# FIFO

**Free ware, 2021**
**Fen Logic Ltd.**

## Description

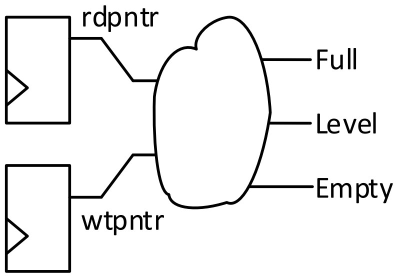
The FIFO directory contains a set of various FIFOs.

### Synchronous FIFOs:

(Writing and reading from the same clock domain).

**sync_fifo.v**
This is the FIFO I use in most of my designs. It has a separate register bank to keep track of the FIFO fill level. The empty and full flags can be derived from the level or they can be generated straight from registers (See Parameters: REGFLAGS).
The FIFO depth is always a power of two.

**sync_fifo1.v**
In this FIFO all status ports are derived from the read and write pointers:



This FIFO will be smaller than sync_fifo, but the status ports will be slower.

**sync_fifo2.v**
This FIFO is identical to the sync_fifo but the depth does not have to be a power of two. Do not use this with a depth of one. The user must specify the number of bits for the level with the LEVLBITS parameters. This is

```
LEVLBITS = ciel( log2( DEPTH ))      if DEPTH is not a power of two.
LEVLBITS = ciel( log2( DEPTH ))+1    if DEPTH is a power of two.
```

**sync_fifo_hd.v**
This FIFO is a data width conversion FIFO. The data width of the output is half that of the input. Thus for every write you can do two reads. For the rest this FIFO is a variant on the sync_fifo2.v. The ‘level’ indicates the number of units which can be read. The number of units written is half that (level>>1).
The ‘empty status indicates the FIFO is empty and thus if data can be read (empty=0) or not (empty=1).
The ‘full’ status indicates if data can be written (full=0) or not (full=1). It means that the FIFO is full or contains “full-1” units.

**sync_fifo_dd.v**
This FIFO is a data width conversion FIFO. The data width of the output is double that of the input. Thus for every two writes you can do one read. For the rest this FIFO is a variant on the sync_fifo2.v. The ‘level’ indicates the number of units which have been written. The number of units which can be read is half that (level>>1).
The ‘full’ status indicates if the FIFO is full, thus if data can be written (full=0) or not (full=1).

The ‘empty status indicates if the FIFO can be read (empty=0) or not (empty=1). The FIFO is flagged to be empty if the level is zero ***or one***.

## A-synchronous FIFOs:

(Writing and reading from a different clock domain).

### async_fifo.v

This the FIFO is full synthesizable. For timing the user must set a false-path to all “xxx_meta” registers.

- rd_meta (Dimension is [L2D:0])
- wt_meta (Dimension is [L2D:0])

### async_fifo_lib.v

This FIFO use library cells for the synchronisers. The library cell name is ‘lib_sync_cell’. You have to provide a wrapper which includes your library cell or replace the library cell name with your own type.
The test bench directory has a behavioural model for that cell. Beware that the library cell must contain TWO registers!

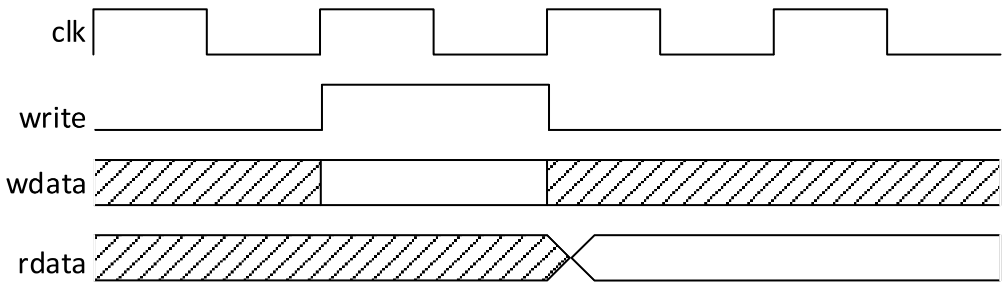
## Parameters

All FIFOs have parameters to control the data width and the FIFO depth.

- WIDTH : The data width in bits. For data conversion FIFOs it is the width of the smallest data port.
- L2D: The FIFO Depth in Log-2 data units. L2D=4 gives a FIFO with 2^4 = 16 entries.
- DEPTH: The FIFO Depth in data units. Depth = 48 gives a FIFO with 48 entries.
- LEVLBITS: The number of bits of the ‘level’ port. This is normally ceiling (log2 (DEPTH))+1.
- REGFLAGS: If set the full and empty flags come straight from registers.

## Fall-through.

All FIFOs are of the fall-through type. That is: The write data appears at the output without having to perform a read. The following is a diagram of this behaviour:



Write to empty Synchronous FIFO.

Each FIFO has several status ports:

- full : High if the FIFO is full. A write to a full FIFO is ignored.
- empty : High if the FIFO is empty. A read from an empty FIFO is ignored.
- level: The FIFO fill level. The width is always L2D+1 bits:
  “**`wire [L2D:0] level;`**”

## Write on Full, Read on Empty

A very important feature of a FIFO is what happens if the user makes an error when reading or writing:

- Write when full.
- Read when empty.

In all implementations the erroneous operation is ignored.

**Write when full.**
Ignoring the write means the write data is lost.
Honouring the operation would be worse as it would cause the write pointer to increment and it would become equal to the read-pointer. The FIFO would suddenly appear to be empty. Thus not one entry but all entries would be lost.

**Read when empty.**
Ignoring the read means the users sees a stale read entry. In most cases this would be the same as the last entry.
Honouring the operation would be worse as it would cause the read pointer to increment and it would move one step past the write-pointer. The FIFO would suddenly appear to be full. Thus the user gets not one stale entry but a whole FIFO full of stale entries.

These protection mechanisms seem trivial and obvious[1]. However I worked with a low-cost sub-1 GHz transceiver where the manufacturer DID have that error in the design.

**Gates usage**

The fall through behaviour is costly as it prevents the usage of synchronous memories. Therefore I use these FIFOs only for limited depth.

e.g. on a Xilinx device you can use a LUT as asynchronous read SRAM and a FIFO of depth 64 and width 8 would use 8 or 16 LUTs.

## Defines:

### ASSERT_ON

If the define "`assert_on" is found the FIFOs have extra code which detects and reports when a "write when full" or "read when empty" attempt is made.

[1] They are also very cheap to implement: a single AND gate is all that is required.