Chapter 6

Application Development

After investigating the incremental placement algorithm, the guided placement methodology, the incremental million-gate FPGA integrated design environment (IDE), and its background refiner, design applications with sizes varying from tens of thousands to approximately a million logical gates are built to evaluate the execution of the algorithms and the design tool. This chapter describes the development of these design applications and the experimental setup.

Generally, standard benchmark circuits are employed in algorithm evaluations and comparisons in VLSI computer aided design (CAD). MCNC benchmark circuits [Koz91] maintained and distributed by the Microelectronics Center of North Carolina are the most widely used benchmark circuits. Betz and Rose [Bet97] tested their Versatile Place and Route (VPR) tool using the MCNC standard circuits and also set up an FPGA Place-and-Route competition. Mulpuri and Hauck [Mul01] used these benchmark circuits to analyze the runtime and quality tradeoffs in FPGA placement and routing. Built to test the conventional placement algorithms, most of these benchmark circuits are designed for standard cells. Four of the standard cell benchmarks, *primary1, primary2*, *industry2, and industry3*, could be used as gate array benchmarks. These are distributed

in a .net format that removes the design hierarchy; therefore, they cannot be employed to test the core-based incremental placement algorithm described in this dissertation.

Five block placement benchmark circuits have been designed in MCNC for testing algorithms that handle blocks with different sizes. Converting their format into JBits API requires a lot of work, and the sizes of these circuits are too small to be used in million-gate FPGA design tests. Built in the late 1980's, the largest block placement benchmark circuit *ami49* only has 408 nets [Koz91], which are far from the design size targeted in this dissertation.

To validate the algorithms and the design tool, a number of large designs are required. As a means of creating a large design data set, a collection of handcrafted designs is augmented with synthetic random circuits. Two sets of designs are developed and their implementations are described in the following sections.

6.1 Random Circuit Generator

There are two requirements for building the test circuit generator. First, a large number of circuits with various design sizes need to be generated; second, the created circuits must represent the common characteristics of generally used design circuits in terms of connectivity, shape, size, and total number of design components.

A circuit with a random number of randomly connected randomly sized cores is a good candidate to simulate a large number of real design circuits. In a real design, the probability of connectivity, shape, size and total number of design components is uniformly distributed. To develop such a circuit generator, a `RandomCore` class has been implemented using the JBits API.

The `RandomCore` is actually a dummy core; it does not have any real task except to simulate a certain sized component that may connect to other cores. This dummy core is implemented by extending the Xilinx RTPCore library. The constructor of this core is defined as:

```
public RandomCore (String name, Net clock, Bus inBus, Bus outBus,
                   int Width)
```

To instantiate this Randomcore, five parameters are necessary:

- *Name*- a string to define the name of this core;
- *Clock*- an external net to be connected to the Clock port;
- *InBus*- an external bus to be connected to the input Port;
- *OutBus*-an external bus to be connected to the output port;
- *Width*-an integer to define the width of the core.

The height of this core is determined by the width of the output bus. To implement a RandomCore with the output bus width ten, the height of this core should be defined as five. Similar to the JBits RTPCore, an `implement()` method has been developed to map the circuit into configurable logic blocks. A `ConnectNet()` method has been created to achieve the internal routing for this core.

Three connectivity patterns are defined in the random circuits. The randomly sized random cores can be connected in a one-by-one order, or in partially or fully random configurations. In a one-by-one pattern, the currently generated core's input is connected to the previously generated core's output, while the current core's output will be connected to the next generated core's input. In the fully random connection, a previously generated core is randomly chosen to connect with the currently generated core, and the previously generated core's output is connected to the newly generated core's input. In the partially random connection pattern, only ten percent of the cores choose their connection objects randomly, and the others are all connected in the one-by-one order. Because the height and width of each core is randomly generated, if the two connected buses do not have the same width, the minimum bus width is chosen as the real connection bus width, thus making routing feasible.

This synthetic circuit generator creates about 100,000 different circuits with different sizes to evaluate the incremental placement algorithms and the design tool. All of the test

designs are built on Xilinx Virtex FPGA XCV300 and XCV1000 respectively, where the XCV1000 is a Xilinx FPGA with more than one million system gates [Xil98]. The design descriptions are shown in Table 6.1.

Device	Number of System Gates	Number of Rows	Number of Columns	Mean of core width	Mean of core height	Mean number of cores
XCV300	322970	32	48	2.5 Cols	5 Rows	60
XCV1000	1124022	64	96	2.5 Cols	5 Rows	246
XCV1000	1124022	64	96	5 Cols	10Rows	62

Table 6.1 Random circuit design description

During the design test, the first random number is generated to determine how many cores will be needed to fill up the FPGA; the second and the third random numbers set the height and the width of the currently generated core; the fourth random number determines the connection pattern among cores. The entire design was developed in the Java language by extending the `UserDesign` class discussed in Chapter 4.

The following tables describe the statistics of the designs employed to evaluate the incremental design tool. Table 6.2 shows the design information for circuits with a mean core height of five rows and a mean core width of 2.5 columns on the Virtex XCV300. Table 6.3 shows the design information for circuits with the same mean size of cores on the Virtex XCV1000. Table 6.4 presents a set of circuits with larger sizes at a mean height of ten rows and a mean width of five columns on the Virtex XCV1000. The average number of CLBs, cores, and nets is calculated and then displayed in these tables. The number of nets in those designs with over eighty percent CLB utilization is close to that of the large standard cell MCNC circuits, which average over ten thousand CLBs.

Device utilization	# of CLBs	Average # of cores	Average # of nets		
			One-by-one	Partially	Fully
20%	307	25	519	542	518
30%	461	37	790	824	790
40%	615	49	1058	1114	1056
50%	768	61	1335	1392	1334
60%	922	74	1599	1680	1597
70%	1075	86	1874	1958	1868
80%	1229	98	2145	2239	2147
90%	1383	111	2418	2512	2409

Table 6.2 Design statistics for random circuits with mean height of five rows and mean width of 2.5 columns on the Virtex XCV300

Device utilization	# of CLBs	Average # of cores	Average # of nets		
			One-by-one	Partially	Fully
20%	1229	98	2194	2267	2169
30%	1843	147	3225	3365	3250
40%	2458	197	4262	4504	4306
50%	3072	246	5384	5631	5401
60%	3687	295	6482	6772	6458
70%	4301	344	7577	7883	7551
80%	4916	393	8644	9050	8640
90%	5530	442	9693	10142	9732

Table 6.3 Design statistics for random circuits with mean height of five rows and mean width of 2.5 columns on the Virtex XCV1000

Device utilization	# of CLBs	Average # of cores	Average # of nets		
			One-by-one	Partially	Fully
20%	1229	25	2222	2270	2210
30%	1843	37	3342	3419	3414
40%	2458	49	4552	4549	4578
50%	3072	61	5730	5662	5721
60%	3687	74	6863	6875	6872
70%	4301	86	8055	8031	8051
80%	4916	98	9198	9199	9204
90%	5530	111	10301	10383	10316

Table 6.4 Design statistics for random circuits with mean height ten rows and mean width of five columns on the Virtex XCV1000

6.2 Polynomial Computation Example

After building a dummy random circuit generator, the second design class is derived from actual applications that compute large polynomials for data compression. In high performance computing, to transfer data fast and efficiently, a large mesh of data can be approximated using polynomials. Coefficients of these polynomials are transferred through the network in place of the entire mesh to speed up the transmission rate and to reduce network load. At the receiver side, the mesh data are recovered by computing large polynomials. This computation is often achieved using FPGAs due to the intense resource requirements and the desirable reconfigurability property.

This example simulates the data retrieval function at the receiver side. It reads the coefficients of a polynomial $y = \sum_{i=0}^{n} a_i x^i$ from a FIFO and computes the value of the polynomial according to a given degree *n* and a variable *x*. To make the computation faster, x^n is calculated using $x^n = x^{n-1} * x$, a typical circuit with polynomial degree three is shown in Figure 6.1.

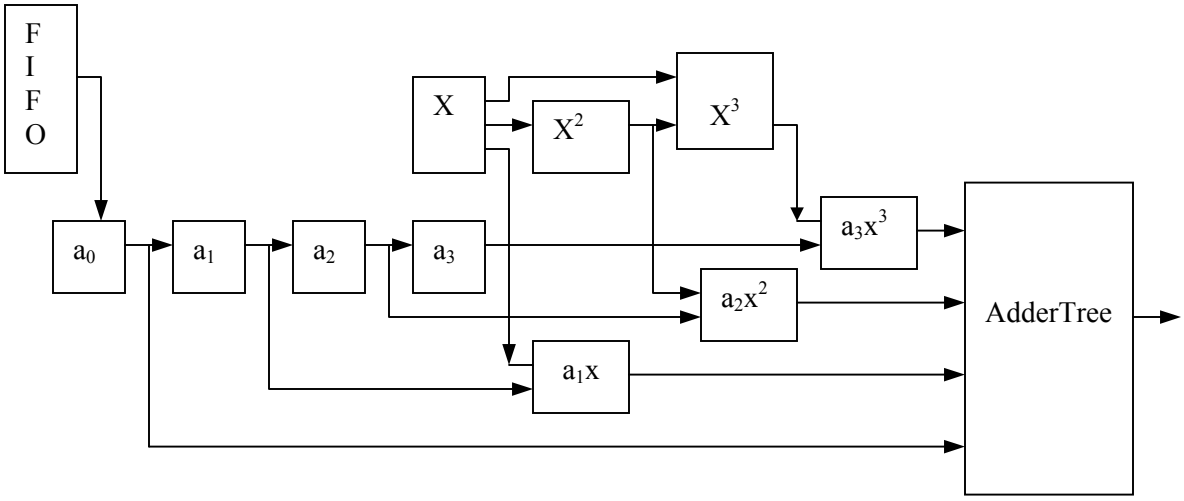

Figure 6.1 Circuit schematic of a polynomial with degree 3

Degree	# of CLBs	CLB utilization	# of cores	# of nets
3	113	7.36%	17	311
4	169	11%	20	498
5	307	19.97%	23	725
6	362	23.57%	26	992
7	477	31.06%	29	1299
8	589	38.35%	32	1646
9	808	52.6%	35	2033
10	952	61.98%	38	1460
11	1134	73.83%	41	2927

Table 6.5 Design Statistics for polynomial computation design on Virtex XCV300

Degree	# of CLBs	CLB utilization	# of cores	# of nets
11	1134	18.46%	41	2927
12	1304	21.22%	44	3434
13	1546	25.16%	47	3981
14	1744	28.38%	50	4568
15	1988	32.36%	53	5195

16	2212	36%	56	5862
17	2638	42.94%	59	6569
18	2896	47.14%	62	7316
19	3208	52.21%	65	8103
20	3492	56.84%	68	8930
21	3880	63.15%	71	9797
22	4192	68.23%	74	10704
23	4566	74.32%	77	11651

Table 6.6 Design Statistics for polynomial computation design on the Virtex XCV1000

Basic cores used in this example are a FIFO, registers, counters, constants, multipliers, and adders. All of the cores are instantiated directly from the JBits RTPCore library. The entire design was developed in Java following the format discussed in Chapter 4. To test the function of handling large cores, an AdderTree was used instead of several adders. The size of the design varies by changing the degree and the variable bit width of the polynomial. The largest core in this design is an AdderTree with a width of 24 columns and a height of 50 rows. Tables 6.5 and 6.6 present the design description of the polynomial computation design class.

Both sets of designs were tested on a Virtex XCV300 and a XCV1000 under Windows 2000 on a 1-GHz PC with 1.5GB of RAM. Both sets of the circuits were placed using the incremental techniques, and were routed using JBits JRoute. For comparison purposes, the polynomial computation design was implemented using the incremental design tool and JHDL [BYU01], and synthesized by the incremental design techniques and the standard Xilinx M3 (version 3.3.08) tool respectively. Although originally designed as the background refiner for the incremental design tool, the core-based simulated placer implemented in Chapter 5 was also employed as a competitor for the incremental placement algorithms and was tested using the polynomial computation circuit family. The following sections briefly introduce the JHDL implementation and Xilinx Tools.

6.3 Just Another Hardware Description Language

The Just another Hardware Description Language (JHDL) [BYU01][Bel98] developed at Brigham Young University is a set of object-oriented FPGA CAD tools that allows hardware developers to create FPGA designs with an original Java development system. Based on a popular computer language, this platform provides FPGA designers a front-end tool that can be used to design and debug a circuit, simulate the functionality, and netlist and interface a design with back-end tools for bit-stream synthesis. In addition, it also supports both partial and global configurations. Just Another Browser (Jab) is the integrated design environment for JHDL. As shown in Figure 6.2, this IDE supports the Slaac1V board [ISI01] that uses three Xilinx XCV1000 FPGAs as Processing Elements (PEs).

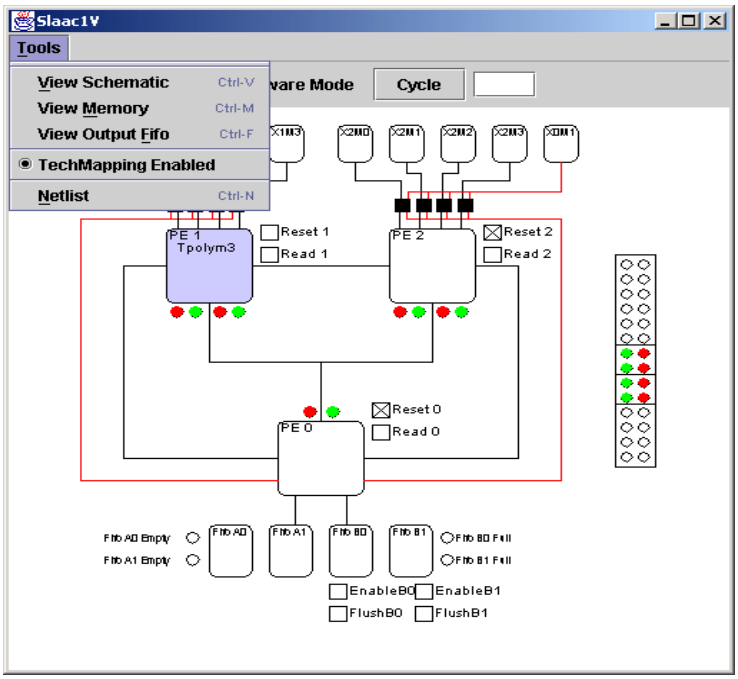

Figure 6.2 Interface of the Slaac1v Platform in Jab

To implement a hardware design using Java, and to simulate and netlist the design using JHDL, the design class should extend the super class `pelca` and define the input/output port of the circuit first. A typical design example is shown as follows:

```
import byucc.jhdl.platforms.slaac1v.*;
import byucc.jhdl.Xilinx.*;
….
public class Tpolym3 extends pelca {
  public static CellInterface[] cell_interface = {
    in("RESET", 1),
    out("ADDR",8),
  …
    };

  public Tpolym3 ( pe parent) {
    super(parent);

    Wire RESET = connect("RESET", wa("Reset"));
    Wire Right = connect("Right", wa("Right",47,0));

    Wire regout = wire(WIDTH,"regout");
    reg_o(DATA,regout,"register0");
     ….
```

In this example, the input/output ports are first connected with their corresponding wires on the Slaac1v board, and then a register is instantiated using the JHDL component library.

After building the user's design, the compiled byte code can be loaded into one of the PEs for functional simulation or netlist synthesis as demonstrated in Figure 6.2, where `Tpolym3` is the Java class of the design file. After netlisting, designers can browse the circuit by tracing the input/output ports of a specific component or viewing the schematic of the design layout. Figures 6.3 and 6.4 present the interfaces. The netlisted design is placed and routed using the Xilinx M3 tools (version 3.3.08).

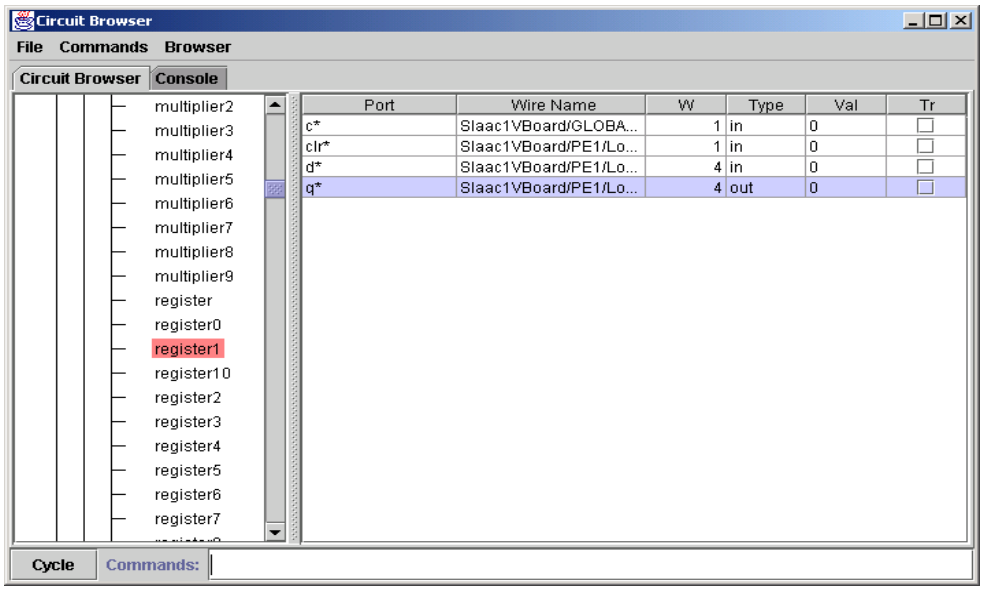

Figure 6.3 The circuit browser window

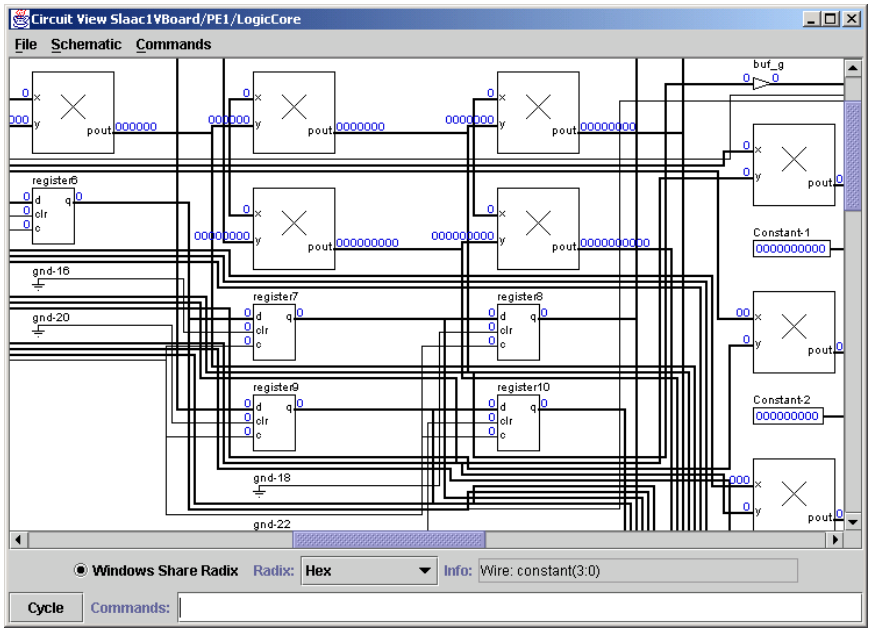

Figure 6.4 The schematic view window

6.4 Xilinx Tools

The netlisted designs are synthesized using the standard Xilinx M3 Tools. The placement tool is part of the commercial Xilinx Foundation Series 3.3i. The placer runs at five different effort levels from 1 to 5. The higher the effort level, the higher the quality of the placement but the longer the processing time. The effort level was set to 3 during the design tests in this dissertation to find a tradeoff between the processing time and the placement quality, making it comparable with other placement tools.

The synthesis was run under the Cygwin [Cyg01] environment, which assists developers in migrating applications from UNIX/Linux to the Windows platform. The processing time for placement, routing, and the entire synthesis procedure was recorded in a report file and compared to the incremental placement tool. The placed-and-routed designs were saved in a Native Circuit Description (.NCD) format and were read using the Xilinx FPGA Editor to browse the design details. Figure 6.5 shows a placed design in FPGA editor.

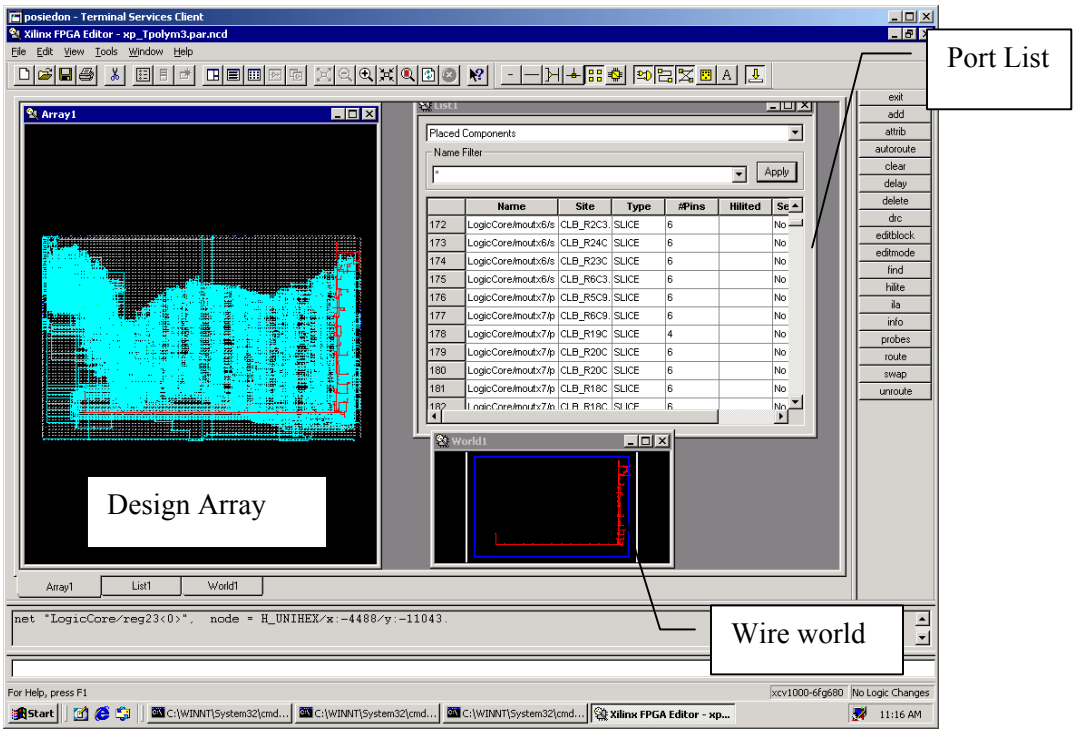

Figure 6.5 Browsing a placed design using FPGA Editor

6.5 Summary

Two sets of handcrafted testing circuits have been developed in this chapter. A large number of testing circuits with gate sizes varying from tens of thousands to approximately one million have been generated to evaluate the incremental placement algorithms and design tool. To compare the incremental placer with its competing approaches, some of the designs are developed in JHDL and synthesized using a commercial Xilinx placer. The JHDL implementations and the features of Jab design environment and the Xilinx tools have been introduced in this chapter as well. The experimental results of the testing circuits are analyzed and presented in Chapter 7.