
# RTL Coding Guidelines for Datapath Synthesis

Reto Zimmermann (DesignWare Staff R&D)
Arshid Syed (DesignWare Sr. CAE)
Synopsys, Inc.


# Outline

1. Introduction
2. Datapath Synthesis
3. RTL Coding Guidelines for Datapath
4. Summary
5. Q & A


# 1. Introduction

- Arithmetic Content in Designs Increasing
  - Graphics, Multimedia, DSP and Communications
  - Unsigned and Signed (2's complement) Arithmetic
  - Influence on QoR (Quality-of-Results) and Functional Correctness
- Goals of the Tutorial
  - Understanding Datapath Synthesis Solution
  - RTL Coding for Functional Correctness
  - RTL Coding for Best Possible QoR
  - Understanding Datapath Synthesis Reports


# 2. Datapath Synthesis

- Datapath Synthesis in DC
- Understanding Functionality Supported by Datapath Synthesis
- Understanding DC Datapath Synthesis Flow
- Example with
  - DC
  - DC + DesignWare Library
  - DC-Ultra + DesignWare Library

# Datapath Synthesis – How to Use

- DC-Ultra and DesignWare Library licenses are required for the datapath synthesis solution discussed in this tutorial
- The following are also included in DesignWare Library
  - DesignWare Building Block IP
  - SoC infrastructure IP (on-chip bus, peripherals, microcontrollers, memory controller, BIST, etc.)
  - Verification IP (memory, PCI, USB, ethernet,…)
  - For more information, please go to:

**http://www.synopsys.com/designware**

# Datapath Synthesis – How to Use

- Access DesignWare Library
  - set synthetic_library “dw_foundation.sldb”
  - set link_library [concat  $target_library $synthetic_library]
- Access DC-Ultra
  - set_ultra_optimzation true

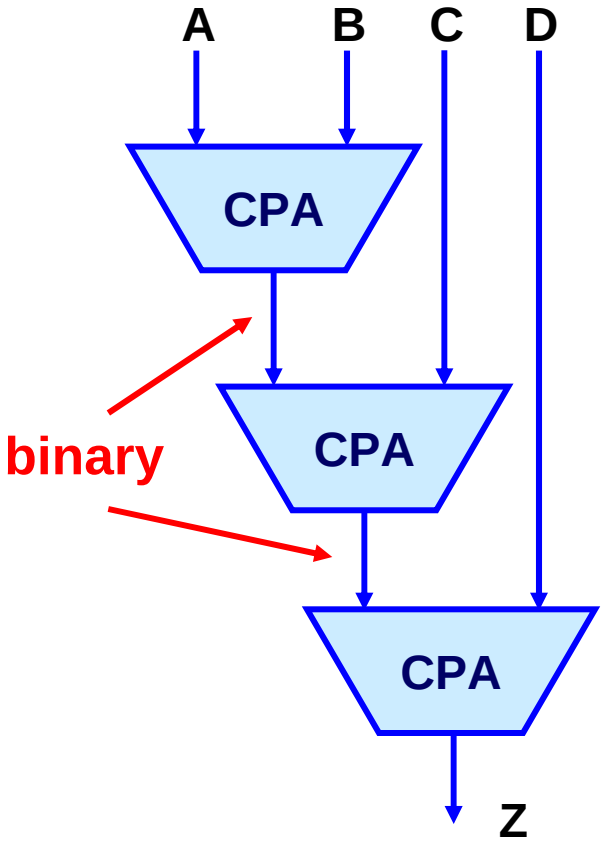
**Note:** From 2004.12 onwards, dw_foundation.sldb is added to the synthetic library list automatically in ultra mode.

# Datapath Optimization Techniques

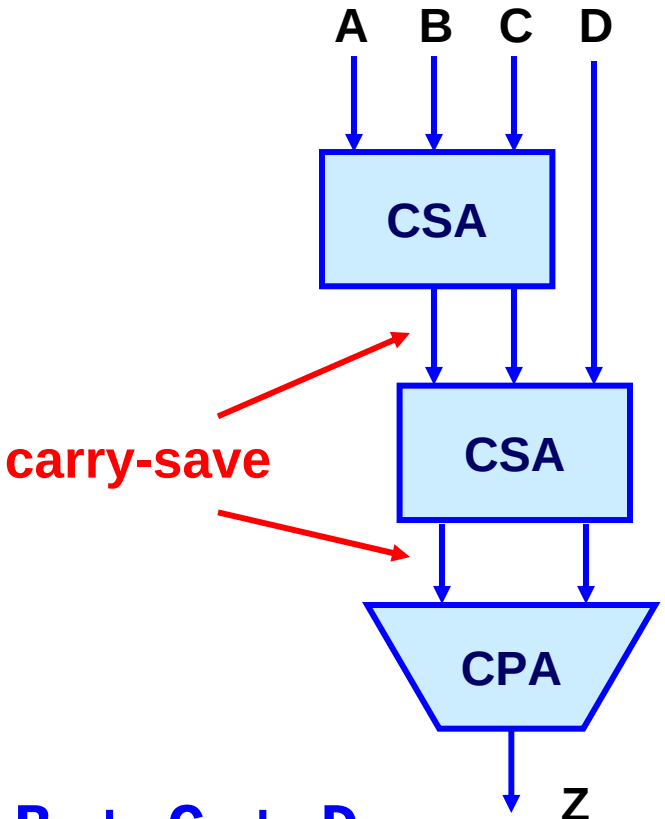
- Datapath optimization techniques:
  - Avoid expensive carry-propagations (conversion to binary representation) (large, slow)
  - Use redundant representations instead
    - Carry-save
    - Partial-product
  - High-level arithmetic optimizations
    - Common sub-expression sharing, constant folding, etc.
  - Apply above techniques on the biggest possible datapath partitions

# Carry-Save Representation

- Carry-propagate adder (CPA)
- Binary results




- Carry-save adder (CSA)
- Carry-save results




$$Z = A + B + C + D$$

SYNOPSYS®

# Carry-Save Representation

- Carry-save results save CPAs
- Carry-save results only possible if supported by following operation
- Operations that support carry-save inputs:
  - Addition / subtraction
  - Multiplication (only one input in carry-save)
  - Select-operation
  - Comparison

# Supported Datapath Functionality

- Biggest possible datapath blocks can be extracted by supporting the following functionality:
  - **Sum-of-Product** (SoP): Arbitrary sum-of-products can be implemented in one datapath block with one single carry-propagate final adder (CPA)

    Ex: `z = a*b + c*d – 456*e + f - g + 238`
  - **Product-of-Sum** (PoS): Limited product-of sums can be implemented in one datapath block with one single carry-propagate final adder (CPA)

    Ex: `z1 = (a+b) * c; z2 = a * b * c`

# Supported Datapath Functionality

- Datapath functionality (cont.)
  - **Select-op:** Select-operations can be implemented as part of a datapath block on redundant internal results (i.e., without carry-propagation before the select-op).

    **Ex:** `z = (sign ? (a * b) : (c * d)) + c`
  - **Comparison:** Comparisons can be implemented as part of a datapath block on redundant internal results (i.e., without carry-propagation before the comparison).

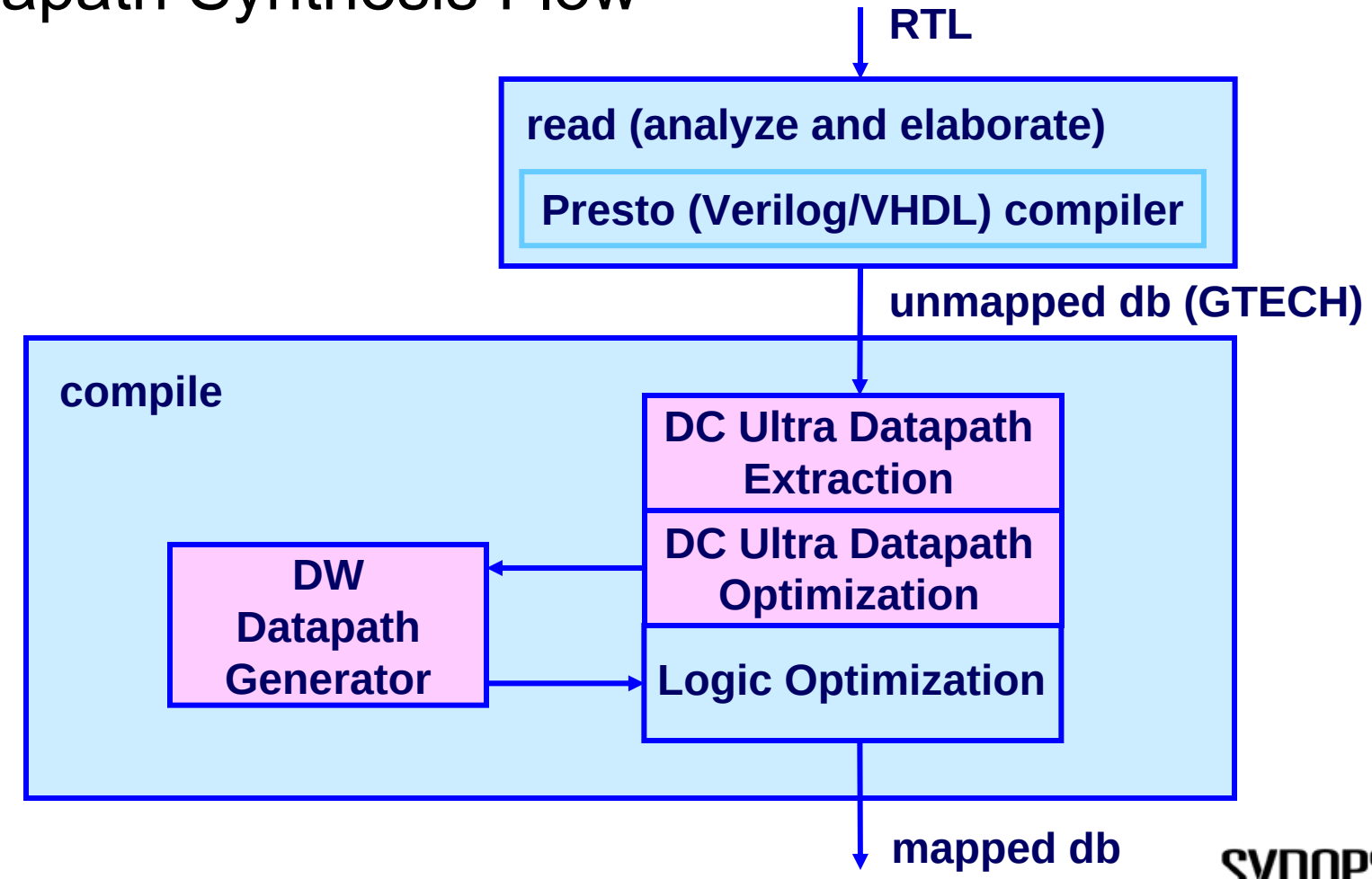
    **Ex:** `t1 = a + b; t2 = c * d; z = t1 > t2`

# Datapath Synthesis Flow

- DC-Ultra Datapath Synthesis Flow
  1. Datapath Extraction
  2. Datapath Optimization
  3. Datapath Generation

# Datapath Synthesis Flow

- Datapath Synthesis Flow



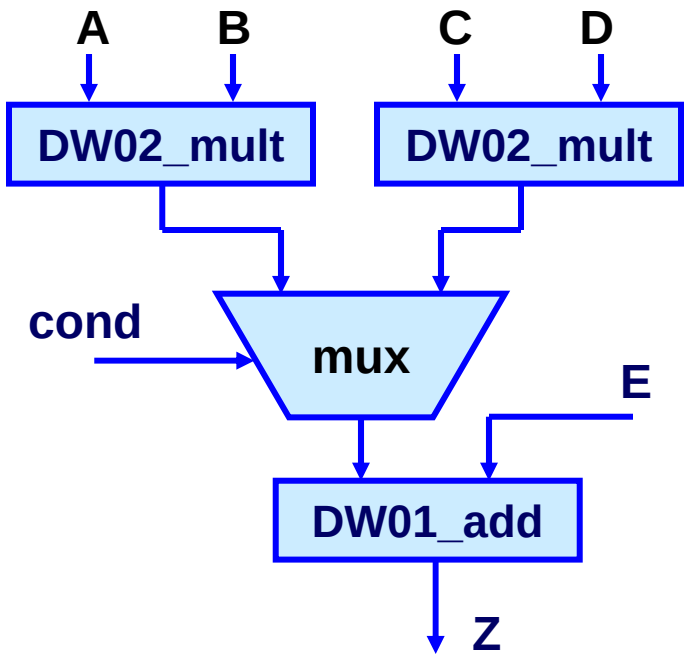
# Datapath Extraction

- DC-Ultra datapath extraction
  - Extract biggest possible datapath blocks from the RTL code.
  - Merge arithmetic operators (ex: **+**, **-**, *****, **?**, **>**) into single datapath blocks.

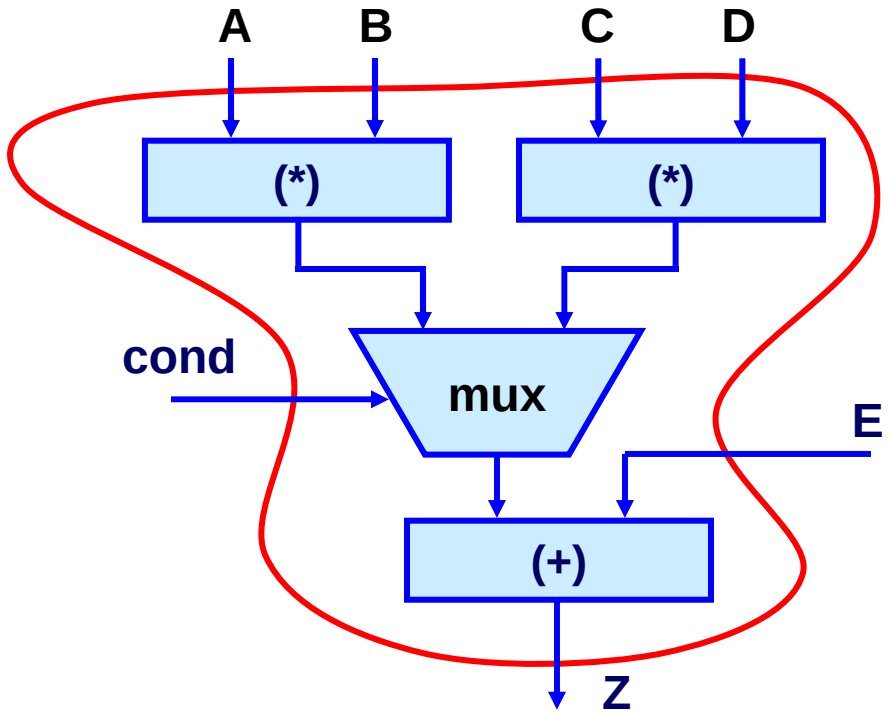
    Ex: `z = (cond ? (a * b) : (c * d)) + e`

# Datapath Extraction

- Example: `Z = (cond ? (A * B) : (C * D)) + E`




**No extraction with DC-Expert + DWL**




**Extraction with DC-Ultra + DWL**

# Datapath Optimization

- DC Ultra datapath optimization
  - Resource / common subexpression sharing
  - Carry-save arithmetic technique
  - High-level arithmetic optimizations
  - Explores different solutions (e.g. sharing configurations)

# Datapath Generation

- DC-Ultra datapath generation:
  - Flexible, context-driven datapath generators implement optimized netlist for the datapath blocks under specified constraints, conditions and libraries.
- Reporting:
  - Use commands report_timing, report_area, and report_qor to check QoR.
  - Use report_resources command to determine which operators were implemented in a datapath block.
  - Checking warning messages always recommended.

# Datapath Synthesis Example

```
// Example to show Datapath Extraction and Optimization
//  and comparing the DC, DC+DWL and DC-Ultra+DWL QoR
input  [  width-1 : 0] a, b, c, d, f;
input  [2*width-1 : 0] e, g;
output [2*width-1 : 0] z1, z2;

wire   [2*width-1 : 0] i1, i2, i3, i4, i5;

assign i1 = (a * b);
assign i2 = (a * c);
assign i3 = (d * f);
assign i4 = e - g;
assign i5 = i1 + i2;

assign z1 = i5 + i3 + i4;
assign z2 = i5 + e;
```

# Datapath Synthesis Example

- Resource Sharing Report **(DC+DWL)**
  - 8 DW modules

| Resource | Module | Parameters | Contained Resources | Contained Operations |
|---|---|---|---|---|
| r299 | DW_mult_uns | a_width=32<br>b_width=32 | | mult_13 |
| r301 | DW_mult_uns | a_width=32<br>b_width=32 | | mult_15 |
| r303 | DW_mult_uns | a_width=32<br>b_width=32 | | mult_17 |
| r305 | DW01_add | width=64 | add_2_root_add_23_2 | |
| r307 | DW01_add | width=64 | | add_21 |
| r309 | DW01_sub | width=64 | sub_1_root_add_23_2 | |
| r311 | DW01_add | width=64 | add_0_root_add_23_2 | |
| r313 | DW01_add | width=64 | | add_24 |

# Datapath Synthesis Example

- Implementation report: **(DC+DWL)**

| Cell | Module | Current Implementation | Set Implementation |
|---|---|---|---|
| add_0_root_add_23_2 | DW01_add | pparch | |
| add_2_root_add_23_2 | DW01_add | pparch | |
| add_21 | DW01_add | pparch | |
| add_24 | DW01_add | pparch | |
| mult_13 | DW_mult_uns | pparch | |
| mult_15 | DW_mult_uns | pparch | |
| mult_17 | DW_mult_uns | pparch | |
| sub_1_root_add_23_2 | DW01_sub | pparch | |

# Datapath Synthesis Example

- Resource Sharing Report **(DC-Ultra+DWL)**
  - 1 single DP module

| Resource | Module | Parameters | Contained Resources | Contained Operation |
|---|---|---|---|---|
| r262 | add_21_DP_OP_245_8068 | | | mult_13 |
| r264 | add_21_DP_OP_245_8068 | | | mult_15 |
| r266 | add_21_DP_OP_245_8068 | | | mult_17 |
| r268 | add_21_DP_OP_245_8068 | | | sub_19 |
| r270 | add_21_DP_OP_245_8068 | | | add_21 |
| r272 | add_21_DP_OP_245_8068 | | | add_23 |
| r274 | add_21_DP_OP_245_8068 | | | add_23_2 |
| r276 | add_21_DP_OP_245_8068 | | | add_24 |

# Datapath Synthesis Example

- Datapath Report: **(DC-Ultra+DWL)**

RTL-Datapath Connections for add_21_DP_OP_245_8068-str

| RTL Wire | Datapath Port | Direction | Bus Width |
|---|---|---|---|
| a | I1 | input | 32 |
| b | I2 | input | 32 |
| c | I3 | input | 32 |
| d | I4 | input | 32 |
| f | I5 | input | 32 |
| e | I6 | input | 64 |
| g | I7 | input | 64 |
| z1 | O1 | output | 64 |
| z2 | O2 | output | 64 |

# Datapath Synthesis Example

**(DC-Ultra+DWL)**

Datapath Blocks in add_21_DP_OP_245_8068-str

| Port | Out Width | Datapath Block | Contained Operation_Line | Operation Type |
|---|---|---|---|---|
| Fanout_3 | 64 | I1 * I2 + I1 * I3 | add_21 | UNSIGNED |
| | | | mult_13 | UNSIGNED |
| | | | mult_15 | UNSIGNED |
| O1 | 64 | Fanout_3 + I4 * I5 + I6 - I7 | | |
| | | | add_23_2 | UNSIGNED |
| | | | add_23 | UNSIGNED |
| | | | sub_19 | UNSIGNED |
| | | | mult_17 | UNSIGNED |
| O2 | 64 | Fanout_3 + I6 | add_24 | UNSIGNED |

# Datapath Synthesis Example

- Implementation report: **(DC-Ultra+DWL)**

```
============================================================
|                |           |Current        |Set            |
| Cell           | Module    |Implementation |Implementation|
============================================================
|add_21_DP_OP_245_8068_2|add_21_DP_OP_245_8068|str|     |
============================================================
```

# Datapath Synthesis Example

- **Results:**

| Flow | Slack (ns) | Area (lib units) | Run Time (secs) |
|---|---|---|---|
| DC | -5.80 | 124506 | 3837 |
| DC + DW | -1.51 | 76848 | 1400 |
| DC-Ultra + DW | 0.00 | 56316 | 912 |

- Target Library TSMC 90nm
- set_max_delay –from [all_inputs] –to [all_outputs] 3
- set_max_area 0

# Summary: DC-Ultra Datapath Synthesis

- DC-Ultra datapath synthesis flow
  - Datapath extraction
  - Datapath optimization
  - Datapath generation

- All these steps help improve Quality-of-Results (QoR) → better timing & area
- Only with DC-Ultra & DesignWare

# 3. RTL Coding Guidelines for Datapath

- Guidelines serve two purposes:
  - Help datapath synthesis to achieve best possible Quality-of-Results (QoR).
  - Help achieve functional correctness and intended behavior of arithmetic expressions in RTL code.


# RTL Coding Guidelines for Datapath

- RTL code can be optimized 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.


# Guideline 1: Signed Arithmetic Overview

## 1. Signed Arithmetic

- **Rule:** Use type '**`signed`**' (VHDL, Verilog 2001) for signed/2's complement arithmetic (i.e., do not emulate signed arithmetic using unsigned operands/operations). Also, do not use the '**`integer`**' type except for constant values.

# Guideline 1: Signed Arithmetic Example

```
input   [7:0] a, b;                  // Case 1: bad QoR
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;       // Case 2: good QoR
output signed [15:0] z;

assign z = a * b;
 // -> signed 8x8=16 bit multiply
```

# Guideline 1: Signed Arithmetic Example

```
input   [7:0] a, b;              // Case 1: bad QoR
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          [7:0] a, b;     // Case 3: good QoR
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
```

# Guideline 1: Signed Arithmetic Example

- Design Compiler Version W-2004.12
- Target Library TSMC 0.18um
- set_max_delay –from [all_inputs] –to [all_outputs] 0
- set_max_area 0


# Guideline 1: Signed Arithmetic Results

- **Results:**

| | Slack | Area |
|---|---|---|
| Case 1 | -2.42 | 14296 |
| Case 2 | -2.11 | 8598 |

- **Rationale:** Better QoR for a signed datapath (case 2) when compared with an unsigned datapath emulating signed behavior (case 1).

- **Checks:** Check the resources report for type and size of datapath operands ('`report_resources`' command).

# Guideline 1: Signed Arithmetic Reports

- report_resources (Case 1: bad QoR)

Resource Sharing

| Resource | Module | Contained Parameters | Resources | Contained Operations |
|---|---|---|---|---|
| r251 | DW_mult_uns | A_width=16<br>B_width=16 | | mult_7 |

Implementation Report

| Cell | Module | Current Implementation | Set Implementation |
|---|---|---|---|
| mult_7 | DW_mult_uns | pparch | |

# Guideline 1: Signed Arithmetic Reports

- report_resources (Case 2: good QoR)

Resource Sharing

```
=================================================
|          |        | Contained  |            | Contained |
| Resource |Module  | Parameters | Resources  | Operations|
=================================================
| r251     |DW_mult_tc|A_width=8|            | mult_6    |
|          |          |B_width=8|            |           |
=================================================
```

Implementation Report

```
=================================================
|          |              | Current       | Set           |
| Cell     | Module       | Implementation| Implementation|
=================================================
| mult_6   | DW_mult_tc   | pparch        |               |
=================================================
```

# Guideline 2: Sign-/zero-extension Overview

## 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.

# Guideline 2: Sign-/zero-extension Examples

```
input  signed [7:0] a, b;          // Case 1: Verilog
output signed [8:0] z;

// a, b implicitly sign-extended
assign z = a + b;
```

```
port (a, b : in  signed(7 downto 0); -- Case 2: VHDL
      z    : out signed(8 downto 0));

-- a, b explicitly (sign-)extended
z <= resize (a, 9) + resize (b, 9);
```

- **Rationale:** Better QoR because synthesis can more easily/reliably detect extended operands for optimal implementation.


# Guideline 3: Mixed unsigned/signed Overview

## 3. Mixed unsigned/signed expression (Verilog)

- **Rule:** Do not mix unsigned and signed types in one expression.
  - Expressions with mixed types are interpreted as unsigned in Verilog (does not make sense from arithmetic point of view).
  - Can result in unexpected behavior or functional incorrectness.

# Guideline 3: Mixed unsigned/signed Examples

```
input           [7:0] a;        // Case 1: Incorrect
input  signed   [7:0] b;
output signed [15:0] z;

// expression becomes unsigned
assign z = a * b;
// -> unsigned multiply
```

```
input           [7:0] a;        // Case 2: Correct
input  signed   [7:0] b;
output signed [16:0] z;

// zero-extended, cast to signed (add '0' as sign bit)
assign z = $signed({1'b0, a}) * b;
// -> signed multiply
```

# Guideline 3: Mixed unsigned/signed Examples

```
input  signed  [7:0] a;          // Case 3: Incorrect
output signed [11:0] z;

// constant is unsigned
assign z = a * 4'b0011;      // -> unsigned multiply
```

```
input  signed  [7:0] a;          // Case 4: Correct
output signed [15:0] z1, z2;

// cast constant into signed
assign z1 = a * $signed(4'b0011);
// mark constant as signed
assign z2 = a * 4'sb0011;    // -> signed multiply
```

# Guideline 3: Mixed unsigned/signed Results

- **Checks:** Warnings about implicit unsigned-to-signed or 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)
```

  Check the resources report for type of datapath operands.

- **Rationale:** Unexpected behavior / functional incorrectness because Verilog interprets the entire expression as unsigned if one operand is unsigned.


# Guideline 4: Part-select/concatenation Overview

## 4. Signed part-select / concatenation (Verilog)

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

# Guideline 4: Part-select/concatenation Examples

```
input  signed  [7:0] a, b;      // Case 1: Incorrect
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;       // Case 2: Correct
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;
```


# Guideline 4: Part-select/concatenation Results

- **Checks:** 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)
```

  Check the resources report for type of datapath operands.

- **Rationale:** Unexpected behavior/functional incorrectness because the part-select and concatenation results in unsigned value regardless of the sign of the operands.


# Guideline 5: Expression width Overview

## 5. **Expression width** (Verilog)

- **Note:** The width of an expression in Verilog is determined:
  - **Context-determined** expression: left-hand side provides the context, determines the width.
  - **Self-determined** expression: no context available, width is determined by widest operand.
- **Rule:** Avoid self-determined expressions. Use intermediate signals and additional assignments to make widths of arithmetic expressions unambiguous (context-determined expressions). Note: this does not affect datapath extraction.

# Guideline 5: Expression width Examples

```
input  [7:0] a, b;   // Case 1: Context-determined
output [8:0] z;

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

```
input  [3:0] a;      // Case 2: Context-determined
input  [7:0] b;
output [9:0] z;

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

# Guideline 5: Expression width Examples

```
input  signed  [3:0] a;     // Case 1: Self-determined
input  signed  [7:0] b;     // (unintended behavior)
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;    // Case 2: Self-determined
input  signed  [7:0] b;    // (intended behavior)
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
```

# Guideline 5: Expression width Examples

```
input   [7:0] a, b, c, d;  // Case 3: Self-determined
output        z;           // (unintended behavior)

assign z = (a + b) > (c * d);
// -> 8+8=8 bit add + 8x8=8 bit multiply + 8>8=1 bit comp
```

```
input   [7:0] a, b, c, d;  // Case 4: Self-determined
output        z;           // (intended behavior)
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
```


# Guideline 5: Expression width Results

- **Checks:** Check the resources report for implemented datapath blocks and size of input/output operands.
- **Rationale:** Better QoR and/or unambiguous behavior of arithmetic expressions.

# Guideline 6: Numeric Packages (VHDL) Overview

## 6. Numeric Packages in VHDL

- **Rule 1:** 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 several numeric packages at a time.

- **Rule 2:** Use numeric types '**`unsigned`**' and '**`signed`**' in all arithmetic expressions.

# Guideline 6: Numeric Packages (VHDL) Examples

```
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;
```

# Guideline 6: Numeric Packages (VHDL) Examples

```
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;
```

# Guideline 6: Numeric Packages (VHDL) Results

- **Checks:** Check the resources report for implemented datapath blocks and type of operands.
- **Rationale:** Unambiguously specify whether arithmetic operations are unsigned or signed (2's complement).

# Guideline 7: Cluster Datapath Overview

## 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 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.


# Guideline 7: Cluster Datapath Overview

- **Checks:** Check the resources report for the number and functionality of implemented datapath blocks.
- **Rationale:** Better QoR because bigger datapath blocks can be extracted and synthesized.

# Guideline 8: Mixed unsigned/signed Overview

## 8. Mixed unsigned/signed datapath

- **Rule:** Do not mix unsigned and signed types in a datapath cluster. 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.
  - Signed and unsigned operators not merged in a single datapath block and results in more carry-propagations, hence worse QoR.

# Guideline 8: Mixed unsigned/signed Examples

```
input  signed  [7:0] a, b; // Case 1 (signed & unsigned
input         [15:0] c;     // mixed) (bad QoR)
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
```

# Guideline 8: Mixed unsigned/signed Examples

```
input  signed  [7:0] a, b;   // Case 2 (all-signed)
input         [15:0] c;      // (good QoR)
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
```

# Guideline 8: Mixed unsigned/signed Reports

- report_resources (Case 1: bad QoR)

Resource Sharing Report

```
=================================================
|        |           |           |Contained | Contained |
|Resource| Module    |Parameters|Resources | Operations|
=================================================
| r256   |DW_mult_tc|A_width=8 |          | mult_9    |
|        |           |B_width=8 |          |           |
| r258   |DW01_add   |  width=16|          | add_11    |
=================================================
```

Implementation Report

```
=============================================
|       |           | Current        | Set            |
| Cell  | Module    | Implementation | Implementation|
=============================================
| add_11|DW01_add   | pparch         |                |
| mult_9|DW_mult_tc| pparch         |                |
=============================================
```

# Guideline 8: Mixed unsigned/signed Reports

- report_resources (Case 2: good QoR)

Datapath Blocks in add_11_DP_OP_245_1339-str

| Port | Out Width | Datapath Block | Contained Operation_Line | Operation Type |
|---|---|---|---|---|
| O1 | 16 | I1 * I2 + I3 | add_11 | SIGNED |
| | | | mult_9 | SIGNED |

Implementation Report

| Cell | Module | Current Implementation | Set Implementation |
|---|---|---|---|
| add_11_DP_OP_245_1339_2 | add_11_DP_OP_245_1339 | str | |

# Guideline 8: Mixed unsigned/signed Results

- **Results:**

| | Slack | Area |
|---|---|---|
| Case 1 | -3.23 | 12034 |
| Case 2 | -2.71 | 10840 |

- **Checks:** Check the resources report for implemented datapath blocks and the operations they implement.
- **Rationale:** Worse QoR because unsigned and signed operations are not merged together to form one single datapath block.

# Guideline 9: Switchable unsigned/signed Overview

## 9. Switchable unsigned/signed datapath

- **Note:** Datapath that can operate on unsigned or signed operands alternatively, controlled by a switch.
- **Rule:** Use selective zero-/sign-extension for implementing switchable unsigned/signed datapath.
  - Better QoR than two separate datapaths (one unsigned and one signed).

# Guideline 9: Switchable unsigned/signed Examples

```
input   [7:0] a, b, c;        // Case 1 (two datapaths)
input         tc;             // (bad QoR)
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
```

# Guideline 9: Switchable unsigned/signed Examples

```
input   [7:0] a, b, c;      // Case 2 (single datapath)
input         tc;           // (good QoR)
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
```

# Guideline 9: Switchable unsigned/signed Reports

- report_resources (Case 1: bad QoR)

Datapath Blocks reported in add_11_DP_OP_246_2062-str
& add_11_DP_OP_246_2063-str

| Port | Out Width | Datapath Block | Contained Operation_Line | Operation Type |
|---|---|---|---|---|
| O2 | 16 | I1 * I2 + I3 | add_11 | UNSIGNED |
| | | | mult_11 | UNSIGNED |
| O1 | 16 | I1 * I2 + I3 | add_14 | SIGNED |
| | | | mult_14 | SIGNED |

# Guideline 9: Switchable unsigned/signed Reports

- report_resources (Case 2: good QoR)

Datapath Blocks in add_16_DP_OP_245_3385-str

| Port | Out Width | Datapath Block | Contained Operation_Line | Operation Type |
|---|---|---|---|---|
| O1 | 16 | I1 * I2 + I3 | add_16 | SIGNED |
| | | | mult_16 | SIGNED |

# Guideline 9: Switchable unsigned/signed Results

- **Results:**

| | Slack | Area |
|---|---|---|
| Case 1 | -2. 48 | 18092 |
| Case 2 | -2. 42 | 10624 |

- **Checks:** Check the resources report for implemented datapath blocks and size/type of operands.
- **Rationale:** Better QoR as compared to having two datapaths (unsigned and signed) followed by a selector.

# Guideline 10: Product-of-Sum Overview

## 10. Product-of-Sum (PoS) expressions

- **Rule:** Make use of Product-of-Sum expressions (i.e. add-multiply structures, "**`(a + b) * c`**").
  - Special carry-save multiplier accepts one input in carry-save format (without conversion to binary).
  - Allows efficient implementation of Product-of-Sum (PoS) expressions.
  - Particularly efficient if second addend '**b**' is a single bit (increment, carry-in).

# Guideline 10: Product-of-Sum Examples

```
input   [7:0] a, b;          // Case 1 (without PoS)
input         ci;            // (bad QoR)
output [15:0] z;

// trick for handling increment with regular multiplier
assign z = a * b + (ci ? b : 0);
```

```
input   [7:0] a, b;          // Case 2 (with PoS)
input         ci;            // (good QoR)
output [15:0] z;

// PoS expression uses carry-save multiplier
assign z = (a + ci) * b;
```

# Guideline 10: Product-of-Sum Reports

- report_resources

RTL-datapath Connections for add_8_DP_OP_245_8177-str

```
=======================================
|         |                |          |Bus   |
|RTL Wire|Datapath Port|Direction|Width|
=======================================
| a      | I1             | input    |  8   |
| b      | I2             | input    |  8   |
| N10-N3| I3             | input    |  8   |
| z      | O1             | output   | 16   |
=======================================
|ci-ci   | I2             | input    |  1   |
```

(Case 1: bad QoR)

(Case 2: good QoR)

SYNOPSYS®

# Guideline 10: Product-of-Sum Results

- **Results:**

| | Slack | Area |
|---|---|---|
| Case 1 | -2.76 | 10764 |
| Case 2 | -2.57 | 10903 |

- **Rationale:** Synthesis can implement PoS efficiently using carry-save multipliers 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.

# Guideline 11: Instantiation Overview

## 11. Component instantiation

- **Rule:** Do not instantiate arithmetic DesignWare components if possible (e.g., for explicitly forcing carry-save format on intermediate results). Write expressions using arithmetic operators instead.
  - Instantiated components are not extracted but implemented as individual DW components (possibly worse QoR).
  - Use arithmetic operators to enable datapath extraction to achieve best possible QoR.

# Guideline 11: Instantiation Example

```
input   [7:0] a, b;                          // Case 1 (instantiation)
input  [15:0] c0, c1;                        // (bad QoR)
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));
```

# Guideline 11: Instantiation Example

```
input   [7:0] a, b;     // Case 2 (operator inference)
input  [15:0] c0, c1;   // (good QoR)
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;
```

# Guideline 11: Instantiation Reports

- report_resources

  Implementation report

```
============================================
|      |            | Current       | Set           |
| Cell | Module     | Implementation| Implementation|
============================================
| add0 | DW01_add   | pparch        |               |
| add1 | DW01_add   | pparch        |               |
| csa0 | DW01_csa   | str           |               |
| csa1 | DW01_csa   | str           |               |
| mult | DW02_multp| nbw           |               |
============================================

=============================================
|     |               |Current       |Set              |
| Cell| Module        |Implementation|Implementation   |
=============================================
| mult_12_DP_OP_247_3685_1|mult_12_DP_OP_247_3685|str| |
=============================================
```

# Guideline 11: Instantiation Results

- **Results:**

| | Slack | Area |
|---|---|---|
| Case 1 | -3.31 | 17520 |
| Case 2 | -2.93 | 17180 |

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

# Guideline 12: Pipelining Overview

## 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. Do not use DW component instantiations and place the registers manually**.**
  - Preferably place registers at inputs.

# Guideline 12: Pipelining Example

```
input           clk;        // Pipelining example
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;
```

(Cont.)

# Guideline 12: Pipelining Example

(Cont.)

```
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
```

# Guideline 12: Pipelining Example

```
set period 1.0                                  // Sample Script
set num_stages 3
analyze -f verilog mac_pipe.v
elaborate mac_pipe
# adjust clock period for pipelined parts
# (multiply 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 "DC Ref Man"
```


# Guideline 12: Pipelining Results

- **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.


# Guideline 13: Complementing Overview

## 13. Complementing an operand

- **Rule:** Do **not** complement (negate) operands manually by inverting all bits and add a '1' (e.g., "`a_neg = ~a + 1`"). Instead, arithmetically complement operands by using the '`-`' operator (e.g., "`a_neg = -a`").
  - Manual complements prevent datapath extraction.
  - Note: Simple cases can be extracted.

# Guideline 13: Complementing Examples

```
input  signed  [7:0] a, b;          // Case 1 (bad QoR)
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
```

```
// complement multiplier instead of product // Case 2 (good QoR)
assign a_int = sign ? -a : a;
assign z     = a_int * b + c;
// -> complement + SOP (multiply + add)
```

# Guideline 13: Complementing Results

- **Results:**

| | Slack | Area |
|---|---|---|
| Case 1 | -3.39 | 13997 |
| Case 2 | -3.15 | 13528 |

- **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.


# Guidelines Summary

- Signed arithmetic
  - Use '**signed**' type, numeric packages (VHDL)
  - Do not mix unsigned and signed types
  - Careful with part-select / concatenation (Verilog)
- Expression widths
  - Use intermediate signals for better controlling widths
- Cluster datapath portions
- Use arithmetic expressions, not instantiations
- Do not apply "manual" arithmetic tricks
- Make use of special (datapath) functionality
  - Product-of-Sum
  - Pipelining
- Check resources report and warnings

# Other Coding Examples
# Shifter: Shift Operators

- Use shift operators instead of case statements
  - Dedicated DW shifter implementations are inferred
  - Better QoR

```
case (sh)                          // Case 1 (bad QoR)
  2'b00 : z = a[3:0];
  2'b01 : z = a[2:0];
  2'b10 : z = a[1:0];
  2'b11 : z = a[0];
endcase
```

```
assign z = a >> sh;                // Case 2 (good QoR)
```

# Other Coding Examples
# Shifter: Shifting Distance

- Reduce number of bits in shifting distance to minimum required
  - Inferred shifters are smaller, better QoR

```
input  [15:0] a;                    // Case 1 (bad QoR)
input  [15:0] sh;
output [15:0] z;

assign z = a >> sh;
```

```
input  [15:0] a;                    // Case 2 (good QoR)
input   [4:0] sh;
output [15:0] z;

assign z = a >> sh;
```

# Other Coding Examples
# Priority Encoder

- Not use comparators for encoding / decoding
- Use DW parts through function inference / component instantiation if no operator is available

```
input  [3:0] a;                          // Case 1 (bad QoR)
output [2:0] idx;

assign idx = (a >= 4'b1000) ? 4 :
             (a >= 4'b0100) ? 3 :
             (a >= 4'b0010) ? 2 :
             (a >= 4'b0001) ? 1 : 0;
```

```
assign idx = DWF_prienc (a);  // Case 1 (good QoR)
DW01_prienc #(4, 3) U1 (.A(a), .INDEX(idx));
```

# 4. Summary

- DC-Ultra + DWL gives best possible QoR for datapath designs.
- RTL coding style has impact on QoR and functional correctness.
- Coding guidelines also in DW Technical Bulletin
  **http://www.synopsys.com/products/designware/dwtb/**
- Questions on this tutorial, please email our CAE:
  Arshid Syed (**arshid@synopsys.com**)
- For any other DWL and datapath related issues please open a CASE through SolvNet:
  **https://solvnet.synopsys.com/**


# 5. Q & A

**Thank you**

**Q & A**