Module 18

Coding and Synthesizing an Example Verilog Design

cadence

This page does not contain notes.

Module Objective

In this module, you:

- Construct a sample Verilog RTL design for synthesis

Topics

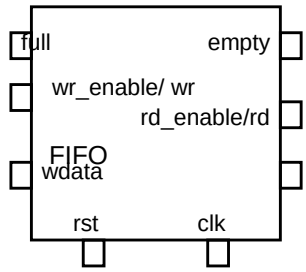
- FIFO specification
- FIFO implementation
  - Parameters
  - Ports
  - Variables
  - Functionality
  - Outputs
- Testbench design and implementation

cadence

This module applies the Verilog RTL for synthesis standards to construct a sample design.

FIFO Specification

- Parameterized for width and depth
  - Depth restricted to power of 2
- Asynchronous high-active reset
- Write/read synchronous to rising clock
- Ignores write when full
- Ignores read when empty



cadence

The sample design is a queue with a FIFO protocol, parameterized for width and depth, reset asynchronously and otherwise synchronized to a clock.

FIFO Implementation

Store FIFO contents in a register array.

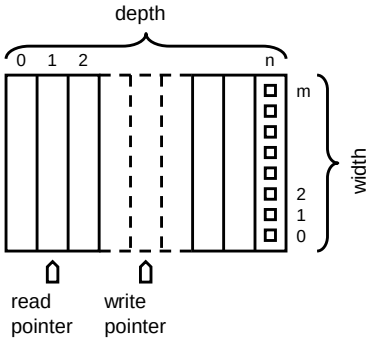
Access array with write/read pointers.

- For write:
  - Store data at current address
  - Increment pointer, maybe wrap
- For read:
  - Continuously drive current data
  - Increment pointer, maybe wrap

Derive the FIFO status from the pointer addresses.

Full – Addresses are equal and last operation was write.

Empty – Addresses are equal and last operation was read.



- Both pointers start at 0.
- Write pointer deposits value and increments.
- FIFO outputs data at read pointer.
- Read pointer increments to point at new data.
- Read pointer follows write pointer.
- Both pointers wrap around to array beginning.

Implement the FIFO by using a register array addressed by separate write and read pointers:

- Reset the write pointer to 0. Upon each write operation, if the FIFO is not already full, store the data at the current write address and increment the write pointer to point to the next location, wrapping back to the array beginning if necessary.
- Reset the read pointer to 0. Continuously present the array data at the current read address to the data output port. Upon each read operation, if the FIFO is not already empty, increment the read pointer to point to the next location, wrapping back to the array beginning if necessary.
- The read address thus follows the write address around the array. At any point where the read address is the same as the write address, you know that the FIFO is either empty or full. It is full if the most recent operation was to write and empty if the most recent operation was to read. Include a register to track whether the most recent operation was to write.

FIFO Module Structure

- Parameters
- Ports
- Variables
- Function

```
module fifo
#(
  // parameter declarations
 )
 (
  // port declarations
 );

  // variable declarations

  // functional code

endmodule
```


The basic building block of the design hierarchy is the Verilog module. Every module description starts with the module keyword followed by the module identifier (“fifo”) and ends with the endmodule keyword. The module definition name must be unique within the definitions name space.

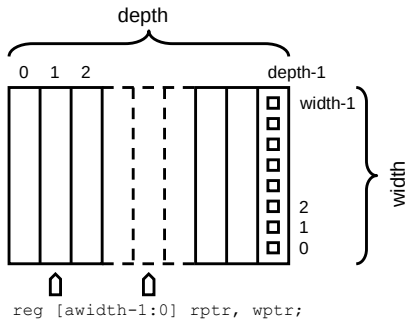
Following the module name is an optional module parameter port list, an optional list of ports or list of port declarations, and optional module items.

As for this design you will parameterized the input and output port width, you must declare the parameters before you declare the input and output ports of the module.

You must also declare all constants and variables before you use them.

FIFO Parameters

- Use the module **parameter** construct to define width and depth.
- Declare the parameters before you declare the ports.
- You can override module parameter values for each individual instance.



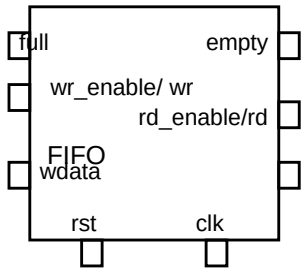

```
module fifo
#(
  // parameter declarations
  parameter awidth = 5,
  parameter dwidth = 8
 )
 (
  // port declarations
 );

  // variable declarations

  // functional code

endmodule
```

```
// 16 wide by 16 deep FIFO
fifo #(16,4) fifo16x16 (...
```


cadence

Use the module parameter construct to parameterize your module definition. You can override module parameter values on a per-instance basis. If you parameterize the input and output port widths, you must declare the parameters before you declare the input and output ports of the module. Module parameters are runtime constants that you can use wherever the syntax requires a literal value. You cannot change module parameters during the simulation.

FIFO Ports

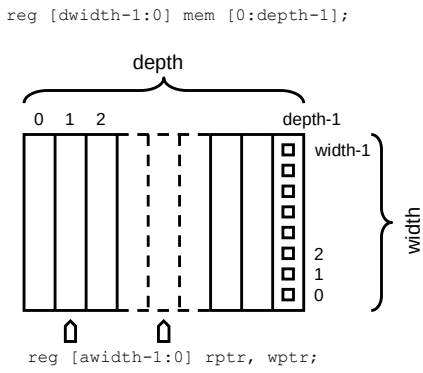



```
module fifo
#(
  // parameter declarations
  parameter AWIDTH = 5,
  parameter DWIDTH = 8
 )
 (
  // port declarations
  input clk, rst, wr_en, rd_en,
  input  [DWIDTH-1:0] data_in,
  output full, empty,
  output reg [DWIDTH-1:0] data_out
 );

  // variable declarations

  // functional code

endmodule
```


Utilize a module parameter to define the width of the data input and output ports. Parameterizing the data width allows you to modify it for each instance of the FIFO.

FIFO Variables

- Register array
- Write and read pointers
- Most recent operation type




```
module fifo
#(
  // parameter declarations
 )
 (
  // port declarations
 );

  // variable declarations
  localparam depth = 2**awidth;
  reg [dwidth-1:0] mem [0:depth-1];

  reg [awidth-1:0] wptr;
  reg [awidth-1:0] rptr;
  reg wrote;

  // functional code

endmodule
```

Verilog 2001
Power Operator

Utilize module parameters to define the width and depth of the register array and the width of the write address register and the read address register. Parameterizing the width and depth allows you to easily modify them for each instance of the FIFO.

FIFO Function

Asynchronous high-active reset

Write/read synchronous to rising clock

- For write:
  - Ignore write when full
  - Store data at current address
  - Increment pointer, maybe wrap
- For read:
  - Ignore read when empty
  - Increment pointer, maybe wrap

Track last operation

```
// functional code
always @(posedge clk or posedge rst)
begin
  if (rst)
    begin
      rptr <= 1'b0;
      wptr <= 1'b0;
      wrote <= 1'b0;
    end
  else
    begin
      if (rd && !empty)
         begin
           rdata <= mem[rptr];
           rptr <= rptr +1;
           wrote <= 0;
         end
      if (wr && !full)
         begin
           mem[wptr] <= wdata;
           wptr <= wptr +1;
           wrote <= 1;
         end
     end
end
```

356

cadence

Code a sequential procedure to describe the main functionality of the FIFO. The FIFO uses an asynchronous reset, so place the reset input as well as the clock input in the procedure event list. Remember that for synthesis all variables in an event list for a sequential procedure must be edge-qualified, so trigger the procedure by the positive edge of the reset and the positive edge of the clock.

While reset, initialize the write address and read address to location 0 and the tracking register to indicate that the most recent operation was to read.

Upon an active clock edge, check for a valid write or read operation:

- If the write enable is active and the FIFO is not full, store input data to the location of the current write pointer and increment the write pointer, wrapping back to the array beginning if necessary.
- If the read enable is active and the FIFO is not empty, increment the read pointer to address data in the next location, wrapping back to the array beginning if necessary.
- For this design, you can let the address registers wrap naturally. If the depth was not a power of 2 then you would need to explicitly wrap the address registers from their maximum value back to 0.

Upon an active clock edge, also update the tracking register.