SYNOPSYS®

Coding Guidelines for Datapath Synthesis

Reto Zimmermann – Synopsys

July 2005

Abstract

This document summarizes two classes of RTL coding guidelines for the synthesis of datapaths:

- Guidelines that help achieve functional correctness and intended behavior of arithmetic expressions in RTL code.
- Guidelines that help datapath synthesis to achieve best possible QoR (Quality of Results).

Datapath Synthesis

In order to write RTL code that gives best possible QoR during datapath synthesis, it is important to understand what datapath functionality current synthesizers can efficiently implement, how the datapath synthesis flow works and how coding can influence the effectiveness of datapath synthesis.

Supported Datapath Functionality

The most important technique to improve the performance of a datapath is to avoid expensive carry-propagations and instead to make use of redundant representations (like carry-save or partial-product) wherever possible. Other techniques include high-level arithmetic optimizations (for example, common-subexpression sharing and constant folding). These optimization techniques are most effective when the biggest and most complex possible datapath blocks are extracted from the RTL code. This is enabled by supporting the following functionality:

- **Sum-Of-Product (SOP):** Arbitrary sum-of-products (that is, multiple products and summands added together) can be implemented in one datapath block with one single carry-propagate final adder. Internal results are kept in redundant number representation (for example, carry-save) wherever possible.
  Example: "`z = a * b + c * d - 483 * e + f - g + 2918`".
- **Product-Of-Sum (POS):** Limited product-of-sums (a sum followed by a multiply) can be implemented in one datapath block with one single carry-propagate final adder (no carry-propagation before the multiply). The limitation is that only one multiplication operand can be in redundant (carry-save) format while the other operand has to be binary.
  Example: `"z = (a + b) * c" , "z = a * b * c"`.
- **Select-op:** Select-operations (selectors, operand-wide multiplexers) can be implemented as part of a datapath block on redundant internal results (without carry-propagation before the select-op).
  Example: "`z = (sign ? -(a * b) : (a * b)) + c`".
- **Comparison:** Comparisons can be implemented as part of a datapath block on redundant internal results (without carry-propagation before the comparison).
  Example: "`t1 = a + b; t2 = c * d; z = t1 > t2`" ('t1', 't2' in carry-save only if not truncated internally).

Synthesis Flow

1. **Datapath extraction:** Biggest possible datapath blocks are extracted from the RTL code.
2. **Datapath optimization:** High-level arithmetic optimizations are carried out on the extracted datapath blocks.
3. **Datapath generation:** Flexible, context-driven datapath generators implement optimized netlists for the datapath blocks under specified constraints, conditions, and libraries.

Coding Goals

You can optimize RTL code for datapath synthesis by achieving the following goals:

- Enable datapath extraction to extract biggest possible datapath blocks.
- Enable datapath optimization to effectively perform high-level arithmetic optimizations.
- Enable datapath generation to fully exploit the functionality it can implement.

The following guidelines help you to write code to achieve these goals.

General Guidelines

1. Signed Arithmetic

- **Rule:** Use type '`signed`' (VHDL, Verilog 2001) for *signed/2's complement arithmetic* (do not emulate signed arithmetic using unsigned operands/operations). Also, do not use the '`integer`' type except for constant values.
- **Rationale:** Better QoR for a signed datapath as compared to an unsigned datapath emulating signed behavior.
- **Example:** Signed multiplication (Verilog)

Bad QoR	Good QoR

`input  [7:0] a, b;`
`output [15:0] z;`
 	
`// a, b sign-extended to width of z`
`assign z = {{8{a[7]}}, a[7:0]} *`
`{{8{b[7]}}, b[7:0]};`
`// -> unsigned 16x16=16 bit multiply`
	`input  signed  [7:0] a, b;`
	`output signed [15:0] z;`
	
	`assign z = a * b;`
	`// -> signed 8x8=16 bit multiply`

`input  [7:0] a, b;`
`output [15:0] z;`
 	
`// emulate signed a, b`
`assign z = (a[6:0] - (a[7]<<7)) *`
`(b[6:0] - (b[7]<<7));`
`// -> two subtract + unsigned 16x16=16 bit multiply`
	`input          [7:0] a, b;`
	`output        [15:0] z;`
	`wire   signed [15:0] z_sgn;`
	
	`assign z_sgn = $signed(a) * $signed(b);`
	`assign z     = $unsigned(z_sgn);`
	`// -> signed 8x8=16 bit multiply`

- **Checks:** Check resources report for type and size of datapath operands (see resources report).

2. Sign-/zero-extension

- **Rule**: Do not manually sign-/zero-extend operands if possible. By using the appropriate unsigned/signed types correct extension is done in the following way:
  - **VHDL**: Use standard functions ('resize' in 'ieee.numeric_std', 'conv_*' in 'ieee.std_logic_arith').
  - **Verilog**: Extension is automatically done.
- **Rationale**: Better QoR because synthesis can more easily/reliably detect extended operands for optimal implementation.

- **Example**:

VHDL	Verilog

`port (a, b : in  signed(7 downto 0);`
`      z    : out signed(8 downto 0));`
 	
`-- a, b explicitly (sign-)extended`
`z <= resize (a, 9) + resize (b, 9);`
	`input  signed [7:0] a, b;`
	`output signed [8:0] z;`
	
	`// a, b implicitly sign-extended`
	`assign z = a + b;`

Verilog Guidelines

3. Mixed unsigned/signed expression (Verilog)

- **Rule**: Do not mix unsigned and signed types in one expression.
- **Rationale**: Unexpected behavior / functional incorrectness because Verilog interprets the entire expression as unsigned if one operand is unsigned.
- **Example**: Multiplication of unsigned operand with signed operand (Verilog).

Functionally incorrect	Functionally correct

`input          [7:0] a;`
`input  signed  [7:0] b;`
`output signed [15:0] z;`
 	
`// expression becomes unsigned`
`assign z = a * b;`
`// -> unsigned multiply`
	`input          [7:0] a;`
	`input  signed  [7:0] b;`
	`output signed [15:0] z;`
	
	`// zero-extended, cast to signed (add '0' as sign bit)`
	`assign z = $signed({1'b0, a}) * b;`
	`// -> signed multiply`

`input  signed  [7:0] a;`
`output signed [11:0] z;`
 	
`// constant is unsigned`
`assign z = a * 4'b1011;`
`// -> unsigned multiply`
	`input  signed  [7:0] a;`
	`output signed [15:0] z1, z2;`
	
	`// cast constant into signed`
	`assign z1 = a * $signed(4'b1011);`
	`// mark constant as signed`
	`assign z2 = a * 4'sb1011;`
	`// -> signed multiply`

- **Checks**: Check for warnings about implicit unsigned-to-signed/signed-to-unsigned conversions/assignments (see warning).

4. Signed part-select / concatenation (Verilog)

- **Note**: *Part-select* results are unsigned, regardless of the operands. Therefore, part-selects of signed vectors (for example, "a[6:0]" of "input signed [7:0] a") become unsigned, even if part-select specifies the entire vector (for example, "a[7:0]" of "input signed [7:0] a").
- **Rule**: Do not use part-selects that specify the entire vector.
- **Note**: Concatenation results are unsigned, regardless of the operands.

- **Example**:

Functionally incorrect	Functionally correct

`input  signed  [7:0] a, b;`
`output signed [15:0] z1, z2;`
 	
`// a[7:0] is unsigned -> zero-extended`
`assign z1 = a[7:0];`
`// a[6:0] is unsigned -> unsigned multiply`
`assign z2 = a[6:0] * b;`
	`input  signed  [7:0] a, b;`
	`output signed [15:0] z1, z2;`
	
	`// a is signed -> sign-extended`
	`assign z1 = a;`
	`// cast a[6:0] to signed -> signed multiply`
	`assign z2 = $signed(a[6:0]) * b;`

- **Checks**: Check for warnings about implicit unsigned-to-signed/signed-to-unsigned conversions/assignments (see warning).

5. Expression widths (Verilog)

- **Note**: The width of an expression in Verilog is determined as followed:
  - *Context-determined expression:* In an assignment, the left-hand side provides the context that determines the width of the right-hand side expression (that is, the expression has the width of the vector it is assigned to).
    Example:

```
input  [7:0] a, b;
output [8:0] z;

assign z = a + b;   // expression width is 9 bits
```

```
input  [3:0] a;
input  [7:0] b;
output [9:0] z;

assign z = a * b;   // expression width is 10 bits
```

  - *Self-determined expression:* Expressions without context (for example, expressions in parenthesis) determine their width from the operand widths. For arithmetic operations, the width of a self-determined expression is the width of the widest operand.
    Example:

Unintended behavior	Intended behavior

`input  signed  [3:0] a;`
`input  signed  [7:0] b;`
`output        [11:0] z;`
 	
`// product width is 8 bits (not 12!)`
`assign z = $unsigned(a * b);`
`// -> 4x8=8 bit multiply`
	`input  signed  [3:0] a;`
	`input  signed  [7:0] b;`
	`output        [11:0] z;`
	`wire   signed [11:0] z_sgn;`
	
	`// product width is 12 bits`
	`assign z_sgn = a * b;`
	`assign z     = $unsigned(z_sgn);`
	`// -> 4x8=12 bit multiply`

`input   [7:0] a, b, c, d;`
`output        z;`
 	
`assign z = (a + b) > (c * d);`
`// -> 8+8=8 bit add + 8x8=8 bit multiply +`
`// 8>8=1 bit compare`
	`input   [7:0] a, b, c, d;`
	`output        z;`
	`wire    [8:0] s;`
	`wire   [15:0] p;`
	
	`assign s = a + b;  // -> 8+8=9 bit add`
	`assign p = c * d;  // -> 8x8=16 bit multiply`
	`assign z = s > p;  // -> 9>16=1 bit compare`

```
input  [15:0] a, b;
output [31:0] z;



assign z = {a[15:8] * b[15:8],
            a[ 7:0] * b[ 7:0]};
// -> two 8x8=8 bit multiplies, bits z[31:16] are 0
```

```
input  [15:0] a, b;
output [31:0] z;
wire   [15:0] zh, zl;

assign zh = a[15:8] * b[15:8];
assign zl = a[ 7:0] * b[ 7:0];
assign z  = {zh, zl};
// -> two 8x8=16 bit multiplies
```

- *Special cases:* Some expressions are not self-determined even though they appear to be. The expression then takes the width of the higher-level context (for example, left-hand side of an assignment).
  Example: Concatenation expression (Verilog).

Bad QoR

```
input          [7:0] a, b;
input                tc;
output signed [15:0] z;



// concatenation expression (9 bits) expanded to 16 bits
assign z = $signed({tc & a[7], a}) *
           $signed({tc & b[7], b});
// -> 16x16=16 bit multiply
```

Good QoR

```
input          [7:0] a, b;
input                tc;
output signed [15:0] z;
wire   signed  [8:0] a_sgn, b_sgn;

assign a_sgn = $signed({tc & a[7], a});
assign b_sgn = $signed({tc & b[7], b});
assign z     = a_sgn * b_sgn;
// -> 9x9=16 bit multiply
```

- **Rule**: Avoid using self-determined expressions. Use intermediate signals and additional assignments to make widths of arithmetic expressions unambiguous (context-determined expressions).
- **Rationale**: Better QoR and/or unambiguous behavior.
- **Checks**: Check resources report for implemented datapath blocks and size of input/output operands (see resources report).

VHDL Guidelines

6. Numeric Packages (VHDL)

- **Rule**: Use the official *IEEE numeric* package 'ieee.numeric_std' for numeric types and functions (the Synopsys package 'ieee.std_logic_arith' is an acceptable alternative). Do not use multiple numeric packages at a time.
- **Rule**: Use numeric types 'unsigned'/'signed' in all arithmetic expressions.
- **Rationale**: Unambiguously specify whether arithmetic operations are unsigned or signed (2's complement).

- **Example**:

Alternative 1	Alternative 2

`library ieee;`
`use ieee.std_logic_1164.all;`
`use ieee.numeric_std.all;`
 	
`entity dp1 is`
`  port (a, b : in  signed(7 downto 0);`
`  z  : out signed(15 downto 0));`
`end dp1;`
 	
`architecture str of dp1 is`
`begin`
`  -- consistent use of numeric types in datapath blocks`
`  z <= a * b;`
`end str;`
	`library ieee;`
	`use ieee.std_logic_1164.all;`
	`use ieee.numeric_std.all;`
	
	`entity dp2 is`
	`  port (a, b : in  std_logic_vector(7 downto 0);`
	`  z  : out std_logic_vector(15 downto 0));`
	`end dp2;`
	
	`architecture str of dp2 is`
	`begin`
	`  -- on-the-fly casts to/from numeric types`
	`  z <= std_logic_vector(signed(a) * signed(b));`
	`end str;`

- **Checks**: Check resources report for implemented datapath blocks and type of operands (see resources report).

Guidelines for QoR

7. Cluster datapath portions

- **Rule**: Cluster related datapath portions in the RTL code together into a single combinational block. Do not separate them into different blocks. In particular:
  - Keep related datapath portions within one single hierarchical component. Do not distribute them into different levels or subcomponents of your design hierarchy.
  - Do not place registers between related datapath portions. If registers are required inside a datapath block to meet QoR requirements, use retiming to move the registers to the optimal location after the entire datapath block has been implemented (see guideline on Pipelining).
- Note: Related datapath portions are portions of RTL code that describe datapath functionality and that allow for certain optimizations/sharings if implemented together. This includes datapath portions that share common inputs or that feed each other (the output of one datapath is used as input to another datapath), as well as datapath portions that have mutually exclusive operations that can possibly be shared.
- **Rationale**: Better QoR because bigger datapath blocks can be extracted and synthesized.
- **Checks**: Check resources report for number and functionality of implemented datapath blocks (see resources report).

8. Mixed unsigned/signed datapath

- **Rule**: Do not mix unsigned and signed types in a datapath cluster (several expressions that form one single complex datapath). Use `signed` ports/signals or cast unsigned ports/signals to signed (using '$signed') to make sure that all operands are signed in a signed datapath.
- **Rationale**: Worse QoR because unsigned and signed operations are not merged together to form one single datapath block (synthesis restriction).

- **Example**: Signed multiply and unsigned add (Verilog)

Bad QoR	Good QoR

`input  signed  [7:0] a, b;`
`input         [15:0] c;`
`output        [15:0] z;`
`wire   signed [15:0] p;`
 	
`// signed multiply`
`assign p = a * b;`
`// unsigned add -> not merged`
`assign z = $unsigned(p) + c;`
`// -> 2 carry-propagations`
	`input  signed  [7:0] a, b;`
	`input         [15:0] c;`
	`output        [15:0] z;`
	`wire   signed [15:0] p;`
	
	`// signed multiply`
	`assign p = a * b;`
	`// signed add -> merged into SOP`
	`assign z = $unsigned(p + $signed(c));`
	`// -> 1 carry-propagation`

- **Checks**: Check resources report for implemented datapath blocks and what operations they implement (see resources report).

9. Switchable unsigned/signed datapath

- **Rule**: Use selective zero-/sign-extension for implementing switchable unsigned/signed datapath (a datapath that can operate on unsigned or signed operands alternatively, controlled by a switch).
- **Rationale**: Better QoR as compared to having two datapaths (unsigned and signed) followed by a selector.
- **Example**: Switchable unsigned/signed multiply-add (Verilog)

Bad QoR	Good QoR

`input          [7:0] a, b, c;`
`input                tc;  // two's compl. switch`
`output        [15:0] z;`
`wire          [15:0] z_uns;`
`wire   signed [15:0] z_sgn;`
 	
`// unsigned datapath`
`assign z_uns = a * b + c;`
`// signed datapath`
`assign z_sgn = $signed(a) * $signed(b) +`
`                $signed(c);`
`// selector`
`assign z     = tc ? $unsigned(z_sgn) : z_uns;`
`// -> two 8x8+8=16 bit datapaths`
	`input          [7:0] a, b, c;`
	`input                tc;  // two's compl. switch`
	`output        [15:0] z;`
	`wire   signed  [8:0] a_sgn, b_sgn, c_sgn;`
	`wire   signed [15:0] z_sgn;`
	
	`// selectively zero-/sign-extend operands`
	`assign a_sgn = $signed({tc & a[7], a});`
	`assign b_sgn = $signed({tc & b[7], b});`
	`assign c_sgn = $signed({tc & c[7], c});`
	`// signed datapath`
	`assign z_sgn = a_sgn * b_sgn + c_sgn;`
	`assign z     = $unsigned(z_sgn);`
	`// -> one 9x9+9=16 bit datapath`

- **Checks**: Check resources report for implemented datapath blocks and size/type of operands (see resources report).

10. POS (Product-Of-Sum) expressions

- **Rule**: Make use of POS (Product-Of-Sum) expressions (add-multiply structures such as "(a + b) * c").
- **Rationale**: Synthesis can implement POS efficiently using carry-save multipliers (for example, a multiplier that allows one input to be in carry-save format) without performing a carry-propagation before the multiply. Better QoR is often achieved when compared to alternative expressions that try to avoid the POS structure.

- **Note**: Synthesis of increment-multiply structures (for example, "`(a + ci) * c`" with '`ci`' being a single bit) is especially efficient (same QoR as a regular multiply "`a * c`" when using Booth-recoding).
- **Example**: Increment-multiply unit (Verilog)

Bad QoR	Good QoR

`input  [7:0] a, b;`
`input  ci;`
`output [15:0] z;`
 	
`// trick for handling increment with regular multiplier`
`assign z = a * b + (ci ? b : 0);`
	`input  [7:0] a, b;`
	`input  ci;`
	`output [15:0] z;`
	
	`// POS expression uses carry-save multiplier`
	`assign z = (a + ci) * b;`

11. Component instantiation

- **Rule**: Do not instantiate arithmetic DesignWare components if possible (for example, for explicitly forcing carry-save format on intermediate results). Write arithmetic expressions in RTL instead.
- **Rationale**: Better QoR can be obtained from RTL expressions by exploiting the full potential of datapath extraction and synthesis (e.g., by using implicit carry-save formats internally).
- **Example**: Multiply-accumulate unit (Verilog)

Bad QoR	Good QoR

`input  [7:0] a, b;`
`input  [15:0] c0, c1;`
`output [15:0] z0, z1;`
`wire  [17:0] p0, p1;`
`wire  [15:0] s00, s01, s10, s11;`
 	
`// shared multiply with explicit carry-save output`
`DW02_multp #(8, 8, 18) mult (`
`  .a(a), .b(b), .tc(1'b0),`
`  .out0(p0), .out1(p1));`
 	
`// add with explicit carry-save output`
`DW01_csa #(16) csa0 (`
`  .a(p0[15:0]), .b(p1[15:0]), .c(c0),`
`  .ci(1'b0), .sum(s00), .carry(s01));`
`DW01_csa #(16) csa1 (`
`  .a(p0[15:0]), .b(p1[15:0]), .c(c1),`
`  .ci(1'b0), .sum(s10), .carry(s11));`
 	
`// carry-save to binary conversion (final adder)`
`DW01_add #(16) add0 (`
`  .A(s00), .B(s01), .CI(1'b0), .SUM(z0));`
`DW01_add #(16) add1 (`
`  .A(s10), .B(s11), .CI(1'b0), .SUM(z1));`
	`input  [7:0] a, b;`
	`input  [15:0] c0, c1;`
	`output [15:0] z0, z1;`
	
	`// single datapath with:`
	`// - automatic sharing of multiplier`
	`// - implicit usage of carry-save internally`
	`assign z0 = a * b + c0;`
	`assign z1 = a * b + c1;`

- **Checks**: Check resources report for implemented datapath blocks (see resources report).

12. Pipelining

- **Rule:** For pipelining of datapaths, place the pipeline registers at the inputs or outputs of the RTL datapath code and use retiming ('optimize_registers' in DC) to move them to the optimal locations. (For more information, refer to “Register Retiming” in the *Design Compiler Reference Manual*.) Do not use DesignWare component instantiations and place the registers manually.

- **Rationale:** Better QoR can be obtained if datapath synthesis can first implement the entire datapath blocks (without interfering registers) and later move the registers to the optimal locations.
- **Note:** Place the pipeline registers at the inputs of the datapath if the registers have reset/preset and the reset/preset state needs to be preserved during retiming (which vastly improves CPU time). Otherwise, pipeline registers can be placed on either inputs or outputs for same retiming functionality and QoR.
- **Example:** Multiply-accumulate unit with 3 pipeline stages (Verilog)

Verilog code

```
module mac_pipe (clk, a, b, c, z);

  input         clk;
  input   [7:0] a, b;
  input  [15:0] c;
  output [15:0] z;
  reg     [7:0] a_reg, a_pipe, a_int;
  reg     [7:0] b_reg, b_pipe, b_int;
  reg    [15:0] c_reg, c_pipe, c_int;
  wire   [15:0] z_int;
  reg    [15:0] z_reg;

  // datapath
  assign z_int = a_int * b_int + c_int;
  assign z     = z_reg;

  always @(posedge clk) begin
    a_reg <= a;   // input register
    b_reg <= b;
    c_reg <= c;
    a_pipe <= a_reg;   // pipeline register 1
    b_pipe <= b_reg;
    c_pipe <= c_reg;
    a_int <= a_pipe;   // pipeline register 2
    b_int <= b_pipe;
    c_int <= c_pipe;
    z_reg <= z_int;   // output register
  end

endmodule
```

Sample script

```
set period 1.0
set num_stages 3

analyze -f verilog mac_pipe.v
elaborate mac_pipe

# adjust clock period for pipelined parts
# (multiply target period by number of stages
# before retiming)
create_clock clk -period \
  [expr $period * $num_stages]
set_max_area 0
compile

# exclude input/output registers from retiming
set_dont_touch *_reg_reg* true

# retime
create_clock clk -period $period
optimize_registers -period $period

# find more information for retiming in the
# "Design Compiler Reference Manual: Register Retiming"
```

13. Complementing an operand

- **Rule:** Do not complement (negate) operands manually by inverting all bits and add a '1' (for example, "a_neg = ~a + 1"). Instead, arithmetically complement operands by using the '-' operator (for example, "a_neg = -a").
- **Rationale:** Manual complementing is not always recognized as an arithmetic operation and therefore can limit datapath extraction and result in worse QoR. Arithmetically complemented operands can easily be extracted as part of a bigger datapath.
- **Example:** see first example in the following item.

14. Special arithmetic optimizations

- **Note:** There are special arithmetic optimizations that are currently not automatically carried out by datapath synthesis but that can potentially improve QoR. With some understanding of the datapath

synthesis capabilities and some experience in arithmetics, different solutions can be found that can give better results.

- **Example:** Conditionally add/subtract a product -> conditionally complement one multiplier input (Verilog)

Bad QoR	Good QoR

`input  signed  [7:0] a, b;`
`input  signed [15:0] c;`
`input                sign;`
`output signed [15:0] z;`
`wire   signed [15:0] p;`
 	
`// manual complement prevents SOP extraction`
`assign p = a * b;`
`assign z = (sign ? ~p : p) +`
`           $signed({1'b0, sign}) + c;`
`// -> multiply + select + 3-operand add`
	`input  signed  [7:0] a, b;`
	`input  signed [15:0] c;`
	`input                sign;`
	`output signed [15:0] z;`
	`wire   signed  [8:0] a_int;`
	
	`// complement multiplier instead of product (cheaper)`
	`assign a_int = sign ? -a : a;`
	`assign z     = a_int * b + c;`
	`// -> complement + SOP (multiply + add)`

`input  signed  [7:0] a, b;`
`input  signed [15:0] c;`
`input                sign;`
`output signed [15:0] z;`
`wire   signed [15:0] p;`
 	
`// arithmetic complement allows SOP extraction`
`// (includes selector)`
`assign p = a * b;`
`assign z = (sign ? -p : p) + c;`
`// -> SOP (multiply + carry-save select + add)`
	`input  signed  [7:0] a, b;`
	`input  signed [15:0] c;`
	`input                sign;`
	`output signed [15:0] z;`
	
	`// complement multiplier using (+1/-1) multiply`
	`assign z = ($signed({sign, 1'b1}) * a) * b + c;`
	`// -> SOP (complement + carry-save multiply + add)`

Synthesis Tool Interaction

Warning Messages

- Warnings about implicit unsigned-to-signed/signed-to-unsigned conversions/assignments:

```
Warning:  ./test.v:31: unsigned to signed assignment occurs. (VER-318)
Warning:  ./test.v:32: signed to unsigned conversion occurs. (VER-318)
```

Resources Report

Use the command 'report_resources' to get a detailed report about the datapath components that were synthesized, including function and operands width/types.

Example: Reports for complex datapath and singleton component.

1. Report of arithmetic operations that were found in the RTL code, what resources are allocated for them, and which modules implement them:

```
Resource Sharing Report for design bad13a in file ./test.v

===============================================================================
|          |                          |            | Contained |                          |
| Resource | Module                   | Parameters | Resources | Contained Operations |
===============================================================================
| r251     | mult_343_DP_OP_243_8993 |            |           | mult_343             |
| r253     | mult_343_DP_OP_243_8993 |            |           | sub_344              |
| r255     | mult_343_DP_OP_243_8993 |            |           | add_344              |
| r262     | DW_mult_tc               | a_width=8  |           | mult_345             |
|          |                          | b_width=8  |           |                          |
===============================================================================
```

2. Report of the interface of modules that implement complex datapath blocks.
   Includes wire and port names, port direction and port widths.

```
Datapath Report for design bad13a in file ./test.v

RTL-datapath Connections for mult_343_DP_OP_243_8993-str
=========================================================================
|                      |                        |           | Bus   |
| RTL Wire             | Datapath Port          | Direction | Width |
=========================================================================
| a                    | I1                     | input     | 8     |
| b                    | I2                     | input     | 8     |
| c                    | I3                     | input     | 16    |
| N0                   | C0                     | control   | 1     |
| N1                   | C1                     | control   | 1     |
| z                    | O1                     | output    | 16    |
=========================================================================
```

3. Report of the functionality of complex datapath blocks.
   Reports function, type (unsigned/signed/mux_op) and reference to RTL operation for output and internal ports. Internal ports are in carry-save format whenever possible/beneficial.

```
Datapath Blocks in mult_343_DP_OP_243_8993-str
=============================================================================
|             | Out   |                          | Contained      | Operation |
| Port        | Width | Datapath Block           | Operation_Line | Type      |
=============================================================================
| Fanout_3    | 16    | I1 * I2                  | mult_343       | SIGNED    |
| Fanout_2    | 16    | 0 - Fanout_3             | sub_344        | SIGNED    |
| Fanout_4    | 16    | { C0 , C1 } ? Fanout_2 : Fanout_3       |                |
|             |       |                          | op8            | MUX_OP    |
|             |       |                          | op9            | MUX_OP    |
| O1          | 16    | Fanout_4 + I3            | add_344        | SIGNED    |
=============================================================================
```

4. Report of the implementation for each module.
   'str' is the generic name for the flexible SOP/POS implementation that is used for all complex datapath blocks.
   For singletons (individual operations implemented by a discrete DesignWare component), the according implementation name is reported (see datasheets).

```
Implementation Report

=============================================================================================
|                           |                          | Current        | Set            |
| Cell                      | Module                   | Implementation | Implementation |
=============================================================================================
| mult_343_DP_OP_243_8993_2 | mult_343_DP_OP_243_8993 | str            |                |
| mult_345                  | DW_mult_tc               | pparch         |                |
=============================================================================================
```