SolvNet®

Coding Guidelines for Datapath Synthesis

Doc Id: 015771 **Product:** Design Compiler **Last Modified:** 06/01/2012

Coding Guidelines for Datapath Synthesis

Contents

Introduction

This document summarizes coding guidelines addressing the synthesis of datapaths. Two classes of guidelines are distinguished:

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

Datapath Synthesis

To write RTL code that gives the 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 to make **use of redundant representations** instead (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 data 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 (that is, a sum followed by a multiply) can be implemented in one datapath block with one single carry-propagate final adder (that is, no carry-propagation before the multiply). The limitation is that only one multiplication operand can be in redundant (carry-save) format; the other operand has to be binary.

  Example: "`z = (a + b) * c`" , "`z = a * b * c`".

- **Select-op:** Select-operations (that is, selectors, operand-wide multiplexers) can be implemented as part of a datapath block on redundant internal results (that is, 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 (that is, 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).

- **Shift:** Constant and variable shifts can be implemented as part of a datapath block on redundant internal results (that is, without carry-propagation before the comparison).

  Example: "`t1 = a * b; z =` (t1 << c) + d".

Synthesis Flow

The datapath synthesis flow consists of the following stages:

1. **Datapath extraction:** The largest 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.

Datapath Extraction

Largest possible datapath blocks are extracted from the RTL code by merging connected operators into a single partition.

- The larger the partition is the more high-level optimizations can be applied and the fewer carry-propagations are needed (shorter delay).
- All internal operands of a datapath block can be in redundant carry-save representation to improve delay. Therefore operands that need to be in binary representation cannot be internal to a datapath block.
- Extraction of larger datapath blocks can be blocked by
  - datapath leakage (discussed in **guideline 19**).
  - Operators that require binary inputs or outputs → must be at the boundary of a datapath block.
    - Comparator: need a binary conversion for flag calculation → must be at output boundary.
    - Selector / shifter: need binary control input → control input must be at input boundary.
  - Instantiated DesignWare components (**guideline 11**).
  - Registers between extractable operators (**guideline 7**).

Coding Goals

RTL code can be optimized for datapath synthesis by achieving the following goals:

- Enable datapath extraction to **extract the biggest possible datapath blocks**

- Enable datapath extraction to **extract the 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 will help you write code to achieve these goals.

Tool Support

Synopsys' Design Compiler supports some of the guidelines described in this article in the sense that synthesis gives good results even if the code does not follow the guidelines. In other words, synthesis can efficiently handle bad code as well. This is mentioned in the affected guidelines. This effectively makes these specific guidelines obsolete, however, we still recommend to follow the guidelines for writing code that is unambiguous, clean, easy to read and portable.

General Guidelines

1. Signed Arithmetic

- **Rule:** Use type '`signed`' (VHDL, Verilog 2001) for *signed/2's complement arithmetic* (that is, 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:**
  - **Resources report:** Check for type and size of datapath operands (see **resources report**).
- **Tool Support:** *(partial)* Simple sign-extension of unsigned inputs (as in the first "bad QoR" example above) is fully supported by Design Compiler and automatically implemented as a signed operation. The second "bad QoR" example is not directly recognized as a simple signed multiplication, resulting in a larger circuit.

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. The code is also easier to read and coding is less error-prone.
- **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;`

- **Tool Support:** *(full)* Manual extensions are fully recognized by Design Compiler and do not affect the synthesis result.

Verilog Guidelines

3. Mixed unsigned/signed expression (Verilog)

- **Rule:** Do not mix *unsigned and signed* types in one expression. Mixed expressions are interpreted as unsigned in Verilog.
- **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:**
  - **Elaboration warnings:** Check for warnings about implicit unsigned-to-signed/signed-to-unsigned conversions/assignments (see **warning**).
  - **Resources report:** Check for operation type (unsigned or signed?) (see **resources report**).

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`").
- **Note:** *Concatenation* results are unsigned, regardless of the operands.
- **Rule:** Use type casts (`$signed`) to restore the correct type. Do not use part-selects that specify the entire vector.
- **Example:**

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

- **Checks:**
  - **Elaboration warnings:** Check for warnings about implicit unsigned-to-signed/signed-to-unsigned conversions/assignments (see **warning**).
  - **Resources report:** Check for operation type (unsigned or signed?) (see **resources report**).

5. Expression widths (Verilog)

- **Note:** The width of an expression in Verilog is determined as follows:
  - *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 parentheses) 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 seem to be. Then the expression takes the width of the higher-level context (for example, the left-hand side of an assignment).

  Example: Concatenation expression (Verilog)

Bad QoR	Good 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`
	`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 the widths of arithmetic expressions unambiguous (*context-determined* expressions).
- **Rationale:** *Better QoR* and/or *unambiguous behavior*.
- **Checks:**
  - **Resources report:** Check for implemented datapath blocks and size of input/output operands (see **resources report**).
- **Tool Support:** *(partial)* Design Compiler supports simple cases (like the "bad QoR" example above) where inputs just get extended. But it is thinkable that in more complex expressions these effects have more impact and can cause worse extraction and QoR.

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 several 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:**
  - **Resources report:** Check 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 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 (that is, 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:**
  - **Resources report:** Check for the number and functionality of implemented datapath blocks (see **resources report**).
  - **Datapath extraction analysis:** Message issued when sequential cells block extraction.

    ```
    Information: There is sequential cell in between
    operator associated with 'mult_11' and 'add_15'. (HDL-121)
    ```

    Blocked extraction due to user hierarchy is not reported.
- **Tool Support:** If datapath portions are spread across several levels of hierarchy, the blocks can be manually ungrouped (`ungroup` command) before compilation in order to move all datapath portions into the same hierarchy level.

8. Mixed unsigned/signed datapath

- **Rule:** Do not mix *unsigned and signed* types in a datapath cluster (that is, 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:**
  - **Elaboration warnings:** Check for warnings about implicit signed-to-unsigned conversions (see **warning**).

- **Resources report:** Check for implemented datapath blocks and the operations they implement (see **resources report**).
- **Datapath extraction analysis:** Message issued when datapath leakage due to sign mismatch blocks extraction.

```
Information: Operator associated with resources 'mult_5' in design ‘test' breaks
the datapath block because there is leakage due to driver/load sign mismatch.  (HDL-120)
```

- **Tool Support:** *(full)* Mixed-sign extraction is now fully supported in Design Compiler. Datapath extraction can still be blocked when mixed-sign expressions lead to *datapath leakage*.

9. Switchable unsigned/signed datapath

- **Rule:** Use selective zero-/sign-extension for implementing *switchable unsigned/signed datapath* (that is, 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

```
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
```

Good QoR

```
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:**
  - **Resources report:** Check for implemented datapath blocks and size/type of operands (see **resources report**).
- **Tool Support:** *(none)*.

10. Product of Sum (POS) expressions

- **Rule:** Make use of *POS expressions* (that is, add-multiply structures, "`(a + b) * c`").
- **Rationale:** Synthesis can implement POS expressions efficiently by 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

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

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);
```

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

- **Tool Support:** *(partial)* Depending on how different the code is when using or avoiding POS Design Compiler can give similar or very different QoR. If the code is very different, it is not possible to synthesize the same datapath with same QoR.

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. Also, do not *hand-craft arithmetic circuits* with logic gates (logic operators or GTECH cells).
- **Rationale:** *Better QoR* can be obtained from RTL expressions by exploiting the full potential of datapath extraction and synthesis (for example, by using implicit carry-save formats internally).
- **Example:** Multiply-accumulate unit (Verilog)

Bad 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));
```

Good QoR

```
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:**
  - **Resources report:** Check for implemented datapath blocks (see **resources report**).
  - **Datapath extraction analysis:** Message issued when DW component instantiations block extraction.

```
Information: Cell 'mult' in design 'guideline_11_a' can not be extracted because it
instantiates 'DW02_multp', which could be inferred with operator '*' instead.  (HDL-125)
...
Information: Cell 'csa0' in design 'guideline_11_a' can not be extracted because it
instantiates 'DW01_csa', which could be inferred with operator '+/-' instead.  (HDL-125)
...
Information: Cell 'add0' in design 'guideline_11_a' can not be extracted because it
instantiates 'DW01_add', which could be inferred with operator '+/-' instead.  (HDL-125)
```

- **Tool Support:** *(none)* Instantiated DesignWare components are currently excluded from datapath extraction in Design Compiler.

12. Pipelining

- **Rule:** For *pipelining of datapaths*, place the pipeline registers at the inputs or outputs of the RTL datapath and use retiming ('`optimize_registers`' in Design Compiler; see "Design Compiler Reference Manual: Register Retiming") to move them to the optimal locations. 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.
- **Example:** Multiply-accumulate unit with 3 pipeline stages (Verilog)

Verilog code	Sample script

`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`
	1-pass retiming flow
	
	`set_optimize_registers`
	`compile_ultra`
	
	2-pass retiming flow
	
	`set period 1.0`
	`set num_stages 3`
	`set scale 1.0`
	
	`analyze -f verilog mac_pipe.v`
	`elaborate mac_pipe`
	
	`# adjust clock period for pipelined parts`
	`# (multiply target period by number of stages before retiming)`
	`# try some iteration by reducing the scale_factor (e.g. between 1.0 and 0.7)`
	`# and choose the one that gives the best results`
	`create_clock clk -period [expr $period * $num_stages * $scale]`
	`compile_ultra`
	
	`# exclude input/output registers from retiming`
	`set_dont_retime *_reg_reg* true`
	
	`# reset to actual clock period and retime`
	`# -no_compile option prevents automatic incremental compile so that`
	`# incremental compile can be run afterwards with preferred options`
	`create_clock clk -period $period`
	`optimize_registers` `-no_compile`
	
	`# run incremental compile`
	`compile_ultra –incremental`
	
	`# This is a bottom-up approach.  A top-down solution is`
	`# also possible (1-pass retiming flow).  Find more information on retiming in the`
	`# "Design Compiler Reference Manual: Register Retiming" or`
	`# contact support.`

13. Complementing an operand

- **Rule:** Do not complement (negate) operands manually by inverting all bits and adding 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 the first example in the following item.
- **Tool Support:** *(partial)* Design Compiler converts simple manual complements (like "`a_neg = ~a + 1`") into negations/subtractions ("`a_neg = -a`"). But it doesn't optimize more complex cases as in the example below.

negations/subtractions (`a_neg = -a`). But it doesn't optimize more complex cases as in the example below.

- **Note:** When a complement and a multiplication are involved, it is often beneficial to do the complement at the inputs rather than the output (see examples in the following guideline). In simple cases this is done automatically by Design Compiler.

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;`
	
	`// complement multiplier using (+1/-1) multiply`
	`assign z = ($signed({sign, 1'b1}) * a) * b + c;`
	`// -> SOP (complement + carry-save 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;`
	`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)`

- **Tool Support:** *(partial)* Only certain high-level transformation of this kind are done automatically. The simple transformation to move a complement from the output of a multiplier to the input is supported by Design Compiler. In more complex cases an automatic transformation might not be feasible.

15. Parallel constant multipliers

- **Note:** Parallel constant multipliers, as in the example below, can be optimized for area by implementing them using shifts and adds and sharing common constant terms.
- **Example:**

Better timing	Better area

`input  signed  [7:0] a;`
`output signed [15:0] x, y, z;`
 	
`// parallel constant multiplies`
`assign x = a * 107;`
`assign y = a * 75;`
`assign z = a * 91;`
`// -> fastest implementation`
	`input  signed  [7:0] a;`
	`output signed [15:0] x, y, z;`
	`wire   signed [15:0] t;`
	
	`// shift-and-add, common terms shared`
	`assign t = a + (a<<1) + (a<<3) + (a<<6);`
	`assign x = t + (a<<5);`
	`assign y = t;`
	`assign z = t + (a<<4);`
	`// -> worse delay, but better area`

- **Tool Support:** *(full)* The above transformation to optimize for area (power) is done automatically in Design Compiler. There is no need to optimize parallel constant multipliers in the RTL code.

16. Common subexpression sharing and unsharing

- **Note:** Common subexpressions among different arithmetic expressions can be shared (to improve area) or unshared (to improve timing).
- **Example:**

Better timing	Better area

`input  signed [7:0] a, b, c, d, e;`
`output signed [7:0] x, y;`
 	
`// unshared expressions`
`assign x = a + b - c + d;`
`assign y = a - c + d - e;`
`// -> fastest implementation`
	`input  signed [7:0] a, b, c, d, e;`
	`output signed [7:0] x, y;`
	`wire   signed [7:0] t;`
	
	`// shared common subexpression`
	`assign t = a - c + d;`
	`assign x = t + b;`
	`assign y = t - e;`
	`// -> worse delay, but better area`

- **Tool Support:** *(full)* Unsharing (to optimize for timing) and sharing (to optimize for area) is automatically done in Design Compiler based on the timing context, no matter whether sharing is done in the RTL code or not. Therefore the two code examples will result in the same QoR.

17. Rounding

- **Note:** The choice of rounding mode as well as how it is implemented can affect datapath extraction. All asymmetric rounding modes can be implemented by adding constant bits at certain locations, resulting in optimal datapath extraction. However, the symmetric (unbiased) rounding modes require accessing individual (variable) bits from the operand to be rounded, which in turn requires this operand to be in binary representation and therefore blocks extraction.

  Summary of rounding modes:

Rounding mode	IEEE [1]	Rounding direction	Tie resolution	Symmetric	DP Extraction [2]
Round down [4]	floor	Towards minus infinity	None	No	Yes
Round up	ceil	Towards plus infinity	None	No	Yes
Round towards zero	trunc	Towards zero	None	Signed only [3]	Yes
Round away from zero		Away from zero	None	Signed only [3]	Yes
Round to even		To even	None	Yes	No
Round to odd		To odd	None	Yes	No
Round half down		To nearest	Towards minus infinity	No	Yes
Round half up [5]		To nearest	Towards plus infinity	No	Yes
Round half towards zero		To nearest	Towards zero	Signed only [3]	Yes
Round half away from zero		To nearest	Away from zero	Signed only [3]	Yes
Round half to even	round	To nearest	To even	Yes	No
Round half to odd		To nearest	To odd	Yes	No

1) IEEE standard rounding modes.
2) Datapath extraction is possible when rounding can be done by addition of constant bits only.
3) No overall bias if numbers are positive and negative with equal probability.
   However, positive bias exists for positive numbers and negative bias for negative numbers.
4) Corresponds to truncation in hardware.
5) Most common asymmetric rounding mode in hardware because of lowest complexity (add 1 at `lsb-1`).

- **Rule:** Use *asymmetric rounding* whenever possible. Perform asymmetric rounding by *adding constant bits* only.
- **Rule:** Implement rounding mode 'round half up' by *adding a constant 1* at bit position `lsb-1`. Do not add variable bits for rounding if possible.

- **Rationale:** Adding constant bits allows better datapath extraction and gives better QoR. Adding variable bi requires binary conversion and can block datapath extraction.
- **Example:**

Bad QoR	Good QoR

`input  signed  [7:0] a, b;`
`output signed  [7:0] z;`
`wire   signed [15:0] t;`
 	
`// rounding mode: round half up`
`// adding variable bit at lsb`
`assign t = a * b;`
`assign z = t[15:8] + t[7];`
`// -> t needs to be converted to binary`
	`input  signed  [7:0] a, b;`
	`output signed  [7:0] z;`
	`wire   signed [15:0] t;`
	
	`// rounding mode: round half up`
	`// adding constant 1 at lsb-1`
	`assign t = a * b + (1<<7);`
	`assign z = t[15:8];`
	`// -> implemented by single SOP`

- **Checks:**
  - **Resources report:** Check for the number and functionality of extracted datapath blocks (see **resources report**).
- **Tool Support:** *(Partial)* In some simple cases the tool is able to do automatic conversion.
  - All rounding modes from the table above are available in **DesignWare Datapath Functions** (bottom of the document) that are implemented for best possible datapath extraction and QoR.

18. Avoid excessive saturation

- **Note:** The saturation of an operand that is internal to a datapath
  - reduces the width of the driven operations → better QoR,
  - can block datapath extraction because saturation requires a binary conversion → worse QoR.

  Saturation therefore is a QoR trade-off. Doing saturation at certain locations in a datapath where operand widths can be reduced significantly can be beneficial. However, excessive saturation on many operands with small width reductions however can hurt QoR more than it helps.
- **Rule:** *Avoid excessive saturation* on internal operands to reduce operand widths. Better carry some extra bits further and saturate at the end. Only do saturation when the width reduction is significant.
- **Example:**

Bad QoR	Good QoR

`// internal saturation`
`input   [7:0] a, b;`
`input   [7:0] c;`
`output  [8:0] z;`
`wire   [15:0] p;`
`wire    [7:0] s;`
 	
`// multiply`
`assign p = a * b;`
`// saturate & truncate`
`assign s = (p > 8'b11111111) ?`
`8'b11111111 : p[7:0];`
`// add saturated value`
`assign z = s + c;`
`// -> 2 carry-propagations`
	`// saturation at the end`
	`input   [7:0] a, b;`
	`input   [7:0] c;`
	`output  [8:0] z;`
	`wire   [16:0] p;`
	
	`// multiply & add`
	`assign p = a * b + c;`
	`// saturate & truncate`
	`assign z = (p > 9'b111111111) ?`
	`9'b111111111 : p[8:0];`
	`// -> 1 carry-propagation`

- **Checks:**
  - **Resources report:** Check for the number and functionality of extracted datapath blocks (see **resources report**).

19. Avoid datapath leakage

- **Note:** datapath leakage is when an internal operand is not wide enough to store the result of an operation, but the full result is required later. This is caused by upper-bit truncation followed by a later extension.
  - Upper-bit truncation
    - Truncated MSB is never used

```
assign t[7:0] = a[7:0] + b[7:0];  // truncation
assign z[7:0] = t[7:0] + c[7:0];  // no extension
```

  - Upper-bit extension (sign-/zero-extension)
    - MSB for correct extension is available

```
assign t[8:0] = a[7:0] + b[7:0];  // no truncation
assign z[9:0] = t[8:0] + c[7:0];  // extension
```

  - Upper-bit truncation + later extension = **datapath leakage**
    - Truncated MSB is required but not there for correct extension → arithmetically incorrect results

```
assign t[7:0] = a[7:0] + b[7:0];  // truncation
assign z[8:0] = t[7:0] + c[7:0];  // extension
```

  - Types of datapath leakage
    - **True leakage:** internal operand is not wide enough to hold all values that can occur in an application → incorrect behavior (e.g. in simulation).
    - **False leakage:** internal operand is wide enough to hold all values that can occur in an application → correct behavior.
    - Synthesis cannot distinguish between the two → all leakage can impact synthesis.
  - Operands with leakage *must be binary* (cannot be carry-save).
    - Must be at the boundary of a DP block.
    - Prevent extraction with the driven operation
  - datapath leakage due to *mixed sign*
    - Conversion of a signed into an unsigned operand → operand has always upper-bit truncation.
    - Unsigned subtraction → operand has always upper-bit truncation.
- **Rule:** Avoid upper-bit truncation that results in *datapath leakage* on internal results. Make internal operands wide enough to hold the result of the driving operation (also for false leakage).
- **Rule:** Avoid the conversion of signed to unsigned operands.
- **Rule:** For unsigned subtractions make sure the resulting operand is as wide as the final result of the datapath (i.e. it is not followed by an extension that would result in leakage).
- **Rationale:** Better QoR and/or correct behavior of arithmetic expressions.
- **Example:**

Bad QoR	Good QoR

`input  [7:0] a, b, c;`
`input [15:0] d;`
`output [16:0] z;`
`wire  [15:0] p;`
 	
`// 8*8+8=17 bits, assigned to 16 bits (MSB truncated)`
`assign p = a * b + c;`
	`input  [7:0] a, b, c;`
	`input [15:0] d;`
	`output [16:0] z;`
	
	`wire  [16:0] p;  // wide enough`
	
	`// 8*8+8=17 bits, assigned to 17 bits`

// used in 17 bit expression (extended) -> datapath leakage
assign z = p + d;
// -> 2 carry-propagations
	assign p = a * b + c;
	// used in 17 bit expression -> no datapath leak
	assign z = p + d;
	// -> 1 carry-propagation

- **Checks:**
  - **Resources report:** Check for the number and functionality of extracted datapath blocks (see **resources report**).
  - **Datapath extraction analysis:** Message issued when datapath leakage blocks extraction.

```
Information: Operator associated with resources 'add_9' in design ‘test' breaks
the datapath block because there is leakage due to truncation on its fanout.  (HDL-120)
```

Synthesis Reports

Design Compiler gives feedback on potential problems and on how datapaths were synthesized through several reports.

Synthesis Flow in Design Compiler

```
analyze -f verilog test.v
        ⇓
elaborate test                  ⇒  RTL elaboration warnings
        ⇓
analyze_datapath_extraction     ⇒  Datapath extraction analysis
        ⇓
compile_ultra
        ⇓
analyze_datapath
report_resources -hier          ⇒  Datapath synthesis reports
```

RTL Elaboration Warnings

- Warnings about implicit *signed-to-unsigned conversions.*

RTL:

```
input  signed [7:0] a;
input         [7:0] b;

output        [7:0] z;
assign z = a + b;
```

Warning:

```
Warning: ./test.v:7: signed to unsigned conversion occurs. (VER-318)
```

Remedy:

```
input  signed [7:0] a;
input         [7:0] b;
output        [7:0] z;
assign z = $unsigned(a) + b;
```

- Warnings about implicit *unsigned-to-signed* or *signed-to-unsigned assignments*.

RTL:

```
input         [7:0] a;
input         [7:0] b;
output signed [7:0] z;
assign z = a + b;
```

Warning:

```
Warning: ./test.v:7: unsigned to signed assignment occurs. (VER-318)
```

Remedy:

```
input         [7:0] a;
input         [7:0] b;
output signed [7:0] z;
assign z = $signed(a + b);
```

- **Note:** These warnings can indicate unwanted behavior (is the conversion / assignment really meaningful?). When this behavior is wanted, the warnings can be avoided with the explicit type casts in the examples above.

Datapath Extraction Analysis

Datapath extraction analysis can be run before a design is compiled to get the following information:

- Points out violations of coding guidelines in the RTL.
- Gives feedback on datapath extraction and messages about why extraction is blocked.

This feature is very useful to get quick feedback on datapath extraction without running a time-consuming full compilation.

Note: Only coding guidelines that affect datapath extraction can be checked. The presence of messages does not always mean the code can be improved. Also, the absence of messages does not mean everything is optimal.

Command:

```
analyze_datapath_extraction [-no_autoungroup] [-no_report]
```

- Option `-no_autoungroup`: Corresponds to the same option of the `compile_ultra` command. When this option is used during compile it must also be used during datapath extraction analysis in get the same datapath extraction.

- Option `-no_report`: By default this command prints out a resources report (similar to `report_resources`) at the end for matching resources names. With this option the resources report is suppressed (much shorter output).

Messages:

HDL-120	datapath leakage blocks extraction
HDL-121	Register blocks extraction
HDL-125	Instantiated DW component cannot be extracted

Example:

```
Information: Operator associated with resources 'add_9' in design ‘test' breaks
the datapath block because there is leakage due to truncation on its fanout.  (HDL-120)
```

Availability:

- New in **G-2012.06** release.

Datapath Content Report

Report the datapath content of a design (summary).

Command:

```
analyze_datapath
```

- Summary of number and area contribution of DW components and datapath blocks (percentage of total area given).

```
Estimate of Datapath Content
Extracted Datapath:    area =  55.6%,  parts =  37
Singleton Datapath:    area =  16.1%,  parts =  33
Total Datapath:        area =  71.7%,  parts =  70
```

- Summary of number and type of instantiated and inferred DesignWare components (singletons).

```
Singletons:
                              out width
  component     count    min    max     ave
  ----------------------------------------
  DW01_add         11     10     16    13.2
  DW01_sub          5      9     12    10.4
  DW01_inc          2      6      6     6.0
  DW01_dec          4      6      8     7.0
  DW_cmp            2     16     20    18.0
  DW_leftsh         1     12     12    12.0
  DW_rightsh        3      5     12     8.6
  DW_mult_uns       1     44     44    44.0
  DW_mult_tc        4      8     12    11.0
```

- Summary of number and type of operators that are extracted into datapath blocks.

```
Extracted Datapaths:
                              out width
  operation     count    min    max     ave
  ----------------------------------------
  + -               9      4     40    23.9
  *                 3     27     35    31.3
  ?                 2     16     40    28.0
  << >>            20     12     16    13.8
  == !=             2     14     15    14.5
  < <= > >=         1     15     15    15.0
```

  - **Note:** for comparators the input width is given (output is always just 1 bit).

Resources Report

- Report all instantiated / inferred DW components for each design module.
- Report extracted DP blocks for each design module.

Command:

```
report_resources -hier
```

Example RTL code:

```
module test (a, b, c, d, e, s, x, y);

  input  signed  [7:0] a, b, c, d, e;
  input                s;
  output signed [15:0] x, y;
  wire   signed [15:0] t;

  assign x = a * b;
  assign t = c * d;
  assign y = (s ? -t : t) + e;

endmodule
```

Example Resources Report: Report for complex datapath and singleton component

```
Resource Report for this hierarchy in file ./test.v
=======================================================================================
| Cell              | Module           | Parameters | Contained Operations |
=======================================================================================
| mult_x_8_1        | DW_mult_tc       | a_width=8  | mult_8               |
|                   |                  | b_width=8  |                      |
| DP_OP_6_296_5095 | DP_OP_6_296_5095 |            |                      |
=======================================================================================
```

Report of modules extracted from RTL code, for singleton components and complex datapath blocks.

The names of contained operations (e.g., "mult_8") are made up from the operation ("mult") and the line number in the RTL code

where they appear ("8").

```
Datapath Report for DP_OP_6_296_5095
=================================================================================
| Cell                  | Contained Operations                                  |
=================================================================================
| DP_OP_6_296_5095     | mult_9 sub_10 add_10                                  |
=================================================================================

=================================================================================
|        |      | Data     |       |                                           |
| Var    | Type | Class    | Width | Expression                                |
=================================================================================
| I1     | PI   | Signed   | 8     |                                           |
| I2     | PI   | Signed   | 8     |                                           |
| I3     | PI   | Unsigned | 1     |                                           |
| I4     | PI   | Unsigned | 1     |                                           |
| I5     | PI   | Signed   | 8     |                                           |
| T1     | IFO  | Signed   | 16    | I1 * I2                                   |
| T2     | IFO  | Signed   | 16    | $signed(1'b0) - T1                        |
| T4     | IFO  | Signed   | 16    | { I3, I4 } ? T2 : T1                      |
| O1     | PO   | Signed   | 16    | T4 + I5                                   |
=================================================================================
```

Report for complex datapath blocks. Includes contained RTL operations, interface (input ports PI and output ports PO) and functionality. Internal ports IFO are in carry-save format whenever possible/beneficial.

The names of contained operations (e.g., "mult_8") are made up from the operation ("mult") and the line number in the RTL code where they appear ("8").

```
Implementation Report

=======================================================================
|                   |                  | Current        | Set            |
| Cell              | Module           | Implementation | Implementation |
=======================================================================
| DP_OP_6_296_5095 | DP_OP_6_296_5095 | str (area)     |                |
| mult_x_8_1        | DW_mult_tc       | pparch         |                |
=======================================================================
```

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. In parenthesis the optimization goal during datapath generation is given (such as 'area', 'speed', 'area,speed', 'power', 'power,speed').

For singletons (that is, individual operations implemented by a discrete DesignWare component), the according implementation name is reported (see datasheets).

 2012-06-01