

Synthesizing Datapath Circuits for FPGAs with Emphasis on Area Minimization

Andy Ye, Jonathan Rose, David Lewis

*Department of Electrical and Computer Engineering, University of Toronto
Toronto, Ontario, Canada M5S 3G4
{yeandy, jayar, lewis}@eecg.utoronto.ca*

Abstract

Large circuits, whether they are arithmetic, digital signal processing, switching, or processors, typically contain a greater portion of highly regular datapath logic. Datapath synthesis algorithms preserve these regular structures, so they can be exploited by packing, placement, and routing tools for speed or density. Typical datapath synthesis algorithms, however, sacrifice area to gain regularity. Current algorithms can have as much as 30% to 40% area inflation when compared with traditional flat synthesis algorithms. This paper describes a datapath synthesis algorithm with very low area overhead, which is an enhancement to the module compaction algorithm proposed in [8]. We propose two word-level optimizations — multiplexer tree collapsing and operation reordering. They reduce the area inflation to 3%–8% as compared with flat synthesis. Our synthesis results also retain significant amount of regularity from the original designs.

1. Introduction

As FPGAs are used to implement ever-larger applications, it has become compelling to complement the traditional flat synthesis technology with more advanced datapath synthesis techniques. Although flat synthesis is ideal for small control-logic type circuits, it is not efficient for larger circuits, which typically contain a greater portion of datapath logic [11]. Whether it is arithmetic, digital signal processing, switching, or processors, datapath logic has highly regular structures. These structures are usually destroyed during flat synthesis. Datapath synthesis algorithms, on the other hand, preserve the regularity, so it can be exploited by packing, placement, and routing tools to achieve greater speed or density. By preserving regularity, datapath synthesis also preserves carry chains, which are specially supported by many commercial FPGAs. From a user perspective, FPGA users are accustomed to fully automated design flows. Since datapath synthesis significantly

increases the level of automation for datapath design, it is particularly suitable for FPGAs. For these reasons, there has been increased interest in implementing efficient datapath synthesis for FPGAs.

Previous studies, [4] [7] [8] [9] [12] [13] [16], have shown that the logic density of FPGAs can be substantially increased by exploiting regularity at the placement, routing, and architecture levels. However, there are no extensive studies focusing on the effects of datapath synthesis on FPGA area. Existing datapath synthesis techniques can be roughly classified into four categories: regularity preserving logic transformations [10] [11], hard boundary hierarchical synthesis (synthesis that performs optimizations strictly within user-defined module boundaries), template mapping [5] [6] [14] [15], and module compaction [8] [9]. Among these four synthesis techniques, only regularity preserving logic transformations do not incur significant area overhead [10]. However, their goal is to extract regularity from flattened datapath logic, rather than preserving a given hierarchy. As the result, their effectiveness is limited by the amount of regularity that can be discovered by the extraction process.

Hard boundary hierarchical, template mapping, and module compaction synthesis techniques, all make use of user-defined regularity information. We observed that two of the techniques — hard boundary hierarchical and template mapping — trade area for speed and regularity. For example, when we used Synopsys to synthesize a series of 15 datapath circuits using the hard boundary hierarchical methodology, we measured an average area increase of 38% compared to flat synthesis. For the template mapping methodology, the technique presented in [6] has an area increase of over 48%.

A module compaction algorithm was first proposed by Koch in [8] and [9]. The algorithm consists of two basic steps. First, it selectively merges user-defined datapath modules together. Then, each merged module is synthesized using regular flat synthesis techniques. Area results on only two circuits are published in [8] and [9]. One circuit has 712 4-LUTs and the other has 112 4-LUTs (post

datapath-synthesis area). The module compaction algorithm is shown to have an area inflation of -0.08% to 17% on the 712 LUT circuit and -16% for the 112 LUT circuit when mapped onto the Xilinx XC4000 architecture. In both cases, the module compaction algorithm uses the dedicated carry logic of XC4000, while the flat synthesis does not. One would expect a higher LUT count for the module compaction algorithm, if the dedicated carry logic is not used.

In this paper, we present an enhanced module compaction algorithm, which is augmented with two word-level transformations (by word-level we mean operations that optimize across multiple bits of datapath) — multiplexer tree collapsing and operation reordering. Currently, these two word-level transformations are performed manually, but the algorithms presented here can be easily automated.

Unlike Koch’s algorithm that uses placement information to selectively merge datapath modules, our module compaction algorithm does not require placement information. Instead, we merge modules together based on inter-module connectivity. As the result, our algorithm can be more easily integrated into existing CAD flows. Our enhanced algorithm is shown, empirically, to be able to preserve regularity while incurring an average area overhead of only 3%–8% versus flat synthesis.

In the next section, we describe our synthesis flow in detail. Section 3 presents the experimental results on a series of 15 benchmark circuits, comparing flat, hard-boundary hierarchical, and our enhanced module compaction synthesis. We also show that our synthesis maintains various regular structures from the input netlists. The effect of synthesis granularity on LUT count inflation is also presented. We conclude in Section 4.

2. Enhanced Module Compaction Algorithm

This section describes our datapath synthesis algorithm in detail. It first gives an overview of the input representation and the overall flow of our algorithm. It then discusses each synthesis/optimization step in detail.

2.1 Datapath Circuit Representation

The input to the synthesis algorithm is a netlist of datapath components, described in VHDL or Verilog, which we call the *top-level netlist*. All datapath components used in the netlist are instantiated from a predefined datapath component library. This library contains fundamental datapath building blocks such as multiplexers, adders/subtractors, shifters, comparators, and registers.

These datapath components are in turn composed of bit-level structures that we call *bit-slice netlists*. A bit-slice

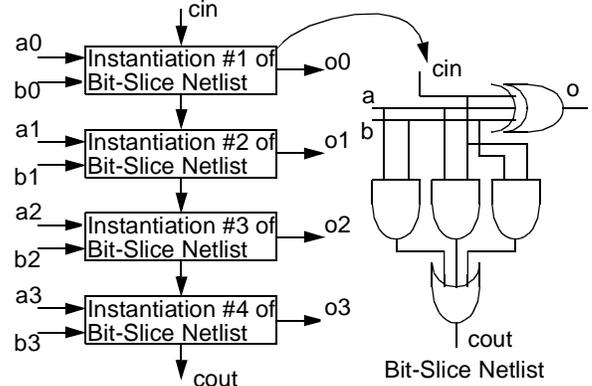


Figure 1: 4-bit Ripple Adder Datapath Component

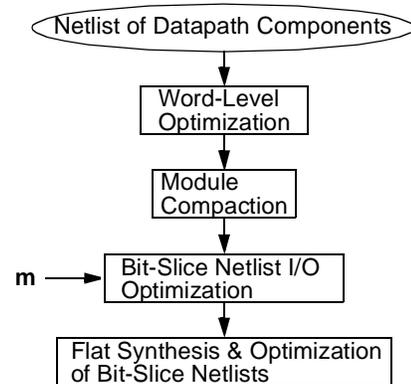


Figure 2: Overall Synthesis Flow

netlist is a netlist of logic gates, representing the function of a single bit-slice of a datapath. The bit-slice netlist is instantiated multiple times and all its instantiations are interconnected into another netlist that describes the function and structure of the datapath component. We call this netlist the *datapath component level netlist*.

The number of bit-slice netlist instantiations corresponds to the width of the datapath. All instantiations are assigned a unique *bit-slice number* from one to the width of the datapath with the least significant bit-slice labeled one.

An example of a datapath component is shown in Figure 1. This datapath component is a 4-bit ripple carry adder. The bit-slice netlist of this datapath component is a netlist of logic gates defining a full adder. This design is instantiated four times to form the 4-bit adder.

2.2 Synthesis Overview

The overall synthesis flow is shown in Figure 2. The flow consists of four major stages. First, the top-level netlist is passed through a three-stage optimization process

where new datapath components are created by transforming and merging bit-slice netlists. During the optimization process, some logic will be created which does not belong to specific bit-slices, for example, logic generating signals that fan out to several bit-slices. This is called *irregular logic* (to distinguish it from logic that fits nicely into a datapath) and is represented directly as logic gates in the top-level netlist. Each distinct optimization type is discussed in the sections below.

After the three-stage optimization, each bit-slice netlist is synthesized and mapped into 4-input lookup tables (4-LUTs) and D-type Flip-Flops without set and reset signals using a traditional flat synthesis algorithm. The irregular logic gates are also synthesized and mapped into LUTs independently from datapath components using the same flat synthesis algorithm.

2.3 Word-Level Optimization

The first set of optimizations that we perform are word-level optimizations. Two types of word-level transformations are performed. One is used to extract common sub-expressions across bit-slice boundaries. The other uses operation reordering to reduce area. Currently, these two optimizations are performed manually. Their algorithms, which are suitable for automation, are presented here.

Each datapath component represents a set of arithmetic operations. In a top-level netlist, datapath components are connected together to form mathematical functions. Each of these functions has multiple bit outputs, where the output bits can be individually described using logic expressions. Often, common sub-expressions exist across these logic expressions. More precisely, let both x ($[x_0, x_1, \dots, x_n]$) and y ($[y_0, y_1, \dots, y_n]$) be bit vectors of width n . Let $y = f(x)$ be a mathematical function of x . Each individual bit of y can be expressed in terms of bits of x as follows:

$$\begin{aligned} y_0 &= f_0(x_0, x_1, \dots, x_n) \\ y_1 &= f_1(x_0, x_1, \dots, x_n) \\ &\dots \\ y_n &= f_n(x_0, x_1, \dots, x_n) \end{aligned}$$

There can exist a function $g(x_0, x_1, \dots, x_n)$, such that:

$$\begin{aligned} y_0 &= f'_0(g(x_0, x_1, \dots, x_n), x_0, x_1, \dots, x_n) \\ y_1 &= f'_1(g(x_0, x_1, \dots, x_n), x_0, x_1, \dots, x_n) \\ &\dots \\ y_n &= f'_n(g(x_0, x_1, \dots, x_n), x_0, x_1, \dots, x_n) \end{aligned}$$

We call $g(x)$ the common sub-expression of $f_0(x), f_1(x), \dots, f_n(x)$. The implementation area of mathematical functions can be reduced by properly discovering and extracting these common sub-expressions so that they are only implemented once.

In a flat synthesis process, common sub-expressions are extracted through logic transformations. This extraction process usually destroys the regularity of datapath circuits, since flat synthesis independently transforms logic expressions one bit at a time. We have found that many of these common sub-expressions can be discovered at the word-level. Furthermore, datapath regularity can easily be preserved by extracting these common sub-expressions at the word-level where datapath structures remain clearly identifiable.

For our benchmarks, the most effective word-level transformation that extracts common sub-expressions is multiplexer tree collapsing. In a multiplexer tree, the multiplexers, their data inputs, outputs, and the interconnection signals form a tree topology. Each node of the tree, which has multiple inputs and a single output, represents a multiplexer. Each input of a node corresponds to a multiplexer data input. The output of a node corresponds to a multiplexer output. An edge in the graph represents a net connecting a multiplexer output to a multiplexer data input, a primary input, or the primary output of the multiplexer tree.

A multiplexer tree sometimes can be substituted by a single multiplexer, which requires much less logic to implement. An example is shown in Figure 3. Here the multiplexer tree in the left circuit is substituted by a single multiplexer in the right circuit. To implement the two multiplexers and the **and** gate in the left circuit we need two 4-input LUTs for every bit-slice as indicated by the shaded regions in the figure. To implement the multiplexer and the **and** gate in the right circuit, we need only one 4-input LUT for every bit-slice. The extra random logic in the right circuit is the common sub-expression extracted by the transformation. It usually is shared by several bit-slices, so its area cost is small in wide datapath circuits.

The algorithm used to collapse multiplexer trees is as follows: First we identify multiplexer trees in the top-level netlist. This is easy to perform since the functionality of each datapath component is known. We then identify the total number of unique data inputs to each tree. We replace each tree by a single multiplexer whose width is equal to the number of unique data inputs of the tree. Each input of the new multiplexer is connected to a unique multiplexer tree primary data input. The output of the new multiplexer is connected to the primary output of the tree. Finally, the select signal of the new multiplexer is generated using the select signals of the original multiplexer tree. If the replacement reduces the area cost, it is retained. Otherwise, the replacement is rejected.

A second word-level transformation that we perform uses operation reordering to reduce area. In particular, the optimization reorders result selections into operand selections. Arithmetic operators such as multiplications are, in

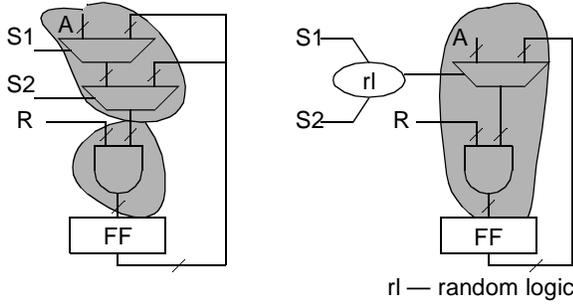


Figure 3: Mux Tree Collapsing Example

general, much more expensive than multiplexers. In the event that several identical operations are performed on independent data sets and only one result is used, it usually is much cheaper to preselect the input data than to perform all operations in parallel and select the final results.

An example is shown in Figure 4. Here the result of two addition operations is selected by a 2:1 mux. The operation can be more efficiently performed by preselecting adder inputs and using a single adder instead of two. Before optimization, five 4-input LUTs are needed to implement the function. After optimization, only four 4-input LUTs are needed to implement the same function. This optimization is not obvious at the bit-slice level. Since **cout0a** and **cout0b** appear to be two independent signals at this level. However, when viewed from the top-level netlist, the optimization is clearly identifiable.

More generally, assume that we have a function $y = f(x)$ where $x ([x_0, x_1, \dots, x_n])$ is an n bit wide bit vector and $y ([y_0, y_1, \dots, y_p])$ is a p bit wide bit vector. The function:

if $(s == 0)$ then
 $z = f(u)$
 else

$z = f(v)$

can be more cheaply implemented as:

if $(s == 0)$ then

$w = u$

else

$w = v$

$z = f(w)$

if $f(x)$ requires more area to implement than the extra multiplexers. Our algorithm searches for multiplexers whose data inputs are from the outputs of identical functions where these outputs have no other fan-outs. It then compares the area implementation cost of $f(x)$ with the area implementation cost of the multiplexers. If the area cost of $f(x)$ is greater than the area cost of the additional multiplexers, the transformation is performed.

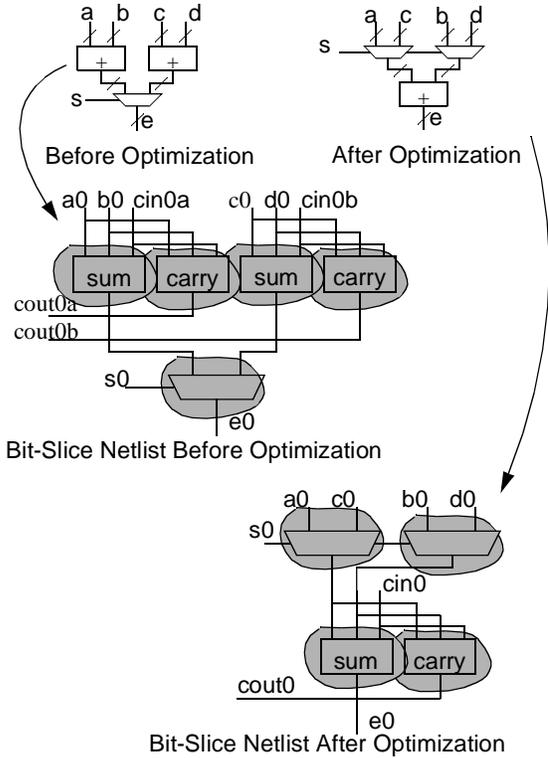


Figure 4: Result Selection to Operand Selection Transformation

2.4 Module Compaction

In the second stage of optimization, we perform module compaction. Here we iteratively merge two connected bit-slice netlists together to form a larger bit-slice netlist. Also, by creating larger bit-slice netlists, we create more optimization opportunities for the flat synthesis stage shown in Figure 2, where synthesis is restricted to the boundaries of bit-slice netlists. This merging process is similar to the module compaction algorithm proposed by Koch in [8]. Our algorithm differs from Koch's algorithm in its merging criteria; unlike Koch's algorithm, our algorithm does not depend on any placement information.

The basic merging operation is a pattern identification process. Two groups of bit-slices from two datapath modules are merged if the following conditions are met:

1. These two groups contain equal numbers of bit-slices.
2. All bit-slices in each group have consecutive bit-slice numbers as defined in Section 2.1.
3. All bit-slices in one group are identically connected to their corresponding bit-slices in the other group. Here we define two corresponding bit-slices to be bit-slices from two distinct groups, each with the same offset from the lowest bit-slice number in its group.

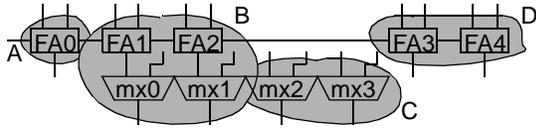


Figure 5: A Bit-Slice Netlist Merging Example

Each merging operation creates a new datapath component. The bit-slice netlist of the new component combines the two original bit-slice netlists. If a merging group does not include all the bit-slices of its datapath module, the remaining slices in the module are split into two modules — one module with all the bit-slices whose bit-slice numbers are smaller than the bit-slice numbers of the merging group, the other module with all the bit-slices whose bit-slice numbers are larger than the bit-slice numbers of the merging group.

An example of module compaction is shown in Figure 5. Here, we start with two modules. One module contains five slices of full adders labeled **FA0** to **FA4**. The other module contains four slices of single-bit 2:1 multiplexers labeled **mx0** to **mx3**. Based on the merging rules stated above, two full adders, **FA1** and **FA2** can be merged with two single-bit 2:1 multiplexers, **mx0** and **mx1**, to form a new datapath module. The remaining three full adders are broken into two new modules after merging. The four new modules created by the merging process is indicated in shades in the figure. They are labeled **A**, **B**, **C**, and **D**.

We impose two extra conditions to prevent a carry type signal from causing all bit-slices connected to it to be merged into a single module. For example, consider a second merging iteration on the circuit of Figure 5 after the initial merging described above. Module **A** will be qualified to be merged with the first slice of module **B** since they are connected by the carry signal. Then in the third merging iteration, module **A** and **B** will be completely merged into a single bit-slice. After two more iterations, the carry chain will cause **A**, **B**, and **D** in the figure to be merged into a single bit-slice, which completely destroys the regularity of our datapath.

To prevent this, first, we order merging operations, so that operations that will create the widest datapath components are performed first. Second, for every bit-slice netlist we define an *ancestors* field, which is a set of bit-slice netlists. Initially each bit-slice netlist has only itself in its *ancestors* set. When two bit-slice netlists are merged, the *ancestors* set of the new bit-slice netlist is the union of the *ancestors* sets of the two merging bit-slice netlists. If the intersection of the *ancestors* of two bit-slice netlists is not empty, these two bit-slice netlists cannot be merged together.

With the *ancestors* field, nothing can be merged during the second merging iteration in Figure 5, since all mergable

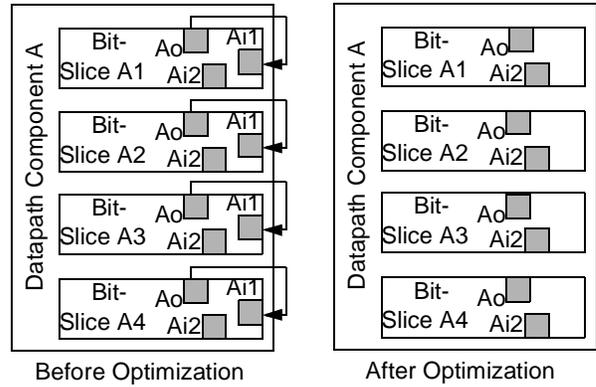


Figure 6: Feedback Absorption Example

module pairs, (**A**, **B**) and (**B**, **D**) share at least one common ancestor.

2.5 Bit-Slice Netlist I/O Optimization

Each bit-slice netlist has a set of predefined I/O signals that enter and exit the netlist. Depending on the usage of these signals, some of them can be eliminated and converted into internal signals of the netlist. Since each bit-slice netlist is flat synthesized in our synthesis flow, converting I/O signals into internal signals can reduce the implementation area of bit-slices by providing extra information to the flat synthesizer. In our optimization process, four types of bit-slice I/O signals are converted into internal signals of bit-slices. Each type is discussed below.

Before any I/O optimization is performed, each datapath component in the top-level netlist is first divided into **m**-bit wide subcomponents, where **m** is specified by the user. Each subcomponent is a self-contained datapath component with its own bit-slice netlist definition and a netlist of **m** instantiations of the bit-slice netlist. The division starts from the least significant bit of each datapath component and groups adjacent **m** bit-slices into a subcomponent. If the width of the datapath component is not an integer multiple of **m**, the subcomponent containing the most significant bits will be less than **m**-bits wide. We call **m** the granularity of the synthesis flow. A larger **m** preserves more datapath regularity at the expense of increased area, while a smaller **m** decreases area at the expense of preserving less datapath regularity. After division, each original datapath component in the top-level netlist is substituted by its corresponding subcomponents.

The first type of I/O optimization is constant absorption. When an input of a bit-slice netlist is always connected to the same constant value (either zero or one) for all instantiations of the netlist in a datapath component, we convert this input signal into a constant internal signal of the netlist.

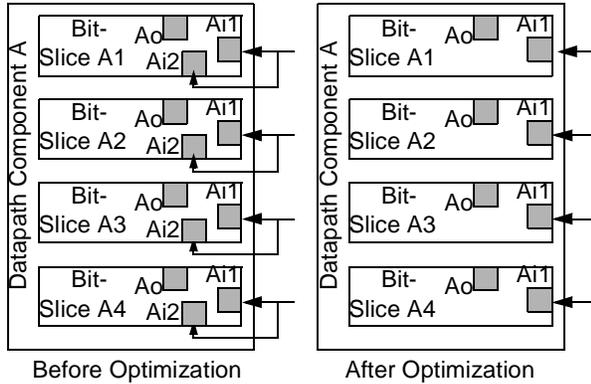


Figure 7: Duplicated Input Absorption

The second type of I/O optimization is feedback absorption. When a connection exists between a bit-slice netlist input and a bit-slice netlist output for all instantiations of the netlist, we convert this input signal into an internal signal and reconnect it to the corresponding output inside the netlist.

An example of feedback absorption is shown in Figure 6. Here **Datapath Component A** consists of four bit-slices, which are all instances of the same bit-slice netlist. Since each of the slice inputs labeled **Ai1** is connected to a corresponding slice output labeled **Ao** from the same slice, **Ai1** is eliminated as an input of the bit-slice netlist and is converted to an internal signal. **Ai1** is reconnected to **Ao** inside the netlist.

The third type of I/O optimization is duplicated input absorption. When two bit-slice netlist inputs are connected together for all instantiations of the design, we convert one of the input signals into an internal signal and reconnect it to the other input signal inside the netlist.

An example of duplicated input absorption is shown in Figure 7. As before, **Datapath Component A** consists of four bit-slices, which are all instances of the same bit-slice netlist. Since each of the slice inputs labeled **Ai1** is always connected to a corresponding slice input labeled **Ai2** in the same slice, **Ai2** is eliminated as an input of the bit-slice netlist and is converted to an internal signal. **Ai2** is reconnected to **Ai1** inside the netlist.

The last type of I/O optimization that we perform is unused output elimination. When a bit-slice netlist output does not connect to any other signals in all instantiations of the bit-slice netlist, this output signal is converted into an internal signal of the bit-slice netlist.

3. Experimental Results

In this section, we present experimental results of applying the enhanced module compaction synthesis on

fifteen datapath benchmarks. These fifteen circuits are from the Pico-Java processor [1]. Note that the word-level optimizations, described in Section 2.3, were performed manually. The other optimizations were done by automated algorithms implemented in the C-language. We used the Synopsys Design Compiler and FPGA Compiler [2] to perform flat synthesis. Unless specified otherwise, all the data presented here are synthesized using a granularity value (**m**), as defined in Section 2.5, of 4.

Table 1: LUT & Flip-flop Inflation for Structured Synthesis

	Best Flat Synthesis LUT and flip-flop Count		Inflation of Structured Synthesis			
			Hard-Boundary Hierarchical		With Cross Boundary Optimization	
	LUT	FF	LUT	FF	LUT	FF
dcu_dpath	960	288	24%	0.0%	0.63%	0.0%
ex_dpath	2530	364	39%	0.0%	0.91%	0.0%
icu_dpath	3120	355	42%	0.28%	3.7%	0.0%
imdr_dpath	1182	170	31%	0.0%	3.1%	0.0%
pipe_dpath	443	218	24%	0.92%	6.3%	0.0%
smu_dpath	490	190	16%	0.0%	0.61%	0.0%
ucode_dat	1243	224	9.6%	0.0%	4.9%	0.0%
ucode_reg	78	74	121%	8.1%	5.1%	0.0%
code_seq_dp	218	216	68%	4.6%	2.3%	0.0%
exponent_dp	477	64	52%	0.0%	5.0%	0.0%
incmod	779	72	55%	0.0%	11%	0.0%
mantissa_dp	846	192	38%	0.0%	3.8%	0.0%
multmod_dp	1558	193	46%	0.0%	4.9%	0.0%
prils_dp	377	0	79%	0.0%	2.9%	0.0%
rsadd_dp	346	0	52%	0.0%	-12%	0.0%
Total	14647	2620	38%	0.73%	3.2%	0.0%

For every benchmark circuit, we compared the final LUT and flip-flop count of our enhanced module compaction synthesis with the counts achieved by Synopsys flat synthesis. In order to assure that the best achievable flat synthesis results are used to compare with our synthesis, we use the best flat synthesis result from two flows: the flat synthesized input netlist, and the flat synthesized output netlist of our enhanced module compaction synthesis. In some cases, one flat synthesis flow offers slightly better results than the other.

Table 1 summarizes the LUT and flip-flop inflation of each benchmark for flat synthesis, hard-boundary hierarchical synthesis, and our new enhanced module compaction synthesis. Each inflation figure is calculated by comparing the enhanced module compaction synthesis with the best flat synthesis. The formula,

$inflation = \frac{DA}{FA} - 1$, is used to calculate the inflation for

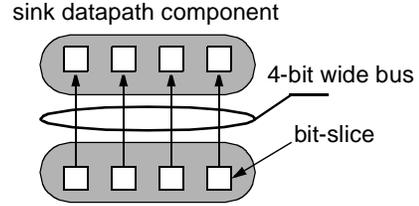
both LUTs and flip-flops. In the formula, DA represents the datapath-oriented synthesis area; FA represents the flat synthesis area.

Table 2: Percentage of Two Terminal Connections that are 4 Bit Wide Busses and Percentage of Two Terminal Connections that are Fan-Out Four Control Signals

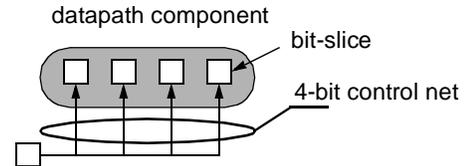
	Total Two Terminal Conn.	Percentage of Two Terminal Conn. that are 4 Bit Wide Busses	Percentage of Two Terminal Conn. that are Fan-Out Four Control Signals
dcu_dpath	2232	49%	43%
ex_dpath	6547	52%	39%
icu_dpath	8047	47%	36%
imdr_dpath	3100	50%	36%
pipe_dpath	1049	48%	42%
smu_dpath	1167	48%	25%
ucode_dat	3143	52%	41%
ucode_reg	194	72%	21%
code_seq_dp	799	58%	18%
exponent_dp	1362	32%	23%
incmod	2013	42%	33%
mantissa_dp	2533	47%	36%
multmod_dp	3380	39%	25%
prils_dp	864	41%	32%
rsadd_dp	722	52%	27%
Total	37152	48%	35%

Column one of the table lists the name of each benchmark circuit. Columns two and three give the LUT and flip-flop count of each circuit from the best flat synthesis. Columns four and five give the inflation figures of hard-boundary hierarchical synthesis. Here, synthesis is performed without any of the optimizations described in Section 2. The inflation figures of enhanced module compaction with full optimization are listed in columns six and seven. The average LUT inflation without optimization is 38% and the average flip-flop inflation is 0.73%. With the optimizations, the average LUT inflation is reduced to 3.2% and the flip-flop inflation is zero. These numbers show that our algorithm does not significantly increase the LUT and flip-flop count for these benchmarks and is much more area efficient than hard-boundary hierarchical synthesis. For the circuit, `rsadd_dp`, our synthesis even discovered more optimizations than flat synthesis, resulting in much smaller area.

We now present measurements of various aspects of the datapath regularity of the circuits after enhanced module



source datapath component
Figure 8: 4-bit Wide Bus Topology



control logic (can be either datapath or random)
Figure 9: 4-bit Control Net Topology

compaction as a mean of showing how much regularity is preserved by the synthesis. The granularity of the synthesis, m , is again set at 4. Higher granularities typically result in higher regularity.

First we measure the number of LUTs that remain in datapath components. Several optimizations described in Section 2 create irregular logic during the optimization process. In our representation, regular logic is represented by LUTs that belong to datapath components; and irregular logic is represented by LUTs that do not belong to any datapath component. Before synthesis, near all logic in our benchmark is regular.

After synthesis, 94% of the LUTs remain in datapath components, while only 6% of the logic resides in irregular logic. This shows that our synthesis flow preserves regularity for logic blocks. We also measured the regularity of nets after synthesis. Table 2 shows two major types of two terminal connections exist in datapath benchmarks after synthesis - bus and control signals. The first column of Table 2 lists the name of each benchmark circuit. The second column lists the total number of two terminal connections in each circuit.

A two terminal bus is defined as an m -bit wide bus (4 in this table) that connects one datapath component to another and obeys the following two conditions: First, each bit of the bus must be generated by a distinct bit-slice in the source datapath component and absorbed by a distinct bit-slice in the sink datapath component. Second, the source bit-slice and the sink bit-slice must have the same bit-slice number. The topology of a 4-bit wide bus is shown in Figure 8. On average 48% of two terminal connections in these benchmarks can be grouped into 4-bit wide busses. The percentage number for each benchmark is summarized in column three of Table 2.

A control net is a single net that enters a datapath component and fans out to all m bit-slices (4 in this table). The topology of a 4-bit control net is shown in Figure 9. The control nets on average consist of 35% of the total two terminal connections in these benchmarks. The detailed percentage number for each benchmark is shown in column four of Table 2.

Overall, there are 83% of two terminal connections that belongs to either a bus or a control net. There are few two terminal connections that belong to both a bus and a control net at the same time.

Table 3: Granularity vs. LUT Count Inflation

m	1	4	8	12	16	20	24	28	32
Avg. LUT Inflation (%)	0.0	3.5	4.6	6.3	6.7	6.5	6.7	6.8	7.4

Finally, Table 3 presents LUT count inflation as a function of m . DFF count did not increase with increasing m . Here we see that the LUT inflation increases from 3.5% to 7.4% as m , the granularity of synthesis, is increased from 4 to 32. The cause of this increase is the less efficient I/O optimization as described in Section 2.5.

4. Conclusion

This paper presented an enhanced module compaction synthesis algorithm targeting FPGAs. We empirically demonstrated that our datapath-oriented synthesis is nearly as efficient as the regular flat synthesis. In terms of LUT count, our algorithm produces circuits on average with only 3%–8% LUT count inflation and no increase in register count. We also measured the regularity of the fifteen benchmark circuits. We found that there is a high degree of regularity in these synthesized benchmarks, with 48% of two terminal connections that can be grouped into 4-bit wide busses and 35% of two terminal connections from highly regular control signals with at least 4-bit fan-out.

References

- [1] Pico-Java Processor Design Documentation, Sun Microsystems Inc., 1999.
- [2] Synopsys Design Compiler Manual, Synopsys Inc., 1999.
- [3] Vaughn Betz, Jonathan Rose, Alexander Marquardt, Architecture and CAD for Deep-Submicron FPGAs, Kluwer Academic Publishers, 1999.
- [4] Don Cherepacha, David Lewis, “DP-FPGA: An FPGA Architecture Optimized for Datapaths”, Proceedings of Ninth International Conference on VLSI Design, Pages 329–343, 1996.
- [5] Timothy J. Callahan, Philip Chong, Andre DeHon, John Wawrzynek, “Fast Module Mapping and Placement for Datapaths in FPGAs”, Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays, Pages 123–132, 1998.
- [6] Miguel R. Corazao, Marwan A. Khalaf, Miodrag Potkonjak, Jan M. Rabaey, “Performance Optimization Using Template Mapping for Datapath-Intensive High-Level Synthesis”, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Pages 877–888, August 1996.
- [7] Scott Hauck, Thomas W. Fry, Matthew M. Hosler, Jeffrey P. Kao, “The Chimaera Reconfigurable Functional Unit”, IEEE Symposium on FPGAs for Custom Computer Machines, Pages 87–96, 1997.
- [8] 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, Pages 151–157, 1996.
- [9] Andreas Koch, “Module Compaction in FPGA-based Regular Datapaths”, Proceedings of the 33rd Design Automation Conference, Pages 471–476, 1996.
- [10] Thomas Kutzschebauch, Leon Stok, “Regularity Driven Logic Synthesis”, Proceedings of IEEE/ACM International Conference on Computer Aided Design, Pages 439–446, 2000.
- [11] Thomas Kutzschebauch, “Efficient Logic Optimization Using Regularity Extraction”, Proceedings of 2000 International Conference on Computer Design, Pages 487–493, 2000.
- [12] 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, Pages 135–143, February 1999.
- [13] Ethan Mirsky, Andre DeHon, “MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources”, Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines, Pages 157–166, April 1996.
- [14] A. R. Naseer, M. Balakrishnan, Anshul Kumar, “FAST: FPGA Targeted RTL Structure Synthesis Technique”, Proceedings of the Seventh International Conference on VLSI Design, Pages 21–24, 1994.
- [15] A. R. Naseer, M. Balakrishnan, Anshul Kumar, “Direct Mapping of RTL Structures onto LUT-Based FPGAs”, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Pages 624–631, July 1998.
- [16] Satnam Singh, “Death of the RLOC”, 2000 IEEE Symposium on Field-Programmable Custom Computing Machines, Pages 145–152, April 2000.
- [17] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, Anant Agarwal, “Baring It All to Software: Raw Machines”, IEEE Computers, Pages 86–93, September 1997.