CHAPTER 3 CSL FIFO

TABLE 3.1 Chapter Overview

3.1 CSL FIFO Command Summary
3.2 CSL FIFO Commands

3.1 CSL FIFO Command Summary

3.1.1 Usage tables

CSL Fifo
can be defined and instantiated
- can be defined in

TABLE 3.2 CSL unit definition in other CSL classes

CSL class	Can be defined in
global scope	YES
CSL Unit	-
CSL Signal Group	-
CSL Interface	-
CSL Testbench	-
CSL Vector	-
CSL State Data	-
CSL Register	-
CSL Register File	-
CSL Fifo	-
CSL Memory Map	-
CSL Memory Map Page	-
CSL Isa Element	-
CSL Isa Field	-
CSL Field	-

- can be instantiated in

TABLE 3.3 CSL unit instantiation in other CSL classes

CSL class	Can be instantiated in
global scope	-
CSL Unit	YES
CSL Signal Group	-
CSL Interface	-
CSL Testbench	YES
CSL Vector	-

TABLE 3.3 CSL unit instantiation in other CSL classes

CSL class	Can be instantiated in
CSL State Data	-
CSL Register	-
CSL Register File	-
CSL Fifo	-
CSL Memory Map	-
CSL Memory Map Page	-
CSL Isa Element	-
CSL Isa Field	-
CSL Field	-

CSL FIFO mandatory commands

```
set_width(numeric_expression);
set_depth(numeric_expression);
```

CSL FIFO Architecture

```
add_logic(programmable_depth, default_depth);
set_physical_implementation(sram | ffa);
add_logic(priority_bypass);
add_logic(sync_fifo|async_fifo);
add_logic(depth_extend,numeric_expression);
add_logic(width_extend,numeric_expression);
add_logic(wr_hold, numeric_expession);
set_prefix(prefix_name);
```

CSL FIFO Data

```
add_logic(parallel_output, all | lower_address,upper_address);
add_logic(parallel_input,all | lower_address,upper_address);
add_logic(rd_words,address_range);
add_logic(wr_words,address_range);
add_logic(sram_rd);
add_logic(sram_wr);
add_logic(async_reset);
```

CSL FIFO Control

```
add_logic(pushback);
add_logic(flow_through, numeric_expression);
add_logic(stall);
add_logic(stall_rd_side);
add_logic(stall_wr_side);
```

```
add_logic(wr_release);
```

CSL FIFO Status

```
add_logic(almost_empty,address);
add_logic(almost_full,address);
add_logic(output_wr_addr);
add_logic(output_rd_addr);
add_logic(credit);
add_logic(rd_credit);
add_logic(wr_credit);
add_logic(flow);
```

CSL FIFO Custom Port naming

```
set_reset_name(string);
set_clock_name(string);
set_rd_clock_name(string);
set_wr_clock_name(string);
set_push_name(string);
set_pop_name(string);
set_full_name(string);
set_empty_name(string);
set_wr_data_name(string);
set_rd_data_name(string);
set_valid_name(string);
```

The following are the ports automatically generated when a csl_fifo is instantiated:
//move signals that don't have examples here

3.2 CSL FIFO Commands

set_width(numeric_expression);

DESCRIPTION :

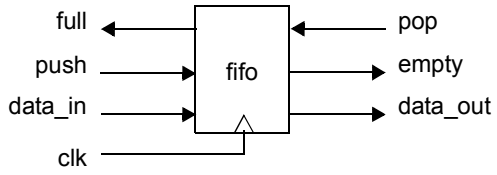
Creates a FIFO named *fifo_name.* Use a read and write pointer and full and empty signal to control the fifo**.** The param width and depth there are numeric expresion and are mandatory for FIFO.

[CSL FIFO Command Summary]

EXAMPLE :

In this example it is created a fifo named *fifo_name.*

FIGURE 3.1



CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_width(4);
set_depth(16);
}
};
```

VERILOG CODE

```
module fifo_name(data_out, empty, full, data_in, push, pop, clk,
reset);
    parameter stack_width=4;
    parameter stack_depth=16;
    parameter stack_ptr_width=4;

    input [stack_width-1: 0] data_in;
    input push, pop, reset, clk;
    output full, empty;
    output [stack_width-1:0] data_out;

    reg [stack_width-1:0] data_out;
    //gap between wr pointer and rd pointer
    reg [stack_ptr_width:0] gap;

    reg [stack_width-1:0] fifo[0:stack_depth-1];
    reg [stack_width-1:0] stack_widthr, rd;
```

```
assign empty=(gap==0);
assign full=(gap==stack_depth);
    always@(posedge clk or negedge reset)
  begin
      if(reset==0) begin
        data_out=0;
        gap=0;
        stack_widthr=0;
        rd=0;
      end
      else
        begin
            if(!full & push & !pop) begin
                fifo[stack_widthr]=data_in;
                stack_widthr=stack_widthr+1;
                gap=gap+1;
            end
            else if(!empty & pop & !push) begin
                data_out=fifo[rd];
                rd=rd+1;
                gap=gap-1;
            end
            else if(!full & pop & push) begin
                data_out=fifo[rd];
                fifo[stack_widthr]=data_in;
                rd=rd+1;
                stack_widthr=stack_widthr+1;
            end
            else if(empty & pop & push) begin
                fifo[stack_widthr]=data_in;
                stack_widthr=stack_widthr+1;
                gap=gap+1;
            end
            else if(full & pop & push) begin
                data_out=fifo[rd];
                rd=rd+1;
                gap=gap-1;
            end
        end
 end
```

```
endmodule
```

```
set_depth(numeric_expression);
```

DESCRIPTION :

Creates a FIFO named *fifo_name.* Use a read and write pointer and full and empty signal to control the fifo**.** The param depth is a numeric expresion and is mandatory for FIFO.

EXAMPLE :

//

CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_width(4);
set_depth(16);
}
};
```

add_logic(programmable_depth, *default_depth*);
port: input - programmable_depth

DESCRIPTION :

The FIFO depth is controlled by an input to the fifo. The write and read pointers are reseted when their value equals the value of *signal_name.* The *default_depth* is the maximum depth for the fifo, and the maximum value that *signal_name* can have.

[CSL FIFO Command Summary]

EXAMPLE :

Create a fifo that uses *prg_depth* input signal to controll the depth of the fifo. //desen

CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_depth(16);
//size_sgn input signal controlls the depth of the fifo
add_logic(programable_depth,1);
set_width(4);}
};
```

VERILOG CODE

```
module fifo_name(data_out, empty, full, data_in, prg_depth, push, pop,
clk, reset);
    parameter stack_width=4;
    parameter stack_depth=16;
    parameter stack_ptr_width=4;
    input [stack_width-1: 0] data_in;
    input push, pop, reset, clk;
    input [stack_ptr_width:0] prg_depth;
    output full, empty;
    output [stack_width-1:0] data_out;
    reg [stack_width-1:0] data_out;
    reg [stack_ptr_width:0] gap;
    reg [stack_width-1:0] fifo[0:stack_depth-1];
    reg [stack_ptr_width-1:0] wr, rd;
    assign empty=(gap==0);
    assign full=(gap==stack_depth);
    always@(posedge clk or negedge reset)
      begin
        if(reset==0) begin
          data_out=0;
          gap=0;
          wr=0;
```

```
            rd=0;
        end
        else
          begin
              if(!full & push & !pop) begin
                  fifo[wr]=data_in;
                  wr=wr+1;
                  gap=gap+1;
              end
              else if(!empty & pop & !push) begin
                  data_out=fifo[rd];
                  rd=rd+1;
                  gap=gap-1;
              end
              else if(!full & pop & push) begin
                  data_out=fifo[rd];
                  fifo[wr]=data_in;
                  rd=rd+1;
                  wr=wr+1;
              end
              else if(empty & pop & push) begin
                  fifo[wr]=data_in;
                  wr=wr+1;
                  gap=gap+1;
              end
              else if(full & pop & push) begin
                  data_out=fifo[rd];
                  rd=rd+1;
                  gap=gap-1;
              end
              if(rd==(prg_depth-1)) begin
                  rd=0;
                  $display("rd= %d - 1", prg_depth);
              end
              if(wr==(prg_depth-1)) begin
                  wr=0;
                  $display("wr= %d - 1", prg_depth);
              end
          end
    end
```

```
endmodule
```

set_physical_implementation(*sram | ffa*);

DESCRIPTION :

FIFO implementation type which is either SRAM or flip flop array.

[CSL FIFO Command Summary]

EXAMPLE :

Sets FIFO implementation which is SRAM.

CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_depth(16);
set_width(4);
set_physical_implementation(sram);
}
};
```

VERILOG CODE

```
//verilog code goes here
```

```
add_logic(priority_bypass);
   port: input - priority_select
   port: output - priority_bypass
```

DESCRIPTION :

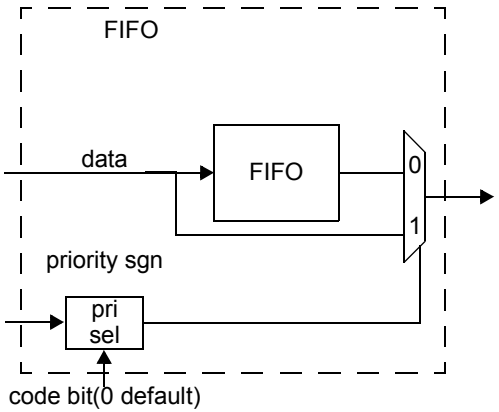
This signal is asserted at the same time as a write to the fifo. When asserted the write to the FIFO is sent to the input of the high prioriy bypass unit.

[CSL FIFO Command Summary]

EXAMPLE :

Adds a priority bypass to the output port of a fifo.

FIGURE 3.2 Bypass



CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_depth(16);
set_width(4);
add_logic(priority_bypass);
}
};
```

VERILOG CODE

```
//verilog code goes here
```

add_logic(sync_fifo|async_fifo);

DESCRIPTION :

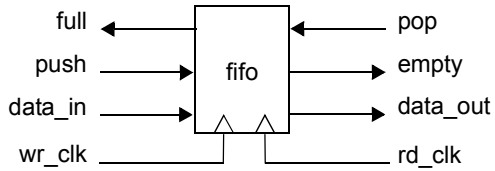
Generate an asynchronous FIFO architecture. Requires the *fifo_name* .wr/rd_clk signals be specified. The read and write counters will be clocked by their respective clocks.
The read and write counters will be grey coded to avoid glitches.
The read and write counters will be synchronized to the other clock domain prior to being compared to the opposite counter.

[CSL FIFO Command Summary]

EXAMPLE :

In this example is generated an asynchronous FIFO architecture.

FIGURE 3.3



CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_depth(16);
set_width(4);
add_logic(async_fifo);
}
};
```

VERILOG CODE

```
//verilog code goes here
```

add_logic(depth_extend,*numeric_expression***);**

DESCRIPTION :

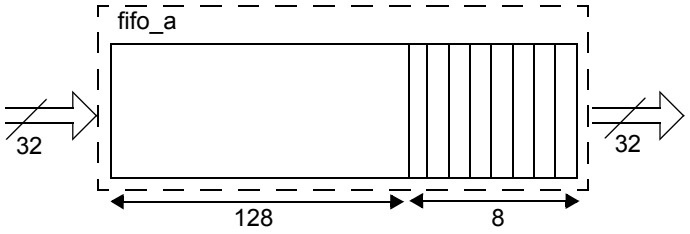
Extend the depth of the FIFO by adding a number of registers to the fifo. The registers or FIFO must be the same width as initial fifo.

[CSL FIFO Command Summary]

EXAMPLE :

Extend *fifo_a* by adding 8 registers. The final depth of *fifo_a* is 136.

FIGURE 3.4 FIFO depth extension



CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_depth(128);
set_width(32);
//connect 8 registers to the out of fifo_name
add_logic(depth_extend,8);
}
};
```

VERILOG CODE

```
//verilog code goes here
```

add_logic(width_extend,*numeric_expression***);**

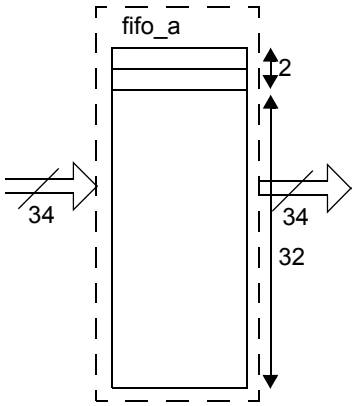
DESCRIPTION :

Extend the width of the FIFO by adding a number of registers FIFO .The registers must be the same depth as initial fifo.

[CSL FIFO Command Summary]

EXAMPLE :

Extend *fifo_a* by adding 2 registers. The final width of *fifo_a* is 34.



CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_depth(128);
set_width(32);
//connect 2 registers to the out of fifo_name
add_logic(width_extend,2);
}
};
```

VERILOG CODE

```
//verilog code goes here
```

add_logic(wr_hold, *numeric_expression*);

DESCRIPTION :

Generate a FIFO with the write hold architecture.

Now we get into the rather esoteric parts of FIFO architeture, design, and implementation. Values may be written by the write side of the FIFO using a push signal. The values are not available to the read of side of the FIFO until the read pointer is allowed to increment into the section of the FIFO memory which contains the write values. In effect we can have a signal which is required after one or more write operations which allows the read pointer to increment into the newly written write region. In other words a write to a FIFO is initiated by a push. The read pointer can access the value written if the < fifo_name > write_hold flag is received. The wr_hold flag will increment a read limit counter which controls the limit count that the read counter can increment to. The wr_addr_hold_limit controls the empty flag. If the FIFO is empty then the pop operation is disabled.

[CSL FIFO Command Summary]

EXAMPLE :

Create a fifo architecture that requires 5 elements in the FIFO between it can be read.

FIGURE 3.5 Write hold architecture

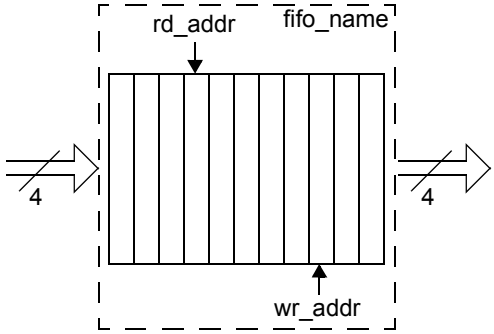


TABLE 3.4 Write hold architecture example

wr_addr	15	16	17	18	19	20	20	21	22	23	23	23	23	23	23	23	23
rd_addr	15	15	15	15	15	15	16	16	16	16	17	18	19	20	21	22	23
rd_wr_gap	0	1	2	3	4	5	4	5	6	7	6	5	4	3	2	1	0
can_pop	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	0

CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_depth(32);
set_width(4);
add_logic(wr_hold,5);
}
```

```
};
```

VERILOG CODE

```
wire [ADDR_MAX-1:0] rd_addr;
wire [ADDR_MAX-1:0] wr_addr;
wire [ADDR_MAX-1:0] wr_addr_hold_limit     ; // the address of the memory location which was written
wire  [ADDR_MAX-1:0] wr_addr_release_limit; // the address of the current upper limit of released memory locations
wire empty = (rd_addr -wr_addr -1) && (rd_addr !=  wr_addr_hold_limit - 1);
if (wr_release) begin
    wr_addr_hold_limit <=  wr_addr_release_limit;
end
wire rd_addr_inc = pop &&  (rd_addr  <= wr_addr_hold_limit);
```

set_prefix(*prefix_name***);**

DESCRIPTION :

Sets the prefix of all the FIFO signals with *prefix_name*.

EXAMPLE :

Sets the prefix “fifo_a” for the signals of FIFO *fifo_a*.

CSL CODE

```
csl_fifo fifo_a{
fifo_a(){
set_depth(32);
set_width(4);
set_prefix_name("fifo_a.");
}
};
```

VERILOG CODE

```
//verilog code goes here
```

add_logic(parallel_output, all | *lower_address,upper_address*);

DESCRIPTION :

Connects the specified FIFO words to individual outputs of the FIFO.

The FIFO read side is an SRAM interface. The FIFO can be read in any order.

[CSL FIFO Command Summary]

EXAMPLE :

In this example are connected all FIFO words to individual outputs.

FIGURE 3.6 FIFO with SRAM read side

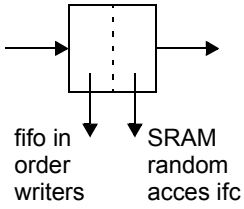
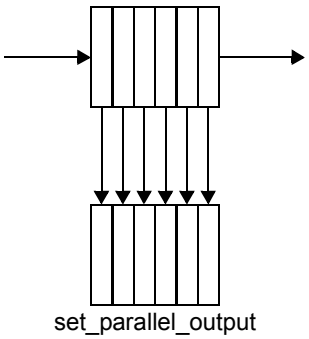


FIGURE 3.7 Paralel output FIFO



CSL CODE

```
csl_fifo fifo_a{
fifo_a(){
set_width(24);
set_depth(4);
set_parallel_output(all);
}
};
```

VERILOG CODE

```
//verilog code goes here
```

add_logic(parallel_input,all | *lower_address,upper_address*);

DESCRIPTION :

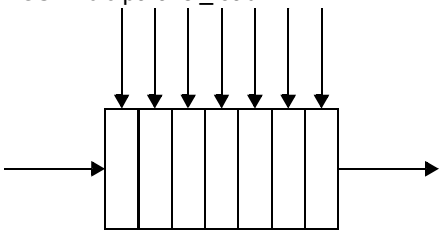
Connects individual inputs of the FIFO to the specified FIFO words.

[CSL FIFO Command Summary]

EXAMPLE :

In this example are connected all FIFO words to individual inputs.

FIGURE 3.8 parallel_load



CSL CODE

```
csl_fifo fifo_a{
fifo_a(){
set_width(24);
set_depth(4);
set_parallel_input(all);
}
};
```

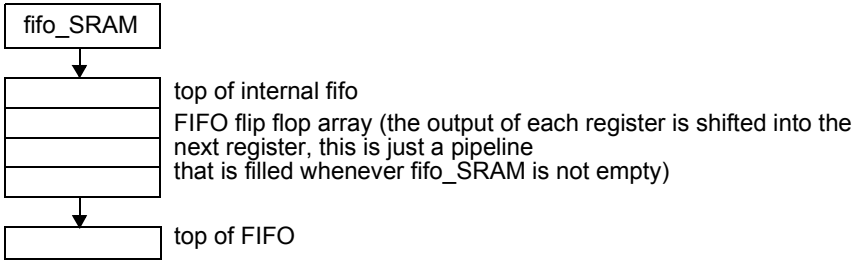
VERILOG CODE

```
//verilog code goes here
```

add_logic(rd_words,*address_range*);

DESCRIPTION :

Connects the FIFO words in the FIFO address range to the interface of the block.
If the architecture type is specified as SRAM then the FIFO words in the address_range are implemented as flip flops. The address range can be up to n (figure out what n can be) words which are implemented as FF's.All words in FIFO are connected to the FIFO block interface.
Connect top words



[CSL FIFO Command Summary]

EXAMPLE :

The words from the fifo address range [0:29] are connected to the interface of the block.

CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_depth(8);
set_width(32);
add_logic(rd_words,0,29);
}
};
```

VERILOG CODE

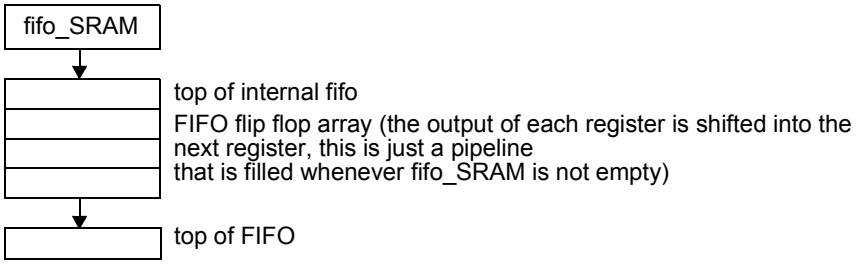
```
//verilog code goes here
```

add_logic(wr_words,*address_range*);

DESCRIPTION :

Connects the FIFO words in the FIFO address range to the interface of the block.
If the architecture type is specified as SRAM then the FIFO words in the address_range are implemented as flip flops. The address range can be up to n (figure out what n can be) words which are implemented as FF's.

FIGURE 3.9



All words in FIFO are connected to the FIFO block interface.

[CSL FIFO Command Summary]

EXAMPLE :

The words from the fifo address range [0:32] are connected to the interface of the block.

CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_depth(8);
set_width(10);
add_logic(wr_words,0,32);
}
};
```

VERILOG CODE

```
//verilog code goes here
```

```
add_logic(sram_rd);
   port: input - <sram_rd>_en
   port: input - <sram_rd>_addr
   port: output - <sram_rd>_data
```

DESCRIPTION :

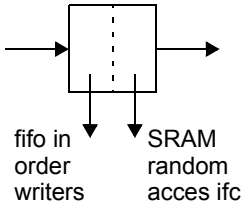
The FIFO read side is an SRAM interface. The FIFO can be read in any order.

[CSL FIFO Command Summary]

EXAMPLE :

Sets an SRAM interface for a FIFO read side.

FIGURE 3.10 FIFO with SRAM read side



CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_depth(2);
set_width(4);
add_logic(sram_rd);
}
};
```

VERILOG CODE

```
//verilog code goes here
```

NOTE: add the options for the SRAM read signals which are the same as the CSL SRAM read signals

```
add_logic(sram_wr);
   port: input - <sram_wr>_addr
   port: input - <sram_wr>_data
   port: input - <sram_wr>_en
```

DESCRIPTION :

The FIFO write side is an SRAM interface. The FIFO can be written in any order.

[CSL FIFO Command Summary]

EXAMPLE :

Sets an SRAM interface for a FIFO read side.

CSL CODE

```
   csl_fifo fifo_name {
   fifo_name(){
   set_depth(2);
   set_width(4);
   add_logic(sram_wr);
   }
   };
```

VERILOG CODE

```
   //verilog code goes here
```

NOTE: add the options for the SRAM write signals which are the same as the CSL SRAM write signals

```
add_logic(async_reset);
   port: input - async_reset
```

DESCRIPTION :

This is an asynchronous reset command. When the reset signal is on the low level (logic “0“) the fifo is filled up with 0 values, the clock signal edge doesn’t matter.

[CSL FIFO Command Summary]

EXAMPLE :

Adds an asynchonous reset for the FIFO named *fifo_name*.

CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_depth(2);
set_width(4);
add_logic(async_reset);
}
};
```

VERILOG CODE

```
//verilog code goes here
```

```
add_logic(pushback);
   port: input - pushback
```

DESCRIPTION :

Push back the value popped off of the top of the fifo. Don't really push it back. Instead move the read pointer back one position. This feature requries that the full and empty signals are generated from logic that seperates the write and the read pointers by at least one postion to allow the push_back operation to change the read pointer.

[CSL FIFO Command Summary]

EXAMPLE :

In this example the read pointer is pushed back in a fifo named *fifo_name.*

CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_depth(8);
set_width(10);
add_logic(pushback);
}
};
```

VERILOG CODE

```
//verilog code goes here
```

add_logic(flow_through, *numeric_expression*);

DESCRIPTION :

The FIFO is popped automatically whenever there are at least *numeric_expression* valid words in the FIFO. The FIFO can be controlled by the stall signal.

[CSL FIFO Command Summary]

EXAMPLE :

In this example the FIFO is popped automatically when there are at least 9 valid words in the FIFO named *fifo_name*.

CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_depth(8);
set_width(10);
add_logic(flow_through,9);
}
};
```

VERILOG CODE

```
//verilog code goes here
```

```
add_logic(stall);
   port: input - stall
```

DESCRIPTION :

The entire FIFO is stalled and pop/push operations are not allowed. This signal is used for flow through fifos which pop themselves whenever the FIFO is not empty.

[CSL FIFO Command Summary]

EXAMPLE :

In this example the FIFO named *fifo_name* is stalled.

CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_depth(8);
set_width(10);
add_logic(stall);
}
};
```

VERILOG CODE

```
//verilog code goes here
```

```
add_logic(stall_rd_side);
   port: input - stall_rd_side
```

DESCRIPTION :

The read side of the FIFO is stalled and FIFO pop operations are not allowed.

[CSL FIFO Command Summary]

EXAMPLE :

For the FIFO named *fifo_name*, the read side is stalled.

CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_depth(8);
set_width(10);
add_logic(stall_rd_side);
}
};
```

VERILOG CODE

```
//verilog code goes here
```

```
add_logic(stall_wr_side);
   port: input - stall_wr_side
```

DESCRIPTION :

The write side of the FIFO is stalled and FIFO push operations are not allowed .

[CSL FIFO Command Summary]

EXAMPLE :

For the FIFO named *fifo_name*, the write side is stalled.

CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_depth(8);
set_width(10);
add_logic(stall_wr_side);
}
};
```

VERILOG CODE

```
//verilog code goes here
```

```
add_logic(wr_release);
   port: input - wr_release
```

DESCRIPTION :

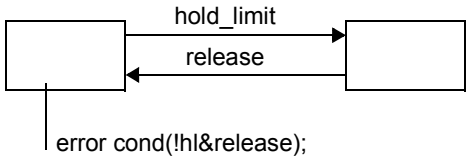
This will set the wr_addr_release_limit register equal to the current wr_addr_hold_limit used in conjuction with the wr_hold_arch switch.

[CSL FIFO Command Summary]

EXAMPLE :

Sets the write address released for the FIFO *fifo_name.*

FIGURE 3.11 Write release
1.hold data
2.release data



CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_depth(8);
set_width(10);
add_logic(wr_release);
}
};
```

VERILOG CODE

```
//verilog code goes here
```

```
add_logic(almost_empty,address);
   port: output - almost_empty
```

DESCRIPTION :

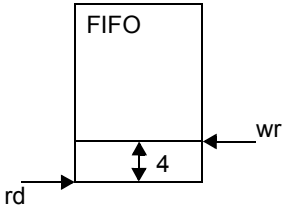
An almost_empty signal is generated when the FIFO almost empty address is reached.
If the FIFO is asynchronous then the signal is generated in the read clock domain.
Note that there is a delay of n cycles, where n is the synchronization delay of the FIFO write address, until the almost_empty signal is generated.
Optionally use *name* as the name of the almost_empty signal.

[CSL FIFO Command Summary]

EXAMPLE :

Sets an almost empty signal for a FIFO named *fn.*

FIGURE 3.12



CSL CODE

```
csl_fifo fn{
fn(){
set_depth(32);
set_width(4);
//add optional FIFO signals
add_logic(almost_empty, 4);
}
};
```

VERILOG CODE

```
//verilog code goes here
```

add_logic(almost_full,*address*);
port: output - almost_full

DESCRIPTION :

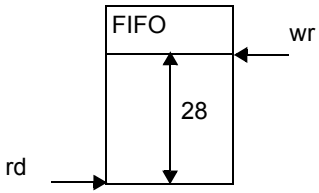
A almost full signal is generated when the FIFO almost full address is reached
If the FIFO is asynchronous then then the signal is generated in the read clock domain.
Note that there is a delay of n cycles, where n is the synchronization delay of the FIFO write address, until the almost_full signal is generated.
Optionally use *name* as the name of the almost_full signal.

[CSL FIFO Command Summary]

EXAMPLE :

Sets an almost full signal for a FIFO named *fn.*

FIGURE 3.13 Almost full



CSL CODE

```
csl_fifo fn{
fn(){
set_depth(32);
set_width(4);
//add optional FIFO signals
add_logic(almost_full,28);
}
};
```

VERILOG CODE

```
//verilog code goes here
```

```
add_logic(output_wr_addr);
   port: output - wr_addr
```

DESCRIPTION :

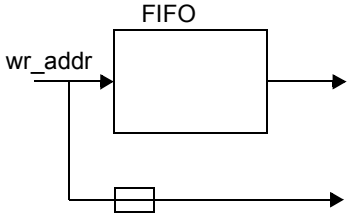
The FIFO write address is available at the FIFO interface.

[CSL FIFO Command Summary]

EXAMPLE :

Sets the output write address for the FIFO *fifo_name.*

FIGURE 3.14 Status write address



CSL CODE

```
    csl_fifo fifo_name {
    fifo_name(){
    set_depth(16);
    set_width(32);
    add_logic(output_wr_addr);
    }
    };
```

VERILOG CODE

```
    //verilog code goes here
```

```
add_logic(output_rd_addr);
   port: output - rd_addr
```

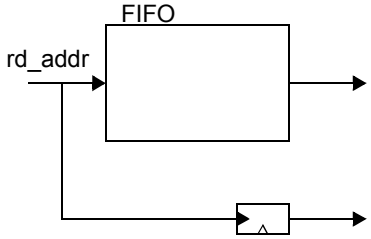
DESCRIPTION :

The FIFO read address is available at the FIFO interface.

EXAMPLE :

Sets the output read address for the FIFO *fifo_name*

FIGURE 3.15 Status read address



CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_depth(16);
set_width(32);
add_logic(output_rd_addr);
}
};
```

VERILOG CODE

```
//verilog code goes here
```

```
add_logic(credit);
   port: output - credit
```

DESCRIPTION :

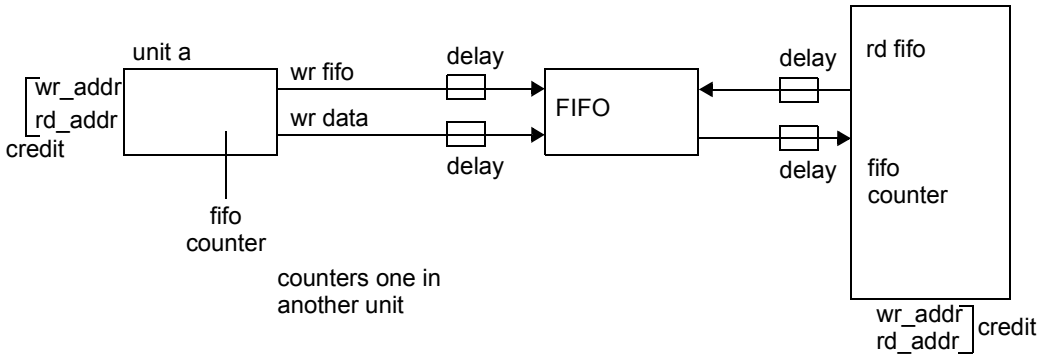
Use a distributed credit debit mechanism to control the fifo. No full or empty signals are generated. Instead FIFO write status/control is handled by the producer and the FIFO read status and control is handled by the consumer.

[CSL FIFO Command Summary]

EXAMPLE :

Adds a credit debit mechanism to control a FIFO named *fifo_name.*

FIGURE 3.16 Credit debit mechanism



CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_depth(16);
set_width(32);
add_logic(credit);
}
};
```

VERILOG CODE

```
//verilog code goes here
```

```
add_logic(rd_credit);
   port: output - rd_credit
```

DESCRIPTION :

Use a distributed credit debit mechanism to cotnrol the FIFO read. No full or empty signals are generated. Instead fifo write status/control is handled by the producer and the fifo read status and control is handled by the consumer.

[CSL FIFO Command Summary]

EXAMPLE :

Adds a credit debit mechanism to control a FIFO read named *fifo_name.*

CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_depth(16);
set_width(32);
add_logic(rd_credit);
}
};
```

VERILOG CODE

```
//verilog code goes here
```

```
add_logic(wr_credit);
   port: output - wr_credit
```

DESCRIPTION :

Use a distributed credit debit mechanism to cotnrol the FIFO write. No full or empty signals are generated. Instead fifo write status/control is handled by the producer and the fifo read status and control is handled by the consumer.

[CSL FIFO Command Summary]

EXAMPLE :

Adds a credit debit mechanism to control a FIFO write named *fifo_name*

CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_depth(16);
set_width(32);
add_logic(wr_credit);
}
};
```

VERILOG CODE

```
//verilog code goes here
```

```
add_logic(flow);
   port: output - overflow
   port: output - underflow
```

DESCRIPTION :

When the fifo is full and the upstream device sends a push request, the overflow signal will be validated so that the upstream device will know that it can not push more data into the fifo.
When the fifo is not full the underflow signal is valid.

[CSL FIFO Command Summary]

EXAMPLE :

In this example are activated the flow signals for a FIFO named *fifo_name.*

FIGURE 3.17

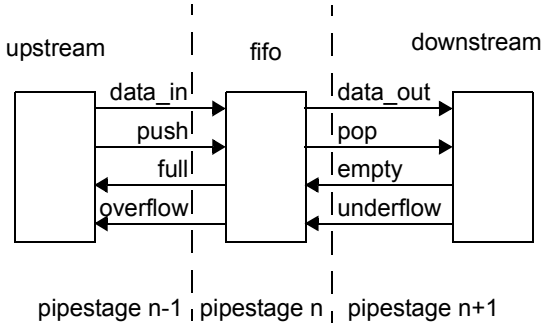


FIGURE 3.18

overflow	output	1	add_logic(flow);
underflow	output	1	add_logic(flow);

CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_depth(16);
set_width(32);
add_logic(flow);
}
};
```

VERILOG CODE

```
//verilog code goes here
```

set_reset_name(string);

DESCRIPTION :

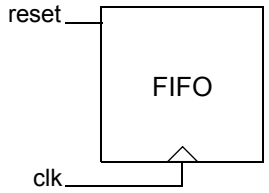
It sets the *name* for the reset port of fifo.

[CSL FIFO Command Summary]

EXAMPLE :

Sets the name “rst” for reset port of a FIFO named *FO*.

FIGURE 3.19



CSL CODE

```
csl_fifo FO{
FO(){
set_width(32);
set_depth(4);
set_reset_name("rst");
}
};
```

VERILOG CODE

set_clock_name(string);

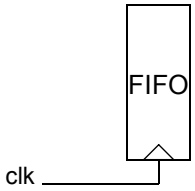
DESCRIPTION :

It sets the *name* for the clock port of fifo.

EXAMPLE :

Sets the name “clk” for clock port of a FIFO named *fifo_name*.

FIGURE 3.20



CSL CODE

```
csl_fifo fifo_name{
fifo_name(){
set_width(32);
set_depth(4);
set_clock_name("clk");
}
};
```

VERILOG CODE

```
set_rd_clock_name(string);
```

DESCRIPTION :

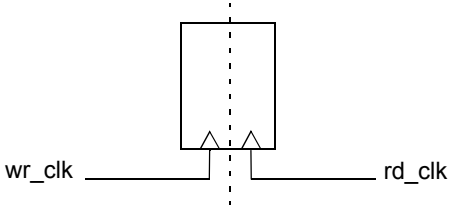
It sets the *name* for the read clock port of fifo.

[CSL FIFO Command Summary]

EXAMPLE :

Sets the name “rd_clk” for the read clock port of FIFO *fifo_name*.

FIGURE 3.21



CSL CODE

```
csl_fifo fifo_name{
fifo_name(){
set_width(32);
set_depth(4);
set_rd_clock_name("rd_clk");
}
};
```

VERILOG CODE

set_wr_clock_name(string);

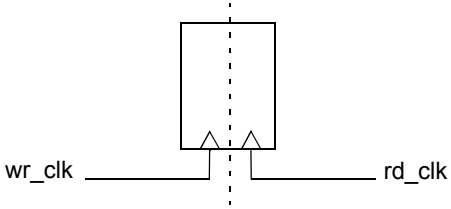
DESCRIPTION :

It sets the *name* for the write clock port of fifo.

EXAMPLE :

Sets the name “wr_clk” for the write clock port of FIFO *fifo_name.*

FIGURE 3.22



CSL CODE

```
csl_fifo fifo_name{
fifo_name(){
set_width(32);
set_depth(4);
set_wr_clock_name("wr_clk");
}
};
```

VERILOG CODE

```
set_push_name(string);
```

DESCRIPTION :

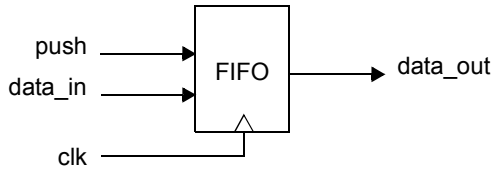
It sets the *name* for the push port of fifo.

[CSL FIFO Command Summary]

EXAMPLE :

Sets the name “push” for the push port of FIFO *fifo_name.*

FIGURE 3.23



CSL CODE

```
csl_fifo fifo_name{
f(){
set_width(32);
set_depth(4);
set_push_name("push");
}
};
```

VERILOG CODE

```
set_pop_name(string);
```

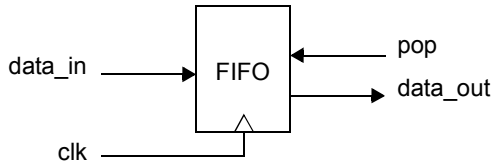
DESCRIPTION :

It sets the *name* for the pop port of fifo.

EXAMPLE :

Sets the name “pop” for the pop port of fifo *FO.*

FIGURE 3.24



CSL CODE

```
csl_fifo FO{
FO(){
set_width(32);
set_depth(4);
set_pop_name("pop");
}
};
```

VERILOG CODE

```
set_full_name(string);
```

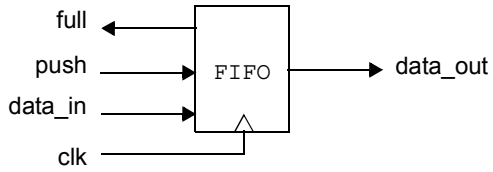
DESCRIPTION :

It sets the *name* for the full port of fifo.

EXAMPLE :

Sets the name “full” for the full port of fifo *FO.*

FIGURE 3.25



CSL CODE

```
csl_fifo FO{
FO(){
set_width(32);
set_depth(4);
set_full_name("full");
}
};
```

VERILOG CODE

set_empty_name(string);

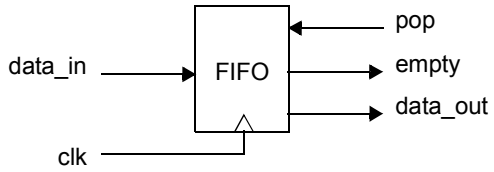
DESCRIPTION :

It sets the *name* for the empty port of fifo.

EXAMPLE :

Sets the name “empty” for the empty port of fifo *FO*.

FIGURE 3.26



CSL CODE

```
csl_fifo FO{
FO(){
set_width(32);
set_depth(4);
set_empty_name("empty");
}
};
```

VERILOG CODE

```
set_wr_data_name(string);
```

DESCRIPTION :

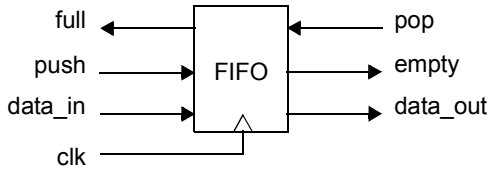
It sets the *name* for the wr_data port of fifo.

[CSL FIFO Command Summary]

EXAMPLE :

Sets the name “write_data” for the wr_data port of fifo *FO.*

FIGURE 3.27



CSL CODE

```
csl_fifo FO{
FO(){
set_width(32);
set_depth(4);
set_wr_data_name("write_data");
}
};
```

VERILOG CODE

set_rd_data_name(string);

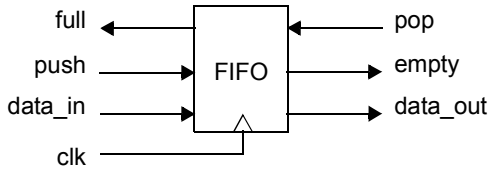
DESCRIPTION :

It sets the *name* for the rd_data port of fifo.

EXAMPLE :

Sets the name “read_data” for the rd_data port of fifo *FO.*

FIGURE 3.28



CSL CODE

```
csl_fifo FO{
FO(){
set_width(32);
set_depth(4);
set_rd_data_name("read_data");
}
};
```

VERILOG CODE

set_valid_name(string);

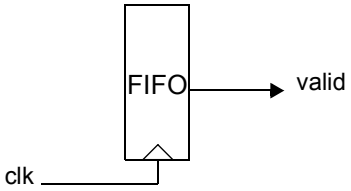
DESCRIPTION :

It sets the *name* for the valid port of fifo.

EXAMPLE :

Sets the name “valid” for the valid port of fifo *FO.*

FIGURE 3.29



CSL CODE

```
csl_fifo FO{
FO(){
set_width(32);
set_depth(4);
set_valid_name("valid");
}
};
```

VERILOG COD