# Ph.D. Annual Monitoring Report

by Andy Gean Ye

## 1. Introduction

This report reviews my Ph.D. research for the academic year 2000–2001. Our research is in the area of FPGA architectures for datapath applications. In particular, the goal is to increase FPGA logic density by utilizing datapath regularity in the application circuits. Our research methodology is empirical. We use custom CAD tools and benchmark circuits to evaluate various architectural alternatives. At the start of the year, we proposed a basic datapath FPGA architecture. During the year, we created a synthesis tool and a packing tool that synthesizes and packs VHDL and Verilog circuits into the proposed architecture. We created a benchmark suite of fifteen datapath circuits. We also made architectural improvements based on empirical results.

We found that current synthesis tools produce much higher Look Up Table (LUT) count when configured to preserve datapath regularity and hierarchy. In many cases, the LUT count inflation can be as much as 50% compared with regular flat compile modes, which do not preserve datapath regularity. We augmented a conventional synthesis tool (Synopsys Design Compiler) to reduce this LUT count inflation. When compared with the conventional tool operating on a flat circuit (not configured to employ hierarchy), the augmented, hierarchy-retained, approach has an average LUT count inflation of only 3%.

We also designed a packing tool that packs the synthesis tool output into the clusters of our proposed architecture. In terms of Basic Logic Element (BLE) utilization, our tool is almost as good as regular packing tools packing LUTs into comparable regular FPGA clusters.

We found that after packing there still is a high degree of regularity in the fifteen benchmark circuits. In particular, the majority of signals in these circuits can be classified into two types — buses and controls. Around 48% of two terminal nets can be grouped into 4 bit wide buses and about 35% of two terminal nets are from high fan out control signals. There is little overlap between these two types of signals.

The rest of this report is divided into six sections. Section 2 discusses the project motivation and

experiment methodology. Section 3 describes the proposed datapath FPGA architecture. Section 4 and Section 5 describes the synthesis and packing tools respectively. Section 6 presents experiment results. The conclusion and future work are presented in Section 7.

## 2. Project Motivation and Experiment Methodology

In this thesis, we investigate FPGA architectures specialized for datapath applications. In recent years, the capacity of FPGAs has been continuously increasing. With this increased capacity, more and more datapath applications are implemented on FPGAs. Nowadays, it has become feasible to implement an entire CPU or graphic processor on a single FPGA. Except for dedicated carry logic, however, the majority of the current commercial FPGAs include little support for datapath applications. We feel that there are two major areas where this increase in datapath logic can be utilized to improve FPGA logic density and performance.

First, current commercial CAD tools do not efficiently use the regularity of datapath applications. Datapath designs are typically flattened into a single level of flat logic and go through the same synthesis process as random logic circuits. The alternative method of structural synthesis, which keeps the bit-level hierarchy intact, is not well supported by commercial or academic tools.

In structural synthesis, one does not arbitrarily flatten a datapath design. Instead, the synthesizer keeps track of the regularity of datapath applications. This regularity information is then passed down to the subsequent stages in the CAD flow. In current commercial tools, structural synthesis often results in much higher LUT count than flat synthesis. This inflation in LUT count has become a strong dis-incentive for using structural synthesis, and is detrimental to the overall efficiency of any datapath FPGA architecture that utilizes datapath regularity.

Previous research, [1] [2], has shown that the regularity of datapath applications can be used to increase the placement and routing density of their FPGA implementations. Furthermore, the work in [3] has shown that under appropriate optimization techniques structural synthesis of datapath circuits can be as efficient as flat synthesis in terms of standard cell literal count. But these capabilities have not yet been implemented in today’s commercial tools. In this thesis, we create CAD tools that can structurally synthesize datapath circuits into LUTs in order to make efficient use of the proposed datapath architecture; furthermore, we use various techniques to ensure that

our structural synthesis approaches the efficiency of flat synthesis.

Second, commercial FPGA architectures do not fully take advantage of the regularity of datapath applications. Especially the technique of configuration memory sharing is not widely used to achieve area saving. In a regular FPGA architecture, reconfigurable resources are independently controlled by their own configuration memory. In a configuration memory sharing architecture, on the other hand, a group of reconfigurable resources share a single set of configuration memory [4]. By sharing configuration memory, all members in the group behave identically. The work in [4] studied configuration memory sharing in FPGAs based on regular single LUT clusters. The effect of configuration memory sharing on routing was estimated. In this thesis, we proposed a new configuration memory sharing scheme for FPGAs based on multi-LUT clusters. Routing architecture will also be studied in detail, where actual place and route tools will be designed for the new architecture.

**Experiment Methodology**

We use an empirical methodology to evaluate the efficiency of our proposed FPGA architecture. First, we collect a suite of datapath intensive applications to be used as benchmarks. These benchmarks are then converted into a consistent hierarchical description in either Verilog or VHDL format as specified by our datapath-oriented CAD flow. Currently, our benchmark set consists of fifteen datapath circuits from the Pico-Java processor [7]. We are planning to convert and add more circuits including DSPs, ASICs and other CPUs to our benchmark suite in the coming year.

Then, we use the datapath-oriented CAD flow, designed as part of this thesis to synthesize, pack, place, and route these benchmarks into the proposed FPGA architecture. Data are then collected for choosing the best architectural alternative and comparing the proposed architecture with the traditional architectures. In the past year, we have completed the synthesis and packing tools. These tools are described in detail later in the report.

## 3. Features of the Datapath FPGA Architecture

This section briefly describes the datapath FPGA architecture that we are currently investigating. All features except the datapath control signal distribution network are described in more detail in [5]. Throughout our architectural work, we assume the basic building blocks of our datapath

FPGAs are BLEs as shown in Figure 1. Each BLE consists of a single LUT and a single D type Flip Flop (DFF). This basic building block is widely used in many academic studies including [6].

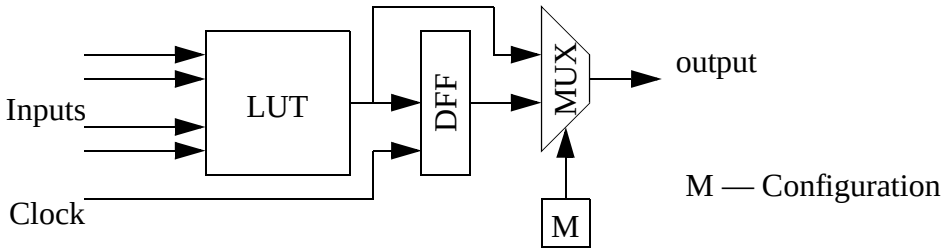



**Figure 1: A Basic Logic Element (BLE)**

This section is divided into three subsections. Subsection 1 describes the clustering strategy of our architecture. Subsection 2 describes how configuration memory sharing is used to route bus signals. Finally, Subsection 3 describes how high fan out control signals are distributed in our proposed architecture.

**Clusters and Subclusters**

BLEs are grouped into two levels of clustering hierarchies. We call the top level *clusters*, and the lower level *subclusters*. A subcluster is shown in Figure 2. Each subcluster consists of **N** BLEs with **I** subcluster inputs and **N** subcluster outputs. **I** and **N** are parameters that we will use our empirical methodology to explore. Each BLE output is connected to a corresponding subcluster output. Each BLE input is fully connected to all subcluster inputs and subcluster outputs through a subcluster local routing network.

A single cluster consists of **m** subclusters. Since datapath circuits usually consist of identical bit slices, a single cluster can be used to implement an **m** bit wide datapath circuit with **N** or less LUTs per slice. For larger datapath circuits, we decompose them into smaller datapath circuits each of which can be implemented by a single cluster. When implementing random logic circuits, each subcluster behaves as a regular FPGA cluster as defined in [6]. Clusters are used in conjunction with the bus routing and the control signal distribution networks to efficiently implement datapath circuits. A cluster is shown in Figure 3.

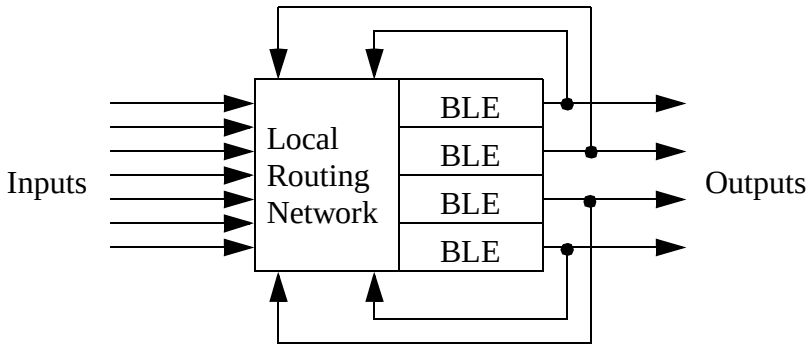



**Figure 2: A Subcluster**

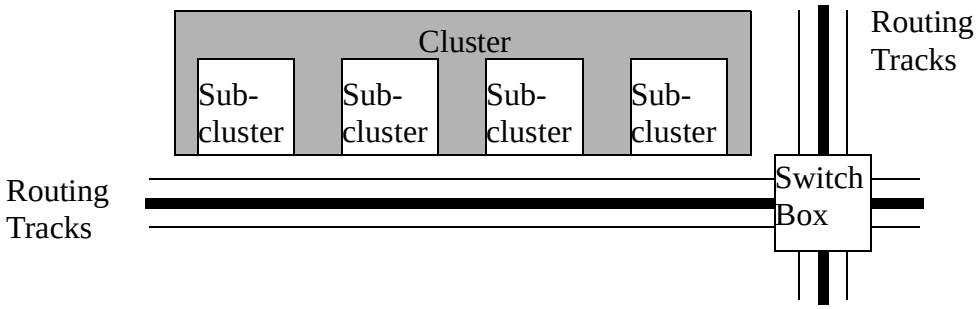



**Figure 3: A Cluster**

**Coarse Grain Routing Channels**

Our FPGA routing architecture contains two types of routing channels — coarse grain routing channels and fine grain routing channels. Coarse grain routing channels are designed to route bus signals. Tracks in a coarse grain routing channel are grouped into **m** bit wide buses. **m** is equal to the number of subclusters in a cluster. Within each routing bus, corresponding switches on distinct tracks share the same configuration memory. By sharing configuration memory, coarse grain routing channels can route bus signals in less area than regular fine grain routing channels, and this is one of the area savings that we anticipate to achieve with the datapath architecture.

Not all signals in datapath circuits can be grouped into buses. For example, random logic signals from control logic can rarely be grouped into buses. It is inefficient to use a wide routing bus for

just a single bit signal. For these signals, we include regular routing resources in our routing architecture. We call these routing resources *fine grain routing channels.* We will study the optimal proportion of the two types of routing resources in this thesis. A cluster with one coarse grain routing bus and one fine grain routing track is illustrated in the Figure 4.

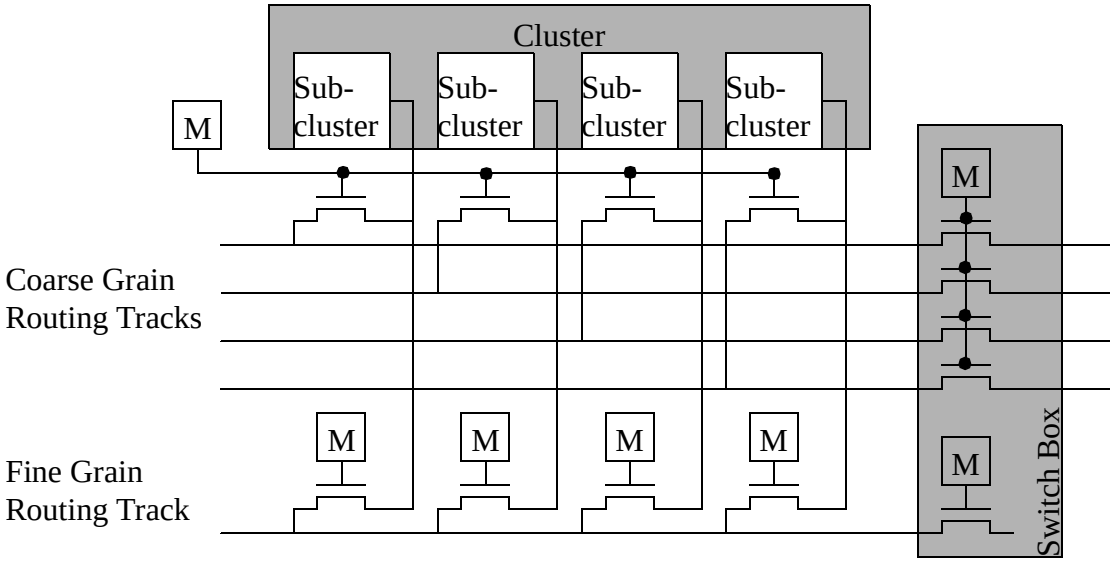
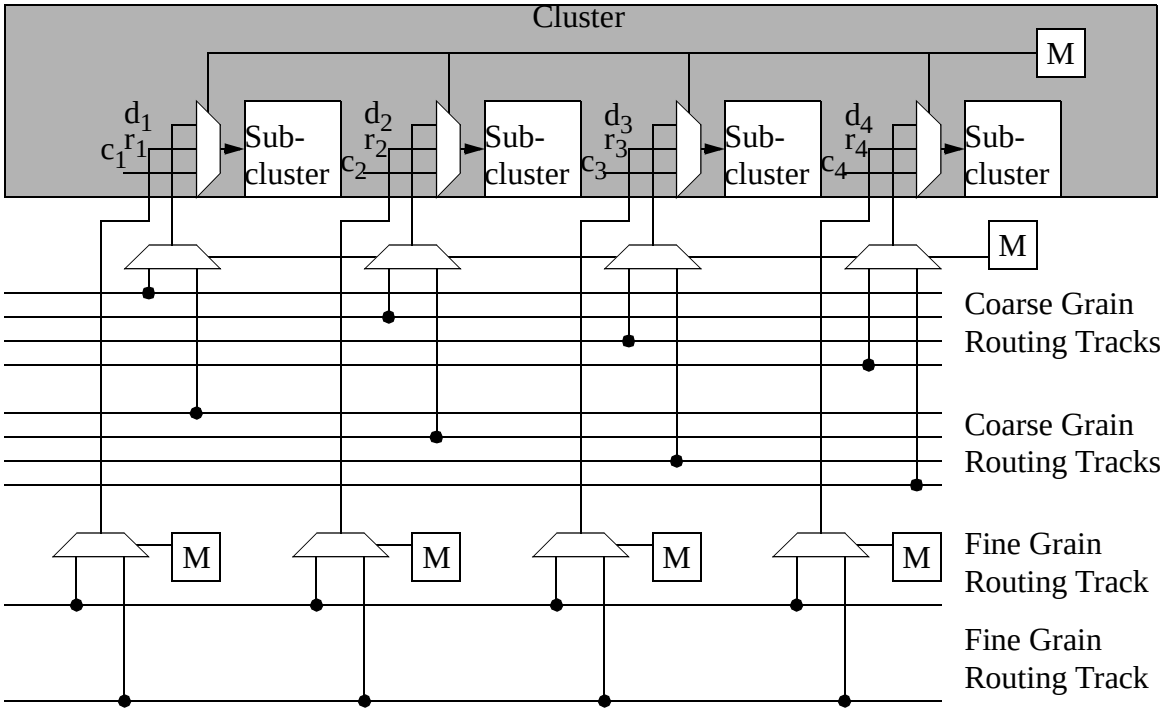



**Figure 4: Coarse Grain Routing**

Figure 4 also illustrates how cluster outputs are connected to coarse grain routing buses. When connecting to a coarse grain routing bus, each subcluster output only connects to one distinct routing track. As usual the connection is configurable through a routing switch; however, all connections share the same configuration memory. The input connections to coarse grain routing buses are similar and are illustrated in Figure 5.

Also shown in Figure 4 and Figure 5 are the regular output and input connections to fine grain routing tracks, respectively. When connecting to fine grain routing tracks, each subcluster behaves as a regular FPGA cluster. The connection boxes between subclusters and the fine grain routing tracks are similar to the connection boxes in a regular FPGA architecture [5].

In Figure 5, signals $c_1$, $c_2$, $c_3$, and $c_4$ are from a control input connection, which is not shown in




$d_1$, $d_2$, $d_3$, $d_4$ — datapath input connections from coarse grain routing buses
$r_1$, $r_2$, $r_3$, $r_4$ — regular inputs connections from fine grain routing tracks
$c_1$, $c_2$, $c_3$, $c_4$ — control input connections from fine grain routing tracks

**Figure 5: Input Connection Boxes**

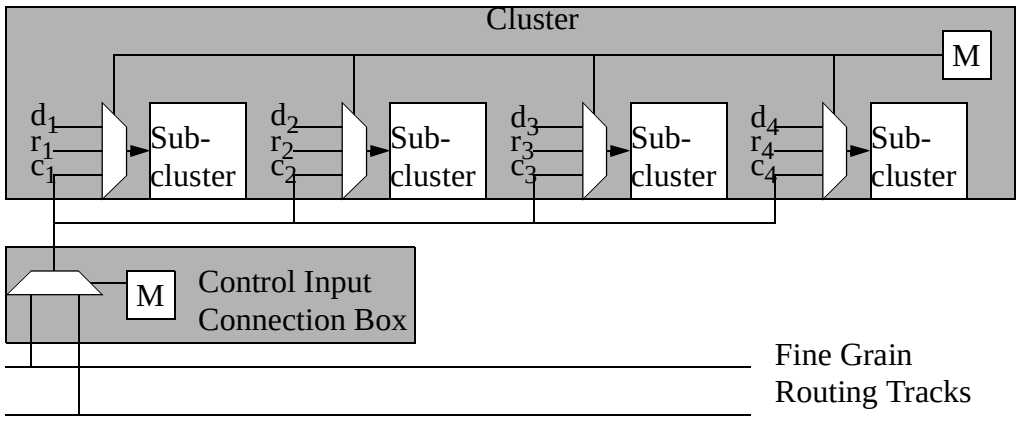
the figure. Control input connections are discussed in detail in the next subsection.

**Special Control Signal Distribution Network**

Although control signals only constitute a small percentage of total nets in datapath circuits, they are usually high fan out and constitute a large percentage of two terminal nets in datapath applications. We find almost as many two terminal control nets as two terminal bus nets in our benchmark set. Section 6 presents this data in detail.

A typical control signal is generated by random logic (non-datapath) and used in several bit slices. Due to the regularity of datapath circuits, a control signal typically fans out to a group of identical bit slices. This regularity can be potentially used to increase the efficiency of control signal routing networks.

In our routing architecture, we route control signals through the fine grain routing tracks. Special control input connection boxes shown in Figure 6 are designed to efficiently distribute control signals in clusters. The connection box selects a track from the fine grain routing channel and distributes this signal to all subclusters in a cluster. When a control input connection box is used, it functions as **m** regular fine grain input connection boxes. As with coarse grain routing channels, we anticipate area savings from this architectural feature.




$d_1$, $d_2$, $d_3$, $d_4$ — datapath input connections from coarse grain routing buses
$r_1$, $r_2$, $r_3$, $r_4$ — regular inputs connections from fine grain routing tracks
$c_1$, $c_2$, $c_3$, $c_4$ — control input connections from fine grain routing tracks

**Figure 6: Control Signal Distribution**

## 4. Datapath-Oriented Structural Synthesis

Our proposed FPGA architecture requires synthesis tools that preserve datapath regularity. In our datapath-oriented structural synthesis, this is achieved by keeping bit slice descriptions in their own hierarchy. Similar to the methodology used in [3], optimization steps that will destroy the regularity of datapath circuits are not performed.

The synthesis CAD flow is illustrated in Figure 7. This flow takes HDL (Hardware Description Language) specifications as inputs. The datapaths must be specified by instantiating components from our predefined library called *Datapath Component Library.* The HDL descriptions are then optimized and mapped into LUTs and DFFs through a three stage iterative optimization process,

which uses our own structural optimization algorithm in stage two, *Structural Optimization*, and the Synopsys Design Compiler in the other two stages, *Initial Synopsys Compile* and *Final Synopsys Compile*. Two iterations are currently used. The final output of the CAD flow is in VHDL and is labeled *Final Structural VHDL Description* in Figure 7. It is discussed in detail in Section 5.

For every benchmark, we compare the final LUT count of our structural synthesis with the LUT count of flat Synopsys compile (**flatten compile**). We flat compile the initial structural HDL description and the optimized structural VHDL descriptions from all two iterations. These flat compile sample points are shown in shaded ovals in Figure 7. One is labeled *Initial Structural HDL Description*; the other labeled *Optimized Structural VHDL Description.* The best flat compile result is used in comparison with our structural compile results.

The rest of this section describes the synthesis CAD flow in detail. It is divided into three subsections. The datapath component library is discussed in Subsection 1. The Synopsys Design Compiler configuration is discussed in Subsection 2; and finally our structural optimization algorithm is discussed in Subsection 3.

**Datapath Component Library**

The datapath component library is written in Verilog. It contains fundamental datapath building blocks including multiplexors, adders/subtractions, shifters, comparators, and registers.

To capture the structure of a datapath component, we use a two level hierarchy which consists of bit slices and datapath components. Each bit slice is described behaviorally in its own Verilog module. Each datapath component is also described by a Verilog module. Inside the datapath component modules, corresponding bit slices are instantiated multiple times based on datapath width. Datapath component modules also contain behavior descriptions of all logic that is not part of bit slices.

**Synopsys Design Compiler Configuration**

Synopsys Design Compiler is a powerful and versatile synthesis tool. It can read several popular HDL description languages and apply various optimization methods. It does not, however, effectively deal with the trade off between optimization and datapath preservation. The most effective optimization methods in Synopsys, **flatten compile** and **uniquify compile**, destroy either design

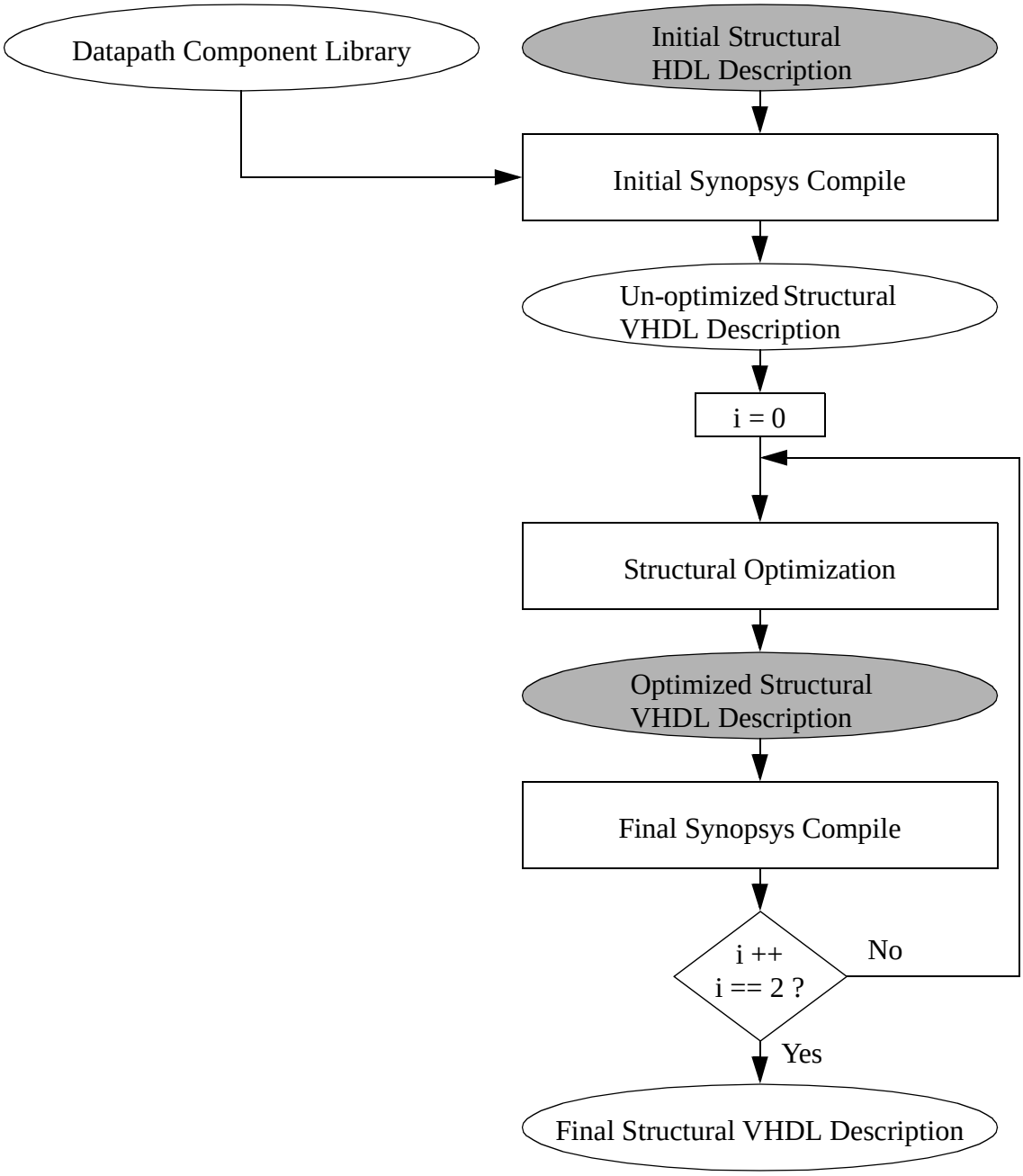



**Figure 7: Overall Flow of the Synthesis Tool**

hierarchy or datapath regularity. The third compile option, **set dont touch compile**, on the other hand, can be used to preserve datapath hierarchy and regularity, but it does not perform optimization across hierarchical boundaries. As the result, **set dont touch compile** is much less effective

than the other two compile options [8].

In order to balance the trade off between optimization and datapath preservation, we use the three stage synthesis process as shown in Figure 7. The initial Synopsys compile stage uses the **set dont touch compile** method. It maps the HDL description into LUTs and DFFs. Before each compile, we set the **dont touch** attribute for each bit slice module and each datapath component module. The output of this stage has a three level hierarchy as shown in Figure 8. These three levels capture the bit slices, the datapath components and the overall design description, respectively.

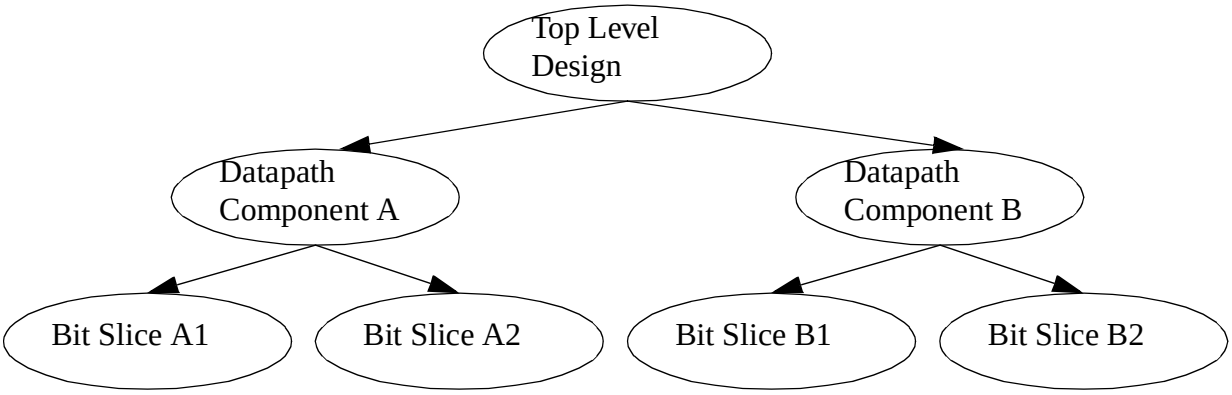



**Figure 8: Hierarchical Structural of the Initial Synopsys Compile Output**

Since the initial Synopsys compile does not optimize across hierarchy boundaries, the LUT count of its output can be as high as 1.5 times of the best Synopsys flat synthesis. In the structural optimization stage we attempt to reduce the LUT count by performing logic optimization across hierarchy boundaries. Using our own structural optimization algorithm, we selectively perform optimizations that are across hierarchy boundaries, but still preserve datapath regularity. The algorithm used in the structural optimization stage is discussed in detail in the next subsection. Designs are then re-mapped into LUTs and DFFs in the final Synopsys compile stage, which uses the **set dont touch compile** method and is similar to the initial Synopsys compile stage. We found that by repeating stage two and three, we can reduce the LUT count to within 1.03 times of the best Synopsys flat synthesis.

**Structural Optimization Algorithm**

At the start of the structural optimization, we divide each datapath component into **m** bit wide chunks, where **m** corresponds the number of subclusters in a cluster. Each chunk of datapath is contained in its own module. An optimization is performed only when it is applicable to all **m** bit slices in a module. Three major optimizations are currently performed during the structural optimization. They are mux tree collapsing, bit slice merging, and bit slice I/O optimization.

The first stage of our optimization is mux tree collapsing[1]. A mux tree sometimes can be substituted with a single mux which requires less logic to implement. An example is shown in Figure 9. Here we can substitute the mux tree on the left with the single mux on the right. To implement the two muxes and the **and** gate on the left we needs two 4 input LUTs. To implement the mux and the **and** gate on the right, we need only one 4 input LUT. The extra random logic in right circuit usually is shared by several bit slices, so its cost is negligible in wide datapath circuits.

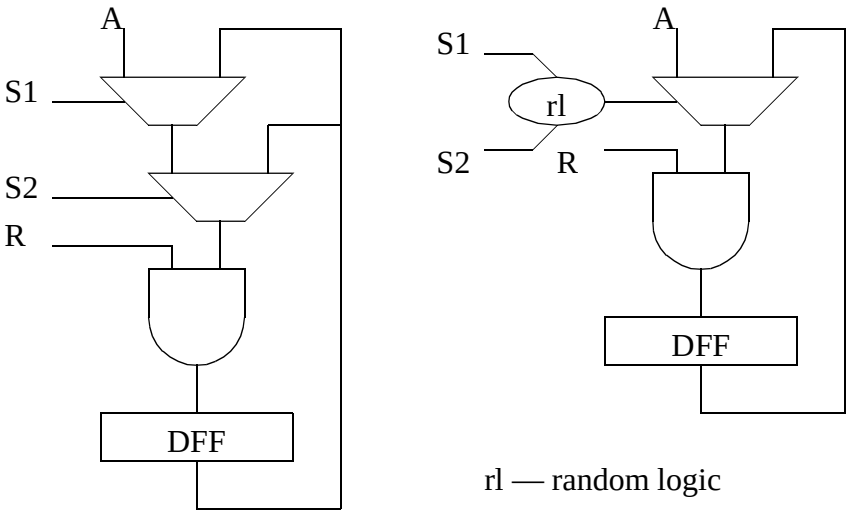



**Figure 9: Mux Tree Collapsing Example**

The second optimization that we perform is bit slice merging. In this stage, we merge two modules together to form larger bit slices. This is a pattern identification process. Two datapath modules are merged together if all bit slices in one module are identically connected to their

1. It is currently performed manually, but will be automated in the future.

corresponding bit slices in the other. An example is shown in Figure 10. Here each bit slice in Module A are connected to a corresponding bit slice in Module B. Furthermore, the nets connecting these slices all have output pin **Ao1** as sources and input pin **Bi2** as sinks. By creating larger bit slices, we create more optimization opportunities for the next stage and the final Synopsys compile.

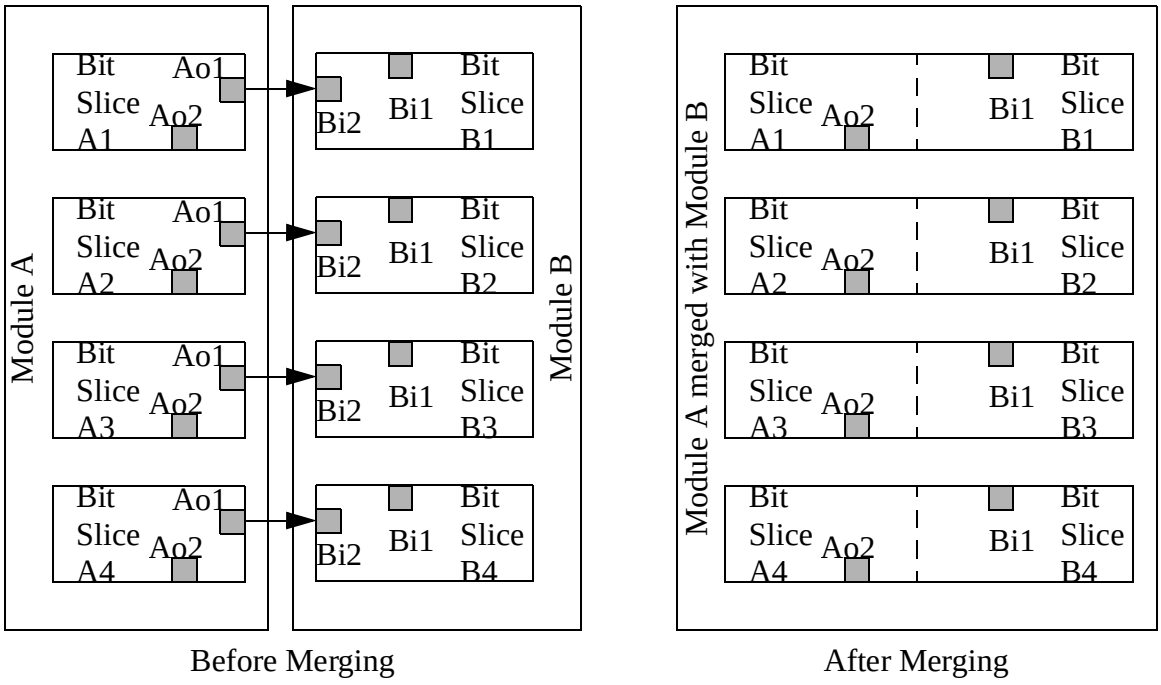



**Figure 10: A Bit Slice Merging Example**

Finally, bit slice I/O optimization is performed. Three types of datapath module input buses and one type of output buses are converted into bit slice internal signals at this stage. Optimization is performed only when all signals in a module's I/O bus meet the same optimization criteria. We eliminate input buses whose signals are all constant one or constant zero as illustrated in Figure 13. Input buses that are connected to output buses from the same module are eliminated as illustrated in Figure 13. Input buses that have the same sources as other input buses are eliminated as illustrated in Figure 13. Output buses that do not have any sinks are also eliminated at this stage.

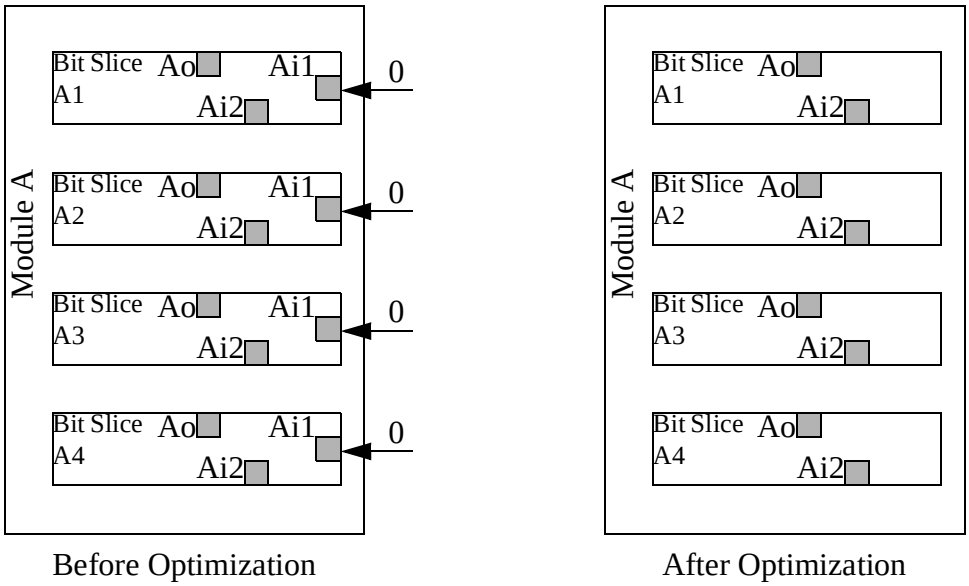



**Figure 11: Input Optimization Criteria A**

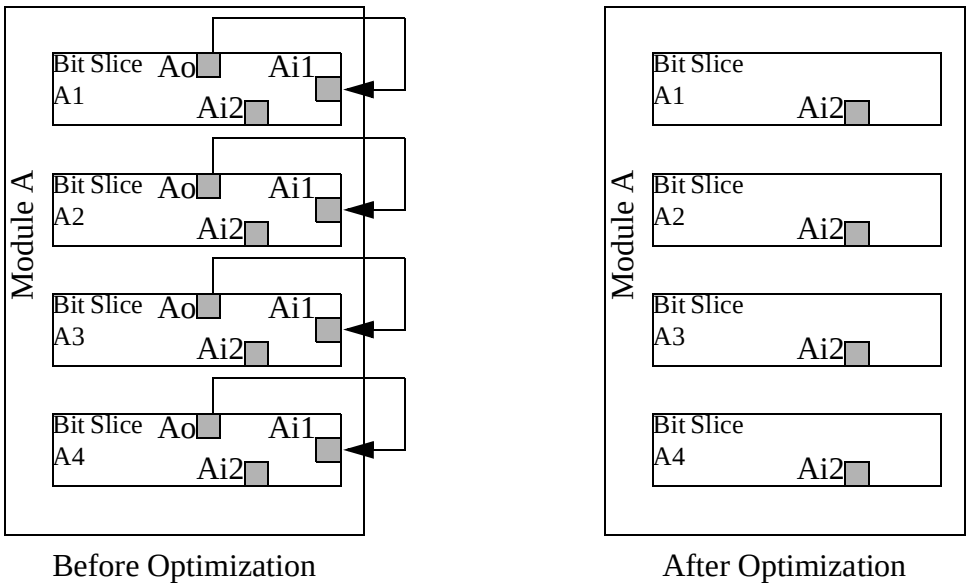
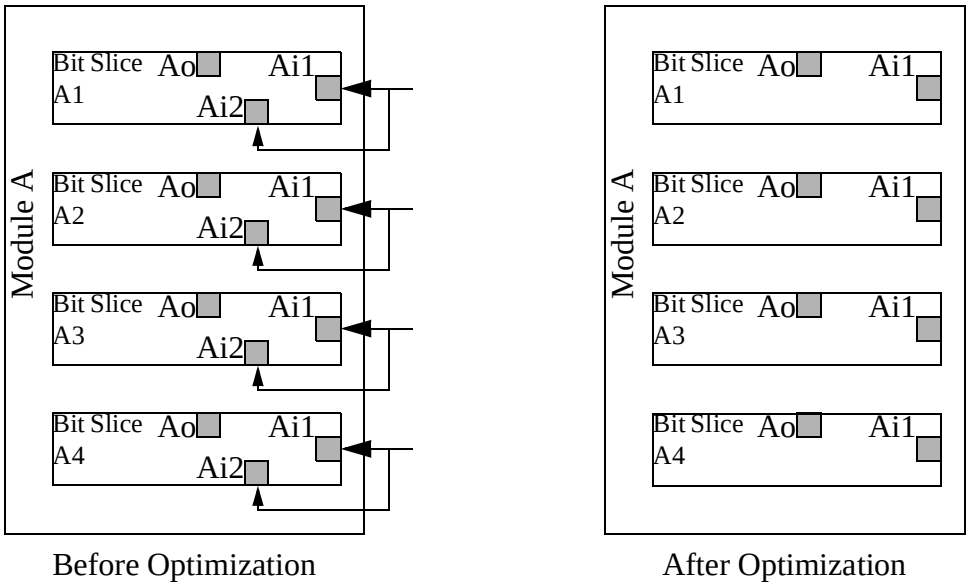



**Figure 12: Input Optimization Criteria B**




Before Optimization

After Optimization

**Figure 13: Input Optimization Criteria C**

## 5. Datapath-Oriented Packing

The input to our packing tool is the output of the synthesis tool. The final structural VHDL description produced by the synthesis tool consists of LUTs and DFFs. Datapath LUTs and DFFs are grouped into modules. Each module is divided into **m** identical bit slices, where **m** is equal to the number of subclusters in a cluster. Random logic LUTs and DFFs are not grouped into any module.

Our packing tool packs these LUTs and DFFs into clusters. We first pack LUTs and DFFs both inside and outside the modules into BLEs using an algorithm similar to the one presented in [6].

A normal packing tool packs BLEs into clusters one BLE at a time. Our cluster, however, is **m** bits wide. When dealing with datapath modules, our packer packs **m** identical BLEs into one cluster at a time. The packing algorithm is a modified T-Vpack algorithm [6] and has three stages.

First, we create a graph consists of large nodes and small nodes connected by nets representing signals. Each large node represents **m** identical BLEs, each from a distinct bit slice in a datapath module. Each small node represents a single BLE that does not belong to any datapath.

Then, large nodes are packed into clusters. Clusters are created one at a time and packed until the cluster has no room left for any unpacked large nodes. Throughout the packing process, each BLE in a large node is associated with a unique subcluster in the target cluster. The large node can be packed into the target cluster if each individual BLE can be packed into its corresponding subcluster.

Typically, there are many large nodes that can be packed into a target cluster. We choose one with the highest attraction to the cluster. To compute the attraction of a large node to a cluster, we first compute the attraction between each BLE in the node with its corresponding subcluster using an algorithm similar to the one used in T-Vpack. The attraction of a large node to a cluster is then set to be the maximum attraction of all of its BLEs to their subclusters.

In the final stage, BLEs outside the modules are packed into subclusters using the T-Vpack algorithm. New subclusters created at this stage will be grouped into clusters during placement.

Our packing algorithm achieves high BLE utilization. We packed our datapath benchmarks into clusters. Each cluster contains 4 subclusters each of which has 10 inputs and four 4-input LUTs. On average, we achieve BLE utilization of 97%. The detailed utilization results are listed in Table 1. Column one lists the name of each benchmark circuit. Column two lists the number of BLEs in each circuit. Column three lists the number of clusters datapath BLEs are packed into. Each cluster contains 16 BLEs. Column four lists the number of subclusters random-logic BLEs are packed into. Each subcluster contains 4 BLEs. As previously stated, these subclusters remain unpacked and we are working on placement tools that will group these subclusters into clusters in the placement stage. Finally column five lists the BLE utilization over both clusters and unclustered subclusters for each circuit.

**Table 1: Utilization for Datapath Oriented Packing**

| | BLE Count | Clusters | Unclustered Subclusters | BLE Utilization |
|---|---|---|---|---|
| dcu_dpath | 966 | 57 | 15 | 99% |
| ex_dpath | 2649 | 161 | 37 | 97% |

**Table 1: Utilization for Datapath Oriented Packing**

| | BLE Count | Clusters | Unclustered Subclusters | BLE Utilization |
|---|---|---|---|---|
| icu_dpath | 3245 | 208 | 23 | 95% |
| imdr_dpath | 1255 | 76 | 22 | 96% |
| pipe_dpath | 473 | 29 | 3 | 99% |
| smu_dpath | 557 | 31 | 20 | 97% |
| ucode_dat | 1304 | 77 | 20 | 99% |
| ucode_reg | 84 | 5 | 1 | 100% |
| code_seq_dp | 368 | 19 | 17 | 98% |
| exponent_dp | 517 | 23 | 44 | 95% |
| incmod | 867 | 49 | 21 | 99% |
| mantissa_dp | 942 | 55 | 31 | 94% |
| multmod_dp | 1634 | 85 | 74 | 99% |
| prils_dp | 393 | 20 | 20 | 98% |
| rsadd_dp | 313 | 18 | 10 | 95% |
| Total | 15567 | 913 | 358 | 97% |

## 6. Experiment Results

In this section, we present data collected on fifteen datapath benchmarks after synthesis and packing. Table 2 summarizes the LUT and DFF inflation of each benchmark. Each inflation figure is calculated by comparing the structural synthesis with the best flat synthesis as defined in Section 4. Column two and three list the LUT and DFF count from the best flat synthesis, respectively. Column four and five list the LUT and DFF count from the structural synthesis, respectively. The inflation figures for LUTs and DFFs are listed in column six and seven, respectively. The average LUT inflation is 3.2% and the average DFF inflation is 0.0%. These numbers show that structural synthesis does not significantly increase the LUT and DFF counts for these benchmarks.

**Table 2: LUT and DFF Inflation after Structural Synthesis**

| | Best Flat Synthesis | | Structural Synthesis | | Inflation | |
|---|---|---|---|---|---|---|
| | LUT Count | DFF Count | LUT Count | DFF Count | LUT | DFF |
| dcu_dpath | 960 | 288 | 966 | 288 | 0.63% | 0.0% |
| ex_dpath | 2530 | 364 | 2553 | 364 | 0.91% | 0.0% |
| icu_dpath | 3120 | 355 | 3235 | 355 | 3.7% | 0.0% |
| imdr_dpath | 1182 | 170 | 1218 | 170 | 3.1% | 0.0% |
| pipe_dpath | 443 | 218 | 471 | 218 | 6.3% | 0.0% |
| smu_dpath | 490 | 190 | 493 | 190 | 0.61% | 0.0% |
| ucode_dat | 1243 | 224 | 1304 | 224 | 4.9% | 0.0% |
| ucode_reg | 78 | 74 | 84 | 76 | 5.1% | 0.0% |
| code_seq_dp | 218 | 216 | 223 | 216 | 2.3% | 0.0% |
| exponent_dp | 477 | 64 | 501 | 64 | 5.0% | 0.0% |
| incmod | 779 | 72 | 867 | 72 | 11% | 0.0% |
| mantissa_dp | 846 | 192 | 878 | 192 | 3.8% | 0.0% |
| multmod_dp | 1558 | 193 | 1634 | 193 | 4.9% | 0.0% |
| prils_dp | 377 | 0 | 388 | 0 | 2.9% | 0.0% |
| rsadd_dp | 346 | 0 | 305 | 0 | -12% | 0.0% |
| Total | 14647 | 2620 | 15118 | 2620 | 3.2% | 0.0% |

The next two tables show two major types of nets that exist in datapath benchmarks after packing. A two terminal bus is defined as an **m** bit wide bus (4 in this case) going from a single cluster to another with each bit generated by a distinct subcluster. They can be efficiently routed by the coarse grain routing channels in our proposed architecture. On average 48% of two terminal nets in these benchmarks can be grouped into 4 bit wide buses. The details for each benchmark are summarized in Table 3. In the table, column two lists the total number of two terminal nets in each circuit. Column three lists the total number of two terminal nets that belong to 4 bit wide two terminal buses. Finally column four lists the net count in column three as a percentage of the total

two terminal nets.

**Table 3: Percentage of Two Terminal Nets that are 4 Bit Wide Buses**

| | Total Two Terminal Nets | 4 Bit Wide Two Terminal Buses | |
|---|---|---|---|
| | | Net Count | as a Percentage of Total Two Terminal Nets |
| dcu_dpath | 2232 | 1087 | 49% |
| ex_dpath | 6547 | 3411 | 52% |
| icu_dpath | 8047 | 3782 | 47% |
| imdr_dpath | 3100 | 1547 | 50% |
| pipe_dpath | 1049 | 500 | 48% |
| smu_dpath | 1167 | 564 | 48% |
| ucode_data | 3143 | 1631 | 52% |
| ucode_reg | 194 | 140 | 72% |
| code_seq_dp | 799 | 464 | 58% |
| exponent_dp | 1362 | 436 | 32% |
| incmod | 2013 | 843 | 42% |
| mantissa_dp | 2533 | 1196 | 47% |
| multmod_dp | 3380 | 1332 | 39% |
| prils_dp | 864 | 352 | 41% |
| rsadd_dp | 722 | 372 | 52% |
| Total | 37152 | 17657 | 48% |

A control net is a single net that enters a cluster and fans out to all **m** subclusters (4 in this case). They can be efficiently distributed by our special control signal distribution network inside clusters. The control nets on average consist of 35% of the total two terminal nets in these benchmarks. The details for each benchmark is shown in Table 4. In the table, column two lists the total number of two terminal nets in each circuit. Column three lists the total number of two terminal nets that belong to 4 bit fan out control signals. Finally column four lists the net count in column

three as a percentage of the total two terminal nets.

**Table 4: Percentage of Two Terminal Nets that are 4 Bit Fan Out Control Signals**

| | Total Two Terminal Nets | 4 Bit Fan Out in Cluster | |
|---|---|---|---|
| | | Net Count | as a Percentage of Total Two Terminal Nets |
| dcu_dpath | 2232 | 964 | 43% |
| ex_dpath | 6547 | 2572 | 39% |
| icu_dpath | 8047 | 2860 | 36% |
| imdr_dpath | 3100 | 1108 | 36% |
| pipe_dpath | 1049 | 440 | 42% |
| smu_dpath | 1167 | 296 | 25% |
| ucode_data | 3143 | 1304 | 41% |
| ucode_reg | 194 | 40 | 21% |
| code_seq_dp | 799 | 144 | 18% |
| exponent_dp | 1362 | 312 | 23% |
| incmod | 2013 | 670 | 33% |
| mantissa_dp | 2533 | 900 | 36% |
| multmod_dp | 3380 | 848 | 25% |
| prils_dp | 864 | 276 | 32% |
| rsadd_dp | 722 | 196 | 27% |
| Total | 37152 | 12928 | 35% |

## 7. Conclusion and Future Work

This report summarized my major Ph.D. research results in the academic year 2000–2001. We discussed the current upward trend of implement datapath application on FPGAs and how CAD and FPGA architectures can be designed to achieve area savings by exploring datapath regularity. We proposed a datapath FPGA architecture with a two level clustering hierarchy, coarse grain routing channels, and special control signal distribution networks. We also discussed our empiri-

cal methodology of measuring the efficiency of our proposed architecture.

CAD tools needed by the empirical study are currently being built. Two tools, the datapath-oriented structural synthesizer and the datapath-oriented packer, have been completed and their algorithms were discussed in this report. We also measured the regularity of fifteen benchmark circuits after packing. We found that there is high degree of regularity in these packed benchmarks, with 48% of two terminal nets that can be grouped into 4 bit wide buses and 35% of two terminal nets from control signals with at least 4 bit fan out. There is very little overlap between these two types of two terminal nets.

**Future Work**

Finally there are two major pieces of work remaining in my thesis. First, we need to gather more datapath benchmarks. All current benchmarks come from the Pico-Java processor. We want to add new circuits from other datapath applications to increase diversity. Preferably, we will add circuits form DSP and ASIC applications.

Second, we need to finish and implement datapath-oriented place and route tools. To fully understand the impact of our architectural changes on routing track utilization, we need to place and route the benchmark circuits. A specialized router is needed due to the inclusion of the coarse grain routing networks, the specialized control signal distribution networks and the unique connection boxes in our architecture. Placement tools also will be modified to take advantage of the regularity of datapath circuits. When these goals are completed, our proposed FPGA architecture can be further studied.

## 8. Bibliography:

[1] Andreas Koch, “Structured Design Implementation — A Strategy for Implementing Regular Datapaths on FPGAs”, *Proceedings of the 1996 ACM Fourth International Symposium on Field-Programmable Gate Arrays,* 1996, Pages 151–157.

[2] A. R. Naseer, M. Balakrishnan, Anshul Kumar, “An Efficient Technique for Mapping RTL Structures onto FPGAs”, *Proceedings of the Fourth International Workshop on Field Programmable Logic and Applications,* September 1994, Pages 99–110.

[3] Thomas Kutzschebauck, Leon Stok, “Regularity Driven Logic Synthesis”, *Proceedings of IEEE/ACM International Conference on Computer Aided Design,* 2000, Pages 439–446.

[4] Don Cherepacha, David Lewis, “DP-FPGA: An FPGA Architecture Optimized for Datapaths”, *VLSI Design 1996,* 1996, Pages 329–343.

[5] Andy Gean Ye, “Ph.D. Thesis Proposal: Routing Architecture and Place and Route Tools for DP-FPGA”, University of Toronto Technical Report, June. 2000.

[6] Vaughn Betz, Jonathan Rose, Alexander Marquardt, *Architecture and CAD for Deep-Submicron FPGAs,* Kluwer Academic Publishers, 1999.

[7] *Pico-Java Processor Design Documentation,* Sun Microsystems Inc., 1999.

[8] *Synopsys Design Compiler Manual*, Synopsys Inc., 1999.

[9] Alan Marshall, Jean Vuillemin, Brad Hutchings, “A Reconfigurable Arithmetic Array for Multimedia Applications”, *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays,* February 1999, Pages 135–143.