AFRL-RI-RS-TR-2012-091

A SECURE AND RELIABLE HIGH-PERFORMANCE FIELD PROGRAMMABLE GATE ARRAY FOR INFORMATION PROCESSING

CORNELL UNIVERSITY

MARCH 2012

FINALTECHNICAL REPORT

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■**UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

/s/	/s/
THOMAS RENZ	PAUL ANTONIK, Technical Advisor
Work Unit Manager	Computing & Communications Division
	Information Directorate

REPORT DOCUMENTATION PAGE	*Form Approved* **OMB No. 0704-0188**

1. REPORT DATE *(DD-MM-YYYY)* MAR 2012	**2. REPORT TYPE** Final Technical Report	**3. DATES COVERED** *(From - To)* DEC 2008 – DEC 2011

4. TITLE AND SUBTITLE **A SECURE AND RELIABLE HIGH-PERFORMANCE FIELD PROGRAMMABLE GATE ARRAY (FPGA) FOR INFORMATION PROCESSING**	**5a. CONTRACT NUMBER** FA8750-09-2-0010
	5b. GRANT NUMBER N/A
	5c. PROGRAM ELEMENT NUMBER 61102F
6. AUTHOR(S) Rajit Manohar	**5d. PROJECT NUMBER** CORC
	5e. TASK NUMBER CC
	5f. WORK UNIT NUMBER 09
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Cornell University, 330 Rhodes Hall, Ithaca, NY 14853	**8. PERFORMING ORGANIZATION REPORT NUMBER** N/A
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/RITB, 525 Brooks Road, Rome NY 13441-4505	**10. SPONSOR/MONITOR'S ACRONYM(S)** AFRL/RI
	11. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-RI-RS-TR-2012-091

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. PA# 88ABW-2012-1124
Date Cleared: 6 MARCH 2012

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Cognitive systems have requirements that are not met by existing commercially available architectures such as multi-core microprocessors or reconfigurable logic. In collaboration with Air Force Research Laboratory (AFRL), this project had the goal of creating a Field Programmable Gate Array architecture that is both high-performance and provides mechanisms to enforce secure operation. A new reconfigurable fabric was created with a number of unique features that enables secure bit-stream operation based on a hardware root of trust. A chip was created in collaboration with AFRL that included an AFRL design and the new asynchronous fabric, and the chip was submitted for fabrication through the Trusted Foundry Access Program.

15. SUBJECT TERMS
Field Programmable Gate Array, Asynchronous FPGA, Clustered Logic, Hybrid Processor, Secure Design, Dynamic Reconfiguration

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** THOMAS E. RENZ
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UU	27	**19b. TELEPHONE NUMBER (*Include area code*)** N/A

TABLE OF CONTENTS

LIST OF FIGURES ii

1. SUMMARY 1

2. INTRODUCTION 2

3. METHODS, ASSUMPTIONS AND PROCEDURES 3

3.1. Asynchronous FPGA Concepts 3
3.2. Asynchronous FPGA Software 6
3.3. Clustered Logic Block 7
3.4. Secure Operation with Meta Bits 10

4. RESULTS AND DISCUSSION 11

4.1. Dynamic Reconfiguration Ability 11
4.2. Secure Configuration 12
4.3. Physical Implementation 16
4.4. Software Flow and Bitstream Generation 17

5. CONCLUSIONS 19

6. REFERENCES 20

7. List of Acronyms 22

LIST OF FIGURES

Figure 1. Island-style FPGA Architecture. 3

Figure 2. Building Blocks for Dataflow Computation... 4

Figure 3. Dataflow translation of a straight-line program. 6

Figure 4. Translation of conditional execution into a dataflow graph. 7

Figure 5. Original AFPGA Logic Block. 8

Figure 6. Logical configuration of the core of the new AFPGA logic block. 8

Figure 7. Internal connectivity provided by the AFPGA logic block. 9

Figure 8. An example of supported routing re-configuration in the AFPGA. 12

Figure 9. Physical layout of an arrayable tile of the new AFPGA. 16

Figure 10. Overview of the software tool flow. 17

1. SUMMARY

Previous research on asynchronous FPGA architectures at Cornell resulted in the development of a new high performance reconfigurable fabric. This funded effort, along with a sister effort also funded by AFRL, transitioned the new technology to an AFRL project that combined general-purpose computing and reconfigurable logic for cognitive information processing applications. In a joint development effort, a large 65 nm chip was created with AFRL contributing portions of the design and Cornell contributing a high-performance asynchronous FPGA. The AFPGA was designed to ensure that the configuration is trustworthy even if the design was compromised. A chip to demonstrate the new architecture was fabricated through the Trusted Foundry Access Program, and is currently undergoing testing at AFRL. A more robust software flow has been created that can handle a large set of designs than previously possible.

2. INTRODUCTION

There are two major computation platforms that have survived the test of time when it comes to a combination of efficiency and programmability: the microprocessor, and the Field Programmable Gate Array (FPGA). Microprocessors are designed to implement a von Neumann abstraction of computation, where the computation is specified using a sequence of instructions that specify how the state of the system is to be modified. High-level languages are translated into these instruction sequences using a compiler, hiding the complexity of the detailed instructions from a user of such a system. FPGAs are designed to implement hardware described using combinational logic and state holding elements. The hardware itself is described using a high-level hardware description language, and logic synthesis tools are used to translate the description into the appropriate configuration information for the FPGA. Both these approaches translate a static, functional description of a computation into a static, physical implementation that implements the computation.

Conventional computer architectures such as uniprocessor, multicore, or massively parallel processors have a number of limitations when it comes to the performance requirements of DoD missions. These limitations include high power consumption during floating-point computations, high latency of global reduction operations causing performance degradation of parallel simulations, and the lack of security in the embedded systems context. In the context of massively parallel computing, fault-tolerance and reliability also have become significant issues that are not currently addressed in conventional architectures.

FPGAs have the potential to address some of the requirements of DoD missions. FPGAs are massively parallel, and hence can provide a high-throughput compute platform. FPGAs can also be customized for mission-specific tasks, acting as hardware accelerators for general-purpose computing. However, FPGAs used for this purpose are susceptible to malicious code, especially when FPGAs are integrated with general-purpose microprocessors. The FPGA configuration memory, in particular, is especially sensitive.

This report contains a summary of the work that was conducted in creating a platform that contains support for security and fault isolation for FPGAs that are integrated with conventional processors. The platform consists of a hybrid between a microprocessor and FPGA, with additional enhancements to the FPGA for secure operation and fault isolation.

3. METHODS, ASSUMPTIONS AND PROCEDURES

The Cornell Asynchronous Very Large Scale Integration (VLSI) and Architecture group previously developed a high-performance FPGA fabric for general-purpose computing. Compared to state-of-the-art commercial FPGAs from industry, the performance of the fabric was three times higher—a significant improvement. Compared to the best previously developed asynchronous FPGAs, the Cornell FPGA was almost twenty times faster in terms of application throughput [2,3,4]. This dramatic performance increase makes the fabric ideally suited to be integrated into a system containing a high-performance microprocessor.

3.1. Asynchronous FPGA Concepts

In terms of the major building blocks, the asynchronous FPGA (AFPGA) architecture looks like a traditional synchronous island-style FPGA such as a Xilinx Virtex [5]. The FPGA contains a configurable logic block (LB), a configurable interconnect (SB), and connection boxes (CB) that are used to connect the interconnect to the logic blocks. Figure 1 shows a high-level view of a generic FPGA architecture.

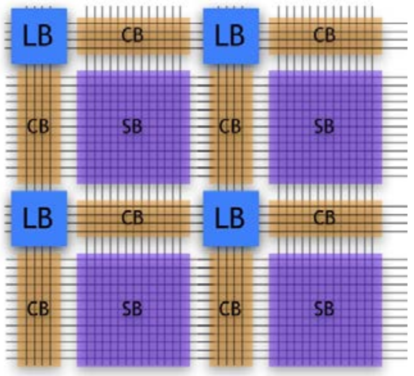

Figure 1. Island-style FPGA Architecture.

The major differentiating feature of the AFPGA versus a conventional FPGA architecture is the underlying computation model used to implement the configurable fabric. Instead of thinking of computation in terms of gates and registers, the AFPGA implements a computation specified by a dataflow graph [6]. In the dataflow graph model, computation is described by operations on data values or "tokens" flowing through the graph. Tokens correspond to valid data items being processed by elements of the dataflow graph. Nodes in the dataflow graph include function blocks that can perform computation, as well as routing elements for sending tokens to the appropriate destinations. Token arrival at a dataflow node can be thought of as an "event" that triggers activity in the AFPGA.

Dataflow computations can be implemented in a variety of ways. The AFPGA uses a set of basic building blocks for dataflow computations based on over ten years of experience designing both high-performance asynchronous microprocessors as well as low power asynchronous microprocessors. In designing these complex systems, it was found that there were only a few circuit topologies that led to efficient implementations. These are summarized below, and described briefly. The key building blocks are shown in Figure 2, and their functionality is summarized as follows:

- **Function**. The function block has N inputs and one output. This is the basic logic computation element. It receives a data token from each of its inputs, computes a function of the received input data, and produces the value as an output token.
- **Source**. A source produces a stream of constant tokens on its output.
- **Sink**. A sink consumes any tokens it may receive on its input.
- **Copy**. A copy is used to implement the equivalent of signal fanout. It replicates every input token it receives on all of its outputs.
- **Initial**. An initial block begins by producing a token on its output, and then after that it simply copies any input token it receives to its output.
- **Merge**. The merge block is a conditional block. It receives a data token from its control input (shown as a horizontal arrow above). The value of this data token is used to select an input port. The input data on the selected input port (vertical arrows) will be sent to the output. No other input tokens are consumed.
- **Split**. The split block is the dual of a merge. It receives a data token from its control input (shown as a horizontal arrow above). The value of this data token is used to select an output port. The input data value (vertical arrow) will be sent to the selected output port.

Arbitrary computations can be constructed from these basic building blocks [6,7]. The logic block for the FPGA contains all the necessary components to implement any dataflow computation. In particular, it contains programmable implementations of all the seven dataflow elements shown above.

As an example, the function element is implemented using a programmable four-input lookup table (LUT). Such a dataflow LUT waits for valid data tokens to arrive on all of its inputs, and then produces a new data token on its output based on the truth-table configuration of the LUT. Other support logic such as fast carry-chains and multiplier

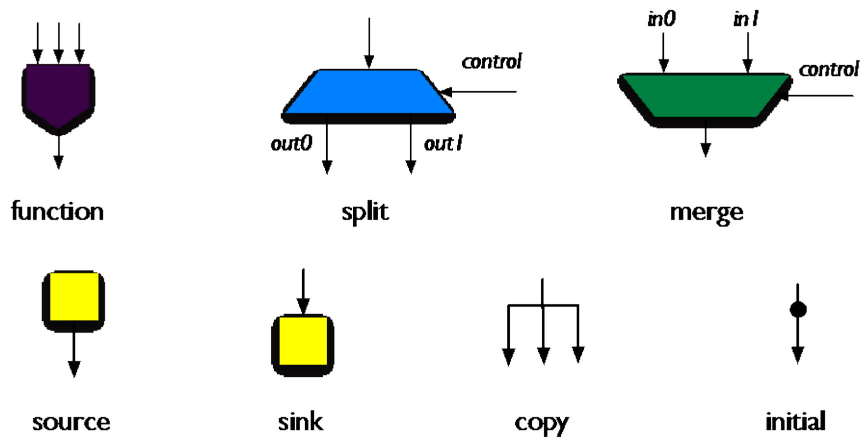

Figure 2. Building Blocks for Dataflow Computation.

support operate in a similar fashion.

The key performance amplifier in the AFPGA is its flexible routing network. A conventional FPGA has over 70% of its delay in the routing network [8]. Since the AFPGA operates using a dataflow model, pipeline stages corresponding to queues can be introduced into the routing network without impacting the correctness of the computation being performed by the AFPGA! This means that designs can benefit from pipelining without the additional cost required from electronic design automation (EDA) tools to support interconnect pipelining. In the first AFPGA implementation, pipelined stages were introduced in the switch boxes in the AFPGA interconnect [2,3].

The asynchronous nature of the fabric and the deeply pipelined implementation implies that only local paths limit AFPGA performance. For instance, even though the interconnect is configurable, it is impossible to create a path that contains a large number of switches as in a conventional FPGA because all such paths are partitioned by the presence of pipelining in the switch boxes. This is a dramatic difference from a conventional FPGA, where long paths through the routing network that have multiple switches and buffers are usually on the critical path [8]. The net effect is that the critical path limiting the peak frequency of the AFPGA corresponds to the lookup table access rather than the routing network.

The nature of the pipelined interconnect makes the entire AFPGA highly modular. In particular, because communication between components on the AFPGA uses the dataflow model, the delay of the communication link is not part of the interface specification. This enables a highly modular approach to the design of the AFPGA, where sub-blocks can be pre-placed without significantly impacting performance. Indeed, if data flow between one sub-block and another is unidirectional (as in a computation pipeline), there is no loss in throughput by using a modular approach to synthesis and place-and-route.

The impact of aggressive pipelining on the overall performance of the AFPGA is significant. In a 0.18 μm feature size, the measured peak performance of the AFPGA architecture was 674 MHz. For reference, the baseline Xilinx architecture in a similar feature size performs at 240 MHz [4]. More important, first pass synthesis results for a variety of benchmarks demonstrate robust performance. For example, a synthesized Finite Impulse Response (FIR) filter core would exhibit a performance of 75% of the peak performance of the AFPGA.

The only real source of performance loss in the AFPGA arises due to data-dependent loops in the computation graph. To illustrate this, consider a loop in a synchronous circuit that has a logic depth of d seconds, and contains k registers. The design cannot operate at a throughput that exceeds k/d, simply because there aren't a sufficient number of registers on the loop to exceed that performance. This is a fundamental or "algorithmic" limit of the design. The AFPGA is constrained by this limit, and we have found that in practice the AFPGA can achieve a performance that is close to 80% of the algorithmic limit or higher.

3.2. Asynchronous FPGA Software

In addition to hardware, an FPGA platform requires significant software development. In particular, design automation tools that take computations described in some language, map them to the FPGA architecture, and then generate the configuration bits required for the FPGA must be provided for an FPGA to be usable. This task is broken down into three major steps.

Logic Synthesis. The first step in the software flow is logic synthesis. In this step, arbitrary computations are specified in a language that resembles a concurrent version of the C programming language. This description is converted into a dataflow implementation using a new compiler representation called "static token form." Wide data path operations are broken down into bit-level operations, and various standard logic optimizations are performed on the dataflow graph.

Logic Packing. In this step, the dataflow elements are clustered to match the logic structure in the programmable logic blocks in the asynchronous fabric. This improves the logic utilization of the overall design.

Place and Route. For placement and routing, an open-source place-and-route tool known as "`vpr`" (for versatile place and route) developed by the University of Toronto can be used. The output of this process is converted into a configuration file to match the interconnect topology of the AFPGA architecture.

The key new step in logic synthesis is illustrated below, by describing a simple computation and how it is implemented using a dataflow graph. As a first step, consider straight-line programs that do not have any conditional execution. The code below shows a simple computation where two different functions are computed.

```
        a:=RECV(A); b:=RECV(B);                            x:=f      b);
         SEND(x,X);                                       x:=g  , b);   ND(x, Y);
```

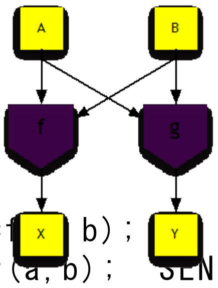

Figure 3. Dataflow translation of a straight-line program.

The inputs are received on ports `A` and `B`, and two values `f(a,b)` and `g(a,b)` are transmitted on output ports `X` and `Y` respectively. The static token form representation was used as a systematic way to translate the computation described in a high-level language into a dataflow graph. This representation matches the requirements for hardware implementation of dataflow graphs, and the use of this representation provides a simple mechanism for mapping designs to dataflow elements. This approach has been automated, and the net effect is the creation of the dataflow graph shown in Figure 3 [9].

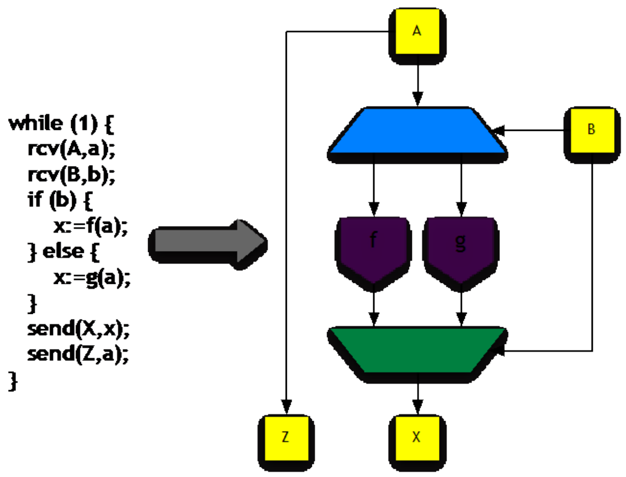

Figure 4. Translation of conditional execution into a dataflow graph.

While in this particular case, the dataflow graph may be "obvious", it should be stressed that the static token form approach is a *systematic* and *automated* method to generate this graph from the computation.

The translation is more complicated when we have conditional execution. Figure 4 shows an example of a program with an if-statement and its corresponding dataflow graph. In this case, the value of `a` is used either to compute `f(a)` or `g(a)` depending on the value of another variable `b`. To implement this functionality, special split and merge dataflow elements are introduced. These elements are used to route the data to the appropriate function, and then to collect the result and transmit it on the primary output of the block. The dataflow building blocks in Figure 2 are sufficiently expressive so that any program can be implemented using those blocks.

The Defense and Advanced Research Projects Agency (DARPA) initially supported the work described above as part of the Architectures for Cognitive Information Processing (ACIP) program [10].

3.3. Clustered Logic Block

Based on prior AFPGA design experience, a number of changes were made to the architecture of the AFPGA logic block. These changes were accompanied by corresponding changes to the software tool flow to support the new logic block architecture.

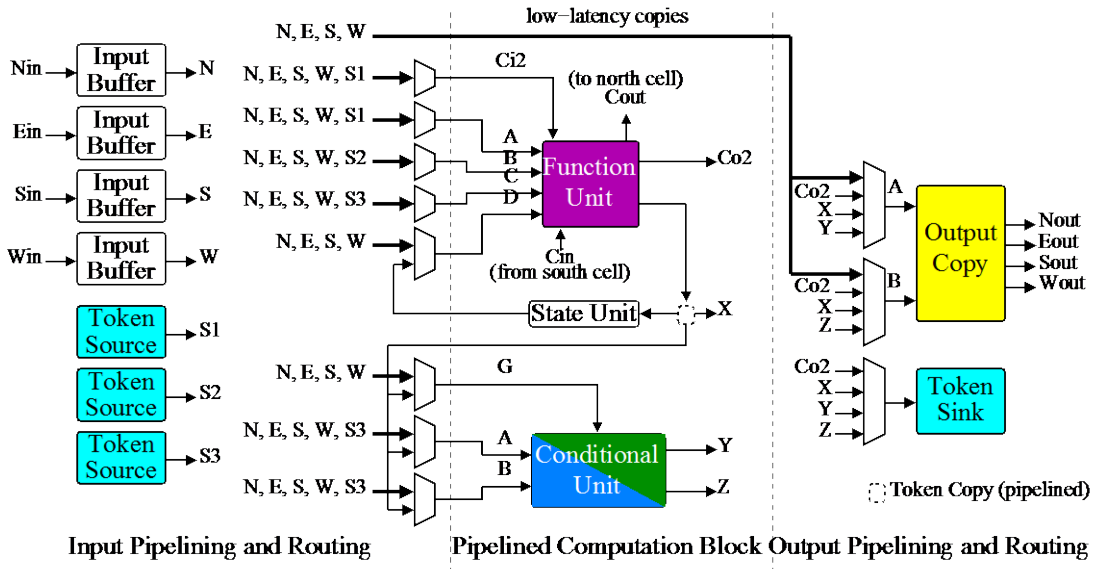

Figure 5. Original AFPGA Logic Block.

The original AFPGA resembled the logic block architecture of a Xilinx Virtex series FPGA [5]. The core computation block was a four-input lookup table, which has previously been shown to be a good compromise between performance and area-efficiency. The logic block also contained hardware support for fast ripple carry adders via both additional logic and a dedicated routing path for fast carry chains that bypassed the general-purpose routing network. In addition, multiplier support was provided through a programmable AND gate. Figure 5 shows the detailed block diagram of the original AFPGA architecture [3].

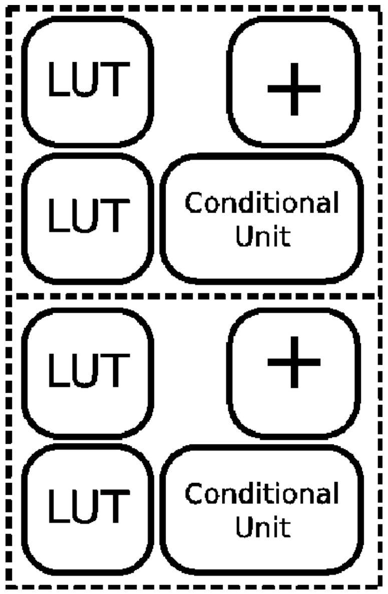

Figure 6. Logical configuration of the core of the new AFPGA logic block.

There were a number of limitations of this architecture that were remedied in the new AFPGA design. First, the logic block architecture in Figure 5 contains a single four-input lookup table. While this was suitable for a first-generation AFPGA, modern reconfigurable fabrics have clustered logic blocks containing multiple lookup tables per logic block. This choice results in significantly reduced routing requirements, as larger sections of logic can be mapped to a single logic block without resorting to the global routing network. The result of this analysis was that our new AFPGA architecture contains four four-input lookup tables per logic block. A second major change was made in the design of the function unit. In the original AFPGA (as in conventional synchronous FPGAs from Xilinx/Altera) the lookup table that is part of the function unit is re-used to implement a ripple-carry adder by the introduction of some support logic (exclusive OR gates). Instead of this, our new AFPGA design uses dedicated hardware adder structures that share inputs with the lookup table. The logic block can either be configured to use the lookup table, or configured as an adder. The benefit of this approach was that carry chains in our architecture are twice as fast, because the adder block can be designed to perform two bit additions in a single stage rather than just a one-bit addition. Figure 6 provides a logical view of the new AFPGA logic block core. Each block can be thought of as two halves, each containing two four-input lookup tables, a two-bit adder, and a conditional unit containing the split and merge dataflow elements.

The internal connectivity supported by the logic block is significantly enhanced compared to the original AFPGA architecture. Inputs arriving from the switch box are connected to the input connection box. This connection box provides simple point-to-point connectivity that allows inputs to be connected to a small number of dedicated input channels for the overall logic. This is followed by an input cross-bar with copying support, that converts the small number of inputs from the switch box into a set of inputs for each half of the AFPGA logic core (Figure 6). Copying support in this cross-bar allows a single input from the routing fabric to be copied to multiple input pins of the logic core. After passing through the logic core, the output of the logic block can also be copied to multiple destinations by the output copying block, and this is followed by the output connection box that provides connectivity back to the global routing network. Figure 7 is a block diagram that describes the overall logic block with the local routing network support. Another interesting feature of the new AFPGA architecture is the generality of feedback connections that are available. Outputs from the logic core can be fed back into the input cross-bar, enabling general connectivity within the logic core itself. Finally, the logic block contains a number of buffers that are used to decouple different parts of the design and increase throughput through pipelining.

The enhancements detailed in this section were supported by the Air Force Research Laboratory (AFRL), in a joint project where the FPGA fabric was integrated with an AFRL design using the Trusted Foundry flow in 65 nm.

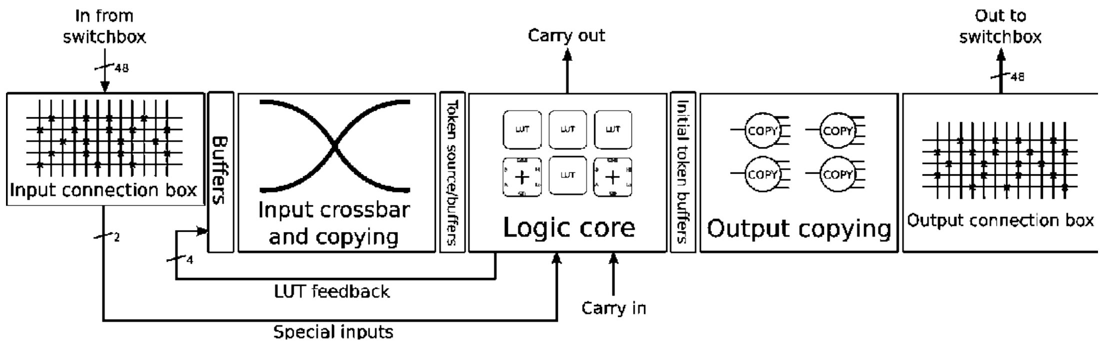

Figure 7. Internal connectivity provided by the AFPGA logic block.

3.4. Secure Operation with Meta Bits

The AFPGA was designed with dynamic re-configurability as a requirement for two major reasons: (i) FPGAs have significant area overhead due to their flexibility [11]. Even the largest commercially available FPGAs are too small to accommodate sophisticated software applications. It is important to be able to time-multiplex the FPGA area for different functions based on evolving application requirements. (ii) Dynamic re-wiring seems to be an important property of cognitive architectures. It was important for the AFPGA to be able to support this functionality.

There are a number of technical difficulties that arise when creating an AFPGA architecture whose configuration can be dynamically changed. These arise due to two kinds of dynamic configuration changes: (i) Using available real-estate for new computations; (ii) Modifying existing real-estate based on new information.

Dynamic reconfiguration creates a security challenge, because an application may want permission to modify the FPGA configuration memory while the FPGA is in use by a different application. If one of the applications is compromised, then the entire system can be compromised indirectly through the FPGA configuration memory.

The method used to protect configuration bits was to introduce a set of *meta-configuration* bits throughout the AFPGA fabric. These meta-configuration bits can be used to write-protect parts of the AFPGA configuration memory. The meta-configuration bits are designed so that they control access to configuration information for logically coherent sections of the AFPGA fabric. For example, one bit controls access to all the configuration information required for LUTs in a logic block since it is expected that these LUTs would be grouped together for purposes of reconfiguration. Meta-configuration bits can also be used to partially modify an existing configuration. Once again, the meta-bits are used to select the section of the design to be updated. The bits serve as a selection mask at a granularity that is appropriate for AFPGA reconfiguration.

The introduction of meta configuration bits enables both dynamic configuration as well as secure operation, and the overall area impact of this additional information was found to be approximately 1%.

4. RESULTS AND DISCUSSION

The physical design of the AFPGA layout was completed in IBM's 65 nm LPe process (10LPe) available through the Trusted Foundry Access Program. The AFPGA was designed as an intellectual property (IP) block with the appropriate design files so that it could be integrated with AFRL's design in the same way as third-party IP blocks such as phased locked loops, high-speed serial interfaces, memories, etc. are integrated in commercial design flows. AFRL submitted the chip for fabrication through the Trusted Foundry Access Program, and has received chips back from the foundry. At the end of this project, the chips were undergoing testing at AFRL.

4.1. Dynamic Reconfiguration Ability

The new AFPGA routing network contains significant support for dynamic reconfiguration. Apart from the per-logic block configuration support described above, an important consideration in the design of the AFPGA was configuration support for routing. Meta-configuration bits were designed to allow a small group of tracks to be re-configured without impacting existing configurations of adjacent tracks.

Figure 8 provides an illustration of the reconfiguration support provided in the new AFPGA. The figure on the left corresponds to a configuration where the blocks highlighted in green correspond to some computation mapped to the AFPGA, and the blue path is configured to connect a source to a destination. The new configuration preserves the configuration in green, but changes the source-destination path to the one shown on the right. The meta-configuration bits are organized so as to support this level of dynamic reconfiguration. Observe that the new path occupied by the blue routing track utilizes resources that are physically adjacent to the green configuration. This is made possible by the meta-configuration bits, as only the configuration corresponding to the blue path is modified.

The meta configuration bits are valuable as a global mask which identifies the parts of the AFPGA fabric impacted by reconfiguration. The key difference between meta configuration bits and other possible mechanisms (e.g. using bit-addressable configuration memory) is that the meta bits group configuration information into logical units that are meaningful from the standpoint of dynamic reconfiguration. Meta-configuration bits are also grouped into words of a size that corresponds to the normal granularity of the configuration memory organization.

By writing the appropriate meta-configuration bits, the appropriate subsets of ordinary configuration bits are selected for modification. The update pictorially depicted in Figure 8 would proceed as follows:

- Step 1. All meta-configuration bits would be cleared, thereby preventing any configuration modifications.
- Step 2. The meta-configuration bits corresponding to the original and new blue route are set, thereby enabling modifications to the configuration bits controlling all the gates impacted by the route change.

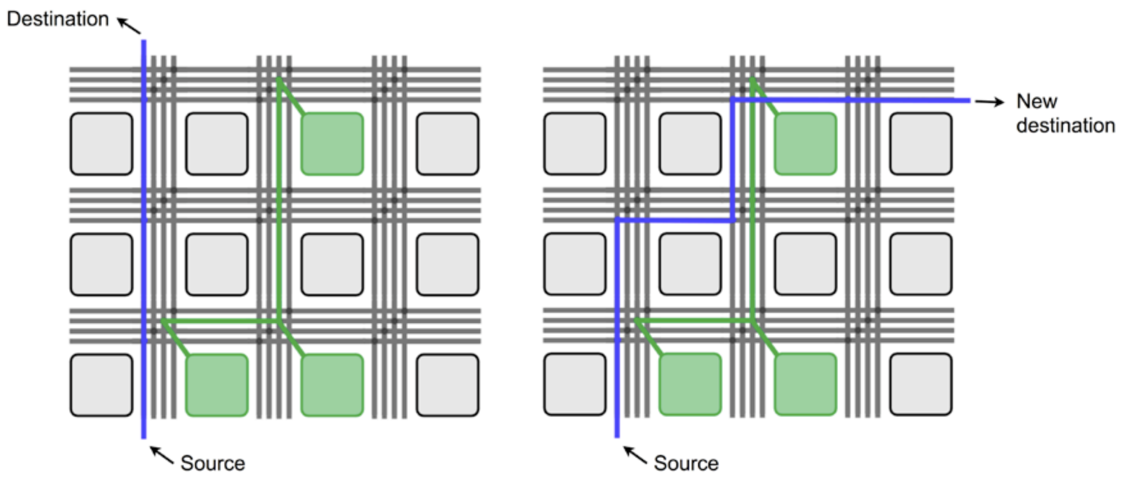

Figure 8. An example of supported routing re-configuration in the AFPGA.

- Step 3. The configuration bits are updated. While full configuration word writes are attempted, the masking due to meta-configuration bit values ensures that only the appropriate subset of configuration information is modified.
- Step 4. The route is initialized by following the reset protocol for the asynchronous logic. Once again, meta configuration bits are used to ensure that only the new route is initialized. The green configuration is not impacted by the reset protocol.

4.2. Secure Configuration

The clustered AFPGA architecture uses a modified static memory cell as the core configuration storage element to minimize the amount of area required for configuration bits. This is similar to the approach taken in commercial FPGA implementations [8].

There are a number of unique challenges when designing a configuration memory cell. It may seem that a standard static random access memory (SRAM) can serve the function of configuration memory. However, there are a number of requirements for a configuration memory storage element that differ from those of a standard SRAM bit cell.

A traditional SRAM bit cell has to operate correctly only in an array environment. Reading and writing the bit is performed through shared lines, and hence the capacitive loading on various access ports for a standard SRAM is well-defined and deterministic. Apart from edge effects (that sometimes require dummy SRAM cells to surround the core memory array), bit-cells have an identical—or at least very predictable—environment.

FPGA configuration memory is used to control gate voltages on pass transistors. This control voltage is required *at all times*—which means that all the bits of configuration memory need to be accessed continuously, and in parallel. To simplify the design of the AFPGA configuration memory, isolation buffers were introduced between the stored configuration bit and the pass transistor gates so that a dense configuration memory bit cell would have a deterministic electrical environment. Wiring considerations imply that area-optimized FPGA layout places configuration memory bits near the locations where they are used. Therefore it is impractical to create large, dense configuration memory arrays—since each bit needs at the very least a wire that connects it to the appropriate pass transistor gate. The AFPGA configuration memory was organized in very small arrays that were placed near the locations where their values were needed. This observation also means that the configuration memory bits cannot use standard high-density foundry SRAM cells, because foundry cells require that the bits be organized in a dense array configuration for them to achieve high yield.

The AFPGA uses approximately 1 Kbit of configuration memory per four input lookup table. Of this total storage requirement, only sixteen bits are required for the lookup table configuration; the rest correspond to the average cost of configuration information for routing. Even a modest AFPGA size (e.g. 1024 LUTs) requires a configuration memory of the order of 1 Mbit! However, the bits of this memory are not organized in a densely packed configuration as discussed earlier, but look more like a sparse array that logically behaves as a normal SRAM. The result is that the AFPGA configuration memory requires much longer bit lines and word lines compared to a traditional SRAM that has the same total storage.

To handle the cross-talk and coupling issues created by the long word and bit lines, the AFPGA configuration memory is partitioned into smaller banks, with local drivers placed periodically throughout the AFPGA array. Critical parameters that govern the read and write margins in the AFPGA configuration memory are therefore isolated to a small physical region sized to maximize robustness for a reasonable area budget.

The AFPGA configuration memory is located at a fixed location in the address space. Issuing store operations to a certain set of physical addresses can modify the configuration bits and destroy the functionality implemented in the AFPGA. Hence, a malicious application can corrupt the system by the appropriate sequence of memory operations.

It is not sufficient to simply write-protect the entire AFPGA configuration memory. Legitimate uses of the configuration memory include modifying it to implement new functionality for the application (dynamic reconfiguration), and so access to the configuration memory address space must be permitted.

For an efficient AFPGA implementation, it is important that AFPGA configuration bits are physically close to the locations where the bits are used; otherwise, local wiring overheads would be significant. Hence, configuration bits in a typical AFPGA are "scattered" throughout the physical design. This means that the address space of AFPGA configuration bits is complex—a single word may contain configuration bits that correspond to completely unrelated logical resources in the AFPGA simply because the layout of those resources happens to be physically adjacent due to efficiency constraints.

Hence, protecting logically connected AFPGA bits is challenging in a conventional architecture because it would require protection granularity that is at the per-bit level—effectively doubling the wiring required for the configuration memory bit-lines. Supporting bit-level protection would require write-protect information that is comparable to the size of the AFPGA configuration memory—a very large overhead.

Instead, our meta-configuration bits can be used to solve this problem efficiently. Meta configuration bits are also organized in the same manner as ordinary configuration bits. The key difference is that meta-configuration bits are organized so that they can be easily identified by their address—all meta-configuration bits belong to words that are located at addresses with a well-defined signature, and can be easily identified by a simple comparison operation. These meta-configuration bits can be protected with conventional means—writes to these bits are treated as privileged operations, and these writes can be easily distinguished from writes to ordinary configuration bits.

Reads and writes to ordinary configuration bits were *gated* by the appropriate meta-configuration bits. Each AFPGA tile contains meta-configuration bits that control ordinary configuration bits within the tile. The overall area overhead of this scheme was found to be ≈1%. Reads and writes to configuration bits that are not enabled (because their meta-configuration bits are clear) are ignored; this effectively write-protects any configuration bit that is protected through the meta-configuration information.

Since meta-configuration bit addresses are easily identified externally, memory address protection mechanisms can be used to protect not just meta-configuration bits but ordinary configuration bits too. To write-protect any AFPGA configuration, the following steps can be followed:

1. Set the meta-configuration bits so that reads and writes to the appropriate ordinary configuration bits are disabled.
2. Place the system in a mode that disables any modifications to the meta-configuration bits.

At this point, any writes to AFPGA configuration memory will not modify the protected configuration information. Write protection of the meta-configuration bits is supported by a single input signal to the AFPGA array. Reading the configuration memory can also be disabled by another input pin to the AFPGA. By disabling reads and writes through a secure path, the current configuration of the AFPGA can be protected from an external attacker.

Protection of these meta-configuration bits can be performed in multiple ways. In the context of a microprocessor with full operating system support and memory protection, the meta-configuration bits can be placed under operating system control—assuming the operating system is trustworthy. Another approach is to use the one taken by the Nexus operating system where the hardware root of trust for the operating system is a Trusted Platform Module [16]. In the context of an embedded system without hardware support for security and memory protection, critical configuration information can be protected via a secure hardware access port. For example, the JTAG port can be used to set up the protection information and the software on the microprocessor can be denied physical access to this port. While this approach limits in-field flexibility, it does provide a strong security guarantee.

Under certain circumstances it may be useful to deny access to operations that read the configuration memory state. These write-protect bits can also be used to override the values read out of the configuration memory.

4.3. Physical Implementation

The physical design of the AFPGA was undertaken using a combination of commercially available layout editors as well as a custom-designed tool to automate some of the physical design process.

Figure 9 shows a screenshot of the physical layout of the AFPGA. The tile consists of two sections that are approximately equal in size. The left hand side of the tile corresponds to the internals of the logic block, including the lookup tables, arithmetic

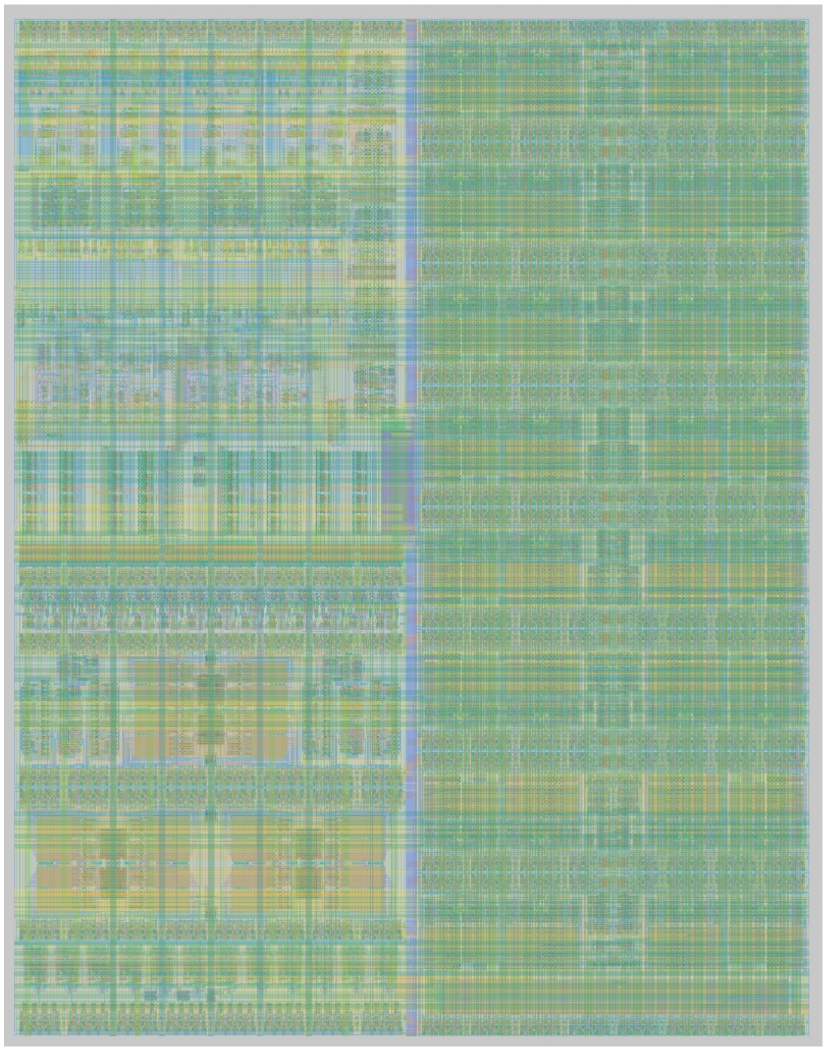

Figure 9. Physical layout of an arrayable tile of the new AFPGA.

functions, and internal tile routing. The right hand side of the tile consists of the pipelined interconnection network for intra-tile routing. The area of the tile used by logic functions is ≈10%, with configuration memory using ≈30% of the area and the rest being used by the routing network.

There were a number of design issues that required detailed analysis of analog effects by circuit simulations that modeled parasitic resistances and capacitances. While large simulations that contain accurate models are beyond the scope of existing commercial tools, simplified but conservative models were used to ensure robust functionality of the final AFPGA. Many changes to the underlying circuits were made based on these simulations, including partitioned wires for the configuration memory and appropriate spacing and shielding for the interconnect lines in the AFPGA.

Extensive SPICE simulations predict that the peak performance of this AFPGA is in the range of 800 MHz. This frequency was chosen based on the performance of the rest of the system (below 150 MHz) that will include the AFPGA fabric. The total number of AFPGA tiles in the system can be changed easily by simply creating an array of the appropriate size.

The edge of the tile contains interface circuits as well as programming support for the configuration memory. This interface exports a synchronous interface to the rest of the system for easy integration with commercial physical design flows. With help from Prof. Stine at Oklahoma State, a flow was created that enabled the entire physical AFPGA block to resemble a synchronous IP block that was placed and routed with Cadence Encounter.

In collaboration with AFRL, the final physical implementation was submitted for fabrication through the Trusted Foundry Access Program to IBM's foundry services.

4.4. Software Flow and Bitstream Generation

A new software flow was developed using commercially available tools, and this flow is

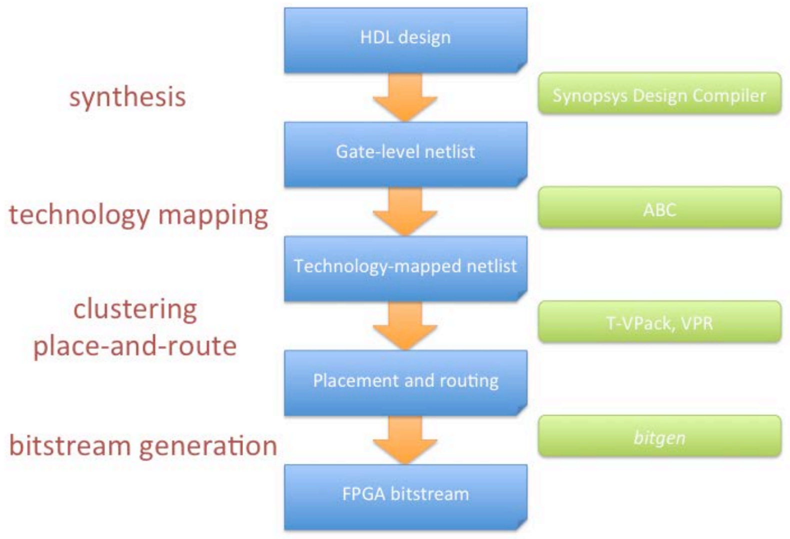

Figure 10. Overview of the software tool flow.

currently being evaluated over a wide range of benchmarks obtained from the opencores web site. The steps in the tool flow are summarized in Figure 10.

Synopsys' Design Compiler was used as the front-end of the flow. This synthesis tool is widely recognized as the best logic synthesis tool in the industry and has been adopted by the ASIC industry. As a result, this tool has broad support for commercial hardware description languages such as Verilog and VHDL. A challenge with these hardware description languages is that only some parts of the language are amenable to logic synthesis. The benefit of using Design Compiler is that a very wide range of inputs can be converted into a synthesized design, since the tool has undergone decades of development.

The output of Design Compiler is a gate level description of the design. Since Design Compiler is an ASIC synthesis tool, it maps the design into a set of gates that are part of typical ASIC standard cell libraries. The AFPGA contains a different set of "gates"—namely four input lookup tables, arithmetic blocks, and multiplexor circuits. The next step in the flow maps the gate level description generated by Design Compiler into a logically equivalent gate level description that only contains building blocks that are present in the AFPGA. This conversion is performed using a state-of-the-art open-source synthesis tool—called "abc"—developed by Prof. Brayton's group at UC Berkeley.

The "abc" tool has certain limitations in its input and output format, as well as limitations on the types of designs it can handle. To work around these limitations, a Python script was developed that modifies the input to "abc" as well as its output so that it is suitable for FPGA implementation.

The technology mapped design is then converted into a placed and routed design using the University of Toronto's open source FPGA placement and routing package called "vpr" (for Versatile Place and Route). A custom bit-stream generation tool was developed to create the appropriate configuration bits for the AFPGA based on the placed and routed design. More work is currently underway on the bit-stream generation process.

Initial benchmarks that complete the entire flow successfully from the opencores suite include an Advanced Encryption Standard, (AES) core, as well as interface circuits including Inter-Integrated Circuit, Peripheral Component Interconnect, and Serial Peripheral Interface. Additional benchmark testing is underway, and currently the major limitation preventing more designs from passing the complete flow is the presence of multiple clock domains. We are in the process of creating benchmarks that are more suitable for the set of applications that we plan to map to the AFPGA in the context of a processor-FPGA hybrid architecture.

5. CONCLUSIONS

This project developed a mechanism for protection of configuration memory for an on-chip FPGA integrated with an embedded processor. The results show that the area overhead of protection can be limited to a small amount (approximately 1%), which also implies that the protection scheme has negligible impact on the performance and power of the FPGA. The project also showed that conventional design entry languages can be used with high-performance asynchronous FPGAs, thereby reducing the re-design effort needed to utilize the new asynchronous FPGA technology.

6. REFERENCES

[1] J. Martinez, C. Gomes, R. Linderman, “Research Gap Analysis Report”, *Workshop on Research Directions in Architectures and Systems for Cognitive Information Processing*, July 2005.

[2] John Teifel and Rajit Manohar, “Programmable Asynchronous Pipeline Arrays”, *Proceedings of the 13th International Conference on Field Programmable Logic and Applications*, Lisbon, Portugal, September 2003.

[3] John Teifel and Rajit Manohar, “Highly Pipelined Asynchronous FPGAs”, *12th ACM International Symposium on Field-Programmable Gate Arrays*, Monterey, CA, February 2004.

[4] David Fang, John Teifel, and Rajit Manohar, “A High-Performance Asynchronous FPGA: Test Results”, *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2005.

[5] Xilinx, VirtexTM 2.5V Field Programmable Gate Arrays, Xilinx Data Sheet, 2002.

[6] J. B. Dennis, “The Evolution of ’Static’ Data-flow Architecture”, *Advanced Topics in Data-Flow Computing*, J.-L. Gaudiot and L. Bic, ed., Prentice-Hall, 1991.

[7] Song Peng, David Fang, John Teifel, and Rajit Manohar, “Automated Synthesis for Asynchronous FPGAs”, *13th ACM International Symposium on Field Programmable Gate Arrays*, February 2005.

[8] I. Kuon, R. Tessier, J. Rose, “FPGA Architectures: Survey and Challenges”, *Foundations and Trends in Electronic Design Automation*, **Vol. 2**, No. 2, pp. 135-253, 2007.

[9] John Teifel and Rajit Manohar, “Static Tokens: Using Dataflow to Automate Concurrent Pipeline Synthesis”, *Proceedings of the 10th International Symposium on Asynchronous Circuits and Systems*, April 2004.

[10] Jon Russo, Mohammed Amduka, Keith Pendersen, Richard Lethin, Jonathan Springer, Rajit Manohar, Rami Melhem, “Enabling Cognitive Architectures for UAV Mission Planning”, *Proceedings of the High Performance Embedded Computing Workshop*, September 2006.

[11] I. Kuon and J. Rose, “Measuring the Gap between FPGAs and ASICs”, *IEEE Transactions on Computer-Aided Design of Intergrated Circuits and Systems*, **Vol. 26**, No. 2, February 2007.

[12] Cadence Design Systems, “Clock Domain Crossing: Closing the Loop on Clock Domain Functional Implementation”, Cadence Technical paper, 2004.

[13] R. Ginosar, “Fourteen Ways To Fool Your Synchronizer”, *Proceedings of the IEEE International Symposium on Asynchronous Circuits and Systems,* May 2003.

[14] R. Manohar, “Systems And Methods For Performing Automated Conversion Of Representations of Synchronous Circuit Designs To And From Representations Of Asynchronous Circuit Designs”, US Patent 7,610,567, October 2009.

[15] B. Hu, M. Marek-Sadowska, “Multilevel Fixed-Point-Addition-Based VLSI Placement”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **Vol. 24**, No. 8, 2005.

[16] Emin Gün Sirer, Willem de Bruijn, Patrick Reynolds, Alan Shieh, Kevin Walsh, Dan Williams, and Fred B. Schneider, “Logical Attestation: An Authorization Architecture For Trustworthy Computing**”,** *Proceedings of the Symposium on Operating Systems Principles*, Cascais, Portugal, October 2011.

7. List of Acronyms

Acronym	Expanded Form
ACT	Asynchronous Circuit Tools
AFPGA	Asynchronous Field Programmable Gate Array
AFRL	Air Force Research Laboratory
ASIC	Application Specific Integrated Circuit
CB	Connection box
DARPA	Defense Advanced Research Projects Agency
EDA	Electronic Design Automation
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
I/O	Input/Output
IP	Intellectual Property
LB	Logic block
LUT	Lookup Table
SB	Switch box
SRAM	Static Random Access Memory
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VLSI	Very Large Scale Integration
VPR	Versatile Place and Route