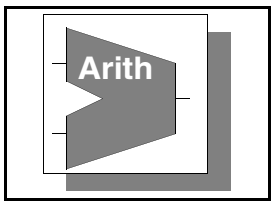
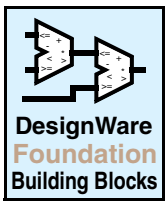


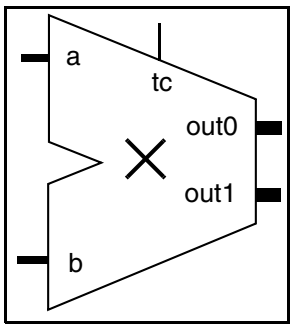


DW02_multp

Partial Product Multiplier

Version, STAR and Download Information: IP Directory

Features and Benefits



- Parameterized word lengths
- Parameterized sign extension of partial product outputs for use in summing products
- Unsigned and signed (two's-complement) data operation
- Parameter control over carry-save (CS) design verification method

Description

DW02_multp determines the partial product of the operands `a` and `b`. The multiplier does not use a final carry-propagate adder and the result is in redundant carry-save format. The actual product of `a` and `b` is the sum of the DW02_multp outputs `out0` and `out1`, where `a × b = (out0 + out1) mod` 2^{out_width}. The control signal `tc` determines whether the input data is interpreted as unsigned (`tc` = 0) or signed (`tc` = 1) numbers.

Table 1-1 Pin Description

Pin Name	Width	Direction	Function
a	*a_width* bit(s)	Input	Multiplier
b	*b_width* bit(s)	Input	Multiplicand
tc	1 bit	Input	Two's complement control: 0: Unsigned 1: Signed
out0	*out_width* bit(s)	Output	Partial product of (`a` × `b`)
out1	*out_width* bit(s)	Output	Partial product of (`a` × `b`)

Table 1-2 Parameter Description

Parameter	Values	Description
a_width	≥ 1	Word length of `a`
b_width	≥ 1	Word length of `b`
out_width	≥ *a_width* + *b_width* + 2	Word length of `out0` and `out1`

Table 1-2 Parameter Description (Continued)

Parameter	Values	Description
verif_en[a]	0 to 3 Default: 2	Verification Enable Control 0: Outputs `out0` and `out1` are always the same for a given input pair (`a`, `b`) 1: MSB of `out0` is always '0'; `out0` and `out1` change with time (are random) for the same input pair (`a`, `b`) 2: MSB of `out0` or `out1` is always '0'; `out0` and `out1` change with time (are random) for the same input pair (`a`, `b`) 3: No restrictions on MSBs of `out0` and `out1`; `out0` and `out1` change with time (are random) for the same input pair (`a`, `b`)

a. Although the *verif_en* value can be set for all simulators, carry-save (CS) randomization is only implemented when using Synopsys simulators (VCS, VCS-MX). For more information about *verif_en*, refer to "Simulation Using Random Carry-save Representation (VCS/VCS-MX only)" on page 4.

Table 1-3 Synthesis Implementations

Implementation Name	Function	License Feature Required
pparch[a]	Delay-optimized flexible parallel-prefix	DesignWare
apparch[a]	Area-optimized flexible parallel-prefix	DesignWare

a. The 'pparch' (optimized for delay) and 'apparch' (optimized for area) implementations are dynamically generated to best meet your constraints. The 'pparch' and 'apparch' implementations can generate a variety of multiplier architectures including Radix-2 non-Booth, Radix-4 non-Booth, Radix-4 Booth recoded and Radix-8 Booth recoded. Generation of 'pparch' and 'apparch' implementations automatically detects which input would be best suited to be considered the multiplier as opposed to the multiplicand (for Booth recoding). The pparch and apparch implementations are generated making use of any special arithmetic technology cells that are found to be available in your target technology library. The `dc_shell` command, `set_dp_smartgen_options`, can be used to force specific multiplier architectures. For more information on forcing generated arithmetic architectures, use 'man set_dp_smartgen_options' (in dc_shell) to get a listing of the command options.

Table 1-4 Simulation Models

Model	Function
DW02.DW02_MULTP_CFG_SIM	Design unit name for VHDL simulation
dw/dw02/src/DW02_multp_sim.vhd[a]	VHDL simulation model source code
dw/sim_ver/DW02_multp.v	Verilog simulation model source code

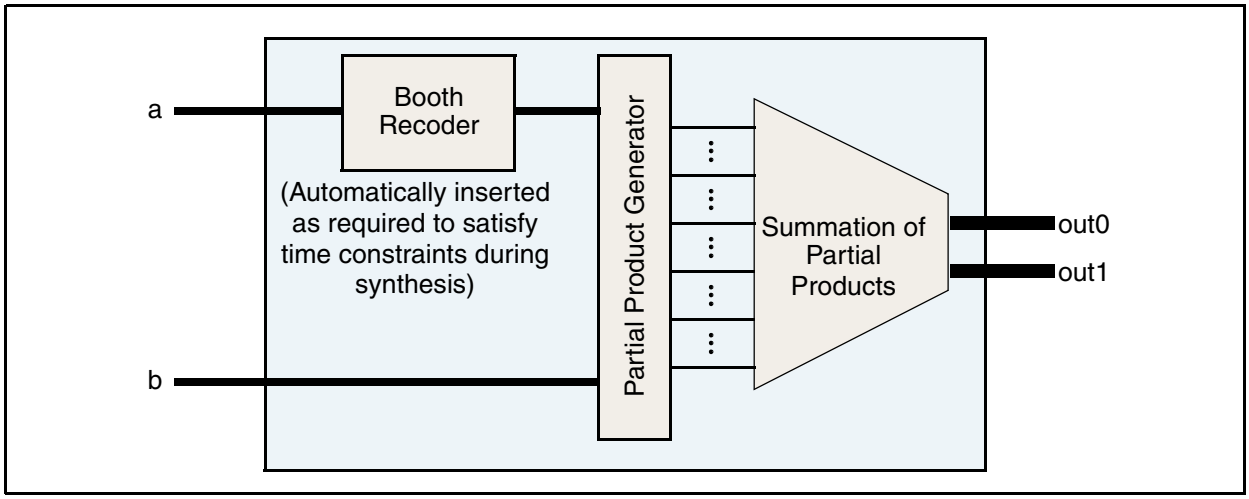
a. This is a plain-text simulation model file for use with 3rd-party VHDL simulators, and parenthetically does not support the *verif_en* control of CS random simulation.

Attention The simulation architecture does not produce the same values on `out0` and `out1` as produced by the synthetic architecture, but once added together by a component such as DW01_add, the resulting SUM (discarding the carry-out bit) is the same for the synthetic and simulation architectures.

Functional Description

Figure 1-1 shows a high-level block diagram of DW02_multp. The use of Booth Recoder depends on the parameter values and time constraints during synthesis. For example, it may not be used if the operands have a small bit width and time constraint is loose. The Summation of Partial Products may change from a simple carry propagate structure to a more parallel tree, depending on the delay constraint.

Figure 1-1 DW02_multp Block Diagram



The two partial product outputs are signed numbers, regardless of whether the `tc` input indicates that the `a` and `b` inputs are signed or not.

If fewer than *a_width* + *b_width* + 2 bits are required for the final product or sum of products, you may disregard any unnecessary upper bit(s) of `out0` and `out1`. The DWBB Booth Wallace Tree multiplier, DW02_mult(wall), is constructed from a DW02_multp (with *out_width* set to *a_width* + *b_width* + 2) followed by an instance of DW01_add, which adds together only the lower *a_width* + *b_width* bits of `out0` and `out1`. However, the DWBB non-Booth Wallace Tree multiplier, DW02_mult (nbw) is not constructed from DW02_multp.

When summing many products and/or adding a product to a number of greater size, DW02_multp performs sign extension using the *out_width* parameter. If the designer wants to extend the output of DW02_multp even further, then sign extension of both vectors (`out0` and `out1`) is required to generate correct extended CS representation.

Note in Table 1-1 that port `a` is defined as the "multiplier", and therefore is Booth recoded (see Figure 1-1 on page 3). Therefore, if constant multiplication is desired, the constant should be connected to port a. When multiplying numbers of different size, since port `a` gets Booth recoded, a faster and smaller multiplier is obtained by having the smaller operand connect to port `a`.

Simulation Using Random Carry-save Representation (VCS/VCS-MX only)

The carry-save (CS) representation of product `a` * `b` is "redundant" and, therefore, there are many ways to represent the same product value. The simulation model of DW02_multp (most likely) does not match the behavior of the synthesized circuit of this component, although at all times their outputs are numerically equivalent ((`out0` + `out1`) mod 2^{out_width} = `a` * `b`)

Instead of having a "static" behavior, the simulation models of DW02_multp have the parameter *verif_en* to let you adjust the level of randomness in the CS representation of the output. This parameter applies only to simulation models. The term "static" is used here to denote when the CS representation of `a` * `b` is always the same. Such a behavior is not ideal for verification of designs that manipulate the CS values produced by DW02_multp because the design that instantiates DW02_multp should work independently of any particular "static" implementation of DW02_multp. A "random" CS representation of the output is one that still represents `a` * `b` but changes every time a new pair of inputs is applied. The random behavior has a better chance of exposing issues in the use of CS representation, but it is not full proof that the design works properly for any implementation of DW02_multp. However, it does provide better coverage than the static simulation model.

There are 4 possible levels of control for the behavior of the CS representations produced by DW02_multp:

1. **"static" CS representation (*verif_en* = 0):** Allows sign extension of the CS representation. The MS bit of `out0` is always '0'.
2. **"random" CS representation (*verif_en* = 1)**: Allows sign extension of the CS representation. The MS bit of `out0` is always '0'. The other bits are formed based on a random `out1` vector.
3. **"random" CS representation (*verif_en* = 2 (default))**: Allows sign extension of the CS representation. One of the MS bits of `out0` or `out1` is always '0'. The MSB of `out0` and `out1` are not '1' at the same time. The other bits are formed based on a random `out1` vector.
4. **"random" CS representation (*verif_en* = 3)**: Does not allow direct sign extension of the CS representation. All bits of `out0` and `out1` are randomly selected, but still represent `a` * `b`.

Using this mechanism, the designer has a better simulation environment to discover design problems related to incorrect CS manipulation. These problems could be masked by a static behavior of the simulation model, and could manifest later after synthesis of DW02_multp. The default value defined for *verif_en* gives the designer more assurance that the CS representations generated by any type of implementation of DW02_multp will be correctly handled in the design that uses it. Any circuit implementation created by DC for this component allows sign extension of the CS representation at the output.

Although the default value for *verif_en* is consistent with the present synthesis behavior in DC, we recommend using the more aggressive value (*verif_en* = 3) to have more assurance that CS representations generated by any type of implementation will be correctly handled in the design.

Table 1-5 shows the behavior of DW02_multp for a sequence of input pairs (hexadecimal values) and all possible *verif_en* values. The sequence repeats the input pair (24,81) to demonstrate the component behavior.

- When *verif_en* = 0, the output is always the same when (24,81) is applied, and the MS bit of `out0` is always '0'.
- When *verif_en* = 1, the output value that corresponds to input (24,81) is not always the same, but the MS bit of `out0` is always '0'.
- When *verif_en* = 2 (default), `out0` or `out1` has a '0' on the MS bit position.
- When *verif_en* = 3, the output has no constraint, and the MS bits of `out0` and `out1` may get any value. The last row of the table shows a case when both MS bits of `out0` and `out1` are '1' when *verif_en* = 3.

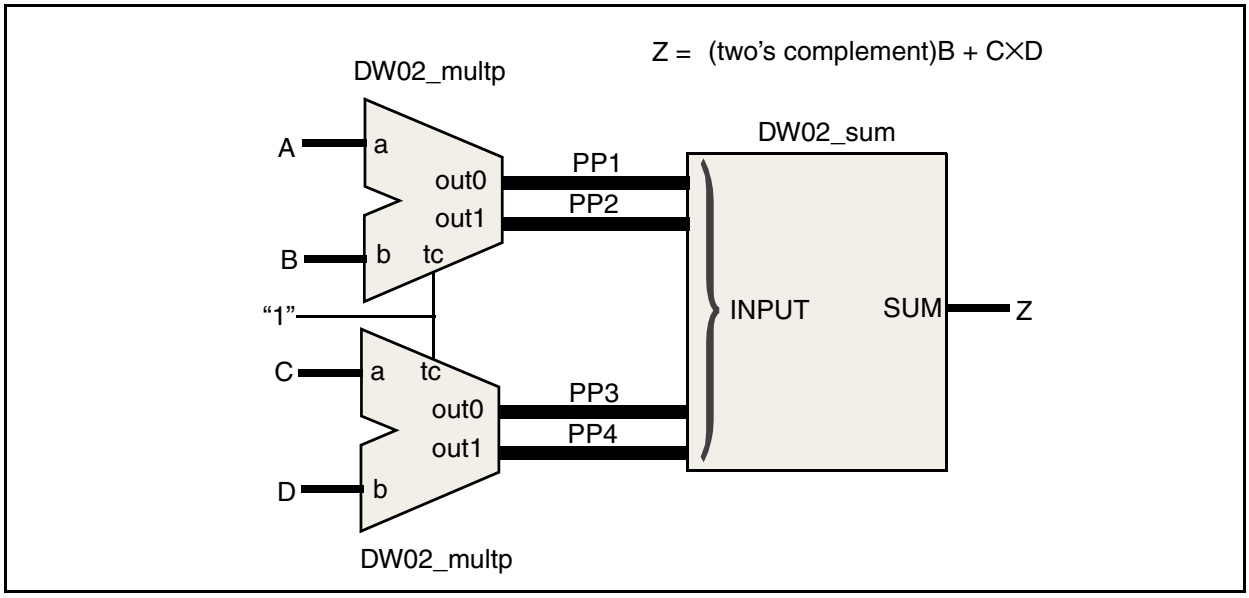
Table 1-5 DW02_multp behavior for *a_width* = 8, *b_width* = 8, *out_width* = 18

Inputs		verif_en = 0		verif_en = 1		verif_en = 2		verif_en = 3	
a	b	out0	out1	out0	out1	out0	out1	out0	out1
24	81	18924	28900	13C1B	2D609	13C1B	2D609	33C1B	0D609
64	0D	1FA87	20A80	06B7A	3998D	06B7A	3998D	06B7A	3998D
24	81	18924	28900	18DBF	28465	18DBF	28465	18DBF	28465
12	01	1FFD2	20040	03305	3CD0D	03305	3CD0D	03305	3CD0D
24	81	18924	28900	020AE	3F176	3F176	020AE	020AE	3F176
3D	ED	1B679	28200	040ED	3F78C	3F78C	040ED	240ED	1F78C
24	81	18924	28900	0282B	3E9F9	0282B	3E9F9	2282B	1E9F9
C6	C5	1675E	33100	0C5B4	3D2AA	3D2AA	0C5B4	0C5B4	3D2AA
24	81	18924	28900	0025B	00FC9	0025B	00FC9	2025B	29FC9

Application Notes

The sample application in Figure 1-2 illustrates how to use two instances of DW02_multp and one instance of DW02_sum to calculate the value of `(a×b) + (c×d)`. The resulting circuit from Figure 1-2 is smaller and faster than a similar circuit that uses two instances of DW02_mult and one instance of DW01_add. All the inputs in this circuit are considered signed integers, since the tc input of both DW02_multp instances have a constant value '1'.

Figure 1-2 Application Example



Although this is a valid example of the application of DW02_multp, it is recommended that for this type of application (sum of products), the designer use a hardware language description of the same operation, and let DC synthesize the circuit using data path generators, which deliver the best results for any given design constraint.

References

[1] Israel Koren, Computer Arithmetic Algorithms,
Prentice Hall, Englewood Cliffs, NJ 07632

Related Topics

- Math – Arithmetic Overview
- DesignWare Building Block IP Documentation Overview

HDL Usage Through Component Instantiation - VHDL

```
library IEEE,DWARE;
use IEEE.std_logic_1164.all;
use DWARE.DW_foundation_comp.all;

-- Multiply & Accumulate performed by instances of
-- DW02_multp, DW01_csa & DW01_add

entity DW02_multp_inst is
  generic ( inst_a_width : NATURAL := 6;
            inst_b_width : NATURAL := 8;
            inst_out_width : NATURAL := 18;
            inst_verif_en : INTEGER := 3 ); -- value 3 is the most aggressive
                                            -- verification mode
  port ( inst_a : in std_logic_vector(inst_a_width-1 downto 0);
         inst_b : in std_logic_vector(inst_b_width-1 downto 0);
         inst_c : in std_logic_vector(inst_out_width-1 downto 0);
         inst_tc : in std_logic;
         accum_inst : out std_logic_vector(inst_out_width-1 downto 0) );
end DW02_multp_inst;

architecture inst of DW02_multp_inst is
  signal part_prod1,
         part_prod2 : std_logic_vector(inst_out_width-1 downto 0);
  signal part_sum1, part_sum2 : std_logic_vector(inst_out_width-1 downto 0);
  signal tied_low, no_connect1, no_connect2 : std_logic;
begin
  -- Instance of DW02_multp to perform the partial
  -- multiply of inst_a by inst_b with partial product
  -- results at part_prod1 & part_prod2
  -- The value of verif_en does not affect the synthesis result
  U1 : DW02_multp
    generic map ( a_width => inst_a_width, b_width => inst_b_width,
                  out_width => inst_out_width, verif_en => inst_verif_en )
    port map ( a => inst_a, b => inst_b, tc => inst_tc,
               out0 => part_prod1, out1 => part_prod2 );

  -- Instance of DW01_csa used to add the partial products
  -- from inst_a times inst_b (part_prod1 & part_prod2) to
  -- the input inst_c in carry-save form yielding the two
  -- vectors, part_sum1 & part_sum2.
  U2 : DW01_csa
    generic map (width => inst_out_width)
    port map ( a => part_prod1, b => part_prod2, c => inst_c,
               ci => tied_low, sum => part_sum1, carry => part_sum2,
               co => no_connect1 );

  -- Finally, an instance of DW01_add is used to add the carry-save
```

```
    -- partial results together forming the final binary output
    U3 : DW01_add
      generic map (width => inst_out_width)
      port map ( A => part_sum1, B => part_sum2,
                 CI => tied_low, SUM => accum_inst, CO => no_connect2 );

    tied_low <= '0';
  end inst;

  -- pragma translate_off
  configuration DW02_multp_inst_cfg_inst of DW02_multp_inst is
    for inst
    end for; -- inst
  end DW02_multp_inst_cfg_inst;
  -- pragma translate_on
```

HDL Usage Through Component Instantiation - Verilog

```
// Multiply & ACcumulate performed by instances of
// DW02_multp, DW01_csa & DW01_add

module DW02_multp_inst( inst_a, inst_b, inst_c, inst_tc,
                        accum_inst);
  parameter a_width = 6;
  parameter b_width = 8;
  parameter out_width = 18;
  parameter verif_en = 3; // value 3 is the most aggressive
                          // verification mode

  input [a_width-1 : 0] inst_a;
  input [b_width-1 : 0] inst_b;
  input [out_width-1 : 0] inst_c;
  input inst_tc;
  output [out_width-1:0] accum_inst;

  wire [out_width-1 : 0] part_prod1, part_prod2;
  wire [out_width-1 : 0] part_sum1, part_sum2;
  wire tied_low, no_connect1, no_connect2;

  // Instance of DW02_multp to perform the partial
  // multiply of inst_a by inst_b with partial product
  // results at part_prod1 & part_prod2
  // The value of verif_en does not affect the synthesis result
  DW02_multp #(a_width, b_width, out_width, verif_en) U1
              ( .a(inst_a), .b(inst_b),.tc(inst_tc),
                .out0(part_prod1), .out1(part_prod2) );

  // Instance of DW01_csa used to add the partial products
  // from inst_a times inst_b (part_prod1 & part_prod2) to
  // the input inst_c in carry-save form yielding the two
  // vectors, part_sum1 & part_sum2.
  DW01_csa #(out_width) U2 (.a(part_prod1), .b(part_prod2), .c(inst_c),
                            .ci(tied_low), .sum(part_sum1),
                            .carry(part_sum2), .co(no_connect1) );

  // Finally, an instance of DW01_add is used to add the carry-save
  // partial results together forming the final binary output
  DW01_add #(out_width) U3 (.A(part_sum1), .B(part_sum2),
                            .CI(tied_low), .SUM(accum_inst),
                            .CO(no_connect2) );

  assign tied_low = 1'b0;

endmodule
```

Copyright Notice and Proprietary Information

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at https://www.synopsys.com/company/legal/trademarks-brands.html.

All other product or company names may be trademarks of their respective owners.

Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

Synopsys, Inc.
690 E. Middlefield Road
Mountain View, CA 94043

www.synopsys.com