SYNOPSYS®

DesignWare® Building Block IP

Application Notes

June 2015

Copyright Notice and Proprietary Information Notice

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

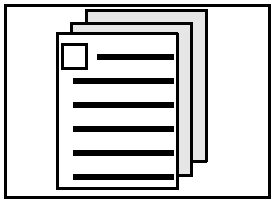
Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at http://www.synopsys.com/Company/Pages/Trademarks.aspx.

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

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

www.synopsys.com

Contents



Producing Syndrome Decoded Tables for DW_ecc and Related Components

AN 09-001

Description

The document describes a method that can be used to create a text file listing the corrected data and check bits decoded from the non-zero "syndrome" values based on the DW_ecc or any DesignWare component utilizing the DW_ecc. With the large configuration space that the DW_ecc supports, this information is provided in lieu of including comprehensive tables in the applicable DesignWare component datasheets.

A Verilog testbench called "syndrome_decode_tb" is listed below that contains an instance design of DW_ecc. In the source code, the parameters *data_width* and *chk_width* are modified to the desired values for the `datain` width and `chkin`/`chkout` port widths, respectively, of DW_ecc. A UNIX script called "run.csh" is provided and when executed will run the Verilog simulator VCS that generates an ASCII text file called "syndromeDecodeTable.txt" based on the user-specified *data_width* and *chk_width* values.

The "syndrome" is the `chkout` from the DW_ecc when the parameter *synd_sel* is 1.

Verilog Testbench Source Code

The following Verilog testbench contains an instance of DW_ecc, stimulus vectors that drive single-bit errors into the DW_ecc, and produces a text file called "syndromeDecodeTable.txt."

```
// Filename: syndrome_decode_tb.v
////////////////////////////////////////////////////////////////
//    Testbench that produces a decoded table of the errored bit
//    for each syndrome value for a particular data_width and
//    chk_width combination from the DW_ecc method.
//
//    Generates text file: syndromeDecodeTable.txt
////////////////////////////////////////////////////////////////
`timescale 1ns/10ps
module syndout_decode_tb ();

parameter data_width = 26;  // RANGE 1 to 8178
parameter chk_width  = 6;   // RANGE 5 to 14

wire [data_width-1:0]   dataout;
wire [chk_width-1:0]    syndout;
wire                    err_detect, err_multpl;

reg  [data_width-1:0]           datain, datain_saved;
reg  [chk_width-1:0]            chkin, chkin_saved;
reg  [data_width+chk_width-1:0] DataChkMask;

reg  [40*8-1:0]         Filename;

integer                 i, OPF;

// DW_ecc in "read_mode" with synd_sel=1
```

```
   DW_ecc #(data_width, chk_width, 1) U1 (
            .gen(1'b0), .correct_n(1'b1), .datain(datain),
            .chkin(chkin), .err_detect(err_detect),
            .err_multpl(err_multpl), .dataout(dataout),
            .chkout(syndout)
            );

  initial begin : PROC_mk_vectors
    Filename = "syndromeDecodeTable.txt";
    OPF      = $fopen(Filename);

    datain = 0;
    chkin = 0;
    DataChkMask = 0;
    // Store datain with accompanying 'correct' chkin
    datain_saved  = 0;
    chkin_saved   = {{chk_width-4{1'b0}}, 4'b1100};

    // Single-bit error generation: invert each datain and chkin bit
    //
    $fdisplay(OPF, "syndrome[%0d:0]     Errored Bit", chk_width-1);
    $fdisplay(OPF, "=============     ===========");
    for (i=data_width+chk_width-1; i>=0; i=i-1) begin
      if (i==data_width+chk_width-1)
        DataChkMask[data_width+chk_width-1] = 1;
      else
        DataChkMask = DataChkMask >> 1;
      // Invert a single bit of {chkin, datain};
      {chkin, datain} = {chkin_saved, datain_saved} ^ DataChkMask;
      #1;

      if (i < data_width)
        $fdisplay(OPF, "%0d'b%b          datain[%0d]", chk_width, syndout, i);
      else
        $fdisplay(OPF, "%0d'b%b         chkin[%0d]", chk_width, syndout, i-data_width);
    end

    $display("### Note: Creating text file \"%0s\" for data_width: %0d, chk_width: %0d
###", Filename, data_width, chk_width);

    #0 $finish;
  end // PROC_mk_vectors
endmodule
```

Run Script

The following is the contents from a UNIX run.csh script that executes VCS and simulates the testbench code shown above. This assumes that an environment variable $SYNOPSYS is set to the Design Compiler image being used and a Verilog file named syndrome_decode_tb.v which contains the contents of the Verilog source code from above.

NOTE: VCS is shown only as an example and there are no restrictions on using any other Verilog simulators to run the testbench code from above.

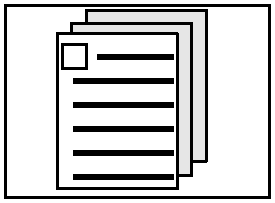
The following is the run.csh script contents:

```
#!/bin/csh -f
vcs -R syndrome_decode_tb.v  -y $SYNOPSYS/dw/sim_ver +libext+.v+
```

Example of "syndromeDecodeTable.txt" contents

The following shows the contents produced in the "syndromeDecodeTable.txt" file for "data_width=16" and "chk_width=6" ("syndrome" is from the "chkout" of the DW_ecc):

```
syndrome[5:0]      Errored Bit
=============      ===========
6'b100000          chkin[5]
6'b010000          chkin[4]
6'b001000          chkin[3]
6'b000100          chkin[2]
6'b000010          chkin[1]
6'b000001          chkin[0]
6'b101111          datain[25]
6'b110010          datain[24]
6'b110111          datain[23]
6'b111000          datain[22]
6'b111011          datain[21]
6'b111101          datain[20]
6'b111110          datain[19]
6'b000111          datain[18]
6'b001101          datain[17]
6'b011111          datain[16]
6'b110100          datain[15]
6'b110001          datain[14]
6'b101100          datain[13]
6'b101010          datain[12]
6'b101001          datain[11]
6'b100110          datain[10]
6'b100101          datain[9]
6'b100011          datain[8]
6'b011100          datain[7]
6'b011010          datain[6]
6'b011001          datain[5]
6'b010110          datain[4]
6'b010101          datain[3]
6'b010011          datain[2]
6'b001011          datain[1]
6'b001110          datain[0]
```



DesignWare Datapath Generator Strategies

AN 07-001

Datapath Generators

The `pparch` implementation of DW arithmetic components as well as complex datapath blocks optimized with Design Compiler Ultra are synthesized using the DW datapath generators. Flexible circuit structures are generated to optimize QoR for a given context. See also the Datapath Generator Overview in the *DesignWare IP Family Reference Guide*.

Basic Architecture

The following text describes the basic circuit architecture that is generated for arithmetic operations such as adders and multipliers.

Partial-Product Generation

Partial products are generated in a very straightforward way using simple AND-gates or different flavors of Booth encoding. Optimized partial products are generated for special cases like constant multiplication or squaring.

Carry-Save Addition

A delay-driven algorithm is used to construct flexible adder trees with minimal area and delay. Compressor cells (such as half-adder, full-adder, 4-to-2 compressors) are selected to optimize QoR.

Carry-Propagate Addition

A delay- and constraint-driven algorithm is used to construct flexible parallel-prefix adders that cover the entire range between fastest Sklansky-type prefix adders to smallest ripple adders depending on the constraints. These adders are optimized for input arrival times and output required times. Full-adder cells are used wherever beneficial. The same optimization is used for any operation that involves carry-propagation (such as addition, subtraction, increment, magnitude comparison, etc.).

Datapath Smart-Generation Strategies

In addition to the basic architecture the datapath generators implement a variety of micro-architectures that allow to fine-tune the circuit for QoR. This section describes the smart-generation strategies that exist to control the micro-architecture of arithmetic datapath components. These strategies apply to all components with `pparch` implementation as well as complex datapath blocks extracted and synthesized with Design Compiler Ultra.

The smart-generation strategies can be set by the Design Compiler command `set_dp_smartgen_options`.

Syntax:

```
int set_dp_smartgen_options
    [-all_options auto | true | false | default]
    [-booth_encoding auto | true | false]
    ...
```

Please refer to the man page for complete syntax information.

Each strategy option can be set to three different values:

- `auto` : Smart-generation automatically decides whether or not to use the strategy for best possible QoR.
- `false` : The strategy is never selected.
- `true` : The strategy is always selected.

Booth Encoding

Smart-generation option: `-booth_encoding`

Affected operations: multiplication

Description

If this strategy is selected, modified radix-4 Booth encoding is used on one multiplier operand to reduce the number of partial products to be added. Area of medium to large sized multipliers is reduced due to smaller adder tree. Delay is only marginally affected but can also improve slightly.

Otherwise regular partial-product generation is used (AND-gate).

Radix-8 Booth Encoding

Smart-generation option: `-booth_radix8`

Affected operations: multiplication

Description

If this strategy is selected, radix-8 Booth encoding is used to further reduce the number of partial products. This helps reduce area in large multipliers even further. Since the generation of partial products is more complex (requires a carry-propagate adder), delay is increased.

Otherwise radix-4 Booth encoding is used.

Booth Encoder Cells

Smart-generation option: `-booth_cell`

Affected operations: multiplication

Description

If this strategy is selected, special Booth encoder/selector cells are used if available in the technology library. Depending on the performance of these library cells, area and delay of a multiplier can be improved.

Otherwise Booth encoding/selection is implemented using simple gates.

If no special cells exist in the library and the strategy is selected, an alternative Booth encoding/selection is used, which bases on multiplexers. This can give better QoR for libraries with fast and/or small multiplexer cells.

4-to-2 Compressor Cells

Smart-generation option: `-4to2_compressor_cell`

Affected operations: multiplication, multi-operand addition; carry-save addition

Description

If this strategy is selected, 4-to-2 compressor cells are used to add multiple operands in a carry-save adder tree, if applicable and available in the library. QoR can be improved if the 4-to-2 compressor cells are implemented efficiently. Layout regularity can also be improved due to the larger cell size and the binary compression rate of 2:1 (as opposed to a compression rate of 3:2 for full-adder cells). Half-adders and full-adders are still used where there are not enough addend bits for a 4-to-2 compressor.

Otherwise only half-adders and full-adders are used.

Carry-Select Adder Cells

Smart-generation option: `-carry_select_adder_cell`

Affected operations: binary addition; carry-propagate addition

Description

If this strategy is selected, a flexible carry-select adder is implemented using special carry-select-adder cells, if available in the library. Area can be reduced for loose timing constraints, but tight timing constraints might not be met.

Otherwise a flexible parallel-prefix adder architecture is used.

Conditional-Sum Adder

Smart-generation option: `-cond_sum_adder`

Affected operations: binary addition; carry-propagate addition

Description

If this strategy is selected, a flexible conditional-sum adder architecture is used. Delay can be improved because the critical path is shorter by one complex gate. However, area is much larger and larger loads can also offset the delay advantage.

This strategy can be used together with the `-mux_based` option to control whether multiplexers should be used in the carry generation.

Otherwise a flexible parallel-prefix adder architecture is implemented.

Bounded-Fanout Adder

Smart-generation option: `-bounded_fanout_adder`

Affected operations: binary addition; carry-propagate addition

Description

If this strategy is selected, a parallel-prefix adder with bounded-fanout characteristic is implemented (Kogge-Stone prefix adder). Delay can be reduced because of smaller loads on the critical path, but area is always much larger.

Otherwise a flexible unbounded-fanout parallel-prefix adder is implemented (Sklansky/Ladner-Fisher prefix adder, ripple adder or anything in between).

Multiplexer-Based Implementation

Smart-generation option: `-mux_based`

Affected operations: binary addition; carry-propagate addition

Description

If this strategy is selected, carry-propagate logic in adders is implemented using multiplexers. QoR can be improved for technology libraries with efficient multiplexer cells.

Otherwise, AOI/OAI (and-or-invert/or-and-invert) gates are used.

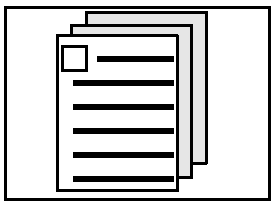
Inverting Full-Adder Cells

Smart-generation option: `-inv_adder_cell`

Affected operations: binary addition; carry-propagate addition

Description

If this strategy is selected, inverting full-adder cells (with either the carry-in or the carry-out inverted) are used in ripple-carry adders, if available in the library. Area and delay can be reduced if these cells are implemented efficiently.



Design-for-Testability with 1149.1 JTAG Components

AN 99-008

The DesignWare Building Block IP IEEE 1149.1 (JTAG) offers a flexible set of technology-independent IP that enable you to easily add 1149.1 compliant test capabilities to your ASIC designs.

The DesignWare 1149.1 solution consists of a highly parameterized Test Access Port (DW_tap and DW_tap_uc) and 10 boundary scan cells (DW_bc_1,2,3,4,5,78,9,10).

Note

Starting Y-2006.06-SP1 DW_tap and DW_tap_uc are scan testable. An input signal "test" has been added to these components. For scannable designs, the test pin is held active (HIGH) during testing. For normal operation, it is held inactive (LOW).

The DW_tap module provides you with the flexibility to implement test solutions that are custom-fit for your ASICs. If you require only a minimal implementation of JTAG to satisfy your printed wiring assembly manufacturing needs, you can use the DesignWare Building Block IP directly without additional design work. If you are a high-end user, you can easily integrate your proprietary test strategies using the flexible set of parameters and ports provided in the DesignWare Building Block IP. The parameterization allows for a wide variety of implementations, and only as much test overhead logic as is required for the application. Table 4-1 lists the parameters associated with the DW_tap.

Table 4-1 DW_tap Parameter Descriptions

Parameter	Values	Description
width	2 to 32 Default: None	Width of instruction register
id	0 or 1 Default: 0	Determines whether the device identification register is present 0 = not present, 1 = present
version	0 to 15 Default: 0	4-bit version number
part	0 to 65535 Default: 0	16-bit part number
man_num	0 to 2047, man_num ≠ 127 Default: 0	11-bit JEDEC manufacturer identity code
sync_mode	0 or 1 Default: 0	Determines whether the bypass, device identification, and instruction registers are synchronous with respect to tck 0 = asynchronous, 1 = synchronous

To further extend the flexibility of the DesignWare TAP controller, ports are made accessible for users. The following capabilities are possible through special ports provided by the component:

- You may optionally specify your own sentinel patterns, allowing them to shift out internal chip status during 1149.1 instruction loads

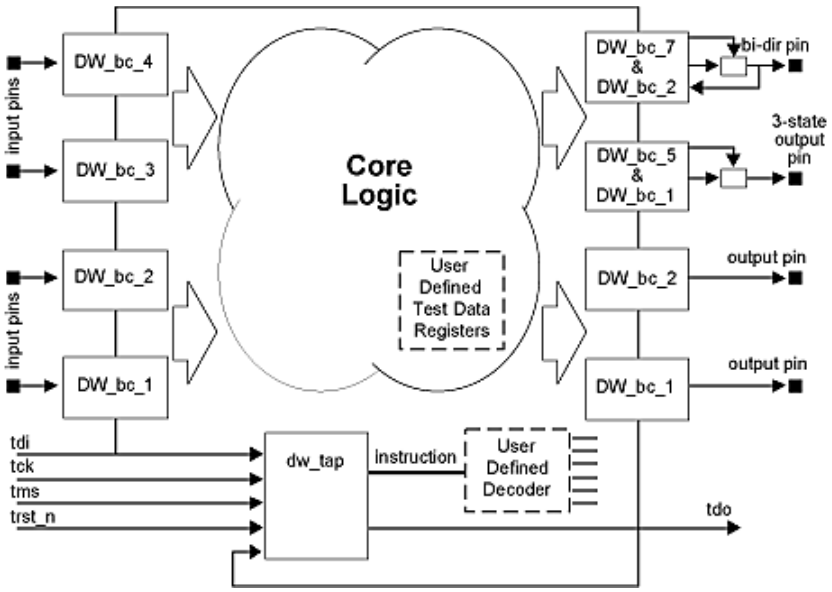
- Direct access to the one-hot encoded TAP state machine allows you to construct add-on circuits
- Direct access to the Instruction Register allows you to design your own decode logic, providing you with the flexibility to choose an instruction encoding convention suited for your applications, such as binary, one-hot, or gray. The component automatically provides decode logic for Bypass, IDCode, Extest and Sample/Preload modes, as the encoding for these instructions is specified by the 1149.1 standard.

In addition to the TAP controllers, ten boundary scan cell components are available for you to place in front of the ASIC's I/O pads. These components offer the standard capabilities described in the 1149.1 specification, and are designed to interface directly with the DW_tap component. The boundary scan cells may be synchronous or asynchronous with respect to `tck`, depending on the connections made to the clock signals and `clock_enable` signals of the boundary scan cell.

The boundary scan cells are compatible with the optional Intest, RunBIST, Clamp, and HighZ instructions.

Some ASIC technology libraries provide 1149.1 compliant boundary scan cells that minimize the performance impact associated with boundary scan cells. These cells can be easily incorporated with the DW_tap controller to provide an optimal boundary scan solution, allowing high performance I/Os and flexible test capabilities.

Figure 4-1 Block Diagram



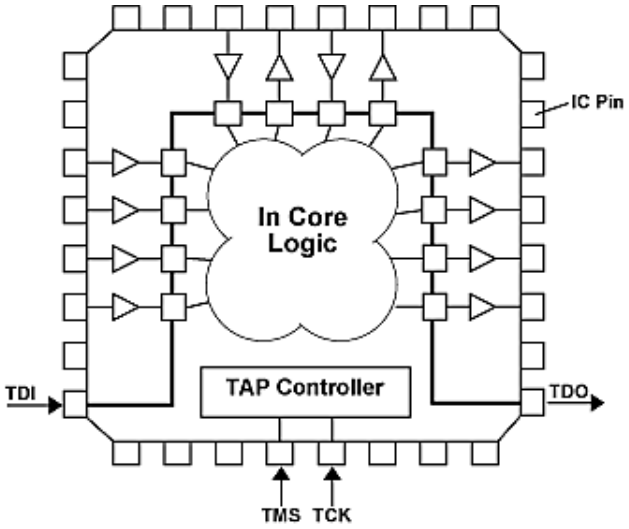
Applications for DW1149.1 IP

Boundary scan is used for a wide variety of test related functions, and can be used at various levels of abstraction in the testing of digital systems.

Chip-Level Testing

For chip-level testing, vectors can be applied serially through the boundary scan cells to perform static testing of internal chip logic via the TAP. Stimulus vectors are applied through the TDI (Test Data Input) port, and results are scanned out through the TDO (Test Data Output) port. Also, the 1149.1 standard provides for direct support for on-chip Built-In-Self-Test (BIST).

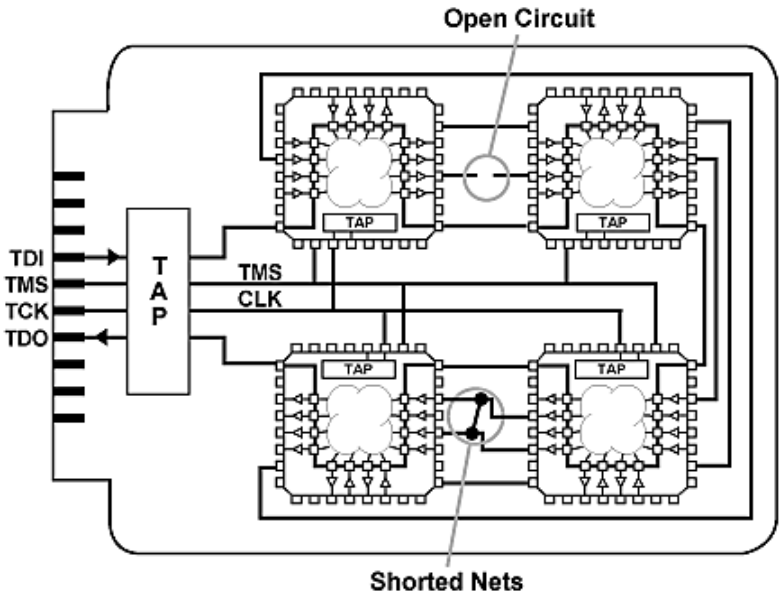
Figure 4-2 Chip Test



Board-Level Testing

Board level diagnostics can utilize the scan structure of multiple 1149.1 ASICs to test the wiring and device I/O drivers of nets connecting the chips on the board. This is the most common application of boundary scan testing.

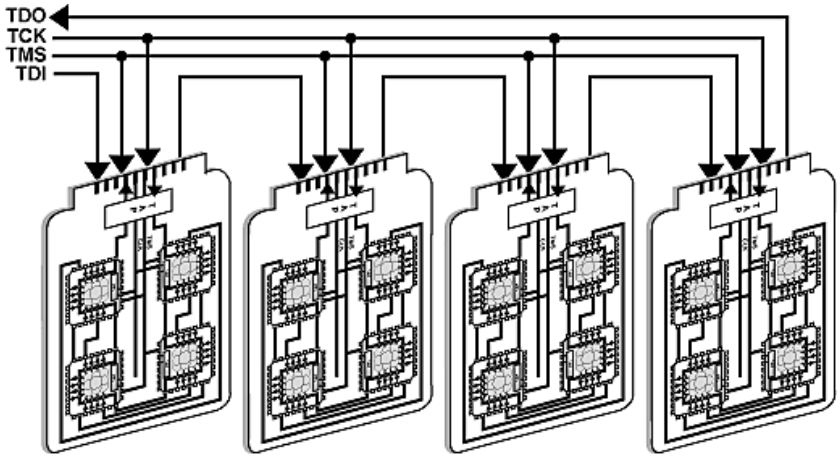
Figure 4-3 Board-Level Test



System-Level Testing

Moving to the next level of abstraction, system-level testing can also be accomplished in a system that incorporates multiple boards containing 1149.1 ASICs.

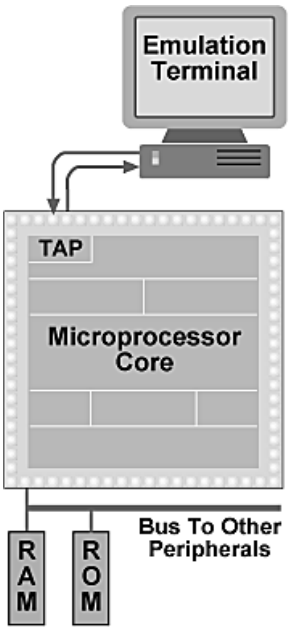
Figure 4-4 System-Level Test



Microprocessor Emulation

Because an IP with boundary scan has the ability to control its I/O pins directly from the TAP, it is possible to provide microprocessor emulation by direct manipulation of its pins. A 1149.1 bus controller located in a PC or other test system controls the I/O pins of the microprocessor via its Test Access Port. This approach provides a high degree of flexibility and sophistication, since emulation is performed entirely in the test system software. This is advantageous in manufacturing test environments, since debugging can take place without desoldering the on-board microprocessor.

Figure 4-5 Microprocessor Emulation



Connect Different Bit-Width Subsystems With Asymmetric I/O FIFOs

AN 99-007

Connecting data with different bit-widths is like trying to connect a small garden hose to a fire hydrant. It doesn't work without a proper adapter (see Figure 5-1).

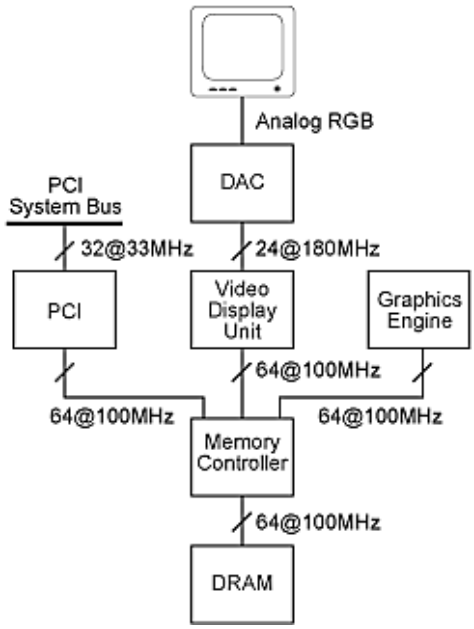
This application note introduces a new FIFO family: the Asymmetric Input/Output FIFOs and FIFO controllers. These new FIFOs and controllers connect data with different bit-widths between subsystems in the same clock domain or in different clock domains.

Figure 5-1 Trying to Connect Data with Different Bit Widths

You often must contend with subsystems that use different bit-widths to pass data to each other. Another common dilemma occurs when external data buses have different bit-widths than the internal subsystems to which their data connects.

To illustrate these types of problems, Figure 5-2 portrays a graphics application where Asymmetric FIFOs are needed in a design. The problems occur when the 24 bit/180 MHz Video Display Unit is reading from the 64 bit/100 MHz Memory Controller, while the 32 bit/33 MHz PCISystem Bus is writing to the same Memory Controller.

Figure 5-2 Graphic Application



Another variable data-width example can be found in low power applications. The clock buffering in Deep Sub-Micron (DSM) designs is a major power consumer. A common strategy to achieve lower power consumption is to use a special clock distribution technique. This technique uses lower-speed but wider synchronous buses to connect the modules. Also, a clock is transmitted as one of the bus signals. Each module has its own local clock that synchronizes to the bus clock, and runs at a higher frequency for faster performance. The module may use a narrower data bit-width than the buses. A design using multiple clocks and multiple data-widths can reduce power consumption while achieving the required data throughput.

You may face combinations of the following scenarios when connecting data with different widths:

- Data width conversion.
  The data must be converted from bit-width Wa to bit-width Wb, where Wa ≠ Wb. In most cases, subsystems talk with each other in both directions; thus, conversions from Wa to Wb and Wb to Wa are required. These two conversions should be compatible with each other.
- Assume that Wa > Wb;
  the converter from Wa to Wb is called the “serializer”, while the converter from Wb to Wa is called the “parallelizer.” Due to their different functionalities, the structure of the serializer is remarkably unlike that of the parallelizer.
- Burst control.
  Subsystems using different bit-widths may have a relatively continuous data flow, but different rates and/or burst characteristics. Therefore, they cannot directly connect. FIFO buffering is often required when connecting two subsystems.
- Latency constraint.
  Data transformation latency is often constrained, especially in high-performance designs. For example, in an 8-to 24-bit-width conversion, after the transmitter sends out three 8-bit bytes, the receiver should be able to receive 24-bit-width data as soon as possible. To reduce delay, both the data path and the synchronized flags that indicate availability of data and storage must have minimum latency. A zero clock-cycle latency is optimal.

- Data bit alignment.
  It is important to ensure that data bits are correctly aligned at the end of a block of data, or at the beginning of a block of new data.
- Separate clock domains.
  Two subsystems commonly work in two separate clock domains. Usually, the processing speeds of two subsystems are the same or similar (for example, they satisfy the formula Fa × Wa ≠ Fb × Wb, where Fa and Fb are frequencies of subsystems A and B). This type of design will have metastability problems. Approaches for dealing with metastability problems can be found in Application Note "What Is Metastability, And How To Live With It" on page 47.

A traditional solution for connecting two subsystems with different bit-widths is to design two converters: a serializer and a parallelizer. The serializer usually consists of a FIFO controller, RAM, MUX, and control circuit. The parallelizer usually consists of a FIFO controller, RAM, register, and control circuit.

The following challenges are involved with this solution:

- The converters and FIFOs are relatively complex. Bi-directional data flow requires both a serializer and a parallelizer. This design task is time-consuming and error-prone.
- Designing synchronous flags with zero clock-cycle latency for FIFO full and empty conditions is difficult without accessing internal signals of the FIFO controllers in the DesignWare Building Block IP.
- This solution is neither scalable nor portable. For example, if the RAM bit-width, conversion ratio, or byte order change, the converters must be redesigned.
- Due to the requirement of working with two clock domains, designing such converters with FIFOs becomes more difficult and complicated.

DesignWare Building Block IP offers the asymmetric FIFO family as an easy and high-quality solution to connecting subsystems with different bit-widths. The DesignWare Building Block IP asymmetric FIFOs are pre-verified, high-quality, technology-independent, and fully testable "off-the-shelf" parts with the following characteristics:

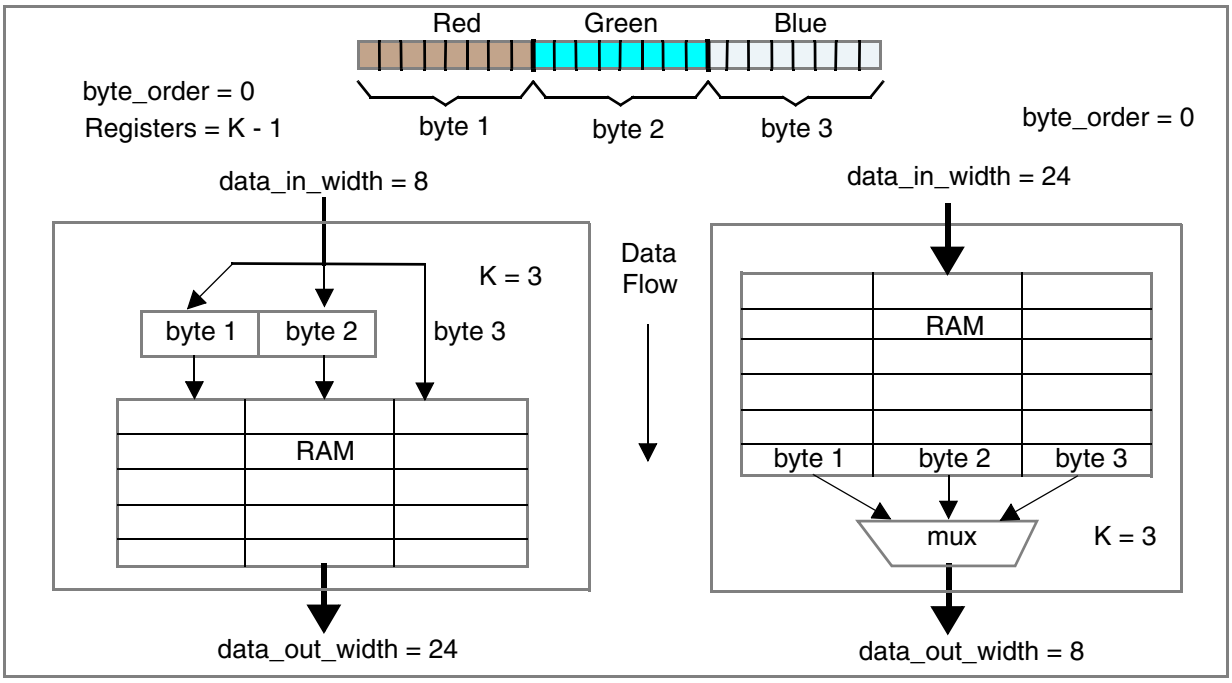
- Parameterizable input/output bit-widths and byte order
- Comprehensive synchronous flags with zero latency
- At least two architectures for better "timing vs. area" results
- Different versions for single- and dual-clock domain applications

The New Family: Asymmetric FIFOs and FIFO Controllers

For ease-of-use, the asymmetric FIFOs and FIFO controllers are packaged in a "box" style. You merely plug into the "box", which contains the data bit-width converters and FIFOs (or FIFO controllers) for the parallelizer and serializer. You implement the asymmetric FIFOs and FIFO controllers through the unified wrap without worrying about the complex circuits hiding inside the "box."

Figure 5-3 shows the high-level structure of an asymmetric FIFO. Although the parallelizer is quite different from the serializer, you do not need to worry about the differences in the implementation details because the Building Block IP asymmetric FIFO properly selects the parallelizer or serializer according to the ratio of the input data bit-width and the output data bit-width. This ratio is shown as K in Figure 5-3.

Figure 5-3 High-Level Structure of an Asymmetric FIFO

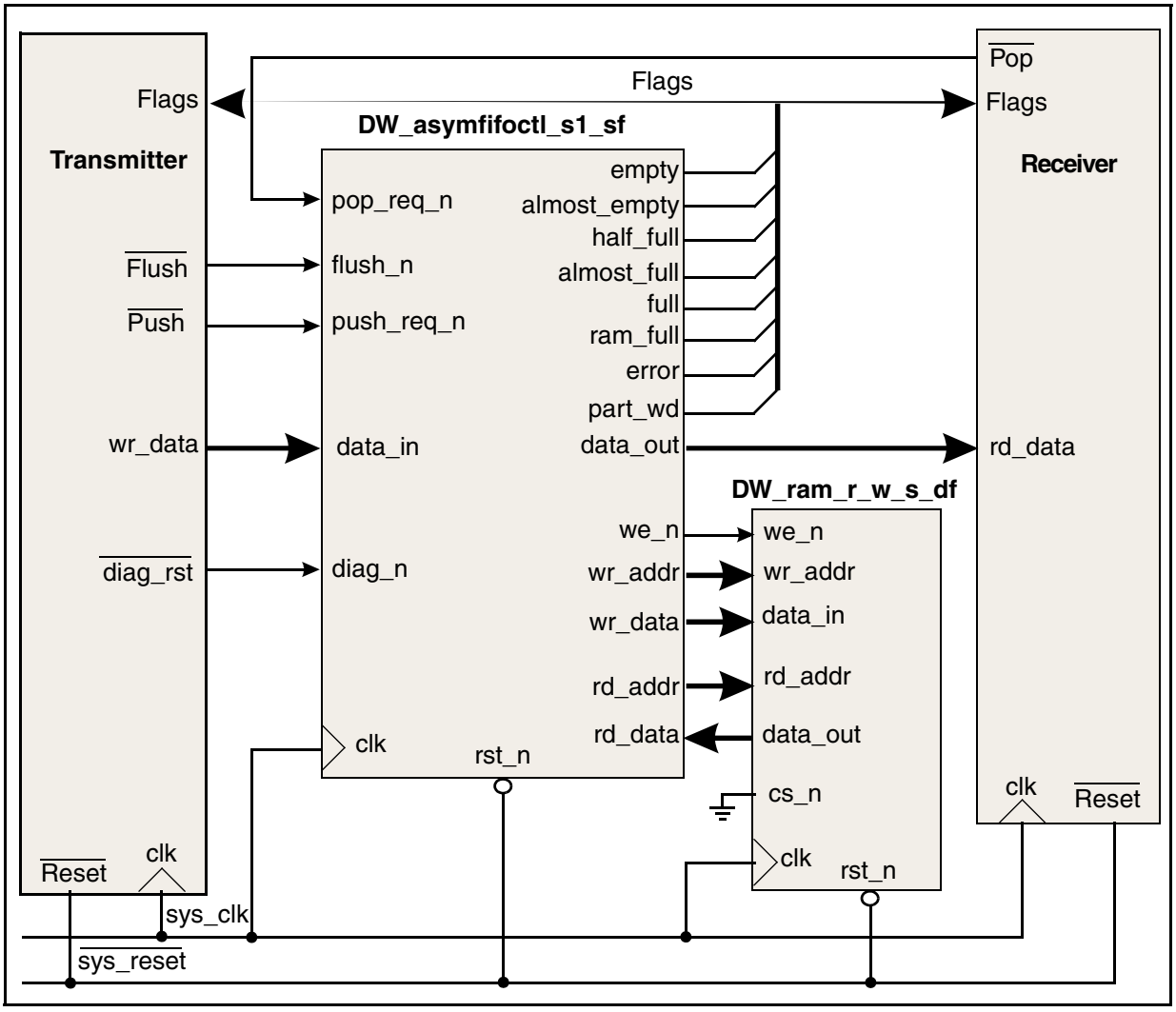


In the special case when the input data bit-width is the same as the output data bit-width (i.e. K=1), the DesignWare Building Block IP asymmetric FIFO selects a regular FIFO. Again, this is all handled by the asymmetric FIFO, leaving you free to develop the other portions of the design.

The order of subwords in a full word is important when creating parallel and serial converters. For the example on the left in Figure 5-3, the first subword or byte is placed in the Most Significant Bits (MSBs) within a 24-bit data. DesignWare Building Block IP asymmetric FIFOs and FIFO controllers are flexible in subword order. The subwords can be arranged either from MSBs to Least Significant Bits (LSBs), or from LSBs to MSBs by setting the `byte_order` parameter.

Figure 5-4 shows the connection of an asymmetric FIFO controller with a transmitter and receiver. The Building Block IP asymmetric FIFO controller has comprehensive flags: full, almost full, half full, almost empty, and partial word. To minimize area, Design Compiler automatically removes the circuits for all unnecessary flags.

Figure 5-4 Connection of an Asymmetric FIFO Controller with Transmitter and Receiver



Sometimes data alignment is required for a parallelizer. For example, at the end of a data block when there is no further data to send, the last word in the block is a "partial" word. This is often the case in transmission interruption. Without special handling, subwords in the next data block are placed in the wrong slots. In this case, the last partial word should be "flushed" out so that the subwords in the next new data block properly align with the full word. The Building Block IP asymmetric FIFOs and FIFO controllers offer a flag, `part_wd`, for detecting partial words, and a control pin, `flush_n`, for alignment.

Table 5-1 lists the available versions of Building Block IP asymmetric FIFOs and FIFO controllers.

Table 5-1 Building Block IP Asymmetric FIFO Family

IP Name	Description	Availability
DW_asymfifo_s1_sf	Asymmetric I/O synchronous single-clock FIFO with static flags	CD 1998.08
DW_asymfifo_s1_df	Asymmetric I/O synchronous single-clock FIFO with dynamic flags	CD 1998.08

Table 5-1 Building Block IP Asymmetric FIFO Family

IP Name	Description	Availability
DW_asymfifo_s2_sf	Asymmetric I/O synchronous dual-clock FIFO with static flags	DWF February EST, 1999
DW_asymfifoctl_s1_sf	Asymmetric I/O synchronous single-clock FIFO controller with static flags	CD 1998.08
DW_asymfifoctl_s1_df	Asymmetric I/O synchronous single-clock FIFO controller with dynamic flags	CD 1998.08
DW_asymfifoctl_s2_sf	Asymmetric I/O synchronous dual-clock FIFO controller with static flags	DWF February EST, 1999
DW_asymdata_inbuf	Asymmeteric Data Input Buffer	2007.03
DW_asymdata_outbuf	Asymmeteric Data Output Buffer	2007.03

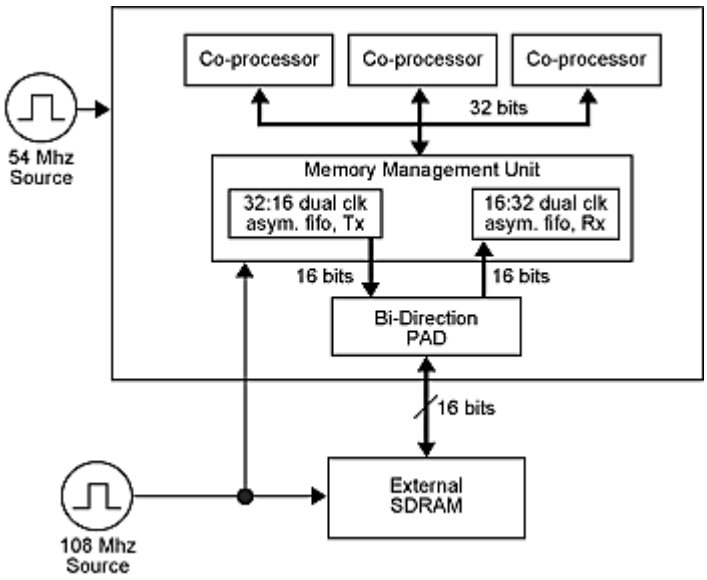
Building Block IP asymmetric FIFOs use the D flip-flop-based memory arrays for high testability. They are designed for relatively small memory configurations. For large FIFOs or other memories, use the asymmetric FIFO controllers in conjunction with compiled or full-custom RAM arrays.

The current version of DesignWare Building Block IP asymmetric FIFOs and FIFO controllers has limitations on the ratio of the input and output data bit-width. The ratio K = (input bit-width) / (output bit-width) or K = (output bit-width) / (input bit-width) must be an integer. If you need an arbitrary ratio asymmetric FIFO or FIFO controller, please e-mail designware@synopsys.com, or call 1-877-4-BEST-IP.

Application Example

Let's look at a production example using the DesignWare Building Block IP dual-clock asymmetric FIFOs (as shown in Figure 5-5).

Figure 5-5 High-Level Diagram of a Multimedia Application



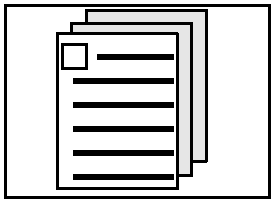
The following considerations apply to this example:

- The design context is a multimedia application
- The frequency of the processor is 54 MHz
- A 32-bit bus is used in the processor subsystem
- The processor subsystem is connected to a vendor's SDRAM, which is running at 108 MHz and is 16-bits wide
- The processor uses a wider bus and slower frequency, while the memory chips use a narrower bus and faster frequency
- The data throughputs of the processor and SDRAM are the same
- Although one clock appears to be twice the frequency of the other, they come from separate sources, which means that the phase relationship between them is not only unknown, but continually changing

In this example, dual-clock asymmetric I/O FIFOs are the best fit for connecting the processor subsystem with the memory chips. In MMUs (Memory Management Units), a 32:16 dual-clock asymmetric FIFO (DW_asymfifo_s2_sf) is used for converting the 32-bit data flow from the processor subsystem to the 16-bit data feeding into the memory unit. Another 16:32 dual-clock asymmetric FIFO is used for converting 16-bit data from the memory unit to the 32-bit data feeding into the processor subsystem. Using dual-clock asymmetric FIFOs in this application eliminates metastability problems, and ensures that data flows smoothly, thus eliminating the possibility of choke or under-feeding.

Summary

The DesignWare Building Block IP asymmetric FIFOs and FIFO controller family allows you to connect subsystems with different data bit-widths, even in different clock domains. You can use asymmetric FIFOs and FIFO controllers through the unified wrap without worrying about the complex circuits hiding inside the "box."



Error Management Using DesignWare ECC

AN 99-006

As systems get faster and denser, they are required to process, store, and move more data than ever before. This has led to an increasing need for such systems to manage errors. One of the common techniques for managing errors in systems is Error-Detect-&-Correct (EDC). The EDC method is popular in storage media and high density memory applications. The DesignWare Building Block IP offers technology-independent, pre-verified, synthesizable module (Dw_ecc) that address this method of error management.

Error Correction Code (ECC)

As chip designs grow larger, they tend to contain more memory. Fed by the "need for speed," IC designs requiring internal caches are becoming much more common. In addition to on-chip primary caches, many systems use external secondary caches. Not only cache, but main memory requirements keep growing as well. With all of this on-chip and off-chip memory, coupled with the goal of increasing reliability, many designers are now turning to on-chip error management schemes. The most popular error management scheme for RAMs is an EDC method known as the Single Error Correcting, Double Error Detecting (SECDED) Modified Hamming Error Correction Code (ECC). Hamming codes derive their redundant information (check bits) from Exclusive-OR sums. Several bits of redundancy are generated, and each of these check bits are derived from a different combination of roughly half the data bits. The combinations of data bits that contribute to each check bit are chosen such that a single erroneous bit can be determined by the combination of check bits that individually detected the error. An additional check bit is used to rule out the possibility of being misled into believing a single bit error is present when, in fact, a double bit error exists. The EDC scheme can correct corrupted check bits as easily as it corrects corrupted data bits. Table 6-1 lists the minimum number of check bits required to use SECDED ECC.

Table 6-1 Minimum Number of Check Bits Required

Width	Check Bits
3 - 4	4
5 - 11	5
12 - 26	6
27 - 57	7
58 - 120	8
121 - 247	9
248 - 502	10

In Example 6-1and Example 6-2, three data bits are protected by four check bits. Although this example may seem rather simple, designing an EDC system can easily get rather confusing and unwieldy as word widths reach that of modern systems. In addition, there is a subtle method that can improve the likelihood of catching a certain percentage of triple bit errors, thus increasing system reliability even further. You won't need to reinvent the wheel if you use the EDC scheme

implemented in the DW_ecc DesignWare Building Block IP module from Synopsys. The DW_ecc module is a parameterizable drop-in SECDED Modified Hamming EDC component that can be synthesized for memory word widths from 8 bits up to 502 bits. The DW_ecc module's pass-through design fits easily into any system architecture. Most systems contain one instance of DW_ecc for check bit generation in the memory write path and another instance of DW_ecc in the memory read path for actual error detection and correction. For designers who want to go so far as to perform error scrubbing, the DW_ecc module can correct check bits as well as data bits. Some types of systems may take advantage of the ability (through parameterization) of the DW_ecc module to emit the error syndrome associated with an error. Designers of such systems may want to log the existence and location of errors that occur during operation. This information can be helpful in monitoring the "health" of the system, as well as aiding in locating faulty hardware.

Example 6-1 Encoding Equations

```
Check bit 0: C0 = D0 XOR D1
Check bit 1: C1 = D0 XOR D2
Check bit 2: C2 = D0 XOR D1 XOR D2
Check bit 3: C3 = D1 XOR D2
Encoding 110 results in a combined (data + check) of 0011.110
```

Example 6-2 Decoding Equations

```
Syndrome bit 0: S0 = D0 XOR D1 XOR C0
Syndrome bit 1: S1 = D0 XOR D2 XOR C1
Syndrome bit 2: S2 = D0 XOR D1 XOR D2 XOR C2
Syndrome bit 3: S3 = D1 XOR D2 XOR C3
Decoding 0011.110 results in an error syndrome of 0000 (i.e., no error)
Decoding 0011.100 results in an error syndrome of 1101, which uniquely identifies data
bit 1.
```

Figure 6-1 Memory Write Diagram

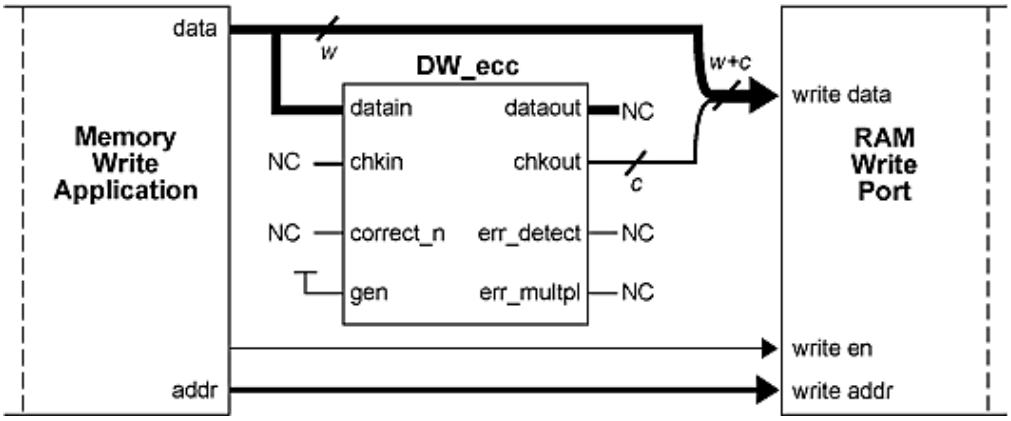
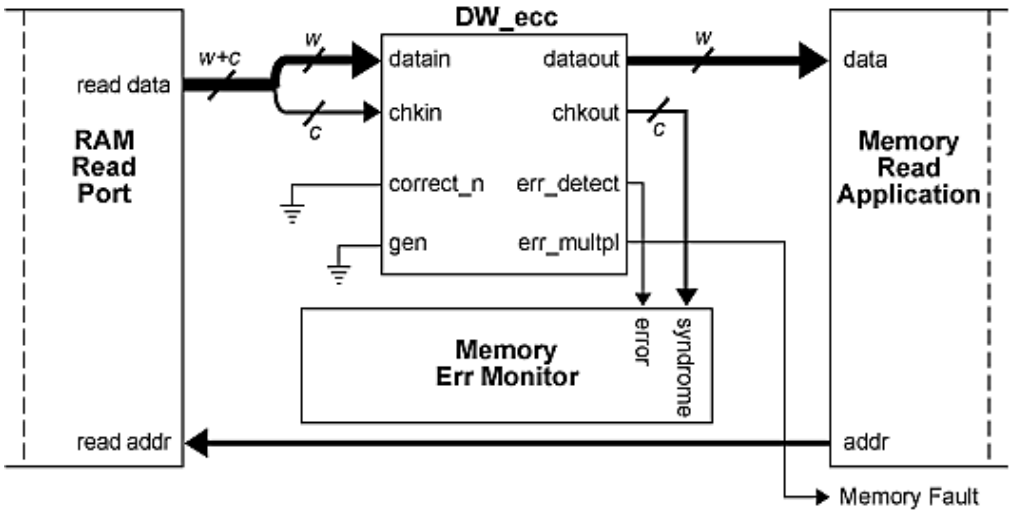
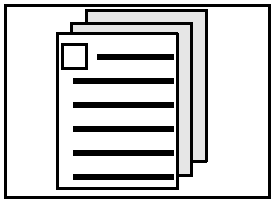


Figure 6-2 Memory Read Diagram with Syndrome Logging



Summary

The DesignWare Building Block IP allows you to build high-reliability systems without the time and tedium of developing and verifying your own modules for error management. Instead of spending time on developing ECCs and Hamming codes, you now can simply use pre-engineered synthetic modules, and spend more time on the higher-level issues relating to your specific designs.



IEEE Numeric Standard Package

AN 99-005

Introduction

Prior to the 1998.08 Synthesis CD release, HDL Compiler supported the Synopsys-proprietary arithmetic package. In the 1998.08 Synthesis CD release, HDL Compiler supports the IEEE numeric standard package (as well as the arithmetic package).

To synchronize with the Synopsys product line, the November 1998 and all future EST releases of DesignWare Building Block IP Library support the IEEE numeric standard package. The arithmetic package will continue to be supported, and the IEEE numeric standard and Synopsys arithmetic packages will co-exist.

Both packages define SIGNED and UNSIGNED data types. Due to a VHDL language limitation, any data type cannot be defined twice. Therefore, these two packages cannot be used at the same time. You should select the proper DesignWare Building Block IP packages to achieve the desired results.

DesignWare Building Block IP Packages: Old and New

To improve the usability of DesignWare Building Block IP, the packages have been reorganized. Table 7-1 lists the old DesignWare Building Block IP packages as well as the new and more user-friendly IP packages supporting both Synopsys arithmetic and IEEE numeric standard packages.

The old packages are still supported for arithmetic package users for design reuse purposes. However, it is recommended that new users use the new packages.

DesignWare Building Block IP deliverables are package-independent, and can work with either package. All ports of any DesignWare Building Block IP, including synthetic and simulation models, have std_logic or std_logic_vector types as the interface to the outside world, although some simulation models may internally use SIGNED and UNSIGNED data types defined in either the numeric or arithmetic package. None of the ports are SIGNED or UNSIGNED type. Inference functions and operators may have SIGNED or UNSIGNED type parameters, or returned values.

If you do not use the Synopsys VCS MX simulator, you may need to analyze simulation models within your own simulation environment. The comprehensive package source codes offered under $SYNOPSYS/packages/dware/src enable you to easily run the analysis with either the arithmetic or numeric package.

The DesignWare R&D team has verified that every component, whether in instantiation, function, or operator formats, behaves identically for upper-level design using either the Synopsys arithmetic package or the IEEE numeric package.

For quality control purposes, DesignWare Building Block IP is regression tested against both packages before each release.

Package Usage

The DW_Foundation (or DW_Foundation_arith for arithmetic users) package should be used if functions such as `minimum()`, `maximum()`, `bit_width()`, DW function inference, or operator inference are used.

The following examples show different DesignWare Building Block IP package usage:

Example 7-1 Function Inference, IEEE Numeric Package

```
library IEEE, DWARE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use DWARE.DW_Foundation.all;
entity DW02_cos_func is
  generic(wordlength1 : integer:= 8;wordlength2: integer := 8);
  port( angle : in SIGNED(wordlength1-1 downto 0);
       cos_out: out SIGNED(wordlength2-1 downto 0));
end DW02_cos_func;

architecture func of DW02_cos_func is
begin
  -- infer DW02_cos
  cos_out <= cos(angle,wordlength2);
end func;
```

Example 7-2 Function Inference, Arithmetic Package

```
library IEEE, DWARE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use DWARE.DW_Foundation_arith.all;
entity DW02_cos_func is
  generic(wordlength1 : integer := 8;
          wordlength2 : integer := 8);
  port( angle : in  SIGNED(wordlength1-1 downto 0);
       cos_out: out SIGNED(wordlength2-1 downto 0));
end DW02_cos_func;

architecture func of DW02_cos_func is
begin
  -- infer DW02_cos
  cos_out <= cos(angle,wordlength2);
end func;
```

Example 7-3 Component Instantiation

```
library IEEE,DWARE;
use IEEE.std_logic_1164.all;
use DWARE.DW_Foundation_comp;
entity DW02_cos_inst is
  generic (inst_A_width   : INTEGER := 8;
           inst_cos_width : INTEGER := 8);
  port (inst_A   : in  std_logic_vector(inst_A_width-1   downto 0);
        COS_inst : out std_logic_vector(inst_cos_width-1 downto 0));
end DW02_cos_inst;

architecture inst of DW02_cos_inst is
begin
  -- Instance of DW02_cos
  U1 : DW02_cos
    generic map ( A_width => inst_A_width, cos_width => inst_cos_width )
    port map    ( A => inst_A, COS => COS_inst );
end inst;
```

DesignWare Building Block IP Releases and IEEE Numeric Standard Support

The 1998.08 CD release of DesignWare Building Block IP does not support the IEEE numeric standard. Support for the IEEE numeric package 87 standard begins with the November 1998 EST Building Block IP release.

Possible Impacts

Using the new generic DW_Foundation and DW_Foundation_comp packages may impact CPU run time and memory usage by a negligible amount, as compared to using a single DW01 (or DW02, DW03, etc.) package. Table 7-1 identifies old and new packages. Table 7-2 shows the possible impact of using these generic packages.

Table 7-1 Old and New DesignWare Building Block IP Packages (November 1998 EST Release)

Signed/Unsigned Standard	Package Name	Content
Synopsys std_logic_arith (OLD)	DWpackages	Some commonly used functions, such as minimum(), maximum(), bit_width()
	DW01_components	Declarations of components and functions in DW01
	DW02_components	Declarations of components and functions in DW02
	DW03_components	Declarations of components and functions in DW03
	DW04_components	Declarations of components and functions in DW04
	DW06_components	Declarations of components and functions in DW06
	DW07_components	Declarations of components and functions in DW07
	DW08_components	Declarations of components in DW08 (new for November, 1998 EST)
IEEE numeric_std (NEW)	DW_Foundation	Some commonly used functions previously defined in DWpackages, such as minimum(), maximum(), bit_width, and all functions defined in DW01, DW02, DW03, DW04, DW06, and DW07.
	DW_Foundation_comp	Declarations of all components in DW01, DW02, DW03, DW04, DW06, and DW07, and the new DW08 family.

Table 7-1 Old and New DesignWare Building Block IP Packages (November 1998 EST Release) (Continued)

Signed/Unsigned Standard	Package Name	Content
Synopsys std_logic_arith (NEW)	DW_Foundation_arith	Some commonly used functions previously defined in DWpackages, such as minimum(), maximum(), bit_width(), and all functions defined in DW01, DW02, DW03, DW04, DW06, and DW07.
	DW_Foundation_comp_arith	Declarations of all components in DW01, DW02, DW03, DW04, DW06, and DW07, and the new DW08 family.

Table 7-2 CPU Runtime and Memory Usage Impact

	DW_Foundation_arith DW_Foundation_comp_arith	**DW_Foundation DW_Foundation_comp**
Vhdlan	<0.06s	<0.1s
Vhdlan -i	<0.08s	<0.15s
dc_shell analyze for synthesis	<0.15s <5MB	<0.15s <5MB
elaboration+compiling	<2s <3MB	<2s <3MB
Note: Building Block IP releases before 9/98 do not support the IEEE numeric standard. For comparison purposes, a design that is compatible with both packages was used for measurement. In the benchmark, designs using DW01_components is comparable to the same design using DW_Foundation (_arith) and DW_Foundation_comp (_arith), although this comparison is not favorable to the latter.		

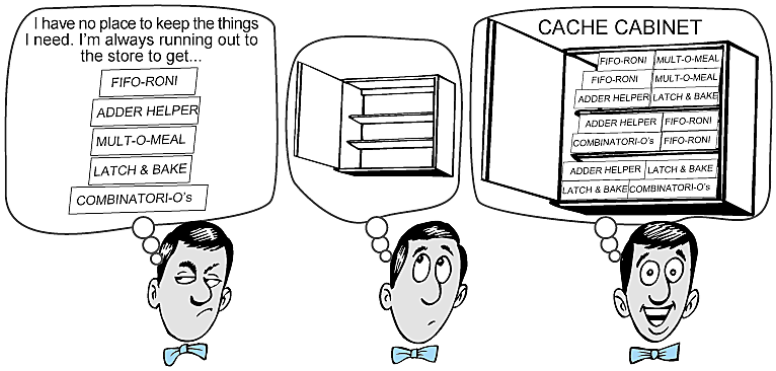
Speed Up Your Compile Time Using Synthetic Cache

AN 99-004

Compile time is becoming a key concern for users of synthesis tools as designs increase in complexity. Different techniques are used to speed up compile time, including better hardware, improved algorithms, and better coding styles.

Synthetic Cache is like a kitchen cabinet; without one, you are always running out to pick up the things you need. A "Cache Cabinet" allows you to store your favorite items, in different sizes, for fast and easy access (see Figure 8-1).

Figure 8-1 "Cache Cabinet"



The synthetic library cache feature of any Synopsys design tool that supports the DesignWare Building Block IP is a powerful aide in reducing compile time for today's complex designs. Several Synopsys tools support DesignWare Building Block IP. These include Design Compiler and DFT Compiler.

The synthetic library cache feature of synthesis tools improves synthesis runtime by caching implementations of DesignWare IP as they are built. No user action is necessary to take advantage of the synthetic library cache when using Synopsys tools; by default, caching is automatically turned on to improve runtime. However, by understanding synthetic cache behavior and taking advantage of the controls provided in synthesis tools, you can tune the cache's operation to your environment. This article discusses the cache's implementation and its control features, and provides strategies for effective cache management.

Current Implementation

The synthetic caching mechanism improves synthesis runtime performance by caching an implementation once it is generated. For each DesignWare IP generated, an optimized timing model and generic netlist are placed in the synthetic library cache. Once cached, if these components are required for a subsequent synthesis operation (for example, `compile`, `replace_synthetic`, or `schedule`), they are read from the cache instead of being rebuilt.

Timing models and netlists placed in the synthetic library cache are uniquely identified based on the following:

- Component name
- Synthetic library name

- Set of parameters used to build the component
- Operating environment under which the component was optimized

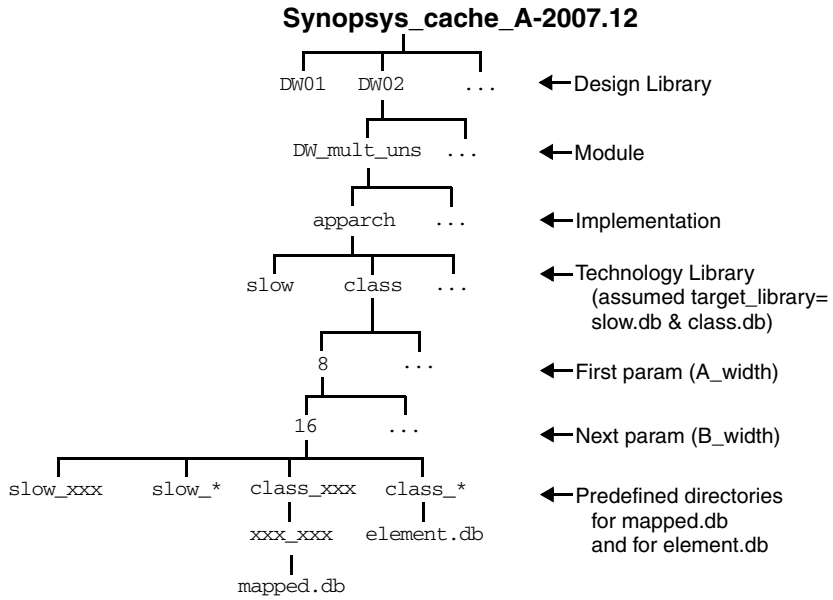
The synthetic cache mechanism creates a directory structure that uniquely identifies components according to the following:

- Component name
- Design library name
- Parameter values
- Target technology library list
- Wire load model name
- Operating condition name

Additionally, a time stamp for both the synthetic library and the target technology library are stored as attributes on the components stored in the cache.

To illustrate the cache directory structure of a synthetic component, the “apparch” implementation of the DW02_mult module is used as an example. For this example, the target technology library has been set to `{slow.db, class.db}`, and the parameters are (`A_width = 8`, `B_width = 16`). The cache directory structure for this component is illustrated in Figure 8-2.

Figure 8-2 The Cache Directory Structure of a Synthetic Component



How the Cache Mechanism Works

The dc_shell variables cache_write and cache_read are used to turn the caching mechanism of the synthetic components on and off. Detailed information on how to use the caching mechanism is found under “Maintaining the Cache” in the DesignWare User Guide.

The variable cache_write specifies the location at which the cache root directory is created. The default value for cache_write is the "~" string (your home directory). The name of the root directory consists of the "synopsys_cache" string and the software release version (for example, the A-2007.12 release cache root directory is named "synopsys_cache_A-2007.12").

The cache_read variable is a list variable that has the default value of '{"~"}'. Each string in this list represents a path (the full path or relative path) to a directory that contains a synthetic cache root. Each directory in this list is searched for the synthetic components in the order specified in the dc_shell variable in .synopsys_dc_setup, or your synthesis script. The permissions for the synthetic cache directory and included files are controlled by the cache_dir_chmod_octal and cache_file_chmod_octal variables respectively. The default permission for the directory is "777"; the default permission for the files is "666". Permissions for the synthetic cache directory and files behave in the same manner as the UNIX environment.

For a detailed description of other cache control variables, refer to the DesignWare User Guide.

Cache Invalidation

It is important to ensure that cached information is consistent with the current operating environment. Synopsys tools invalidate cache elements under the following conditions:

- For every new version of the synthesis tool, previous caches are considered invalid.
- Technology library timestamps are saved in the cached elements and used to validate the cache. Therefore, newly created libraries make the cached elements invalid.
- The order of the technology libraries that are specified by the target_library variable controls the validity of the cache element. Therefore, different orders of the target libraries cause cache elements to be invalidated. If the order of libraries in the target_library list changes, new cache elements are generated.
- Adding a new technology library to the target_library list causes invalid caches. For instance, if you add a technology library that contains only wire load models, the existing cache is not used, regardless of whether any of the wire load models are used in the design.
- A newer version of the synthetic library invalidates cached elements.
- If the cached element is corrupted (for example, an unrecognized file), the cache becomes invalid.

Each time an invalid element is detected, it is removed before a new cached element is created. If the entire cache is invalid, a new cache structure is created.

Tips for Improving Cache Usage

Although the synthetic cache operates in the background, there are several ways to make better use of the caching mechanism:

- Share the cache
- Populate the cache with the most commonly used components
- Reduce the runtime for the cache creation step
- Limit the frequency of cache invalidation

Sharing the Cache

A project team can set up a cache hierarchy to maximize the reusability of the cache. For instance, you can set up an individual-level cache and a project-level cache as the default cache_read for everyone on the same project. Another approach is to set up the default cache_write as an individual-level cache, as shown below:

```
set cache_read {"individual_cache" "project_cache"}
set cache_write {individual_cache}
```

The create_cache command populates the project cache with the most commonly used components. The individual cache contains components that are either not found in the project cache or have different default setups. It is best to periodically update the project cache using the `create_cache` command as more components are identified.

You can also set up only one cache location for everyone on a project. However, the problem of the individuals overriding each others' cached elements may arise.

Populating the Cache with the Most Commonly Used Components

Use the create_cache command to pre-populate the cache with components that are widely used in a project. Refer to "Controlling the Cache" in the DesignWare User Guide for the procedure on pre-populating the cache. Manage your cache by using the `report_cache`, `remove_cache`, `cache_ls`, and `cache_rm` commands. Refer to "Controlling the Cache" in the DesignWare User Guide for detailed information on the syntax for remove_cache and report_cache.

The cache_rm and cache_ls UNIX commands are the equivalent of the remove_cache and report_cache dc_shell commands, respectively. The syntax for these commands is shown in Example 8-1.

Example 8-1 Command Syntax

```
cache_ls <cache_dir> <regular expression>
  Example: cache_ls ~/my_cache/synopsys_cache_A-2007.12 "lsi_10k|generic"
           cache_ls ~/my_cache/synopsys_cache_A-2007.12 add
           cache_rm <cache_dir> <regular expression>
  Example: cache_rm
           ~/my_cache/synopsys_cache_A-2007.12 lsi_10k
           cache_rm ~/my_cache/synopsys_cache_A-2007.12 DW01_add
```

Reducing the Runtime for the Cache Creation Step

To reduce the runtime for the cache creation step, set the synlib_optimize_non_cache_elements variable to FALSE. This, in effect, turns off the logic optimization step for the new implementation model. An unoptimized implementation model decreases the quality of results of timing-driven resource sharing and implementation selection. Therefore, it is important that your design is not timing-sensitive before considering this option.

Limiting the Frequency of Cache Invalidation

One of the most common cases for cache invalidation is changes in the target_library list—whether it is the specified order of the technology libraries or the difference in the number of technology libraries.

You can reduce the frequency of cache invalidation due to changes in the target_library list as follows:

- Determine up front all the technology libraries that you intend to use for your design, and set the target_library list based on your preferred order. Use this target_library list for all your synthesis runs.
- When the information in a technology library does not affect the QOR (Quality of Results) of the blocks containing synthetic components, you may add this library to the link_library list instead of the target_library list. For example, you can safely add a technology library onto the link_library list if it only contains new wire load models that are not used on the blocks that contain synthetic components.

Limitations

There are some limitations to consider regarding the current synthetic cache:

- Before the creation of the timing model for a synthetic component, the component is optimized with a default map effort. The mapping effort is not saved in the cached element. Therefore, the synthesis QOR may be impacted by using a low effort (or no effort) optimized model in a timing-sensitive design. Refer to the DesignWare User Guide for more information.
- The cache_ls and the cache_rm commands are not currently supported on the NT platform.

Conclusion

There are many ways to increase the productivity of your design efforts by taking advantage of the synthesis tool's user-controllable features and avoiding the pitfalls that cause invalid caches. Understanding how the synthetic caching mechanism works enhances your productivity and QOR.

Pressure to pack more and more of a system onto a single die leads to designs with relatively large sub-chip blocks. These blocks are often designed each with its own autonomous control over data flow into and/or out of its interfaces to other blocks within the chip. The result is a collection of blocks that must be connected together to allow data to flow as illustrated in Figure 9-1.

The most suitable method of connecting data ports between blocks of a given design depends on such factors as flow dynamics differences between the sending and receiving blocks as well as the desired data interface as illustrated in Figure 9-2. The DesignWare Building Block IP has ready-to-use solutions to data connectivity problems in ASIC designs.

Figure 9-1 Data Plumbing in ASICs

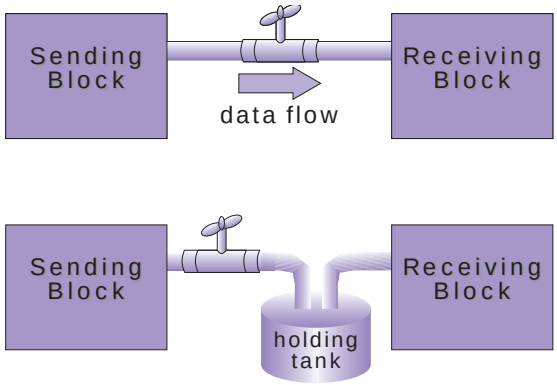
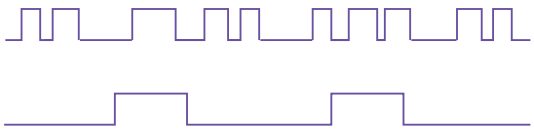


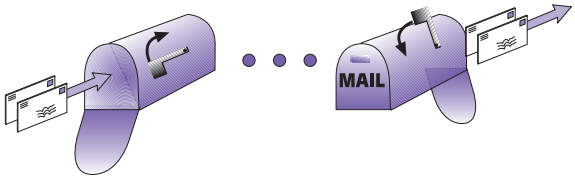
Figure 9-2 Different Blocks Often Have Different Flow Schedules



Dual-Port RAM With Flags

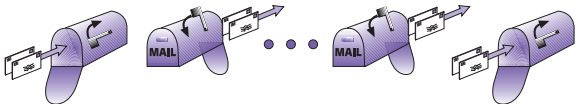
One common method of connecting data streams between two blocks is using a dual-port RAM with flags. When using a dual-port RAM, with the sending block connected to the write port and the receiving block connected to the read port, as temporary storage of data in transit, semaphore flag logic is used to manage control of the RAM (allocation and freeing of “buffers”). This technique (known as single buffering as illustrated in Figure 9-3) has the undesirable side affect of forcing the sending block to wait for the receiving block to process one batch of data before the next can be written to the one and only buffer.

Figure 9-3 Single Buffered



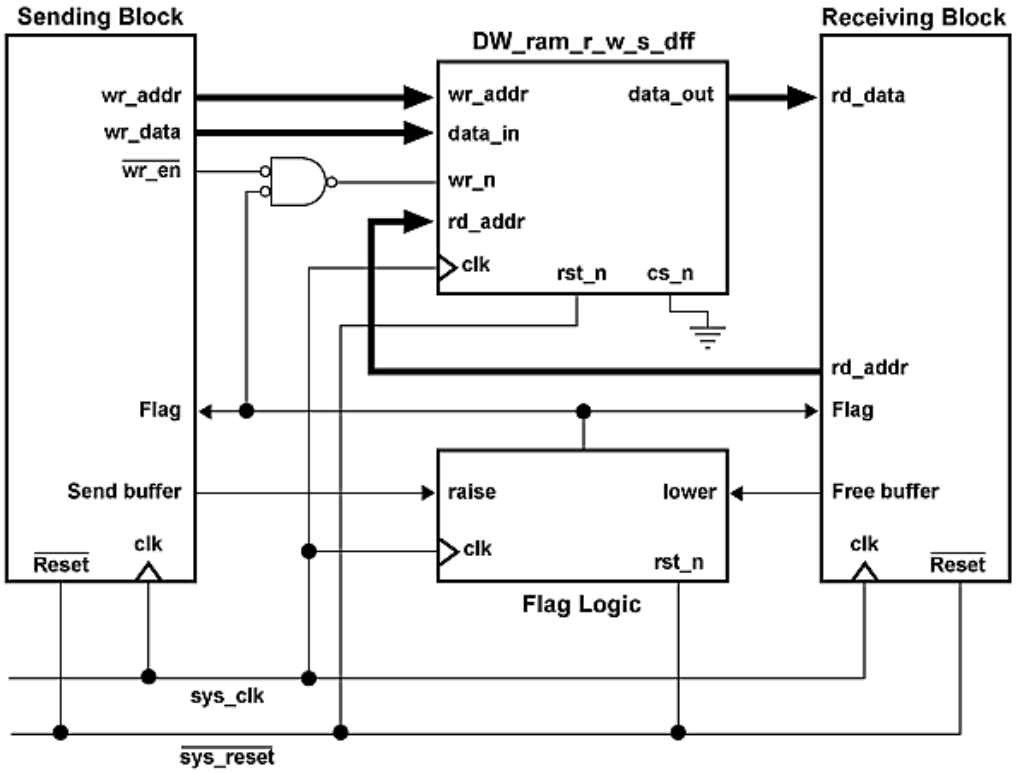
A similar technique uses a second buffer (usually by doubling the size of the RAM instead of using two RAMs) and a second flag. This technique (called double buffering or ping-pong buffering as illustrated in Figure 9-4) allows data to be shuttled efficiently in buffered packets. Dual-port RAM buffering schemes are attractive data connectivity techniques in designs where the interfacing blocks already contain address and data bus facilities (as in blocks with embedded microprocessor cores). RAM buffer methods easily allow out-of-order access of data within a buffer, as well as multiple access of data elements within a buffer.

Figure 9-4 Double Buffered



When small buffers are needed, Synopsys' DesignWare RAMs offer very high-performance buffers without the hassle of dealing with memory compilers and RAM built-in-self-test (RAM BIST). With D-type flip-flop-based synchronous dual-port RAMs (like the DW_ram_r_w_s_dff), you can use traditional scan chain testing to avoid the need for RAM BIST. See Figure 9-5. Also, since the RAM is essentially a flip-flop-based state-machine, its performance is automatically controlled and monitored by Synopsys' Design Compiler™ in the same way that other sequential logic in the design is handled.

Figure 9-5 RAM Buffer with Semaphore Flag



FIFOs

Another common method of connecting data streams between two blocks uses a first-in-first-out buffer (FIFO). FIFOs fit well into designs that need to connect blocks that have relatively continuous data flow but at differing rate and/or burst characteristics. FIFOs interface to sending blocks with a data bus and a control line used to request that the FIFO push the value on the data bus into the FIFO. Similarly, the FIFOs interface to receiving blocks with a data bus and a control line to request that the FIFO pop a value out of the FIFO. In addition to the data and control signals, each block can monitor the FIFO's status flags. These flags usually include an output to indicate the FIFO is empty and an output to indicate the FIFO is full.

Most FIFOs also provide an output to indicate the FIFO is almost empty and another to indicate the FIFO is almost full. In some cases, these status outputs may be condensed into fewer signals by encoding the state of the FIFO.

The almost empty and almost full flags are often used to help maintain a continuous flow of data through the FIFO without encountering overflow or underflow errors. Overflow errors occur when the sending block attempts to push data into a full FIFO, while underflow errors occur when the receiving block attempts to pop data out of a FIFO that is empty. If the sending block can stop the flow of its data in a single clock cycle, then the full flag on the FIFO is sufficient for flow control status from the FIFO. However, if it takes more than a single cycle to stop the sending block's data flow, then you can use the almost full flag as an "early warning" to allow enough time to stop the data flow before an overflow error occurs. Conversely, if the receiving block can stop its data flow (i.e., stop requesting data to be popped) in a single cycle, then the FIFO empty flag is adequate status for flow control. However, if the receiving block requires more than a single clock cycle to halt its request for data, then you will typically employ the almost empty flag to give fair warning to the receiving block.

When aggregate flow rates are slightly mismatched in an unknown direction (as in data repeaters between systems running off of separate crystals), the probability of overflows and underflows can be minimized by filling the FIFO to the halfway mark at the beginning of each burst of data before the receiving block begins its retransmission. A half-full flag on a FIFO can help coordinate the receiving block to the sending block in such a case.

DesignWare's FIFOs and FIFO controllers provide all the features required to meet virtually any continuous flow application. The DW_fifo_s1_sf and DW_fifo_s1_df FIFOs offer self-contained FIFO modules with `almost_empty` and `almost_full` flags that are statically or dynamically programmable.

The FIFO modules are meant for moderately small aggregate buffer sizes. When larger FIFOs are required, you can use one of the FIFO controllers (DW_fifoctl_s1_sf or DW_fifoctl_s1_df), along with compiled synchronous memory modules from your ASIC vendor.

All four of these new modules provide a full compliment of status flags (`empty`, `almost_empty`, `half_full`, `almost_full`, and `full`), as well as a configurable error indicator. In systems where flow dynamics don't change with changing configurations and outside parameters, you can determine the optimal `almost_empty` and `almost_full` slack during system design, and compile these values "statically" into their chips (DW_fifo_s1_sf FIFO and DW_fifoctl_s1_sf FIFO controller have "static" flags).

When it is determined that the flow dynamics of a system will change under various configurations or outside conditions, you can opt to use dynamic `almost_empty` and `almost_full` flags and have the system itself determine the optimal slack for almost empty and/or almost full either at system initialization and/or during periodic performance characterizations. (DW_fifo_s1_df FIFO and the DW_fifoctl_s1_df FIFO controller have dynamic flags).

Figure 9-6 Single Module FIFO

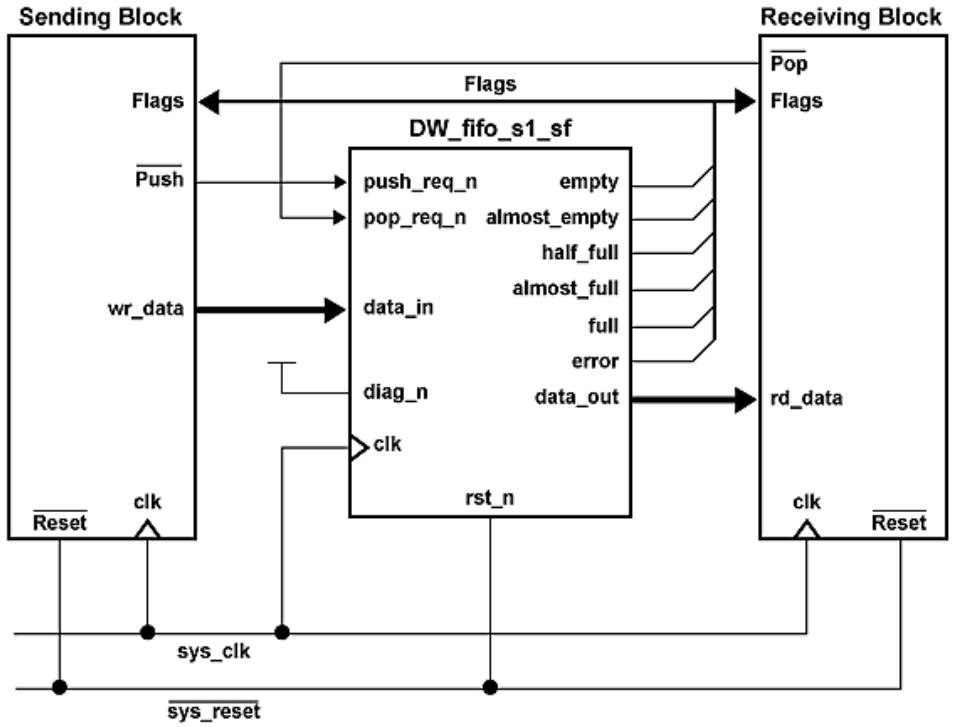
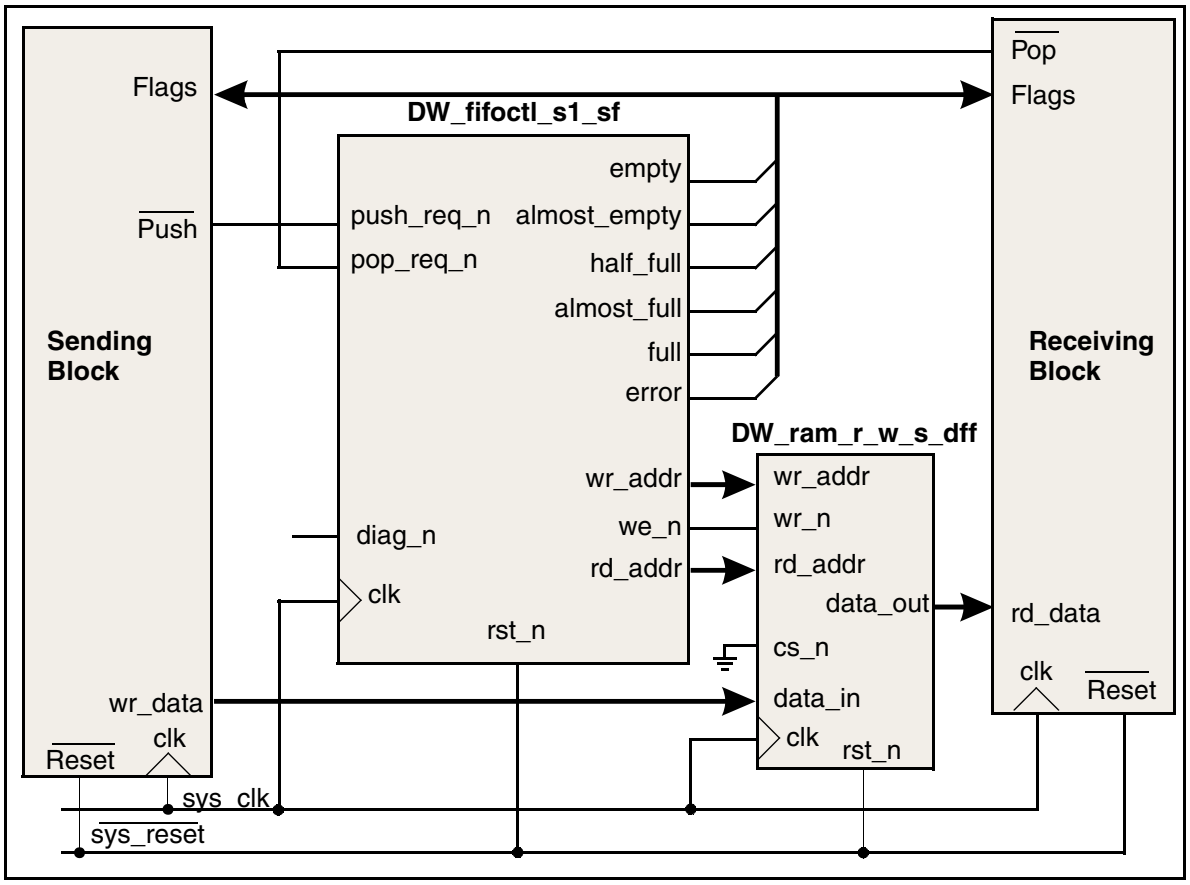


Figure 9-7 FIFO Controller with RAM Module

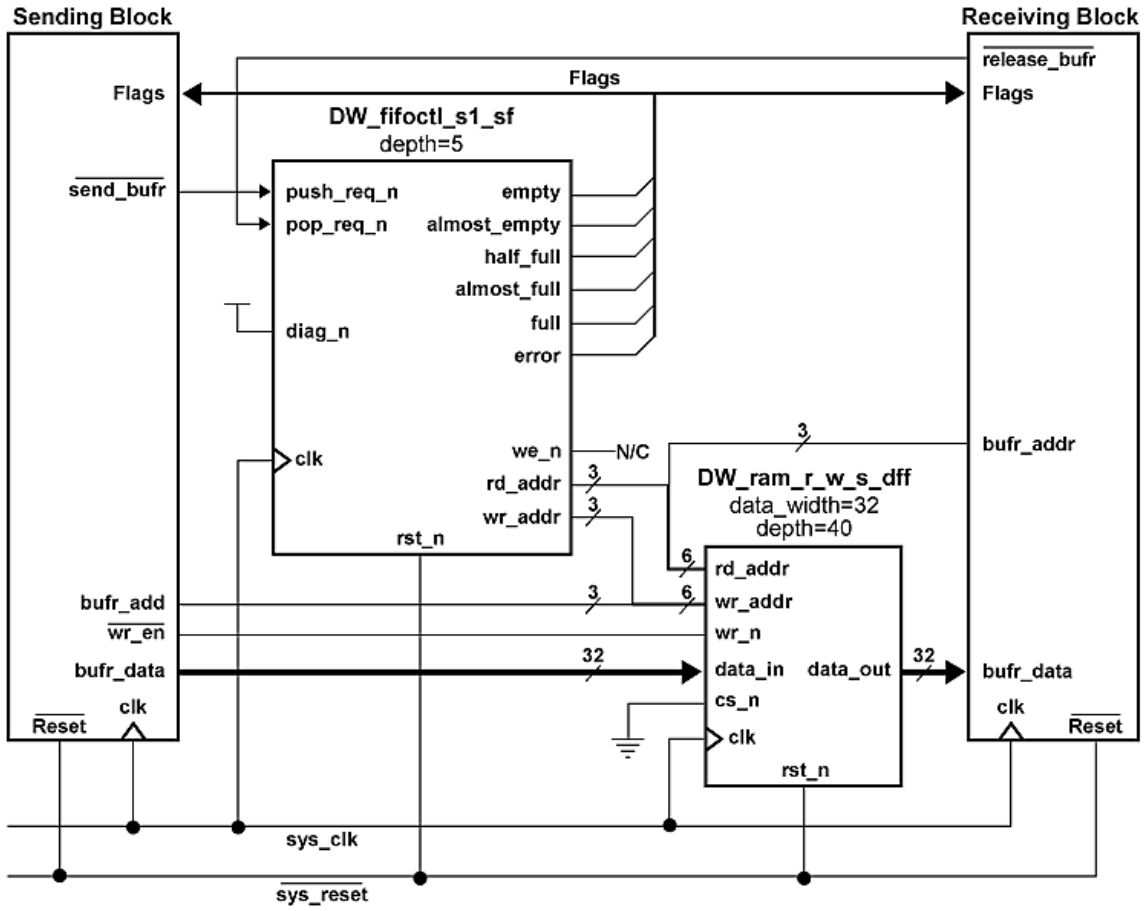


FIFOed RAM Buffers

Some system designs that require RAM buffers cannot achieve optimal performance with only one or even two buffers. Such systems may need to use a RAM buffer interface to process discrete packages of data, but need to be able to collect a number of these buffers for processing (the scheduling of sending block activities may be very different from the scheduling of receiving block activities).

Using a FIFO controller to allocate and free buffers and provide status flags for sending and receiving blocks, a RAM block can be conveniently split into many buffers. This is accomplished by simply connecting the RAM addresses given by the FIFO controller to the upper address bits of the two ports of the RAM, and controlling the lower address bits by the appropriate sending or receiving block (see Figure 9-8). This hybrid FIFO-RAM buffer approach easily lends advantages of both RAM buffer and FIFO techniques to a system design. Once again, DesignWare Building Block IP FIFO controller and RAM synthetic modules offer the functions to meet the demands of this approach.

Figure 9-8 Example of FIFOed Buffers



Summary

ASIC designs often need to make use of some sort of “data plumbing” to connect “data faucets” of differing flow characteristics together. Methods used vary as needed from dual-port RAM buffers to traditional FIFOs to FIFOed buffers. In all of these cases, DesignWare Building Block IP synthetic RAM, FIFO, and/or FIFO Controller modules meet your needs.

What Is Metastability, And How To Live With It

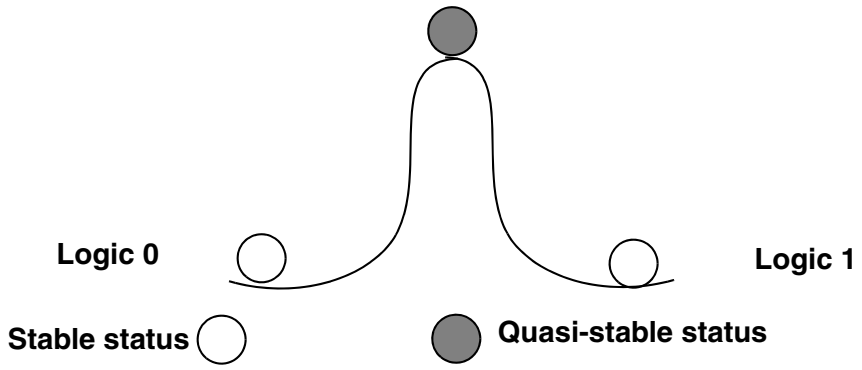
AN 99-001

Designers normally consider flip-flops to be devices that have two stable states—logic 0 and logic 1. However, because of the nature of their construction, there is a third quasi-stable state between logic 0 and logic 1. A flip-flop in this quasi-stable state is said to be experiencing metastability.

What is a “Quasi-Stable” State?

The relative stability of states shown in Figure 10-1 shows that the logic 0 and logic 1 states (being at the base of the hill) are much more stable than the somewhat stable state at the top of the hill. In theory, a flip-flop in this quasi-stable hilltop state could remain there indefinitely – but in reality it won’t. Just as the slightest air current would eventually cause a ball on the illustrated hill to roll down one side or the other, thermal and induced noise will jostle the state of the flip-flop causing it to move from the quasi-stable state into either the logic 0 or logic 1 state.

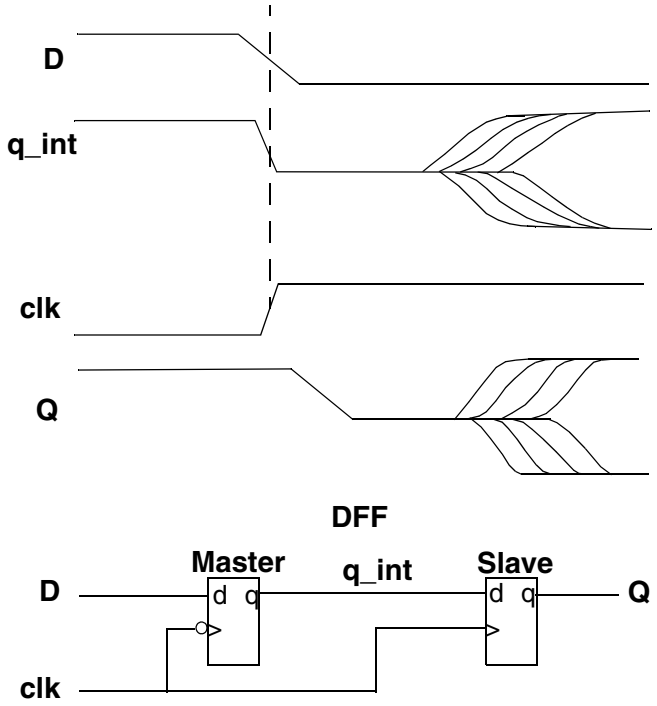
Figure 10-1 "Ball on a Hill" Analogy of a Quasi-Stable State



How Does a Flip-Flop Get Into a Quasi-Stable State?

The quasi-stable state is reached when a flip-flop’s setup and hold times are grossly violated. Assuming the use of a positive edge triggered “D” type flip-flop, when the rising edge of the flip-flop’s clock occurs at a point in time when the `D` input to the flip-flop is causing its master latch to transition, the flip-flop is highly likely to end up in a quasi-stable state (see Figure 10-2). This rising clock causes the master latch to try to capture its current value while the slave latch is opened allowing the `Q` output to follow the “latched” value of the master. The more perfectly “caught” quasi-stable state (on the very top of the hill) results in the longest time required for the flip-flop to resolve itself to one of the more stable states.

Figure 10-2 Internal View of a Metastable Event



The probability of a single random transition on the `D` input of a synchronizing flip-flop to cause metastability is proportional to the amount of time it takes for `q_int` to pass through the "twilight zone" between stable states divided by the period of the synchronizing clock. Even though this probability is, in most cases, very low for a single event, the probability of metastability over a specific period of time is multiplied by the number of asynchronous transitions seen on the `D` input during this period of time.

To complicate matters even more, abnormal propagation delays can, and do, occur in synchronizing flip-flops even if quasi-stable states are not reached. If the setup or hold time of a flip-flop is violated even slightly, the clock-to-`Q` delay will most likely be increased, even though the flip-flop has successfully sampled the correct value.

So How Do You Stay Off the "Hill"?

In reality, metastability and increased clock-to-`Q` delays cannot be avoided when synchronizing asynchronous inputs without the use of tricky self-timed circuits. So a more appropriate question might be "How do you tolerate metastability?"

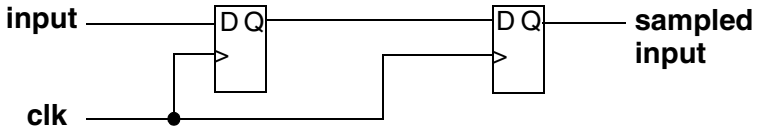
In the simplest case, you can tolerate metastability by making sure the clock period is long enough to allow for the resolution of quasi-stable states, as well as whatever logic may be in the path to the next flip-flop. This approach, while simple, is rarely practical given the performance requirements of most modern designs.

The most common way to tolerate metastability is to add one or more successive synchronizing flip-flops to the synchronizer. This approach allows for an entire clock period (except for the setup time of the second flip-flop) for metastable events in the first synchronizing flip-flop to resolve themselves. This does, however, increase the latency in the synchronous logic's observation of input changes.

Neither of these approaches can guarantee that metastability cannot pass through the synchronizer, they simply reduce the probability to practical levels.

In quantitative terms, if the Mean Time Between Failure (MTBF) of a particular flip-flop in the context of a given clock rate and input transition rate is 20,000 seconds (about 5.55 hours) then the MTBF of two such flip-flops used to synchronize the input would be (20,000 × 20,000) = 400,000,000 seconds (about 12.67 years).

Figure 10-3 Increasing MTBF Through Double Synchronization



DesignWare Building Block IP includes a variety of parameterizable synchronizers (see Overview of Synchronizers). Each of these synchronizers uses a different method of crossing clock domains.

Functional Inference In Verilog: Signed/Unsigned

AN 98-004

Introduction

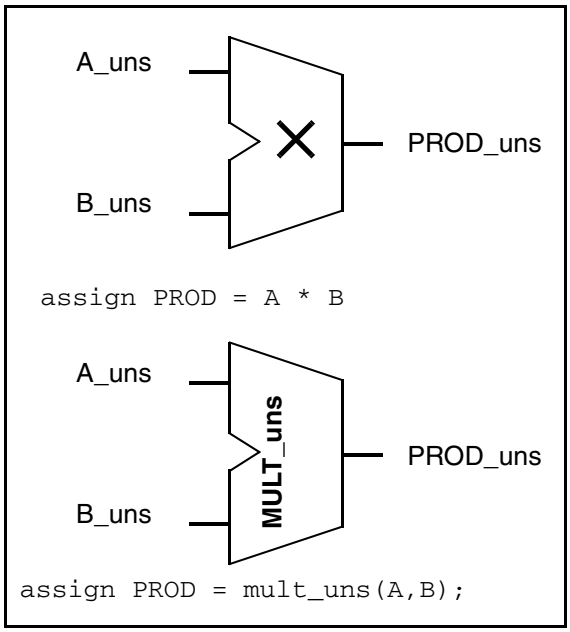
DesignWare Building Block IP are accessible either by instantiating or inferring them through operators and functions. When you infer an IP with a VHDL operator, you can define the input data types as signed, unsigned, std_logic, and so on. Therefore, VHDL provides the flexibility to do arithmetic operations in signed or unsigned mode.

Verilog, however, does not provide such a mechanism. If you infer an IP through a Verilog operator, you cannot specify signed mode or unsigned mode. Verilog, therefore, restricts operator inferencing to unsigned operations. As a solution, DesignWare provides Verilog functions that infer DesignWare Building Block IP for both signed and unsigned arithmetic operations.

Unsigned Inference

If the input operands are unsigned numbers, you can infer DW02_mult through either the standard "*" operator or through the mult_uns function. Both methods yield the same result, as illustrated in Figure 11-1.

Figure 11-1 Unsigned Inference



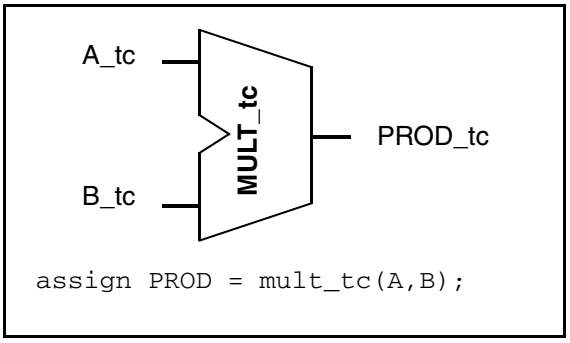
Signed Inference

If the input operands are signed numbers, you cannot use the standard Verilog "*" operator to infer a multiplier, because the "*" operator does not recognize the types of the inputs. The inputs are interpreted as unsigned numbers, and the compiler builds incorrect logic. Instead, use the mult_tc function to multiply two signed numbers, as illustrated in the following example and Figure 11-2:

```
assign PROD = mult_tc(A,B);
```

A and B are signed numbers that are inputs to the multiplier, and PROD is the output from the multiplier in signed mode.

Figure 11-2 Signed Inference, Example 1



To use the DesignWare Verilog functions, you must include the file that defines the function, as shown in the Verilog examples in this databook and the example in the "Multiple Inferencing" topic in this application note.

Certain arithmetic operators (for example, "+") do not need to know the type of operand and, therefore, no special function is needed for signed inputs. You can use the Verilog operator directly, as shown below:

```
assign SUM = A + B;
```

Multiple Inferencing

You can infer the same function multiple times with different parameter values (for example, for inputs of different bit-widths). The first time you infer the function, define the parameter values and map the parameters to the function using the Verilog parameter statement. To infer the function a second time, use Verilog defparam statements to redefine the parameter values. You can continue to use defparam statements to redefine the parameter values and infer the function multiple times.

When using the defparam statement for function inference, Synopsys HDL Compiler™ issues a warning about the usage of defparam (which can be disregarded), unless you use the translate off/on feature. The defparam directive is used only for simulation; Synopsys HDL Compiler does not support the use of defparam to redefine parameter values.

For inference functions (functions using the map_to_operator pragma) the Synopsys HDL Compiler infers parameter values directly from the widths of signals connected to the function making the use of courier unnecessary for synthesis.

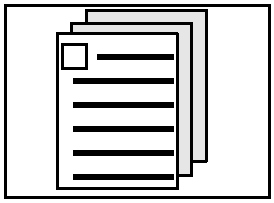
The following example illustrates the above technique for inferring the same function multiple times:

```
module multpl_infr( a1, a2, b1, b2, c1, c2 );

  parameter width1 = 4;
  parameter width2 = 6;

  input [width1-1 : 0] a1, b1;
  input [width2-1 : 0] a2, b2;

  output [2*width1-1 : 0] c1;
output [2*width2-1 : 0] c2;

    // For parameters used in inference functions, give initial values
```

```
  // that are greater than or equal to the maximum values used
  // for any particular function call
parameter A_width = 64;
parameter B_width = 64;

  // Add +incdir+$SYNOPSYS/dw/sim_ver+
  // to your verilog simulator command line (for simulation)
`include "DW02_mult_function.inc"

  // defparam directives will be used for simulation
  // only.  So, it will be hidden from DC
// synopsys translate_off
defparam A_width = width1;
defparam B_width = width1;
// synopsys translate_on

  // First inference using A_width = B_width = width1
assign c1 = DWF_mult_tc( a1, b1 );


  // defparam directives will be used for simulation
  // only.  So, it will be hidden from DC
  // synopsys translate_off
defparam A_width = width2;
defparam B_width = width2;
  // synopsys translate_on

  // Second inference using A_width = B_width = width2
assign c2 = DWF_mult_tc( a2, b2 );

endmodule
```

Summary

The Verilog functions in DesignWare Building Block IP provide the flexibility for you to infer arithmetic IP in both signed and unsigned mode. You can also infer Verilog functions multiple times with different parameter values by making use of defparam statements. You will receive a warning message during the analyze phase, but this warning message does not affect the final compiled output.



DW Building Block IP Synthesis: “Under The Hood”

AN 98-002

Introduction

The process of elaborating and compiling your design is complex. This application note shows you “under the hood” of Design Compiler, and what occurs during the elaboration and compilation of designs that use DesignWare Building Block IP. After reading this application note, you will have a better understanding of how your designs are synthesized.

To illustrate the flow, we will use a simple design. This design infers two adders through the “+” operator and one subtractor through the “–” operator. It also contains an instantiation of the Building Block IP Counter (DW03_updn_ctr) and some glue logic to control the circuit.

This application note assumes a compile with `map_effort medium`:

- The map_only attributes are ignored if you are running `compile -map_effort high`.

Example 12-1 VHDL Code Example

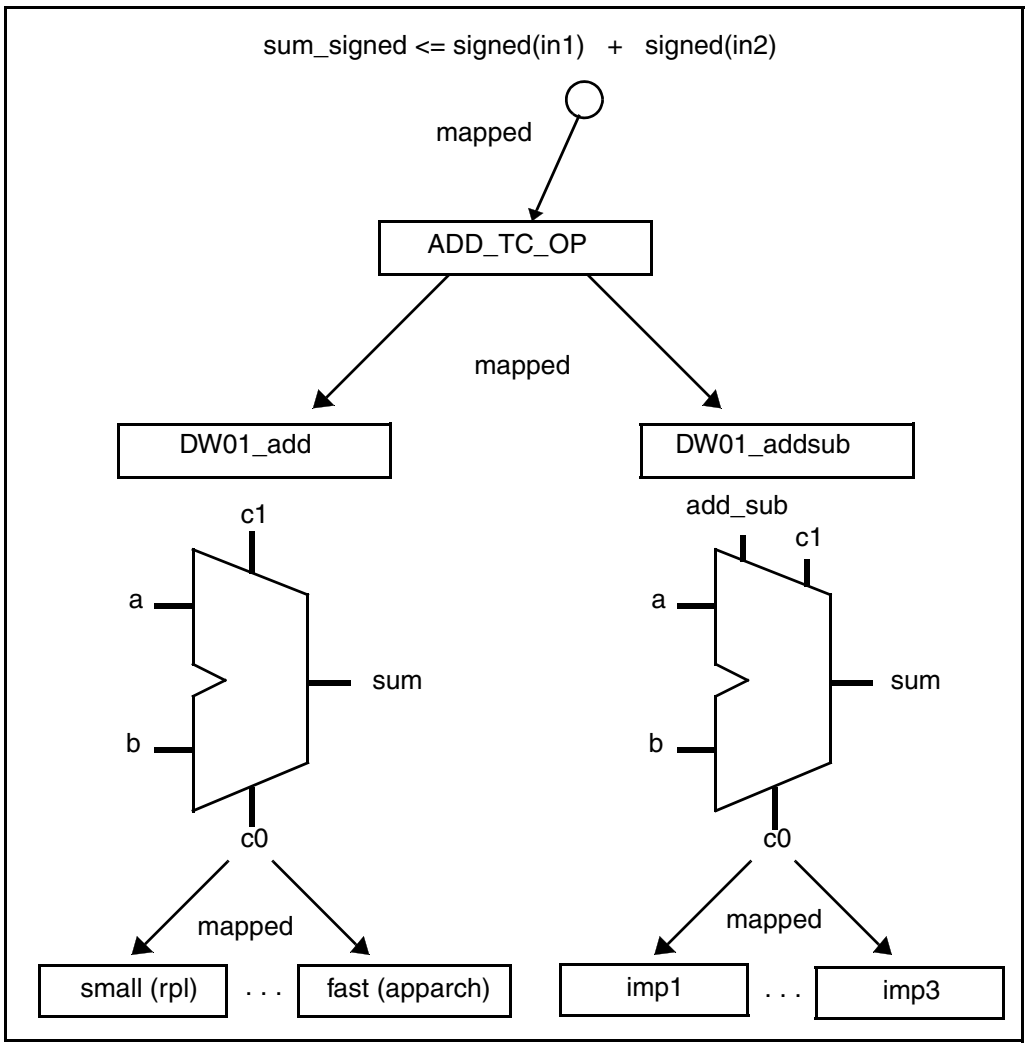
```
library IEEE, DW03;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use DW03.DW03_components.all;

entity DW_example is
  port( in1, in2, in3 : in std_logic_vector(7 downto 0);
        ctl1, ctl2, ctl3 : in std_logic;
        clk, rst  : in std_logic;
        data_out : out std_logic_vector(7 downto 0));

end DW_example;

architecture behv of DW_example is
  signal sum_signed : signed(7 downto 0);
  signal count_out : std_logic_vector(7 downto 0);
  signal load, control, logic_one, end_count : std_logic;
  constant bit_width : integer := 8;

begin
  control <= ctl2 or ctl3;
  logic_one <= '1';

  -- multiplexer logic
  load <= end_count when ctl2 = '1' else '1';
  process (in1, in2, ctl1)
  begin
    if (ctl1 = '0') then

      -- inferring adder through add_op
      sum_signed <= SIGNED(in1) + SIGNED(in2);
    else

      -- inferring subtracter through add_op
      sum_signed <= SIGNED(in1) - SIGNED(in2);
```

```
        end if;
      end process;

      -- instantiate up-down counter

      U1: DW03_updn_ctr
        generic map (width => bit_width)
        port map(data => in3,
                 up_dn => control,
                 load => load,
                 cen => logic_one,
                 clk => clk,
                 reset => rst,
                 count => count_out,
                 tercnt => end_count);

      -- inferring adder through add_op
      data_out <= sum_signed + SIGNED(count_out);

end behv;
```

dc_shell-t> elaborate

During elaboration, the inferred operators (+, –) in the HDL code are mapped to synthetic operators (refer to Figure 12-1). Synthetic operators are high-level functions that represent a number of DesignWare Building Block IP that can perform the required operation (in the example, add or subtract).

Figure 12-1 Mapping to Synthetic Operators

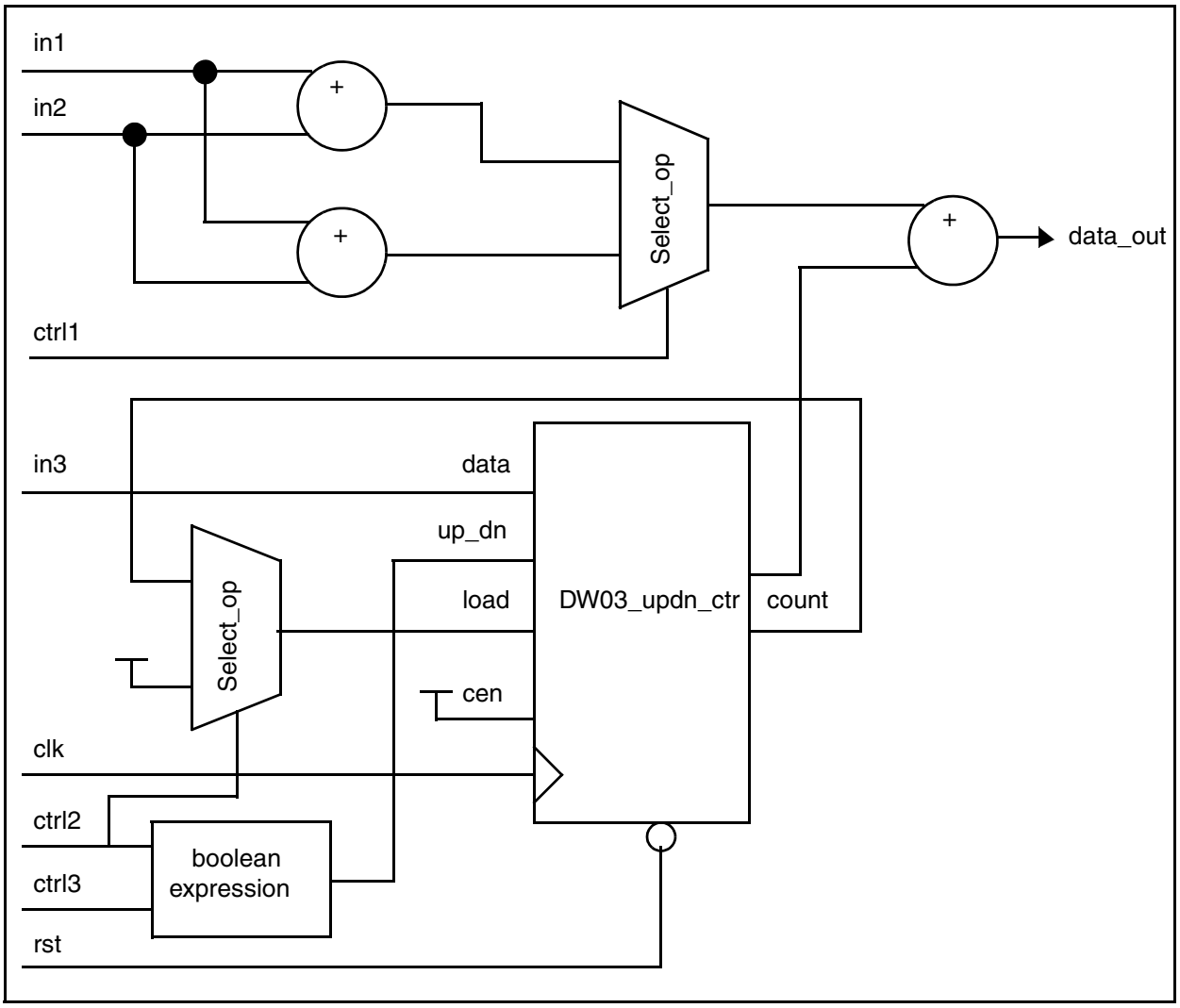


In Example 12-1 on page 55 the “+” operator is mapped to the ADD_TC_OP synthetic operator, and the “-” operator is mapped to the SUB_TC_OP synthetic operator. These synthetic operators are symbolically mapped to the following DW parts:

- DW01_add,
- DW01_addsub,
- DW01_dec,
- DW01_inc,
- DW01_incdec
- DW01_sub

Instantiated DesignWare parts are only connected at the top level. Information on the logic of these instantiated designs is not yet available. In Example 12-1, only the ports of the DW03_updn_ctr are known. The random/glue logic is mapped to Boolean expressions, and the random sequential elements are mapped to `seq_gens`, which is a high-level representation of a sequential circuit. Multiplexer logic is mapped into `select_ops`, which is a high-level representation of a mux (see Figure 12-2).

Figure 12-2 Design Example After Elaboration



dc_shell-t> compile -map_effort medium

The compile process has several steps:

- In the first step, the compiler does a quick map of all the Boolean expressions to equivalent cells from the technology library.
- In the second step, the compiler reads in the design constraints. If no constraints are provided, or only a set_max_area constraint is used, the compiler optimizes the design for area only. If any other constraints are present, including the *create_clock* command, the compiler optimizes the design to meet timing/clock constraints.
- In the third step, the compiler selects the DesignWare Building Block IP for each of the synthetic operators. In timing-driven compile, the compiler considers all possible IP for each synthetic operator. For an area-driven compile, the compiler tries to perform resource sharing to save area. Resource sharing may restrict the choice of IP that can be mapped to a shared operator.

Mapping Boolean Expressions

For the timing-driven compile, DW01_add, DW01_sub, and DW01_addsub IP are all considered. For example, in the sample design, one of the add_ops and the sub_ops is shared by using the DW01_addsub IP. Thus, for the area-driven compile, only the DW01_addsub is considered.

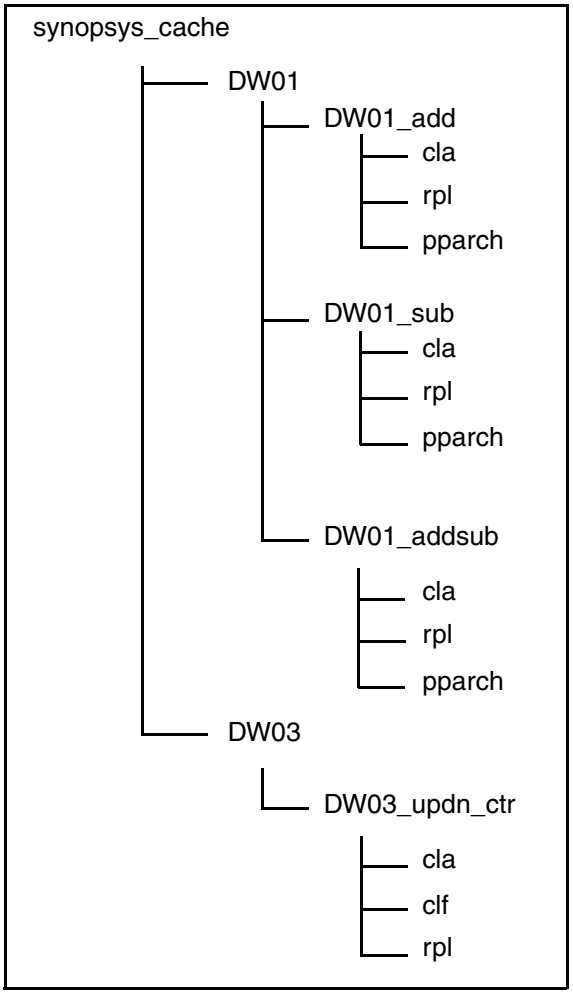
Reading Design Constraints

Once the compiler has selected the required IP and their different implementations, timing and area estimates are generated. These estimates are stored in the Synopsys cache in the form of a module file for each IP. If these estimates are not present, then the compiler maps the design to gates to calculate them. Timing models derived from these mapped designs are stored in the cache.

User-defined wire-load and operating conditions are considered when generating these estimates. If no operating conditions are available, then the default conditions are considered.

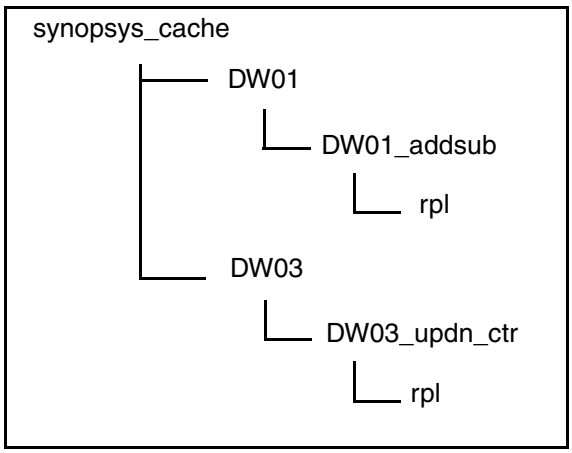
For a timing-driven compile, all of the timing models are generated for available implementations of an IP. For the sample design, the rpl, cla, and clf (only for DW02_updn_ctr) implementations for DW01_addsub, DW01_add, DW01_sub, and DW03_updn_ctr are modeled, and the timing models are cached (see Figure 12-3).

Figure 12-3 Cache Contents for Timing-Driven Compile



For an area-driven compile, a model is generated and cached for only the smallest implementation of the IP (see Figure 12-4). For the sample design, only the rpl implementation is built for the DW01_addsub, DW01_add, and DW03_updn_ctr.

Figure 12-4 Cache Contents for Area-Driven Compile



Selecting DesignWare Building Block IP

The compiler uses the following to choose an implementation that meets the design constraints (refer to Figure 12-5):

- Area and timing estimates of the DesignWare Building Block IP, and
- Area and timing information of the random logic (that has already been mapped to gates)

Note

The seq_gens are not mapped to sequential cells at this time; only timing/area estimates are used.

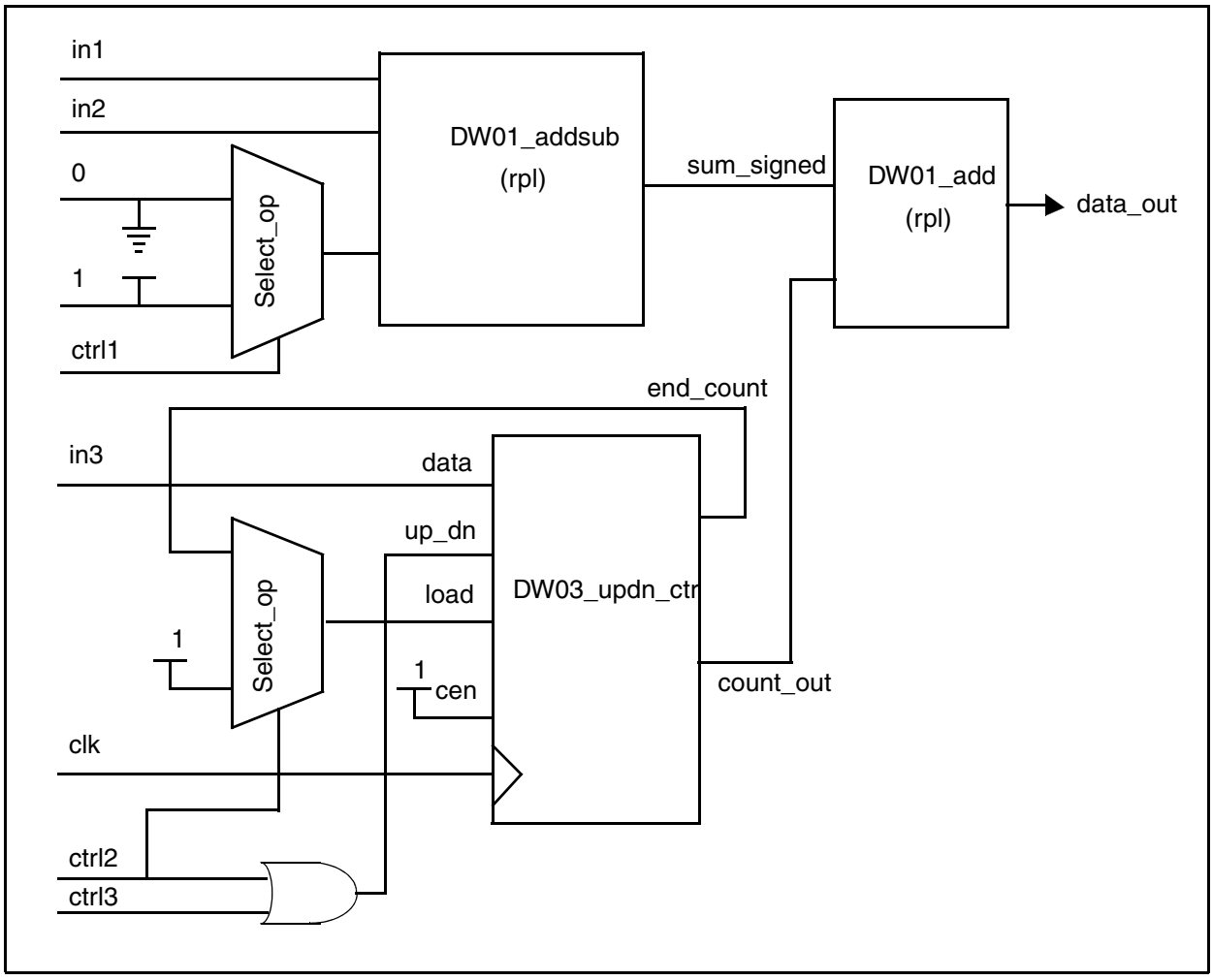
Next, the compiler performs a rough optimization recursively down the hierarchy, mapping the combinational logic of all the DesignWare Building Block IP. Any sequential cells in the IP are mapped to seq_gens. For the sample design, DW01_addsub, DW01_add, and the combinational logic of DW03_updn_ctr are mapped to the target technology. The sequential port of the DW03_updn_ctr is mapped to seq_gen.

The compiler then performs design optimization (boundary optimization, constant propagation) and generates a gate-level netlist for the design, with seq_gens in place of sequential cells.

After design optimization, the compiler performs incremental implementation selection (IIS). The compiler revisits the choice of the Building Block IP implementations. If another implementation exists that better meets the constraints, then the better implementation is swapped in and built from gates.

For timing-driven compiles, IIS is very important. In timing-driven compiles, the compiler traverses the most critical paths in the design. When it crosses a synthetic module in its path, it gets the timing model for each available implementation, plugs each implementation in the design, and calculates the cost function for this implementation. The compiler then selects the implementation with the best cost function.

Figure 12-5 Design Example After Implementation Selection (During Compile)



The Final Step

In the final step, the compiler maps the seq_gen cells into technology sequential cells and fully optimizes the entire design. This is the first time that sequential parts are mapped.

In understanding the processes of elaboration and compilation, you have a better overall view of what DesignWare Building Block IP can do to meet your design requirements.

Inferring Carry-In and Carry-Out Bits

AN 98-001

Introduction

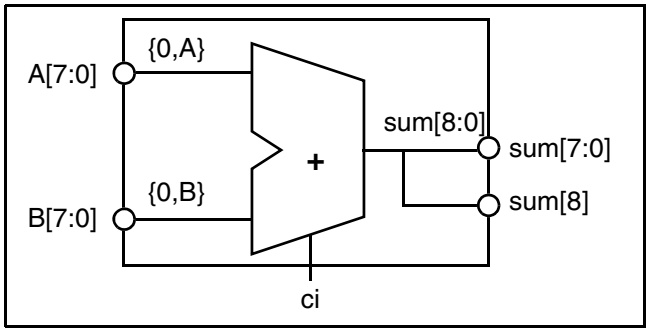
By making use of the basic operator inference mechanism, you can take advantage of the carry-in and carry-out capabilities of several of the arithmetic DesignWare Building Block IP. For example, through minor changes in your source code, you can infer IP such as the DesignWare Building Block IP adder and vector adder with carry-in and with the signed and unsigned modes of carry-out.

DW01_add and DW02_sum are used as examples to show how to infer arithmetic IP with carry-in and carry-out. The same techniques can be used with any inferable arithmetic IP that supports carry-in and carry-out.

Inferring DW01_add

You can infer the DesignWare Building Block IP adder (DW01_add) by using the "+" operator in either Verilog or VHDL. However, inferring DW01_add with a basic "A + B" expression does not make use of DW01_add's carry-in and carry-out pins. The following examples illustrate how to infer DW01_add with carry-in and carry-out:

Figure 13-1 Inferring DW_add with a Carry-In and Carry-Out.



Inferring Carry-out (Verilog)

To infer DW01_add with carry-out, pad the MSBs of the input pins with zeros and increase the bit-width of the output sum pin by one, as illustrated in the following Verilog code:

```
module DW01_add_co_test (COUT, SUM, A, B);
   parameter width = 8;
  input [width-1:0] A, B;
  output [width :0] SUM;

  output  COUT;
  wire [width :0] SUM;
  wire   COUT;
  assign {COUT,SUM} = A + B;
endmodule
```

Inferring Carry-in (Verilog)

To infer DW01_add with carry-in, declare the carry-in as an input pin to the module, and use the "+" operator to add the carry-in to the other two inputs. The following Verilog code infers DW01_add with both carry-in and carry-out, as illustrated in Figure 13-1:

```
module DW01_add_ci_co_test (SUM, A, B, CI);
  parameter width = 8;
  input [width-1:0] A, B;
  input CI;
  output [width :0] SUM;
  wire [width :0] SUM;
  assign SUM = {0, A} +
               {0, B} + CI;
endmodule
```

Inferring DW01_add with Signed Inputs (Verilog)

When the inputs to the "+" operator are signed, the carry-out bit has no logical significance and should be ignored. However, you can still infer the carry-in bit as illustrated in the previous example.

Inferring Carry-in and Carry-out (VHDL)

You can use the same techniques to infer DW01_add with carry-in and carry-out in VHDL. However, in VHDL you must declare the inputs as either signed or unsigned. Also, you must pad the carry-in bit with zeros on the MSB side so that the width of the carry-in signal matches the width of the other inputs. The following VHDL code infers DW01_add with carry-in and carry-out:

```
library IEEE, DW01; use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use DW01.DW01_components.all;
entity DW01_add_ci_co_test is
  generic(A_width : integer:= 16);
  port(A,B : in unsigned(A_width-1 downto 0);
        CI : in std_logic;
       SUM : out unsigned(A_width downto 0));
end DW01_add_ci_co_test;

architecture oper of DW01_add_ci_co_test is
  begin
  process (A, B, CI)
    variable cin_int:
    unsigned(A_width-1 downto 0) := (others => '0');
  begin
    cin_int(0) := CI;
    SUM <= ('0' & A) +
    ('0' & B) + cin_int;
  end process;
end oper;
```

Inferring DW02_sum

You can infer the Building Block IP vector adder (DW02_sum) by using the DW_sum function in either Verilog or VHDL. However, to preserve the carry bits from additions within a vector sum, you must pad the MSB side of the incoming vector with extra bits. The extra bits ensure that the output port is wide enough to

contain the maximum sum value. The number of bits to be concatenated to the MSB of the incoming vector depends on the number of inputs (num_inputs). For num_inputs = n, the number of extra bits is ceil(ln_2[n]).

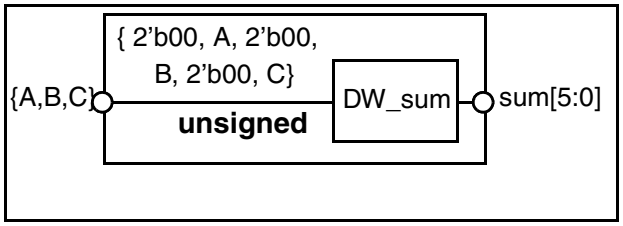
For unsigned numbers, pad the MSB of the incoming vector with ceil(ln_2[n]) zeros. For signed numbers, pad the incoming vector with a number of bits equal to ceil(ln_2[n]).

Verilog Example (Unsigned)

The following Verilog code infers DW02_sum to sum three 4-bit unsigned numbers, pads the MSB of each input with "00", and creates a 6-bit output port to accommodate the carry bits of the summation, as shown in Figure 13-2:

```
parameter num_inputs = 3, input_width = 6;
assign sum_out = DW_sum({ 2'b00, A, 2'b00, B, 2'b00, C });
```

Figure 13-2 Inferring Unsigned Vector Adder



Verilog Example (Signed)

The following Verilog code infers DW02_sum to sum three 4-bit signed numbers, pads the MSB of each input with two extra bits, and creates a 6-bit output port to accommodate the carry-out of the summation, as shown in Figure 13-3:

```
parameter num_inputs = 3, input_width = 6;
assign sum_out = DW_sum({2{A[3]}, A, 2{B[3]}, B, 2{C[3]}, C });
```

Figure 13-3 Inferring Signed Vector Adder

VHDL Example (Unsigned)

In VHDL, the input vector to DW02_sum must be 3 × 6 = 18 bits (instead of 3 x 4 = 12 bits) to allow for extra bits in each operand, and to create a 6-bit output port to accommodate the carry bits of the summation.

The following VHDL code infers DW02_sum to sum three 4-bit unsigned numbers and pads the MSB of each input with "00":

```
in1 <= "00" & A & "00" & B & "00" & C;
sum_out <= DW_sum(in1, 3));
```

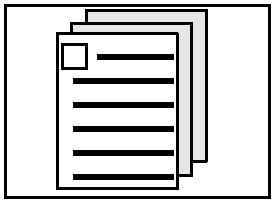
VHDL Example (Signed)

The following VHDL code infers DW02_sum to add three 4-bit signed numbers, and pads the MSB of each input with two extra bits:

```
in1 <= A(3) & A(3) & A & B(3) & B(3) & B & C(3) & C(3) & C;
sum_out <= DW_sum(in1, 3));
```

Summary

By making minor changes in your source code, you can take advantage of the carry-in and carry-out capabilities when inferring arithmetic DesignWare Building Block IP. The techniques illustrated in this article for DW01_add and DW02_sum can be applied to any DesignWare arithmeticIP that supports carry-in and carry-out.



Test Access Port and Boundary Scan IP

AN 96-004

The IEEE Standard Test Access Port and Boundary Scan Architecture (IEEE Standard 1149.1) defines a standard set of test logic that can be added to an integrated circuit (IC) to aid in testing the functionality and the board interconnections of the IC. The test logic defined by IEEE Standard 1149.1 supports two major modes of operation:

- Testing interconnections and communication with the other components of the system that contains the IC. In this mode, the test logic is independent of the IC's core logic.
- Testing of the IC's core logic. In this mode, the test logic is used with the IC's core logic, but separately from the surrounding system logic. The testing of the core logic is independent of the activities taking place at the IC's pins.

The DesignWare Building Block IEEE 1149.1 components provide a flexible set of technology-independent modules that you can use to add IEEE 1149.1 compliant test capabilities to your ASIC designs.

The DesignWare Building Block IP IEEE 1149.1 components, listed in Table 14-1, provide the following features:

- IEEE Std. 1149.1 compliant
- Synchronous or asynchronous scan chains with respect to `tck` (the IEEE 1149.1 Test Clock (TCK) signal)
- One-hot, glitch-free TAP controller
- Boundary scan cell building blocks for input, output, and bidirectional pins
- Parameterized instruction register width (2 to 32 bits)
- Support for the required IEEE 1149.1 instructions EXTEST, SAMPLE/PRELOAD, and BYPASS
- Support for the optional IEEE 1149.1 instructions INTEST, RUNBIST, CLAMP, and HIGHZ
- Optional use of the Device Identification Register and the IDCODE instruction
- User-defined instruction decoding

Table 14-1 Building Block IP Test Access Port and Boundary Scan Components

Component	Function
DW_tap	TAP Controller, instruction decode, and IEEE 1149.1 standard registers
DW_tap_uc	TAP Controller with user code support
DW_bc_1	Boundary scan cell for system input or output pins
DW_bc_2	Boundary scan cell for system input or output pins
DW_bc_3	Boundary scan cell for system input pins
DW_bc_4	Boundary scan cell for performance-sensitive input pins
DW_bc_5	Boundary scan cell for inputs that control the output enable for a three-state buffer
DW_bc_7	Boundary scan cell for bidirectional pins
DW_bc_8	Boundary scan cell Type BC_8
DW_bc_9	Boundary scan cell Type BC_9
DW_bc_10	Boundary scan cell Type BC_10

Starting Y-2006.06-SP1 DW_tap and DW_tap_uc are scan testable. An input signal "test" has been added to these components. For scannable designs, the test pin is held active (HIGH) during testing. For normal operation, it is held inactive (LOW).

Design Flow for Implementing Boundary Scan Components

After you have designed the core logic of your chip and defined the I/O pins, you are ready to add the DesignWare Building Block IP IEEE 1149.1 boundary scan logic. Figure 14-1 depicts the design flow for implementing Building Block IP boundary scan components. Figure 14-2 illustrates an example of Building Block IP boundary scan implementation.

Figure 14-1 Design Flow for Building Block IP Boundary Scan Components

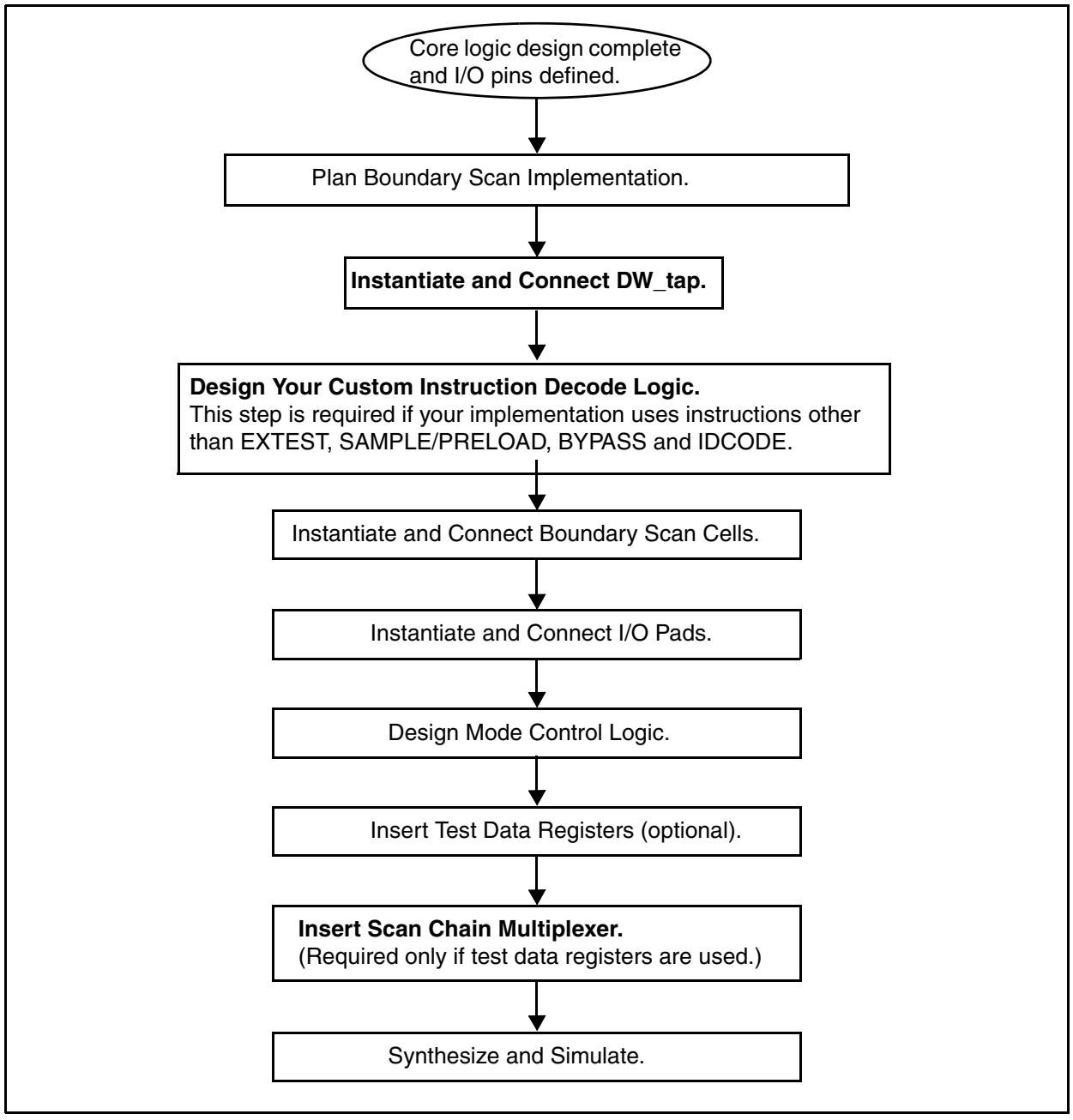
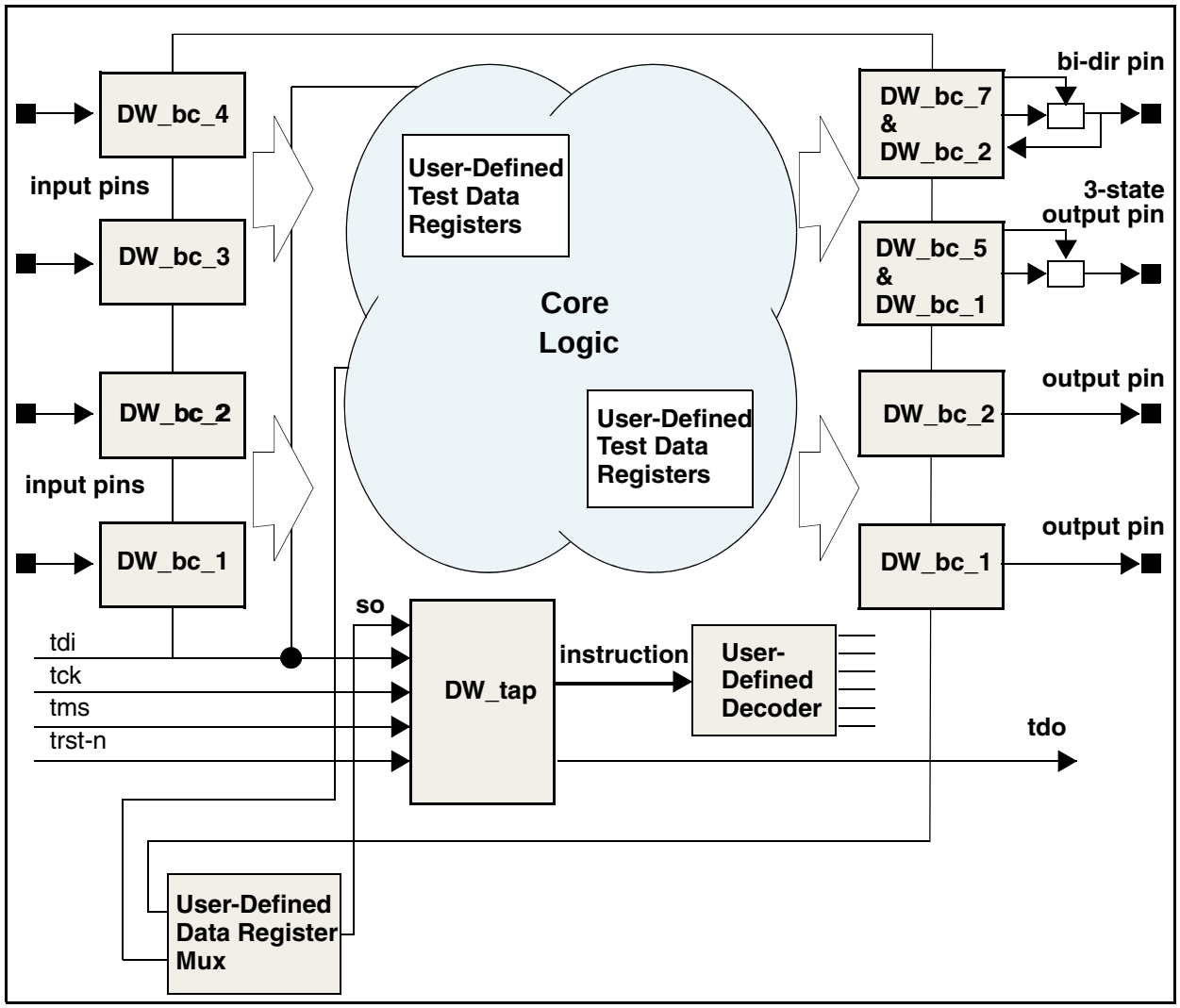


Figure 14-2 Example Test Access Port and Boundary Scan Implementation



Planning Your Implementation

The Building Block IP Test Access Port and Boundary Scan components provide the flexibility to implement a variety of boundary scan configurations. Before you begin your boundary scan implementation, you must make the following decisions:

Do you want to use the optional Device Identification Register?

If yes, then you must know the manufacturer number, part number, and version number that you want to program into your IC. Set the DW_tap parameters as follows:

`id = 1`
`man_num =` 11-bit manufacturer number
`part_num =` 16-bit part number
`version =` 4-bit version number

If no, set `id = 0`.

Which boundary instructions do you want to support in your design?

DW_tap decodes the instructions required by IEEE Standard 1149.1: EXTEST, BYPASS, and SAMPLE/PRELOAD, plus the optional instruction IDCODE. If you want to use the IEEE Standard 1149.1 optional instructions CLAMP, HIGHZ, INTEST, and RUNBIST, or any user-defined instructions, then you must build additional instruction decoding logic as described in the topic titled "Designing Instruction Decode Logic" on page 75.

Also, set the DW_tap width parameter to your desired instruction register width. The legal range is from 2 through 32 bits.

Do you want the boundary scan registers to be synchronous or asynchronous to tck?

If synchronous, set the DW_tap sync_mode parameter to 1, and connect the scan cells according to the synchronous configuration described in the datasheets.

If asynchronous, set the DW_tap sync_mode parameter to 0, and connect the scan cells according to the asynchronous configuration described in the datasheets.

Do you want to implement user-defined test data registers?

The IEEE Standard 1149.1 allows access to test-support features embedded in the design. For example, you may want to access scan-test, self-test, or other key registers in the design. If you choose to use test data registers, then you must define boundary scan instructions that serially connect the selected registers between tdi and tdo.

In addition, you must plan the routing order of your boundary scan chain, and determine which type of boundary scan cell you need for each I/O pin. The selection of cell type depends on whether the pin is input, output, or bidirectional, and which boundary scan instructions you want the cell to support. Table 14-2 lists the Building Block IP boundary scan cells, the IEEE 1149.1 instructions that each cell supports, and the instructions that control IC inputs and outputs through the cell.

Table 14-2 Instructions Supported by Building Block IP Boundary Scan Cells

Cell	Input	Output
DW_bc_1	EXTEST, SAMPLE/PRELOAD, BYPASS, INTEST, RUNBIST, CLAMP, and HIGHZ. INTEST and EXTEST control input to core logic.	EXTEST, SAMPLE/PRELOAD, BYPASS, INTEST, RUNBIST, CLAMP, and HIGHZ. INTEST and EXTEST control IC output pins.
DW_bc_2	EXTEST, SAMPLE/PRELOAD, BYPASS, INTEST, RUNBIST, CLAMP, and HIGHZ. INTEST controls input to core logic.	EXTEST, SAMPLE/PRELOAD, BYPASS, RUNBIST, CLAMP, and HIGHZ. EXTEST controls IC output pins.
DW_bc_3	EXTEST, SAMPLE/PRELOAD, BYPASS, INTEST, RUNBIST, CLAMP, and HIGHZ. INTEST controls input to core logic.	Not used for output pins
DW_bc_4	EXTEST, SAMPLE/PRELOAD, and BYPASS. This cell does not control input to core logic.	Not used for output pins
DW_bc_5	EXTEST, SAMPLE/PRELOAD, BYPASS, INTEST, RUNBIST, CLAMP, and HIGHZ	EXTEST, SAMPLE/PRELOAD, BYPASS, INTEST, RUNBIST, CLAMP, and HIGHZ
DW_bc_7	EXTEST, SAMPLE/PRELOAD, BYPASS, INTEST, RUNBIST, CLAMP, and HIGHZ	EXTEST, SAMPLE/PRELOAD, BYPASS, INTEST, RUNBIST, CLAMP, and HIGHZ
DW_bc_8	EXTEST, SAMPLE/PRELOAD, BYPASS, INTEST, RUNBIST, CLAMP, and HIGHZ	EXTEST, SAMPLE/PRELOAD, BYPASS, INTEST, RUNBIST, CLAMP, and HIGHZ
DW_bc_9	EXTEST, SAMPLE/PRELOAD, BYPASS, INTEST, RUNBIST, CLAMP, and HIGHZ	EXTEST, SAMPLE/PRELOAD, BYPASS, INTEST, RUNBIST, CLAMP, and HIGHZ
DW_bc_10	EXTEST, SAMPLE/PRELOAD, BYPASS, INTEST, RUNBIST, CLAMP, and HIGHZ	EXTEST, SAMPLE/PRELOAD, BYPASS, INTEST, RUNBIST, CLAMP, and HIGHZ

Implementing DW_tap

The DW_tap module consists of the Test Access Port (TAP) controller, instruction register, instruction decode logic, bypass register, and an optional device identification register. The TAP controller contains the IEEE Standard 1149.1 finite state machine.

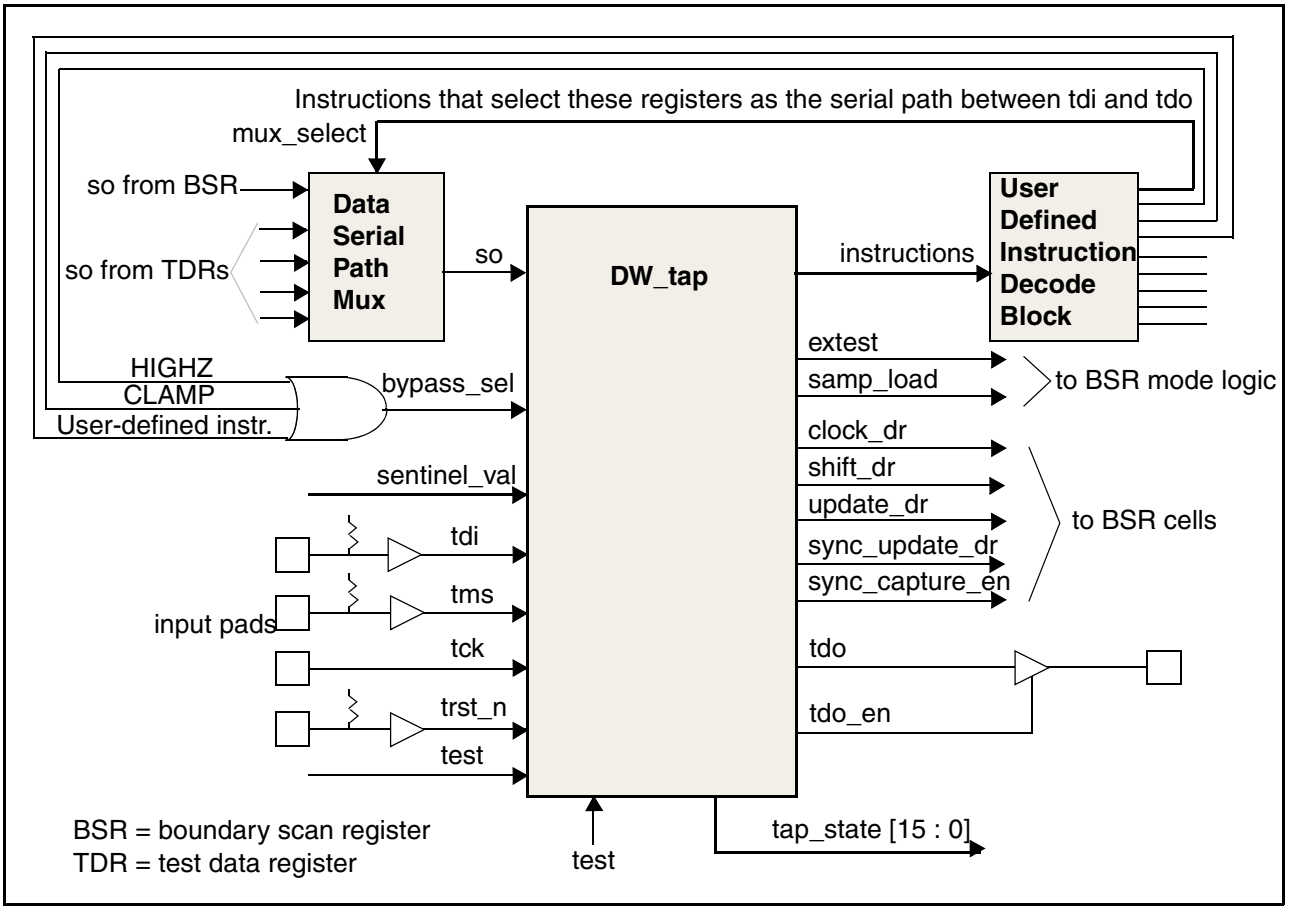
DW_tap Parameters

Set the DW_tap parameters according to your answers to the questions in the topic titled "Planning Your Implementation" on page 70.

DW_tap Connections

Figure 14-3 illustrates how to connect DW_tap to your IC package pins and other boundary scan logic. The following sections describe the DW_tap connections in more detail.

Figure 14-3 DW_tap Connections



TAP Controller Connections

The IEEE Standard 1149.1 specifies four package pins (`TDI`, `TDO`, `TCK`, and `TMS`), plus an optional reset pin (`TRST`), that are dedicated to the TAP. Connect the `TDI`, `TMS` and `TRST` pins, each through an input pad with pull-up, to the DW_tap ports `tdi`, `tms`, and `trst_n`. Connect the `TCK` pin, through an input pad, to the DW_tap `tck` port. Connect the DW_tap `tdo` port to a three-state output pad, with the DW_tap `tdo_en` signal as the enable.

The `tdi` and `tdo` pins are used to serially shift data into and out of the IC. The `tck` and `tms` pins provide the test clock and test mode select functions. The `trst_n` pin is used to asynchronously reset the TAP controller. If the TRST pin is not used, you must design power-on reset logic that drives the `trst_n` signal.

In addition, DW_tap provides the state of the one-hot TAP state machine on the `tap_state` bus that you can connect to your user-defined add-on logic.

Instruction Register Connections

The instruction register is used to serially shift in the instructions that operate the boundary-scan circuitry. The `width` parameter controls the width of the instruction register. The instruction register must be at least two bits wide to hold the instruction codes for the three instructions that are required by IEEE Standard 1149.1: BYPASS, EXTEST, and SAMPLE/PRELOAD. The maximum instruction register width is 32 bits.

The DW_tap component decodes the required instructions BYPASS, EXTEST, SAMPLE/PRELOAD, and the optional IDCODE instruction, as listed in Table 14-3. All decoding of additional instructions must be performed by the user-defined decode logic that is connected to the DW_tap `instructions` bus. If the Device Identification register is not used, the IDCODE decoded instruction value (000...001) may be used to decode another instruction.

Table 14-3 Decoded Instructions

Instruction	Decoded Value
BYPASS	All ones (as specified by IEEE Standard 1149.1)
EXTEST	All zeros (as specified by IEEE Standard 1149.1)
IDCODE	000...001
SAMPLE/PRELOAD	000...010
INTEST	User-defined
RUNBIST	User-defined
CLAMP	User-defined
HIGHZ	User-defined

The decoded BYPASS and IDCODE instructions are implemented within DW_tap. The decoded EXTEST and SAMPLE/PRELOAD instructions activate the DW_tap `extest` and `samp_load` outputs, respectively. Connect the `extest` and `samp_load` outputs to your user-defined mode control logic.

While shifting in a new instruction, the value of the first two bits shifted out on `tdo` are 1 and 0, as specified by IEEE Standard 1149.1. The rest of the bits shifted out are determined by the `sentinel_val` input. You can connect the `sentinel_val` input bus to any signal you want to view when shifting instructions into the instruction register. The `sentinel_val` bus provides extra status bits in the IC design. You must connect the `sentinel_val` bus to logic zero if you do not want to used this feature.

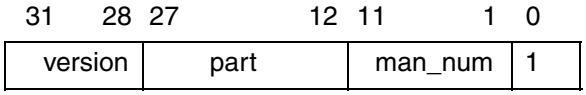
Bypass Register Connections

The bypass register is a mandatory IEEE Standard 1149.1 1-bit register that provides a minimum length serial path through the IC. The BYPASS instruction selects the bypass register as the serial connection between `tdi` and `tdo`. Other decoded instructions can select the bypass register through the `bypass_sel` input signal. If the HIGHZ and/or the CLAMP instructions are implemented in the design, they must select the bypass register as the serial path. The decoded signal for these instructions must be connected to the DW_tap `bypass_sel` port.

Device Identification Register

The device identification register is an optional IEEE Standard 1149.1 register. If the parameter `id` is set to 1, synthesis builds the 32-bit device identification register. If `id` is set to 0, the register is not included in the design. The parameters `part`, `version`, and `man_num` compose the 32-bit identification code to be loaded into the register during the IDCODE instruction. You must set the parameter `part` to the integer value of the 16-bit part number, the parameter `version` to the integer value of the 4-bit version number of the part, and the parameter `man_num` to

the integer value of the 11-bit JEDEC manufacturer ID number. The structure of the instruction register is as follows:



There are no external connections to the device identification register.

Synchronous Mode

The `sync_mode` parameter determines whether access to the bypass register, device identification register, and instruction register is synchronous or asynchronous. If `sync_mode` is set to 1, the input data to these registers is captured synchronous to `tck`. Boundary scan cells are updated on the falling edge of tck.

If `sync_mode` is set to 0, then the rising edge of `clock_dr` clocks input data into the bypass register and device identification register. The rising edge of `clock_ir` (internal to DW_tap) clocks input data into the instruction register.

Designing Instruction Decode Logic

As previously stated, DW_tap decodes the IEEE 1149.1 mandatory instructions EXTEST, SAMPLE/PRELOAD, and BYPASS, and the optional instruction IDCODE. If you want to use additional instructions, you must design your own custom instruction decode logic.

Connect the instruction decoder to the DW_tap `instructions` bus, as illustrated in Figure 14-3 on page 73. The instruction decoder should provide the following output signals:

- Signals needed to drive the mode logic that controls the boundary scan cells.
- Multiplexer control signals for instructions that connect user-defined test data registers between `tdi` and `tdo`.
- A bypass signal for instructions that use the bypass register. Connect this signal to the DW_tap `bypass_sel` input. For example. the decoded CLAMP and HIGHZ instructions must activate the `bypass_sel` signal.

Implementing the Boundary Scan Cells

The boundary scan cells placed at the pins of the IC are interconnected to form a shift-register chain around the border of the design. Test data is applied to the IC, and test results are extracted serially through the boundary scan cells.

Each cell includes serial input and output connections and pins for the control signals from the TAP controller and `mode` generation logic. Building Block IP provides six types of boundary-scan cells. Design requirements for both connectivity and functional operation vary from cell to cell, and are determined by the type of signal the cell is connected to and by the set of instructions to be supported.

Connections Common to All Types of Boundary Scan Cells

The following connections are common to all six types of Building Block IP boundary scan cells:

shift_dr

Every boundary scan cell is controlled by the DW_tap signal `shift_dr`. The DW_tap signal `shift_dr` must be connected to the `shift_dr` port of every boundary scan cell to enable proper shifting of data through the boundary scan chain (in both synchronous and asynchronous mode).

si and so

Every boundary scan cell also has an `si` (serial in) port and an `so` (serial out) port. These ports link the boundary scan cell into the shift register scan chain. Connect the `tdi` signal to the `si` input of the first boundary scan cell of your boundary scan chain. Connect the `so` output of the last boundary scan cell in the chain to either the DW_tap `so` input or, if user-defined test data registers are used, to the input of the serial path multiplexer. For all other cells in the boundary scan chain, connect the `si` pin to the `so` pin of the previous boundary scan cell, and connect the `so` pin to the `si` pin of the next boundary scan cell.

capture_en/capture_clk and update_en/update_clk

The connection of the `capture_en` and `capture_clk` pins, and, if present, the `update_en` and `update_clk` pins depends on whether you are implementing a synchronous or an asynchronous boundary scan design.

In synchronous designs, the goal is to synchronize the capturing of data into the cell with the `tck` signal, as illustrated in Figure 14-4. For synchronous designs, connect the `capture_en` and `capture_clk` pins, and, if present, the `update_en` and `update_clk` pins as listed in Table 14-4.

Figure 14-4 Timing Diagram of a Synchronous Boundary Scan Cell

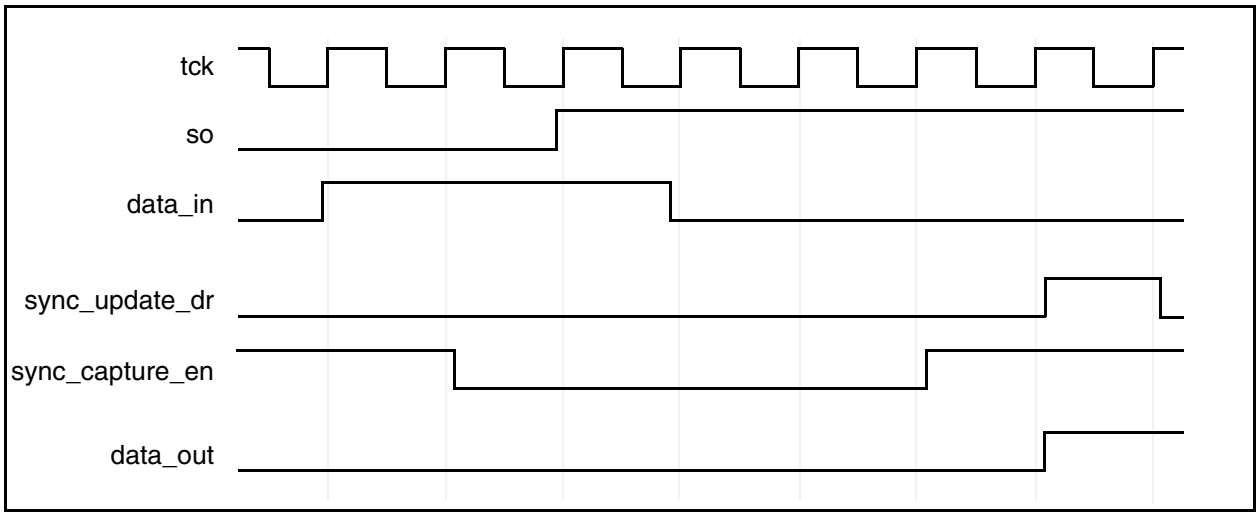


Table 14-4 Connections for Synchronous Boundary Scan Cells

Boundary Scan Cell Pin Name	Connection
capture_clk	tck from system pin
update_clk	tck_n from system pin
capture_en	sync_capture_en from DW_tap
update_en	sync_update_dr from DW_tap

In asynchronous designs, the goal is to capture data into the cell on the rising edge of `clock_dr`, asynchronous to `tck`, as illustrated in Figure 14-5. For asynchronous designs, connect the `capture_en` and `capture_clk` pins, and, if present, the `update_en` and `update_clk` pins as listed in Table 14-5. The synthesis tool optimizes the design, and removes the unnecessary synchronization logic.

Figure 14-5 Timing Diagram of an Asynchronous Boundary Scan Cell

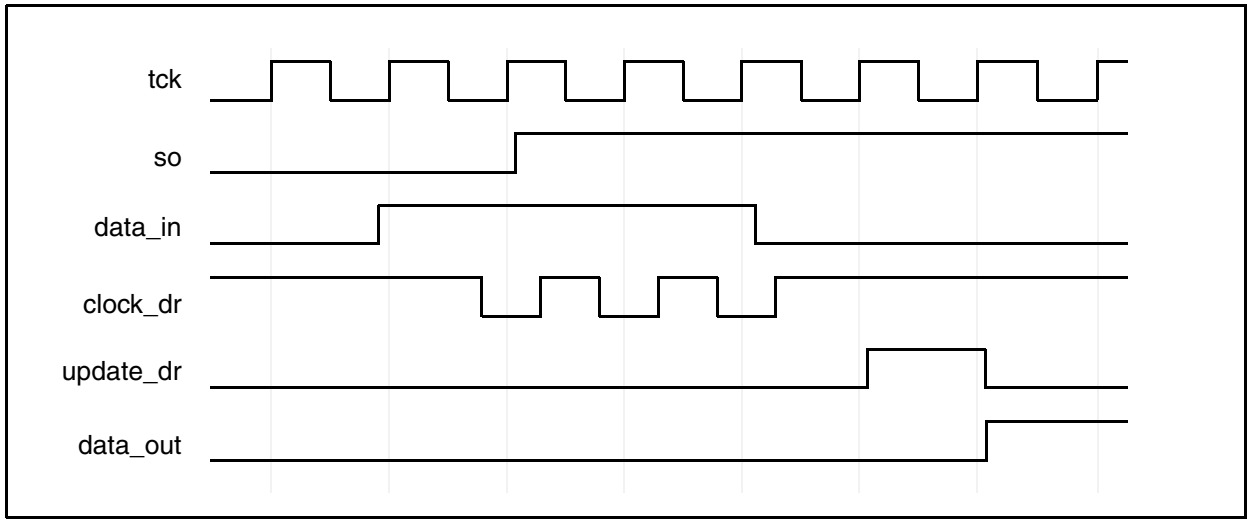


Table 14-5 Connections for Asynchronous Boundary Scan Cells

Boundary Scan Cell Pin Name	Connection
capture_clk	capture_dr from DW_tap
update_clk	update_dr from DW_tap
capture_en	Logic zero
update_en	Logic one

You can also use the DW_tap `clock_dr` signal to control the user-defined test data registers. Although one register is enabled at a time, the other registers are still clocked by the `clock_dr` signal. If you do not want the boundary scan register to be clocked when the user-defined test data register instruction is active, then gate the `clock_dr` signal with the appropriate decoded instruction.

If the CLAMP and HIGHZ instructions are implemented, then the `clock_dr` signal must be gated with the decoded CLAMP and HIGHZ instructions to disable shifting data through the boundary scan chain while the bypass register is active.

Input and Output Boundary Scan Cells

The boundary scan cells DW_bc_1 and DW_bc_2 can be used as either input or output boundary scan cells. DW_bc_3 is an input cell. DW_bc_4 is an observe-only cell that cannot drive signals into the IC logic. Use DW_bc_4 for high-performance input pins, such as clock pins.

DW_bc_1, DW_bc_2, and DW_bc_3 differ in their functionality during the instructions INTEST and EXTEST. When used as an input cell, DW_bc_1 enables you to control the inputs to the core logic during INTEST and EXTEST instructions. When used as an output cell, DW_bc_1 enables you to control the chip outputs during INTEST and EXTEST instructions.

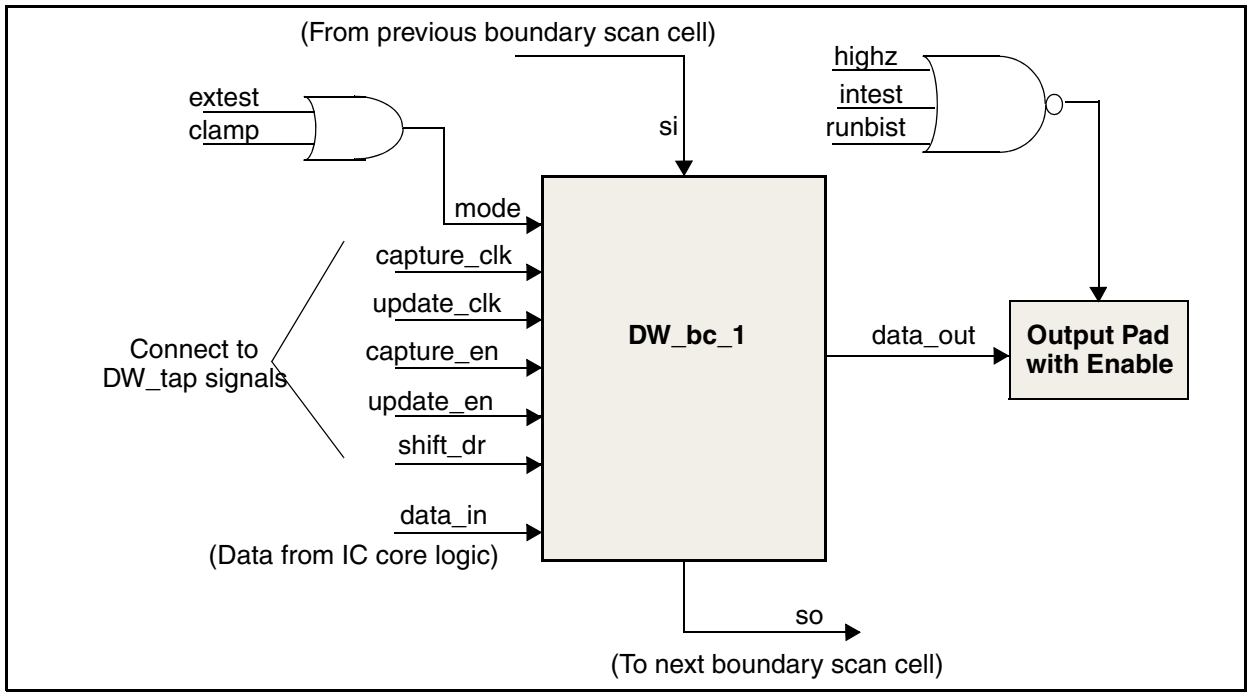
When used as an input cell, DW_bc_2 enables you to control the inputs to the core logic only during INTEST instructions. When used as an output cell, DW_bc_2 enables you to control the chip outputs only during EXTEST instructions. Therefore, you should not use DW_bc_2 as an output cell if your design supports the INTEST instruction.

DW_bc_3 does not enable you to control the chip outputs, because it does not contain an update storage element. Use DW_bc_3 for input cells only. When used as an input cell, DW_bc_3 drives shifted values into the core logic during INTEST instructions.

The generation of the mode signal for DW_bc_1, DW_bc_2, and DW_bc_3 depends on which instructions are supported and how the instructions are implemented. The `mode` generation table in each datasheet provides the required value of the `mode` signal for each instruction.

The IEEE Standard 1149.1 provides two ways to implement an output cell for the RUNBIST and INTEST instructions. During these instructions, the output pins can be driven from pre-loaded data held in the boundary scan register or placed in an inactive drive state. The RUNBIST and INTEST instructions must be added to the mode generation logic if they are to be used to drive output pins with data stored in the boundary scan cell. Otherwise, the instructions should be added to the output enable logic to force every output pin to an inactive drive state, as shown in Figure 14-6.

Figure 14-6 Example Connections for an Output Boundary Scan Cell



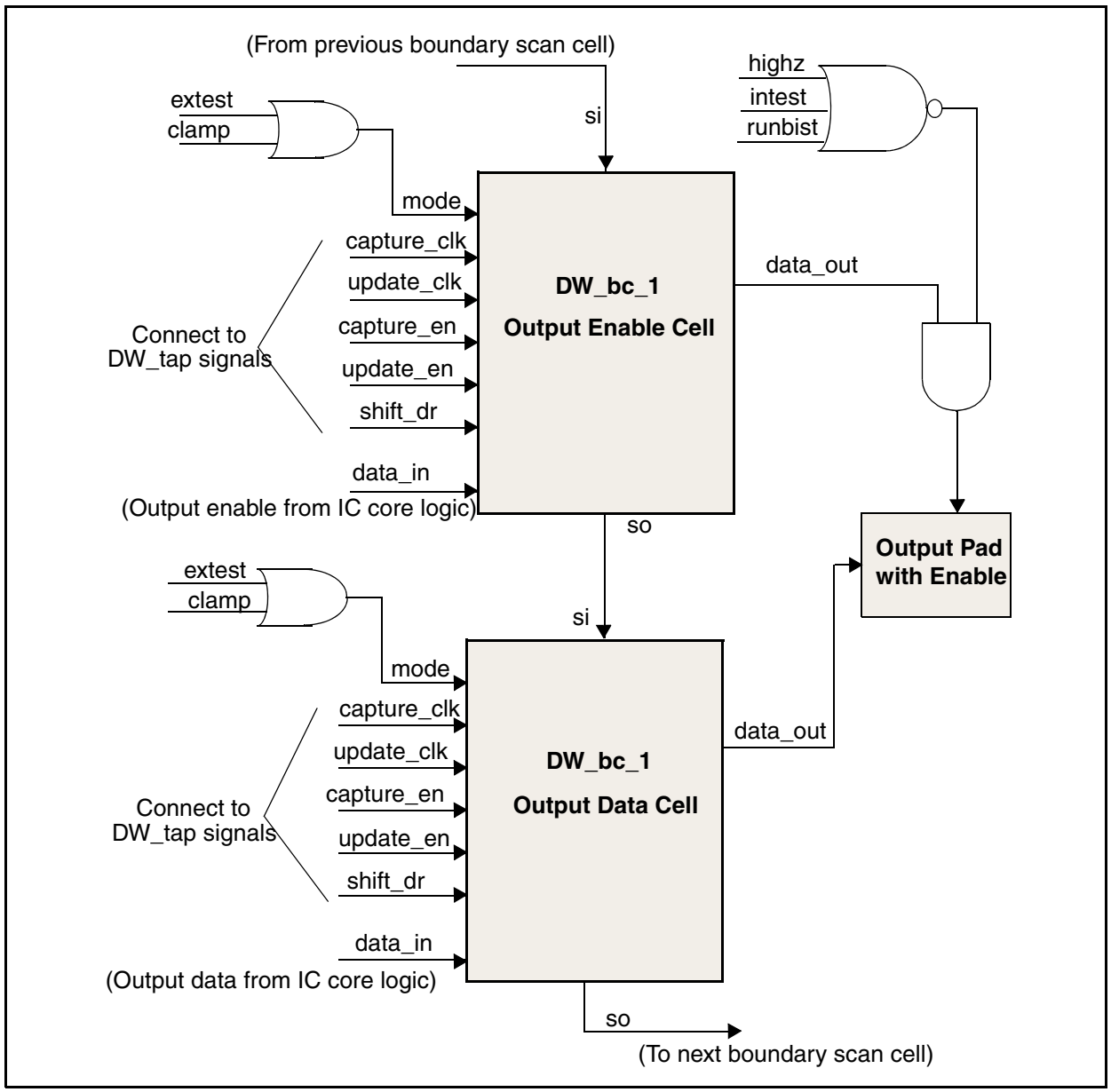
Three-State Output Boundary-Scan Cells

Building Block IP provides the building blocks for three different implementations of a three-state output cell. Two boundary-scan cells are required for a three-state output pin, one to control the output enable and one for the output data. Where many three-state output pins are controlled from a single source, for example, an address bus, only one boundary scan cell is required for the output enable.

One possible implementation of a three-state cell is to use two DW_bc_1 cells; one for the output enable, and one for the output data. The `mode` signal is the same for both cells. For the HIGHZ instruction, the `data_out` from the output enable cell must be logically AND'd with the inverse of the HIGHZ decoded instruction signal.

If you want to implement the INTEST and RUNBIST instruction to force the output in an inactive drive state, then the INTEST and RUNBIST decoded signals should be logically NOR'd with the HIGHZ and AND'd with the output enable signal, as shown in Figure 14-7.

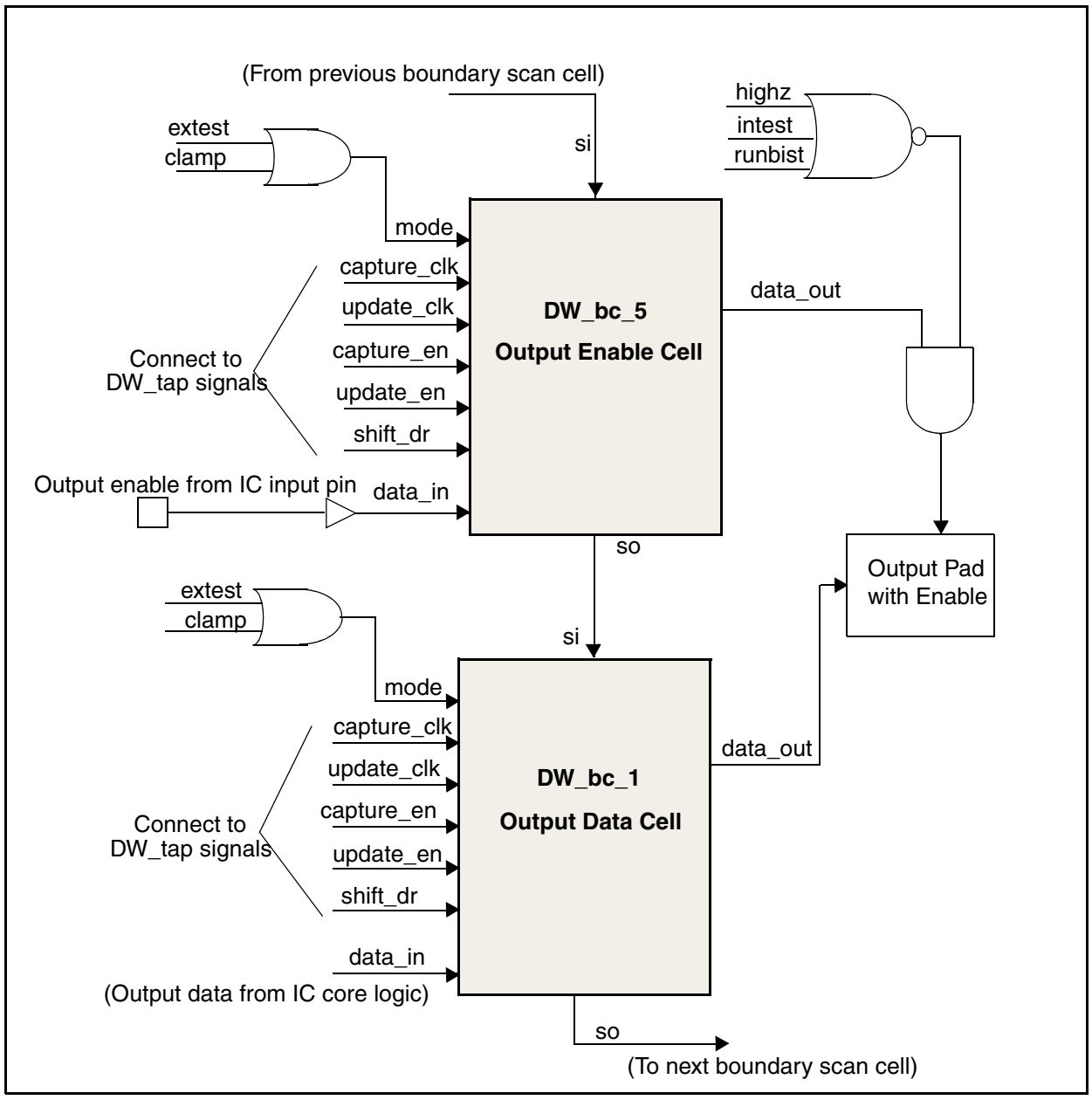
Figure 14-7 Three-State Output Cell Implemented with Two DW_bc_1 Cells



A second implementation of a three-state cell is to use two DW_bc_2 cells, one for the output enable and one for the output data. The connections are similar to the two DW_bc_1 cells implementation.

The third implementation of a three-state cell is used where a signal received at an IC input pin is used solely to provide the enable to output data. DW_bc_5 can be used for the enable cell, instead of using an input cell at the input pad and an output cell to drive the enable. Figure 14-8 illustrates how to connect a three-state boundary scan cell using DW_bc_5 for the enable, and DW_bc_1 as the output cell.

Figure 14-8 Three-State Output Cell Implemented with DW_bc_1 and DW_bc_5 for an input pin that is used solely as the enable)

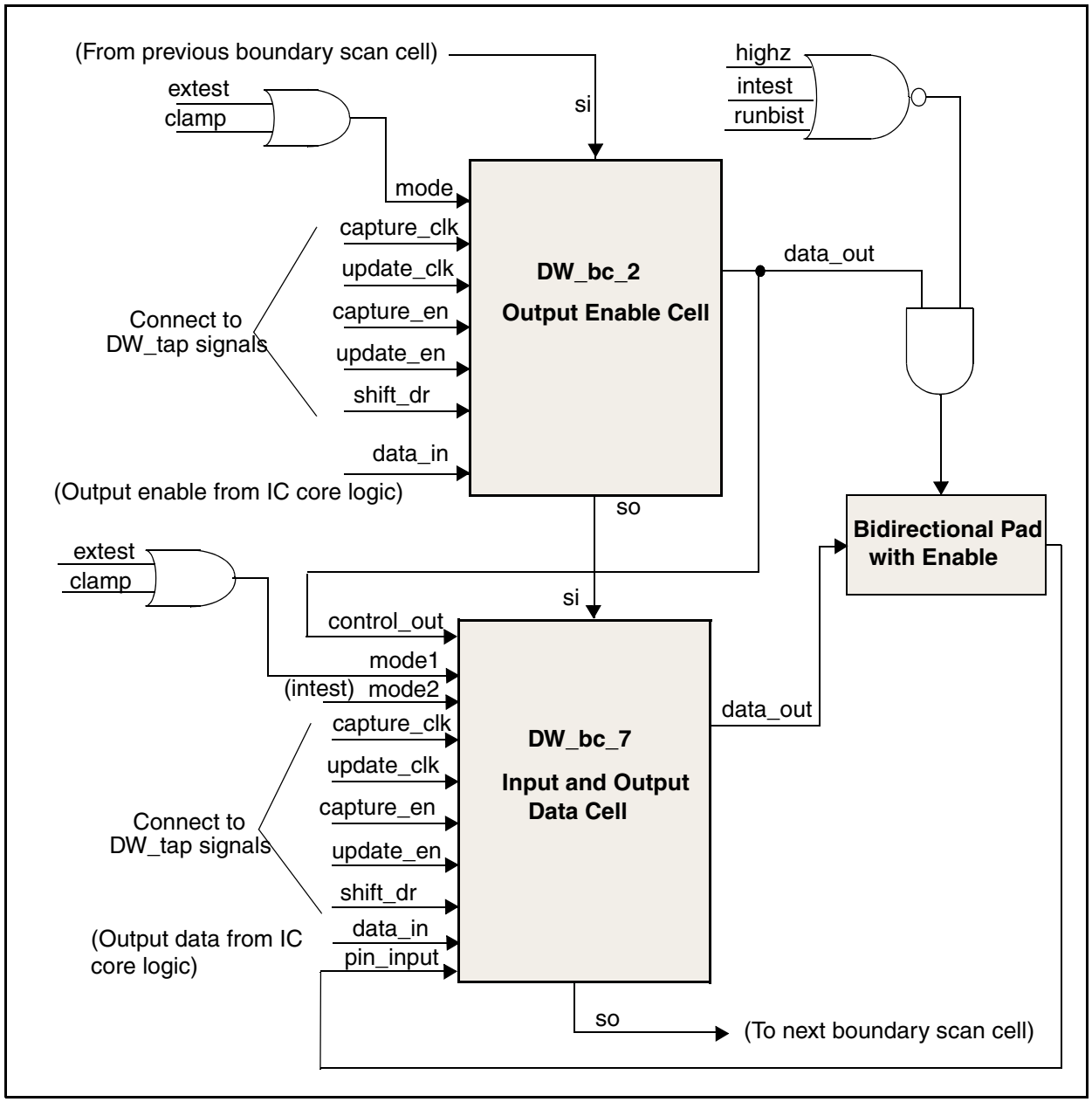


Bidirectional Boundary-Scan Cells

Building Block IP provides the building blocks to implement a bidirectional boundary scan cell. One way to implement a bidirectional cell is to use a three-state output cell to control the output data and output enable,

and use an input cell for the input. Another implementation, illustrated in Figure 14-9, is to use the DW_bc_7 cell to control the input and output, and use the DW_bc_2 cell to control the output enable.

Figure 14-9 Bidirectional Boundary Scan Cell Implementation



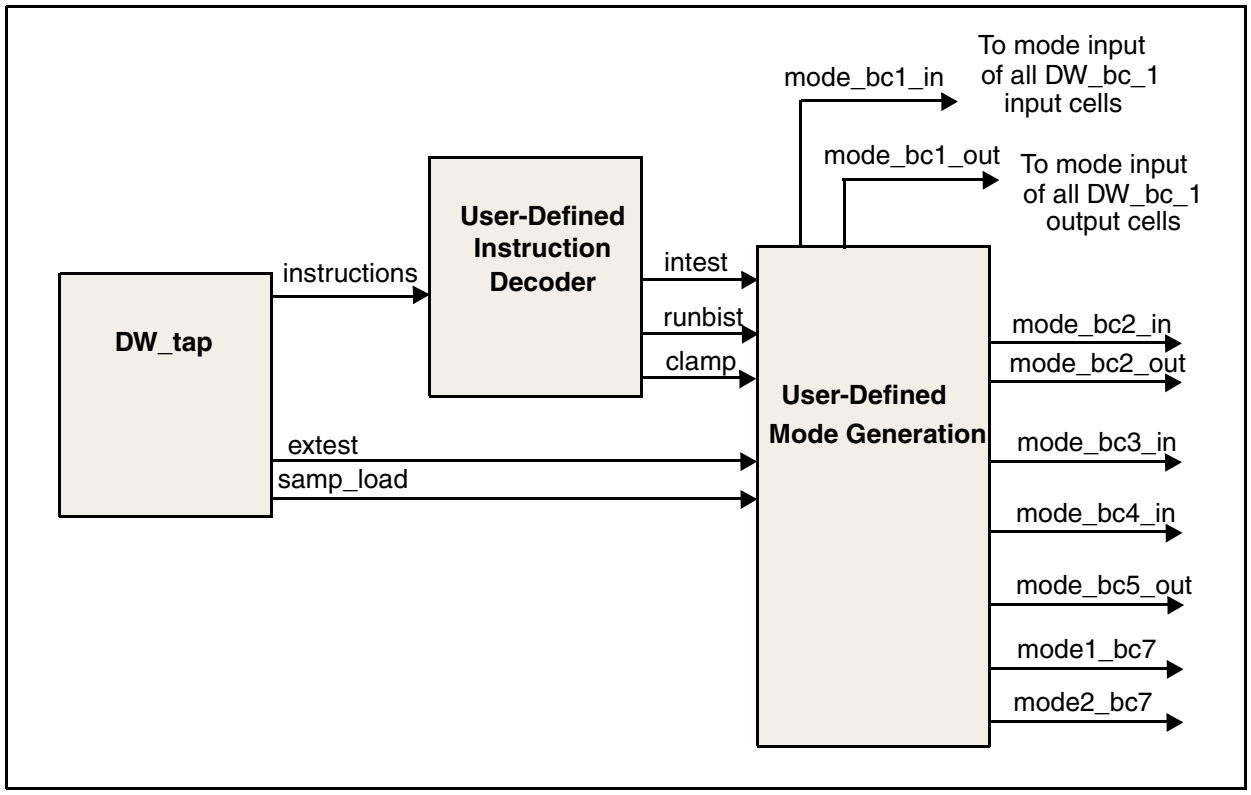
Implementing I/O Pads

Instantiate your technology-specific I/O pads. Connect the I/O pads to the boundary scan cells, and connect the `tdi`, `tdo`, `tms`, `tck`, and `trst_n` signals to the I/O pads.

Designing Mode Control Logic

The datasheet for each Building Block IP boundary scan cell lists the required values of the mode input for of the supported instructions. Because the different cells require different values on mode for the same instructions, your mode control logic must provide individual mode control signals for each type of Building Block IP boundary scan cell used in your design. Figure 14-10 shows an example of how to connect your mode control logic block in a design that uses all six types of Components boundary scan cells and all of the supported instructions.

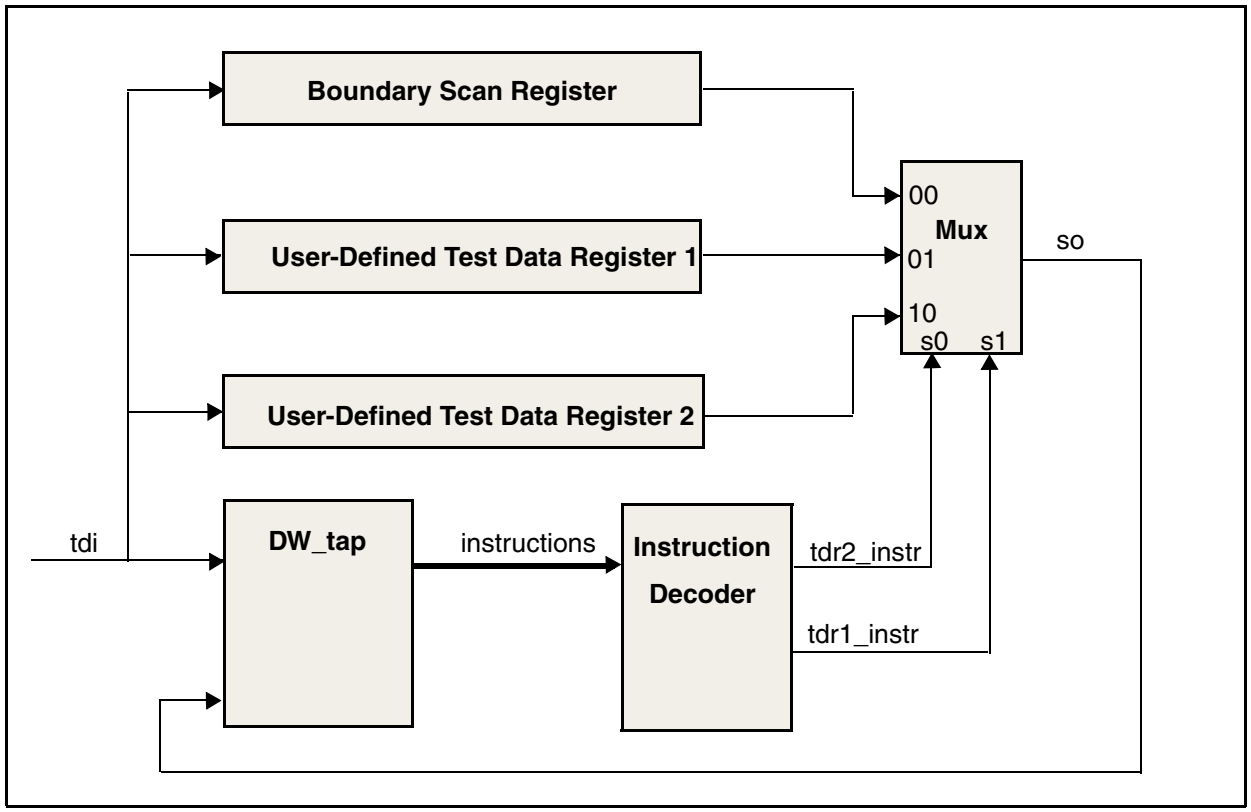
Figure 14-10 Mode Control Logic Connections



Inserting Test Data Registers

The IEEE Standard 1149.1 allows access to any test-support features embedded in the design. These features might include scan-test, self-test registers, or other key registers in the design. An instruction must be defined for each test data register that serially connects the selected register between `tdi` and `tdo`. To achieve this serial path connection, you must define a multiplexer that selects the boundary-scan register (BSR) serial path or any of the user-defined test data registers (TDRs) during the appropriate instruction, as illustrated in Figure 14-11.

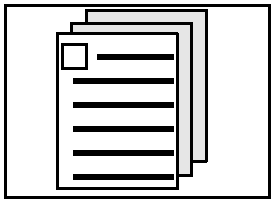
Figure 14-11 Multiplexing Test Data Registers



References

Parker, Kenneth P., *The Boundary-Scan Handbook*, Kluwer Academic Publishers, 1992.

IEEE Standard Test Access Port and Boundary-Scan Architecture, IEEE Std 1149.1a-1993.



DW_debugger Applications

AN 96-003

The DesignWare Building Block IP On-chip ASCII Debugger (DW_debugger) enables you to control up to 2048 nets and monitor up to 2048 nets inside your manufactured chip. Through two pins on your chip you can read and write to selected internal registers, counters, RAM, and so on. This application note illustrates how to use DW_debugger for the following example applications:

- Embedded microprocessor debugging
- Accessing embedded memory
- State machine monitoring and debugging
- Error rate monitoring
- Error injection and detection
- Data probing
- Performance monitoring

Embedded Microprocessor Debugging

Embedding a microcontroller or microprocessor within a chip prevents the use of traditional in-circuit emulation (ICE) techniques for debugging hardware, software, or embedded firmware. However, you can use DW_debugger to execute several hardware/software debugging operations on an embedded microcontroller, such setting breakpoints, forcing interrupts, and accessing internal registers and counters.

For example, Figure 15-1 illustrates an ASIC design that includes an embedded microcontroller, internal RAM, other logic, and an instance of DW_debugger. From the terminal, an operator can read selected nets through the `rd_bits` bus and write to selected nets through the `wr_bits` bus.

Figure 15-1 Example DW_debugger Application

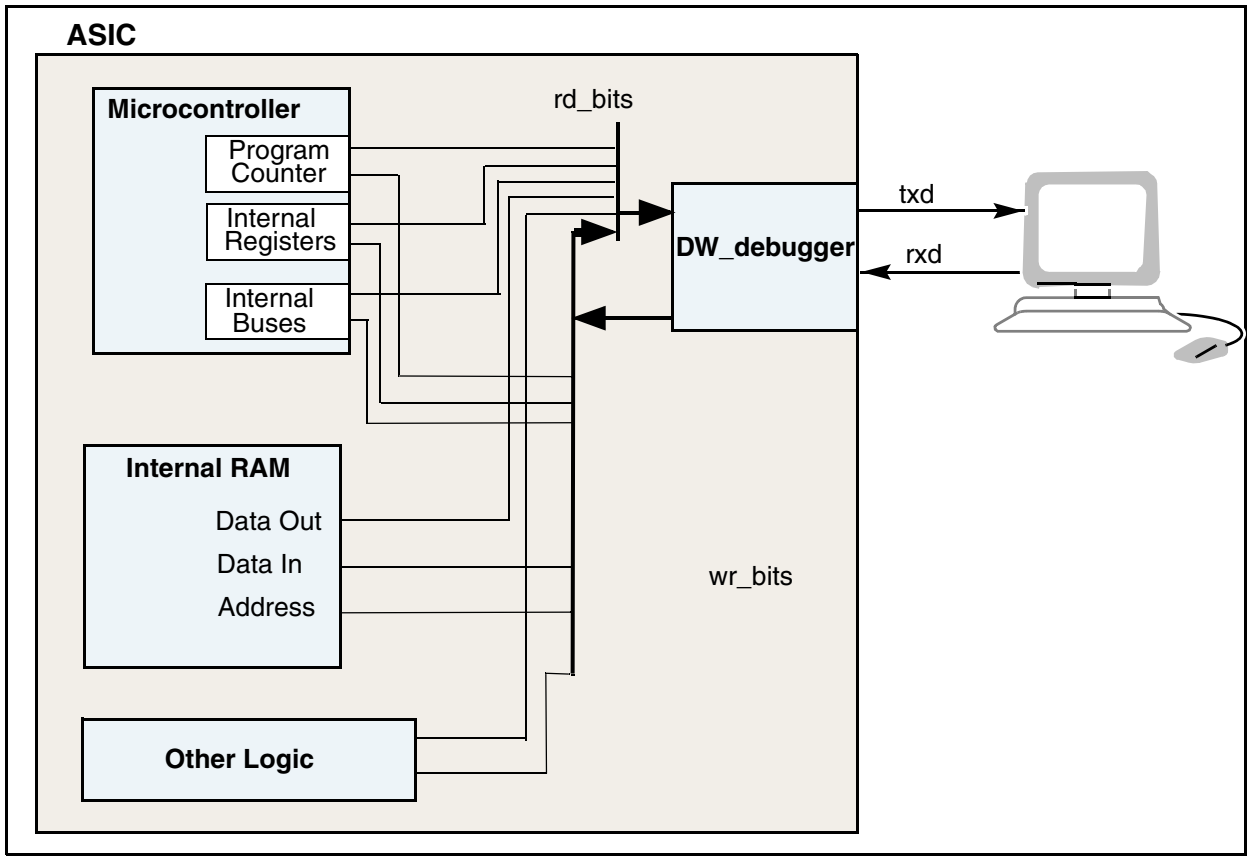
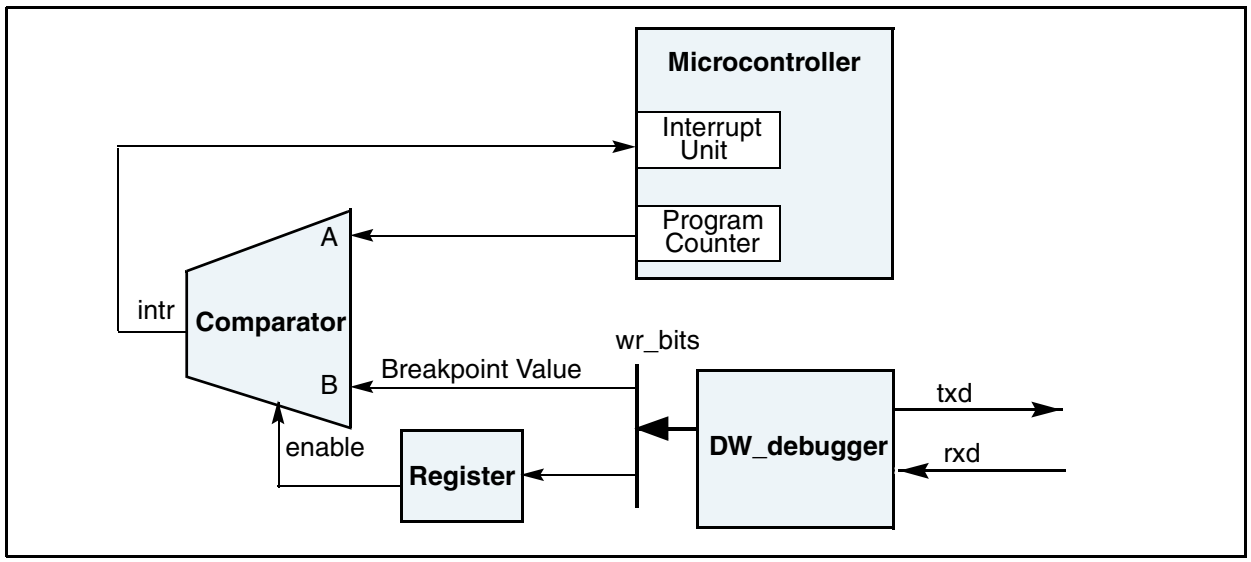


Figure 15-2 Setting a Program Breakpoint



Setting Breakpoints

Figure 15-2 illustrates a technique you can use to set breakpoints for an embedded microcontroller. The comparator compares the current value in the program counter with the breakpoint value from the `wr_bits`

bus. When the values are equal and the enable bit is set, the comparator sends an interrupt to the microcontroller.

This method of setting breakpoints works best if the microcontroller has an unused interrupt that you can use specifically for debugging purposes. This enables you to execute a special interrupt routine when the program counter reaches the breakpoint address.

To write to the wr_bits bus, use the DW_debugger p command. For example, if the A input of the comparator in Figure 15-2 is connected to the microcontroller's 16-bit program counter, and the B input is connected to bytes 3 and 4 of the DW_debugger wr_bits bus, then you can set breakpoint address 1FA3 by entering the following commands at the terminal:

```
% p 3 1F
  loc 3 <= 1F
% p 4 A3
  loc 4 <= A3
```

If you connect the wr_bits bus to the low bits of the rd_bits bus, DW_debugger acknowledges each p command by displaying the updated wr_bits value.

To enable the comparator, set the enable bit. For example, if the enable bit register in Figure 2 is connected to the LSB of wr_bits byte 5, enter the command:

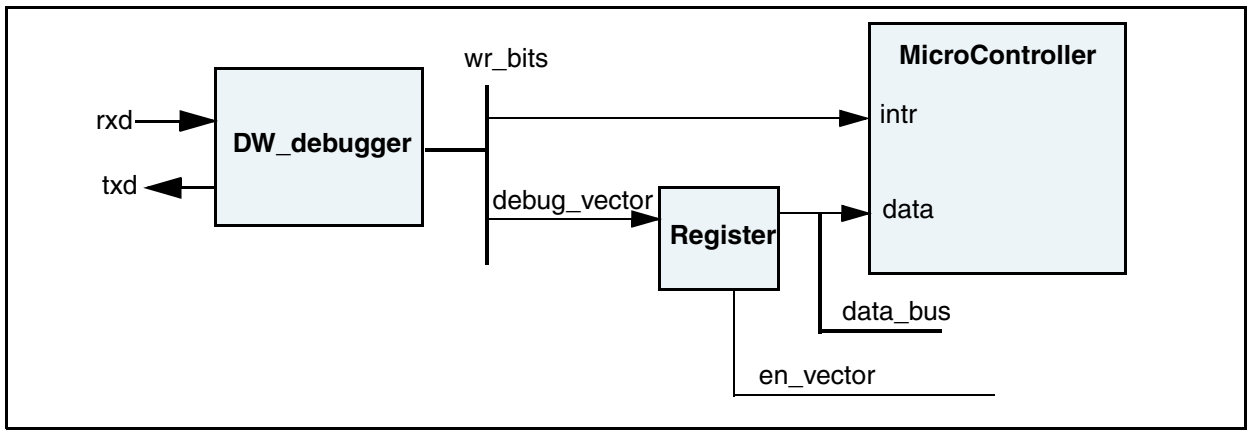
```
% p 5 01
  loc 5 <= 01
```

When you write to the wr_bits bus with the p command, you must always write a full byte value.

Forcing an Interrupt

Figure 15-3 illustrates how to use DW_debugger to force an interrupt to an embedded microcontroller and load a debugging interrupt vector (debug_vector) onto the microcontroller data bus. Load the interrupt vector into the register by writing to the corresponding wr_bits locations, then set the wr_bits bit that is connected to the microcontroller's intr input.

Figure 15-3 Forcing an Interrupt



Accessing Embedded RAM

Figure 15-4 illustrates a technique you can use to read from and write to an embedded RAM through DW_debugger. To read the contents of a RAM address, use the wr_bits bus to write the address value into

the address register. Then, set the rd_enb, rd_sel, and addr_sel bits, and read the Dout value through the rd_bits bus.

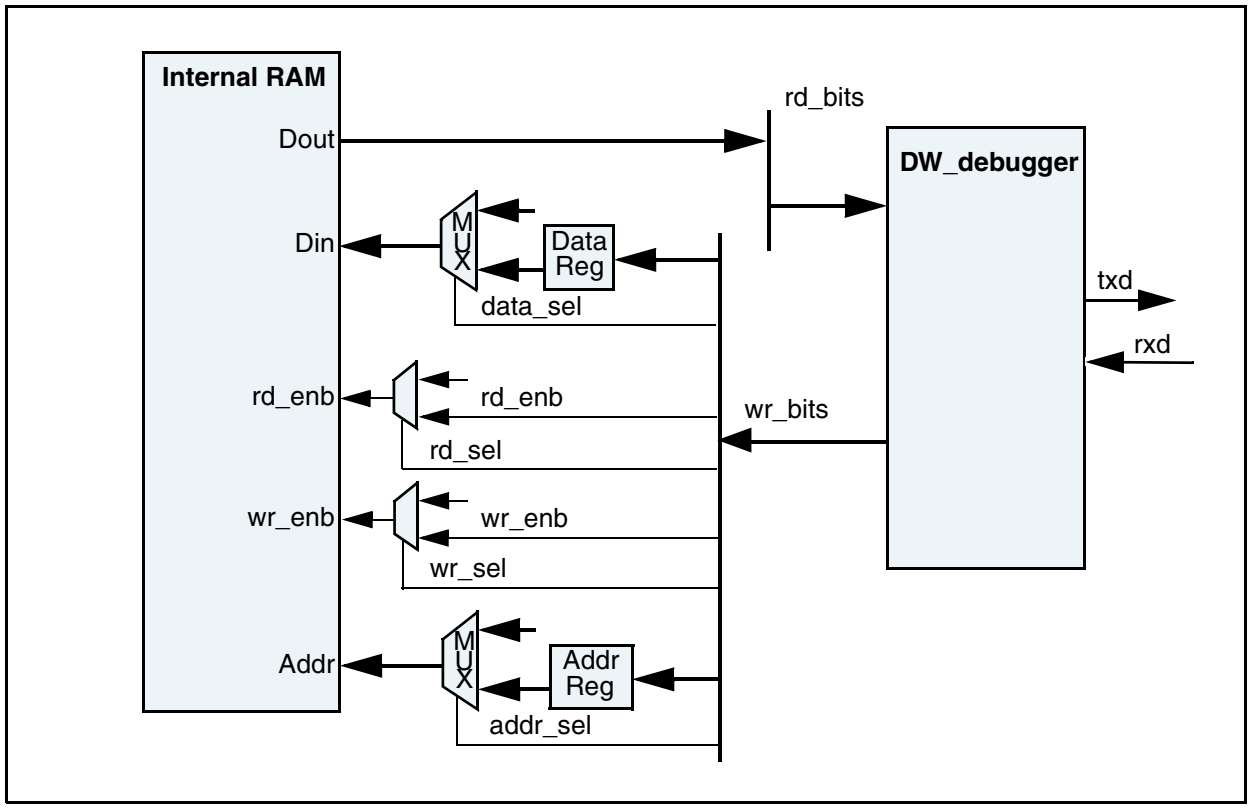
To read a value on the rd_bits bus, use the DW_debugger d command. For example, if Dout is connected to rd_bits bytes 13 and 14, enter the commands:

```
% d 13
  loc 13 <= FF
% d 14
  loc 4 <= FA
```

DW_debugger responds to each d command by displaying the selected rd_bits value.

To write to a RAM location, use the wr_bits bus to write the data into the data register. Then, write the address into the address register and set the wr_enb, wr_sel, data_sel, and addr_sel bits to write the data into the RAM.

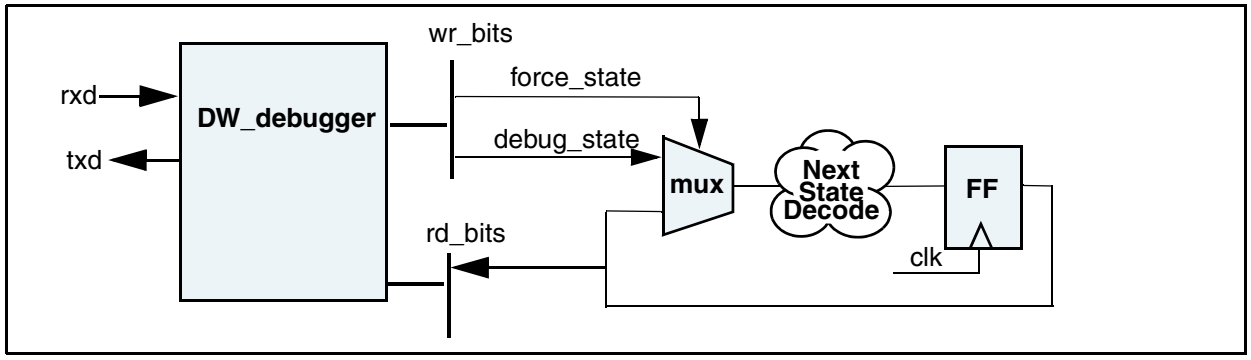
Figure 15-4 Reading and Writing RAM Contents



State Machine Debugging

Figure 15-5 illustrates how to use DW_debugger to debug a state machine. Use the wr_bits to force the state machine into a selected state (debug_state), and read the current state through the DW_debugger rd_bits.

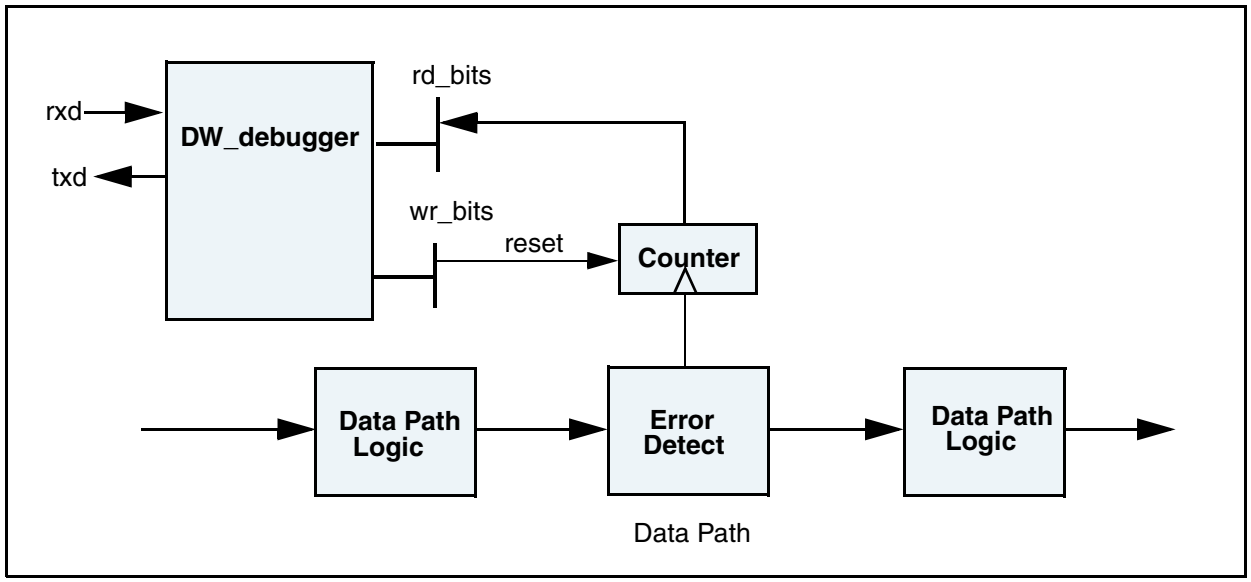
Figure 15-5 State Machine Monitor and Debug



Error Rate Monitoring

Figure 15-6 illustrates how to use DW_debugger to monitor error rate. The counter tracks the number of errors detected by the error detection circuit in the data path. You can read the counter value through the rd_bits, and reset the counter through the wr_bits.

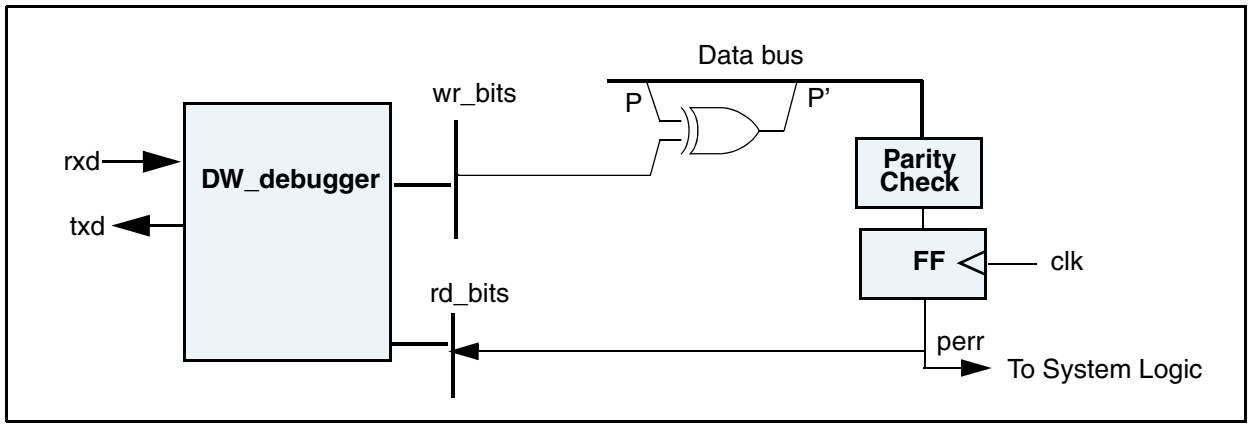
Figure 15-6 Error Rate Monitoring



Error Injection and Detection

Figure 15-7 illustrates how to use DW_debugger for error injection and detection. Use wr_bits to inject a parity error, and use rd_bits to determine whether the error was detected.

Figure 15-7 Error Injection and Detection



Data Probing

Figure 15-8 illustrates how to use DW_debugger wr_bits to select internal nets to probe with an oscilloscope or logic analyzer. Figure 15-9 illustrates how to select internal nets to probe and select an internal net as a trigger source for the oscilloscope or logic analyzer.

Figure 15-8 Selecting Internal Nets to Probe

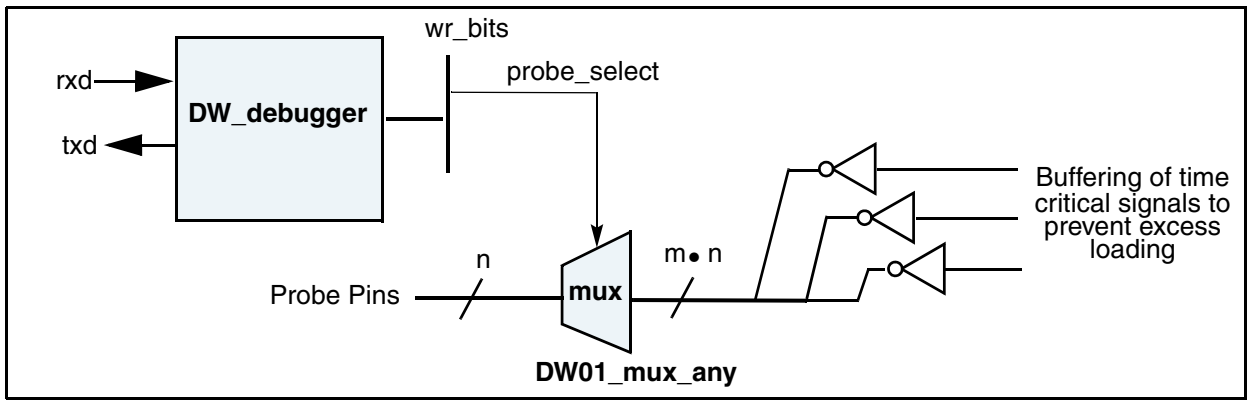
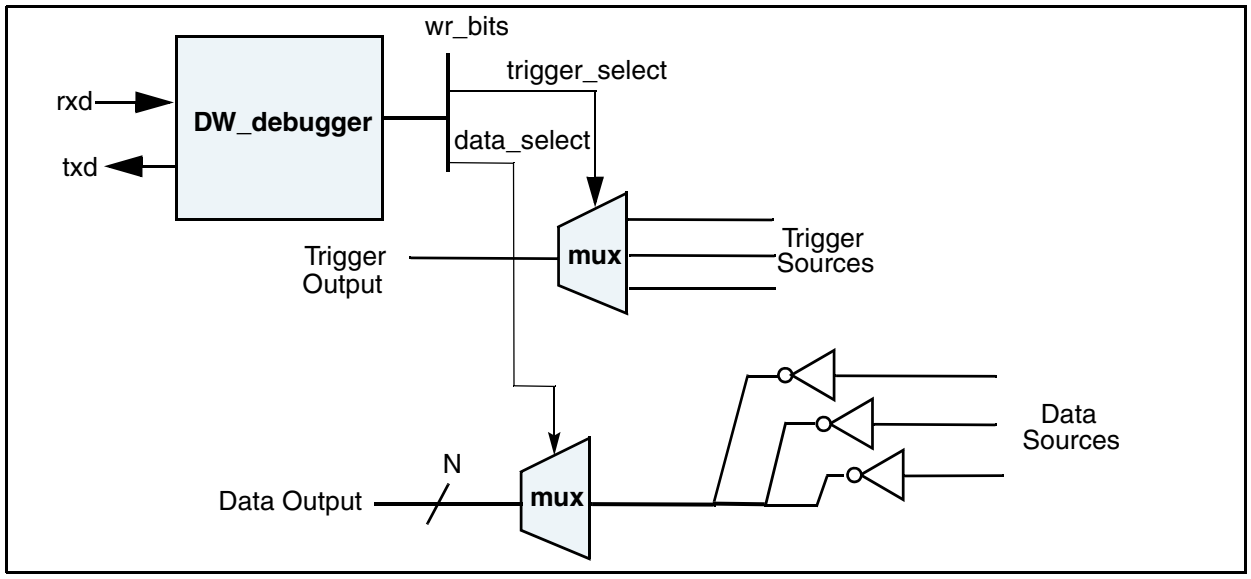


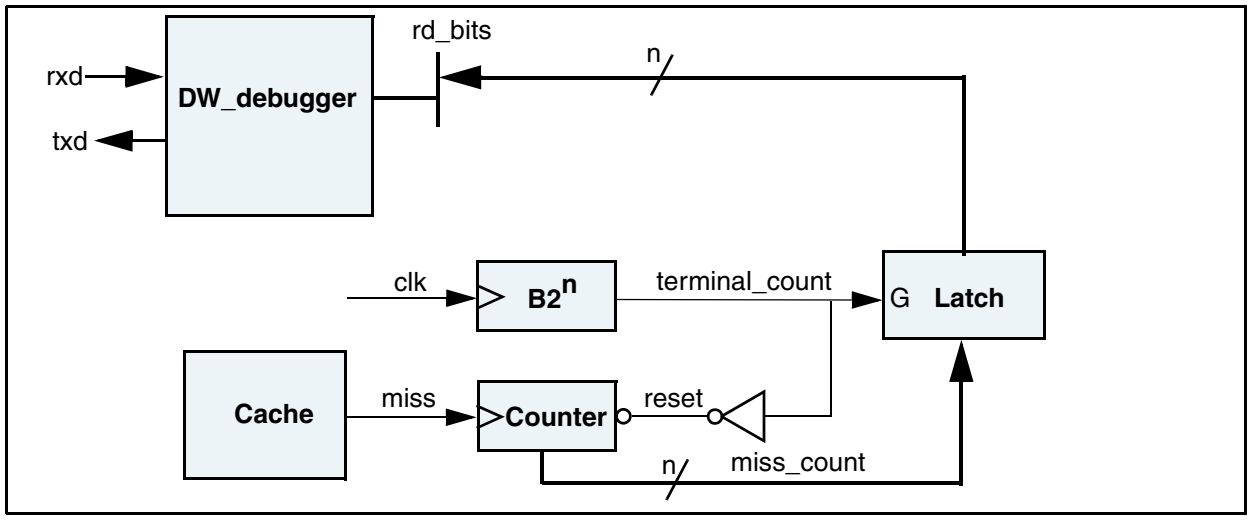
Figure 15-9 Probing with Independent Trigger Selection



Performance Monitoring

Figure 15-10 illustrates how to use DW_debugger wr_bits for performance monitoring. The counter circuit counts the number of misses out of 2n memory cycles. The count stored in the latch is available through the DW_debugger rd_bits.

Figure 15-10 Monitoring Memory Performance



Using DW Building Block IP Pipelined Multipliers

AN 96-002

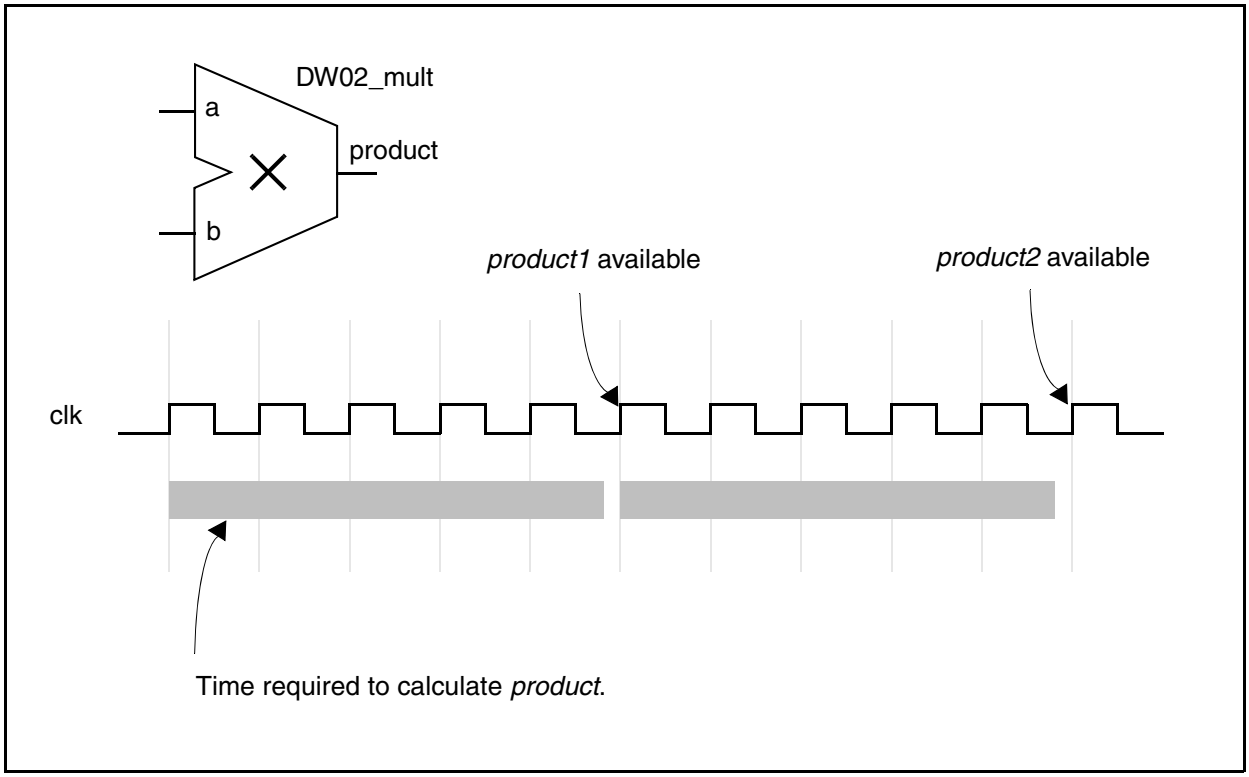
Introduction

For repeated multiplication operations, the pipelined multipliers in the Advanced Math family (DW02_mult_*n*_stage) provide increased throughput over the single-stage multiplier (DW02_mult).

For example, consider a design that uses a single instance of DW02_mult, as shown in Figure 16-1. Suppose that when synthesized to gates in a given technology, DW02_mult requires 19 ns to execute a multiply operation. If the circuit uses a 250 MHz (4 ns) clock, each multiply operation requires 5 clocks cycles. To perform 20 successive multiply operations, the DW02_mult circuit requires 5 x 20 = 100 clock cycles (400 ns).

The circuit delays and clock rates used here are selected for example only. Actual circuit performance depends on input bit-widths and target technology.

Figure 16-1 Successive Multiplications with DW02_mult



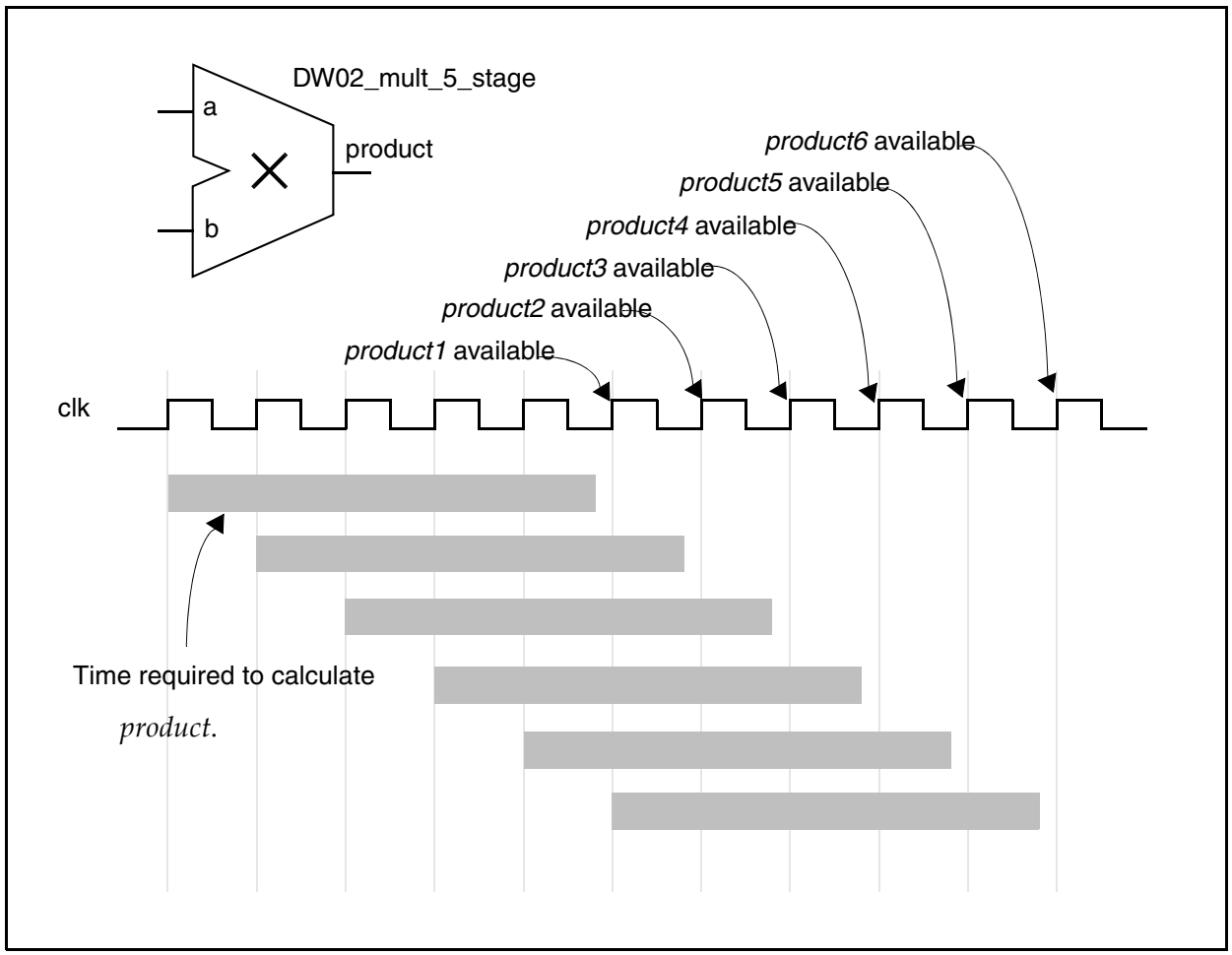
For the same application, the DW02_mult_5_stage pipelined multiplier offers increased throughput over the single-stage DW02_mult (Refer to Figure 16-2). DW02_mult_5_stage has a latency of four clock cycles, which means that result of a multiply operation is available on the fifth clock cycle after the input values are clocked in at pins *a* and *b*. However, because of its pipelined design, DW02_mult_5_stage can begin a new multiply operation on each clock cycle.

In other words, *product1* is available after 5 clocks, *product2* is available after 6 clocks, *product3* is available after 7 clocks, and so on. For the 20 successive multiply functions in this example, *product20* is

available 24 clock cycles after the input data for *product1* is clocked in. The total time required to complete the 20 multiply operations with DW02_mult_5_stage is 24 x 4 ns = 96 ns.

For applications where you require less than a four clock cycle latency (for example, the same application with a slower clock rate) you can select DW02_mult_2_stage, DW02_mult_3_stage, or DW02_mult_4_stage to get the latency you need. For five clock cycle latency, use DW02_mult_6_stage.

Figure 16-2 Successive Multiplications with DW02_mult_5_stage



Instantiating Pipelined Multipliers for Design Compiler

The DW02_mult_*n*_stage pipelined multipliers are instantiated explicitly from Design Compiler, since there are no VHDL or Verilog operators or functions available. For more information, refer to the individual datasheets.

Example 16-1 shows how to instantiate DW02_mult_5_stage in VHDL. Example 16-2 shows how to instantiate DW02_mult_5_stage in Verilog.

Example 16-1 Instantiating DW02_mult_5_stage in VHDL

```
library IEEE, DW02;
use IEEE.std_logic_1164.all;
use DW02.DW02_components.all;

entity pipelined_multiplier is
  generic(wordlength1: NATURAL := 4;
          wordlength2: NATURAL := 6);
  port(in1    : in STD_LOGIC_VECTOR(wordlength1-1 downto 0);
       in2    : in STD_LOGIC_VECTOR(wordlength2-1 downto 0);
       control : in STD_LOGIC;
       clk     : in STD_LOGIC;
       product : out STD_LOGIC_VECTOR(wordlength1+wordlength2-1 downto 0));
end pipelined_multiplier;

architecture str of pipelined_multiplier is
begin

  -- instantiate DW02_mult_5_stage
  U1: DW02_mult_5_stage

  generic map(A_width => wordlength1, B_width => wordlength2)
  port map    (A => in1, B => in2,
               TC => control, CLK => clk, PRODUCT => product);
end str;
```

Example 16-2 Instantiating DW02_mult_5_stage in Verilog

```
module pipelined_multiplier(in1,in2,control,clk,product);
  parameter wordlength1 = 8,wordlength2 = 8;
  input [wordlength1-1:0] in1;
  input [wordlength2-1:0] in2;
  input control;
  input clk;
  output [wordlength1+wordlength2-1:0] product;

  // instantiate DW02_mult_5_stage
  DW02_mult_5_stage #(wordlength1,wordlength2)
  U1(in1,in2,control,clk,product);
endmodule
```

If Design Compiler Ultra licenses are unavailable, Design Compiler uses its own register balance algorithm to move registers to good locations. However, in general, Design Compiler's balancing register algorithms may not generate solutions that are as optimal.