

CAD for FPGAs and Cross-Technology Simulation

A Thesis Presented to
the Faculty of the School of Engineering and Applied Science
University of Virginia

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in Electrical Engineering

by
Kyle Martin Ringgenberg
August 2009

APPROVAL SHEET

This thesis is submitted in partial fulfillment of the requirements for the degree of:

Master of Science in Electrical Engineering

Kyle Ringgenberg

This thesis has been read and approved by the examining committee:

Benton Calhoun

John Lach

Stephen Wilson

Accepted for the School of Engineering and Applied Science:

Dean James H. Aylor
School of Engineering and Applied Science
August 2009

CONTENTS

Approval Sheet.....	ii
Table of Figures.....	iv
Abstract.....	v
Motivation	1
Introduction to Programmable Logic.....	1
Programmable Array Logic.....	1
Field Programmable Gate Arrays.....	3
FPGA Challenges	5
Contribution Overview.....	6
Standard Practice FPGA Architecture	7
SRAM Configuration Bitcells	8
Lookup Tables	9
Basic Logic Elements	10
Logic Clusters	11
Configurable Logic Blocks	12
Connection Blocks.....	13
Switch Blocks.....	14
Top Level Architecture	15
Sub Threshold FPGA Implementation.....	17
Design Modifications	18
Physical Design.....	21
FPGA Array Generator	26
Technology Agnostic Simulation Environment.....	31
Simulation Tool	31
Graphical User Interface	34
Conclusions	41
Addendum	43
Glossary.....	44
Acknowledgements.....	45
Bibliography	46

TABLE OF FIGURES

Figure 1 – Simplified Programmable Array Logic Architecture.....	2
Figure 2 – Performance and Energy Efficiency vs. Flexibility for ASICs, CPUs, and FPGAs	4
Figure 3 – Typical 6T SRAM Bitcell	8
Figure 4 – Transmission-Gate Multiplexer Implementation (16x1).....	9
Figure 5 – Lookup Table (4 Input)	10
Figure 6 – Basic Logic Element (4 Input)	11
Figure 7 – Logic Cluster (4x4)	11
Figure 8 – Configurable Logic Block (4x4) – Architecture	13
Figure 9 – Configurable Logic Block (4x4) – Top Level	13
Figure 10 – Connection Block (2 Input).....	14
Figure 11 – Switch Point and Switch Buffer.....	15
Figure 12 – Switch Block (4 Input)	15
Figure 13 – Standard Practice FPGA Array (2x2).....	16
Figure 14 - Switch Block Wire Length Illustration (W=2)	17
Figure 15 – 5T SRAM Bitcell	18
Figure 16 – Energy Consumption of Sub-Vt CLB	20
Figure 17 – Energy vs. Delay for Sub-Vt CLB	21
Figure 18 – Basic Logic Element – Layout	22
Figure 19 – Basic Logic Element – Floorplan.....	22
Figure 20 – Configurable Logic Block – Layout	23
Figure 21 – Configurable Logic Block – Floorplan	23
Figure 22 – Connection Block – Layout.....	24
Figure 23 – Connection Block – Floorplan	24
Figure 24 – Switch Block – Layout.....	25
Figure 25 – Switch Block – Floorplan	25
Figure 26 – Array Generation Tool – Operation	26
Figure 27 – Array Generation Tool – Architecture.....	27
Figure 28 – FPGA Array (2x2) – Schematic.....	28
Figure 29 – FPGA Array (2x2) – Layout	29
Figure 30 – FPGA Array (2x2) – Floorplan	29
Figure 31 – FPGA Chiptop – Layout	30
Figure 32 – FPGA Chiptop – Floorplan	30
Figure 33 – Technology Agnostic Simulation Environment – Architecture	32
Figure 34 – Technology Agnostic Simulation Environment – Toolflow	33
Figure 35 – TASE GUI Architecture	36
Figure 36 – Configuration Mode Screenshot	37
Figure 37 – Batch Editor Screenshot.....	38
Figure 38 – Extrapolate I Screenshot	39
Figure 39 – Extrapolate II Screenshot	39
Figure 40 – Exploration Mode Screenshot.....	40

ABSTRACT

Field programmable gate arrays (FPGAs) implement arbitrary logic at the cost of transistor utilization. They can provide a cost-effective alternative to custom application specific integrated circuits (ASICs) while decreasing time-to-market. For highly parallelized applications, they can outperform general purpose CPUs while still retaining the ability to dynamically update functionality in the field. Additionally, they are a highly lucrative multi-billion dollar industry and are fast permeating product fields. This work first presents a standard practice FPGA and then discusses three key challenges that face the FPGA community. Additionally, it suggests several methods by which these challenges may be overcome.

The first key challenge is the high energy requirements of most FPGAs. Industry has focused primarily on increasing performance at the cost of energy efficiency; thus, most FPGAs are unable to be used in mobile and embedded systems. This is countered by the development and implementation of a sub-threshold FPGA, which is the first of its kind and allows for ultra-low power operation. The second obstacle is the complex nature of FPGA design, which is addressed with the development of an automated FPGA array generation tool, capable of generating an arbitrarily sized logic array. The final challenge relates to the difficulties associated with regularly porting designs to new process technology nodes. This is addressed via the construction of a technology agnostic simulation environment. Ultimately, these three contributions have already shown varying degrees of success in aiding FPGA designers in their continued development of these unique devices.

MOTIVATION

Field programmable gate arrays (FPGAs) are generally used as cost-effective, rapid time-to-market alternatives to application specific integrated circuits (ASICs). However, their high energy requirements prevent them from being practical in mobile and embedded computing systems, such as wireless sensor nodes. Since the summer of 2007 a two-member team (Joseph Ryan and Kyle Ringgenberg) headed by Professor Benton Calhoun at the University of Virginia has been working to develop an ultra-low power FPGA with minimal performance degradation. The work presented herein is composed primarily of the author's contributions to this research group.

Many mobile systems (e.g. cell phones, biometric monitors, and gaming peripherals) must be designed to strike a balance between cost and battery life. An ultra-low power FPGA could shrink the costs associated with producing custom ASICs for many of these devices while still providing comparable battery life. Likewise, such an FPGA could take the place of less energy efficient processors, allowing for devices with extended battery life that are still capable of receiving updates in the field. In short, the commercial implementation of an ultra-low power FPGA could potentially increase battery life and decrease development costs of a wide range of both commercial and consumer products.

INTRODUCTION TO PROGRAMMABLE LOGIC

PROGRAMMABLE ARRAY LOGIC

Prior to the introduction of programmable array logic devices (PALs), circuit designers had few methods available to develop digital logic. It was commonplace to use standard logic devices (independent gates, multiplexers, flip-flops, etc) which were cumbersome and difficult

to scale into complex logical functions. A limited number of programmable logic devices existed, but they suffered from slow operation, high costs, and weak reliability. This changed in 1978 when Monolithic Memories produced their first commercially available PAL device. [1].

Architecturally, PALs are composed of two components. The first is a programmable read-only memory (PROM) that serves as routing fabric for the input signals. All inputs and their complements are selectively connected to the second PAL component. This block consists of banks of AND gates which feed a global OR gate, resulting in a sum-of-products implementation.

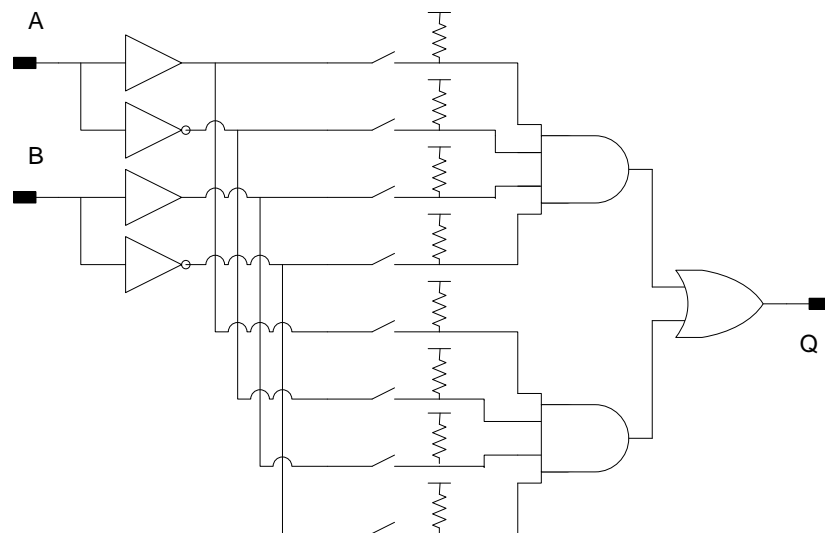


FIGURE 1 – SIMPLIFIED PROGRAMMABLE ARRAY LOGIC ARCHITECTURE

Following the immediate success of the Monolith Memories PAL device, a number of other manufacturers began developing programmable logic chips. However, due to the high costs and precious nature of on-chip transistors, these devices remained limited to basic combinational logic operations (e.g. sum-of-products, product-of-sums) with hard-wired interconnect between logic functions. In 1984 Ross Freeman took a risk on his observation that, if Moore's Law continued, it would become feasible to sacrifice the number of utilized

transistors in exchange for interconnect flexibility. One year later his new company, Xilinx, released the first field FPGA to market. [2].

FIELD PROGRAMMABLE GATE ARRAYS

Since its conception in the 1980s, FPGAs have grown tremendously in complexity and ubiquitousness. The original Xilinx FPGA contained a mere 64 logic blocks and under 10,000 total transistors. [3]. In contrast, variants of the Xilinx Virtex-6 family, released in 2009, contain over 750,000 logic blocks with up to 1,200 I/O pins. [4]. Furthermore, most modern FPGAs contain dedicated computation blocks for high-speed digital signal processing (DSP). These range from multipliers and barrel shifters to PCIe interfaces and CPUs.

By 2010 the FPGA market is expected to surpass \$2.7 billion. These devices are found in innumerable systems from bioinformatics to speech recognition, from ASIC prototyping to radio astronomy. While the maximum internal clock frequency of most FPGAs is limited to 100s of MHz, the inherent parallelism supplied by the FPGA architecture allows them to act as high-performance coprocessors for general purpose CPUs. [5]. Meanwhile their dynamically reprogrammable nature allows for features (bugs) to be added (fixed) after deployment. Furthermore, their highly amortized non-recurring engineering costs make FPGAs a cost-effective alternative to custom ASICs.

Figure 2 illustrates the performance (defined as throughput) vs. flexibility and the energy efficiency vs. flexibility design space of FPGAs, CPUs, and custom ASICs. As shown, ASICs yield the highest performance and efficiency due to the designers' ability to tune circuits for optimality. However, they achieve this at the cost of lowered flexibility. Once an ASIC has been manufactured, it is essentially impossible to modify its fundamental behavior. Furthermore, ASIC fabrication is costly and relatively lengthy. Meanwhile, CPUs provide high flexibility since

they are able to dynamically execute any valid instruction that passes through their pipeline. This flexibility is limited only to the available arithmetic logic unit (ALU) elements and the supported instruction set. The tradeoff for this flexibility is lower performance compared to ASICs. Additionally, most general purpose CPUs exhibit large energy requirements. FPGAs typically sit between ASICs and CPUs with respect to flexibility. They are regularly used as a cost-effective alternative to fabricating a custom ASIC. This drastically reduces non-recurring engineering costs as well as provides the flexibility to update implemented logic when necessary. This flexibility generally comes at the cost of performance. FPGAs typically have maximum clock cycles on the order of 10s to 100s of MHz (compared to the GHz speed of CPUs). [6] This means that, for highly serial applications (e.g.: protocol stack) CPUs will generally perform better than FPGAs. However, throughput is a function of clock speed as well as operations per cycle. Thus, for highly parallelizable applications (e.g.: vector calculations) FPGAs can yield higher performance than CPUs. [7].

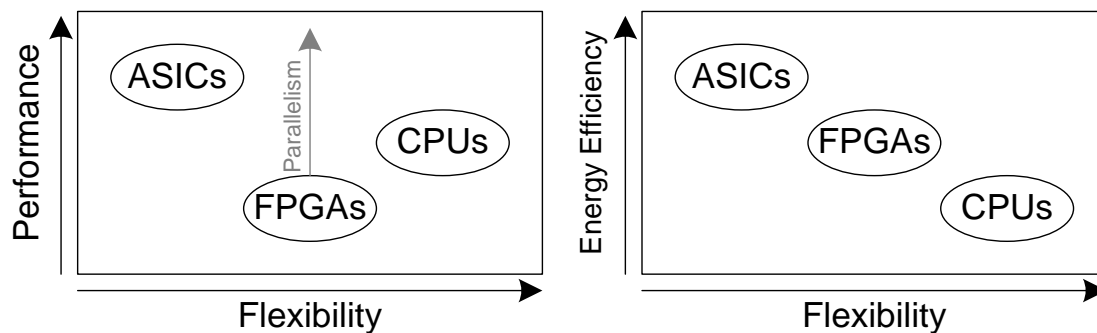


FIGURE 2 – PERFORMANCE AND ENERGY EFFICIENCY VS. FLEXIBILITY FOR ASICS, CPUS, AND FPGAS

It is quite evident that FPGAs are powerful devices that are able to implement arbitrary logic at the cost of transistor utilization. They can provide a cost-effective alternative to custom ASICs while decreasing time-to-market. For highly parallelized applications, they can outperform

general purpose CPUs while still retaining the ability to dynamically update functionality in the field. Finally, they are a highly lucrative multi-billion dollar industry and are fast permeating product fields. FPGAs are not, however, without challenges, several of which will be addressed in the next section.

FPGA CHALLENGES

The primary characteristic that makes FPGAs such extensible devices is the sacrifice of transistor utilization for increased flexibility. All challenges presented herein can be traced back to this design decision. Additionally, these challenges are exacerbated by the fact that the FPGA community tends to be closed-source. Major manufactures such as Xilinx and Altera are reluctant to release architectural designs. There are several academic institutions that have provided excellent resources, such as the University of Toronto, but many architectural details must still be extrapolated from product white pages and usage manuals. Therefore, this work will first present a survey of standard-practice FPGA design. This serves to provide the necessary architectural background information needed by the three key contributions that follow.

For the past 20 years, the FPGA community has been focused primarily on increasing performance. As a result, energy efficiency has suffered to the point where, in the Virtex-6 white paper, Xilinx is quoted as saying: “At 40 and 45 nm process nodes, power has become *the* primary factor for FPGA selection”. [8]. Considering the large number of total transistors, static power from leakage is a major contributor to the overall power consumption of modern FPGAs. This consumption is generally large enough to prohibit them from being used in mobile and embedded computing (e.g. sensor nodes or distributed computing networks). Methods to limit power consumption have become an important element in academic research; this work contributes to the field of knowledge via the implementation of the first sub-threshold FPGA.

This implementation was a joint effort with the work presented herein being the author's primary contributions.

As previously articulated, modern FPGAs are complicated devices composed of thousands of basic logic units, tens of thousands of configuration bits, and millions of transistors. Managing designs of this size, especially with the small teams typical in academia, requires a great deal of forethought as well as dedicated computer aided design (CAD) tools. The second major contribution of this work is the development of an FPGA generator, which is capable of constructing an arbitrarily sized FPGA array (both layout and schematic) given its subcell designs. In this way the designer can remain focused on his circuits and architecture without having to be distracted by assembling the larger array structures.

One final challenge that faces not only FPGA designers, but all circuit designers, is the perpetual miniaturization according to Moore's Law. With new technology nodes being developed roughly every 18 months, designs must regularly be translated between nodes. This process is labor intensive, especially in academia where resources such as time and manpower are quite constrained. Furthermore, circuits may behave in drastically different ways when ported from one technology to another. Thusly, the third and final contribution of this work is the development of a process independent simulation infrastructure that is capable of simulating arbitrary designs across arbitrary process development kits (PDKs). In this way, circuits can be developed not only for the current, state-of-the-art technology, but for upcoming technology nodes as well.

CONTRIBUTION OVERVIEW

As mentioned in the previous section, the work presented herein strives to accomplish three key goals in contributing to the FPGA community. However, before these goals can be

discussed, one must be introduced to standard practice FPGA architecture. Once this background information has been established, the following three contributions can be discussed:

- First, this work presents a sub-threshold FPGA, implemented in a commercial 90nm process. This is the first of its kind, resulting in ultra-low power operation, allowing FPGAs to permeate markets otherwise inaccessible due to energy constraints.
- Second, this work outlines the development of an FPGA generation CAD tool, which helps with rapid and automated FPGA array generation. This simplifies the design process and allows for smaller teams to develop the complex structures that comprise FPGA design.
- Finally, a technology agnostic simulation environment is created which allows test circuits to be analyzed at arbitrary technology nodes. This tool simplifies the creation of future-proof designs and allows for the rapid porting of designs between nodes.

STANDARD PRACTICE FPGA ARCHITECTURE

The primary focus of the FPGA architecture is to be able to implement arbitrary logic alongside a highly flexible interconnect scheme. The topmost components include configurable logic blocks, switch blocks, and connection blocks. The programmable logic is stored in static random access memory (SRAM) configuration bitcells, which are programmed upon initial power-up via extra-FPGA hardware. The following outlines a basic FPGA implementation according to standard practice, starting with the lowest level and moving up hierarchically.

SRAM CONFIGURATION BITCELLS

All configurations within the FPGA are stored as SRAM configuration bitcells (configBits or CBits) which drive either buffers (as is the case with the LUTs) or gates (all other configuration locations). These bitcells are localized within the FPGA fabric itself, so traditional read methods (e.g. bitline droop-driven sense amplifiers) are unnecessary. Instead, data can be tapped directly out of the feedback inverters so long as sufficient care is taken to not drive current back into the bitcell, which could potentially flip the stored data. Traditionally a 6 transistor (6T) SRAM cell is used, as pictured in Figure 3, where the bit lines and word lines are routed throughout the entirety of the FPGA array, touching the sporadically placed bitcells where necessary. [9].

There has been work done investigating alternative memory designs, specifically grouping large banks of SRAM cells together and then running routing fabric out to the required bitcell locations. While this technique may reduce area overhead, it has been shown to be detrimental to energy efficiency. [10]. The following chapter introduces the use of a 5T bitcell that is able to increase density without affecting FPGA runtime energy efficiency or performance.

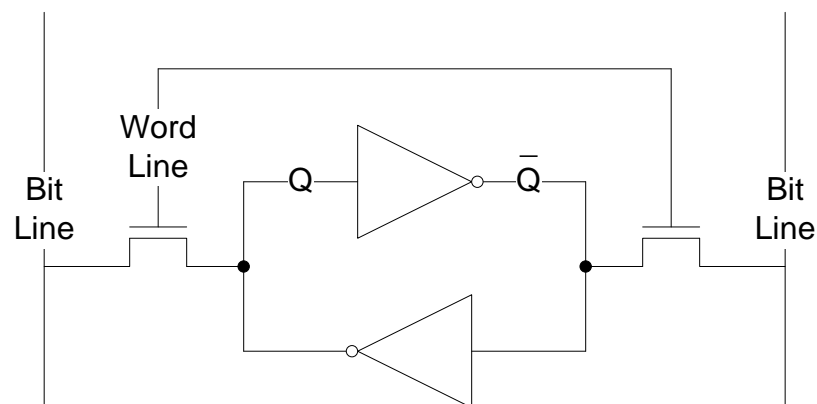


FIGURE 3 – TYPICAL 6T SRAM BITCELL

LOOKUP TABLES

FPGAs make use of lookup tables (LUTs) as their primary programmable logic structure. As depicted in Figure 5, an LUT is composed of numerous bitcells which, via buffers, feed into a multiplexer. The inputs to the transmission-gate mux are driven by locally inverted select signals. Thusly, a unique logical value, Q, is generated for every permutation on the select lines. Until recently, industry has typically used four input LUTs (LUT4) which have been shown to provide near optimal logic utilization. However, beginning with the Virtex-5 family, Xilinx has started implementing six input LUTs to accommodate the modern trend toward wide data paths. [3].

FPGA multiplexers are generally implemented as arrays of transmission gates, as shown in Figure 4. This design is used in lieu of fully driven gates to reduce area overhead. Muxes account for a large portion of the overall logic block design, so saving even just a couple of gates each sums to a rather drastic overall area savings.

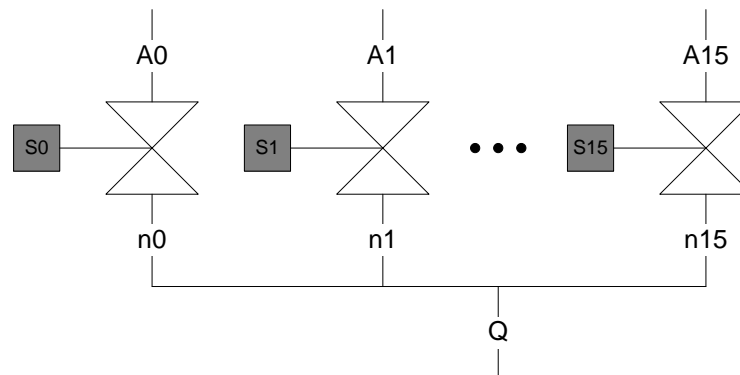


FIGURE 4 – TRANSMISSION-GATE MULTIPLEXER IMPLEMENTATION (16X1)

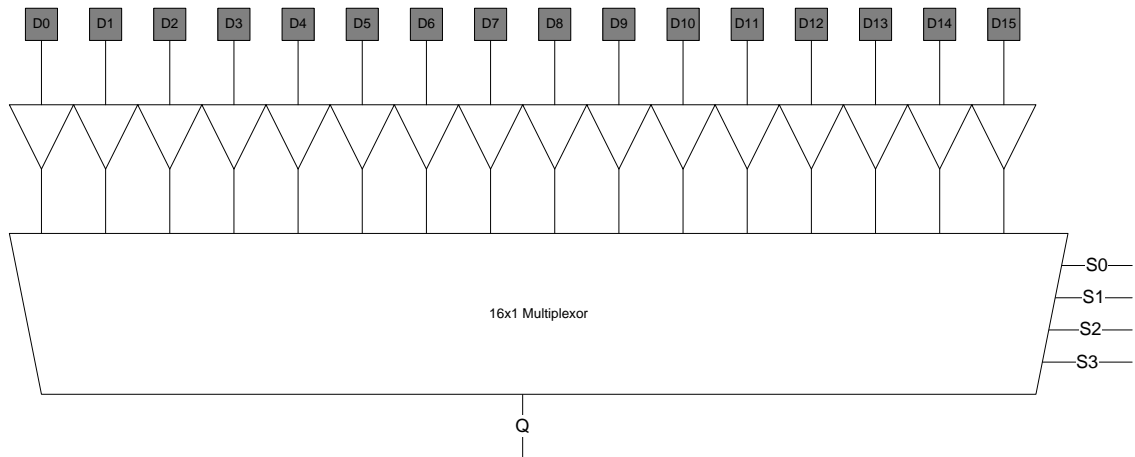


FIGURE 5 – LOOKUP TABLE (4 INPUT)

BASIC LOGIC ELEMENTS

The next hierarchical level up from the LUT is the basic logic element (BLE). This cell allows FPGAs to implement both combinational and sequential logic, according to the value of the configBit driving the select pin of the 2x1 programmable mux. It is important to note that, due to the direct manner by which the configBits are “read”, a 2x1 programmable transmission-gate mux needs only 1 configBit to set its state. This is accomplished by tapping out both Q and \overline{Q} from the associated bitcell and running the values to the appropriate *enable* and $\overline{\text{enable}}$ of the two transmission gates of the multiplexer.

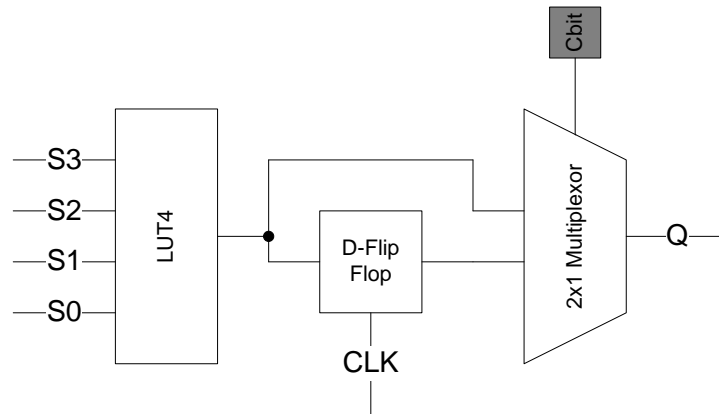


FIGURE 6 – BASIC LOGIC ELEMENT (4 INPUT)

LOGIC CLUSTERS

In FPGA jargon, clustering refers to the number of BLEs that are contained within each configurable logic block. Small clustering typically allows for increased utilization whereas large clustering can potentially minimize the number of times signals must leave the logic block and enter the channel. It is generally accepted that the vast majority of an FPGA's energy consumption comes from the interconnect. Therefore, smaller clustering is desirable when area efficiency is the primary metric of interest, whereas larger clustering is desirable when energy efficiency is of greater importance. Clustering of 4 BLEs is a reasonable estimate of most commercially available FPGAs, although some targeted for lower-power operation cluster as highly as 16 BLEs. [11].

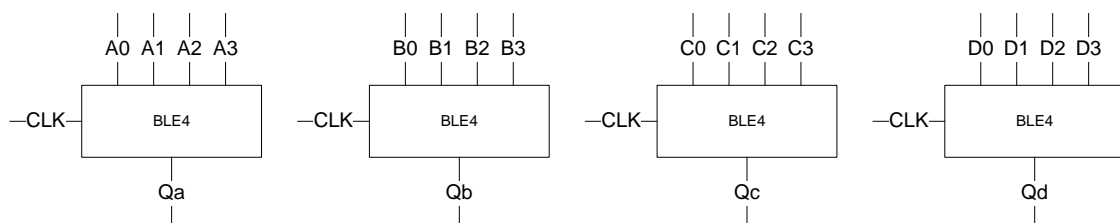


FIGURE 7 – LOGIC CLUSTER (4X4)

CONFIGURABLE LOGIC BLOCKS

The configurable logic block (CLB) is the topmost programmable logic level. Its purpose is to route inputs and outputs of the logic cluster to each other as well as to the block-level I/O pins. This interconnectivity is most easily understood via an example. It has been shown that, for near optimal logic utilization (98%), a CLB should have $2N+2$ inputs, where N is the cluster size. [9]. Thus, for the 4x4 logic cluster described above, the CLB should contain 10 inputs and 4 outputs. These 14 I/O signals, along with a logical 1 and 0 (VDD and VSS) for testing purposes, are fed into the inputs of an array of sixteen 16x1 programmable muxes. The outputs of these 16 muxes are fed into the 16 inputs of the logic cluster. Therefore, any of the 10 inputs to the CLB can be routed to any of the 16 inputs of the logic cluster. Likewise, any of the 4 outputs of the logic cluster can be routed back into the inputs of any of the 4 BLEs.

When symbolically and physically encapsulated, the CLB I/O is typically chronologically patterned around the block as in Figure 9. This is done in order to have both inputs and outputs available from all 4 directions, hereby referred to as N, E, S, & W.

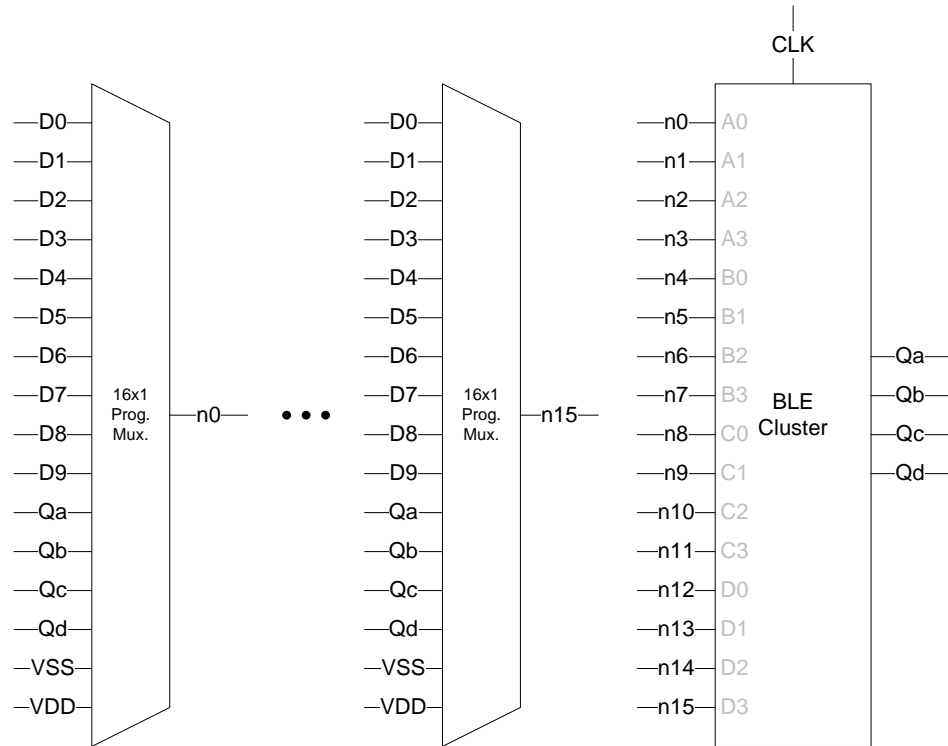


FIGURE 8 – CONFIGURABLE LOGIC BLOCK (4X4) – ARCHITECTURE

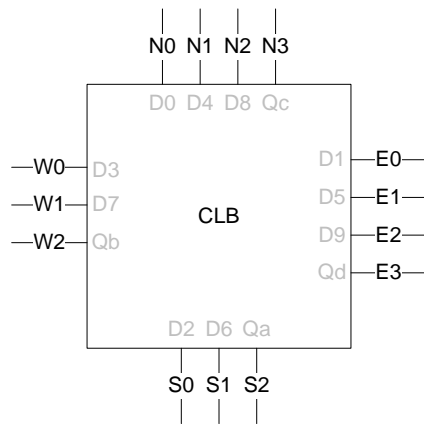


FIGURE 9 – CONFIGURABLE LOGIC BLOCK (4X4) – TOP LEVEL

CONNECTION BLOCKS

The connection blocks of an FPGA serve as the interface between the CLBs and the interconnect. Each of the I/O signals of the CLB are routed through an N/2 programmable mux, where N is the number of channels in the interconnect. To comply with the default functionality of the open source place and route tool VPR, it is advisable to alternate channel connections for

each input or output signal as illustrated in Figure 10. Only connecting to half of the channel wires decreases the load capacitance of each CLB I/O signal, which can grow quite large considering channel widths in excess of 100 wires are regularly used. For illustration purposes, the channel is fixed to 36 wires, a manageable yet still reasonably routable number.

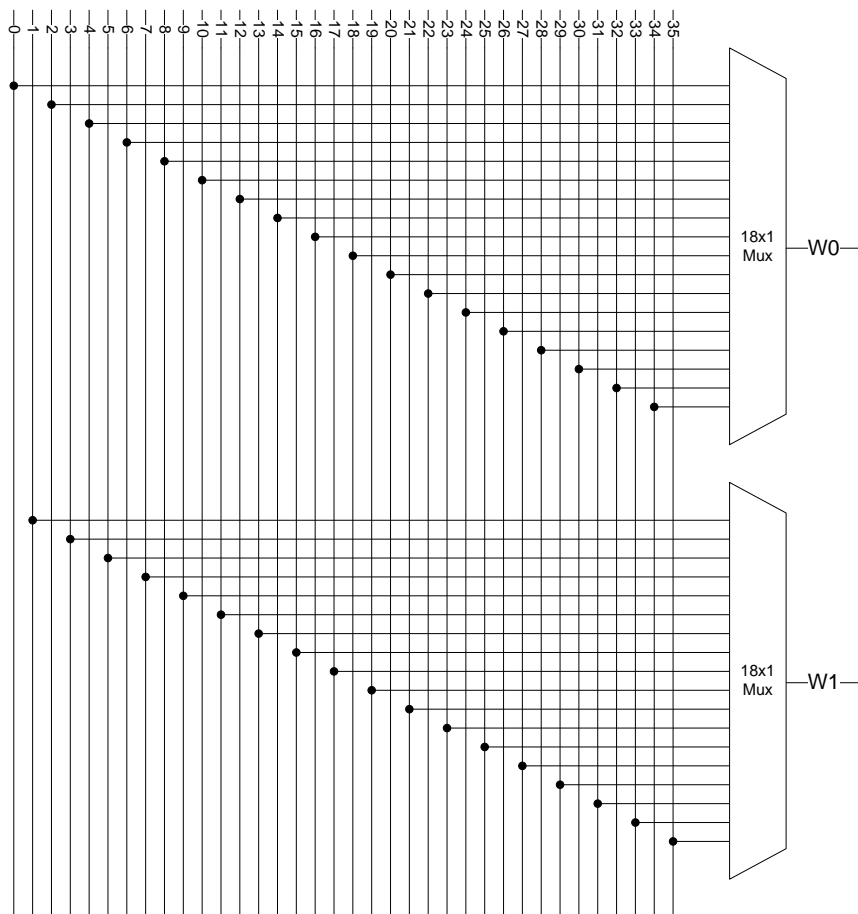


FIGURE 10 – CONNECTION BLOCK (2 INPUT)

SWITCH BLOCKS

The switch blocks make FPGAs unique compared to other programmable logic devices. Each switch block is composed of groups of switch points, one for each wire connection that is desired. A switch point is a 6-way programmable connection as shown in Figure 11. Any of the four wire directions can be connected to any one or more of the others via tri-state buffers.

Since there are 2 programmable tri-state buffers per switch buffer (for bi-directionality), six switch buffers per switch point, and assuming all 1-length wires (discussed in the next section) N switch points per switch block, where N is again the number of wires in the interconnect, it is clear that the number of transistors in a switch block grows quite rapidly. For example, the 4 wire switch block in Figure 12 is composed of $((8+12) \times 6) \times 4 = 480$ transistors.

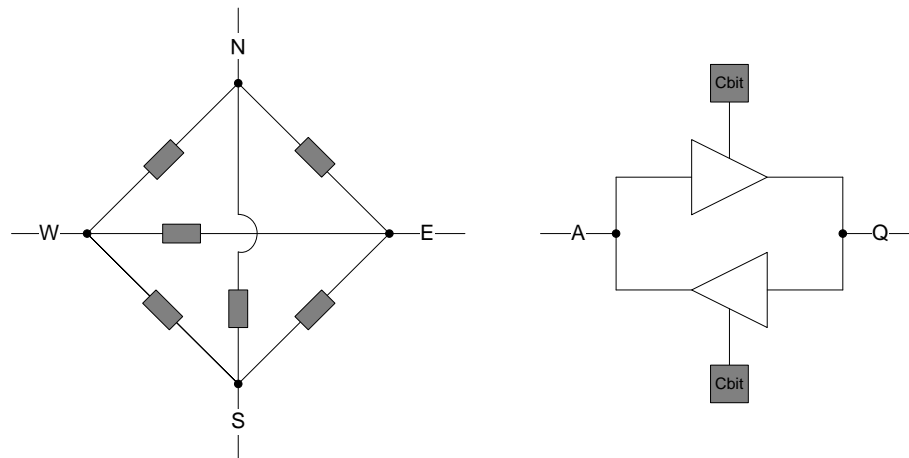


FIGURE 11 – SWITCH POINT AND SWITCH BUFFER

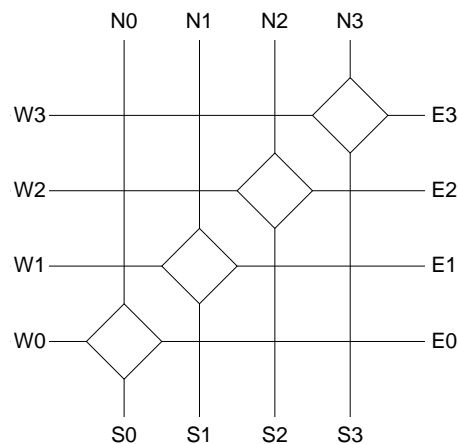


FIGURE 12 – SWITCH BLOCK (4 INPUT)

TOP LEVEL ARCHITECTURE

As stated in the beginning of this chapter, the top-level FPGA fabric is composed of three pieces: CLBs, connection blocks, and switch blocks. Figure 13 depicts the structure by

which these components are tied together. In this example there are 2x2 CLBs; thus, this FPGA is referred to as a 2x2 array. Note: there is always 1 more row/column of switch blocks than the array size, which is based on the number of CLBs. Another idiosyncrasy is the concept of wire lengths, which was alluded to previously. Generally, every interconnect wire does not connect to a switch point at every switch block. Instead, each wire stretches W segments before being connected. This decreases the number of switch points per switch block as shown in Figure 14.

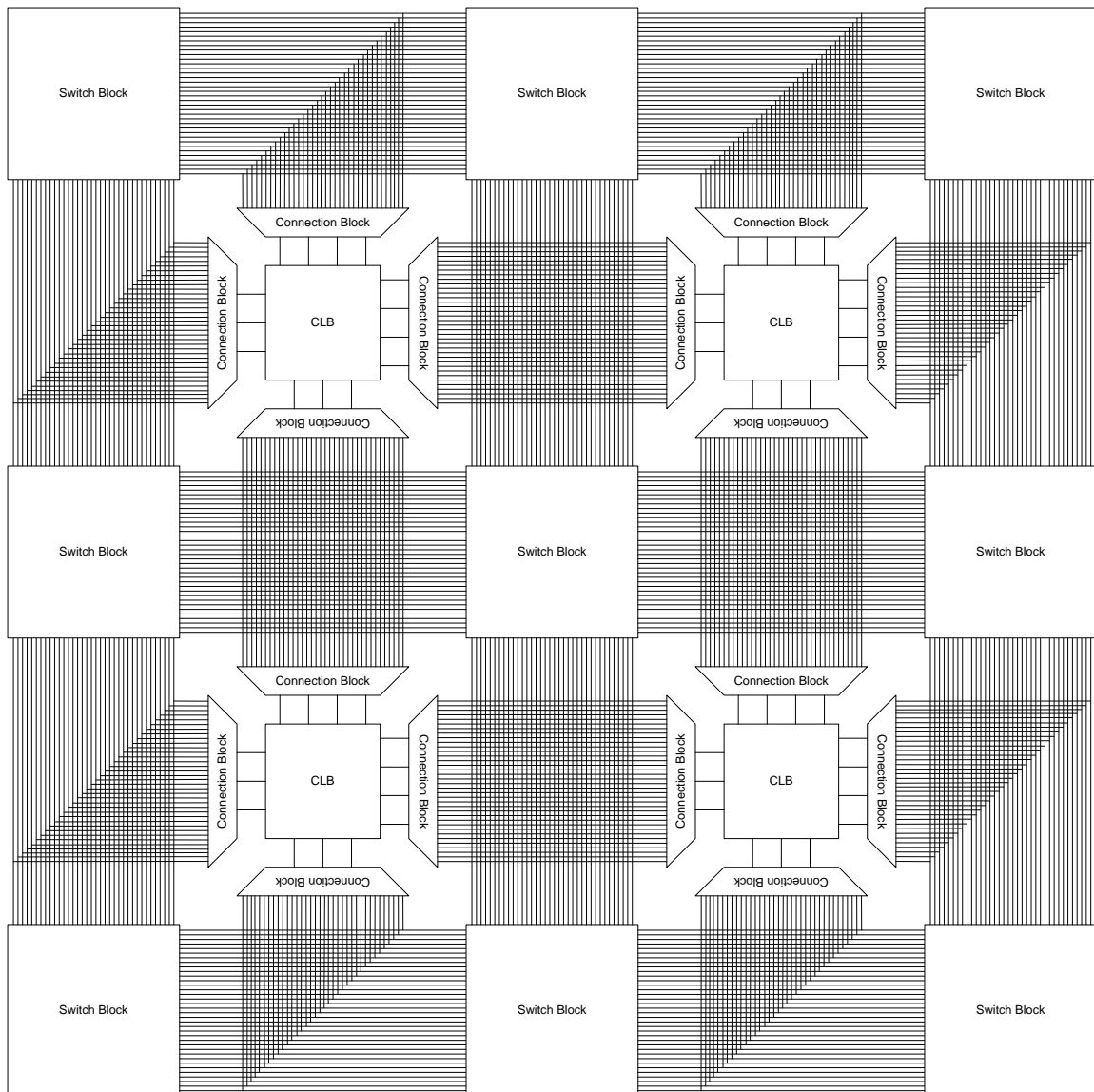


FIGURE 13 – STANDARD PRACTICE FPGA ARRAY (2X2)

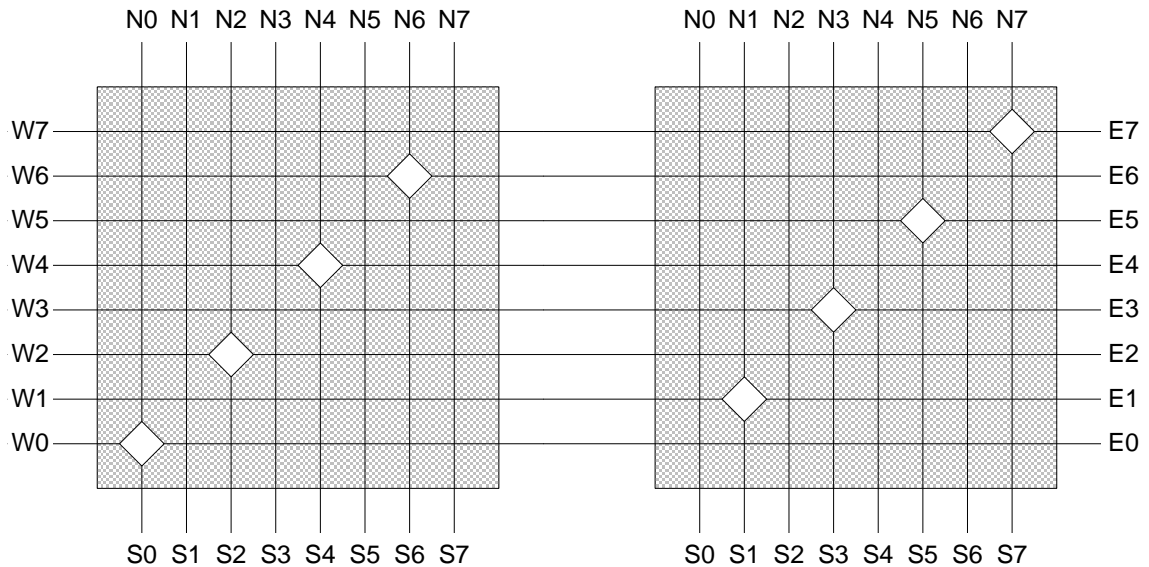


FIGURE 14 - SWITCH BLOCK WIRE LENGTH ILLUSTRATION (W=2)

This concludes the description of a standard-practice FPGA. All architectural decisions are based on public documentation from the primary FPGA manufactures as well as a number of details that have been developed in academia. The next chapter will outline a number of changes made to this default design during the implementation of a sub-threshold FPGA.

SUB THRESHOLD FPGA IMPLEMENTATION

Due to the quadratic relationship between active energy and supply voltage (VDD) as expressed in the first half of the equation below, VDD is a very effective variable for decreasing energy consumption. However, as VDD moves below the threshold voltage (V_t), leakage energy will eventually dominate active energy. The point where active energy and leakage energy cross is referred to as the minimum energy operating point. [12].

$$E_{Total} = C_{eff}V_{DD}^2 + W_{eff}L_{DP}KC_gV_{DD}^2e^{\frac{-V_{DD}}{nV_{th}}}$$

$$E_{Total} = E_{Active} + E_{Leakage}$$

As previously stated, the first major goal of this work is to describe the implementation of an ultra-low power FPGA. This was accomplished by making several changes to the standard practice FPGA which allow it to operate in the sub-threshold region, thereby decreasing active energy and, by extension, total energy.

DESIGN MODIFICATIONS

The first design decision that deviates from the previously described standard practice FPGA is the use of a 5 transistor (5T) SRAM bitcell. [13]. Another member of the FPGA team showed, via exhaustive simulation, that the 5T bitcell can reliably be written with the use of both an overcharged wordline and extra delay holding the value of the bitline. The data stored in the configBits of an FPGA are only written during the programming phase, which generally occurs during power up. This implies that write delay and energy are inconsequential to general operation. Therefore the extra delay accrued by programming the 5T bitcell compared to the 6T bitcell is a fair tradeoff for the gain in area resulting from 1 fewer transistor as well as the decrease in capacitance resulting from 1 fewer bitlines.

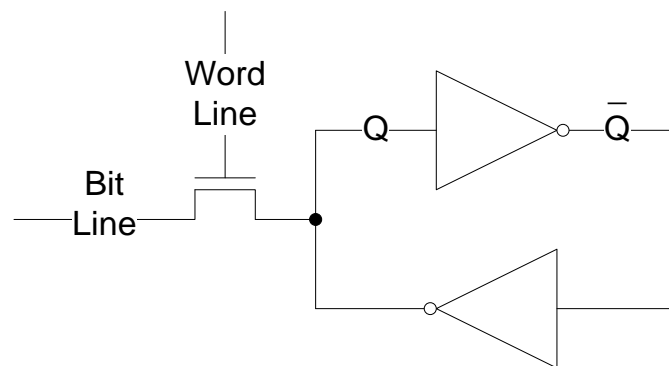


FIGURE 15 – 5T SRAM BITCELL

The second change to the standard practice FPGA is the use of high V_t transistors in all bitcells. High V_t devices decrease the amount of leakage current, effectively lowering the

minimum energy operating point. [14]. This is a common technique among memory designers and it translates effectively to FPGA design.

The final design change used to accommodate sub-threshold operation is the strategic placement of 2 additional buffers in the BLE path. The first is placed between the 3rd and 4th stages of the LUT multiplexer. This combats the signal degradation that occurs after traveling through several stages of transmission gates. The second is placed at the end of the BLE, just before the signal is routed back into the large multiplexors that make up the logic cluster. This counteracts the fanout that the signal must endure before being routed to its final location. Another member of the FPGA team verified these locations via exhaustive simulation.

SIMULATION DATA

The sub-threshold FPGA has been simulated in a commercial 90 nm technology process. In order to obtain energy and delay data, all bitcells were initialized to 0 and the appropriate cells were set to 1 to program an inverter chain into the CLB. The author's contribution has been focused on the CLB, so the following simulation data will ignore the interconnect component of the FPGA. It is important to note that, while the interconnect generally accounts for the majority of FPGA energy consumption, the CLB efficiency is a non-negligible component. Furthermore, as clustering sizes increase, the CLB encapsulates more of the total energy and delay of an FPGA.

Figure 16 illustrates the total energy consumption of the CLB when each of the four BLEs is programmed to invert its input signal. The graph indicates that the total energy consumption of the CLB is dominated by active energy. Additionally, the minimum energy operating point lies within the functional region of the sub-threshold FPGA. This point occurs at roughly 315 mV and represents a total energy consumption of 23 fJ, nearly 13 times less energy than the FPGA draws

at full inversion (1.2 V). However, energy is only one of the two major metrics of interest; the effect that sub-threshold operation has on delay (i.e.: performance) must also be accounted for.

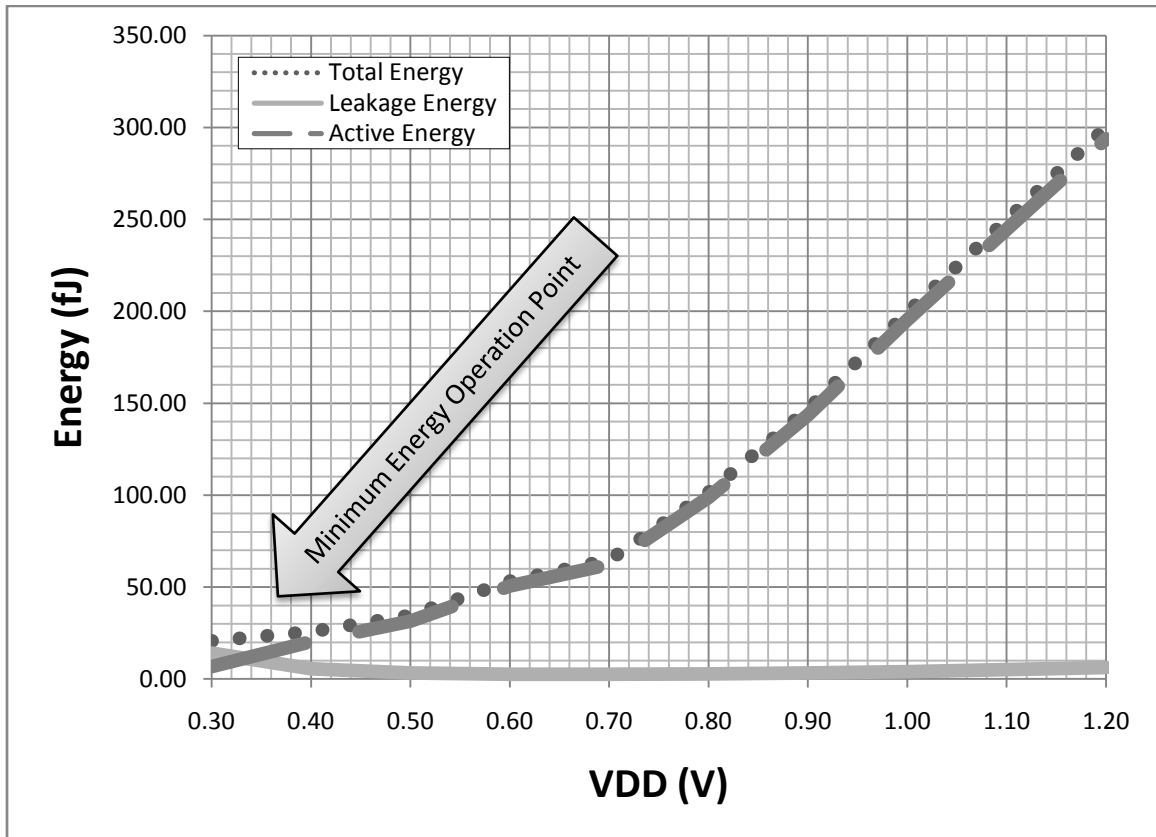


FIGURE 16 – ENERGY CONSUMPTION OF SUB-VT CLB

Figure 17 shows the energy vs. delay plot for this sub-threshold CLB. To generate this plot, VDD has been swept from 1.2 V to 0.3 V (well within the sub-Vt regime) by 100 mV increments. The steep drop in energy on the left-hand side of the plot indicates that lowering VDD has a strong effect on energy, but does not immediately affect delay. At full inversion, the delay through this CLB is 1.38 nS while the energy expenditure is 300 fJ. A relatively minor (10%) decrease in delay, resulting from lowering VDD to 525 mV, yields a dramatic 7.5x decrease in energy (40 fJ). Thusly, sub-threshold operation holds a great deal of potential in drastically reducing energy consumption while minimizing the negative effect on performance.

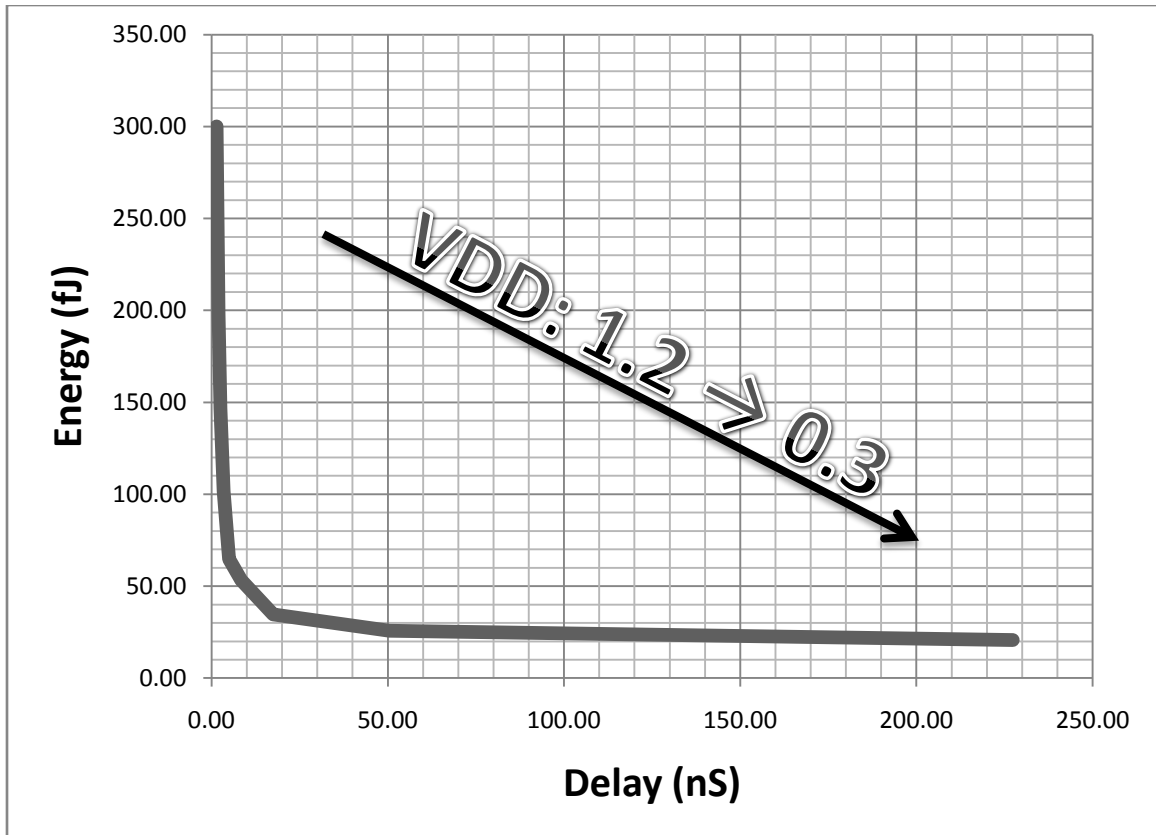


FIGURE 17 – ENERGY VS. DELAY FOR SUB-VT CLB

PHYSICAL DESIGN

The previously described sub-threshold FPGA has been physically implemented in the same 90 nm commercial process that the above simulations were completed in. Additionally, a second, far more radical design has been implemented on the same piece of silicon. This second architecture, however, has primarily been designed by a fellow member of the FPGA team, so it will not be discussed within the context of this work.

The layout that follows has been verified for functionality and sent to the manufacturing foundry. However, due to unforeseen circumstances, the chip will not be fabricated for several months. Nevertheless, this layout is unlikely to change and is a good indication of FPGA physical design.

BASIC LOGIC ELEMENTS

This BLE is organized around the 16 configBits that make up the programmable logic aspect of the FPGA. This layout differs slightly from the sub-Vt design described previously in that the two additional buffers have been moved to the second, more radical design, FPGA only. This is the sole discrepancy between this layout and the original design described above.

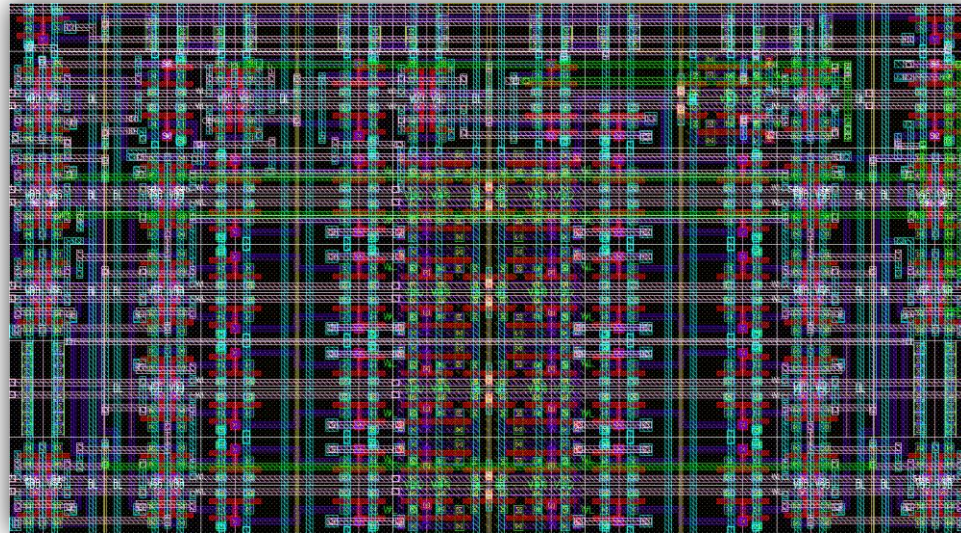


FIGURE 18 – BASIC LOGIC ELEMENT – LAYOUT

D Flip Flop				2x1 Prog Mux	
2x1 Mux	Buffer	Cbit	Cbit	Buffer	8x1 Multiplexor
8x1 Multiplexor		Cbit	Cbit		
		Cbit	Cbit		
		Cbit	Cbit		
		Cbit	Cbit		
		Cbit	Cbit		
		Cbit	Cbit		
		Cbit	Cbit		

FIGURE 19 – BASIC LOGIC ELEMENT – FLOORPLAN

CONFIGURABLE LOGIC BLOCKS

The final CLB area is split evenly between the 4 BLEs and the 16 connection muxes.

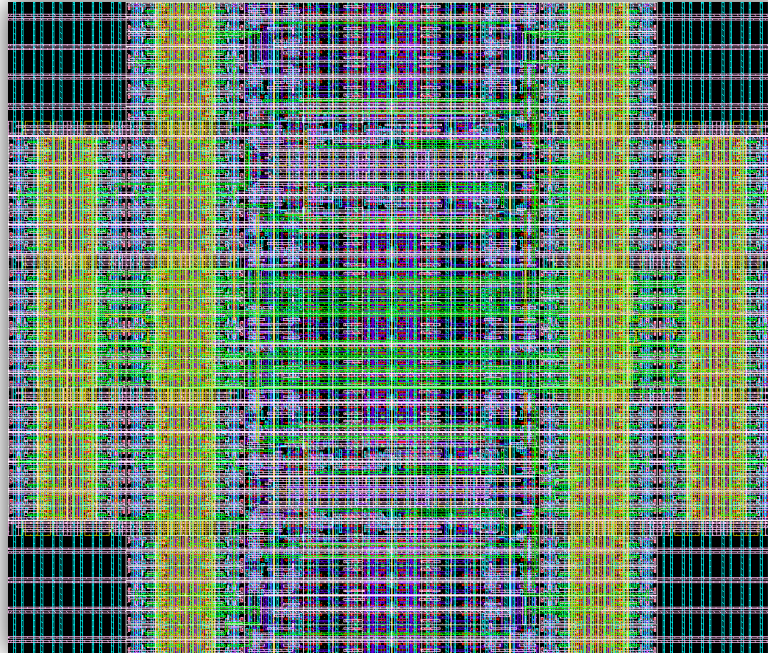


FIGURE 20 – CONFIGURABLE LOGIC BLOCK – LAYOUT

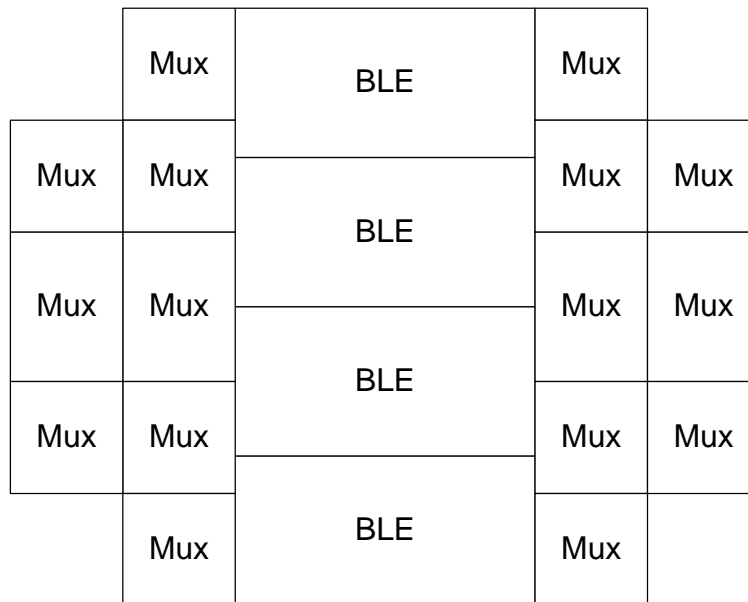


FIGURE 21 – CONFIGURABLE LOGIC BLOCK – FLOORPLAN

CONNECTION BLOCKS

The “fingers” of the connection block need not only reach every wire in the interconnect, but must also not interfere with each other when this block is tiled.

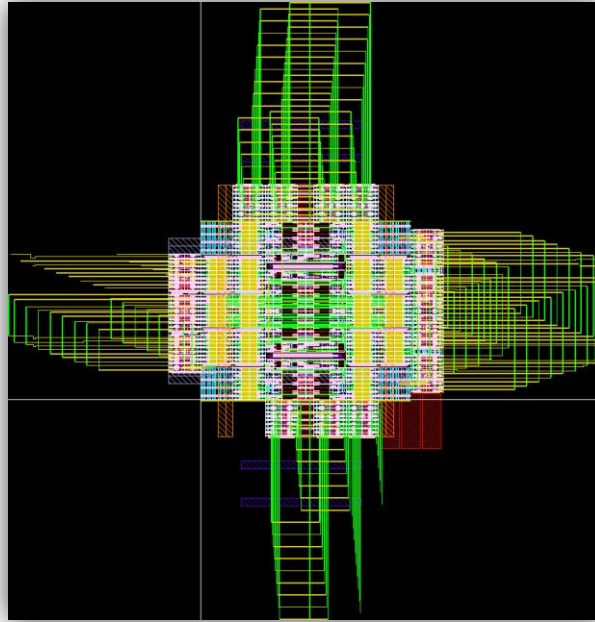


FIGURE 22 – CONNECTION BLOCK – LAYOUT



FIGURE 23 – CONNECTION BLOCK – FLOORPLAN

SWITCH BLOCKS

This switch block layout was done by another member of the FPGA team and is included here for completeness. The 9 switch points indicate that the 36 wires in this interconnect are of length 4 each.

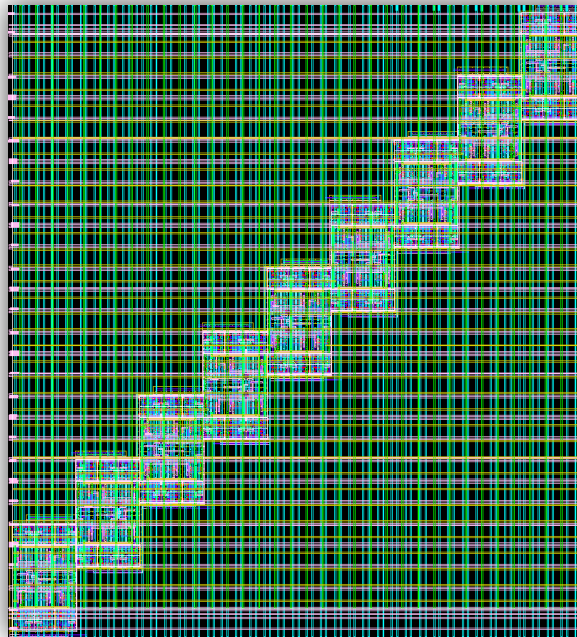


FIGURE 24 – SWITCH BLOCK – LAYOUT

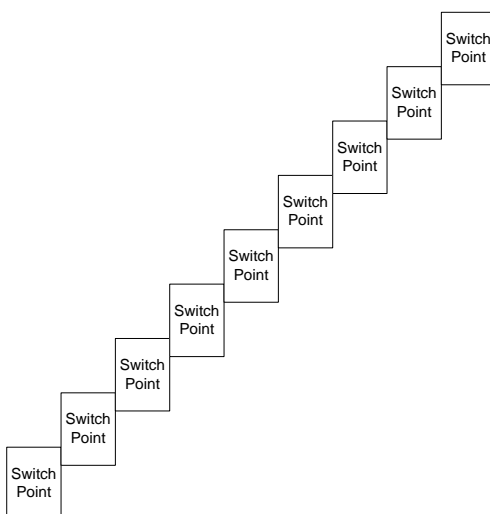


FIGURE 25 – SWITCH BLOCK – FLOORPLAN

FPGA ARRAY GENERATOR

Thus far this work has described the design, simulation, and physical implementation of the major components of a sub-threshold FPGA (thereby accomplishing the second major contribution originally dictated). The final step in the FPGA design process is to assemble the CLB, connection blocks, and switch blocks together into an array. This can be a tedious and time consuming process, so a CAD tool has been developed which is able to generate an arbitrarily sized FPGA array. Interestingly, when this chip was days from being sent to fabrication, the team discovered that the size estimation of one of the two FPGAs was wrong and needed to shrink the array by one row. The CAD tool described herein allowed that change to be completed in a matter of minutes rather than the painstaking hours it would have taken by hand.

TOOL ARCHITECTURE

The array generation tool (AGT) is a set of interconnected SKILL scripts that are launched from the Cadence Virtuoso terminal via the simple command “build(N M)”, as shown in Figure 26, where N is the number of columns of CLBs and M is the number of rows. It then constructs both schematics and layout for every case found in the main loader file. This loader file also holds parameters for where build libraries should go and where dependencies are located.

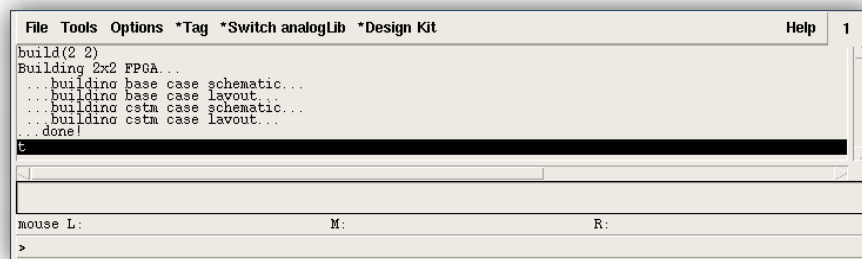


FIGURE 26 – ARRAY GENERATION TOOL – OPERATION

The AGT starts by placing all component symbols in a new schematic file. Next, wires are placed and named to represent block connections. Global signals such as VDD, VSS, CLK, etc are

quite straightforward. In contrast, channel wires are named according to the following formula: D_S_N where D is the wire direction (X or Y), S is the array index of the switch block where this wire originates (e.g.: 0x0), and N is the wire number ($0 \rightarrow 35$) in the channel. In this way, every wire has a unique identifier that allows it to be virtually connected to multiple cell instantiations. Finally, the tool places all I/O pins to the side of the schematic. The separation of the signal pins from the wires themselves serves as a sanity check when the circuit designer later “Checks and Saves” the generated schematic.

In the second step, the AGT starts by opening a new layout file. It maintains an internal state which represents the location of its current progress with respect to the four edges of the array. It then iteratively places components in the appropriate locations, according to global parameters that dictate cell size and relative location. Finally, it places the necessary pins on the top level such that the completed cell will be able to pass layout-versus-schematic (LVS). Figure 27 outlines the top-level AGT architecture.

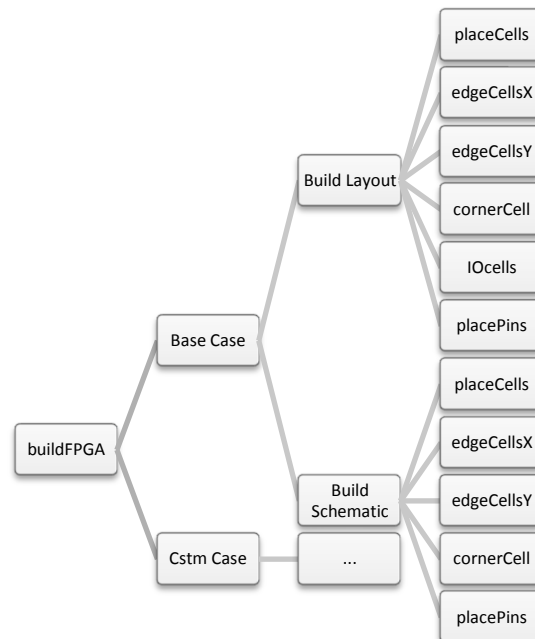


FIGURE 27 – ARRAY GENERATION TOOL – ARCHITECTURE

PHYSICAL IMPLEMENTATION

Figure 28 illustrates the auto-generated schematic for a 2x2 FPGA array. Likewise, Figure 29 shows the AGT built layout for the same FPGA array. These cells are DRC and LVS clean upon creation and require no additional modification from the circuit designer.

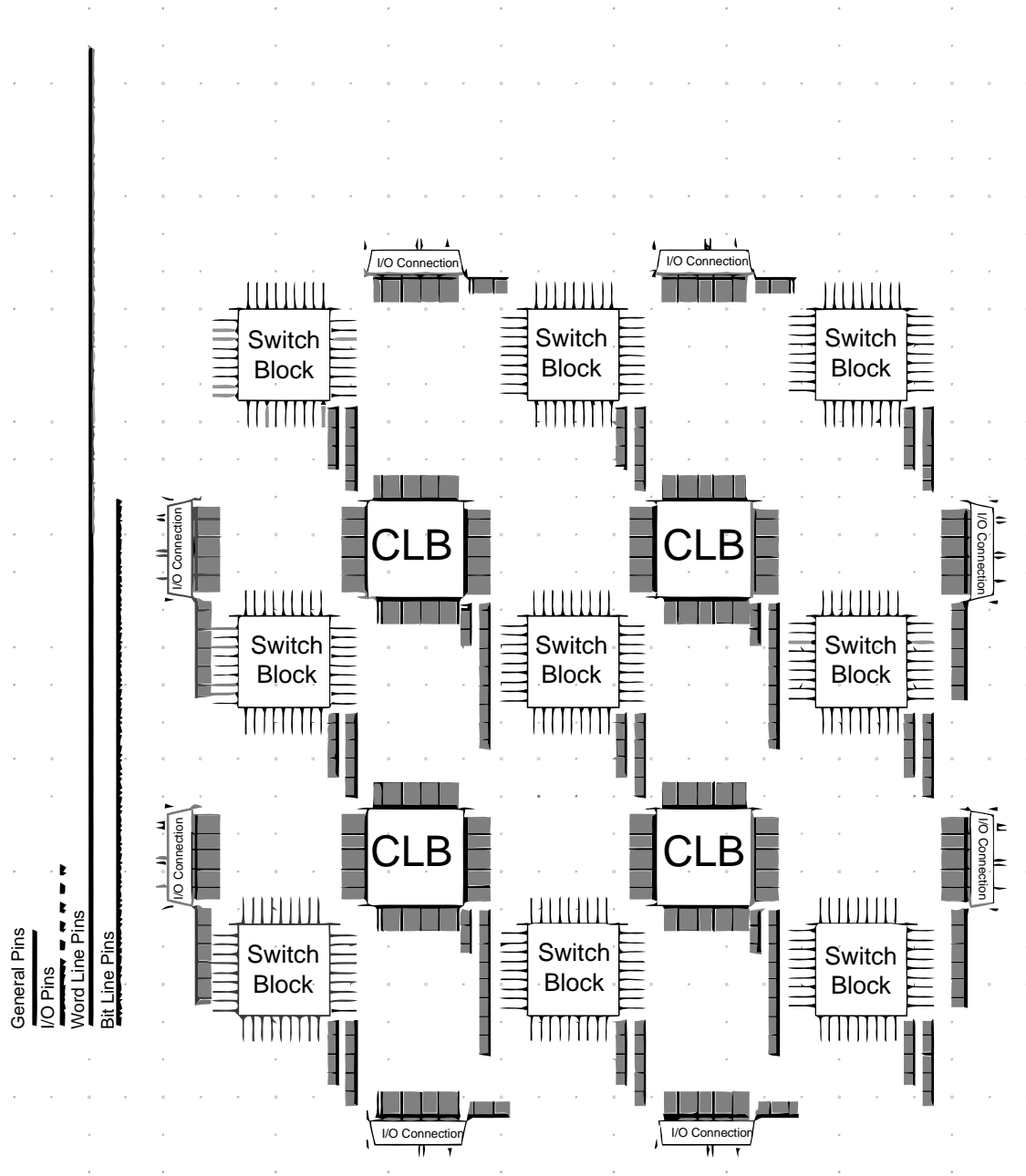


FIGURE 28 – FPGA ARRAY (2X2) – SCHEMATIC

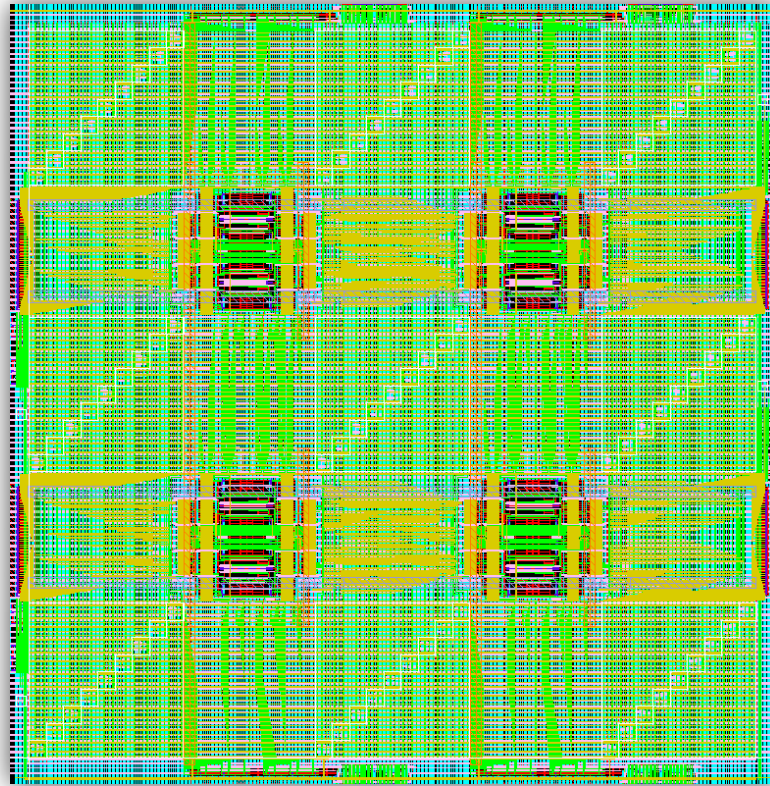


FIGURE 29 – FPGA ARRAY (2X2) – LAYOUT

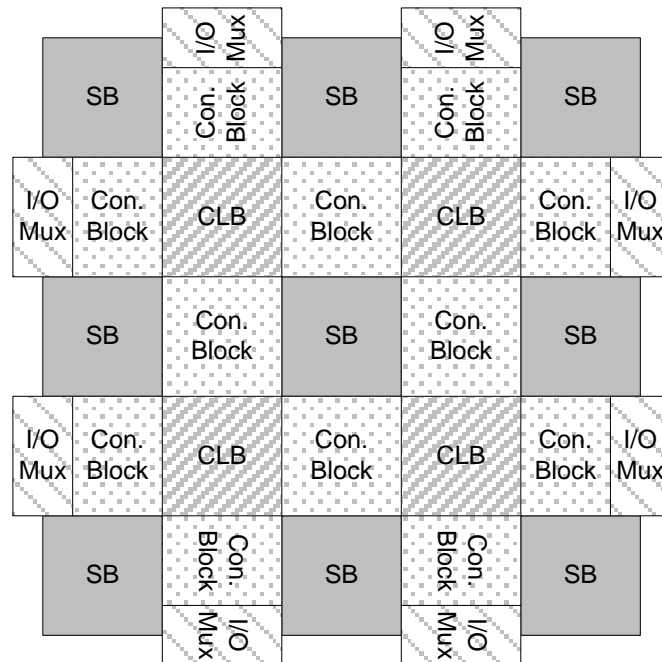


FIGURE 30 – FPGA ARRAY (2X2) – FLOORPLAN

Finally, Figure 31 shows the completed FPGA chiptop, which was sent for fabrication, with the two automatically generated FPGA arrays in the center.

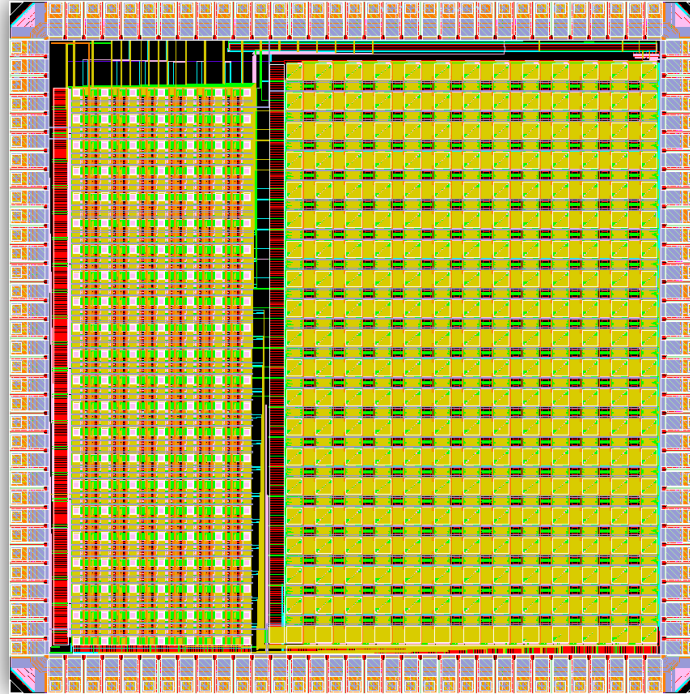


FIGURE 31 – FPGA CHIPTOP – LAYOUT

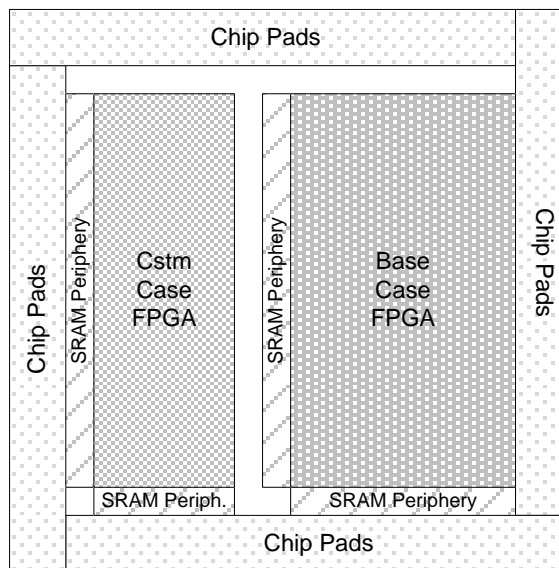


FIGURE 32 – FPGA CHIPTOP – FLOORPLAN

This concludes the outline of an automated FPGA generation tool, the second major contribution of this work. This tool is able to generate an arbitrarily sized FPGA array with minimal overhead needed from the circuit designer. Since it is constructed to make use of pre-generated sub-cells (CLBs, switch blocks, and connection blocks), the tool is easily ported between FPGA architectural designs. As an example, it was able to be ported between the two FPGAs that were designed for the above chiptop in roughly a couple of hours, requiring only new sub-cell paths and layout parameters (e.g.: size, spacing, etc).

TECHNOLOGY AGNOSTIC SIMULATION ENVIRONMENT

The final major contribution of this work is the creation of a cross-technology simulation tool that allows, not just FPGA designers, but also circuit designers in general to rapidly run suites of simulations over broad sets of technology nodes. As Moore's Law continues to hold, circuit designers are faced with the task of porting their designs to a new technology roughly every 18 months. This task is especially daunting within academia where resources such as time and manpower are severely constrained. Thus, it is desirable to be able to test designs not only in the current process node, but in others as well. The technology agnostic simulation environment (TASE) accomplishes this by abstracting all device-level information out of the simulation tests and replacing it with the appropriate data from a given process development kit (PDK) at runtime. [15].

SIMULATION TOOL

The TASE architecture, illustrated in Figure 33, has been designed for usability and robustness. Extensive testing was done to locate parts of the tool flow that could potentially fail, depending on the user's environment or errors in a selected simulation test. These sections

of code were then modified to provide human readable error messages and paths to log files in an attempt to guide the user in correcting the issue. Additionally, every step of the TASE tool flow checks for the existence of plugins, which allow for modifications to the basic operation of the tool without needing to modify its codebase directly.

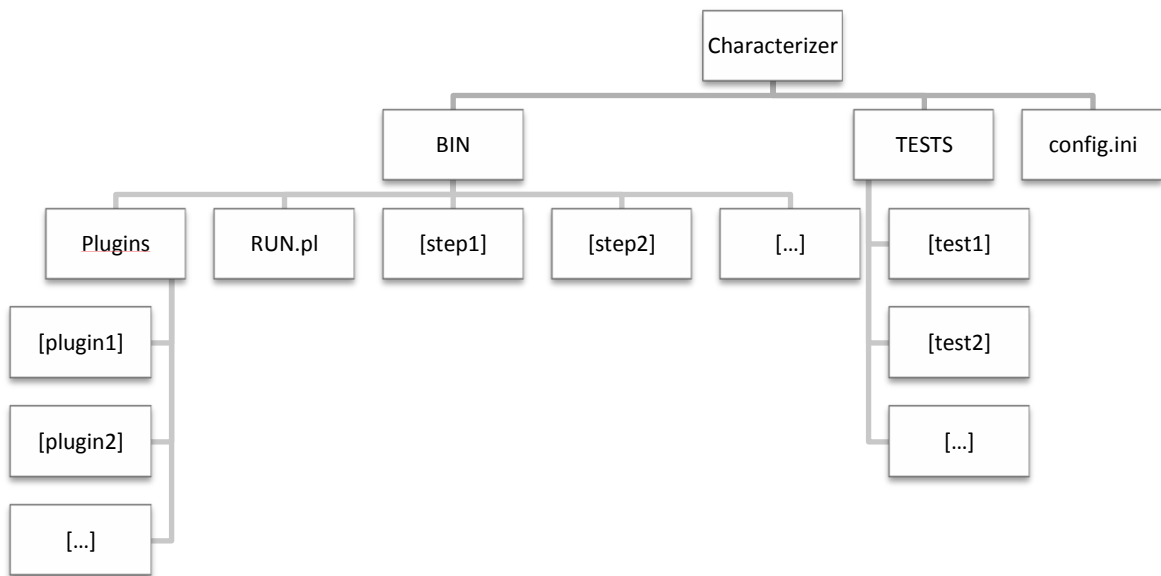


FIGURE 33 – TECHNOLOGY AGNOSTIC SIMULATION ENVIRONMENT – ARCHITECTURE

TASE TOOL FLOW

To use TASE, the circuit designer first creates a config.ini file that contains a collection of variables which describe the PDK that he is interested in simulating. Additionally, this file contains a list of the simulation tests that the user wishes to execute within the selected PDK. He then makes a call to RUN.pl and the TASE tool begins operation.

TASE starts by creating a scratch directory tied to the users login information. In this way, multiple users may work with a single TASE install without interfering with each other's simulations. The tool then creates modified copies of the selected test simulations with

GRAPHICAL USER INTERFACE

The graphical user interface (GUI) of TASE consists of a two parts; the first step (*Configuration Mode*) involves executing a suite of simulations from within the currently existing TASE tool. The simulations selected will vary according the design constraints that the end user is interested in. E.g.: If the user is interested in supply voltage and word-line drooped read assist, the system will select the appropriate test cases from TASE and sweep both of the parameters with respect to each other. This results in fully comprehensive simulation data, which is then passed off to the second stage (*Exploration Mode*).

In this new mode, the previously selected parameters are shown as interrelated slider bars. The software displays a set of graphs depicting permutations of each parameter versus another (one graph for each of the simulations previously selected). Each of these plots is depicted as a thumbnail and may be selected as the primary chart of interest. This primary chart dynamically shifts as the various metrics are changed, allowing for a comprehensive visualization of the design space.

TOOL ARCHITECTURE

The TASE GUI is designed to harness the power of the TASE tool while not requiring any modifications to the current TASE codebase. Considering this, the GUI can be visualized as a wrapper for the TASE tool. It generates standard config.ini files and extracts the TASE-generated encapsulated post script (EPS) plots for use in exploration mode. This is the extent of direct interaction between the GUI and TASE.

The GUI is created as a Matlab user interface, which is built on the JAVA-Swing API. This allows the tool to be executed on any system that has Matlab installed. Furthermore, Matlab is

a dependency for TASE operation, so there are essentially no additional requirements to use the GUI over those that already exist for TASE.

FILE STRUCTURE

As Figure 35 illustrates, the GUI consists of five primary elements and four support directories. Each of the elements consist of a FIG file and an M file. The FIGs are nothing more than a graphical depiction of the GUI. They contain no logic and have been generated by the Matlab 'guide' command. The M files contain all of the control code for the various GUI elements and have been hand-written to allow for extensibility, detailed comments, and object oriented (OO) design.

The *helpers* directory contains extractions of commonly used functions. These helper functions are executed in an OO manner by passing routine names and parameters to a class called *runHelper* which then executes the appropriate helper commands. This creates a point of interception so that no code must change in the primary GUI elements if future helper functions must be modified or relocated.

The *INIs*, *load*, and *plot* directories are scratch locations for temporary files that must be used during the GUI's operation. When the user chooses to execute a simulation state, the *INIs* folder is populated with TASE-format config.ini files, which are then sequentially simulated in the TASE environment. GUI projects are composed of numerous MAT files encapsulated in a single ZIP file. These MAT files are extracted to the *load* directory for the currently active project.

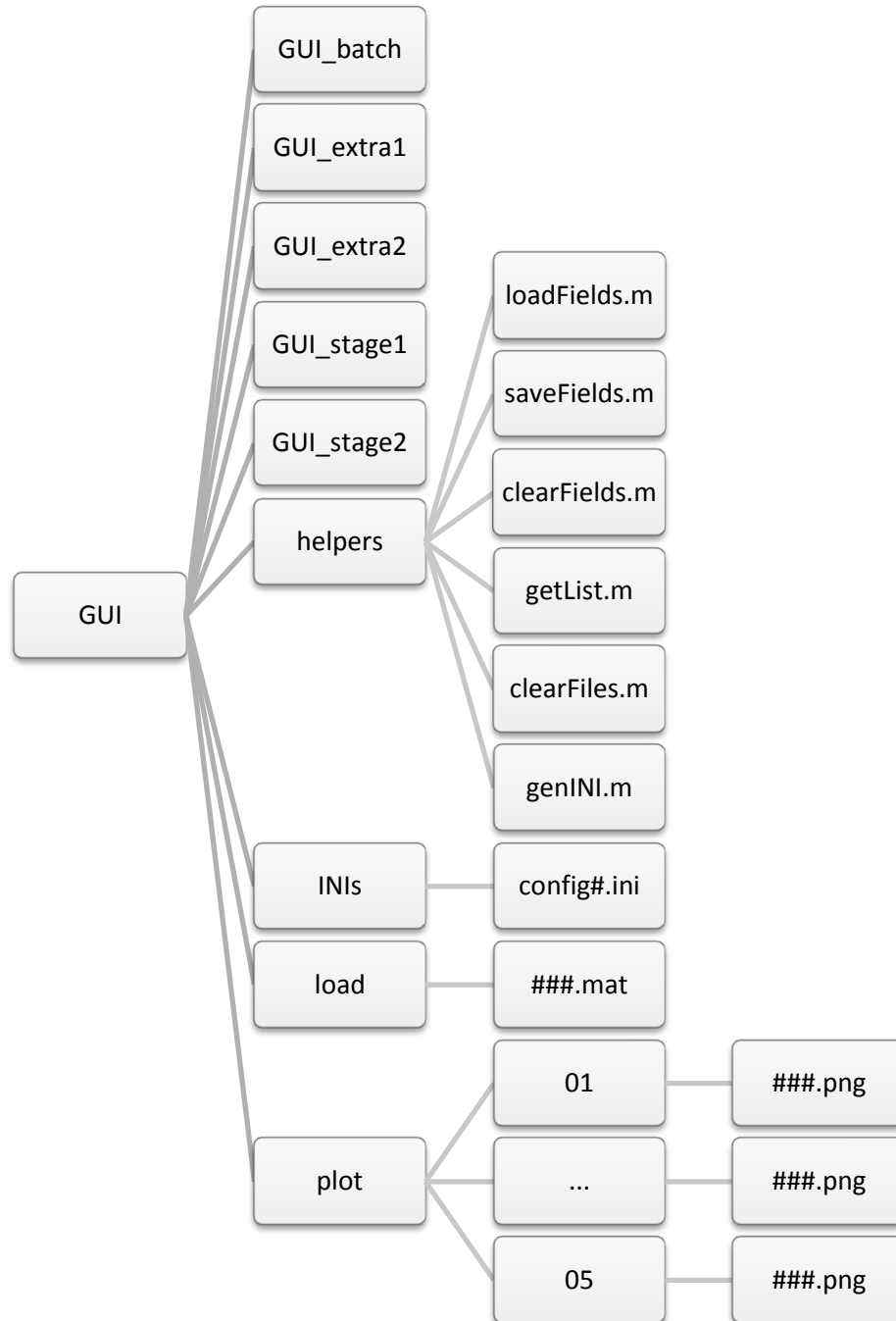


FIGURE 35 – TASE GUI ARCHITECTURE

The *plot* directory is created upon entering Exploration Mode of the GUI. Each of the five subdirectories corresponds to the first five test simulations that were executed with TASE. These subdirectories contain 461x350 pixel rasterizations of the EPS files created by TASE simulations. They are sequentially named according to the following equation:

$$F_{name} = [(a - 1) \times size(b)] + b$$

for swept variables a and b. This formula may be extrapolated in future versions of the tool to allow for sweep arrays greater than two variables.

CONFIGURATION MODE

Figure 36 depicts the GUI upon first launch (in *Configuration Mode*). A dynamically generated list of available simulations is displayed in the upper-left. All environmental parameters are shown as editable text-fields in the main part of the window. The lower left buttons perform environment verification tests for the TASE dependencies: Perl, Matlab, Spectre, and LaTeX. These verifications are done by executing sample files with each of the four dependencies and comparing the output to a known-good output. If the verification succeeds, these buttons turn green to alert the user that is now acceptable to click the *Run* button in the lower-right corner.

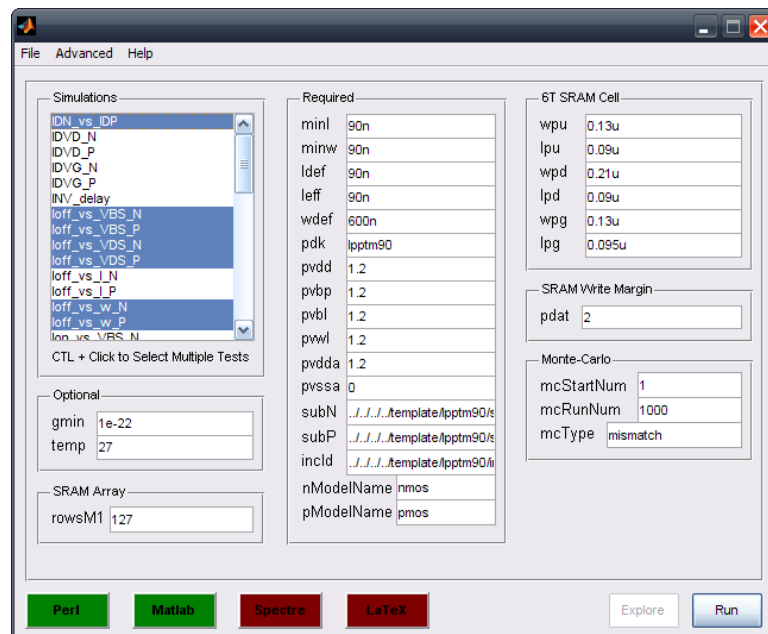


FIGURE 36 – CONFIGURATION MODE SCREENSHOT

The file menu provides five options:

- *New Project* – clears all input data and removes all files from the *load* directory.
- *Clear All Fields* – clears all input data for the current configuration set, but leaves all others intact.
- *Save Project* – compiles all MAT files in the *load* directory into a single zip file and prompts the user as to where to save the output.
- *Load Project* – deletes all files in the *load* directory, extracts the contents of the select zip file to this directory, and loads the first MAT file.

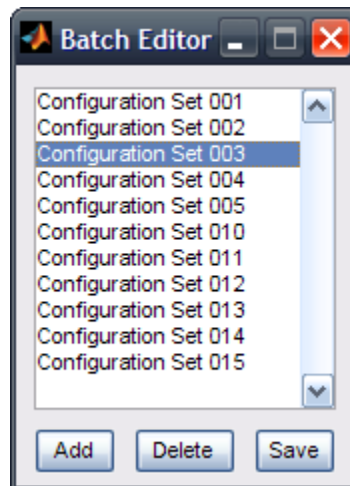


FIGURE 37 – BATCH EDITOR SCREENSHOT

From the *Advanced* menu, the user is able to select from three special tools. The first is the *Batch Editor* (Figure 37). This allows the user to create multiple configuration sets in a single project. Adding a new set copies the currently active set to a new configuration. In this way the user could create, for example, a single GUI project that has configuration sets for PTM 32, 45, 65, and 90 nm. When the user clicks 'Run' from the main window, the GUI creates a config.ini file for each configuration set and executes them all through TASE.

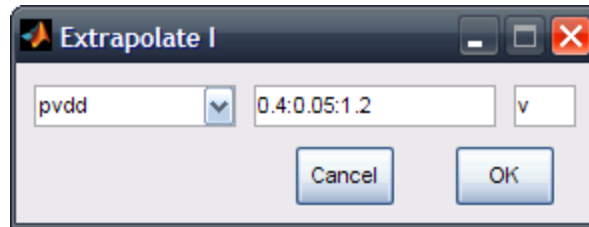


FIGURE 38 – EXTRAPOLATE I SCREENSHOT

The second tool available from the *Advanced* menu is *Extrapolate I* (Figure 38). This allows the user to select a parameter from the dropdown menu and enter a range of values in the first textbox. The parameter suffix (if necessary) is entered in the second textbox. When the user clicks *OK*, a new configuration set is created (based on the currently active set) for each of the swept values. This allows a simple way to generate large projects (up to 999 configuration sets) which sweep full ranges of variables.

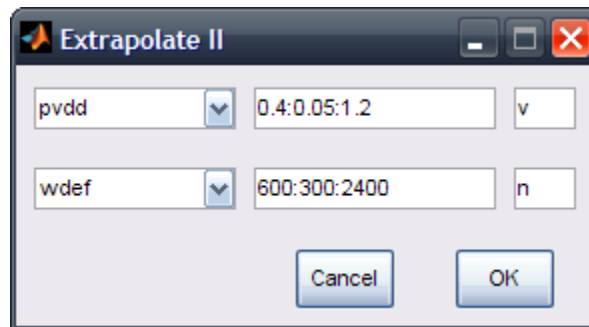


FIGURE 39 – EXTRAPOLATE II SCREENSHOT

The final advanced tool is *Extrapolate II* (Figure 39). This is identical to *Extrapolate I* except that it allows sweeping of two variables. First, variable one is held static for the entirety of the variable two's sweep. Then variable one is incremented and the process repeats. This allows quick, simple multivariable sweeps that generate full two-dimensional interdependency matrices.

Once the user has created all of the configuration sets he wishes to simulate, he clicks the *Run* button from the main window which generates and executes all config.ini files. After this process is complete, the *Explore* button becomes enabled and the user may switch to *Exploration Mode*.

EXPLORATION MODE

Exploration Mode (Figure 40) provides the user a simple way to traverse large sets of simulations and visualize variable interdependence. The first five simulation tests are displayed as thumbnails across the bottom pane. Clicking any of these thumbnails changes it to the primary graph. The left pane depicts up to four swept variables. When the user moves the slider bars associated with a given variable, the primary graph auto-updates to illustrate the change. In this way, the user can easily see the effect a given variable has upon a given simulation test.

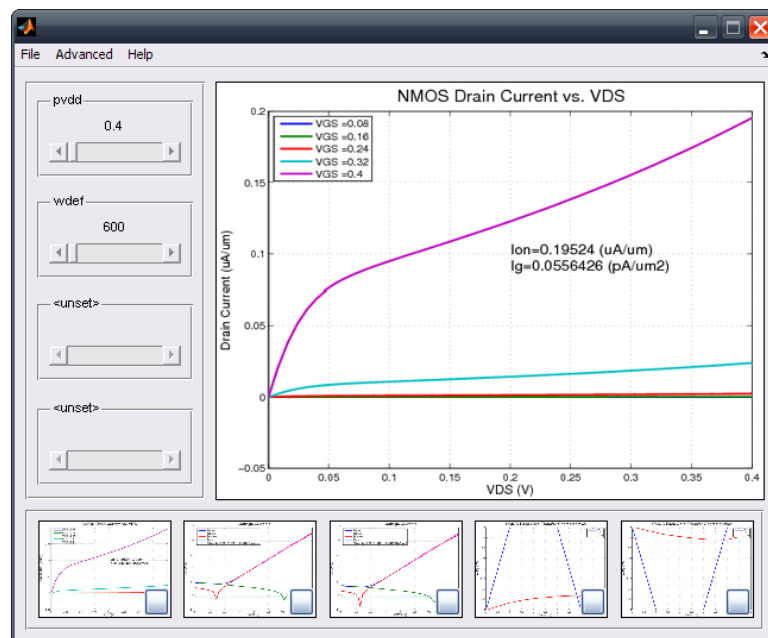


FIGURE 40 – EXPLORATION MODE SCREENSHOT

Just as is the case with *Configuration Mode*, the user can save and load *Exploration Mode* state. This packages / unpackages the contents of the *plots* subdirectory which includes all graph files and a single MAT file (which holds state variables).

Ultimately, the TASE GUI is designed to provide a straightforward method for the user to manage and generate large sets of configuration data. It provides a simple simulation viewer that is able to display the results of many TASE simulations. TASE itself allows for rapid simulation of designs across arbitrary process technology nodes. Thusly, this tool (TASE + GUI) accomplishes the final goal of this document by assisting in analyzing and designing technology resistant circuits, both FPGA and otherwise.

CONCLUSIONS

FPGAs are powerful devices that are able to implement arbitrary logic at the cost of transistor utilization. They can provide a cost-effective alternative to custom ASICs while decreasing time-to-market. For highly parallelized applications, they can outperform general purpose CPUs while still retaining the ability to dynamically update functionality in the field. Additionally, they are a highly lucrative multi-billion dollar industry and are fast permeating product fields. They do, however, face a number of challenges. Three of these challenges have been outlined and addressed in this document.

This document first described standard practice FPGA architecture, which is requisite to the explanation of the three contributions of this work. FPGAs are complex devices and a large-scale overview of their architecture is essential to working with any given aspect of their design.

This work first extended the standard FPGA architecture and presented a sub-threshold FPGA that was capable of a 7.5x decrease in energy at the cost of a 10% decrease in delay. This

reduction in energy could allow FPGAs to permeate energy constrained markets otherwise inaccessible. This sub-Vt FPGA was simulated and then implemented in a commercial 90 nm technology.

The process of physically implementing the sub-Vt FPGA revealed the need for an FPGA array generation CAD tool. This tool was described both architecturally and practically as it was used to generate FPGA arrays that were sent out for fabrication. It was designed for extensibility and ease of use, and it proved itself during tapeout when an underestimation required the FPGA array to change days before the deadline.

Finally, the need to analyze not just FPGA designs, but circuit designs in general, in a cross-technology manner was acknowledged. A technology agnostic simulation environment was discussed which allows test circuits to be analyzed at arbitrary technology nodes. This tool was designed to simplify the creation of future-proof designs and allow for the rapid porting of designs between nodes.

There is much yet to be done in the FPGA community, but the three contributions presented in this document hold potential, and have already showed varying degrees of success in aiding FPGA designers in their continued development of these unique devices.

ADDENDUM

- Unless stated otherwise via inline text, all work contained herein has been completed by Kyle Ringgenberg.
- In all schematics: signals “A”, “B”, “C”, and “D” are inputs; “S” is select; “Q” is output; “n” is an internal node; and “N”, “S”, “E”, & “W” are I/O associated with cardinal directions.
- The FPGA team has been led by Benton Calhoun and consists of Joseph Ryan and Kyle Ringgenberg with Eyad Lababidi.
- The original TASE tool was developed by Kyle Ringgenberg with Robert McNish. Current TASE development is being done by Satya Nalam.

GLOSSARY

- **AGT** Array Generation Tool
- **ALU** Arithmetic Logic Unit
- **ASIC** Application Specific Integrated Circuit
- **BLE** Basic Logic Element
- **CAD** Computer Aided Design
- **CLB** Configurable Logic Block
- **CPU** Central Processing Unit
- **DRC** Design Rule Check
- **DSP** Digital Signal Processing
- **EPS** Encapsulated Post Script
- **FPGA** Field Programmable Gate Array
- **GUI** Graphical User Interface
- **LUT** Lookup Table
- **LVS** Layout Versus Schematic
- **OO** Object Oriented
- **PAL** Programmable Array Logic
- **PCIe** Peripheral Component Interconnect Express
- **PDK** Process Development Kit
- **PROM** Programmable Read-Only Memory
- **SRAM** Static Random Access Memory
- **TASE** Technology Agnostic Simulation Environment

ACKNOWLEDGEMENTS

It has been a very difficult two years working toward this goal. I could not have achieved any of the work presented in this document without the support and constant encouragement of innumerable individuals. To quote Bernard of Chartres, "We are like dwarfs standing upon the shoulders of giants, and so able to see more and see farther than the ancients." Apart from this blanket statement, I need to thank a few individuals by name.

My family, you shaped me into who I am today, and I am eternally grateful for your love and unwavering support. Each time I've stumbled, you're the ones that taught me "do not be discouraged, for the LORD your God will be with you wherever you go".

Jamie Sammis, I would be a very different person today if you weren't there to always provide encouragement. Thank you for being such a caring listener and an amazing source of comfort when I'm in the midst of trying times.

Benton Calhoun, thank you for giving me the opportunity to work in this challenging field and continuing to push me to be a more critical thinker and a better engineer.

Joseph Ryan, thank you for teaming with me on the FPGA project. Much of what I know about layout, simulation, and the research process I learned from working with you.

Steve Jocke, thank you for always being willing to bounce ideas around and generally being a source of support in the office.

Finally, thanks to Satya Nalam, Randy Mann, Jiajing Wang, and Stuart Wooters for all the ways you've impacted my work here. Whether you are aware of it or not, you've each affected my experience in a profound way.

BIBLIOGRAPHY

1. **Birkner, John and Coli, Vincent.** *PAL Programmable Array Logic Handbook*. s.l. : Monolithic Memories, Inc, 1981. pp. 98-105.
2. **Xilinx.** Company History. *Xilinx*. [Online] <http://www.xilinx.com/company/history.htm>.
3. **Maxfield, Clive.** Xilinx unveil revolutionary 65nm FPGA architecture: the Virtex-5 family. *Programmable Logic DesignLine*. 2006.
4. **Xilinx.** *Virtex-6 Product Table*. 2009.
5. **Mohl, S and Sandberg, G.** *A Novel processor Architecture of FPGA Supercomputing*. s.l. : Xcell Journal, 2006.
6. **Altera.** *Stratix III FPGAs vs. Xilinx Virtex-5 Devices: Architecture and Performance Comparison*. s.l. : White Paper, 2007.
7. **Underwood, Keith.** *FPGAs vs. CPUs: Trends in Peak Floating-Point Performance*. s.l. : FPGA, 2004.
8. **Klein, Matt.** *Power Consumption at 40 and 45 nm*. s.l. : Xilinx, 2009.
9. **Betz, V, Rose, J and Marquardt, A.** *Architecture and CAD for Deep-Submicron FPGAs*. 1999.
10. **Chin, S.Y.L., Lee, C.S.P. and Wilton, S.J.E.** Power Implications of Implementing Logic Using FPGA Embedded Memory Arrays. *Field Programmable Logic and Applications*. 2006.
11. **Betz, Vaughn and Rose, Jonathan.** *Cluster-Based Logic Blocks for FPGAs: Area-Efficiency vs. Input Sharing and Size*. s.l. : CICC, 1997.
12. **Rabaey, J, Chandrakasan, A and Nikolic, B.** *Digital Integrated Circuits*. 2003.
13. **Carlson, I., et al.** *A high density, low leakage, 5T SRAM for embedded caches*. s.l. : Solid-State Circuits Conference, 2004.
14. **Kim, C.H., et al.** *A forward body-biased low-leakage SRAM cache: device, circuit and architecture considerations*. s.l. : Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, 2005.
15. **Nalam, S, et al.** *A Technology-Agnostic Simulation Environment (TASE) for Iterative Custom IC Design Across Processes*. s.l. : ICCD, Pending Acceptance 2009.