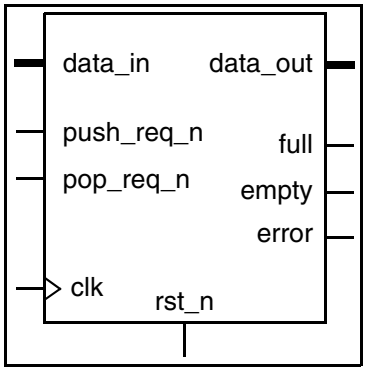
DW_stack

Synchronous (Single-Clock) Stack

Version, STAR and Download Information: IP Directory

Features and Benefits

- Parameterized word width and depth
- Stack empty and full status flags
- Stack error flag indicating underflow and overflow
- Fully registered synchronous flag output ports
- All operations execute in a single clock cycle
- D flip-flop based memory array for high testability
- Parameterized reset mode (synchronous or asynchronous)



Description

DW_stack is a fully synchronous, single-clock stack. It combines the DW_stackctl stack controller and the DW_ram_r_w_s_dff flip-flop-based RAM DesignWare components.

The stack provides parameterized word width and depth, and a full complement of flags: full, empty, and error.

The reset mode is selected at instantiation as either synchronous or asynchronous, and to include or exclude the RAM array.

The DW_stack is recommended for relatively small RAM configurations. For large stacks (dependent on your technology and requirements), use the DW_stackctl in conjunction with a compiled, full-custom RAM array.

Table 1-1 Pin Description

Pin Name	Width	Direction	Function
clk	1 bit	Input	Input clock
rst_n	1 bit	Input	Reset input, active low asynchronous if `rst_mode` = 0 or 2, synchronous if `rst_mode` = 1 or 3
push_req_n	1 bit	Input	Stack push request, active low
pop_req_n	1 bit	Input	Stack pop request, active low
data_in	*data_width* bit(s)	Input	Stack push data
empty	1 bit	Output	Stack empty flag, active high

Table 1-1 Pin Description (Continued)

Pin Name	Width	Direction	Function
full	1 bit	Output	Stack full flag, active high
error	1 bit	Output	Stack error output, active high
data_out	*data_width* bit(s)	Output	Stack pop data

Table 1-2 Parameter Description

Parameter	Values	Description
width	1 to 256 Default: None	Width of `data_in` and `data_out` buses
depth	2 to 256 Default: None	Depth (in words) of memory array
err_mode	0 or 1 Default: 0	Error mode 0 = underflow/overflow error, hold until reset, 1 = underflow/overflow error, hold until next clock.
rst_mode	0 to 3 Default: 0	Reset mode 0 = asynchronous reset including memory, 1 = synchronous reset including memory, 2 = asynchronous reset excluding memory, 3 = synchronous reset excluding memory.

Table 1-3 Synthesis Implementations[a]

Implementation Name	Function	License Feature Required
rpl	Ripple carry synthesis model	DesignWare
cl2	Full carry look-ahead model	DesignWare

a. During synthesis, Design Compiler will select the appropriate architecture for your constraints. However, you may force Design Compiler to use any architectures described in this table. For more, see *DesignWare Building Block IP User Guide*

Table 1-4 Simulation Models

Model	Function
DW06.DW_STACK_CFG_SIM	Design unit name for VHDL simulation
dw/dw06/src/DW_stack_sim.vhd	VHDL simulation model source code
dw/sim_ver/DW_stack.v	Verilog simulation model source code

Table 1-5 Error Mode Description

error_mode	Error Types Detected	Error Output
0	Underflow/Overflow	Registered - hold until reset
1	Underflow/Overflow	Not registered - hold until next clock

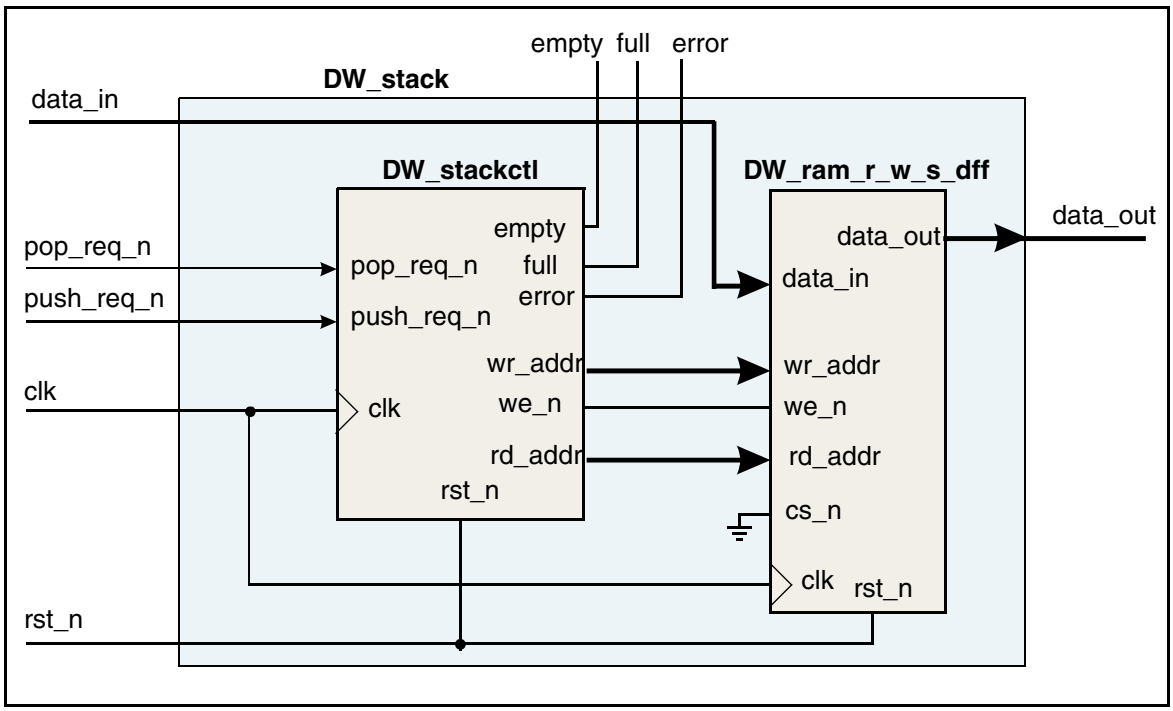
Table 1-6 Push and Pop Operation Function Table

push_req_n	full	pop_req_n	empty	Action	New Error
0	0	X	X	Push operation	No
0	1	X	0	Overrun; incoming data dropped (no action other than error generation)	Yes
1	X	0	0	Pop operation	No
1	0	0	1	Underrun; (no action other than error generation)	Yes
1	X	1	X	No action	No

Table 1-7 Internal Write and Read Address Pointers Relationship

Write Pointer	Read Pointer	Memory Status
0	0	Empty (zero words in memory)
1	0	One word in memory
K	*K* - 1	K words in memory (1 < *K* < *depth*)
depth – 1	*depth* – 2	*depth* – 1 words in memory
depth – 1	*depth* – 1	full (*depth* words in memory)

Figure 1-1 DW_stack Block Diagram



Writing to the Stack (Push)

A push is executed when the `push_req_n` input is asserted (LOW) and the `full` flag is inactive (LOW). Asserting `push_req_n` causes the internal address pointers to increment on the next rising edge of `clk`. Thus, the data at the `data_in` port is written to the next available location in the stack. The data at the `data_in` port must be stable for a setup time before the rising edge of `clk`.

DW_stack is a single-clock cycle stack. When the previous operation (at the previous rising edge of `clk`) ends, both internal address pointers are immediately adjusted, and point to the proper locations for the next operation in the next clock cycle. Refer to Table 1-7 on page 3 for internal address pointer information.

For example, assume that a push occurred in the previous clock cycle. After a transient internal delay, the stack internal write address pointer points to the next empty memory unit ready for another new push operation. Meanwhile, the internal read pointer points to the data that was just pushed, and data at the `data_out` port is ready for "prefetching."

Write Errors

The `error` output is activated if a push is attempted while the stack is full. That is, if:

- The push_req_n input is asserted (LOW), and
- The full flag is active (HIGH)

at the rising edge of `clk`.

Reading from the Stack (Pop)

A pop operation occurs when

- The `pop_req_n` line is asserted (LOW),
- The stack is not empty, and
- The `push_req_n` line is inactive (HIGH).

Asserting `pop_req_n` causes the internal read pointer to decrement on the next rising edge of `clk`. Thus, the stack read data must be captured on the `clk` following the assertion of `pop_req_n`.

DW_stack is a single-clock cycle stack. When the previous operation (at the previous rising edge of `clk`) ends, both internal address pointers are adjusted immediately, and are pointing at the proper locations for the next operation in the next clock cycle. Refer to Table 1-7 on page 3 for internal address pointer information.

For example, assume that a pop occurred in the previous clock cycle. After a transient internal delay, the stack internal read address pointer points to the data in the next lower stack address location. The output data is ready for "pre-fetching" before the next `clk`. Meanwhile, the internal write pointer points to the address location of the data that was just read out.

Read Errors

The `error` output is activated if a pop is requested and the stack is empty. That is, if:

- The `pop_req_n` input is active (LOW), and
- The empty flag is active (HIGH)

at the rising edge of `clk`.

Simultaneous Push and Pop

DW_stack does not support simultaneous push and pop. If a push and pop occur at the same time when DW_stack is not full, only the push occurs, not the pop. DW_stack does not give an error. However, with the stack full, DW_stack activates the `error` output (due to overflow), and does not push. Also refer to Table 1-6 on page 3.

Reset

rst_mode
The `rst_mode` parameter selects whether the DW_stack reset is asynchronous (`rst_mode` = 0 or 2) or synchronous (`rst_mode` = 1 or 3). If an asynchronous mode is selected, asserting `rst_n` (setting it LOW) immediately causes the internal address pointers to be set to 0, and the flags and `error` output to be initialized. If a synchronous mode is selected, the address pointers, flags, and `error` output are initialized at the rising edge of `clk` following the assertion of `rst_n`.

The `error` output and flags are initialized as follows:

- The `full` flag and the `error` output are initialized to 0.

If `rst_mode` = 0 or 1, the RAM array is also initialized when `rst_n` is asserted. If `rst_mode` = 2 or 3, only the internal address pointers, and `error` and flag outputs are initialized; the RAM array is not initialized.

Errors

err_mode
The `err_mode` parameter determines whether the `error` output remains active until reset or for only the clock cycle in which the error is detected.

When `err_mode` = 0, overflow and underflow are detected, and the `error` output (once activated) remains active until reset. When `err_mode` = 1, overflow and underflow are detected, and the `error` output (once activated) remains active only for the clock cycle in which the `error` is detected. Refer to Table 1-5 on page 3 for error mode descriptions.

error
The `error` output indicates a fault in the operation of the stack control logic. There are two possible causes for the `error` output to be activated:

1. Overflow (push while full).
2. Underflow (pop while empty).

The `error` output is set LOW when `rst_n` is applied.

Controller Status Flag Outputs

empty
The `empty` output indicates that there are no words in the stack available to be popped. The `empty` output is set HIGH when `rst_n` is applied.

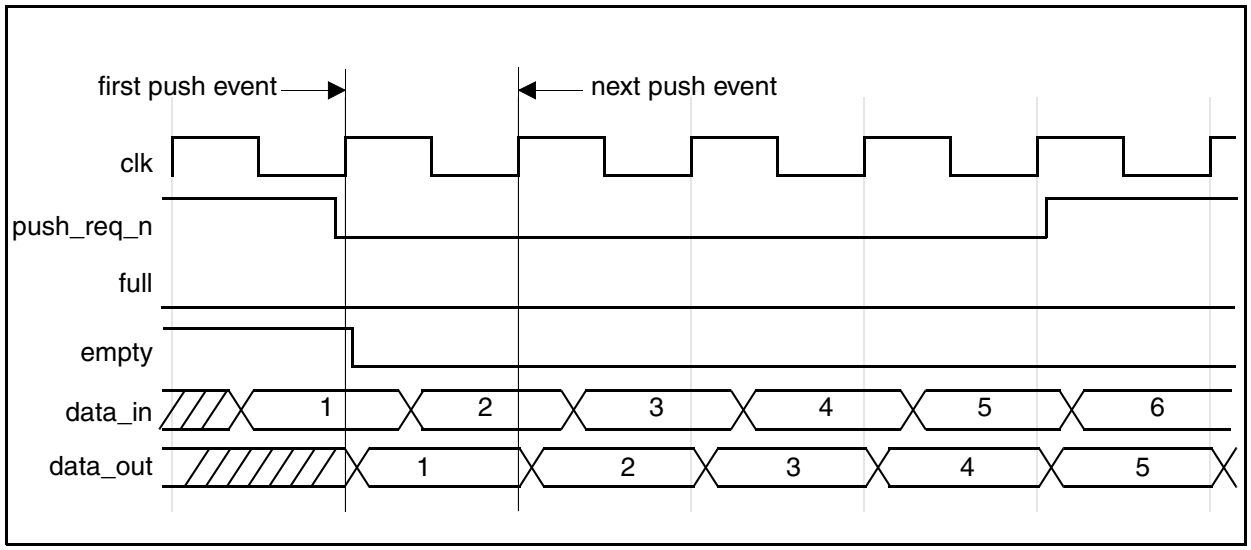
full
The `full` output indicates that the stack is full, and there is no space available for push data. The `full` output is set LOW when `rst_n` is applied.

Timing Waveforms

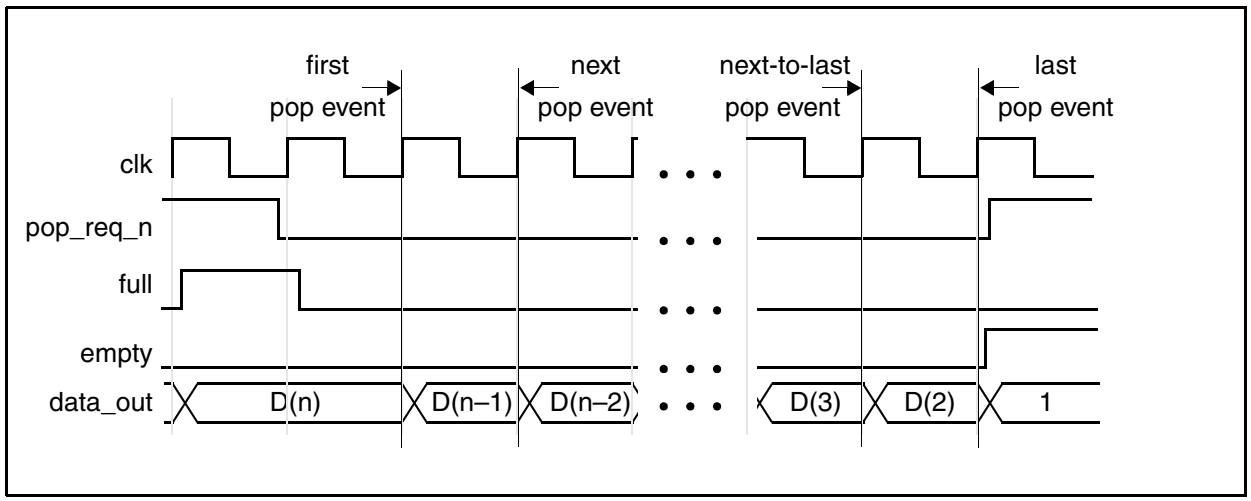
The following figure shows timing diagrams for various conditions of DW_stack.

Figure 1-2 Timing Waveforms

Initial Push Operation



Pop Operation



Related Topics

- Memory – Stacks Listing
- DesignWare Building Bock IP Documentation Overview

HDL Usage Through Component Instantiation - VHDL

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

entity DW_stack_inst is
  generic (inst_width    : INTEGER := 8;
           inst_depth    : INTEGER := 8;
           inst_err_mode : INTEGER := 0;
           inst_rst_mode : INTEGER := 0  );
  port (inst_clk       : in std_logic;
        inst_rst_n     : in std_logic;
        inst_push_req_n: in std_logic;
        inst_pop_req_n : in std_logic;
        inst_data_in   : in std_logic_vector(inst_width-1 downto 0);
        empty_inst     : out std_logic;
        full_inst      : out std_logic;
        error_inst     : out std_logic;
        data_out_inst  : out std_logic_vector(inst_width-1 downto 0) );
end DW_stack_inst;

architecture inst of DW_stack_inst is
begin

  -- Instance of DW_stack
  U1 : DW_stack
    generic map (width => inst_width,   depth => inst_depth,
                 err_mode => inst_err_mode,   rst_mode => inst_rst_mode )
    port map (clk => inst_clk,   rst_n => inst_rst_n,
              push_req_n => inst_push_req_n,   pop_req_n => inst_pop_req_n,
              data_in => inst_data_in,   empty => empty_inst,
              full => full_inst,   error => error_inst,
              data_out => data_out_inst );
end inst;

-- pragma translate_off
configuration DW_stack_inst_cfg_inst of DW_stack_inst is
  for inst
  end for; -- inst
end DW_stack_inst_cfg_inst;
-- pragma translate_on
```

HDL Usage Through Component Instantiation - Verilog

```
module DW_stack_inst(inst_clk, inst_rst_n, inst_push_req_n, inst_pop_req_n,
                     inst_data_in, empty_inst, full_inst, error_inst,
                     data_out_inst );
  parameter width = 8;
  parameter depth = 8;
  parameter err_mode = 0;
  parameter rst_mode = 0;

  input inst_clk;
  input inst_rst_n;
  input inst_push_req_n;
  input inst_pop_req_n;
  input [width-1 : 0] inst_data_in;
  output empty_inst;
  output full_inst;
  output error_inst;
  output [width-1 : 0] data_out_inst;

  // Instance of DW_stack
  DW_stack #(width, depth, err_mode, rst_mode)
    U1 (.clk(inst_clk),   .rst_n(inst_rst_n),   .push_req_n(inst_push_req_n),
        .pop_req_n(inst_pop_req_n),   .data_in(inst_data_in),
        .empty(empty_inst),   .full(full_inst),   .error(error_inst),
        .data_out(data_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