*Journal of Engineering Sciences, Assiut University, Vol. 36, No. 6, pp.1499- 1511, November 2008*

# ADAPTIVE ALGORITHM FOR ROUTING AND PLACEMENT IN FPGA

***M. E. ELBABLY***

*Dept. of Telecommunications and Electronics, Faculty of Engineering, Helwan University, Helwan, Cairo, Egypt*

*(Received May 31, 2008 Accepted June 25, 2008)*

*The design of the placement and routing for an FPGA (whether it's a traditional or coarse grained field programmable gate arrays) is very important process, requiring the care about the flexibility with silicon efficiency. With the motivation growing towards embedding FPGAs into SoC (system on chip) designs, final requirements for the FPGA architectures becomes more critical. The identification of a routing channel requires determining the number of routing paths (tracks), the length of the segments in those paths, and the positioning of the breaks on the paths. We have developed an optimal algorithm to alleviate the routing and placement problem. This research focuses on the maximization of the flexibility and expandability to achieve the final placement with the convenient path(s) (routing). The optimal algorithm finds a solution provided the problem meets a number of restrictions such as busy or faulty path(s) in the routing process and applying the partial configuration to reduce the configuration time to achieve the required placement.*

**KEYWORDS:** *Embedding FPGAs into SoC designs, routing and placement approaches, electronic design automation (EDA) and algorithms design.*

## 1. INTRODUCTION

Automatic placement becomes a very interested area for research in the last years. Most of the previous researches focus on optimization of the wire length, routability, or timing. As placement requires to be performed repeatedly in early design stages, existing placement algorithms are very expensive to use. In addition, circuit size and complexity has been increasing rapidly in the past several decades. Recently [1, 2], shown that existing placement algorithms are not scalable. The objective of this research is to design routing and placement algorithm that are extremely fast, flexible, expandable and generate placements using easy procedures.

Previous researches focus on optimization of the wire length and routability, or timing. As placement requires to be performed repeatedly in early design stages, existing placement algorithms are very expensive to use. In addition, circuit size and complexity has been increasing rapidly in the past several decades. Recently [1, 2], shown that existing placement algorithms are not scalable.

There are two primary methods in conventional computing for the execution of any algorithm for any function. The first is to use hardwired technology, either an Application Specific Integrated Circuit (ASIC) or a group of individual components forming a board-level solution, to perform the operations in hardware. Both are an expensive process, and also somewhat inflexible.

The second method is to use software programmed microprocessors which is more flexible solution without changing the hardware. However, for this method the response can suffer not only in clock speed but also in work rate, and is far below that of an ASIC. Each instruction from memory will be read by processor, decode its meaning, and then execute it. This causes a high execution overhead for each individual instruction.

A reconfigurable device is intended to fill the gap between hardware and software, achieving potentially much higher performance than software, while maintaining a higher level of flexibility than hardware. Reconfigurable devices, such as field-programmable gate arrays (FPGAs), contain an array of computational elements whose functionality is determined through multiple programmable configuration bits. These elements known as logic blocks, are connected using a set of routing that are also programmable to form the necessary circuit or to relocate any selected function(s). To achieve any performance benefits, with support of wide range of applications, reconfigurable systems are usually formed with a combination of reconfigurable logic and a general-purpose microprocessor. There are some operations that cannot be done efficiently in the reconfigurable logic, such as data-dependent control and possibly memory accesses, such operations will be performed by processor, while the computational cores are mapped to the reconfigurable hardware.

In this research earlier researches related to this area will be presented in section two. In section three the proposed algorithm procedures will be introduced. Conclusions will be given in section four.

## 2. EARLIER RESEARCH

For any programmed function which are required to move to another location (new placement). There are some methods such as configuration compression and the partial re-use of already programmed configurations can be used to alleviate the configuration time problem. One form of configuration compression [3-6] has already been implemented in a commercial system. This can be achieved by provides a method to program multiple logic cells with a single address and data value. This is accomplished by setting a special register to indicate which of the address bits should behave as "don’t-care" values, resolving to multiple addresses for configuration. For example, suppose two configuration addresses, 01000 and 01100, are both are to be programmed with the same value. By setting the wildcard register to 00100, the address value sent is interpreted as 01X00 and both these locations are programmed using either of the two addresses above in a single operation. The benefits of this hardware discussed in [3], while [5] covers a potential extension to the concept, where “don’t care” values in the configuration stream can be used to allow areas with similar but not identical configuration data values to also be programmed simultaneously.

A configuration can possibly re-use configuration information already present on the array, such that only the areas differing in configuration values must be reprogrammed. Therefore, configuration time can be reduced through the identification of these common components and the calculation of the incremental configurations that must be loaded [7,8].

Alternately, similar operations can be grouped together to form a single configuration which contains extra control circuitry in order to implement the various functions within the group [9].

### Relocation and De-fragmentation in Partially Reconfigurable Systems

Partially reconfigurable systems are better than single-context systems in that they allow a new configuration to be written to the programmable logic while the configurations not occupying that same area remain intact and available for future use. Because these configurations will not have to be reconfigured onto the array, and because the programming of a single configuration can require the transfer of far less configuration data than the programming of an entire context, a partially reconfigurable system can incur less configuration overhead than a single-context FPGA [10]. However, inefficiencies can arise if two partial configurations have been placed to overlapping physical locations on the FPGA. If these configurations are repeatedly used one after another, they must be swapped in and out of the array each time. This type of conflict could negate much of the benefit achieved by partially reconfigurable systems. A better solution to this problem is to allow the final placement of the configurations to occur at run-time [10, 11]. Storing the configurations in fast memory near to the reconfigurable array reduces the overall time required for the data transfer during reconfiguration [10].

Reconfigurable systems therefore have the potential to achieve greater performance than software as a result of bypassing the fetch-decode-execute cycle of traditional microprocessors.

## Configurable ASIC (CASIC)

More specialized design called a configurable ASIC (cASIC) can be created if we know the actual circuits to be computed. The cASICs are intended as accelerators on domain-specific Systems-on-a-Chip (SoCs), where ASIC-style accelerators would otherwise be used [12]. The cASIC hardware would accelerate the most compute-intense and most common applications for which the SoC is intended, acting as support hardware or coprocessor circuitry to a host microprocessor. The cASIC design flow would be part of the design process for the SoC itself. Although much work has been published in this general area of hardware/software, co-design and hardware compilation, the most relevant addresses extraction of inner loops so as to create cASIC-style designs [13]. The research in [12] focuses on the techniques to design the cASIC hardware after the circuit candidates are known.

The design flow for cASICs ideally would be entirely automatic. CASIC architecture generation occurs in two phases. The logic phase determines the computation needs of the application net lists, creates the computational components (ALUs, RAMs, multipliers, registers, etc.), and orders the physical elements along the one-dimensional data path. Also, the net list instances must be bound to the physical components. The routing phase creates wires and multiplexers to connect the logic and I/O components.

The LUTs are the physical components in traditional FPGAs, while the net list instances are low-level gates or small logic functions. When using traditional FPGAs, this matching is referred to as placement or binding. The terms binding or mapping are used to describe the process of matching an instance to a component. A physical move describes the act of assigning a physical location to a physical component. Placement during cASIC generation utilizes a simulated annealing algorithm [14], commonly used in FPGA placement (binding) to assign net list instances to physical computation units, and standard cell placement to determine locations for actual physical cells. This algorithm [14] operates by taking a random initial placement of elements and repeatedly attempting to move the location of a randomly selected element. The move is accepted if it improves the overall task of the placement.

Related to the routing, the only routing resources are those which are explicitly required by one or more of the net lists. CASIC routing generation techniques applied on the net list and the procedures of the algorithms, greedy, bipartite, and clique are discussed in [12].

The most famous placement and routing tool have been established for the traditional FPGA (its area implemented in two dimensions only x and y) is VPR (Versatile Place and Route) [15].

VPR can be run in one of two basic modes. In its default mode, VPR places a circuit on an FPGA and then repeatedly attempts to route it in order to find the minimum number of tracks (paths) required by the specified FPGA architecture to route this circuit. In case of a routing is unsuccessful, VPR increases the number of tracks in each routing channel and tries again; if a routing is successful, VPR decreases the number of tracks before trying to route it again. Once the minimum number of tracks required to route the circuit is found, VPR exits. The other mode of VPR is invoked when a user specifies a specific channel width for routing. In this case, VPR places a circuit and attempts to route it only once, with the specified channel width. If the circuit will not route at the specified channel width, VPR simply report that it is un-routable. VPR can perform either global routing or combined global and detailed routing. The object of routing generation is to minimize area by sharing wires between net lists while adding as few multiplexers / de-multiplexers as necessary.

The routing requirements will be discussed in the proposed algorithm presented in this research. The routing contained two problems: creating the wires and assigning of the signals to wires. Wire lengths can be adjusted for each module in many current advanced FPGA (as coarse grained which facilitate a larger number of possible connections in x, y, and z dimensions) architectures, by taking advantage of programmable connections (segmentation points) between lengths of wire (potentially forming a single long wire out of several short wires).

The proposed algorithm, identifying the empty path (routing) simply not share at all and also avoiding the faulty (or unavailable) path(s), this identifications are achieved by checking the sequences between any selected addressed source to the selected addressed sink. Also, the proposed algorithm can achieve the placement for any selected module (as a source) implemented in packs of the lookup tables (LUTs) and flip flops FFs together to form more coarse-grained logic blocks, not on its net list as previous algorithms. This is to decrease the complexity of the algorithm procedures.

## 3. THE PROPOSED ALGORITHM

Recently [16], many placement problems consist of a huge number of standard cells together with tens to hundreds significantly larger macros (corresponding to IP blocks, memory, etc) motivate the fast place area for research [17-20].

Currently, fast place uses wire length minimization as its only objective. To alleviate routing congestion, a simple idea is to shift cells away from congested regions. To extend fast place for routability-driven placement, we can perform cell shifting based on routing congestion also, instead of cell overlaps only. In order to maintain the efficiency of fast place, an extremely easy-to-compute pre-routing congestion estimation model is crucial. The idea of the above technique is the basic to achieve the placement in the proposed algorithm in this research. This can be executed as a first step to shift the configuration for any selected addressed module (source) to a new location, we have to be sure this new location is already empty. According to this the FPGA area is modeled as a coarse grained dimensional array (i.e. with x number of columns, y number of rows and z number of perpendicular (all numbers will be positive or negative except x should be positive)), this array is called area matrix [21]. The area matrix data can be updated very efficiently after addition or deletion of a task. Negative weight is assigned to all the cells (CLBs) occupied by the added task. All the positive cells directly below the task are incremented by height of the newly added task. Similarly, after deletion of a task, positive weight is assigned to all the cells previously occupied by the deleted task. New weights reflect the number of continuous empty cells above the cell i.e., the empty cells are already identified. Then to achieve the shifting technique for any selected module LUTs (which contains one or many units (or CLBs)), different routs should be available.

The routing between the logic blocks within the reconfigurable hardware is also of great importance. Routing contributes significantly to the overall area of the reconfigurable hardware. Yet, when the percentage of logic blocks used in an FPGA becomes very high, automatic routing tools frequently have difficulty achieving the necessary connections between the blocks. Good routing structures are therefore essential to ensure that a design can be successfully placed and routed onto the reconfigurable hardware.

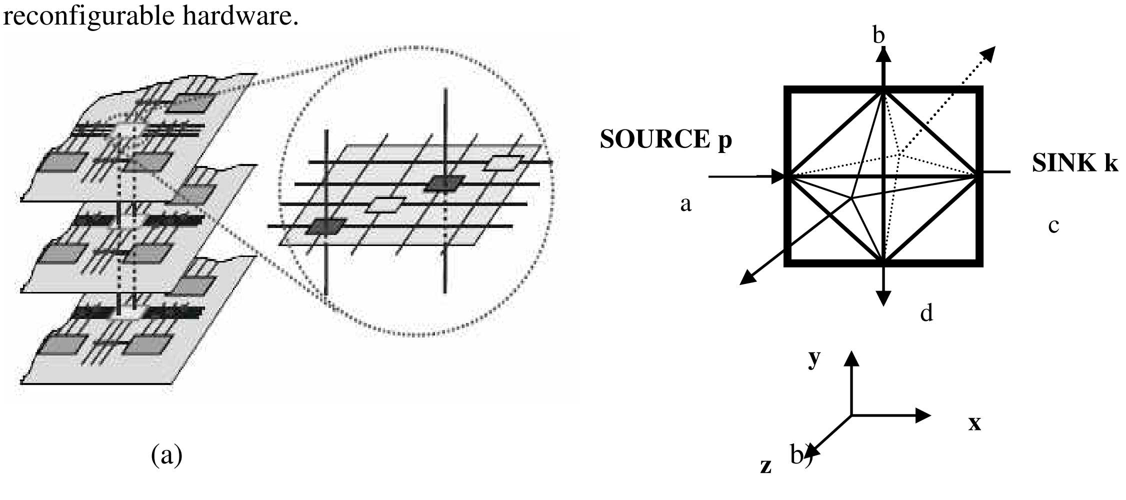

Figure 1 Coarse grained (3 D) FPGA (a), 3 D FPGA switch (b).

The different five paths (routs) between any selected addressed source to any addressed desired sink(s) will be shown in figure 2. Any selected path identified at the programming switches between the CLBs units in FPGA, which contains from one segment up to three segments as shown in Figure 2. Any selected source can be placed in different sink locations (this as required in replication area, it can be reached up to "m" sinks individually).

Once a circuit has been programmed onto the reconfigurable hardware, it is ready to be used by the host processor during program execution. The runtime operation of a reconfigurable system occurs in two distinct phases: configuration and execution.

The programming of the reconfigurable hardware is under the control of the host processor. This host processor directs a stream of configuration data to the reconfigurable hardware, and this configuration data is used to define the actual operation of the hardware. Configurations can be loaded solely at startup of a program, or periodically during runtime, depending on the design of the system.

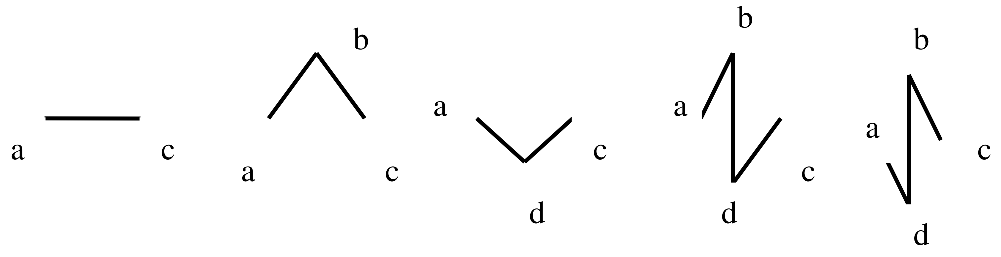

Figure 2 The different paths between the any selected source location (example at a address) and selected destination (sink) (example at **b** address).

Because run-time reconfigurable systems involve reconfiguration during program execution, the reconfiguration must be done as efficiently and as quickly as possible. Therefore, fast configuration is an important area of research for run-time reconfigurable systems. There are a number of different tactics for reducing the configuration overhead [17-20]. An algorithm to simultaneously optimize both short- and long-path timing in a field-programmable gate array (FPGA) presented in [22].

In the proposed algorithm the detailed routing for each module (whether it is a source or a sink modules), will be one by one. Each routing begins with a keyword that identifies a type of routing segment (i.e., identifying the available path). The possible keywords are source (the source of a certain output pin(s) class, its address identified in a counter p) and sink (the sink of a certain input pin(s) class, its address identified in a counter k). Each routing begins on a source and ends on a sink. Finally, the class number (if the source or sink was on a CLB) or path number (in the programming switch at any direction) is listed whichever one is appropriate. The meaning of these numbers should be clearly obvious. If we are attaching to a pad, the pad number given for a resource is the sub-block number defining to which pad at location (x, y, z) we are attached.

The first wiring path will always go from a source to the selected sink. The routing segment listed immediately after the sink is the part of the existing routing to which the new path attaches (i.e., the other sink in this rout will not be activated). It is important to realize that the first pin after a sink is the connection into the already specified routing tree; when computing routing statistics not count the same segment several times at the same moment to avoid the overlapping.

The physical location of configurations at run time based on where the free (empty) area on the hardware is located at any given time without any problem in configuration time**.** The main requirements to achieve any placement in FPGA is to have a technique with a very good trade-off between wire length and maximum cut [21]. This can be achieved using a graph example as shown in Figure 3. The procedure of this technique uses row (column) swaps to sort rows (columns) such that nonzero elements move toward the main diagonal.

For example, for the matrix in Figure 3-a, to shift nonzero elements from the upper half toward the main diagonal (from right to left), you perform column swaps between columns 2 and 3, and then move column 6 between columns 2 and 4.

Repeating this technique on rows and columns moves nonzero elements closer to the diagonal. When we ran this procedure on the graph in Figure 3-b, it created the linear arrangement in Figure 3-c.In fact we discovered there are many other suggested linear arrangements can achieve the same wire or path length (=7) and maximum cut (=2), i.e., the above linear arrangement is not a unique solution.

| | | Vertices | | | | | |
|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 |
| Edges | a | 1 | 0 | 1 | 0 | 0 | 0 |
| | b | 0 | 1 | 0 | 0 | 0 | 1 |
| | c | 0 | 0 | 1 | 1 | 0 | 0 |
| | d | 0 | 0 | 1 | 0 | 0 | 1 |
| | e | 0 | 0 | 0 | 0 | 1 | 1 |

Figure 3- a Initial wire length =10, maximum cut = 3

| | 1 | 4 | 3 | 2 | 6 | 5 |
|---|---|---|---|---|---|---|
| a | 1 | 0 | 1 | 0 | 0 | 0 |
| b | 0 | 1 | 1 | 0 | 0 | 0 |
| c | 0 | 0 | 1 | 0 | 1 | 0 |
| d | 0 | 0 | 0 | 1 | 1 | 0 |
| e | 0 | 0 | 0 | 0 | 1 | 1 |

Figure 3-b Final matrix which achieve minimum wire length and maximum cut.

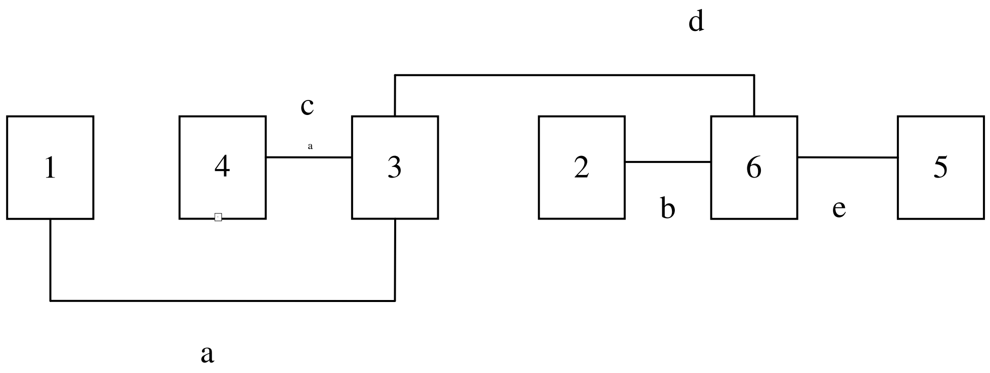

Figure 3-c Initial wire length =7, maximum cut = 2

One of the other suggested solution we can be presented will be shown in the Figure 4, the suggested solution can be depend only on columns moves nonzero elements.

$$\begin{array}{c|cccccc|}
 & 1 & 4 & 3 & 6 & 2 & 5 \\
\hline
a & 1 & 0 & 1 & 0 & 0 & 0 \\
b & 0 & 1 & 1 & 0 & 0 & 0 \\
c & 0 & 0 & 1 & 1 & 0 & 0 \\
d & 0 & 0 & 0 & 1 & 1 & 0 \\
e & 0 & 0 & 0 & 1 & 0 & 1 \\
\hline
\end{array}$$

Figure 4 Another suggested solution

In the proposed algorithm, the final placement for any selected module as showed in Figure 3 can be moved or replicated to the proper location(s) which addressed according to area matrix data. This can be achieved using partial configuration and available routs.

Partial configuration for any selected source to the proper destination executed simply by the following formula: the original configuration of the destination module XOR with the result between the XORing of the original configurations of the source and destination modules.

The movement (routing) of the selected addressed module (source) will be executed step by step without module(s) overlaps according to the required space location(s) as addressed destination(s) (sink).

In the proposed algorithm which is based on the ASM (algorithmic state machine), the sources and sinks are defined as follows:

- "n" is the maximum number of addressed sources, and it will be identified in counter **p**
- "m" is the maximum number of addressed sinks, and it will be identified in counter **k.**

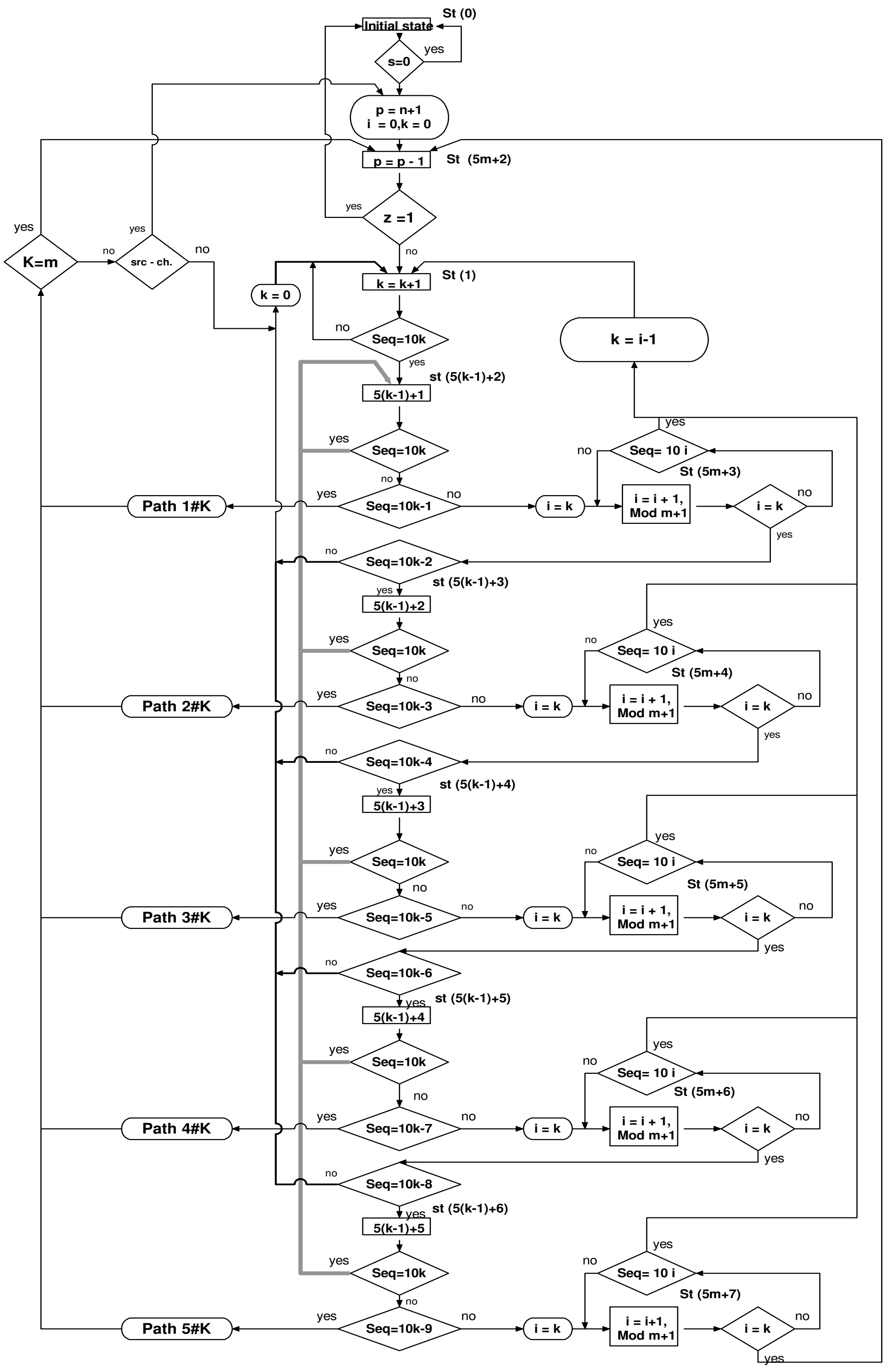
St (0)
Initial state
s=0
yes
p = n+1
i = 0,k = 0
p = p - 1
St (5m+2)
z =1
yes
no
K=m
yes
no
src - ch.
yes
no
k = 0
k = k+1
St (1)
Seq=10k
no
yes
k = i-1
st (5(k-1)+2)
5(k-1)+1
Seq=10k
Seq= 10 i
St (5m+3)
Path 1#K
Seq=10k-1
i = k
i = i + 1, Mod m+1
i = k
Seq=10k-2
st (5(k-1)+3)
5(k-1)+2
Seq= 10 i
St (5m+4)
Path 2#K
Seq=10k-3
Seq=10k-4
st (5(k-1)+4)
5(k-1)+3
St (5m+5)
Path 3#K
Seq=10k-5
Seq=10k-6
st (5(k-1)+5)
5(k-1)+4
St (5m+6)
Path 4#K
Seq=10k-7
Seq=10k-8
st (5(k-1)+6)
5(k-1)+5
St (5m+7)
Path 5#K
Seq=10k-9
i = i+1, Mod m+1

Figure 5

The designed of the proposed algorithm can be considered as an adaptive algorithm for the following reasons:

- it is expandable for any "n" and any "m", and the complexity of the algorithm design procedures can be alleviated by making the state boxes introduced as a variable states (according to the equation inside each variable state box).
- satisfying unconditional microcontroller sequences (this algorithm is high flexible algorithm), i.e., there is complete facility to move from the present state (in the way to achieve the route and placement for any sink) to achieve any other complete available sequences which are required to execute the route and placement to another desired sink. This movement can be achieved by any loop appears on the right hand side in Figure 5, starting by the conditional box i = k followed by the state box i = i + 1 (mode m+1). Also, the flexibility to change the selection to other addressed source at any moment will be achieved by the decision box “src-ch” in the top left of Figure 5.

The proposed algorithm sequences have the ability to identify the available path from the five available paths. This will be according to the optionally selected sequence(s) for each path. Either faulty or unavailable path(s) can be avoided if the selected sequence(s) will not be identified. The process of the proposed algorithm can be terminated when the available path ready to achieve the routing and placement (by partial placement) between any addressed source and desired sink. This will be according to the required sequences in the proposed ASM algorithm design procedures, as shown in the Figure 5.

While in VPR [15] if the routing is unsuccessful, VPR increases the number of paths (tracks) in each routing and tries again; if a routing is successful, VPR decreases the number of paths (tracks) before trying to route it again. For this reason, the proposed algorithm is more accurate than VPR tools for achieving the required routing and placement.

From the proposed algorithm the complexity of its hardware is based on different parameters, one of them is the state boxes (the primary outputs and the feedback flip-flops which depend on these state boxes). Table 1 shows the hardware complexity for different systems:

**Table 1**

| | Number of sources | Number of sinks | HW complexity |
|---|---|---|---|
| System 1 | 8 | 3 | 32 |
| System 2 | 10 | 5 | 58 |
| System 3 | 20 | 20 | 408 |
| System 4 | 40 | 30 | 1208 |
| System 5 | 100 | 70 | 7008 |

In general the hardware complexity can be determined from the following equation:
HW complexity = m * n + 8.

## 4. CONCLUSIONS

As we have shown, the routing and placement problem involved fairly subtle choices, including balancing requirements for paths. Multiple algorithms were presented to solve the routing and placement problem. One of these algorithms is probably optimal for some situations, though it is complex and works for only a relatively restricted set of cases. The proposed algorithm procedures have the ability to avoid both faulty or unavailable path(s)) from the five available paths at any programming switches between the CLBs. We presented an adaptive flexible and expandable algorithm, which appears to be optimal in all cases (number of sources, destinations and path identification) meeting the restrictions of other algorithms, and whose average appears closer optimal overall.

We envision two situations where the results of this paper can be applied. First, we believe that there is a growing need for automatic generation techniques of routing and placement in FPGA for any selected number of sources and destinations. Second, the ability of the identification for the available empty path to success the routing between the selected source and the required destination.

## REFERENCES

[1] P. Berman and G. Schnitger, “On the complexity of approximating the independent set problem,” *Inf. Comput.*, vol. 96, no. 1, pp. 77–94, Jan. 1992.

[2] H. N. Brady, “An approach to topological pin assignment,” *IEEE Trans.Comput.-Aided Des. Integr. Circuits Syst.*, vol. CAD-3, no. 3, pp. 250–255, Jul. 1984.

[3] S. Hauck, Z. Li, E. Schwabe, "Configuration Compression for the Xilinx XC6200 FPGA", *IEEE Symposium on Field-Programmable Custom Computing Machines*, 1998.

[4] S. Hauck, W. D. Wilson, “Runlength Compression Techniques for FPGA Configurations”, Northwestern University, Dept. of ECE Technical Report, 1999.

[5] Z. Li, S. Hauck, “Don’t Care Discovery for FPGA Configuration Compression*”, ACM/SIGDA International Symposium on FPGAs*, pp. 91-98, 1999.

[6] A. Dandalis, V. Prasanna, “Configuration Compression for FPGA-based Embedded Systems”, *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2001.

[7] W. Luk, N. Shirazi, P. Y. K. Cheung, “Compilation Tools for Run-Time Reconfigurable Designs”, *IEEE Symposium on Field-Programmable Custom Computing Machines*, 1997.

[8] N. Shirazi, W. Luk, P. Y. K. Cheung, "Automating Production of Run-Time Reconfigurable Designs", IEEE Symposium on Field-Programmable Custom Computing Machines, 1998.

[9] B. Kastrup, A. Bink, J. Hoogerbrugge, "ConCISe: A Compiler-Driven CPLD-Based Instruction Set Accelerator", IEEE Symposium on Field-Programmable Custom Computing Machines, 1999.

[10] K. Compton, J. Cooley, S. Knol, S. Hauck, “Configuration Relocation and Defragmentation for FPGAs”, *Northwestern University Technical Report*, http://www.ece.nwu.edu/~kati/publications.html, 2000.

[11] Z. Li, K. Compton, S. Hauck, “Configuration Caching for FPGAs”, *IEEE Symposium on Field- Programmable Custom Computing Machines*, 2000.

[12] Katherine Compton, Member, IEEE, and Scott Hauck, Senior Member, IEEE "Automatic Design of Area-Efficient Configurable ASIC Cores" IEEE

TRANSACTIONS ON COMPUTERS, VOL. 56, NO. 5, pp, 662-672, MAY 2007

[13] Z. Huang and S. Malik, “Exploiting Operation Level Parallelism through Dynamically Reconfigurable Datapaths,” Proc. Design Automation Conf., 2002.

[14] S. Kirkpatrick, D. Gelatt Jr., and M.P. Vecchi, “Optimization by Simulated Annealing,” Science, vol. 220, no. 4598, pp. 671-680, May 1983.

[15] "VPR and T-VPack: Versatile Packing, Placement and Routing for FPGAs" http://www.eecg.toronto.edu/~vaughn/vpr/vpr.html - Ver. 4.30 – March 27, 2000

[16] Muhammet Mustafa Ozdal and Martin D. F. Wong, *Fellow, IEEE* "Algorithms for Simultaneous Escape Routing and Layer Assignment of Dense PCBs" IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, VOL. 25, NO. 8, pp. 1510-1522, AUGUST 2006.

[17] T. Cohen, “Practical guidelines for the implementation of back drilling plated through hole vias in multi-gigabit board applications,” in *Proc. IEC DesignCon*, Santa Clara, CA, 2003.

[18] J. Cong and C. L. Liu, “On the k-layer planar subset and topological via minimization problems,” *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 10, no. 8, pp. 972–981, Aug. 1991.

[19] J. S. Corbin, C. N. Ramirez, and D. E. Massey, “Land grid array sockets for server applications,” *IBM J. Res. Develop.*, vol. 46, no. 6, pp. 763–778, Nov. 2002.

[20] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. Cambridge, MA: MIT Press, 1992.

[21] Cristinel Ababei, Yan Feng, Brent Goplen, Hushrav Mogal, Tianpei Zhang, Kia Bazargan, and Sachin Sapatnekar "Placement and Routing in 3D Integrated Circuits" IEEE Design & Test of Computers, , pp. 520-531 November–December 2005.

[22] Ryan Fung, Vaughn Betz, and William Chow “Slack Allocation and Routing to Improve FPGA Timing While Repairing Short-Path Violations” IEEE Tran. ON Computer-Aided Design of Integrated Circuits and Systems, VOL. 27, NO. 4, pp. 686-697 April 2008.

## خوارزم متكيف لتحقيق وضع وتحريك الدوال بمسارات مختلفة من أماكنها (كمصادر) إلي أماكن أخري (كمصبات) في نبائط حقول البوابات المتراصة القابلة للبرمجة FPGAs

يقدم هذا البحث تصميم خوارزم يتكيف علي حسب تغير الوضع الابتدائي لعدد متغير من الدوال (كمصادر) المطلوب تعديل أماكنها للوصول بها الي عدد متغير من المصبات المطلوب تحريك الدوال إليها (يختلف هذا العدد عند تكرار دالة المصدر عند المصبات المطلوبة كما هو مطلوب عند زيادة سرعة النظام وذلك كما في أبحاث replication area) ويتم ذلك من خلال المسارات المتاحة متجنبا

المسارات المشغولة أو التي بها أعطال (بحد أقصي خمسة مسارات وهو المسموح به عمليا فيما بين أي خطين من خطوط الاتصال من خلال برمجة المفاتيح القابلة للبرمجة بين الوحدات الأساسية علي أي من مستويات الفراغ سواء كان بين x, y أو بين مستوي y, z أو بين مستوي z,x ) كما هو في النبائط الحديثة والمسماة ب coarse grained FPGA . يمكن لأي مسار مختار أن يتكون من جزء أو جزئين أو ثلاثة أجزاء بحد أقصي فيما بين أي مصدر وأي مصب يمكن اختيارهما ويتم تحقيق ذلك عن طريق خطوط التمكين عند النقاط المفصلية بين أجزاء المسار.

والوضع الابتدائي لأماكن جميع الدوال ألمحتواه في النظام يكون معتمدا علي مصفوفة الوحدات والتوصيل فيما بينها وهذا الوضع الابتدائي يتم تحديده بقاعد تحريك صفوف و أعمدة المصفوفة لنصل إلي أقل عدد من التقاطعات في الوصلات بين وحدات النظام المبرمج وقد رأينا أنه يمكن الوصول إلي الوضع النهائي المطلوب بتحريك أعمدة المصفوفة فقط .

تعديل أماكن الدوال (المصادر) الهدف منه هو تحسين أداء النظام ككل والاستفادة بأكبر قدر ممكن من مساحة النبائط القابلة للبرمجة لبناء الأنظمة المعقدة بها ويمكن برمجة هذا الخوارزم بداخل نفس النبيطة ليكون نظام متكامل ومتضمن (embedded) في نبيطة واحدة ( system on chip) .

يتميز هذا الخوارزم أيضا بمرونته بإمكان تحقيق تعديل أو تحريك دالة أي مصدر إلي أي مكان مرغوب الانتقال إليه من خلال المسارات المتاحة كهدف تتوفر له التتابعات (sequences) المترافقة له (من خلال الاستشعارات sensors المرتبطة بمولد شفرات هذه التتابعات ) حتى وان كان يتعامل مع هدف آخر لم تكتمل بعد له مجموعة التتابعات الخاصة به وذلك دون أي قيود وذلك حتى لا يكون هناك أي فقد لتحقيق أي هدف مطلوب التعامل معه.

ولسهولة تحقيق التعديل لأي دالة مصدر إلي المصب الجديد يتم استخدام التشكيل الجزئي لدالة المصب عن طريق تحديد الاختلاف فيما بين دالتي المصدر والمصب ( بواسطة XOR بين الدالتين) والتي يفضل أن تكون مبرمجه بأسلوب LUTs وبهذا الاختلاف يمكن تعديل دالة المصب لتتطابق مع دالة المصدر وبذلك يتم تخفيف مشاكل التحريك علي المسارات المتوفرة أو التشكيل الكامل لدالة المصب لتتطابق مع دالة المصدر ويتم تجنب الصعوبات التي يمكن أن تتواجد عند التعامل مع برمجة الدوال بأسلوب net lists والتي يفضل التعامل بها عند برمجة الدوال علي مستوي البوابات المنطقية أو ما شابه