synFPGA

Application-Specific FPGA Synthesis

Kenneth Eguro
Professor Scott Hauck

Acknowledgments

We are indebted to Hasani Steele for helping with the development of the interconnect algorithms.

1. Abstract

FPGAs play a major role in the implementation of many applications because reconfigurable logic allows for rapid prototyping, short time-to-market, and convenient upgradability. Currently, it is common to use general-purpose commodity FPGA components due to the large time and monetary investment required to produce more appropriate application-specific reconfigurable logic. A means of quickly and easily producing application-specific designs might change this trend, as it would make high performance devices much more accessible.

Furthermore, much work has been done on investigating the usefulness of reconfigurable processing units built on the same chip as conventional general-purpose processors. This system-on-a-chip methodology is highly attractive because it allows near custom logic performance for infinite applications while requiring limited area. This type of architecture becomes even more attractive as off and on-line reconfigurable logic placement, routing and scheduling algorithms become more efficient and effective. New design methods and tools akin to those available for general-purpose processor designs are necessary for such architectures to become popular.

Design automation through the use of standard cells is an option, but the results are typically not impressive, as such a system cannot optimize for structural regularities found in reconfigurable arrays. An effective automation system must be able to aggressively utilize structural regularities while maintaining the flexibility to incorporate the large logic cores, heterogeneous logic units, and heterogeneous routing resources that application-specific FPGAs require to maximize usability and processing power.

2. Project Context

For many applications, industry demands a way to receive, with minimal effort, a layout ready for fabrication. Thus, for application-specific FPGAs to become a mainstream reality, it is necessary to provide quick, effective CAD tools. Otherwise, the cost in time-to-market and resources needed for custom design and layout is simply too large. Furthermore, any improvement over the typical area requirements of standard cells is welcome. Our solution is layout generation. synFPGA will take the specifications of a design and create a layout of the overlaying structure. With minimal user intervention it will produce a viable, efficient application-specific, reconfigurable logic array.

3. Implementation

An island style FPGA consists of four basic types of components: logic blocks, routing channels, switchboxes, and local connection blocks. A typical arrangement of the components is shown in Figure 1. The function of a logic block is to perform computational operations. The sizes of these components may differ greatly depending

upon intended usage and design. A small, simple logic block may be designed to perform only two input logical operations while a large, complex logic core may be designed to perform fast 32 bit multiplication. The function of a routing channel is to allow different logic blocks to communicate between themselves quickly. The function of a switchbox is to provide useful routing by allowing different routing channels to interconnect. Finally, the function of a local connection block is to provide both the connections between a logic block and a routing channel and between a logic block and its neighboring logic blocks.

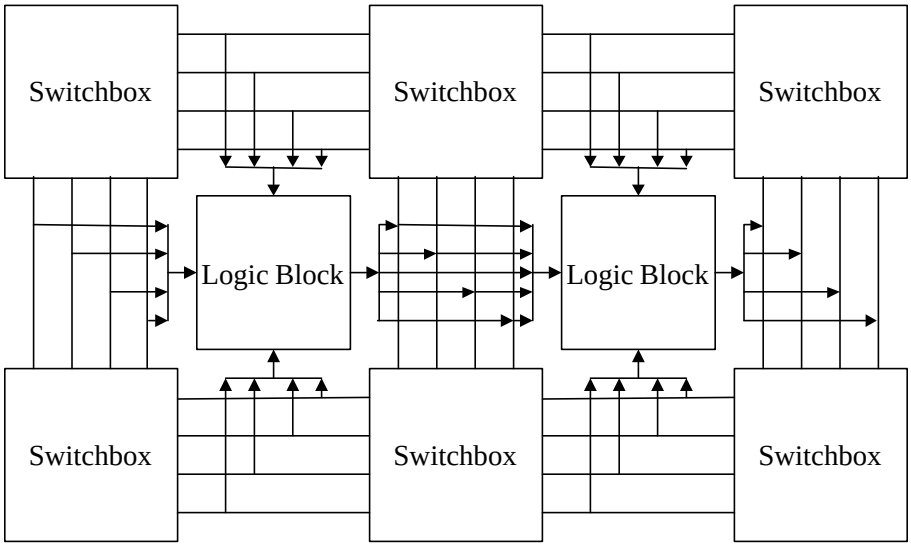

Figure 1 - Simple island style FPGA layout

Most FPGA designs contain a great deal of regularity. Routing channels are organized between logic blocks, switchboxes are organized between routing channels, etc. If conventional standard cell techniques were used, much of this order would disappear in the final layout. Logic blocks would intermingle with local connection blocks, which in turn would intermix with switching resources. This would lead to wasted space as routing channels and common signal such as power and ground would need to twist and turn. Instead, to take advantage of the typical grid structure of an FPGA, we assume straight routing channels and adjust spacing to ensure a regular grid pattern.

synFPGA will take all of the vital information that is necessary to build the support structure, but the matters of reasonable specifications, efficient floorplanning and practical design is left to the user as a part of the design process.

4. Layout Generation

Much of the area and, thus, much of the time required to design a custom FPGA is taken by routing resources. Therefore, it is wise to focus efforts in the automation of these components. Routing resources account for more than 90% of the total area of

modern FPGAs. Thus, less than 10% of the total area consists of logic resources. When one takes this into account, the area contribution of logic resources is very small. In addition, in itself, the creation of logic unit designs is relatively simple as the logic units typically calculate functions of only a few inputs. [1], [2] The user may design such small parts in full custom or, in the interest of time, in standard cells. Therefore, synFPGA does not build logic blocks but assumes that the user already has layouts and dimensions for logic blocks.

A defining characteristic of application-specific FPGA design is the intelligent use of sparse routing resources where signals are likely to travel linearly, to save space, and dense routing resources where signal are likely to be complex, to provide adequate routability. Similarly, some logic blocks should be small where sophisticated functions are not necessary and some should be large to allow common complex functions to be calculated quickly. Therefore, an effective CAD tool would necessarily support heterogeneous components such as those shown in Figure 2. synFPGA supports nearly complete heterogeneous components and will be discussed later.

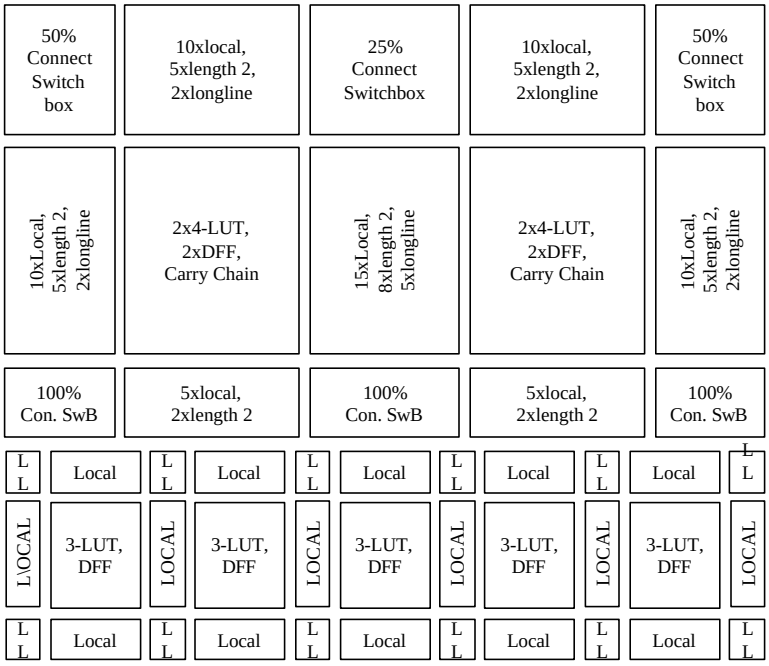

Figure 2 - Heterogeneous FPGA architecture

The automation process consists of four basic steps: specification input, component generation, grid sizing and component placement, and design output. The design flow is shown in Figure 3. With simple input files the user will enter specification information such as the number of routing channels desired or the specific types of switchboxes wanted. Once this information is obtained, the individual components are created to determine relative dimensions. The component generation consists of two sub-generation functions: switchbox generation and connection block / routing channel generation. Then, using the sizes calculated from the prior step, the grid sizing algorithm will set overall dimensions for the grid in order to get the most space efficient

layout possible. This step will also give relative global placements for all of the components. From this, output files for the Mentor Graphics IC tool package will be created. To complete the design cycle the user simply runs the output files through an IC tool design.

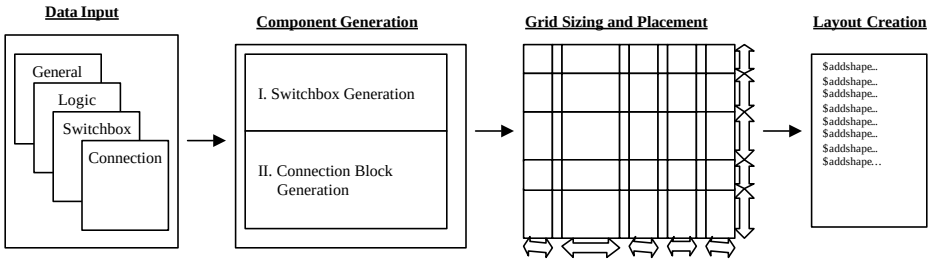

Figure 3 - synFPGA Design Flow

4.1 Specification Input

Specification input comes from four files: general information, logic block information, switchbox information, and connection block information. Assuming a grid containing X logic blocks horizontally and Y logic blocks vertically, a complete grid contains (X + 1) by (Y + 1) switchboxes, (X) by (Y + 1) horizontal connection blocks, and (X + 1) by (Y) vertical connection blocks. See Appendix 1 for example input files.

At this time, the only function of the general information file is to specify overall grid size. As will be explained later, this could expand. Each of the other information files is used to input specifications and global location of all respective block types used in the design. See Appendix 1 for a table of keywords. Please notice that the routing channel specification is included in the connection block input and thus, routing channels with different numbers of wires are required to have separate definitions regardless of connectivity, and vice-versa.

4.2.1 Switching Resource Generation

FPGA switching resources consist of two main components: a memory unit and a pass transistor. Any switching architecture can be built using some combination of these components. The key difference between layout generation and standard cell automated design is the grain size or complexity of the library components. In our system we chose to use a dual clock, shift chain-based memory unit and an n-mos pass transistor, but the design theory still holds true for other technologies. A traditional standard cell implementation would have two components: one memory and one n-mos transistor. A 2 x 2 switch, as shown in Figure 4, would contain 24 standard cell library components - 12 memory units and 12 n-mos transistors. synFPGA has one library component: an optimized, tileable six memory, six n-mos transistor unit as shown in Appendix 2. The same 2 x2 switch implemented through layout generation would only

contain 2 modules tiled together. While the total number of memory units and transistors is the same, much guesswork and thus much intra-component routing is eliminated. Thus our switch will have better performance with a smaller design. See Figure 5.

Figure 4 – 2x2 Switch Schematic

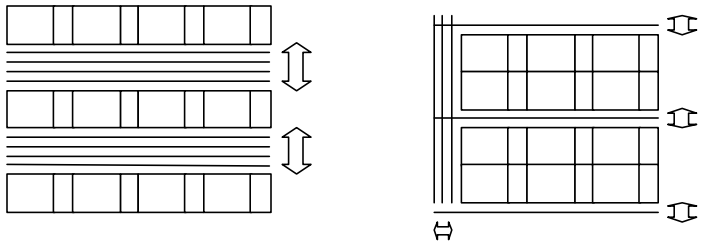

Figure 5 – Automated Switchbox Generation

The standard cell implementation requires many wires to successfully interconnect its basic components while the layout generator implementation needs only the basic global lines with minimal additions

synFPGA creates switching resources of any size specification by tiling together the 6 input library module. First, the largest number of inputs that the switchbox takes determines the overall number of modules. Then the appropriate number of switching modules are tiled and internal signal lines are routed to each side depending upon the number of inputs that the side takes. Finally, these internal signal lines are appropriately connected to the external input lines. A switchbox that takes six inputs from three sides and four from the fourth will require 6 library modules. See Figure 6. These modules are tiled into a roughly square configuration and internal signal lines are routed to each of the sides, ready to connect to external signal lines. Please notice that each side has an additional bank of wires to allow a single external signal to be easily routed to multiple internal wires.

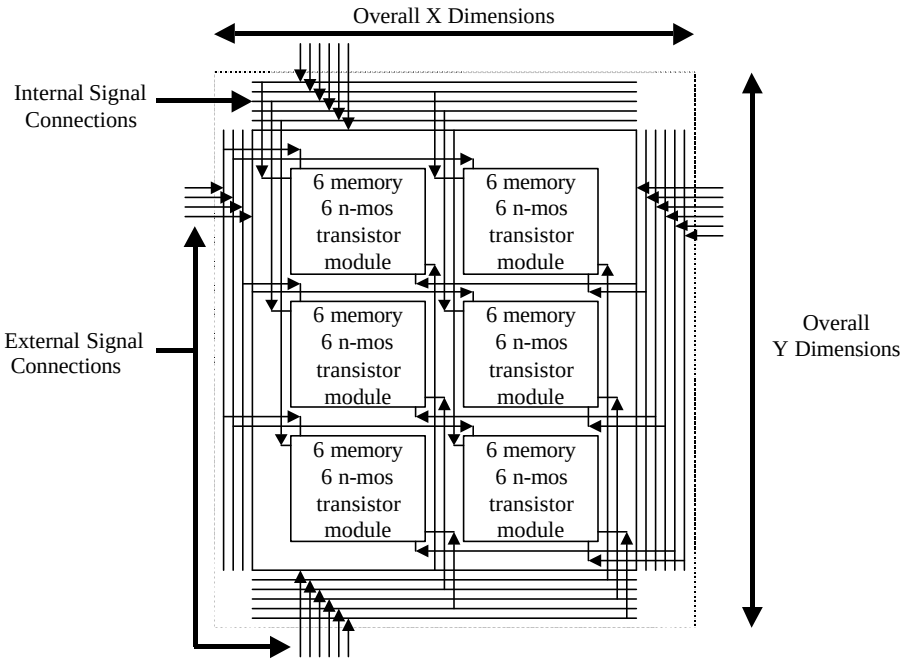

Figure 6 – Heterogeneous Switchbox Generation

An important feature to maximize the efficiency of application-specific architectures is adaptive connectivity. To fully realize the gains of heterogeneous routing resources, it is necessary to implement high or full connectivity switches in crucial areas and low connectivity switches in low traffic areas. In addition, to adapt to various architecture design requirements, it would be advantageous to allow for various switchbox types. Typical connectivity graphs are shown in Figure 7. At this time, only Xilinx 4000 heterogeneous switches are integrated into synFPGA. However, because the internal bank of wires created on each side of the switchbox, it is easily seen how with additional switching modules and/or slight modulation in the internal routing scheme it is possible to support multiple switchbox types with various connectivities.

An additional characteristic feature of application-specific FPGA design is long distance routing resources. Areas of complex routing often require signals to travel long distances. In a design that only contains single length routing resources, the signal must travel through multiple switchboxes and routing segments. This decreases performance because of increased signal latency. To solve this problem, modern commodity FPGA designs integrate routing resources that span multiple switchboxes. That is, in a design that incorporates double length lines, some lines in its routing channels do not connect to every switchbox, but instead skip over some and connect to every other switchbox. synFPGA does not currently support such long distance connection architectures. However, again it is easily seen how with the removal of some switching modules and simply short-circuiting across the intermediate switchboxes, it is possible to support long distance connection architectures. The single-length Xilinx switchboxes that are supported do demonstrate that this layout technique has merit.

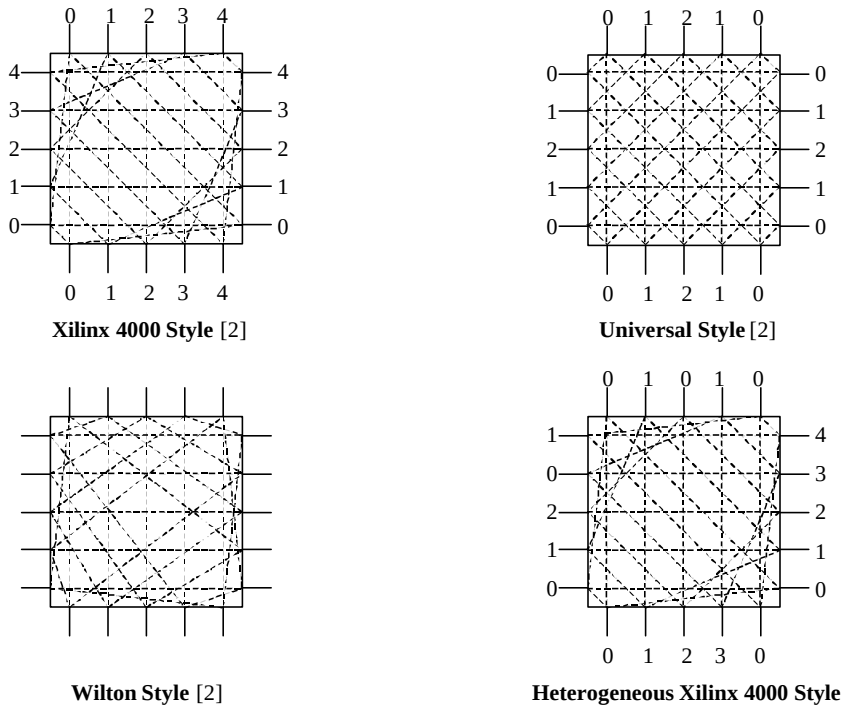

Figure 7 – Different Styles of Switch Blocks

3 different styles of switchboxes which all offer 20% connectivity. Notice the routing scheme and signal duplication in the heterogeneous Xilinx 4000 style switchbox. It is implemented with 3 inputs to the left, 2 inputs to the top, 4 inputs to the bottom and 5 inputs to the right.

4.2.2 Connection Resource Generation

FPGA connection resources consist of three types of components: multiplexors, demultiplexors and routing channel wires. To form the switching components, connection block units consist of 2 pieces: memory and tri-state buffers. Multiplexors consist of memory units combined with multiple tri-state buffers which have all of their outputs tied together. They serve to allow a single logic block input to receive a signal from multiple routing channel wires adaptively. Demultiplexors are simply a similar arrangement with all of the tri-state buffer inputs tied together. They serve to allow a single logic block output to drive a signal onto multiple routing channel wires adaptively. Akin to switchbox generation, synFPGA has an area-optimized, tileable memory and tri-state buffer module for connection block generation. This module is shown in Appendix 2. Again, similar to switchbox generation, this larger grain library component reduces the costly intra-component wiring present in standard cell implementations.

synFPGA creates multiplexors and demultiplexors of any specification by tiling together the appropriate number of tri-state buffer/memory units; one module for each input or output of a component. For example, a four to one multiplexor and a four to one demultiplexor would each consist of four library modules. Such a system continues until components larger than eight to one are created. After this point, in the interest of speed, a multi-level implementation is created. For example, a 32 to one

multiplexor actually consists of four eight to one multiplexors side by side that each feed into an additional four to one multiplexor. Thus, 36 modules are needed. However, this does not disturb the tiling nature of the system.

Architectures utilizing fewer memory units combined with decoders were explored, but as described in Appendix 2, this design was eventually chosen. After all of the appropriate switching components are formed, they are further tiled together to form 2 long chains - one for each side of the routing channel. Then, the appropriate number of routing wires are laid and the respective inputs and outputs of the switching components are connected to the channel wires. A "passthrough" module is simply a special case one to one multiplexor in which neither the input nor the output is connected to a routing channel wire but instead routed directly to either side of the connection resource. synFPGA leaves logic block input and output connection to the user. See Figure 8.

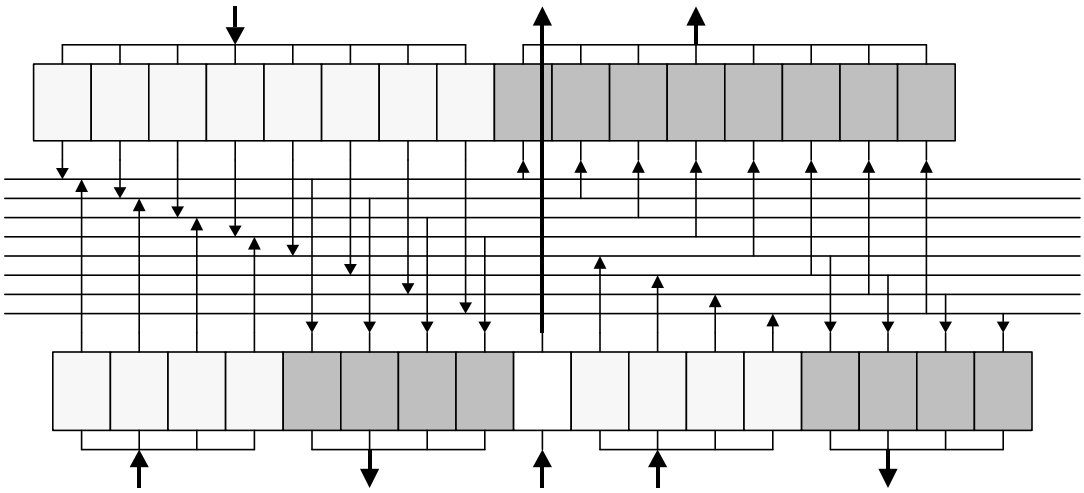

Figure 8 - Diagram of Connection Resource
The top row consists of a 1:8 demultiplexor and an 8:1 multiplexor. The bottom row consists of two 1:4 demultiplexors, two 4:1 multiplexors, and a "passthroughin" component.

4.2.3 Grid Sizing and Placement

The primary goal of this procedure is to fully exploit the structural regularity of island style FPGAs. Towards this goal, we implement a constraint on the entire array – all routing channels will be completely aligned in the final layout. After the component generation, we have all of the dimension information that we need. From this we begin to calculate the global dimensions of the array. Each row is forced to take the height of its tallest member and each column is forced to take the width of its widest member. Thus each component, regardless of actual dimensions, will have the respective global height and global width of the tallest member in its row and widest member in its column.

As Figure 9 shows, this can potentially lead to wasted area. Mostly this occurs in heterogeneous designs in which there are items of vastly different specification, such as narrow routing channel combined with extremely wide routing channels or small logic blocks combined extremely large logic blocks. This would cause problems in designs

with large logic cores as it would waste considerable space above, below and to each side of the core. However, partitioning areas of vastly different feature size into separate groups can solve much of this problem. See Figure 10. In most designs this type of segmentation is logical, such as the difference between control logic and datapath logic.

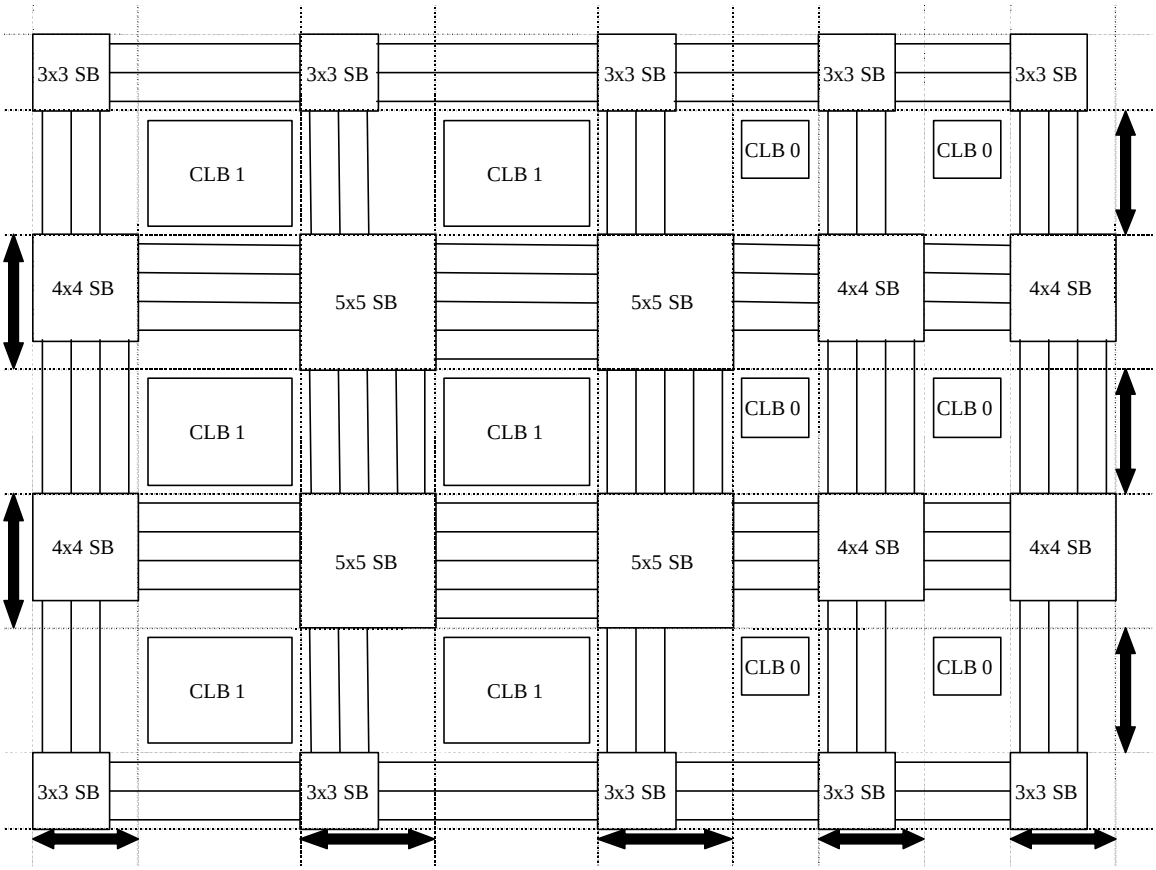

Figure 9 – Grid Sizing Algorithm

Notice that each row and column marked with an arrow has wasted area.

After the separate layouts are created they can be manually merged into a single design. Between regions of different feature size some amount of transition routing structures will be needed. If the user is not pleased with the results due to unanticipated switchbox or connection block sizing, floorplanning can be repeated using the feature sizes calculated previously as an approximate guide. A second run should produce better results. A logical place to incorporate this partitioning would be to indicate relative region locations in the general information file. This feature could be included in future work with a minimum of problems. This optimization technique does have limitations, though, because it will not work in a design that is uniformly heterogeneous, as partitioning would be difficult.

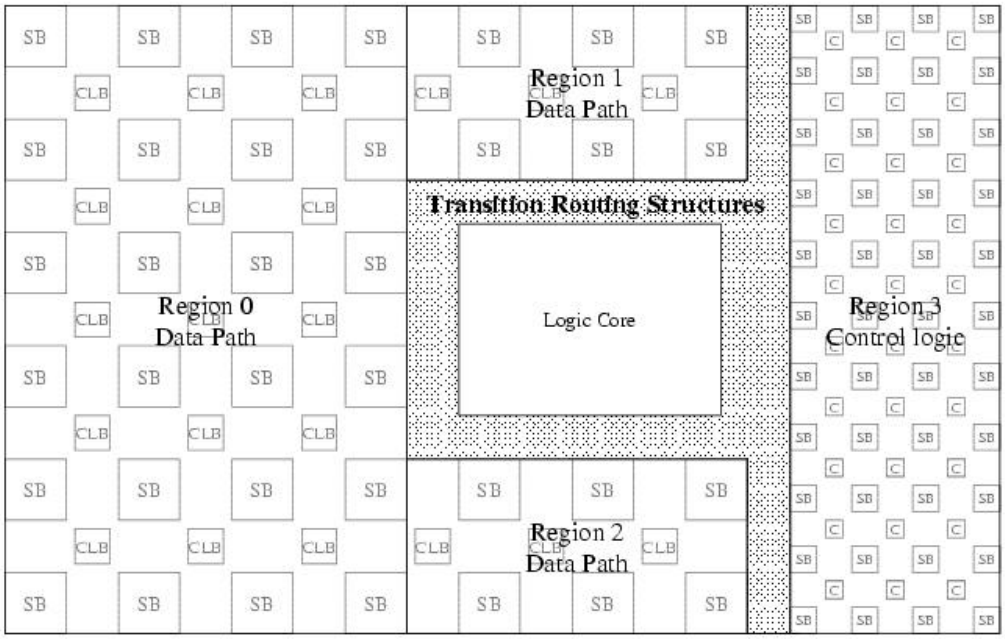

Figure 10 – Region partitioning

In this design, partitioning by feature size is logical. Currently the design and layout of the transition routing structures is not automated.

4. 3 Layout Output

After the grid-sizing algorithm has completed, each feature has both its own dimensions and its global dimensions. Producing the layout is simply a matter of placing library components at the proper relative locations and adding some small amount of material to interconnect the individual components and to make up any difference between its own dimensions and its global dimensions. The output is in the "dofile" format from the Mentor Graphics CAD tool, IC. Finally, this output file can be run from within any IC design. It will place the entire array, which can be moved at will to integrate into any other design.

5. Conclusion

The areas in the most critical need of further work are switchbox design and multiple specification region integration. As mentioned previously, more flexible switching resources that can support multiple types of switchboxes, adaptable connectivity, and long distance routing lines are needed to truly implement the complexity found in application-specific reconfigurable arrays. In addition, to produce more area-optimized layouts and lessen the impact of improper design, the intelligent use of component clustering to further the concept of layout partitioning has to be explored. An aggressive technique would employ module movement and automated partitioning. At the very least, synFPGA needs to possess an easy way to allow region definition and automated transition routing structure implementation.

Despite the need for further work, synFPGA is able to produce finished, functional reconfigurable logic arrays more efficiently than conventional standard cell implementations. Although it is unable to do so without user interaction and must be made within certain constraints, this does prove that the method of layout generation can drastically reduce design time and make high-performance FPGAs accessible with minimal area sacrifice. As this technique matures, the constraints will lessen and the automation of true heterogeneous designs will become a reality.

Appendix 1 -Input specification

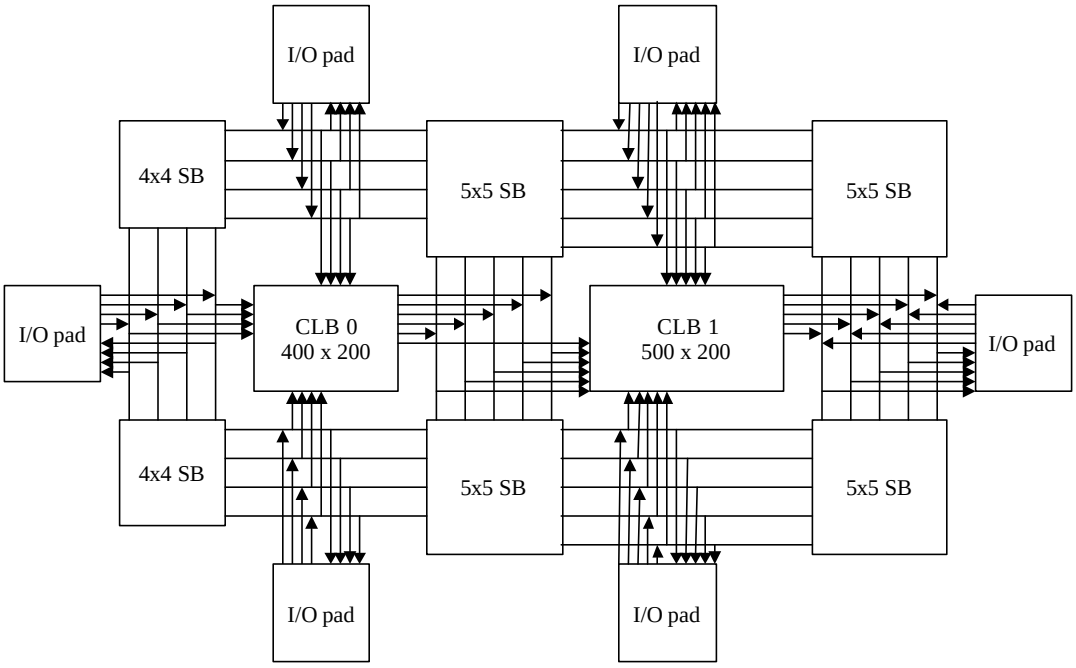

Figure 11 - Example design

General Information File -
Xsize 2
Ysize 1

Logic Block Information File –
Types 2
type 0
xsize 400
ysize 200
instances 1
0, 0
type 1
xsize 500
ysize 200
instances 1
1, 0

Switchbox Information File –
types 3
type 0
leftConnections 4
rightConnections 4
topConnections 4
bottomConnections 4
instances 2
0, 0
0, 1
type 1
leftConnections 4
rightConnections 5
topConnections 5
bottomConnections 5
instances 2
1, 0
1, 1
type 2
leftConnections 5
rightConnections 5
topConnections 5
bottomConnections 5
instances 2
2, 0
2, 1

Connection Block Information File –
horizontal types 4
type 0
numLines 4
numTop 2
type out 4
0, 1, 2, 3
type in 4
0, 1, 2, 3
numBottom 1
type in 4
0, 1, 2, 3
instances 1
0, 0
type 1
numLines 4
numTop 1
type in 4
0, 1, 2, 3
numBottom 2
type out 4
0, 1, 2, 3
type in 4
0, 1, 2, 3
instances 1
0, 1
type 2
numLines 5
numTop 2
type out 5
0, 1, 2, 3, 4
type in 5
0, 1, 2, 3, 4
numBottom 1
type in 5
0, 1, 2, 3, 4
instances 1
1, 0
type 3
numLines 5
numTop 1
type in 5
0, 1, 2, 3, 4
numBottom 2

(Connection Block Information File cont.)	(Connection Block Information File cont.)
type out 5 0, 1, 2, 3, 4 type in 5 0, 1, 2, 3, 4 instances 1 1, 1 vertical types 3 type 0 numLines 4 numRight 1 type in 4 3, 2, 1, 0 numLeft 2 type out 4 3, 2, 1, 0 type in 4 3, 2, 1, 0 instances 1 0, 0 type 1 numLines 5	numRight 1 type in 5 4, 3, 2, 1, 0 numLeft 2 type out 5 4, 3, 2, 1, 0 type passthroughout instances 1 0, 1 type 2 numLines 5 numRight 2 type out 5 4, 3, 2, 1, 0 type in 5 4, 3, 2, 1, 0 numLeft 1 type out 5 4, 3, 2, 1, 0 instances 1 0, 2

Keyword	Function
general :: xsize (int num)	Indicates number of logic blocks horizontally
general :: ysize (int num)	Indicates number of logic blocks vertically
all :: types (int num)	Defines total number of types used in design
all :: type (int index)	Indicates beginning of singular type definition
all :: instances (int num)	Indicates total number of given type – followed by num x,y coordinates to indicate locations
logic :: xsize (int size)	Indicates horizontal dimension
logic :: ysize (int size)	Indicates vertical dimension
connection :: horizontaltypes (int num)	Indicates total number of horizontal connection block types
connection :: verticaltypes (int num)	Indicates total number of vertical connection block types
connection :: numLines (int num)	Indicates number of routing lines intersected by connection block
connection :: numTop (int num)	Indicates number of connection units on top; followed by num types of units
connection :: numBottom (int num)	Indicates number of connection units on bottom; followed by num types of units
connection :: numRight (int num)	Indicates number of connection units on right; followed by num types of units

connection :: numLeft (int num)	Indicates number of connection units on left; followed by num types of units
connection :: type in (int X)	This unit type takes one of X possible inputs from routing channel; followed by X integers indicating wire indexes for input
connection :: type out (int X)	This unit type can output a signal to up to X wires from routing channel; followed by X integers indicating wire indexes for output
connection :: type passthroughin	This unit type allows neighboring logic blocks to connect without utilizing a routing channel wire; in designation indicates unit side is the source of signal
connection :: type passthroughout	This unit type allows neighboring logic blocks to connect without utilizing a routing channel wire; out designation indicate unit side is the reciever of signal
switchbox :: leftConnections (int num)	Indicates number of wires entering left side of switchbox
switchbox :: rightConnections (int num)	Indicates number of wires entering right side of switchbox
switchbox :: topConnections (int num)	Indicates number of wires entering top side of switchbox
switchbox :: bottomConnections (int num)	Indicates number of wires entering bottom side of switchbox

Appendix 2 – Alternative Connection Block Design

As described earlier, under our current design theory, a four to one multiplexor or a one to four demultiplexor would each consist of four connection library modules. That translates to four tri-state buffers and four memory units. Another implementation, to reduce the number of memories required, could use two memory units and a two to four decoder to drive the four tri-state buffers. See Figure 12. While such a system could improve reconfiguration time, we chose our current system because of area considerations. Figure 13 shows a comparison of different multiplexor arrangements. The chart shows both the area required to construct certain multiplexors using a one to one tri-state buffer to memory ratio and the area required to construct the same multiplexors using an x to log (x) tri-state buffer to memory ratio. These numbers should also provide reasonable area estimation for demultiplexing components.

Clearly, in every case besides the two to one, 28 to one, and 32 to one multiplexors, the area of the synFPGA components is smaller than the more traditional x to log (x) ratio components. This is primarily due to the area needs of the decoder. In addition, when switching components of intermediate size are required, such as 12 to one or, worse yet, twenty to one, the synFPGA implementation is able to produce an exact intermediate multiplexing or demultiplexing component. The implementation utilizing decoders is forced to produce a component sized almost as large as one the next largest power of two. This leads to considerable amounts of wasted area. This is quite

acceptable, as it is expected that most of the components actually used in the designs will be larger than two to one components and smaller than 32 to one components.

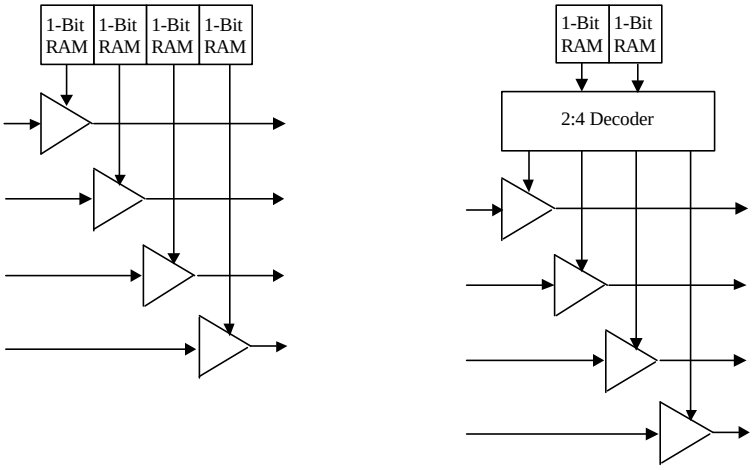

Figure 12 – Two Implementations of Tri-state Buffer Control

	Utilizing synFPGA Modules	Utilizing Decoders
2:1 Mux	13632 Λ^2	7770 Λ^2
4:1 Mux	27264 Λ^2	38763 Λ^2
8:1 Mux	54528 Λ^2	80400 Λ^2
12:1 Mux	95424 Λ^2	110427 Λ^2
16:1 Mux	122688 Λ^2	121627 Λ^2
20:1 Mux	102240 Λ^2	190035 Λ^2
24:1 Mux	184032 Λ^2	201235 Λ^2
28:1 Mux	218112 Λ^2	212435 Λ^2
32:1 Mux	245376 Λ^2	223635 Λ^2

Figure 13 – Area Considerations for Two Implementations
The calculations for all components 12:1 and larger were made based upon 2-level implementations.

In addition, the calculations for the decoder implementation are very conservative because no area penalty was given for the possibly awkward shapes that would be produced. That is, the calculations were mostly based upon simple area addition regardless of what bounding box would actually result. The inter-component routing and complicated memory shift-chain would add considerable area. Further, these designs, regardless of size beyond purely homogeneous components, would not fit nicely into a regular arrangement as shown in Figure 8.
However, this is not to say that the design is not without drawbacks. While the eight to one component saves 32% of area, it includes five more programming bits. Due to the shifting nature of the current architecture, while this does increase reconfiguration time, the area savings are considerable enough to possibly ignore this fact. However, if in the

future we were to attempt to create a parallel loading system, this would not be the case. Doubling of the number of programming bits would greatly increase the area as the number of loading lines would also have to double. In the 16:1 case and above, this would only get worse. The area gains from the missing decoder would quickly be overshadowed by the area penalties caused by the extra programming bits.

Appendix 3 - Library Module Layouts

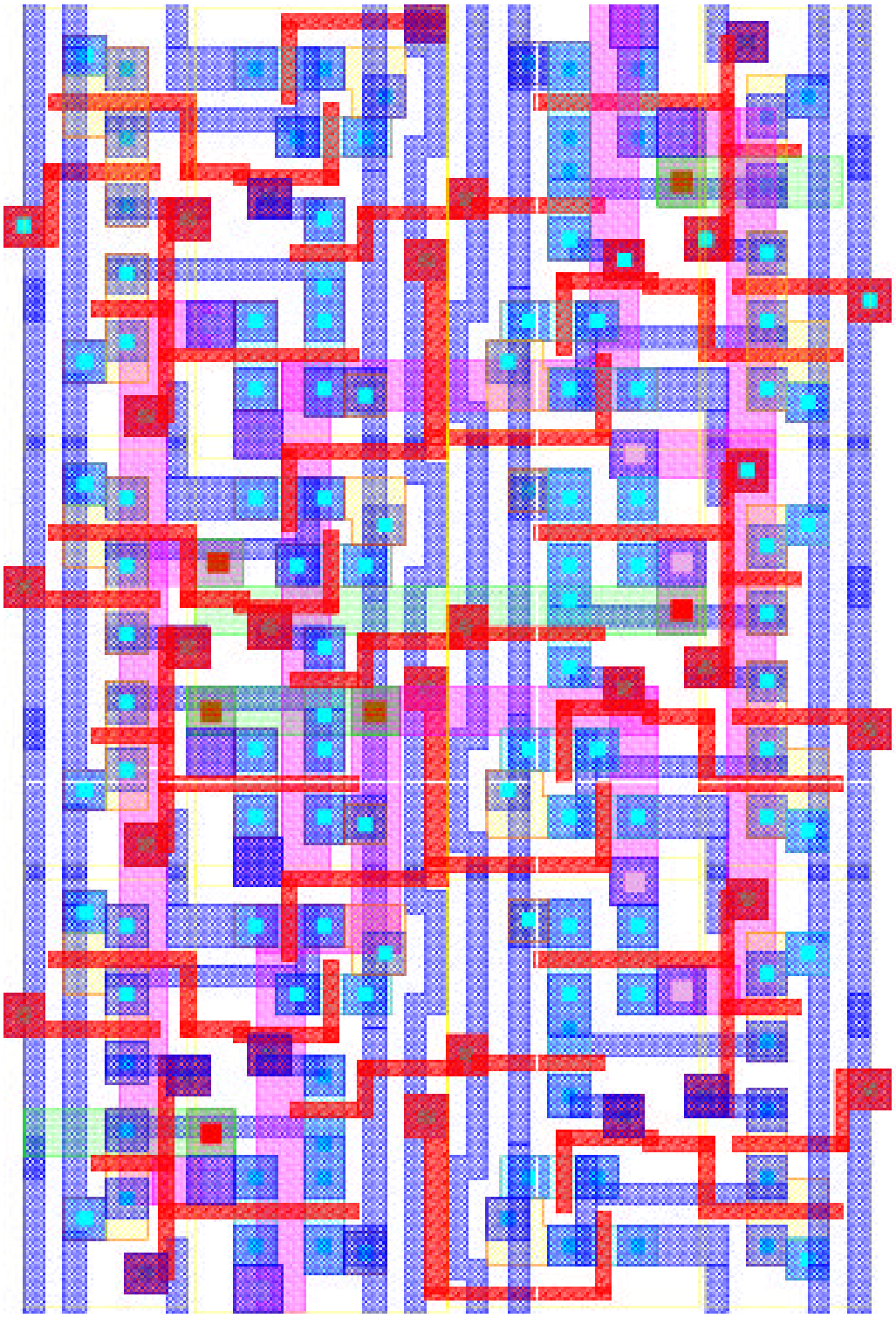

Figure 1 – Basic Switching Module (rotated 90°)

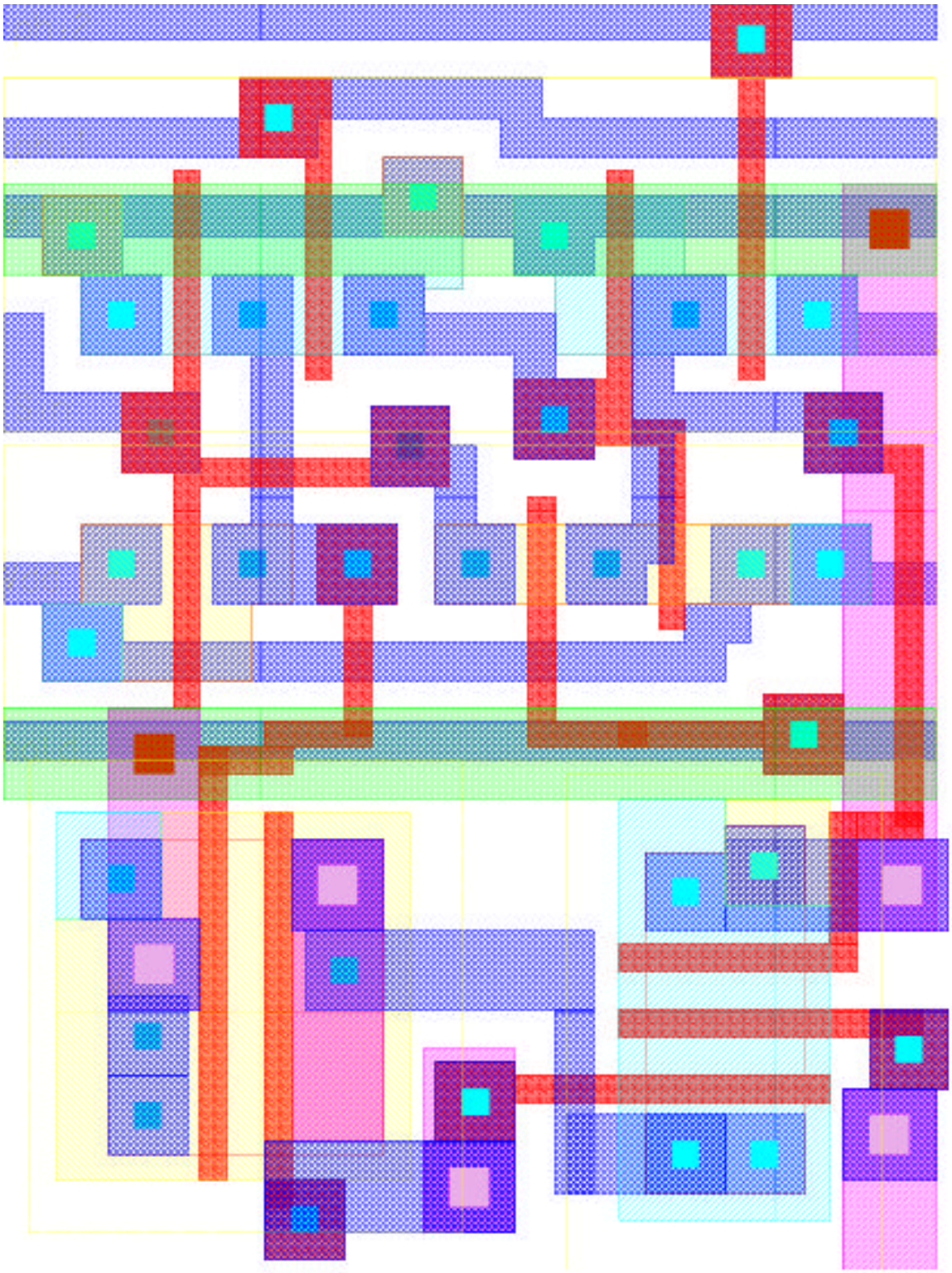

Figure 2 – Basic Connection Block Module

References

[1] Wilton, Steven J. E. “Architecture and Algorithms For Field-Programmable Gate Arrays with Embedded Memory,” Ph.D. Thesis, University of Toronto, 1997.

[2] Xilinx Inc. “The Programmable Logic Data Book”, 1994.

Further Reading

1. Chang, Y. W., D. Wong and C. Wong, “Universal Switch Modules for FPGA Design,” ACM Transactions on Design Automation of Electronic Systems, vol. 1, pp. 80 – 101, January 1996.