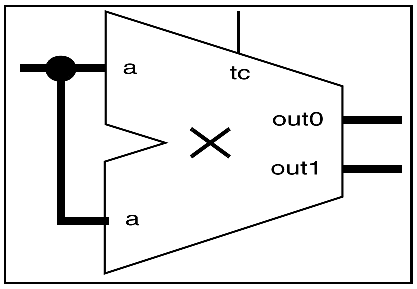
DW_squarep

Partial Product Integer Squarer

Version, STAR and Download Information: IP Directory

Features and Benefits

- Parameterized word lengths
- Unsigned and signed (two's-complement) data operation
- Parameter control over carry-save (CS) design verification method



Description

DW_squarep determines the partial products resulting from the multiplication of `a` by `a`. The actual product of `a` multiplied by `a` is the sum of the DW_squarep outputs, `out0` and `out1` `(a × a = out0 + out1)`.

Table 1-1 Pin Description

Pin Name	Width	Direction	Function
a	*width* bit(s)	Input	Multiplier
tc	1 bit	Input	Two's complement control: 0 = unsigned 1 = signed
out0	*width* × 2 bit(s)	Output	Partial product of `a` × `a`
out1	*width* × 2 bit(s)	Output	Partial product of `a` × `a`

Table 1-2 Parameter Description

Parameter	Values	Description
width	≥ 1	Word length of input signal `a`
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 3.

Table 1-3 Synthesis Implementations

Implementation Name	Function	License Feature Required
pparch[a]	Delay-optimized flexible parallel-prefix	DesignWare
apparch[a]	Area-optimized flexible architecture that can be optimized for area, for speed, or for area, speed	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 (squarer) architectures including Radix-2 non-Booth, Radix-4 non-Booth, Radix-4 Booth recoded and Radix-8 Booth recoded. 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 (squarer) 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.DW_SQUAREP_CFG_SIM	Design unit name for VHDL simulation
dw/dw02/src/DW_squarep_sim.vhd[a]	VHDL simulation model source code
dw/sim_ver/DW_squarep.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 is the same for the synthetic and simulation architectures. In other words, (`out0` + `out1`) mod $2^{2*width}$ is the same in both cases.

Functional Description

The DW_squarep is discussed in the following table and description.

Table 1-5 Functional Description

tc	a	out0 and out1
0	a (unsigned)	Partial product of a × a (unsigned)
1	a (two's complement)	Partial product of a × a (two's complement, but always positive which could be treated as an unsigned number if desired)

The control signal (tc) determines whether the input data is interpreted as an unsigned (tc = 0) or signed (tc = 1) number.

The sample application in Figure 1-1 illustrates how to use an instance of DW_squarep, an instance of DW02_multp, and an instance of DW02_sum to calculate the value of $A^2 + (B \times C)$. The resulting circuit from Figure 1-1 is smaller and faster than a similar circuit that uses two instances of DW02_mult and one instance of DW01_add.

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

The carry-save (CS) representation is a "redundant" representation and, therefore, there are many ways to represent the same value. The simulation model of DW_squarep (most likely) does not match the behavior of the synthesized circuit of this component. Although the results are completely different, they are still numerically equivalent ((out0 + out1) mod $2^{2*width}$ = a^2).

Instead of having only a "static" behavior, the simulation models of DW_squarep 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^2 is always the same. Such a behavior is not ideal for verification of designs that manipulate the CS values produced by DW_squarep because the design that instantiates the component should work independently of any particular "static" implementation of DW_squarep. A "random" CS representation of the output is one that still represents a^2 but changes every time a new input is applied. The random behavior has a better chance of exposing issues in the manipulation of values in the CS representation, but it is not full proof that the design works properly for any implementation of DW_squarep. However, it does provides better coverage than the static simulation model.

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

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 on the design cycle, after synthesis of DW_squarep. The default value defined for *verif_en* gives the designer more assurance that the CS representations generated by any type of implementation of DW_squarep 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.

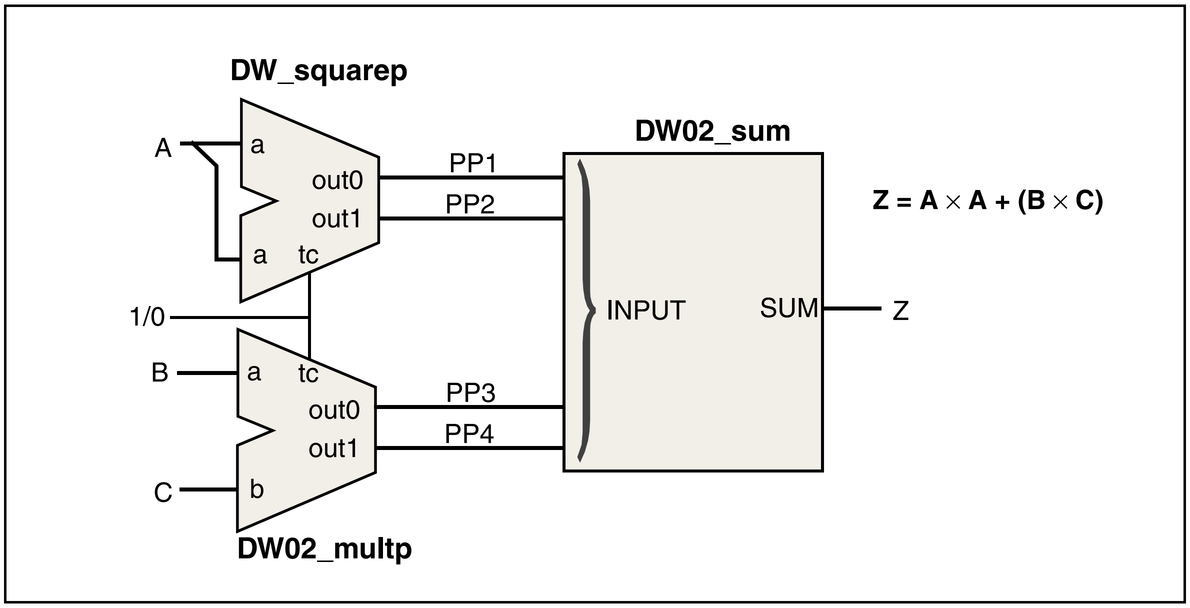
Table 1-6 shows the behavior of DW_squarep for a sequence of inputs (hexadecimal values) and all possible *verif_en* values. The input sequence repeats to demonstrate the component behavior.

When *verif_en* = 0, the output is always the same when a given input is applied. Observe that the output behavior matches the description provided for each *verif_en* value. In particular, when *verif_en* = 3, it may be the case that the output has the MS bits of both `out0` and `out1` with a value of '1'. This is the situation when sign extension would not work.

Table 1-6 DW_squarep Behavior for *verif_en* Values

a	tc	(out0, out1) verif_en = 0	(out0, out1) verif_en = 1	(out0, out1) verif_en = 2	(out0, out1) verif_en = 3
37	0	0BD1, 0000	0A6E, 0163	0A6E, 0163	7E6E, 8D63
A4	0	6110, 0800	62C5, 064B	62C5, 064B	72C5, F64B
37	0	0BD1, 0000	09F1, 01E0	01E0, 09F1	29F1, E1E0
A4	0	6110, 0800	5055, 18BB	5055, 18BB	9055, D8BB
37	0	0BD1, 0000	0650, 0581	0650, 0581	F650, 1581
A4	0	6110, 0800	4AE9, 1E27	4AE9, 1E27	0AE9, 5E27
37	1	0BD1, 0000	0B82, 004F	0B82, 004F	D382, 384F
A4	1	1C10, 0500	1FB3, 015D	1FB3, 015D	9FB3, 815D
37	1	0BD1, 0000	08AF, 0322	0322, 08AF	E8AF, 2322
A4	1	1C10, 0500	2110, 0000	0000, 2110	E336, 3DDA
37	1	0BD1, 0000	0B86, 004B	0B86, 004B	9C86, 6F4B
A4	1	1C10, 0500	156D, 0BA3	156D, 0BA3	956D, 8BA3

Figure 1-1 Application Example



Related Topics

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

HDL Usage Through Component Instantiation - VHDL

```
library IEEE,DW01,DWARE;
use IEEE.std_logic_1164.all;
use DWARE.DW_foundation_comp.all;

-- Square & Accumulate performed by instances of
-- DW_squarep, DW01_csa & DW01_add

entity DW_squarep_inst is
  generic ( inst_width : NATURAL := 6;
            inst_verif_en : INTEGER := 3 );-- level 3 is the most aggressive
                                           -- verification mode for simulation
  port ( inst_a : in std_logic_vector(inst_width-1 downto 0);
         inst_b : in std_logic_vector(2*inst_width-1 downto 0);
         inst_tc : in std_logic;
         accum_inst : out std_logic_vector(2*inst_width-1 downto 0) );
end DW_squarep_inst;

architecture inst of DW_squarep_inst is
  signal part_prod1, part_prod2 : std_logic_vector(2*inst_width-1 downto 0);
  signal part_sum1, part_sum2 : std_logic_vector(2*inst_width-1 downto 0);
  signal tied_low, no_connect1, no_connect2 : std_logic;
begin
  -- Instance of DW_squarep to perform the partial
  -- multiply of inst_a by inst_a with partial product
  -- results at part_prod1 & part_prod2
  U1 : DW_squarep
    generic map ( width => inst_width,
                  verif_en => inst_verif_en )
    port map ( a => inst_a, tc => inst_tc,
               out0 => part_prod1, out1 => part_prod2 );


  -- Instance of DW01_csa used to add the partial products
  -- from inst_a times inst_a (part_prod1 & part_prod2) to
  -- the input inst_b in carry-save form yielding the two
  -- vectors, part_sum1 & part_sum2.
  U2 : DW01_csa
    generic map (width => 2*inst_width)
    port map ( a => part_prod1, b => part_prod2, c => inst_b,
               ci => tied_low, sum => part_sum1,
               carry => part_sum2, co => no_connect1 );

  -- Finally, an instqance of DW01_add is used to add the carry-save
  -- partial results together forming the final binary output
  U3 : DW01_add
    generic map (width => 2*inst_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 DW_squarep_inst_cfg_inst of DW_squarep_inst is
  for inst
  end for; -- inst
end DW_squarep_inst_cfg_inst;
-- pragma translate_on
```

HDL Usage Through Component Instantiation - Verilog

```
module DW_squarep_inst( inst_a, inst_b, inst_tc, accum_inst );

  parameter inst_width = 8;
  parameter inst_verif_en = 3; // level 3 is the most aggressive
                               // verification mode for simulation

  // Square and accumulate using DW_squarep
  input [inst_width-1 : 0] inst_a;
  input [2*inst_width-1 : 0] inst_b;
  input inst_tc;
  output [2*inst_width-1 : 0] accum_inst;

  wire [2*inst_width-1 : 0] part_prod1, part_prod2, part_sum1, part_sum2;

  // Instance of DW_squarep
  DW_squarep #(inst_width,inst_verif_en) U1 ( .a(inst_a), .tc(inst_tc),
                                .out0(part_prod1), .out1(part_prod2) );

  // Instance of DW01_csa
  DW01_csa #(2*inst_width) U2 ( .a(part_prod1), .b(part_prod2),
                                .c(inst_b), .ci(1'b0),
                                .sum(part_sum1), .carry(part_sum2) );

  // Instance of DW01_add
  DW01_add #(2*inst_width) U3 ( .A(part_sum1), .B(part_sum2),
                                .CI(1'b0), .SUM(accum_inst) );

endmodule
```

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

www.synopsys.com