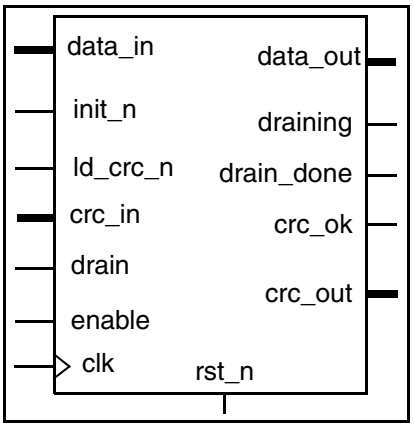
DW_crc_s

Universal Synchronous (Clocked) CRC Generator/Checker

Version, STAR and Download Information: IP Directory

Features and Benefits

- Parameterized arbitrary polynomial (up to 64-bit)
- Parameterized data width (up to polynomial size)
- Parameterized register initialization (all ones or all zeros)
- Parameterized inverted insertion of generated CRC
- Parameterized bit and byte ordering
- Loadable CRC value for use in context switching of interspersed blocks



Description

DW_crc_s is a universal Cyclic Redundancy Check (CRC) Polynomial Generator/Checker that provides data integrity on data streams for various applications.

Table 1-1 Pin Description

Pin Name	Width	Direction	Function
clk	1 bit	Input	Clock input
rst_n	1 bit	Input	Asynchronous reset input, active low
init_n	1 bit	Input	Synchronous initialization control input, active low
enable	1 bit	Input	Enable control input for all operations (other than reset and initialization), active high
drain	1 bit	Input	Drains control input, active high
ld_crc_n	1 bit	Input	Synchronous CRC register load control input, active low
data_in	*data_width* bit(s)	Input	Input data
crc_in	*poly_size* bit(s)	Input	Input CRC value (to be loaded into the CRC register as commanded by the `ld_crc_n` control input)
draining	1 bit	Output	Indicates that the CRC register is draining (inserting the CRC into the data stream)

Table 1-1 Pin Description (Continued)

Pin Name	Width	Direction	Function
drain_done	1 bit	Output	Indicates that the CRC register has finished draining
crc_ok	1 bit	Output	Indicates a correct residual CRC value, active high
data_out	*data_width* bit(s)	Output	Output data
crc_out	*poly_size* bit(s)	Output	Provides constant monitoring of the CRC register

Table 1-2 Parameter Description

Parameter	Values	Description
data_width	1 to *poly_size* Default: 16	Width of `data_in` and `data_out` (also the number of bits per clock) The *poly_size* value must be an integer multiple of *data_width*.
poly_size	2 to 64 Default: 16	Size of the CRC polynomial The *poly_size* value must be an integer multiple of *data_width*.
crc_cfg	0 to 7 Default: 7	CRC initialization and insertion configuration (see Table 1-5 on page 3)
bit_order	0 to 3 Default: 3	Bit and byte order configuration (see Table 1-6 on page 3)
poly_coef0	1 to 65535[a] Default: 4129[b]	Polynomial coefficients 0 through 15
poly_coef1	0 to 65535 Default: 0	Polynomial coefficients 16 through 31
poly_coef2	0 to 65535 Default: 0	Polynomial coefficients 32 through 47
poly_coef3	0 to 65535 Default: 0	Polynomial coefficients 48 through 63

a. The *poly_coef0* value must be an odd number (since all primitive polynomials include the coefficient 1, which is equivalent to X^0).
b. CCITT-CRC16 polynomial is $X^{16} + X^{12} + X^5 + 1$, thus poly_coef0 = $2^{12} + 2^5 + 1 = 4129$.

Table 1-3 Synthesis Implementations

Implementation Name	Implementation	License Feature Required
str	Synthesis model	DesignWare

Table 1-4 Simulation Models

Model	Function
DW04.DW_CRC_S_CFG_SIM	Design unit name for VHDL simulation
dw/dw04/src/DW_crc_s_sim.vhd	VHDL simulation model source code
dw/sim_ver/DW_crc_s.v	Verilog simulation model source code

Table 1-5 CRC Configurations

crc_cfg	CRC Register Initialization Value	Inserted CRC bits
0	Zeros	Not Inverted
1	Ones	Not Inverted
2	Zeros	XORed with ... 010101
3	Ones	XORed with ... 010101
4	Zeros	XORed with ... 101010
5	Ones	XORed with ... 101010
6	Zeros	Inverted
7	Ones	Inverted

Table 1-6 Bit Order Modes

bit_order	CRC Calculation Bit Order	Bytes Ordered[a]
0	MSB first, LSB last	MSB first, LSB last
1	LSB first, MSB last	LSB first, MSB last
2	MSB first, LSB last	LSB first, MSB last
3	LSB first, MSB last	MSB first, LSB last

a. Byte ordering is only available when *data_width* is a multiple of 8. For *data_width* values other than a multiple of 8, the *bit_order* value is restricted to 0 and 1.

Table 1-7 Control Functionality

rst_n	init_n	enable	ld_crc_n	drain	Description
0	X	X	X	X	Reset (asynchronous)
1	0	X	X	X	Initialize (same as reset, but synchronous)

Table 1-7 Control Functionality (Continued)

rst_n	init_n	enable	ld_crc_n	drain	Description
1	1	0	X	X	Disabled (stalled)
1	1	1	0	X	Load CRC register
1	1	1	1	0	Calculate CRC on input data
1	1	1	1	1	Begin CRC insertion

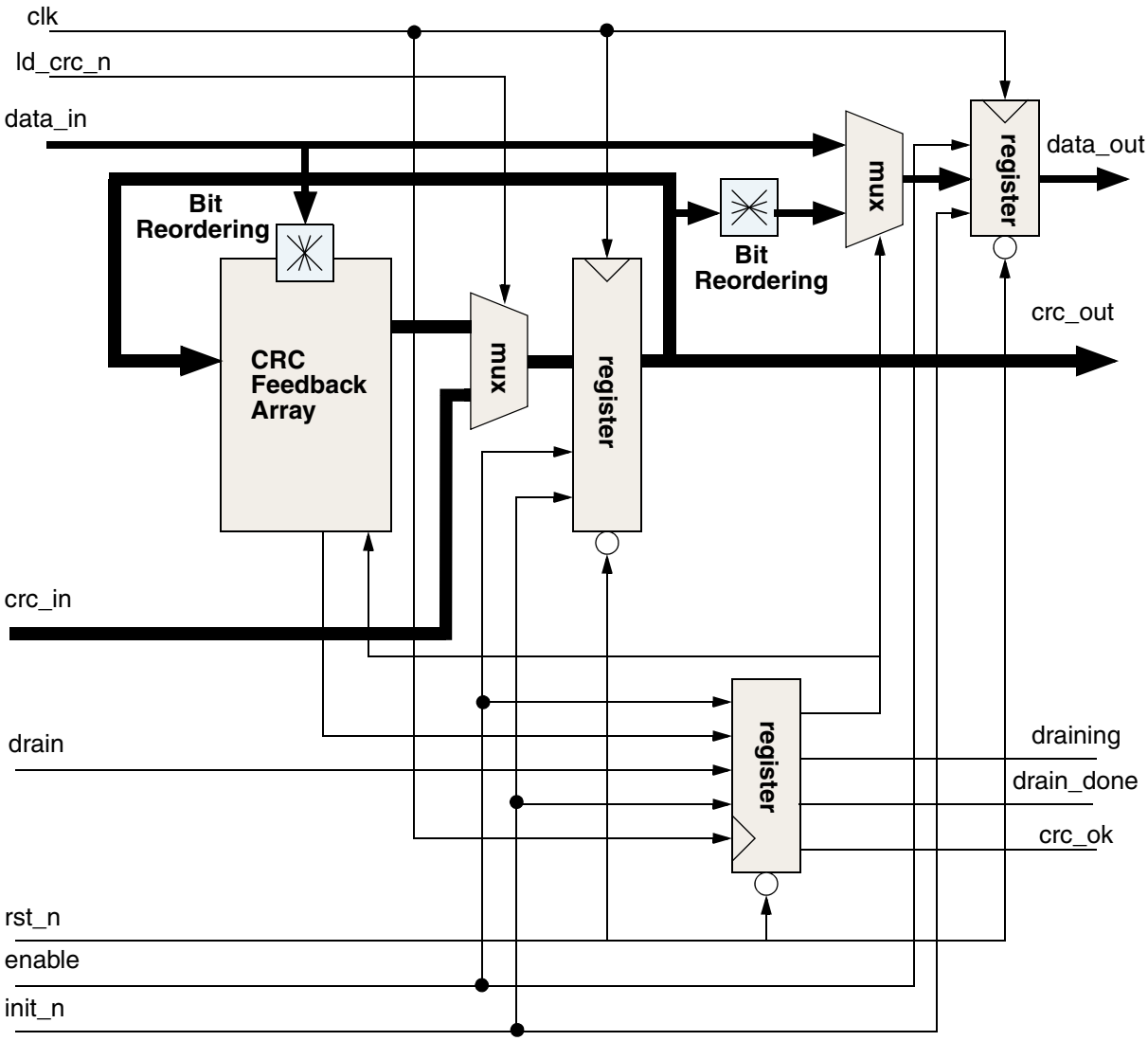
Table 1-8 Parameters for Common CRC Implementations

Description	*poly_size*	*crc_cfg*	*poly_coef0*	*poly_coef1*	*poly_coef2*	*poly_coef3*
CRC-32	32	7	7607	1217	0 (not used)	0 (not used)
CRC-16	16	7	32773	0 (not used)	0 (not used)	0 (not used)
CCIT-CRC16	16	7	4129	0 (not used)	0 (not used)	0 (not used)
ATM Header CRC	8	7	7	0 (not used)	0 (not used)	0 (not used)

The CRC polynomial (size and coefficients) is specified at design time along with bit ordering, initialization, and insertion characteristics. The data width is also specified at design time, and determines the number of bits upon which DW_crc_s calculates CRC values in a single clock cycle. Thus, an instance of DW_crc_s with a 32-bit polynomial and a 16-bit data width performs CRC calculations on 16 bits per clock cycle for a 32-bit CRC.

Figure 1-1 shows the functional block diagram of DW_crc_s.

Figure 1-1 Functional Block Diagram of DW_crc_s



Polynomial Specification

One important aspect of a CRC error detection scheme is the size and quality of the generator polynomial.

Note The size of the CRC generator polynomial affects the possible valid *data_width* values, since the *poly_size* value must be an integer multiple of *data_width.*

Because CRC error detection has been used for many years, standard CRC generator polynomials are widely used. In networking, CRC-32 is the most popular polynomial used for data payload protection.

Some other standard protocols use CRC-16, and others use CCITT-CRC-16. In most cases, system architecture or inter-operability with standard protocols usually determine the polynomial used in an error detection system.

The generator polynomial of DW_crc_s is controlled by user-specified parameters (*poly_coef0, poly_coef1, poly_coef2,* and *poly_coef3*). Table 1-8 on page 4 lists the values of these parameters for several standard generator polynomials. If the generator polynomial required for a CRC system is not listed in Table 1-8, the polynomial coefficient values can easily be calculated with the method described in the following section.

Calculating Polynomial Coefficient Values Specification

Each of the four polynomial coefficient values represent a sixteen-bit slice of polynomial coefficient positions. Thus, the following is true:

- Terms from 1 to x^{15} are contained in *poly_coef0*
- Terms from x^{16} to x^{31} are contained in *poly_coef1*
- Terms from x^{32} to x^{47} are contained in *poly_coef2*
- Terms from x^{48} to x^{63} are contained in *poly_coef3*

The term x^{poly_size} does not need to be contained in the polynomial coefficient values because it is implied by the value of *poly_size* itself (ex. for a 32-bit polynomial x^{32} is implicit and therefore does not need to be specified). Thus, for *poly_coef0*, add up all of the terms of the generator polynomial between 1 and x^{15}, with x = 2. For *poly_coef1*, sum all terms of the generator polynomial between x^{16} and x^{31}, with x =2, all divided by 2^{16}. For *poly_coef2*, sum all terms of the generator polynomial between x^{32} and x^{47}, with x = 2, all divided by 2^{32}. For *poly_coef3*, sum all terms of the generator polynomial between x^{48} and x^{63}, with x = 2, all divided by 2^{48}. This process is essentially a binary to decimal conversion with terms that appear in a polynomial being ones, and terms that do not appear in the polynomial being zeros. The following example shows the coefficients for CRC-32:

$$\text{CRC-32} = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^{8} + x^{7} + x^{5} + x^{4} + x^{2} + x + 1$$

$$\text{poly\_coef0} = x^{12} + x^{11} + x^{10} + x^{8} + x^{7} + x^{5} + x^{4} + x^{2} + x + 1, \text{ evaluated at } x = 2$$

$$\text{poly\_coef0} = 2^{12} + 2^{11} + 2^{10} + 2^{8} + 2^{7} + x^{5} + 2^{4} + 2^{2} + 2 + 1 = 4096 + 2048 + 1024 + 256 + 128 + 32 + 16 + 4 + 2 + 1$$

$$\text{poly\_coef0} = 7607 \quad (\text{binary equivalent} = 0001110110110111)$$

$$\text{poly\_coef1} = (x^{26} + x^{23} + x^{22} + x^{16})/2^{16}, \text{ evaluated at } x = 2$$

$$\text{poly\_coef1} = (2^{26} + 2^{23} + 2^{22} + 2^{16})/2^{16} = 2^{10} + 2^{7} + 2^{6} + 1 = 1024 + 128 + 64 + 1$$

$$\text{poly\_coef1} = 1217 \quad (\text{binary equivalent} = 0000010011000001)$$

Because *poly_size* = *32* for CRC-32, there are no terms beyond `x31`. Thus, *poly_coef2* = *poly_coef3* = `0`.

CRC Register Configuration

CRC register configuration refers to the following:

- How the register that calculates the CRC is initialized (to ones or to zeros)
- Which (if any) bits are inverted when inserting the CRC check bits into the data stream

DW_crc_s can accommodate eight different CRC register configurations as specified by the *crc_cfg* parameter. The most common initial value of CRC registers is all ones, but there are protocols that use zeros as the initial CRC register value. Odd values of *crc_cfg* e.g., 1, 3, 5, and 7) initialize the CRC register to all ones, while even values (e.g., 0, 2, 4, and 6) initialize the CRC register to all zeros.

It is very common to invert bits of the CRC check bits when inserting them into the data stream. DW_crc_s can be configured to invert the following:

- All CRC check bits (*crc_cfg* = 6 or 7)
- Odd check bits (*crc_cfg* = 2 or 3)
- Even check bits (*crc_cfg* = 4 or 5)
- No check bits (*crc_cfg* = 0 or 1)

Thus, the most common configuration (initialize to ones and invert CRC being inserted) is selected using 7 as the value of *crc_cfg*. Table 1-5 on page 3 shows all seven configurations.

Data Bit Order

CRC calculation is order dependent. A CRC system must calculate CRC values in the proper order to conform to a particular protocol and thus be interoperable.

To operate on data one bit at a time, the bit order must be determined outside of the context of the CRC block. Since DW_crc_s can operate on multiple data bits in a single clock cycle, you must specify what order the data bits are to be operated on.

DW_crc_s is configurable to calculate CRC on data (and insert check bits) in one of four bit order configurations, as specified by the *bit_order* parameter. The *bit_order* parameter can be set to the simple most significant bit first (*bit_order* = 0) or least significant bit first (*bit_order* = 1) configuration regardless of the *data_width* parameter. However, some protocols require a more intricate bit ordering scheme as they order bits within a byte in one direction (LSB to MSB), while bytes within a word are ordered in the opposite direction (e.g., MSB to LSB). Thus, *bit_order* values of 2 and 3 support such schemes, and are only valid when the *data_width* value is an even multiple of eight.

Figure 1-2 through Figure 1-5 shows the bit order schemes for DW_crc_s.

Figure 1-2 Bit Order Scheme: *bit_order* = 0

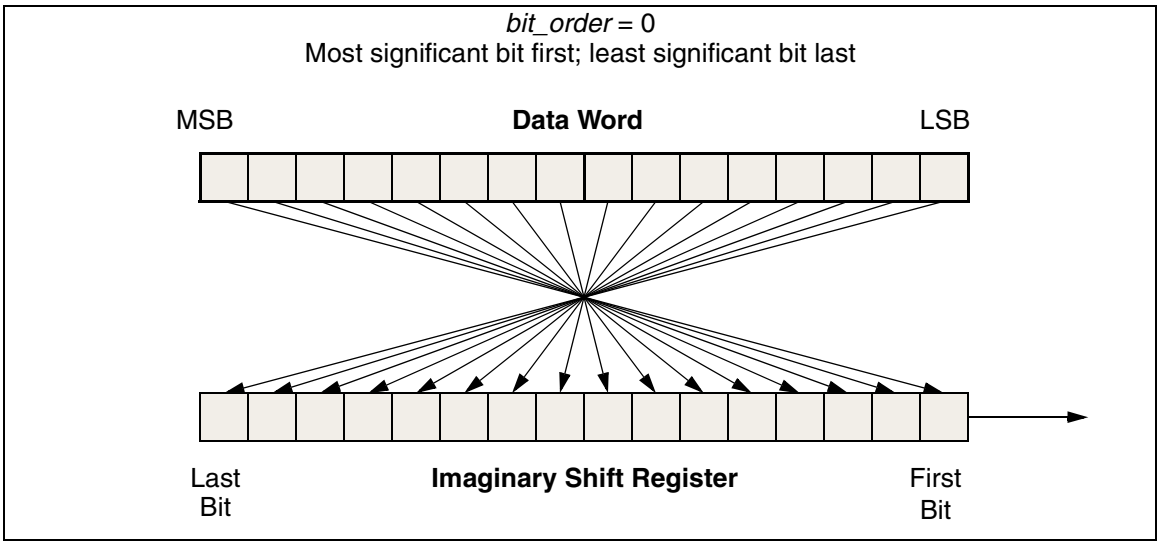


Figure 1-3 Bit Order Scheme: *bit_order* = 1

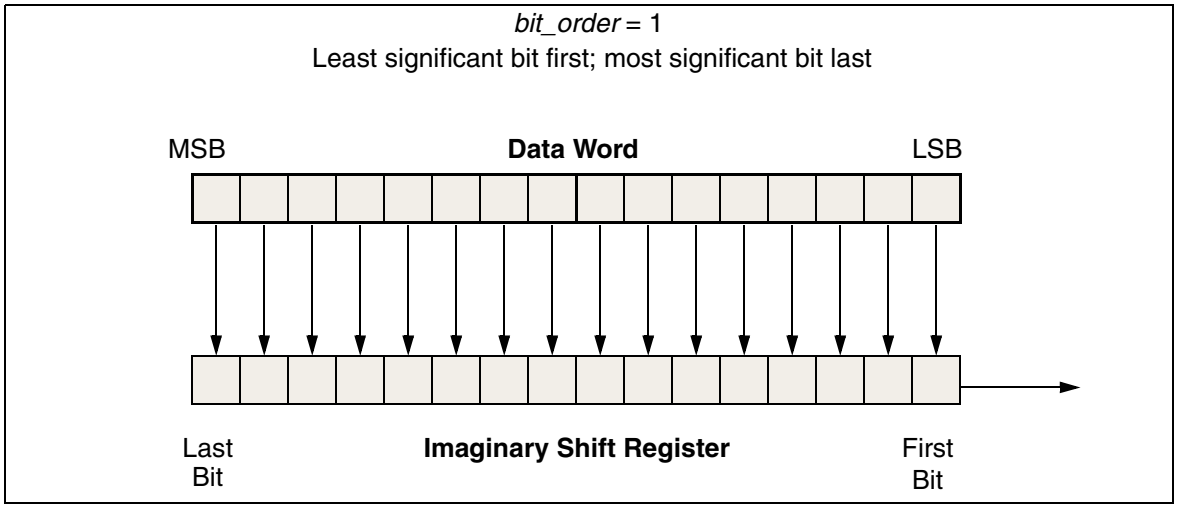


Figure 1-4 Bit Order Scheme: *bit_order* = 2

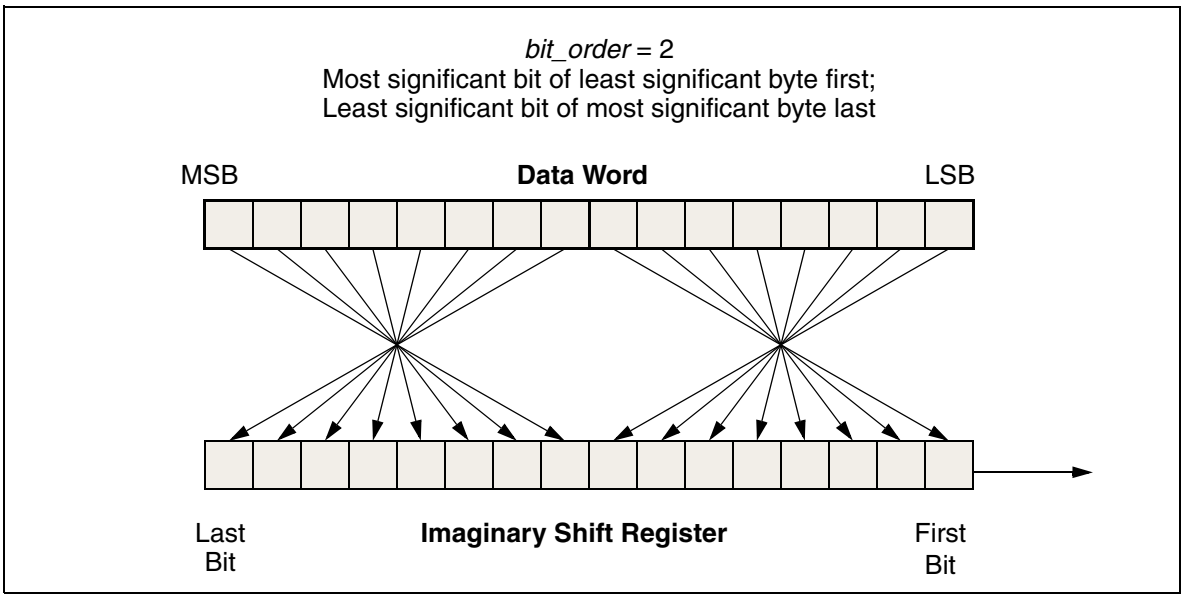
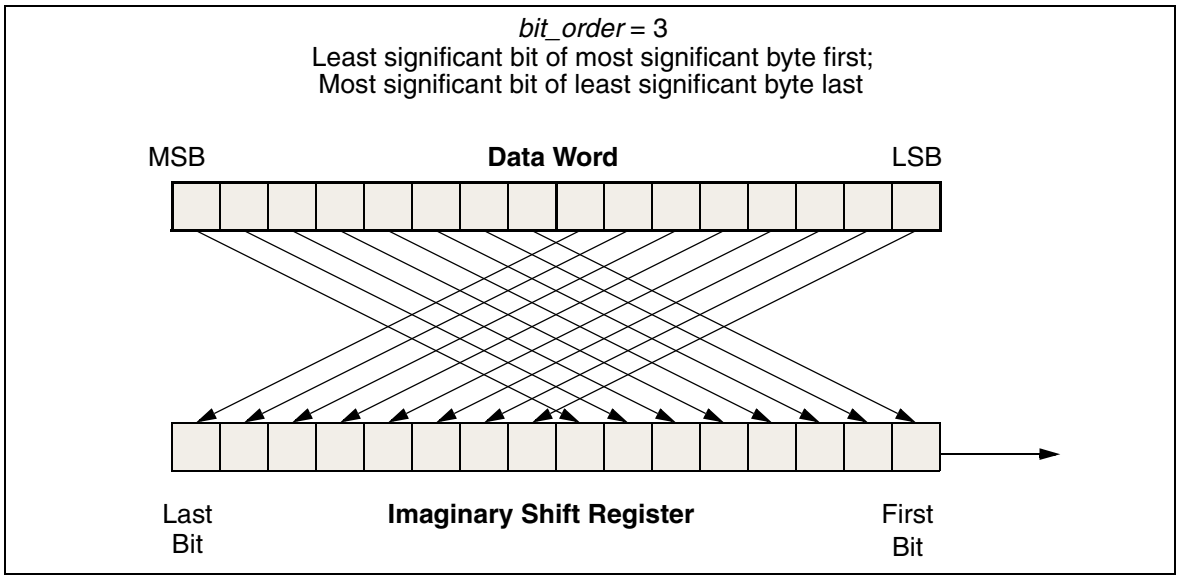


Figure 1-5 Bit Order Scheme: *bit_order* = 3



CRC Generation (Transmit or Store Operation)

Proper CRC calculation always begins with the CRC register initialized to the correct predetermined state. To generate and insert CRC check bits into the data stream, initialization is followed by a data calculation phase and a CRC drain phase. Thus, CRC generation for a transmit operation could occur as follows:

With `rst_n` inactive (HIGH) and `ld_crc_n` inactive (HIGH):

1. One or more cycles of initialization (`init_n` = LOW).
2. Zero or more cycles of stall (`enable` = LOW, `init_n` = HIGH, `drain` = LOW) (waiting for data can occur either in initialization or using stall).
3. Calculate CRC bits from data (`enable` = HIGH, `init_n` = HIGH, `drain` = LOW) (calculation on data words can be interspersed with stall cycles).
4. Once the last word of the data record to be protected has been clocked into DW_crc_s, the drain input is asserted for one or more cycles to initiate the draining of the CRC check bits into the data stream.
5. DW_crc_s drains its CRC register into the data stream, during which the draining output is active (HIGH) and `drain_done` is inactive (LOW).
6. Once all words of the CRC check bits have been inserted into the data stream, the `drain_done` output becomes active (HIGH) and the draining output becomes inactive (LOW). The CRC generation operation is now complete.

DW_crc_s is now ready to be initialized for the next operation.

CRC Checking (Receive or Retrieve Operation)

Proper CRC checking always begins with the CRC register initialized to the correct predetermined state. To check the validity of a data block with CRC, initialization is followed by a data calculation phase followed by a check bits calculation phase that ends with the `crc_ok` port active (HIGH) for valid CRC and inactive (LOW) for invalid CRC.

With `rst_n` inactive (HIGH) and `ld_crc_n` inactive (HIGH):

1. One or more cycles of initialization (`init_n` = LOW).
2. Zero or more cycles of stall (`enable` = LOW, `init_n` = HIGH, `drain` = LOW) (waiting for data occur either in initialization or using stall).
3. Calculate CRC bits from data (`enable` = HIGH, `init_n` = HIGH, `drain` = LOW) (calculation on data words can be interspersed with stall cycles).
4. Once the last word of the record being checked has been clocked into DW_crc_s, the check bits (previously inserted during a CRC generate operation) are clocked in. On the rising edge of `clk` that clocks the last word of check bits into DW_crc_s (which may also be the first word of check bits if *data_width* = *poly_size*), the `crc_ok` output port is updated to reflect the validity of the record being checked.

DW_crc_s is now ready to be initialized for the next operation.

Context Switching

Some applications may wish to share a CRC block between multiple data streams, while other applications may find it desirable to intersperse data from multiple transactions on a single data stream. Either type of application may find it necessary to save and restore the context of a CRC calculation phase.

DW_crc_s supports context switching by continuously providing the state of the CRC register (for saving its state to some storage element, whether it be RAM- or register-based), and providing a way to directly load the CRC register (for restoring a previously saved state).

Context switching is not recommended during the CRC insertion phase (while draining is active). So, it's recommended that an initialization cycle be added between any CRC insertions and subsequent context switches.

Figure 1-6 CRC Generation and Insertion Operation

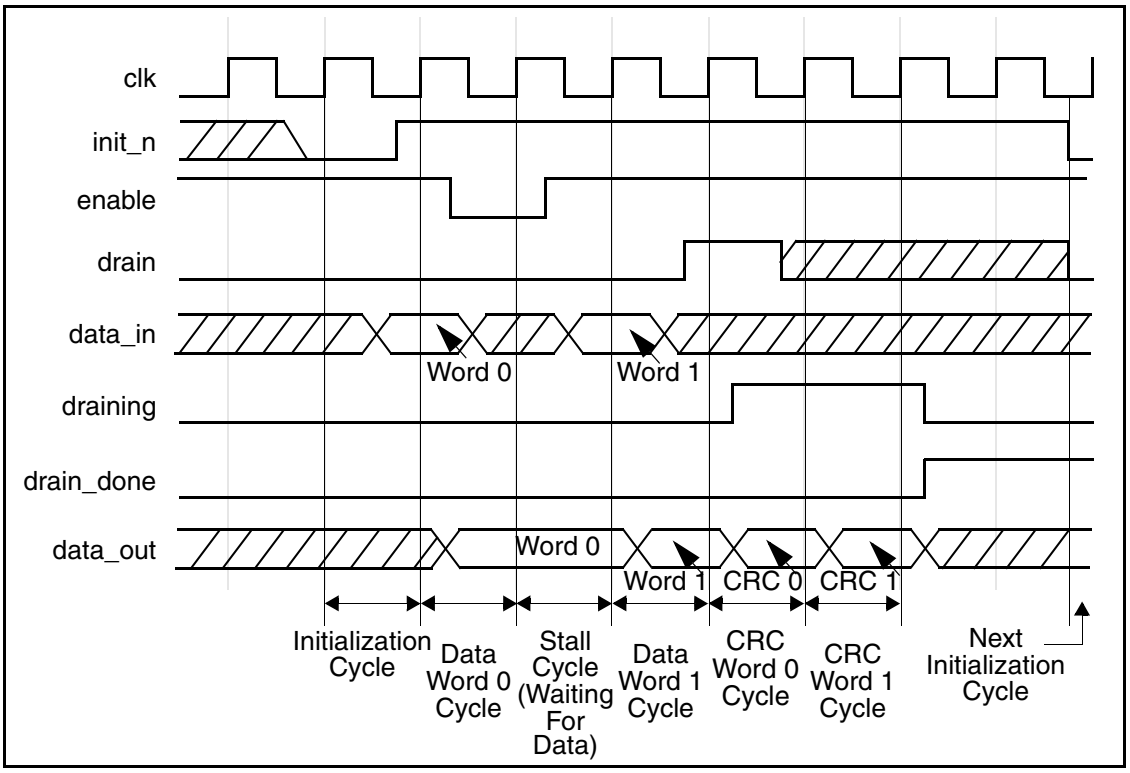
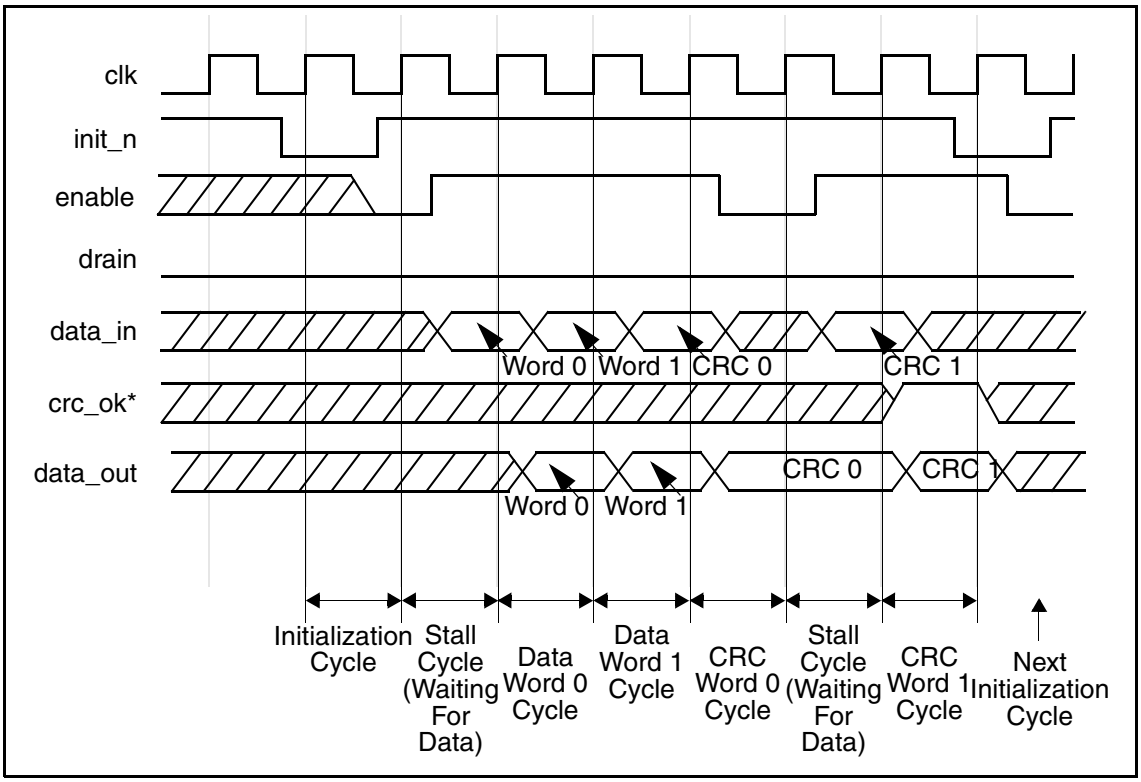


Figure 1-7 CRC Validity Check Operation



* crc_ok flag valid when last CRC is clocked into data_out.

Related Topics

- Application Specific – Data Integrity Overview
- DesignWare Building Block IP Documentation Overview

HDL Usage Through Component Instantiation - VHDL

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

entity DW_crc_s_inst is
  generic (inst_data_width : INTEGER := 16;   inst_poly_size : INTEGER := 16;
           inst_crc_cfg    : INTEGER := 7;    inst_bit_order : INTEGER := 3;
           inst_poly_coef0 : INTEGER := 4129; inst_poly_coef1: INTEGER := 0;
           inst_poly_coef2 : INTEGER := 0;    inst_poly_coef3: INTEGER := 0);
  port (inst_clk        : in std_logic;     inst_rst_n    : in std_logic;
        inst_init_n     : in std_logic;     inst_enable   : in std_logic;
        inst_drain      : in std_logic;     inst_ld_crc_n : in std_logic;
        inst_data_in    : in std_logic_vector(inst_data_width-1 downto 0);
        inst_crc_in     : in std_logic_vector(inst_poly_size-1 downto 0);
        draining_inst   : out std_logic;
        drain_done_inst : out std_logic;
        crc_ok_inst     : out std_logic;
        data_out_inst   : out std_logic_vector(inst_data_width-1 downto 0);
        crc_out_inst    : out std_logic_vector(inst_poly_size-1 downto 0)  );
    end DW_crc_s_inst;

architecture inst of DW_crc_s_inst is
begin

  -- Instance of DW_crc_s
  U1 : DW_crc_s
    generic map (data_width => inst_data_width,  poly_size => inst_poly_size,
                 crc_cfg => inst_crc_cfg,   bit_order => inst_bit_order,
                 poly_coef0 => inst_poly_coef0,
                 poly_coef1 => inst_poly_coef1,
                 poly_coef2 => inst_poly_coef2,
                 poly_coef3 => inst_poly_coef3 )
    port map (clk => inst_clk,   rst_n => inst_rst_n,  init_n => inst_init_n,
              enable => inst_enable,   drain => inst_drain,
              ld_crc_n => inst_ld_crc_n,   data_in => inst_data_in,
              crc_in => inst_crc_in,   draining => draining_inst,
              drain_done => drain_done_inst,   crc_ok => crc_ok_inst,
              data_out => data_out_inst,   crc_out => crc_out_inst );
end inst;

-- Configuration for use with VSS simulator
-- pragma translate_off
configuration DW_crc_s_inst_cfg_inst of DW_crc_s_inst is
  for inst
  end for; -- inst
end DW_crc_s_inst_cfg_inst;
-- pragma translate_on
```

HDL Usage Through Component Instantiation - Verilog

```
module DW_crc_s_inst(inst_clk, inst_rst_n, inst_init_n, inst_enable,
                     inst_drain, inst_ld_crc_n, inst_data_in, inst_crc_in,
                     draining_inst, drain_done_inst, crc_ok_inst,
                     data_out_inst, crc_out_inst );

  parameter data_width = 16;
  parameter poly_size = 16;
  parameter crc_cfg = 7;
  parameter bit_order = 3;
  parameter poly_coef0 = 4129;
  parameter poly_coef1 = 0;
  parameter poly_coef2 = 0;
  parameter poly_coef3 = 0;

  input inst_clk;
  input inst_rst_n;
  input inst_init_n;
  input inst_enable;
  input inst_drain;
  input inst_ld_crc_n;
  input [data_width-1 : 0] inst_data_in;
  input [poly_size-1 : 0] inst_crc_in;
  output draining_inst;
  output drain_done_inst;
  output crc_ok_inst;
  output [data_width-1 : 0] data_out_inst;
  output [poly_size-1 : 0] crc_out_inst;

  // Instance of DW_crc_s
  DW_crc_s #(data_width,   poly_size,   crc_cfg,   bit_order,
             poly_coef0,   poly_coef1,   poly_coef2,   poly_coef3)
    U1 (.clk(inst_clk),    .rst_n(inst_rst_n),   .init_n(inst_init_n),
        .enable(inst_enable),   .drain(inst_drain),
        .ld_crc_n(inst_ld_crc_n),   .data_in(inst_data_in),
        .crc_in(inst_crc_in),   .draining(draining_inst),
        .drain_done(drain_done_inst),   .crc_ok(crc_ok_inst),
        .data_out(data_out_inst),   .crc_out(crc_out_inst)  );
endmodule
```

Copyright Notice and Proprietary Information

Destination Control Statement

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

Disclaimer

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

Trademarks

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

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

Third-Party Links

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

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

www.synopsys.com