Chapter 1

Introduction

Field-programmable gate arrays (FPGAs) are generic, programmable digital devices that can perform complex logical operations. FPGAs can replace thousands or millions of logic gates in multilevel structures. Their high density of logic gates and routing resources, and their fast reconfiguration speed give them the advantage of being extremely powerful for many applications. FPGAs are widely used because of their rich resources, configurable abilities and low development risk, making them increasingly popular.

Since FPGAs offer designers a way to access many millions of gates in a single device, powerful FPGA design tools with an efficient design methodology are necessary for dealing with the complexity of large FPGAs. Currently, most of the FPGA design tools [Men01][Syn03][Syn04] use the following design flow: first, they implement the design using Hardware Description Language (HDL); second, they simulate the behavior and the functionality of the design; finally, they synthesize and map the design in the vendor’s FPGA architecture [Xil00]. When analyzing the typical design flow of an Electronic Design Automation (EDA) tool, place-and-route is the most time-consuming and laborious procedure. It’s hard to find an optimum layout in a limit period of time. Similar to the bin-packing problem, placement is NP-complete [Ger98]. Growing gate
capacities in modern devices intensifies the complexity of design layout; thus, likely increases the computation time required in the place-and-route procedure.

As an added challenge, the contemporary design flow removes the design hierarchy and flattens the design netlist. When modifications are made and the design is reprocessed, the customary design flow re-places and reroutes the entire design from scratch no matter how small the change. Therefore, the FPGA design cycle is lengthened due to the time consumed during the iterative process. Although some methods [Nag98][Tsa88] have been applied to accelerate the processing time, and the iterative process might be acceptable when the FPGA gate sizes are small, it will become a problem as the gate sizes are increased exponentially.

There is a tradeoff between processing speed and layout quality. Simple constructive placement algorithms, such as direct placing and random placing, place the design fast but cannot guarantee the quality; iterative placement methodologies, such as simulated annealing and force-directed method, provide high quality layouts but the processing time is long. Million-gate FPGAs present the possibility of large and complicated designs that are generally composed of individually designed and tested modules. During module tests and prototype designs, the speed of an FPGA design tool is as important as its layout quality. Thus, a methodology that presents fast processing time and acceptable performance is practical and imperative for large FPGA designs.

The objective of this dissertation is to examine and demonstrate a new and efficient FPGA design methodology that can be used to shorten the FPGA design cycle, especially as the gate sizes increase to multi-millions. Core-based incremental placement algorithms are investigated to reduce the overall design processing time by distinguishing the changes between design iterations and reprocessing only the changed blocks without affecting the remaining part of the design. Different from other incremental placement algorithms [Cho96][Tog98][Chi00], the tool presented here provides the ability not only to handle the small modifications; it can also incrementally place a large design from
scratch at a significantly rapid rate. System management techniques, implemented as a background refinement process, are applied to ensure the robustness of the incremental design tool. Incremental approaches are, by their very nature, greedy techniques, but when combined with a background refinement process, local minima are avoided. An integrated incremental FPGA design environment is developed to demonstrate the placement algorithms and the garbage collection technique. Design applications with logical gate sizes varying from tens of thousands to approximately a million are built to evaluate the execution of the algorithms and the design tool. The tool presented places designs at the speed of 700,000 system gates per second tested on a 1-GHz PC with 1.5GB of RAM, and provides a user-interactive development and debugging environment for million-gate FPGA designs.

This dissertation offers the following contributions:

- Investigated incremental placement algorithms to improve the FPGA development cycle. The typical gate-array circuit design process requires the placement of components on a two-dimensional row-column based cell structure space, and then interconnecting the pins of these devices. Placement is a crucial yet difficult phase in the design layout. It is an NP-complete task [Sed90] and computationally expensive. Conventional placement algorithms, such as min-cut methods [Bre77] and affinity clustering methods [Kur65], are proven techniques, and typically succeed in completing a design layout from scratch. These placement algorithms unfortunately will make the FPGA design cycle unacceptably long when the chip size grows larger and larger. Although some placement algorithms achieve almost linear computation characteristics, they still require a significantly long computation time to complete a layout [Roy94][Kle91][Cho96]. For interactive iterative use, a new algorithm is needed that focuses on circuit changes. One of the accomplishments of this dissertation is the investigation and evaluation of incremental compilation-based placement algorithms to speedup the placement time. As a design evolves incrementally, and as components are added as part of the design process, this placement
algorithm can not only process the small modifications, but it can also place a large design from scratch.

- Developed and demonstrated a prototype of an incremental FPGA design tool that can shorten the FPGA design cycle for a million-gate device. Design tools play an important role in the FPGA design cycle; however, the traditional design flow faces great challenges as the FPGA gate sizes grow to multi-millions. For the traditional design flow, the long design cycle, smaller resource reuse, and inefficient compilation for engineering changes make it ill-equipped for multimillion-gate FPGA designs. As one of the accomplishments, this dissertation presents an infrastructure and a prototype of an incremental FPGA design tool that can be used to demonstrate the incremental placement algorithms developed in this work. This tool uses a Java-based integrated graphics design environment to simplify the FPGA design cycle, and to provide an object-oriented HDL design approach that allows Intellectual Property (IP) reuse and efficient teamwork design.

- Explored a garbage collection and background refinement mechanism to preserve design fidelity. Fast incremental placers are inherently greedy, and may lead to a globally inferior solution. Since the incremental placement algorithm proposed in this dissertation positions an element using the information of the currently placed design, the position of the element is best at the moment the element is added. This may not always produce a globally optimum solution. As more elements are added to the design, a garbage collection technique is necessary to manage the design to ensure the performance and the robustness of the application is preserved. Therefore, incorporating a garbage collection mechanism with the placement algorithm and the design tool development is another essential achievement of this dissertation.

- Developed large designs to evaluate the incremental placement algorithm and the
design tool. As another important accomplishment, this dissertation tested and evaluated the performance of the techniques presented in this work. Example designs with the gate sizes varying from tens of thousands to approximately a million have been implemented to assess and improve the incremental placement algorithm, the garbage collection mechanism and the design tools that have been investigated in this dissertation. The computation time, the speed of placement, as well as the performance of the incremental placement algorithm, have been measured, analyzed, and compared with the traditional placement algorithms to verify the speed-up of the incremental design techniques.

Chapter 2 examines the traditional FPGA design cycle and the conventional placement algorithms. Their features and shortcomings for the million-gate FPGA design are analyzed. The incremental compilation technique is investigated to demonstrate the possibility of improving the traditional FPGA design flow. The functionality of the JBits Application Program Interface (APIs) and JBits tools is also examined to explain their potential to shorten the FPGA design cycle.

Chapter 3 presents the implementation of the core-based incremental placement algorithms. Detailed processing flow and methods employed to fine-tune this flow are discussed. Guided placement methodology is investigated to find changed parts in a design and to take advantage of the optimized design from previous iterations. Cluster merge strategies are also implemented in this chapter to complete this core-based guided incremental placement algorithm.

An incremental FPGA integrated design environment is developed in Chapter 4. The program organizations, the data structures, and their implementations are described. Dynamic linking techniques are developed to allow the designer building their design using Java Language and compiling the design using the standard Java compiler. A simple design example is also presented to demonstrate the usage of the incremental design IDE.
Chapter 5 describes the garbage collection techniques employed in this dissertation. A core-based simulated annealing placement algorithm and its implementation as a background refiner of the incremental placement algorithms are discussed. The properties of the simulated annealing placer and its advantages as the background refinement thread are analyzed. When combined with the incremental placement algorithm, it is expected to help the incremental design tool developing performance and robustness.

Chapter 7 tests the algorithms developed in Chapters 3, 4, and 5 using designs generated in Chapter 6. The performances of the incremental placement algorithm, the guided placement methodology and the background refinement techniques are analyzed; the functionality of the incremental design IDE is evaluated as well. Finally, the goals of this dissertation are reexamined in Chapter 8. Feature directions are also discussed in the last chapter.
Chapter 2

Prior Work

This chapter examines the traditional FPGA design cycle from the contemporary FPGA design tools reported in the literature. The common features of the design cycle are analyzed and their shortcomings are evaluated for high-density FPGAs. Incremental compilation [Sun98], a compiler optimization technique, is examined from the literature to demonstrate the possibility of improving the traditional FPGA design flow. The functionality of both the JBits Application Program Interface (APIs) and JBits tools [Xil01] is investigated to explain their potential to shorten the FPGA design cycle.

2.1 FPGA Design Tools

This section reviews the current FPGA design tools, the placement algorithms, and the traditional FPGA design flow. The characteristics of the design flow are investigated and their limitations for million-gate FPGA designs are examined.

2.1.1 Current FPGA design tools and traditional design flow
Field Programmable Gate Arrays (FPGAs) were invented by Xilinx Inc. in 1984 [Xil98].
FPGAs provide a way for digital designers to access thousands or millions of gates in a single device and to program them as desired by the end user. To make efficient use of this powerful device and to deal with its complexity, many design tools have been developed and widely used in FPGA development. FPGA designers use electronic design automation (EDA) tools to simulate their design at the system level before mapping, placing and routing it onto the device vendor’s architecture. EDA companies including Synopsys, Synplicity, Mentor Graphics, Viewlogic, Exempler, OrCAD and Cadence provide FPGA design tools supported by device manufacturers, including Actel, Altera, Atmel, Cypress, Lattice, Lucent, Quicklogic, Triscend, and Xilinx. When reviewing the FPGA design tools used in the market, it is easy to find that their common design flow mimics the traditional flow for application specific integrated circuit (ASIC) design, which is to:

- Implement the design in hardware development language such as VHDL, Verilog, or JHDL.
- Simulate behaviors and functions of the design at the system level.
- Netlist the design if the functional simulation is satisfied.
- Map, place and route the netlisted design in the Vendor’s FPGA architecture.
- Verify the design and check the timing and functional constraints.

Figure 2.1 shows the traditional FPGA design flow. Following the design flow, if all requirements are met, the executable bitstream files are generated and the design is finally put on the chip. Generally, the implementation time ranges from several minutes to many hours to accomplish the whole process.

Compared with ASIC design, the FPGA design flow has significant advantages [Xil00]. One of the advantages is that the systems designed in an FPGA can be divided into sub-modules and tested individually. Design changes can be reprocessed in minutes or hours instead of months per cycle as in ASIC design. Although noticeable improvements have been made from the ASIC to the FPGA design flow, the current design flow still has problems when it faces the next generation of FPGA applications.
2.1.2 Review of placement algorithms

The typical gate array circuit design process requires placing a design in a two dimensional row-column based cell structure space, and interconnecting the pins of these devices. Generally, the goal is to complete the placement and the interconnection in the smallest possible area that satisfies sets of design, technology and performance constraints [Mic87]. Heuristic methods are used to generate a good layout, and they often divide the layout process into four phases: partitioning, placement, global routing and detailed routing [Cho96]. Placement is the most important phase because of its difficulty and its effects on routing performance [Sec98].

Since placement is an NP-complete problem, it is hard to find an optimum solution exactly in polynomial time [Don80]. The use of placement algorithms is necessary to find an exact solution in a limited period of time. Shahookar and Mazumder gave a
comprehensive review of the VLSI cell placement techniques in [Sha91]. They indicated that the goal of the placement algorithm is to establish a placement with the minimum possible cost. An acceptable placement should also be both physically possible and easily routed. There is no cell overlap and every module in a design is placed at a position inside the chip boundaries. Generally, cost of a placement is evaluated using the chip area or timing constraints. It is better to place a design in the smallest possible area and fit more modules in a given area, to reduce customer cost. Wire length, the total distance between connected models, should be minimized to balance delays among nets and speed up the operation of the chip. Finding a tradeoff between the chip area and the timing constraints is always the task most place-and-route researchers are working on. Algorithms that are timing-driven but lead to very poor chip area cannot produce a good design. Similarly, algorithms that achieve minimum chip area but do not meet the timing requirements are also of little interest

[Sha91] discussed five major algorithms for placement: simulated annealing, force-directed placement, min-cut placement, placement by numerical optimization, and evolution-based placement. The basic implementation and the improvements of each algorithm are explained and some examples are also provided. Mulpuri and Hauck analyzed the runtime and quality tradeoffs in FPGA placement and routing in [Mul01]. Twelve MCNC benchmark circuits were implemented in this paper to compare five placement algorithms: Fiduccia-Mattheyses, force-directed, scatter, simulated annealing and the Xilinx placer. A new tradeoff-oriented algorithm was developed to control the quality versus runtime tradeoff. According to the analysis in [Mul01], placement algorithms vary widely in their tradeoffs.

These placement algorithms can be divided into two major classes: constructive placement and iterative placement [Sha91] [Ger98]. Constructive placement places a design from scratch. Once the position of a module has been fixed, it is not changed any more. While iterative placement algorithms start from an initial configuration, they then repeatedly modify the design in the search of for cost reduction. Since constructive
placement algorithms do not modify the placement constantly, they are relatively faster than iterative placement, but generally lead to poor layout performance. On the other hand, iterative placement provides much better performance while the processing is much longer. Placement algorithms, such as scatter, numerical optimization techniques, partitioning algorithm and some force-directed algorithms are constructive algorithms, while algorithms including simulated annealing, the Xilinx placer, and some force-directed algorithms place designs iteratively.

There is a tradeoff between processing speed and layout quality. Simple constructive placement algorithms place the design fast but cannot guarantee the quality; iterative placement methodologies provide high quality layouts while the processing time is long. To ensure the quality of the performance, iterative placement is widely used in EDA CAD tools. Since the processing time is proportional to the number of gates involved in the placement, the larger the gate size, the longer the placement time. The speed of the iterative placement algorithms is acceptable when the gate counts are small and the designs are simple. As the gate counts increase dramatically, million-gate FPGAs present the possibility of large and complicated designs. To efficiently build such a design, it is generally decomposed into individually designed and tested modules. During module tests and prototype designs, the speed of an FPGA design tool is as important as its layout quality. Thus, a methodology that presents fast processing time and acceptable performance is practical and imperative for large FPGA designs.

2.1.3 Problems in traditional FPGA design flow

When analyzing the current FPGA design flow, place-and-route is the most time-consuming and laborious procedure. However, it is hard to find an optimum layout in a limit time period [Pre88], and even a simple bin-packing problem [Kuh90] is NP-complete. Contemporary FPGAs have densities that approach millions of gates and millions of internal pins in a single chip (Xilinx Virtex 300E chip has 1,124,022 system gates and over a million internal pins). Generally, when the design is large and the Configurable Logic Block (CLB) usage is above 50%, it may take many hours to
accomplish placement and routing, and there is no guarantee that the process will succeed for each run. For example, placing a circuit with approximately 3000 nets and 10000 pins takes more than two hours using the traditional min-cut method [Kle91]. Gate capacity is increasing exponentially and provides the possibility of bigger and more complex designs, but it also intensifies the complexity of placement and routing. Thus, the computation time consumed in the place-and-route procedure will be increased.

Once a bitstream is created, it is loaded on the chip and executed to verify the functionality. If some improvements and modifications are required, the entire design procedure has to be repeated from the HDL design. If the modifications become routine, the user will need to recompile and reprocess the design multiple times. In the current design cycle, the user’s design is netlisted after HDL modeling and functional simulation. During netlisting, the design hierarchy is removed and the whole design is flattened. Therefore, when modifications are made and the design is reprocessed, the customary design flow will not use any information from the previous design. Instead, it re-places and reroutes the entire design from scratch. However, most of the time the change in the active design is small. For example, when the designer changes only the size of a counter or adds/deletes an inverter gate, he or she would like to implement the change without affecting the placement, routing and timing in other parts of the design. Unfortunately, the current design flow cannot guarantee this. Although some methods have been applied to accelerate the processing time, they still need to go through the whole procedure and wait for minutes or hours to create a new bitstream no matter how small the change. Contemporary approaches are acceptable if the design is small, but they will emphasize problems when the gate sizes increase to multi-millions.

Generally, the computational complexity of a placement algorithm is of $O(n^{\alpha})$, where $n$ includes all the gates in the layout, and $\alpha$ is a number which is equal to or greater than 1 [Kuh90] [Cho96]. Suppose ten iterations are needed in a design development and $\alpha$ is equal to 1.5. In a small-size gate array that has 3000 gates, to complete a layout, the computational complexity is approximately $10^6$. In large FPGAs that have more than $10^6$
gates, the computational complexity will be at least $10^{10}$. It is clear that the reprocessing time for a million-gate device will be several orders of magnitude longer than that for a small device design. The more frequently the modification occurs, the longer the designer should wait. Obviously, this is not what the FPGA designers want to see.

Decreasing the computational complexity of the placement algorithm is one way to speed up the FPGA design cycle. Tsay and his colleagues presented a placement algorithm for sea-of-gates FPGAs [Tsa88]. In this work, they dealt with the optimal placement problem by solving a set of linear equations and provided an order of magnitude faster performance than the simulated annealing approach [Sec88]. GORDIAN is a placement algorithm that formulated the placement problem as a sequence of quadratic programming problems derived from the entire connectivity information of the circuits [Kle91]. The unique feature of this algorithm is that it maintains simultaneity over all optimization steps, thus obtaining global placement for all sub-modules at the same time and achieving linear computation time. Although these methods accomplish almost linear computational complexity, it still takes a significant amount of time to complete a layout. For example, placing a circuit with approximately 3000 nets and 10000 pins takes 30 minutes on a VAX 8650 machine (a 6-MIPs machine) using the method in [Tsa88], while it takes about 15 minutes on an Apollo DN4500 workstation (a 15-MIPs machine) using the technique in [Kle91]. Using GORDIAN to process a larger circuit with 13419 nets, it takes about 160 minutes on a DEC5000/200 workstation [Sun95].

Some tool designers have noticed this problem, and have been working to make products available for next generation FPGAs. Providing an efficient development cycle is one aspect some EDA companies are working on. Mentor Graphics’ FPGA Advantage sought to integrate the HDL design flow in the initial stage of the FPGA development cycle [Men01]. They tried to make the design cycle from HDL to silicon more efficient by providing an integrated design management environment that can handle all design data at a higher level of abstraction [Rac00]. The integrated HDL design flow may offer a comprehensive FPGA design environment powerful enough for million-gate FPGAs.
Atmel made an attempt similar to Mentor Graphics tool. In Atmel’s FPGA Design Package 5.0, HDLPlanner is used to help the designer create efficient VHDL/Verilog behavioral descriptions and optimized, deterministic layouts [Atm01]. The problem with the above is that no matter whether they are in an integrated HDL design environment or an efficient HDL layout generation, these tools still need to process all the gates in a device to complete a layout because they are not supported by a technique to process the engineering changes by involving only the changed parts of a design.

High-speed compilation can reduce the synthesis and the place-and-route time. It is another method to speed up the FPGA design cycle. Sanker and Rose [San99] focused on the placement phase of the compiling process and presented an ultra-fast placement algorithm for FPGAs. This algorithm combines the concepts of multiple-level, bottom up clustering and hierarchical simulated annealing; it can generate a placement for a hundred-thousand-gate circuit in ten seconds on a 300 MHz Sun UltraSPARC workstation. Nag and Rutenbar [Nag98] presented a set of new performance-driven simultaneous placement and routing techniques. These schemes showed significant improvements in timing and wireability in benchmarks when compared with the traditional place-and-route system used by Xilinx 4000 series FPGAs. To reduce the compilation time, one can also increase the CPU speed or add RAM to the PC or workstation. Even though these methods shorten the compilation time, they still have to compile and process the entire design whenever there is a change. As indicated in [Brz97], these methods do not reduce the number of elements involved in the process that are required to debug or improve design performance; they simply provide some efficient ways to reduce the time per pass. When the chip size grows to many millions of gates, the total processing time is still a huge number. It is necessary, therefore, to find other solutions to this problem.

One possible solution is to find the changes the designer has made between iterations, then re-synthesize, re-place and reroute the changed parts only and reuse the unchanged information. Incremental compilation strategy has the functionality to achieve this requirement.
2.2 Incremental Compilation

Incremental compilation is a compiler optimization intended to improve the software development cycle. It is used to search for the change between the current and the previous design, recompile only the change, and avoid affecting the remaining optimized portions. Because the recompilation time will be proportional to the changes in a design, incremental compilation, if used properly, will significantly reduce the compilation time if the changes are small.

This technique is broadly used in software engineering to improve software development cycles. Montana [Kar98], an open and extensible integrated programming environment provided by IBM and an infrastructure of one of IBM’s production compiler, Visual Age C++ 4.0 [Nac97], supports incremental compilation and linking. This system uses an automatic testing tool to test functions that have been changed since the last computation, which leads to better performance for the tool and the compiler. Venugopala and Srikant applied incremental compilation to an incremental basic block instruction scheduler [Ven98]. In this paper, algorithms for incremental construction of the dependency directed graph and incremental shortest and longest path were investigated and their performances were evaluated by implementing the system on an IBM RISC System/6000 processor. The testing results showed that the compiling time is reduced significantly by using the incremental compilation technique.

Traditional programming environments present program source code as files. These files may have dependencies on each other, so a file will need recompilation if it depends on a file that has changed. This can create a bottleneck in implementing incremental compilation. Appel and his colleague presented a separate compilation for standard modular language [App94]. A feature called the “visible compiler” was implemented and the incremental recompilation with type-safe linkage was incorporated to avoid recompilation with the dependent modules. Their system has been combined with the Incremental Recompilation Manager (IRM) [Lee93] from Carnegie-Mellon University
and has been applied to both educational and commercial uses. Cooper and Wise achieved incremental compilation through fine-grained builds [Coo97]. They presented a “build tool” that can process dependencies between source files, and can update the application with a minimum of recompilation. This tool has been implemented in a system called “Barbados” [Coo95] and is shown to be faster and more efficient in updating the application after small modifications.

Incremental techniques are also widely used in electronic design. They are a standard feature in all ASIC-synthesis and place-and-route tools. ASIC designers use a “divide and conquer” approach [Xil00] to break a chip into embedded cores that can be tested individually. Once a core has reached the desired performance, it is locked and remains unchanged during other design iterations. Vahid and Gajaski presented an incremental hardware estimation algorithm [Vah95] that is useful to determine hardware size during hardware/software functional partitioning. In this work, parameters used to estimate hardware size are rapidly computed by incrementally updating a data structure that represents a design model during functional partitioning, thus leading to fast hardware size estimation. Tessier applied incremental compilation for logic estimation [Tes99]. He described and analyzed a set of incremental compilation steps, including incremental design partitioning and incremental inter-FPGA routing for hard-wired and virtual wired multi-FPGA emulation systems. The experimental results proved that when integrated into the virtual wired system, incremental techniques can be successfully used to lead a valid implementation of modified designs by only re-placing and rerouting a small portion of FPGAs. VCS from ViewLogic System and Synopsys Inc. is an industry standard simulator for Verilog HDL. Sunder implemented incremental compilation in the VCS Environment [Sun98] to determine whether a design unit is changed, and whether it needs to be recompiled in both the single and multiple-user environment. The performance of the method proved the advantages of incremental compilation in minimizing the compilation time and increasing the simulation speed. This technique has been fine-tuned to provide better performance in Verilog HDL design.
Because incremental techniques provide the potential to reprocess only the modified fraction of a design, many researchers have tried to apply this technique to place-and-route to optimize designs and speed up processing time. Choy presented two incremental layout placement modification algorithms: Algorithm Using Template (AUT) and Algorithm by Value Propagation (AVP)[Cho96]. These algorithms found an available slot for an added logic element by selectively relocating a number of logic elements. Because these algorithms only replace elements in a neighborhood of changes, they are several orders of magnitude faster than conventional placement algorithms. Togawa described an incremental placement and global routing algorithm for FPGAs [Tog98]. This algorithm allows placing an added Look Up Table (LUT) in a position that may overlap with a pre-placed LUT, then moves the pre-placed LUTs to their adjacent available positions. Chieh presented a timing optimization methodology based on incremental placement and routing characterization [Chi00]. In his work, timing is evaluated using accurate parasitics from incremental placement during logic optimization, and routing effects during optimization are predicted using fast routing characterization. Thus, better timing optimization is achieved after the placement and routing.

Recently, EDA companies and FPGA vendors have also realized the importance of incremental compilation in the next generation of FPGAs, and have started to use this technique to improve their FPGA design tools. Cadence delivered the industry’s first tool to bring physically accurate timing to front-end synthesis [Cad99]. This tool achieved high timing accuracy by incrementally placing, routing, and timing in the core synthesis loop and provided near-exact timing correlation throughout. Xilinx and its alliance partner Synplify focused on reducing the synthesis time by keeping the design hierarchy and using guided place-and-route [Xil99]. They provided synthesis attributes to preserve the hierarchy of the design in the EDIF netlist, and applied effective strategies for partitioning the design and optimizing the hierarchy. Then, they employed guided place-and-route to handle minor incremental changes. As indicated by Xilinx and Synplify, the new features in this synthesis tool will significantly increase productivity. In Synopsys’
newly released Compiler II and FPGA Express version 3.4 [Syn01], a block level incremental synthesis technique (BLIS) [Ma00] was added to allow designers to modify a subset of a design and re-synthesize just the modified subset [Syn02]. This tool was reported to dramatically reduce the design cycle for multimillion-gate Xilinx Virtex devices.

According to the reports in the literature and the newly released FPGA design tools, we can clearly see that incremental compilation has been playing an increasingly important role in reducing the design cycles for multimillion-gate array.

2.3 Garbage Collection

Garbage collection, in the context of free memory management, was originally a software engineering issue. This technique is generally used in automatic memory and resource management by automatically reclaiming heap-allocated storage after its last use by a program [Jon96]. Memory management is a simple and easy task in small-scale computer programming, but it becomes essential as the complexity of software programming grows, especially in situations where memory allocation/de-allocation is not explicitly handled. Improper resource management could downgrade system performance and distract the concentration of software engineers from the real problems they are trying to solve. Programmer controlled (explicit) resource management provides methods for software engineers, especially object-oriented language programmers, to effectively control the complexity of their program, therefore increasing code efficiency and resource application.

Several object-oriented languages including Java, C++, and Smalltalk utilize garbage collection techniques for free memory management, and an enormous number of papers and books talk about this issue. Richard Jones and Rafael Lins reviewed the development of memory management, the classical and generational garbage collection algorithms, and its applications to the C/C++ language in their book [Jon96]. An age-based garbage collection is discussed in [Ste99]. This paper presented a new copying collection
algorithm, called older-first, to reduce the system cost and improve the performance of the garbage collector by postponing the consideration of the youngest objects. [Coo98] presented a highly effective partition selection policy for object database garbage collection to improve the performance of algorithms for automatic storage reclamation in an object database. Several policies were investigated in this paper to select which partition in the database should be collected. The Updated Pointer policy was shown to require less I/O to collect more “garbage” than others, and its performance was close to a locally optimal Oracle policy.

Because this dissertation focuses on a million-gate FPGA design problem, developing a new computer language and dealing with memory management issues is considered out of the scope of this work. Therefore, we need to assess whether it is necessary to discuss the garbage collection technique, and whether it relates to this dissertation work.

The answer is in the affirmative. Although garbage collection is widely used in software engineering area, this valuable concept can be extended to find applications in other areas. Integrating garbage collection techniques with incremental compilation is a great example. As discussed in Section 2.2, incremental compilation plays an important role in reducing the design cycle for multimillion-gate FPGAs. But incremental approaches are, by their very nature, a greedy technique. It is a local optimization method. It finds a locally desired choice in the hope that it may lead to a globally optimum solution. When this technique is employed to process a million-gate design incrementally, it is optimal at the moment it is added, while it may not be optimal when more and more elements are inserted. Those un-optimal placements are “garbage” during the design processing, and may lead to a globally inferior solution. Thus, a garbage collection technique is necessary to manage the design to ensure the performance and the robustness of the applications. This system management technique, if implemented as a background refinement process, will avoid the local minimal and offer the instant gratification that designers expect, while preserving the fidelity attained through batch-oriented programs.
Current incremental placement algorithms [Cho96][Tog98][Chi00] put the concentration on the functionality of the incremental techniques while neglecting its natural shortcoming. The processing cycle is reduced but the global system performance cannot be guaranteed. The performance and robustness of the incremental compilation technique will be enhanced when combined with garbage collection methodologies. If the garbage collector is running at the background thread, it will not compete with the CPU time required by the incremental compilation. Furthermore, it can provide good references to restore the design fidelity of the incremental compilation using spare CPU cycles. Both the design performance and the resource utilization are improved if a proper garbage collection technique is integrated with the design system. Therefore, the garbage collection is employed in this dissertation work. Its implementations and functionality will be discussed in the following chapters.

2.4 JBits and JBits tools
This section investigates a set of new tools, JBits API and its associated toolkit, which support simple and fast access to Xilinx 4000 and Virtex series family FPGAs.

2.4.1 JBits APIs
JBits is a set of Java classes that provide an Application Program Interface (API) into the Xilinx 4000 and Virtex series FPGA family bitstreams [Xil01]. JBits can read the bitstream files generated from either the Xilinx design tools or directly from the chips. Thus, it has the capability to read and dynamically modify the bitstreams. The traditional FPGA design flow generates executable bitstreams from synthesis, and it may take from minutes or hours to complete a design cycle. JBits presents another way for bitstream generation. It provides the possibility of directly accessing a bitstream file, modifying it and generating a new design in seconds. Although the original motivation of the JBits API was to support applications that require fast dynamic reconfigurations, it can also be used to construct static digital design circuits. Contrasted with the traditional FPGA design flow shown in Figure 2.1, the design flow for JBits is illustrated in Figure 2.2.
The JBits API has direct access to Look Up Tables (LUT) in Configurable Logic Blocks (CLB) and routing resources in Xilinx 4000 or Virtex FPGAs. The programming model used by JBits is a two-dimensional array of CLBs. Because JBits code is written in Java with associated fast compilation times, and the programming control is at the CLB level, bitstreams can be modified or generated very quickly. The detailed information about the JBits API can be found from [Xil01].

![Figure 2.2 Static JBits design flow [Xil01]](image)

### 2.4.2 RTPCores

As the design complexity of FPGAs increases, functional unit reuse is becoming an important consideration for large FPGA designs. The generation of the concept “core” solves this problem to some degree. Most FPGA vendors offer cores. For instance, Lucent Technologies’ Microelectronics FPGA group licenses cores, ranging in functions from a PCI bus interface, ATM and other networking cores to DSP and embedded microprocessor cores [Sul99]. JBits also provides parameterizable cores. In the JBits 2.5
version, not only are there ready-to-use Run Time Parameterizable cores (RTPCores), but also an abstract class that helps designers build their own reusable cores. Instantiating a core is very simple and easy in JBits. Creating a new core object is like creating any Java class constructor. The JBits function setVerOffset()/setHorOffset() is used to place the core in a specific location on the chip.

2.4.3 JRoute
JRoute is a set of Java classes that provide an API for routing Xilinx FPGA devices [Xil01]. This interface provides various levels of controls that include turning on/off a single connection, routing a single source to a single sink, routing a single source to several sinks, routing a bus connection, and routing by specifying a path or a template. This API also allows the user to define ports that can be used for automatic routing. The Unroute option offers the designer the flexibility to free some unused resources. Built on JBits and currently supporting Virtex architectures, the JRoute API presents the functionality to route between and inside CLBs; therefore, it makes JBits-based FPGA design easier. More information about the JRoute API can be found in [Xil01].

2.4.4 BoardScope
BoardScope is a graphical and interactive debug tool for Xilinx Virtex FPGAs [Xil01]. It supplies an integrated environment for designers to look at the operation of a circuit on a real device or on a JBits-based simulator. By stepping and changing the clock, the user can graphically see how the state of each CLB changes and how the circuit operates. This tool has four different views to display the design. The four views, namely state view, core view, power view, and routing density review, can show the states of resources in a CLB, the placement of cores, the activity level of each area, and the routing resource used in each CLB respectively. Using Xilinx Hardware Interface (XHWIF), BoardScope can run on a real device; based on JBits APIs, BoardScope can run on a device simulator. This tool offers the designer a powerful debug environment for JBits-based FPGA design.
2.4.5 A simple example for JBits-based FPGA design

This section presents a simple FPGA design example using JBits tools. In this example, a few numbers saved in an input file are read from a FIFO and saved in a register. The last bit of each number is used to enable a counter. If the bit is “1”, the counter is increased by 1, otherwise the counter holds the previous value. The latest value in the counter will be saved in another register. Figure 2.3 shows the diagram of the simple circuit.

In this example, we will use Xilinx provided RTPCores to implement the logic gates and use JRoute to make connections between these gates. The first step in JBits-based FPGA design is choosing a device and initializing the JBits and JRoute instance. This can be done by three simple function calls:

```java
JBits jbits = new JBits(Devices.XCV300);
JRoute jroute = new JRoute(jbits);
Bitstream.setVirtex(jbits,jroute);
```

To instantiate the logic gates, one needs to define the input and output buses or nets for each core, and then create the gates using the following function calls:

```java
Clock clock = new Clock("Clock", clk);
TestInputVector inputGen = new TestInputVector("FIFO",16,clk,reset);
Register reg1 = new Register("Register1",clk,reset,rout1);
Register reg2 = new Register("Register2",clk,cout,rout2);
Counter counter = new Counter("Counter",cp);
```

Four different RTPCores are instantiated in this example. TestInputVector core is used to
implement the FIFO function. Although clock core is not indicated in the diagram, it is necessary to simulate the real clock on the device. \texttt{Clk, reset, rout1, rout2} and \texttt{cout} are the input and output buses or nets for each core; \texttt{cp} defines the property of the counter. These cores can be placed on the device by specifying the row and the column.

\begin{verbatim}
Offset tvOffset = inputGen.getRelativeOffset();
tvOffset.setVerOffset(Gran.CLB, row);
tvOffset.setHorOffset(Gran.CLB, col);
\end{verbatim}

Where \texttt{row} and \texttt{col} are the row and column of the bottom left corner of the \texttt{inputGen} core.

After manually placing the cores, we can implement them using \texttt{RTPCore.implement()} function. The parameters for this function call vary from core to core. For example, the parameter for the TestInputVector core is the name of the FIFO input file. Routing is the next step in the design. Because buses and nets have been defined for each core, JRoute can be used to connect these cores automatically. To connect all of the cores to net \texttt{clk}, for example, a simple JRoute call: \texttt{Bitstream.connect(clk)} is used. After routing, the entire design can be saved to a bitstream file using \texttt{jbits.write(filename)}. Once the design is compiled using the Java compiler and under a Java virtual machine, the bitstream is generated in seconds. Figures 2.4 and 2.5 show the core view and the state view of this example design.
Figure 2.4 Core view of the example design

Figure 2.5 State view of the example design
2.5 Summary

From this chapter, the importance and the potential of the incremental compilation in shortening FPGA development cycle has been emphasized. Contemporary approaches are starting to apply incremental techniques in the FPGA design tools, but most of them are employed to speedup the processing only when minor changes are made in an application. Xilinx JBits toolkit presents a new way for bitstream generation. Unfortunately, designing a circuit directly using JBits toolkit requires the designer having profound knowledge of FPGA architecture; a JBits-based FPGA design tool does not exist that can help the designer place, route, and generate the bitstreams automatically. Manual placement of a million-gate design is impractical and would limit the popularity of the JBits toolkit.

Therefore, it is necessary to develop a user-interactive integrated FPGA design environment and an efficient design methodology that can process both the small modifications and the entire design from scratch, and can significantly improve the design-and-debug cycle for million-gate FPGA designs, as FPGAs are widely used in prototype developing, design emulation, system debugging, and modular testing.