segments so as to avoid increasing delay. Special wiring exists for clock distribution to each module.

Routing model Traditional channel routing cannot be directly applied since it does not take into account the restricted nature of the wiring resources. A modified channel routing problem known as

[2] The antifuse represents a one-time programmable element comprising of a diffusion layer placed over polysilicon with a dielectric between them. When a voltage exceeding 18 V is applied between the two terminals the dielectric irreversibly breaks down and starts conducting.

segmented channel routing is therefore needed for such FPGAs. The name derives from the fact that each channel comprises of many tracks which are divided into segments of non-uniform lengths. If all segments have the same size, then the problem is called an *uniform* SR problem; otherwise it is a *non-uniform* problem. The segmented channel routing problem can be formally stated as "assigning segments to nets so as to make all connections within the given routing resources (tracks and fuses)." The objectives are foremost to achieve 100% connections, secondly to minimize the number of fuses traversed by each net so as to optimize the timing characteristics. The following points differentiate SR problems from traditional channel routers:

- *Channel density*: This refers to the minimum number of wiring tracks that are required to make all connections. For channel routing, two nets can be assigned the same track if their horizontal spans do not overlap; consequently the density is given by the maximum number of nets crossing any column within the channel. In case of SR, each net has to be assigned to a different segment. Corresponding to the channel density, a segment density can be defined for the uniform model, after extending each net to the nearest adjacent segment switch. For the non-uniform model, there is no such apriori measure as the density depends on the particular segment length distribution.

  For example, Fig. 20 shows the routing for a 2-segmented uniform model with 3 tracks per channel. Note that in a traditional model, the same routing could have been done using only 2 tracks by assigning nets 1 and 3 the same track. However this is not possible in the segmented model because the segment assigned to net 1 overlaps the starting terminal of net 3. For our example, this gives a segment density of 3 and so the routing is optimal.
- *Single-entry logic cell*: In gate arrays, the inputs can be assumed to be available simultaneously in both the adjacent channels. For FPGAs, this may not be true. The input is accessible from either the channel above or below but not both. Thus, there are no vertical constraints because of the fact that at most one terminal can enter the channel at any given column.
- *No doglegs*: This is because programming two fuses on the same column may lead to other unwanted fuses to get programmed as well. This also degrades signal performance as each fuse in the path of the signal adds to the resistance of the net.
- *RC constraints*: The nets are now RC trees and in order to maintain timing integrity, it is preferable that they traverse as few antifuses as possible. Typically each net needs only three

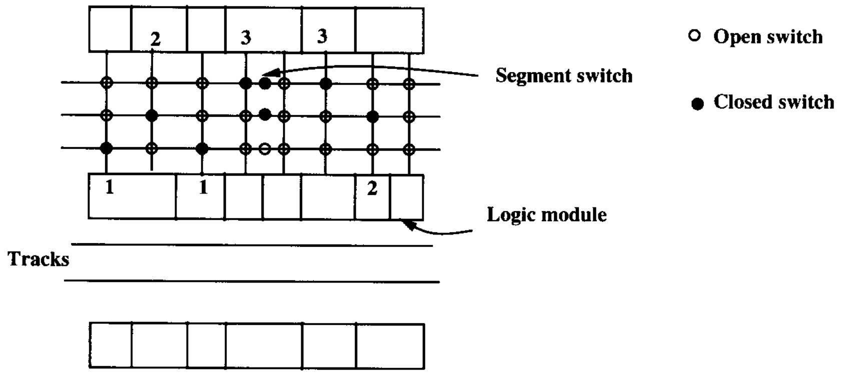
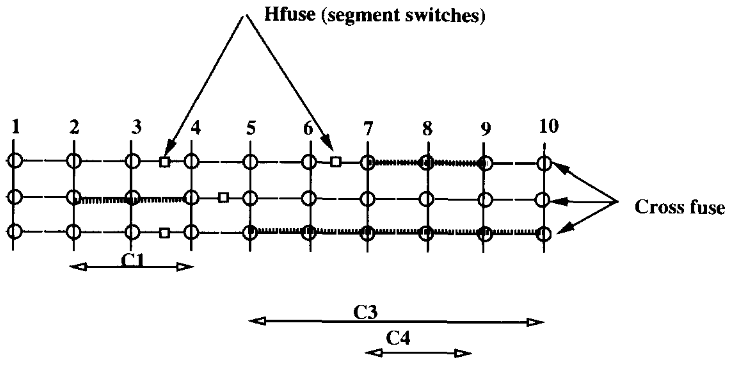


Fig. 20. Segmented channel routing model for FPGAs.



A three track segmented channel routing problem. Track 1 has two hfuses and so can be split into three discrete segments while the other two tracks can be divided into two segments. Note vertical connections can be brought into any track by closing the appropriate cross fuse while longer horizontal segments can be realized by programming the segment switch.

Fig. 21. A segmented channel routing problem and a 1-segment solution to it.

antifuses: two cross fuses to connect the dedicated vertical segments to the horizontal track and one horizontal fuse to connect adjacent horizontal segments.

7.1.2. *Algorithms*

In this section, we review a new type of detailed routing algorithm geared towards segmented channels. For larger FPGAs, a global router may first be invoked. The process entails splitting each multi-pin net into a set of connections and then assigning each connection to a specific channel. The global router can use the vertical feed-through segments for nets that span multiple channels. However, the issues involved are similar to regular custom and semi-custom designs [25].

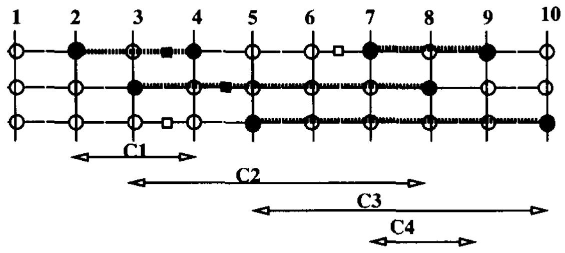
Let us define a K-segment routing problem as one of connecting all nets such that no net occupies more than K segments. If the total lengths of the segments are also targeted for minimization, it is called a K-segment solution with *delay optimization.*

7.1.3. *1-segment channel routing*

The 1-segment problem wherein each net occupies only one segment can be optimally solved by the following greedy algorithm which is essentially a modification of the left-edge algorithm. Assign the connections in the order of increasing left ends. For each connection, find a set of tracks which are free at that column and in which the connection would occupy one segment only. From these, choose the segment whose right-end is farthest to the left. Intuitively, this maximizes the number of free segments for later columns. Since, it is necessary to check each track for each connection, the run time of the algorithm would be $O(NT)$ where N and T denote the number of nets (connections) and number of tracks respectively. Figure 21 shows a solution for a three net three track example.

7.1.4. *K-segment channel routing*

Here, we allow a net to occupy as many as K adjacent segments. The motivation for this problem stems from the fact that the 1-segment solution is quite inefficient in terms of track usage. Hence, for a fixed channel capacity it may not be capable of finding a solution. Not surprisingly, though, the general K-segment channel routing problem turns out to be NP-complete for all K greater than 1



2-segment solution with no delay optimization. The assignment tree is shown below. This solution uses two segment switches and has a total length in terms of columns spanned by the segments in use of 27.

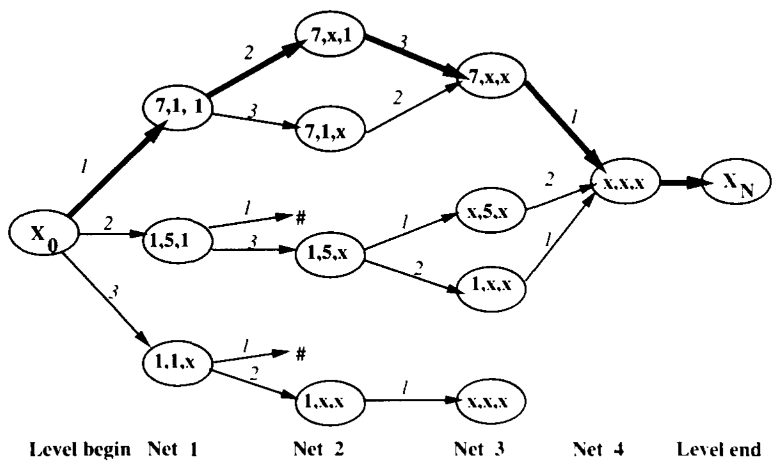


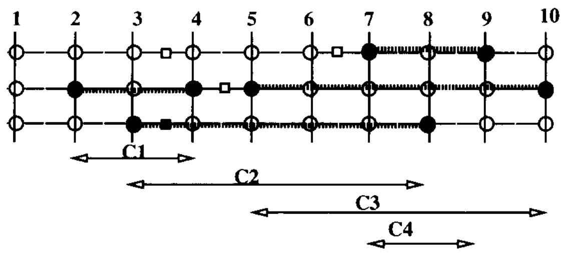
Fig. 22. 2-segmented channel routing solution.

[26]. The reason being that assignment of a track to a net can no longer proceed in a greedy fashion. An exponential time branch and bound algorithm which is guaranteed to find an optimal solution if one exists has been proposed. The main idea of the algorithm is to construct an *assignment tree* which represents the effect of optionally assigning each connection to each track.

Each node of the assignment tree is called a *frontier*. A frontier is simply a T-tuple of the form $(x[1], x[2], \ldots, x[T])$ where $x[j]$ is the leftmost column in track j in which the segment present in that column is not occupied. Clearly the initial frontier is given by an all-ones entry denoted as $x_0 = (1, 1, \ldots, 1)$. Each assignment of a net to one or more segments causes one of the entries in the current frontier to change.

The assignment tree is built up in stages or *levels*. Level i corresponds to all possible frontier nodes that can result subsequent to assigning tracks to the first i connections $C_1, C_2, \ldots, C_i$. Thus, for instance, the first level contains the single node x_0 while the last level contains a single frontier node x_N which denotes the state after all nets have been assigned. Once x_N is reached, a valid routing can be obtained by retracing a path from x_N to x_0. Also, if no nodes get added at an intermediate level, it means that the given problem has no valid solution possible.

In order to construct Level $i+1$ from Level i, we first enumerate all tracks which are free at $left(C_{i+1})$, i.e. the leftmost column of the $i+1$-st connection. This enumeration is done on the basis of the frontier values, $x[j]$, of nodes at level i. Suppose $x_i[t] < left(C_{i+1})$ and C_{i+1} would occupy



2-segmented channel solution with delay optimization. Note this solution uses only 1 segment switch and has a total length of 24. The assignment tree is shown below.

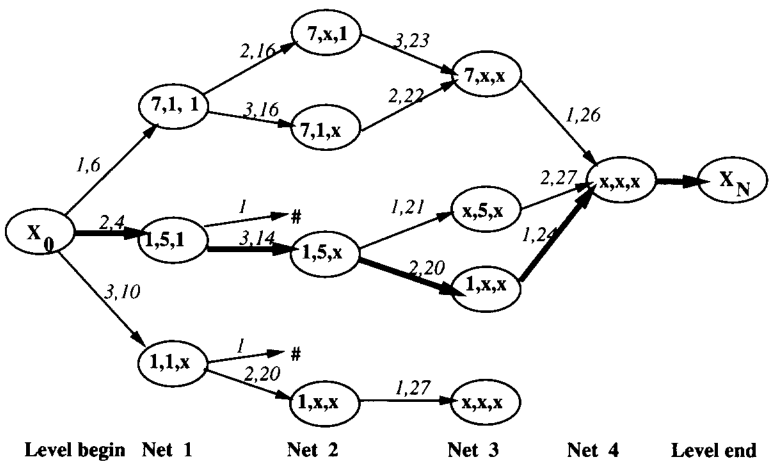


Fig. 23. 2-segmented channel routing solution with delay optimization.

less than K segments in track t. Then C_{i+1} can be assigned to track t and the new frontier x_{i+1} is created by advancing the value of $x_i[t]$. An edge is added between x_i and x_{i+1} and is labeled by t. Note though if x_{i+1} already exists then we only add the corresponding edge without duplicating nodes. Figure 22 shows the assignment tree for a 2-segment solution to the same routing problem considered above with one additional connection. The final path found is shown by bold lines and the solution it represents is shown separately. Paths marked with a # cannot be expanded because the net would occupy more than two segments if assigned to that track. Note that the additional connection, C_2, is impossible in the 1-segment model since all tracks have an intervening switch between columns 3 and 5.

Delay optimizations: One disadvantage of the K-segment problem is the fact that each extra segment increases the delay experienced by the net owing to the presence of an extra switch in the path. Hence, it is desirable that the number of segments per net be minimized. Also larger segments add to the capacitance of the net and hence the delay. These factors can however be easily incorporated into the above algorithm by adding a weight in addition to the track label to each edge. The weight represents some cost function such as the total length in terms of columns spanned of all segments assigned to $C_1, \ldots, C_{i+1}$ for an edge between Levels i and $i+1$. When multiple edges fan-in to a single node, the edge with minimum cost is chosen. Ties can be broken randomly. Similarly, when the backtrace is performed it is only necessary to follow the minimum weight edge from each node in

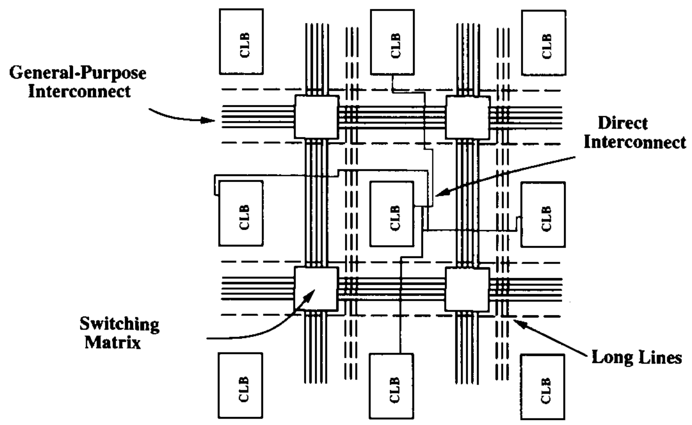


Fig. 24. Wiring resources in a Xilinx FPGA.

case of choices. The assignment tree with weights added is shown in Fig. 23. Note that this solution reduces the overall path length by 3 and also reduces the number of segment fuses that need to be programmed by 1.

7.2. *MPGA style architectures*

This architecture corresponds to the familiar channelled gate array architecture. The Xilinx family of FPGAs [27,8] falls in this category. The device consists of an array of programmable logic cells called Configurable Logic Blocks (CLBs) with wiring surrounded by a ring of configurable I/O blocks, see Fig. 24. The latest model in this family of FPGAs is the XC4000 family which consists of arrays from 8 × 8 to 30 × 30 blocks with 64 to 240 I/O pads.

7.2.1. *Wiring resources*

Wiring resources comprise of horizontal and vertical channels with switchboxes at each intersection. Within the switchbox, pass transistors are provided to switch an incoming wire to one of the three alternate directions. Longer busses are also provided to minimize the number of switches in each net path. Switches are provided for these longer nets at every other row and column intersection. Special wiring for clock and some dedicated wiring for carry calculations are also provided. In addition, all function inputs are connected directly to all single length wire segments; while the outputs are connected to a subset of both vertical and horizontal segments.

A more general view of the overall routing architecture [28] is shown in Fig. 25. The L boxes refer to the logic cells (i.e. the CLBs); the C (*connection*) blocks are rectangular switch boxes that connect the CLB pins to the routing channels via programmable switches; while the S(*switch*) blocks connect the wiring segments in one channel to those in another. The configuration of the C and S blocks largely determine the ease of routability of an FPGA and thus help differentiate between different members of this class. Clearly, the more the wiring segments that can be switched, the easier it is to route. The goal of routing is to configure all the CIPs (configurable interconnection points) within the capabilities of the switchboxes so as to obtain conflict free routes for all the nets. A sample C block structure is shown in Fig. 25(b). Each dark dot indicates a CIP which can be programmed to either make or break the connection.

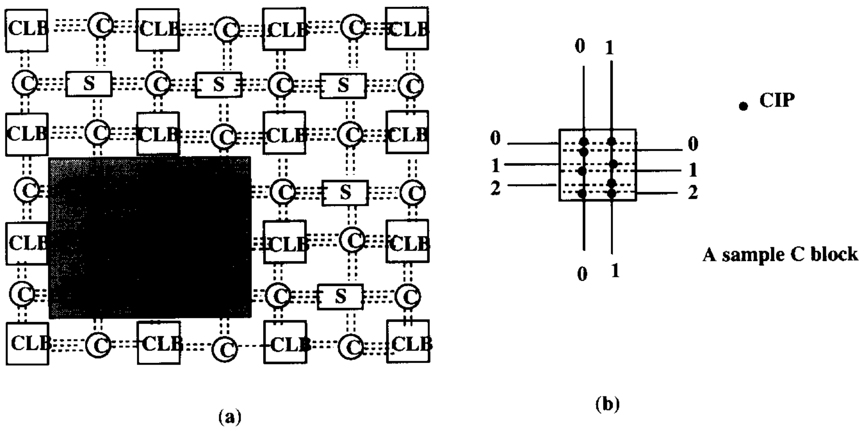


Fig. 25. (a) General FPGA routing model; (b) A sample C block.

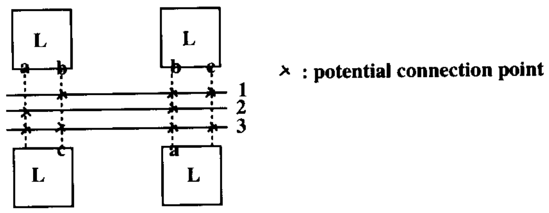


Fig. 26. Potential FPGA routing conflicts.

The features of the FPGA which are especially important from a routing standpoint are as follows:

- The number of wiring segments is predetermined. It is entirely possible that the successful routing of some net may rely on the assignment of a specific wiring segment to that net. If a sequential style of routing such as a maze router is used, then care has to be taken [29] to ensure that such *essential* segments are assigned appropriately. Figure 26 shows an instance wherein the wrong assignment of a segment to net A will lead to a failed connection for either net B or net C.
- If the CLB is implemented in the form of a table-lookup using SRAM [30], then the inputs to the function table are interchangeable. The router takes a request to route to a input as a request to route to any unassigned input. Thus for a maze router, the necessary modification is to start the expansion from all the unassigned input terminals simultaneously.
- CIPs can be of two types: *cross* where the CIP comprises of a bidirectional pass gate and an SRAM cell; and *muxcip* which represents a member of a mutually exclusive group and is controlled by decoding one or more SRAM cells in parallel. The implication for the router is that a single path cannot use more than one member of a decoded CIP group.
- Unlike gate-arrays there can be no doglegs in channels in an FPGA since the metallization pattern is fixed. Thus routing within a channel can use a simple left-edge kind of algorithm.
- Proper allocation of nets to busses of appropriate length is needed for maintaining tighter delay bounds on nets.
- Allocation of nets must also take into account the switchbox resources.

7.2.2. Algorithms

Though, in principle, we can use any MPGA routing algorithm, the restricted nature of the routing segments and of the switching fabric mandates better algorithms. Otherwise, we may find several instances of routing conflicts that lead to failed connections. In this section, we review an FPGA router for matrix-based architectures [29]. This router consists of two phase:

- Global routing wherein each net is assigned a path in the grid graph by assigning it a sequence of channel segments. Its main purpose is to distribute the channel densities among the channels so as to facilitate the detailed routing phase. This step does not require any specific knowledge of FPGA architectures and so a regular MPGA algorithm is used.
- Detailed Coarse Graph Expansion (CGE) router: The goal of this router is to assign specific wiring segments to each net so as to generate the global route determined in the previous step.

The basic idea is as follows:

Global graph expansion: Given a global route for each net, find all possible wire segments that can be used to extend the path from one channel of its global route to the next until either the target is met or no more paths remain to be expanded. The expansion depends on the switching capability of the C and S blocks and the connections used up by the previously routed nets. An example of this process is illustrated in Fig. 27 for a net connecting pin 5 of logic block at coordinates $< 1, 1 >$ to pin 2 of logic block at coordinates $< 3, 3 >$. The global route is shown on the left and the detailed expansion yields three paths. Paths marked with a # lead to blockages.

Path sorting: All the paths found in the graph expansion process are placed in a single path list and are ordered in the increasing order of costs. Costs for each path can be based on two factors: (a) demand on the wiring segments by other nets, (b) timing issues that can be modeled by counting the number of switches used in the path. For example, if a certain segment is required by many nets, then its cost is higher so as to discourage its use as far as possible. On the other hand, if there is only one choice, then it is given a low cost so as to include it in the current expansion.

Update: The net corresponding to the lowest cost path is selected and its route is used to update the costs for the channel segments to be used in the routing of the other nets. This can change the order of the paths in the path-list.

Pruning the search tree: The overheads to maintain all possible paths is very high. Consequently, it is better to maintain a certain limited number of paths. In [29], the authors use two pruning factors limiting the fanout from the root and from intermediate nodes in the coarse grid graph. This method does not mean that the other paths are removed from consideration for all times. If a route cannot be found for a certain net, then the pruning factors are gradually increased till we are successful. A rip-up and reroute strategy is done only when complete expansion of the coarse graph fails to yield a connection.

However, if the switch boxes are fairly simple and provide limited options, then the exponential increase in paths does not happen and consequently the elaborate pruning features may not be really needed.[3]

[3] The latest version of Xilinx FPGAs, the XC4000, for instance, have increased the number of tracks but simplified the S-block structure. Each wiring segment that enters the S-block can only connect to three others, which is half the number found in the XC3000.

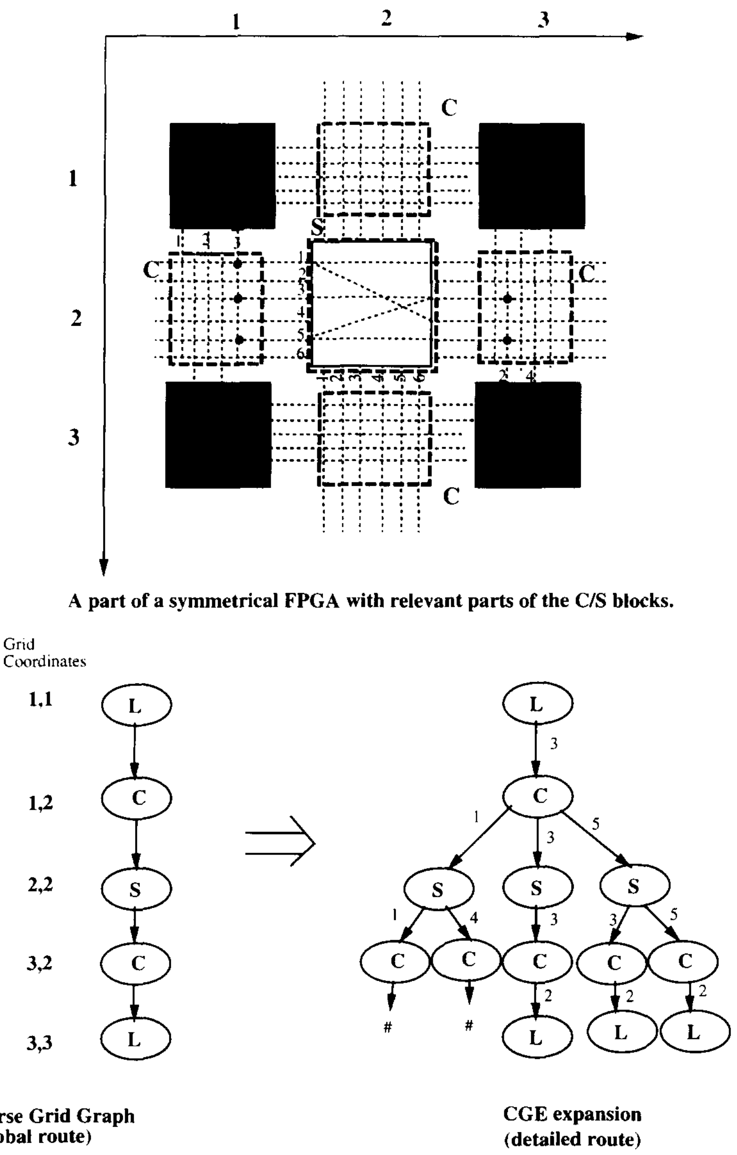

1
2
3
1
2
3
C
S
C
C
C
A part of a symmetrical FPGA with relevant parts of the C/S blocks.
Grid Coordinates
1,1
1,2
2,2
3,2
3,3
L
C
S
C
L
Coarse Grid Graph (global route)
CGE expansion (detailed route)
#
#

Fig. 27. Coarse grid graph router for matrix-based FPGAs.

Cost function: In the path ordering phase, each of the paths P is assigned a cost $c(P) = \sum_{e \in P}[c_f(e) + w_t \cdot n_P]$. $c_f(e)$ represents a cost for the edge e based on the demand for the edge in paths for other nets. For an edge e that has j other occurrences $e_1, e_2, \ldots, e_j$, the cost $c_f(e)$ is given by $c_f(e) = \sum_j 1/alt(e_j)$ where $alt(e_j)$ is the number of edges in parallel with e_j. For instance, in Fig. 27, the edge connecting C(1,2) and S(3,2) has three alternative track assignments possible and hence the *alt* value for it is 3.

Because of the summing process in $c_f(e)$, the more graphs e occurs in, the higher will be the cost. This reflects that edge e is in high demand and should be avoided if possible. Also if $alt(e)$ happens to be zero, then the edge is essential as there are no alternatives. When the cost computation reveals an essential path, the router attempts to schedule it for routing first. The second cost component determines the timing cost of the net in terms of the number of S and L blocks it has traversed so far. If a certain net is deemed timing-critical, then the latter cost can be given a larger weight in determining the next path to be expanded. Note, that many FPGAs also provide long-range connections which skip some S boxes along the way. This might need adjusting the expression for the timing cost appropriately.

The next segment determination and the update steps in the expansion is achieved using a black-box technique. This makes the CGE somewhat independent of the exact architectural features of the S and L blocks in different FPGA implementations.

7.3. Sea of gate FPGAs

The FPGA architectures that have so far been considered have a clear separation between logic cells and interconnection resources. Recent sea-of-gate offerings differ in the sense that they provide more but simpler cells. Also there is no separate routing structure provided. The routes have to be done by the programming of the basic cells themselves. Thus, the architecture resembles a sea-of-gates style. Some offerings which fall in this category include the Labyrinth array (Apple Computer and Concurrent Logic Inc.), the CFA family by Concurrent Logic and the CAL family from Algotronix.

A sample Labyrinth cell can be configured in 4 possible ways:

- As a NAND/XOR pair (a half-adder with inverted carry).
- As a D flip-flop.
- As an AABB routing cell where the A inputs are transferred to the A outputs and the B inputs are transferred to the B outputs.
- As an ABBA routing cell where the A inputs are transferred to the B outputs and the B inputs are transferred to the A outputs.

For the Xilinx and Actel style of FPGA architectures, it is appropriate to perform the placement first using estimated routability and then do the routing separately. However, for sea-of-gate structures with fine-grained components, it is better to treat placement and routing simultaneously. Consequently, the layout problem consists of an initial place and route of cells followed by iterative improvement. During this process cells can get re-placed or re-oriented. Routing typically uses the penalty-driven iterative improvement approach or some other form of rip-up and reroute method.

7.3.1. Routing algorithms

Since input pins are often interchangeable, the router should consider all unassigned pins on the target cell as potential targets. Pfister [31] employs a modified version of the Mighty router to make connections. The work of Beetem [32] is an interesting extension to the penalty driven iterative algorithm of Linsker [33] in that the targets themselves can also be mobile. The main steps of the algorithm are as follows:

(1) **While** improvement exists **do**
(2) **For** each net **do**
(3) Generate a costed wavefront expansion. The cost function includes penalties for path lengths, overlaps, obstacle neighborhoods, and force costs.
  (a) The path length is the fixed cost to expand to an adjacent cell and penalizes long routes.

(b) The overlap costs penalizes illegal overlaps. This cost is small in the beginning but becomes larger in later passes. This is similar to the temperature schedule in simulated annealing and allows more freedom initially for good solutions.
(c) The obstacle neighborhood costs tries to prevent nets from being routed in congested areas and tries to obtain uniform spread of nets.
(d) The force cost component simulates a spring potential function and pulls mobile targets near logically adjacent components. This is an adaptation of the Force-directed placement strategy.

(4) Continue the expansion till no more cells remain to be expanded or the least cost cell in the current frontier is more costly than the cheapest path to the target. Note that since overlaps are permitted, a path to the target is assured for all nets irrespective of the net order used.
(5) Backtrace the path found and mark all such cells *fixed* for this iteration.
end for
(6) Make all cells mobile again; update the overlap and neighborhood costs and repeat.

Some additional features of the algorithm including encouraging the matching of AA and a BB half cell during routing by assigning it a small overlap credit. Certain nets are driven by a logical '1' instead of a cell output. For such cases it is immaterial where the source '1' is located. Hence, for such nets the role of the source and the target are reversed and the source '1' is treated as a mobile target.

7.4. Theoretical modeling for FPGA routing

The last topic of discussion is a theoretical model, based on probabilistic analysis, intended to provide better insight for FPGA routing. It is a synopsis of the work that appears in [34]. The model is applicable to symmetrical FPGAs, such as Xilinx devices, with tracks that consist of short segments spanning the size of one logic block. The output of the analysis will be a theoretical estimate of the probability of successfully routing a connection.

7.4.1. Assumptions

The actual process of FPGA routing is highly dependent on the particular FPGA architecture, problem instance, etc. and so any theoretical model can only hope to approach it to a sufficient degree of satisfaction to be acceptable. The advantage is to derive a more cost-effective way, than actual experiments, for making architectural studies and deriving better routing algorithms. Several simplifying assumptions have been made and have to be understood in order to extend the model for other routing architectures.

(1) It is assumed that the routing will consist of a global and a detailed phase. Only the detailed phase is subjected to the stochastic modeling. Thus, we assume that nets will be routed in sequence and furthermore the global route for each net has been determined, in advance, based on overall factors such as wirelength, channel congestion, etc.
(2) We assume that the circuit has a total of C_T two-point connections and is to be routed in an FPGA with $N \times N$ logic blocks. The length of each connection is drawn from a probability distribution, P_L. It is further assumed that P_L is geometric with mean length $\overline{R}$ which has the physical interpretation that at each C block along the path of a connection, the connection will terminate with probability $1/\overline{R}$ and will continue to the next C block with probability $1 - 1/\overline{R}$.

(3) It is further assumed that the number of connections per cell can be drawn independently from a Poisson distribution with parameter λ, where λ is defined as the ratio of the total number of connections in a circuit to the total number of cells in the array.
(4) Tracks are assumed to consist of short segments that span only one logic block.

While Assumption 1 is acceptable for most FPGAs, that made in Assumption 4 is not for the simple reason that most FPGAs provide longer segments spanning several logic blocks for realizing the longer or global nets. The reason for making this assumption is to make use of an interesting result due to El Gamal developed for Master Slice circuits. It was shown that under Assumptions 2,3, and 4, the densities of the channel segments will also be Poisson distributed, with the average value given by $\lambda_g = \lambda\overline{R}/2$. The result has also been borne out in practice for a number of circuits implemented using FPGAs [34]. However, the result is not an accurate approximation when longer segments are used.

7.4.2. Modeling the events

The key to the stochastic modeling of FPGA routing is the calculation of the probabilities for the events $R_{C_1}, R_{C_2}, \ldots$ where R_{C_i} is the event that the *i*th connection is successfully routed. Furthermore, if we define *routability* as the average probability of completing a connection, we get the following formula:

$$Routability = \frac{1}{C_T}\sum_{i=1}^{i=C_T} P(R_{C_i}),$$

where $P(R_{C_i})$ is the probability of successfully routing connection C_i.

The event R_{C_i}, itself can be split into 3 events which are indicated in Fig. 28:
(1) Event X_1: Connection of source pin to the channel C-block,
(2) Events $S_1, S_2, \ldots, S_n$: Event S_i refers to the traversal of the *i*th S-block,
(3) Event X_2: Connection of the net to the final pin of the target logic block.

Clearly, for the event R_{C_i} to be successful, all the above events have to be successful in order. Thus, for a path of length L_n we have the following relation:

$$P(R_{C_i}|L_n) = P(X_1)P(S_1|X_1)P(S_2|X_1S_1)\ldots P(S_n|X_1S_1\ldots S_{n-1})P(X_2|X_1S_1\ldots S_n). \tag{3}$$

Knowing all the probabilities on the right hand side, we can come up with a general formula for routing net i as

$$\begin{aligned} P(R_{C_i}) &= \sum_{l=0}^{l=l_{\max}} P(L_l)\cdot P(R_{C_i}|L_l) \\ &= \sum_{l=0}^{l=l_{\max}} pq^{l-1}\cdot P(R_{C_i}|L_l) \end{aligned} \tag{4}$$

where $p = 1/\overline{R}$ and $q = 1 - p$ can be obtained based on the fact that $P(L)$ is a geometric distribution.

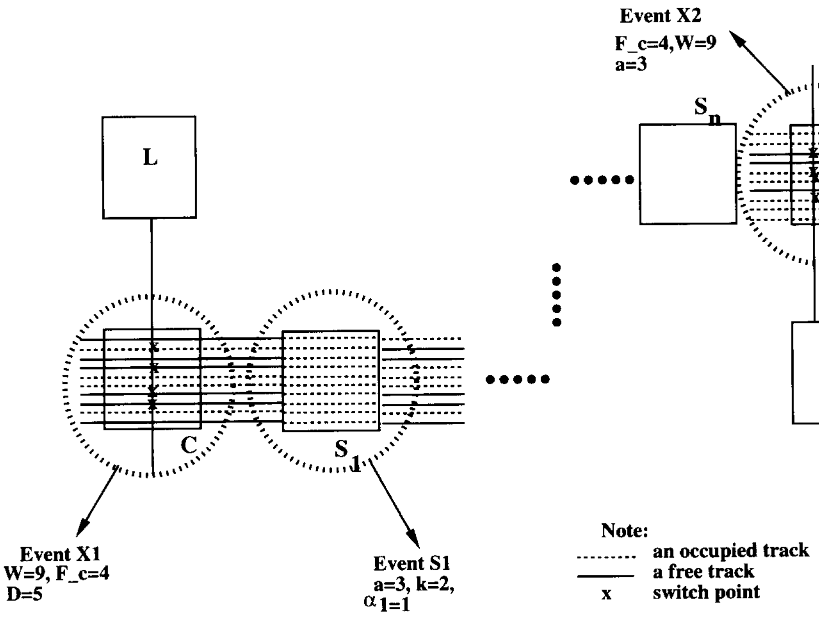


Fig. 28. Events occurring in forming a connection.

Evaluating $P(X_1)$*:* X_1 is successful if at least one of the F_c tracks to which the output pin is connected is free at the time this net is scheduled for routing. If we let the random variable X denote the number of free tracks among these F_c connections that are free, then we have

$$P(X_1) = \sum_{i=1}^{i=F_c} P(X = i).$$

Since the channel densities are assumed to be Poisson distributed, it is possible to consider only the F_c tracks rather than all the W tracks in the calculation of the probabilities. The scaled Poisson process will have a mean $\lambda_g F_c/W$.

Consequently, we have

$$P(X_1) = \sum_{i=1}^{i=F_c} p(\lambda_g F_c/W, i) \tag{5}$$

where $p(\lambda, x) = \mathrm{e}^{-\lambda}\lambda^x/x!$ is the Poisson probability function.

Evaluating $P(S_i|X_1 S_1 \ldots S_{i-1})$*:* To simplify notation, we denote the above conditional probability by $P(S_i|Y)$ where Y is used to denote the event that X_1 and all prior switchboxes have been successfully routed.

Assume that at the ith S-block, there are a possible incoming tracks and k possible outgoing tracks that are free. Since, the net can either go straight through (event E_1) or turn at the S-block (event E_2), we have two possible mutually exclusive events here. Consider the case when there is no turn. Let us assume symmetrical S-blocks, i.e. each track is connected to the same number of tracks, say

α_1, on the opposite side, and α_2 on the two other sides. Again by considering a scaled down Poisson process, we can compute this probability as

$$P(S_i|Y) = P(E_1) \sum_{a=1}^{a=a_{\max}} P(X=a|X)p(\lambda_g \frac{\alpha_1 a}{W}, \alpha_1 a - k)$$
$$+P(E_2) \sum_{a=1}^{a=a_{\max}} P(X=a|X)p(\lambda_g \frac{\alpha_2 a}{W}, \alpha_2 a - k). \quad (6)$$

Here, X refers to the random variable denoting the number of incoming tracks that can be assigned to the net. $a_{\max}$ takes on the value F_c for S_1 and W for for all other cases because of the nature of the routing blocks.

Now, from Bayes rule we have $P(X=a|X) = P(X=a)/\sum_{i=1}^{a_{\max}} P(X=i)$ and furthermore from computation of $P(X_1)$ we know that $P(X=a) = p(\lambda_g a_{\max}/W, i)$. The values $P(E_1)$ and $P(E_2)$ have to be determined experimentally.

Evaluating $P(X_2)$*:* Here the problem is to evaluate the probability of the event that given one or more tracks were available at the outgoing side of the last S block, what is the probability that one or more of these tracks connects to the appropriate logic block pin. Again let us assume that a tracks are available for potential connection. The number of ways of choosing F_c of the W total tracks such that none of the a marked tracks belong to this set is clearly the ratio ${}_{(W-F_c)}C_a / {}_W C_a$ where ${}_W C_a$ is the number of combinations of W things taken a at a time. Consequently, the probability for the event X_2 is given by

$$P(X_2|X_1S_1\ldots S_n) = 1 - \sum_{a=1}^{a=W} \frac{P(X=a)}{\sum_{i=1} W \cdot P(X=i)} \frac{{}_{(W-F_c)}C_a}{{}_W C_a}. \quad (7)$$

7.4.3. Parameter estimation

The parameters that are required to fit the above equations are as follows:

(1) $N, W, l_{\max}$ which can be accurately known from a given FPGA specification,
(2) $C_T, \overline{R}$ which can be generated from a given routing instance,
(3) $P(E_1), P(E_2)$ which have to be empirically derived from perhaps a lot of similar actual FPGA solutions,
(4) $F_c, F_s, \alpha_1, \alpha_2$ which again are fixed for a given C and S block topology.

In [34], the authors compare the results to theoretical prediction versus actual values from applying their FPGA router to actual circuits. Their conclusion is that the two display a surprisingly close match with the theoretical estimate of routability on the average being around 5 percentage points below (s.d. is also around 5) the experimental values. The difference is higher for low values of F_c due to the inaccuracies of the model in describing good C-block topologies. From experimentation, the authors conclude that routability is low for low values of F_c and only approaches 100% as F_c approaches one-half of W. Improving S-block connectivity also improves routability but again for 100% routability must be accompanied by F_c greater than or equal to $W/2$. Another interesting observation was that connections pass through S-blocks straight through more than 70% of the time. Consequently, values of F_s that correspond to higher α_1 values produce markedly greater returns in terms of routability.

8. Final remarks

In this paper, we have reviewed some of the major issues and algorithms pertaining to the design of systems using FPGAs and other PLDs. The choice of an appropriate device for implementing a certain function can be based on two main factors:

- *Size*: The number of inputs, product terms, outputs provided for. The size of the OR terms is also relevant for PAL design. For very large problems, an FPGA may be more suitable for a compact implementation. FPGAs and complex PLDs address the case of multi-level random logic and in devising larger systems. FPGA architectures with large-grained logic block structures favor more register-intensive designs; while small logic blocks with segmented routing is better to efficiently implement large fan-in combinational circuits.
- *Application*: PROMs are useful for table-look up operations such as in code conversion or when the logic function needs too many product terms (arithmetic functions) or as a function generator. In the latter context, a PROM can be looked upon as a *programmable logic element* (PLE).

  PLAs and PALs are more suited for sparser functions. In case of *random* or unstructured logic such as in sequencers, PLAs are more desirable since there is significant product term sharing possibility. On the other hand, for logic comprising of an *array of similar elements* PAL-devices are preferred since there is no product sharing between array elements. Such structures often arise in datapath elements such as registers and counters. Complex PLDs and FPGAs can provide opportunities to implement entire systems and also provide more on-chip memory and logic.

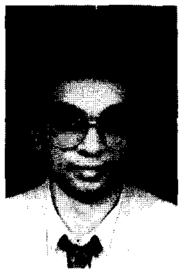
  FPGAs serve three niche markets:
  - *Rapid prototyping*: The FPGA enables the transfer of a new design onto silicon in a matter of hours as opposed to weeks and even months for other types of ASICs. This may be important as an architectural evaluation tool wherein the same hardware can be used to implement and evaluate several options for a particular design. In this reprogrammable mode, the FPGA can also serve as a valuable instructional tool for classroom projects.
  - *Hardware emulation*: Here, the FPGA is used as some sort of silicon breadboarding of the design [35]. Especially, in applications where logic simulation is not possible owing to real-time events, the FPGA can be used as an in-circuit working prototype. Since, speed of the design is critical, programming using metal links or anti-fuses may be desirable.
  - *Functional accelerator*: The FPGA can provide hardware assistance to parallel algorithm development by supporting reconfigurability in which the same hardware can be used to behave differently at different times so as to best match the communication requirements among the parallel processors [36].

Whereas two-level logic synthesis is more relevant for AND/OR type of PLDs, FPGAs are better served by multi-level logic synthesis tools. Good technology mapping and place and route algorithms are also very important for efficient implementations. Though routing for FPGAs are currently extensions of existing gate-array algorithms, there is a need for more specialized routers that take into account the additional constraints caused by the fixed wiring resources.

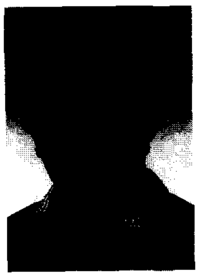
References

[1] G. Bostock, *Programmable Logic Devices: Technology and Applications* (McGraw-Hill, New York, 1988).
[2] M. Bolton, *Digital Systems Design with Programmable Logic*, Electronic Systems Engineering Series (Addison-Wesley, Reading, Mass., 1990).
[3] D. Pellerin and M. Holley, *Practical Design Using Programmable Logic* (Prentice-Hall, Englewood Cliffs, N.J., 1991).
[4] P.K. Lala, *Digital System Design Using Programmable Logic Devices* (Prentice-Hall, Englewood Cliffs, N.J., 1990).
[5] J. Rose, R. Francis et al., Architecture of FPGAs: The effect of logic block functionality on area efficiency, *Journal of Solid State Circuits*, Vol. 25, pp. 1217-1225, 1990.
[6] S. Singh, J. Rose et al., The effect of logic block architecture on FPGA performance, *Journal of Solid State Circuits*, Vol. 27, pp. 281-287, 1992.
[7] *Altera Data Book*, 1990 ed., October.
[8] H.C. Hsieh et al., Third generation architecture boosts speed and density of FPGAs, in: *Custom Integrated Circuits Conf.*, pp. 31.2.1-31.2.7, 1990.
[9] K. El-Ayat, A. El-Gamal et al., A CMOS electrically configurable gate array, *Journal of Solid State Circuits*, Vol. 24, pp. 752-761, 1989.
[10] A. El-Gamal, J. Greene et al., An architecture for electrically configurable gate arrays, *Journal of Solid State Circuits*, Vol. 24, pp. 394-398, 1989.
[11] T. Clark, Fitting programmable logic, *IEEE Spectrum*, pp. 36-39, 1992.
[12] XILINX, 2100 Logic Drive, San Jose, California, 95214, *The Programmable Gate Array Book*, 1991 ed.
[13] R.K. Brayton, G.D. Hachtel, C.T. McMullen, and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis* (Kluwer Academic, Dordrecht, 1984).
[14] T. Villa and A. Sangiovanni-Vincentelli, NOVA: state assignment of finite state machines for optimal two-level logic implementations, *IEEE Trans. on CAD*, Vol. 9, pp. 1326-1334, 1990.
[15] A. Sangiovanni-Vincentelli, A. El Gamal, and J. Rose, Synthesis methods for field programmable gate arrays, *Proceedings of the IEEE*, pp. 1057-1083, 1993.
[16] E. Detjens et al., Technology mapping in MIS, *Proc. ICCAD*, pp. 116-119, November 1987.
[17] S. Devadas, B. Ma, R. Newton, and A. Sangiovanni-Vincentelli, MUSTANG: state assignment of finite state machines targetting multi-level logic implementations, *IEEE Trans. on CAD*, Vol. 7, December 1990.
[18] W. Lin and A.R. Newton, Synthesis of multiple level logic from symbolic high-level description languages, *Proc. VLSI 89 Conference*, Munich, Germany, August 1989.
[19] B. Preas and M. Lorenzetti, *Physical Design Automation of VLSI Systems* (Benjamin/Cummings, Menlo Park, Calif., 1988).
[20] K. Keutzer, DAGON: technology binding and local optimization by DAG matching, *Proc. 24th DAC*, pp. 341-347, June 1987.
[21] G. de Micheli and F. Mailhot, Algorithms for technology mapping based on binary decision diagrams and on boolean operations, *IEEE Trans. on CAD*, pp. 599-620, May 1993.
[22] R. Murgai et al., Logic synthesis for programmable gate arrays, *Proc. 27th DAC*, pp. 620-625, June 1990.
[23] R.J. Francis, J. Rose, and Z. Vranesic, Chortle-crf: Fast technology mapping for lookup-table based FPGAs, *Proc. 28th DAC*, pp. 227-233, June 1991.
[24] N. Woo, A heuristic method for FPGA technology mapping based on edge visibility, *Proc. 28th DAC*, 1991.
[25] C. Sechen, Chip-planning, placement, and global routing of macro/custom cell integrated circuits using simulated annealing, in: *Design Automation Conference*, pp. 73-80, 1988.
[26] J. Greene, V. Roychowdury, S. Kaptanoglu, and A. El Gamal, Segmented channel routing, *Proc. 27th DAC*, pp. 567-572, June 1990.
[27] H.C. Hsieh et al., A 9000-gate user programmable gate array, in: *Custom Integrated Circuits Conf.*, pp. 15.3.1-15.3.7, 1988.
[28] J. Rose and S. Brown, Flexibility of interconnection structures for FPGAs, *Journal of Solid State Circuits*, Vol. 26, pp. 277-282, 1991.
[29] S. Brown, J. Rose, and Z. Vranesic, A detailed router for field programmable gate arrays, *IEEE Trans. on CAD*, Vol. 11, pp. 620-628, 1992.

[30] D.W. Hill and D. Cassiday, Preliminary description of tabula rasa: an electronically reconfigurable hardware engine, in: *Int. Conference on Computer and Design*, 1990.
[31] C. Pfister, The LABRYS project, in: *Int. Workshop on Field Programmable Logic and Applications*, 1991.
[32] J.F. Beetem, Simultaneous placement and routing of the labyrinth reconfigurable logic array, in: *Int. Workshop on Field Programmable Logic and Applications*, 1991.
[33] R. Linsker, An iterative improvement penalty-function driven wire routing system, *IBM Journal of Research and Development*, Vol. 28, No. 5, pp. 613-624, 1984.
[34] S.D. Brown *et al.*, *Field-Programmable Gate arrays* (Kluwer Academic, Dordrecht, 1992).
[35] Quicktum Systems, 325 E. Middlefield Road, Mountain View, CA 94043, *RPM Emulation System*, 1990.
[36] D. Lopresti, M. Gokhale, W. Holmes, A. Kopser, S. Lucas, R. Minnich, and D. Sweely, Building and using a highly parallel programmable logic array, *Computer*, pp. 81-89, January 1991.
[37] D. Hawley, Advanced PLD architectures, in: *Int. Workshop on Field Programmable Logic and Applications*, 1991.



R. Venkateswaran (S'89) received the B. Tech degree in computer science from the Indian Institute of Technology, Bombay, in 1988 and the M. S. degree in computer science and engineering in 1992 from the University of Michigan, Ann Arbor where he is now pursuing a Ph. D. degree. From 1991 to 1994, he has received the IBM Graduate Fellowship in Computer Science. His research interests include optimization problems, VLSI layout algorithms, special-purpose architectures and parallel algorithms particularly for CAD problems, as well as custom CMOS and FPGA designs.



Pinaki Mazumder received a B.S.E.E. degree from the Indian Institute of Science in 1976, an M.Sc. degree in Computer Science from the University of Alberta, Canada in 1985, and a Ph.D. degree in Electrical and Computer Engineering from the University of Illinois at Urbana-Champaign in 1987. Presently, he is working as an Associate Professor at the Department of Electrical Engineering and Computer Science of the University of Michigan at Ann Arbor. Prior to this, he worked two years as a research assistant at the Coordinated Science Laboratory, University of Illinois, and for six years as a Senior Design Engineer at BEL, India's primere and largest electronics industry, where he developed several types of analog and digital integrated circuits for consumer electronics products. During the summers of 1985 and 1986, he worked as a Member of Technical Staff in the Naperville branch of AT&T Bell Laboratories. He is a recipient of Digital's Incentives for Excellence Award, National Science Foundation Research Initiation Award, Bell Northern Research Laboratory Faculty Award and BFGoodrich Collegiate Inventors Award. His research interest includes VLSI memory testing, physical design automation, and ultrafast digital circuit design. He has written over 85 archival journal and rigorously reviewed conference proceedings papers and numerous industrial technical reports and memoranda while he worked at BEL and Bell Labs. He has authored a book, titled *Semiconductor Memories: Testing and Reliability* (Kluwer Academic Publishers, Dordrecht, 1994), and currently he is completing a book on the applications of genetic algorithms for VLSI layout systems. In addition to these books, he coauthored two monographs—one on VLSI routing algorithms and their parallel implementations, and the other on ultra-fast digital circuits. He was a Guest Editor of the special issues on multimegabit memory design and testing of *IEEE Design and Test of Computers* (June 1993) and *Journal of Electronic Testing-Theory and Applications* (August 1994).
Dr. Mazumder is a member of IEEE, Sigma Xi, Phi Kappa Phi and ACM SIGDA.