Table Of Contents

Other projects

Quick search

crc_ops

extras/crc_ops.vhdl

Dependencies

None

Description

This package provides a general purpose CRC implementation. It consists of a set of functions that can be used to iteratively process successive data vectors as well an an entity that combines the functions into a synthesizable form. The CRC can be readily specified using the Rocksoft notation described in "A Painless Guide to CRC Error Detection Algorithms", Williams 1993. A CRC specification consists of the following parameters:

Poly
: The generator polynomial

Xor_in
: The initialization vector "xored" with an all-'0's shift register

Xor_out
: A vector xored with the shift register to produce the final value

Reflect_in
: Process data bits from left to right (false) or right to left (true)

Reflect_out
: Determine bit order of final crc result

A CRC can be computed using a set of three functions **`init_crc()`**, **`next_crc()`**, and **`end_crc()`**. All functions are assigned to a common variable/signal that maintans the shift register state between succesive calls. After initialization with `init_crc`, data is processed by repeated calls to `next_crc`. The width of the data vector is unconstrained allowing you to process bits in chunks of any desired size. Using a 1-bit array for data is equivalent to a bit-serial CRC implementation. When all data has been passed through the CRC it is completed with a call to `end_crc` to produce the final CRC value.

Example usage

Implementing a CRC without depending on an external generator tool is easy and flexible by iteratively computing the CRC in a loop. This will synthesize into a combinational circuit:

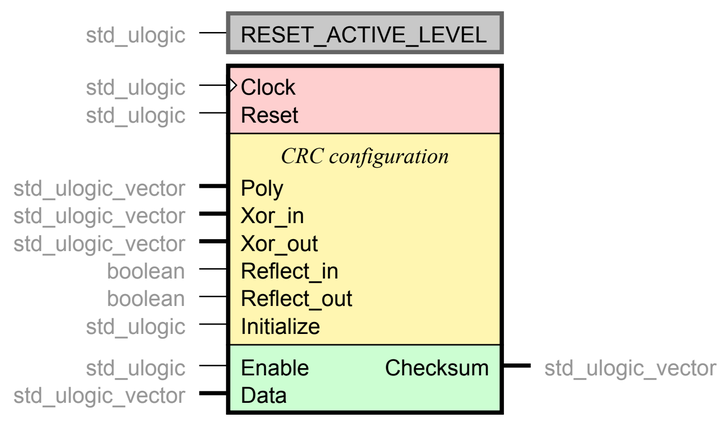
```
-- CRC-16-USB
constant poly        : bit_vector := X"8005";
constant xor_in      : bit_vector := X"FFFF";
constant xor_out     : bit_vector := X"FFFF";
constant reflect_in  : boolean := true;
constant reflect_out : boolean := true;

-- Implement CRC-16 with byte-wide inputs:
subtype word is bit_vector(7 downto 0);
type word_vec is array( natural range <> ) of word;
variable data : word_vec(0 to 9);
variable crc  : bit_vector(poly'range);
...
crc := init_crc(xor_in);
for i in data'range loop
  crc := next_crc(crc, poly, reflect_in, data(i));
end loop;
crc := end_crc(crc, reflect_out, xor_out);

-- Implement CRC-16 with nibble-wide inputs:
subtype nibble is bit_vector(3 downto 0);
type nibble_vec is array( natural range <> ) of nibble;
variable data : nibble_vec(0 to 9);
variable crc  : bit_vector(poly'range);
...
crc := init_crc(xor_in);
for i in data'range loop
  crc := next_crc(crc, poly, reflect_in, data(i));
end loop;
crc := end_crc(crc, reflect_out, xor_out);
```

A synthesizable component is provided to serve as a guide to using these functions in practical designs. The input data port has been left unconstrained to allow variable sized data to be fed into the CRC. Limiting its width to 1-bit will result in a bit-serial implementation. The synthesized logic will be minimized if all of the CRC configuration parameters are constants.

```
signal nibble   : std_ulogic_vector(3 downto 0); -- Process 4-bits at a time
signal checksum : std_ulogic_vector(15 downto 0);
...
crc_16: crc
  port map (
    Clock => clock,
    Reset => reset,

    -- CRC configuration parameters
    Poly        => poly,
    Xor_in      => xor_in,
    Xor_out     => xor_out,
    Reflect_in  => reflect_in,
    Reflect_out => reflect_out,

    Initialize => crc_init, -- Resets CRC register with init_crc function
    Enable     => crc_en,   -- Process next nibble

    Data     => nibble,
    Checksum => checksum
  );
```

Components

crc



`crc_ops.`**`crc`**

Calculate a CRC sequentially.

Generics:	• **RESET_ACTIVE_LEVEL** (*std_ulogic*) – Asynch. reset control level
Port:	• **Clock** (*in std_ulogic*) – System clock • **Reset** (*in std_ulogic*) – Asynchronous reset • **Poly** (*in std_ulogic_vector*) – Polynomial • **Xor_in** (*in std_ulogic_vector*) – Invert (XOR) initial state • **Xor_out** (*in std_ulogic_vector*) – Invert (XOR) final state • **Reflect_in** (*in boolean*) – Swap input bit order • **Reflect_out** (*in boolean*) – Swap output bit order • **Initialize** (*in std_ulogic*) – Reset the CRC state • **Enable** (*in std_ulogic*) – Indicates data is valid for next CRC update • **Data** (*in std_ulogic_vector*) – New data (can be any width needed) • **Checksum** (*out std_ulogic_vector*) – Computed CRC

Subprograms

`crc_ops.`**`init_crc`** (*Xor_in : bit_vector*) → bit_vector

Initialize CRC state.

Parameters:	• **Xor_in** (*bit_vector*) – Apply XOR to initial '0' state
Returns:	New state of CRC.

`crc_ops.`**`next_crc`** (*Crc : bit_vector*; *Poly : bit_vector*; *Reflect_in : boolean*; *Data : bit_vector*) → bit_vector

Add new data to the CRC.

Parameters:	• **Crc** (*bit_vector*) – Current CRC state • **Poly** (*bit_vector*) – Polynomial for the CRC • **Reflect_in** (*boolean*) – Reverse bits of Data when true • **Data** (*bit_vector*) – Next data word to add to CRC
Returns:	New state of CRC.

`crc_ops.`**`end_crc`** (*Crc : bit_vector*; *Reflect_out : boolean*; *Xor_out : bit_vector*) → bit_vector

Finalize the CRC.

Parameters:	• **Crc** (*bit_vector*) – Current CRC state • **Reflect_out** (*boolean*) – Reverse bits of result when true • **Xor_out** (*bit_vector*) – Apply XOR to final state (inversion)
Returns:	Final CRC value