Datapath Synthesis Option in PKS

1 Introduction .......................................................................................................................... 2
2 Basic Technical Background .................................................................................................. 2
3 Datapath Synthesis Features ............................................................................................... 4
4 The Datapath Synthesis Design Flow .................................................................................. 5
5 Running the Datapath Synthesis Option ............................................................................... 5
6 Reference: .............................................................................................................................. 8
1 Introduction

Datapath synthesis starts from the RTL code level. The input is RTL code that infers datapath operators. Both datapath logic and control logic of the design are described in the same piece of RTL code, which can be written in either Verilog (IEEE Std 1364) or VHDL (IEEE Std 1076). Ambit® BuildGates® synthesis reads in the RTL code and synthesizes it down to gates.

The datapath operators that are recognized by the software are: +, -, unary minus, *, ==, !=, <, <=, >, >=, <<, >>, <<< (Verilog 2000 only), >>> (Verilog 2000 only), left rotate (VHDL only), right rotate (VHDL only), and ABS (VHDL only).

The major characteristics of the datapath synthesis option are:

- It has known-good datapath structures built into the tool.
- It combines datapath synthesis and mainstream logic synthesis in one tool.
- It reads industry standard design description languages.
- It leverages industry standard place-and-route tools for layout generation.
- It works in the industry standard ASIC design flow.
- It minimizes manual effort needed to get the job done.

Considering all variations of datapath methodologies and various views of the datapath problem, this datapath synthesis option is not meant to incorporate everything.

- It does not do bit-slicing.
- It does not do layout generation or regular placement (tiling).
- It does not do algorithm refinement or behavioral synthesis.

Instead, it focuses on operator-level optimization, built-in datapath knowledge, standard ASIC flow, and automation.

2 Basic Technical Background

Adder Architectures
When implementing an adder, a synthesis tool does not treat it as one big truth table and rely on logic synthesis and logic optimization to implement that truth table. Instead, the tool usually employs a known, pre-defined scheme to compose the adder. Such a scheme is known as the architecture of an adder. There are various kinds of adder architectures. For example, the ripple adder is well known to be simple and small; the carry-lookahead adder is known to be faster but bigger.

Multiplier at the Gate Level
The gate-level implementation of a multiplier often includes a section that generates partial products, a section that adds up the partial products but leaves them in carriesave form, and a section that resolves the final carry propagation.
**Booth Encoding**
A multiplication is the multiplicand multiplied by the multiplier. In its simplest form, a partial product is the multiplicand multiplied by one of the bits in the multiplier. Booth encoding is one of the ways to implement the partial product generator. It looks at multiple bits in the multiplier while generating each partial product. At the cost of a bigger/slower partial product generator, this leads to a smaller number of partial products. Depending on the width of the multiplicand and the multiplier, as well as the underlying technology library, Booth encoding may make the multiplier faster and/or smaller.

**Carrysave Arithmetic**
While adding up a set of numbers, the most straightforward way is to employ an adder tree. Each adder consumes two numbers and produces one. The adder at tip of the tree generates the final sum. Alternatively, the carrysave technique can be applied to greatly improve both timing and area.

**Carry-Propagate Adder**
When a sum in the carrysave form needs to be transformed into one number, a traditional adder is needed. This adder propagates the carries from the LSB (least significant bit) to the MSB (most significant bit), and it is often referred to as the carry-propagate adder. It can be implemented by adder architectures such as ripple adder, carry-lookahead adder, and so on. Timing analysis on a multiplier of a vector sum often identifies the carry-propagate adder as a significant portion of the critical path.

**Operator Merging**
By employing the carrysave technique, arithmetic operators can be merged to greatly improve timing and area.

Consider the expression, \( y = (a \times b) + (c \times d) \)

The above expression can be implemented using discrete operators, i.e., without operator merging. It takes two multipliers \((a*b, c*d)\) and one adder to implement this functionality. Traditionally, the synthesis tool works hard to optimize each of these discrete operators individually, without taking into account how they interact with each other. Each of these operators has a carry-propagate adder. There will be two carry-propagate adders on the critical path.

With operator merging, this implementation can be changed. The tool looks at the design at the operator level, and recognizes that this is a cluster of arithmetic operators that can be merged. Instead of implementing three discrete components, the tool merges them as one larger complex operator, and it optimizes the entire merged operator. By doing so, there is only one carry-propagate adder on the critical path.

Note: the merged operator is no longer a multiplier or an adder. It is a complex operator computing \( a \times b + c \times d \).

**Architecture Selection**
The best architecture for a datapath operator is a function of the design constraints and its surrounding logic. The choice should not be uniform among all operators since each operator has
its own unique surrounding. Manually selecting an architecture for each individual operator in the design is time consuming and error-prone. Architecture selection is best left to the software because it can perform accurate timing analysis on the fly and make precise decisions based on the on-the-fly delay calculations.

3 Datapath Synthesis Features

Datapath Partitioning
The RTL code fed into the tool describes both the control portion and the datapath portion of the design. Right after reading in the RTL code, the datapath synthesis option partitions the datapath portions of the design from the non-datapath portions of the design. The datapath portions of the design are synthesized using the datapath synthesis engine. The non-datapath portions of the design are synthesized using the traditional logic synthesis engine. Important Partitioning happens as an automatic process. No manual intervention is required.

Operator Merging
As long as the original functionality is not distorted, the datapath synthesis option merges operators to reduce the number of carry-propagate adders in the design in order to improve timing and area. While operator merging is done automatically, without manual intervention, the user can have control as well.

Implementation Selection
For each operator in the design, merged or isolated, the -datapath option selects the best architecture. Furthermore, the implementation of the selected architecture is fine-tuned to optimize the overall QOR. These selection and implementation decisions are a function of timing constraints, surrounding control logic, and the target technology library. For each kind of operator (adders, multipliers, shifters, etc.), knowledge of multiple architectures is built into the tool. However, for any such built-in architecture, there is no hard-coded implementation; there is no hard-coded assumption about the surrounding timing requirement; there is no assumption about any special datapath cells being available in the library. Everything is based on the actual timing information calculated on the fly.

AmbitWare Components
There are 19 pre-defined components that can be instantiated in the RTL code, using either Verilog or VHDL. Some of them are commonly used functionality that cannot be conveniently described in standard languages. Half of these components are arithmetic functions like pipelined multiplier, mac, square, vector sum, and so on. The other halves are logic functions like leading zero counter, encoder, decoder, rotate, and so on.

Supported Languages
The following languages are supported with the Datapath synthesis option:
- Verilog 1995
- Verilog 2000 (only syntax that is related to signed arithmetic, including signed data type, <<< operator, >>> operator, and type casting functions $signed() and $unsigned().
- VHDL 1987
- VHDL 1993
• Cadence Verilog simulation products (Verilog-XL, NC-Verilog) support signed signal types.

Designs Suitable for the Datapath Synthesis Option
Designs that are suitable for the datapath synthesis option include the following:
- Designs described in synthesizable RTL in Verilog/VHDL
- Designs that infer datapath operators in RTL

4 The Datapath Synthesis Design Flow

The Datapath synthesis takes place during the execution of the do_build_generic and do_optimize commands.

During do_build_generic:
- Datapath partitioning is performed and datapath partitions are identified.
- Operator merging is performed on each datapath partition.
- Initial datapath synthesis is performed for each datapath partition.

During do_optimize:
- Implementation selection is exercised for each datapath partition.
- Datapath synthesis and optimization of each datapath partition is performed iteratively.

5 Running the Datapath Synthesis Option

In this tutorial, we will use the same files that we used for the SP&R flow. We can then compare the results of non-datapath synthesis and datapath synthesis.

1. Invoke PKS shell using the datapath option from the unix shell.
   ```
   class3: /home/username/asicdemo/hdlsrc $ . cadasic
class3: /home/username/asicdemo/hdlsrc $ pks_shell -datapath -gui
   ```

2. Load the timing and physical libraries.
   ```
pks_shell[1]>read_alf /net/cadence2001/artisan_tsmc18/aci/sc/synopsys typical.alf
pks_shell[1]>read_lef /net/cadence2001/artisan_tsmc18/aci/sc/lef/tsmc18_4lm.lef
   ```

3. Read the design data.
   ```
pks_shell[1]>read_vhdl alu_rtl.vhd count5_rtl.vhd cpu_rtl.vhd decode_rtl.vhd reg8_rtl.vhd
   ```

4. Build the generic netlist.
   ```
   pks_shell[1]>do_build_generic
   ```
   Note: With the -datapath option, after do_build_generic, datapath blocks have mapped gates. This does not apply to control logic.

5. Generate an initial report showing the arithmetic resources in the design.
   ```
   pks_shell[1]>report_resources –hier
   ```
This command reports the following:

- The datapath partitions created during partitioning of the datapath and control elements.
- The clusters within the datapath partitions created by operator merging
- The initial architecture of each cluster

Using this command here helps examine datapath partitions and clusters.

The output is shown below:

| Report         | report_resources     |
|----------------+----------------------|
| Options        | -hier                |
| Date           | 20020506.164704      |
| Tool           | pks_shell            |
| Release        | v4.0-s008            |
| Version        | Apr 20 2001 04:20:50 |

| Current Module | cpu                  |

+---------------------------------------+
<table>
<thead>
<tr>
<th>Arithmetic Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>AWDP_ADD_partitions_0</td>
</tr>
<tr>
<td>AWDP_EQ_partitions_2</td>
</tr>
<tr>
<td>AWDP_EQ_partitions_4</td>
</tr>
<tr>
<td>AWDP_ADD_partitions_1</td>
</tr>
<tr>
<td>AWDP_EQ_partitions_7</td>
</tr>
<tr>
<td>AWDP_EQ_partitions_8</td>
</tr>
<tr>
<td>AWDP_EQ_partitions_0</td>
</tr>
</tbody>
</table>
6. Set the timing constraints. After building the generic netlist, you can set the timing constraints on the design. Open the TCL file, constraints.tcl and run it to set the constraints on the module.

7. Optimize the design.
   
   \texttt{pks\_shell[1]\textgreater do\_optimize -pks}
   
   Implementation selection takes place during this step.

8. Generate a second report showing the arithmetic resources in the design.
   
   \texttt{pks\_shell[1]\textgreater report\_resources –hier}
   
   Using the command, report\_resources after do\_optimize helps examine the selected architectures.
   
   The result is shown below. Notice the difference from the previous execution of this command.

   +---------------------------------------+  
   | Report         | report\_resources     |  
   +----------------+----------------------+  
   | Options        | -hier                |  
   +----------------+----------------------+  
   | Date           | 20020506.165844       |  
   | Tool           | pks\_shell            |  
   | Release        | v4.0-s008             |  
   | Version        | Apr 20 2001 04:20:50 |  
   +----------------+----------------------+  
   | Current Module | cpu                  |  
   +---------------------------------------+

   +-----------------------------------------------------------------------------------------------------+  
   |                                        Arithmetic Resources                                         |  
   +-----------------------------------------------------------------------------------------------------+  
   |    Module    |    File     | Cluster | Architecture | Operator | Line |    Output    |    Input     |  
   +--------------+-------------+---------+--------------+----------+------|--------------+--------------+  
   | AWDP\_ADD\_par| alu\_rtl.vhd |    1    | Ripple Adder |    +     |  44  |      8u      | 8ux8u        |  
   +-----------------------------------------------------------------------------------------------------+

   
   \texttt{pks\_shell[1]\textgreater do\_place –timing\_driven}
   
   \texttt{pks\_shell[1]\textgreater do\_xform\_optimize\_slack -pks}
Notice the difference in placements between the Non-Datpath option and Datapath Option.

Figure 1. Placement done WITHOUT Datapath Option showing the adder unit.

Figure 2. Placement done WITH Datapath Option showing the adder unit.

6 Reference: