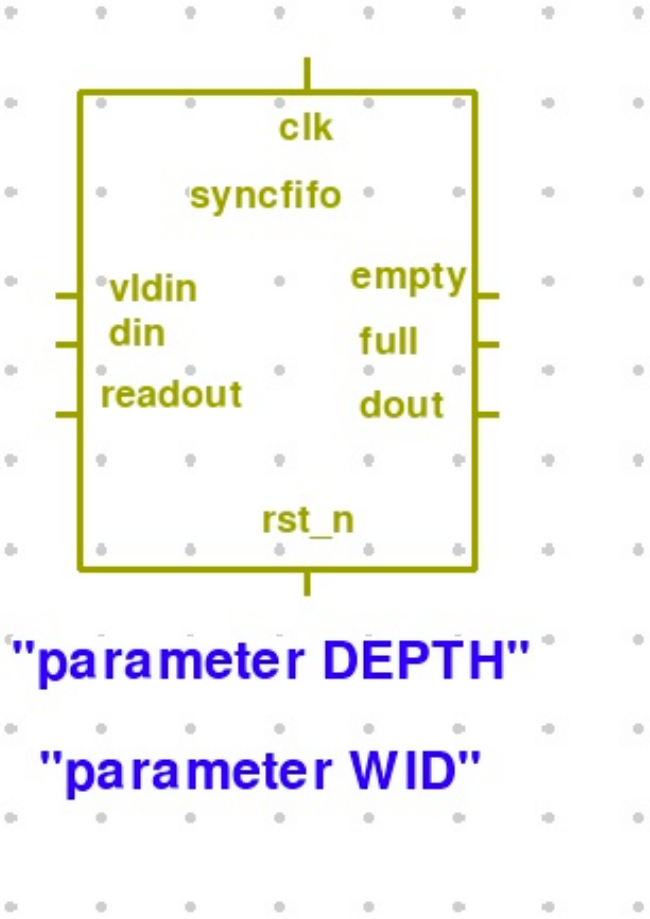
Fifos

Every RTl module i do, will have some fifos in it. Over the years i gathered quite few varieties of these.
On the other hand, most external RTL i read, the fifos are ugly, hard to use and are treated with unexplainable reverence.

So here is my collection of fifos and how they can help with speeding up the verified design.
And as usual all are clonable from my gutHub repositories.
The RTL is open-source, MIT licensed and ok to use in Your design. Unless Your bosses disagree.
If they are not ok with it, better switch companies now, while You can.

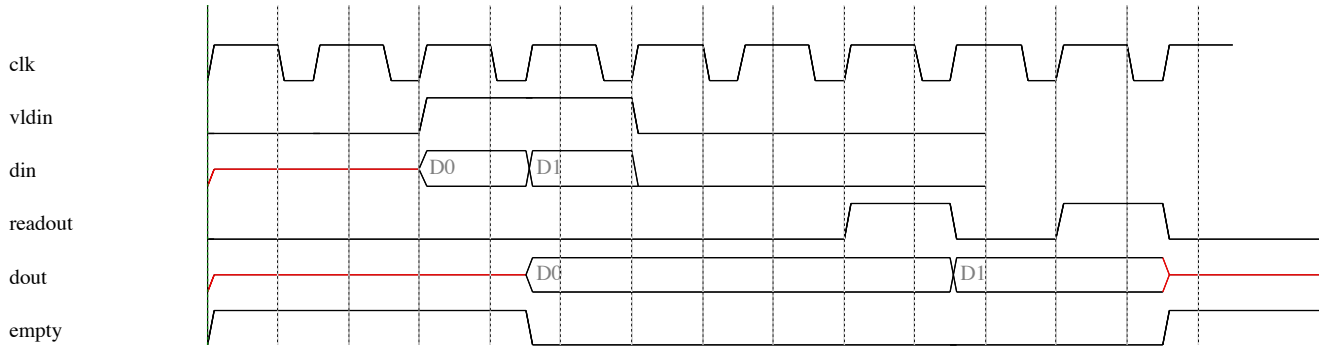
Syncfifo

Syncfifo is the basic vanilla variant. Variant is a popular word now, maybe not for the most benevolent reason.
This variant has just one clock input (clk), so all inputs and outputs are related to this one clock.
It has two parameters, WID for width of the data and DEPTH - the number of entries.
DEPTH can be any number, bigger that 1. There is no valid reason to force number of entries to be power of 2 or any other particular number. The data array is all flops. For larger values of DEPTH, it makes sense to move to RAM based fifo.



- The data array in this fifo is made out of flops (when synthesized).
- There is no combi path from any input to any output.

- This fifo always presents top of it on the "dout" output. When "readout" is pulsed, next top (if not empty) will
  be present on the "dout" output on the very next clock. I emphasize this, because most fifos i see, present top of the fifo on the next clock after read. Which is pretty lame.



pinout

pin	dir	job
clk	i	clock
rst_n	i	async reset, active low
vldin	i	push (write) din into the fifo
din	i	write data
readout	i	pop current top, and next top will show here on the next clock.
dout	o	current top of the fifo. unknown in case of empty fifo.
empty	o	there are no valid entries, reading when empty has no effect
full	o	no free empties left, writing when fukll has no effect
panic	wire	attempt to write to full fifo, or read from empty one
---	----	---

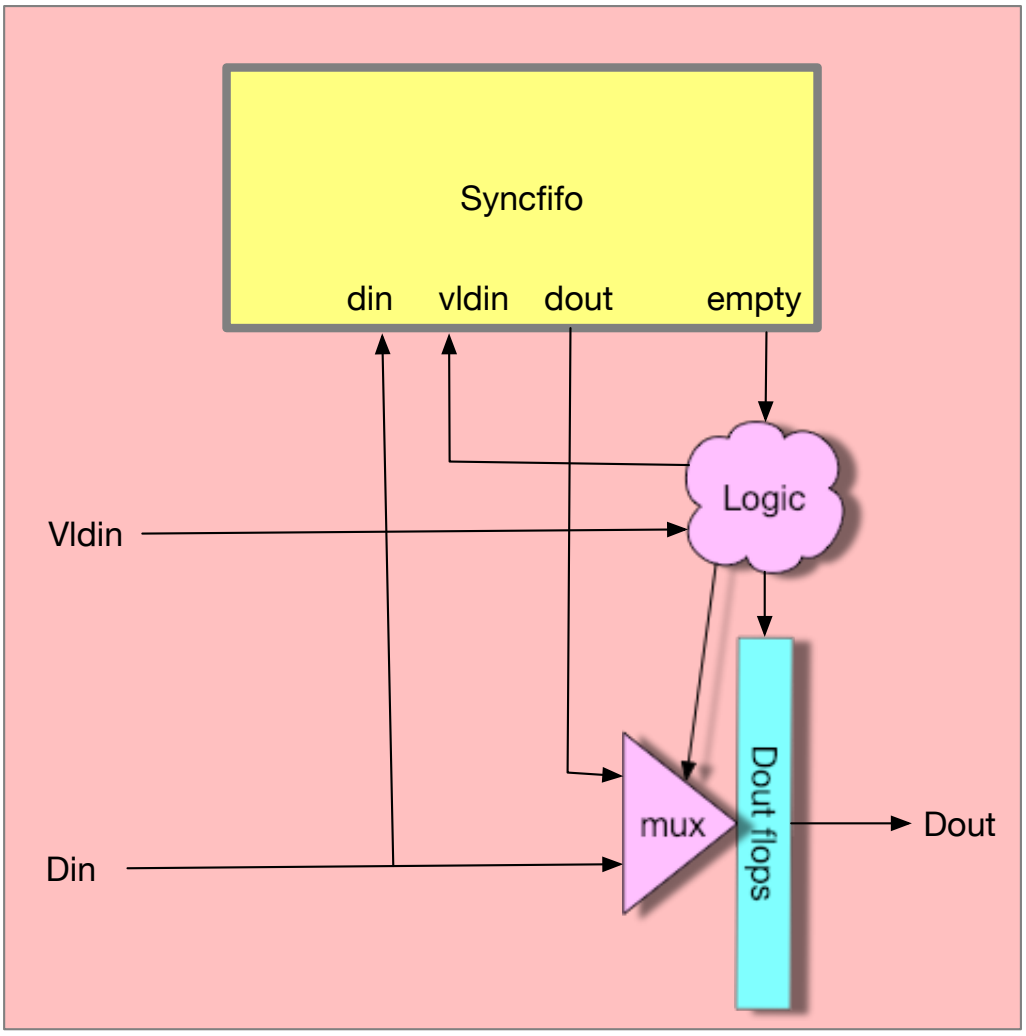
panic is an internal wire, optionally monitored by my python verification and always present in waves to catch simulation errors. Think of it as sane assertion.

Syncfifo_sampled

In vanilla syncfifo, the "dout" - output data of fifo, comes from mux selecting the correct entry to display. This adds another delay on the output.

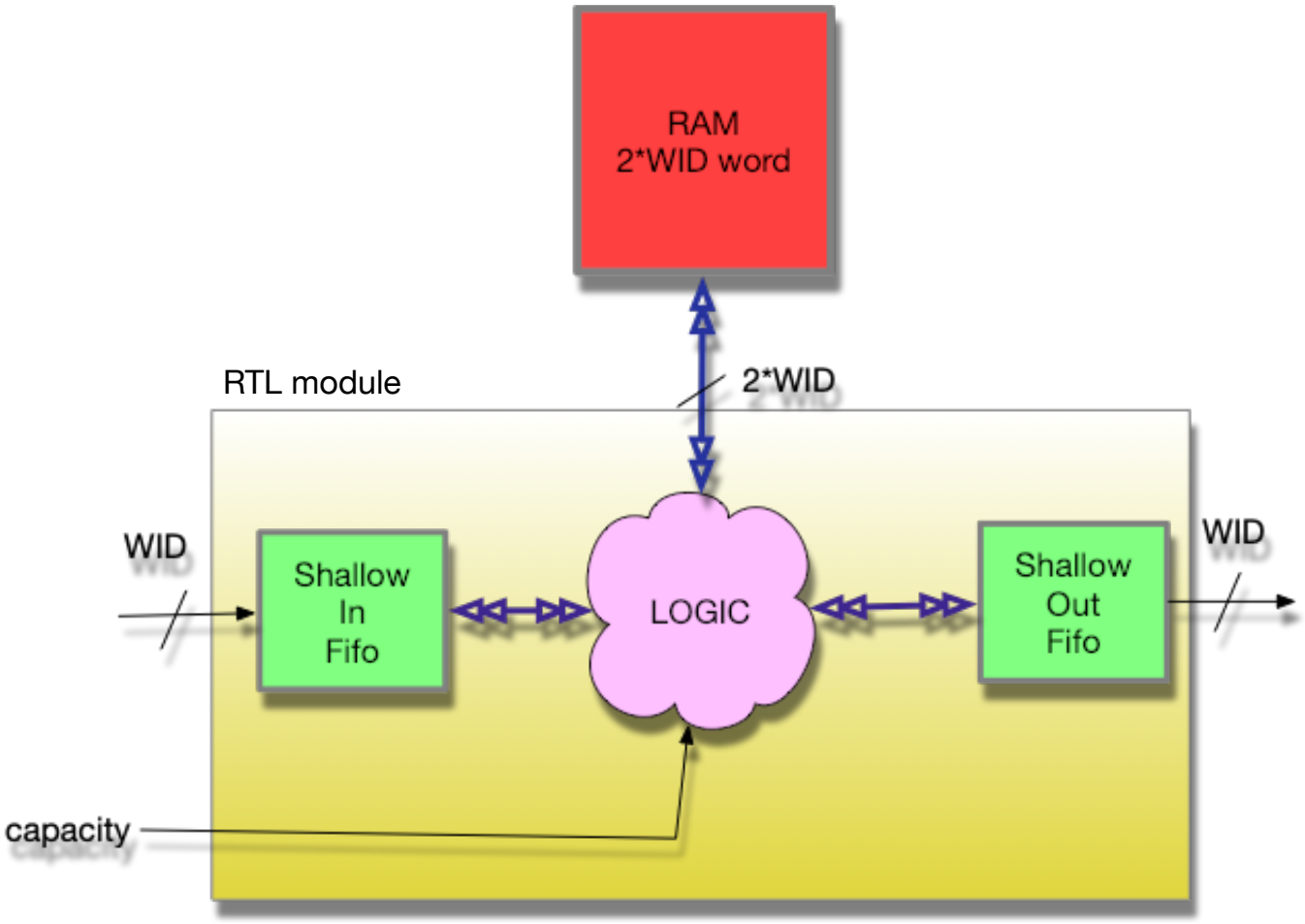
In sampled variant, "dout" is line of flops, so all outputs come from flops. Some additional logic keeps it honest at all times.

Timing closure is easier in times of need. Keeps exactly the same pinout and functionality at the expence of one more entry wide flops vector. As visible from the drawing below, It is based on vanilla syncfifo with some additional elements.



Syncramfifo

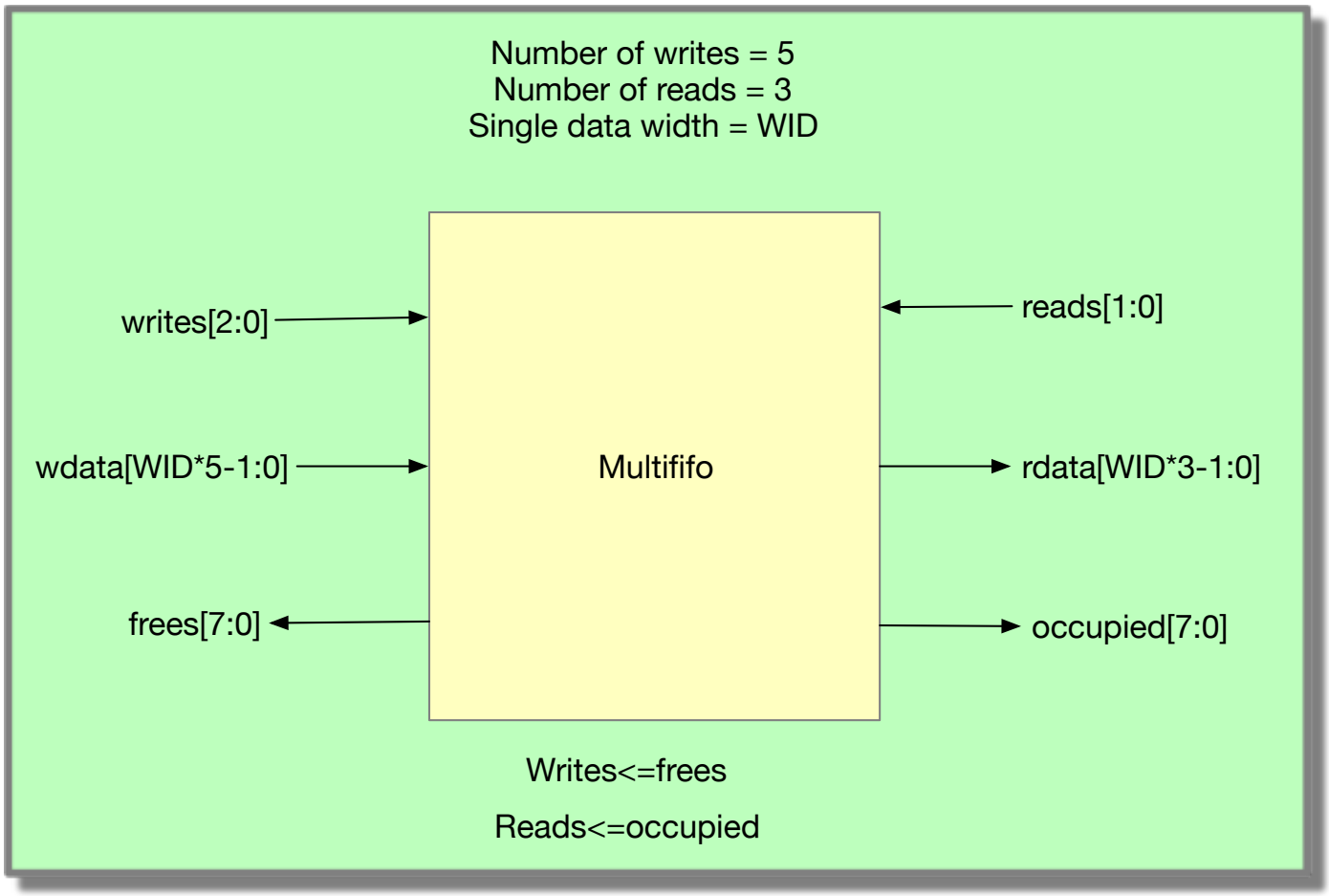
When the DEPTH parameter causes the number of data holding flops to be excessive, this variant offers to employ RAM hard macro to hold the data. Besides that, no other trait of the fifo changes. Including the ability to read/write consecutively and both at the same time. And presenting the top of the fifo on the "dout" at all times.



The number of words in the RAM can be any number. Each word is twice the width of the data. This allows continious operation with reads and writes every cycle. Additional input **capacity[15:0]** is added to fine tune the exact number of entries. The input shallow fifo is working in two modes. When the RAM is empty, it will try to push data to out fifo. If the out fifo is full it will switch modes and will write to RAM. Since the RAM ihas double width, it waits for two new entries and only then writes it to RAM. Double width RAM assures there are open clocks for reading and writing continiously.

Multififos

Sometimes need arises where more than one entry may be written at one clock and more than one entry read on the same clock. Multififos allow that, alas at price of version per number of reads and writes. Special generation python script accepts two parameters: number of reads and number of writes. In the example below, number of concurrent writes is any number from 0 to 5, and number of reads is any number from 0 to 3 at any clock. In most real cases, either Writes or Reads are one. As example, these fifos are usefull when writing from 64bit bus with wstrbs and reading is done from 8bit bus.

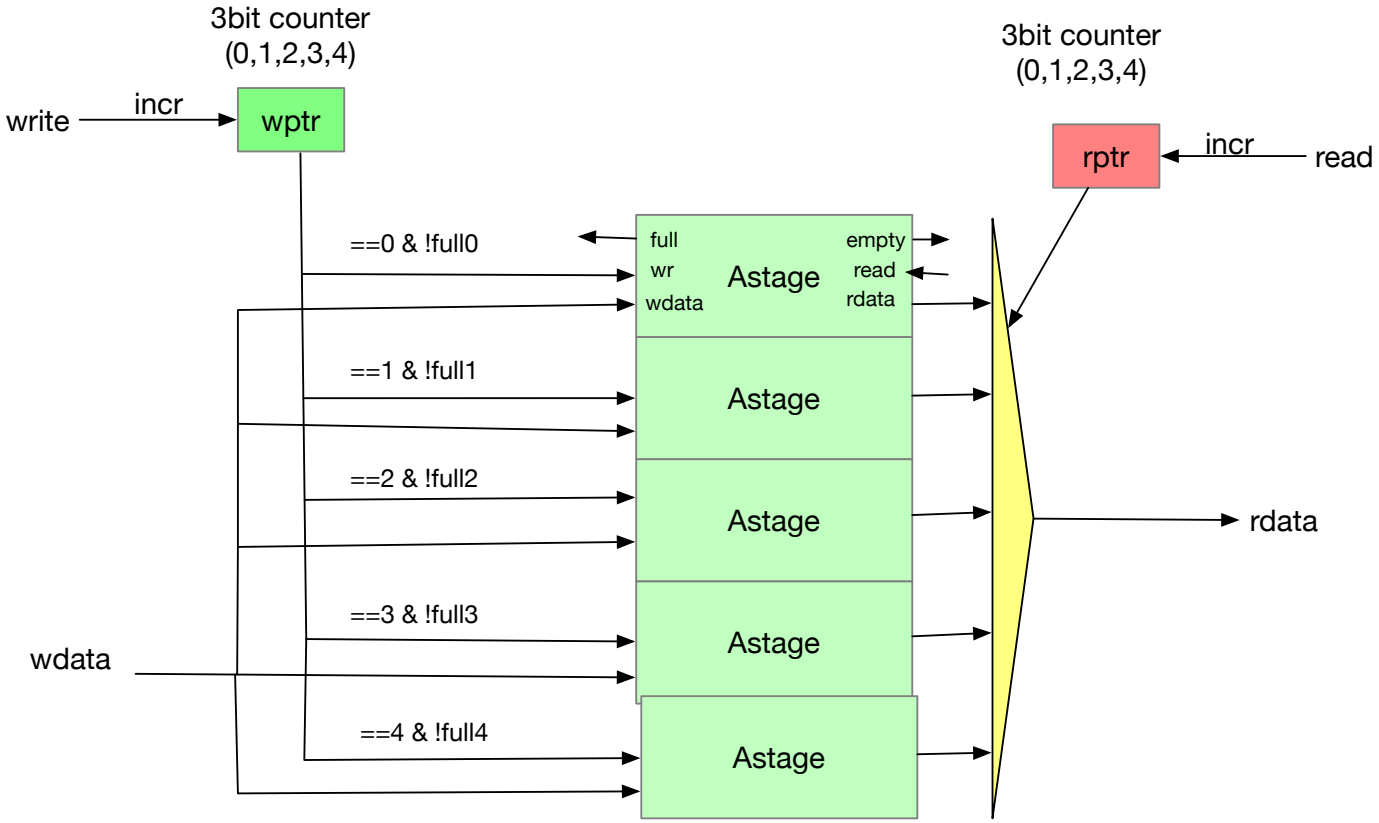


As in all fifos, top entries of the fifo are visible on rdata at all times. As in all fifos there is WPTR and RPTR pointers, just in this case they may be incremented by more than one.

Asyncfifo

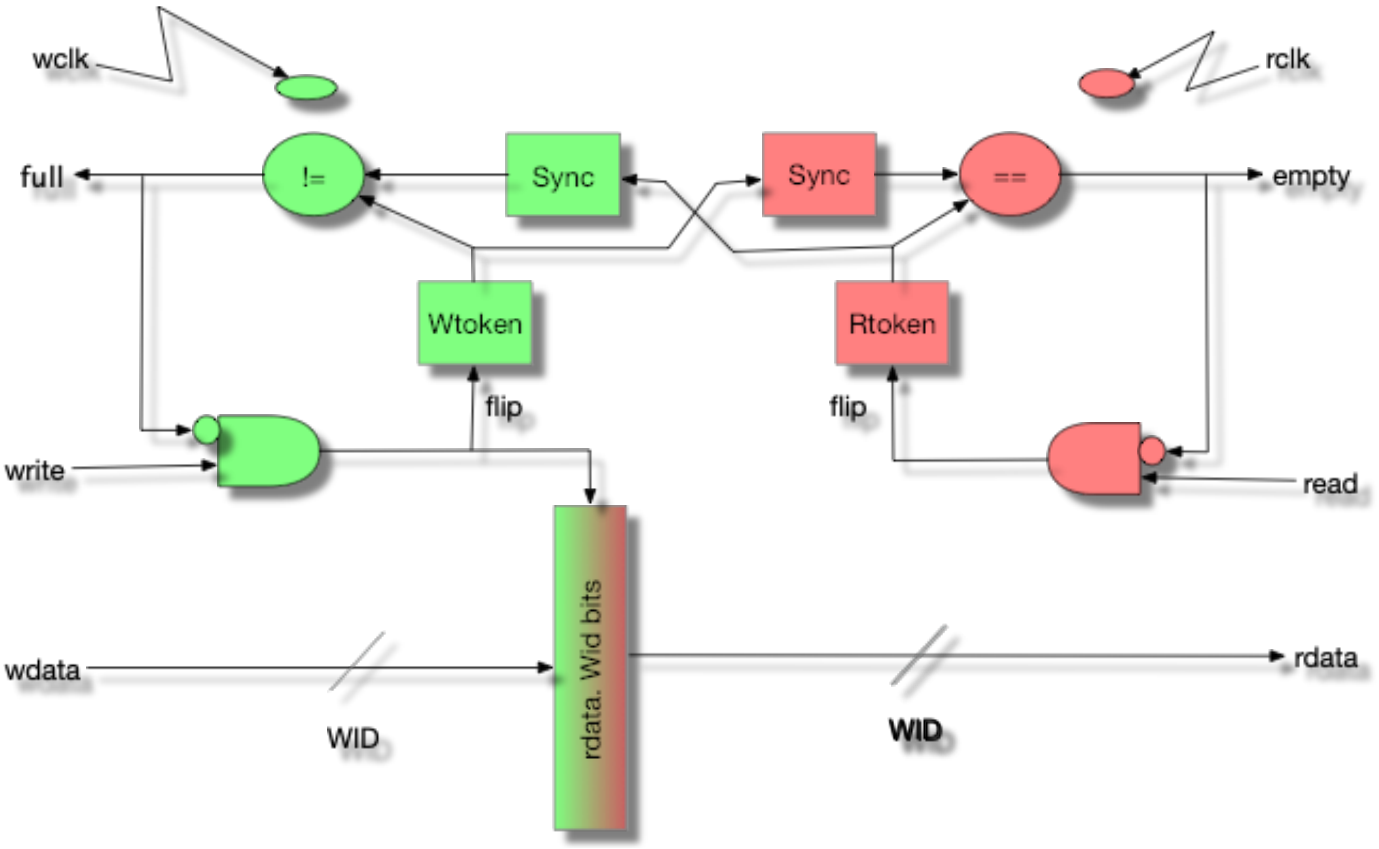
Asyncfifo allows to have two different clocks, one for all write related stuff and another for read. Usual design You may encounter employs gray code to overcome synchronization issues. Here a different approach is used.

There are 5 (or any other number) of async stages. Each stage holds an entry to the fifo. It is filled from write clock and drained from the read clock. They are filled and drained in cyclical fashion. Each write to fifo, checks that the current entry is empty, then fills it and moves the pointer to the next entry. Read works in similar way. If read is active, it checks that the entry is valid and advances the pointer to next entry.

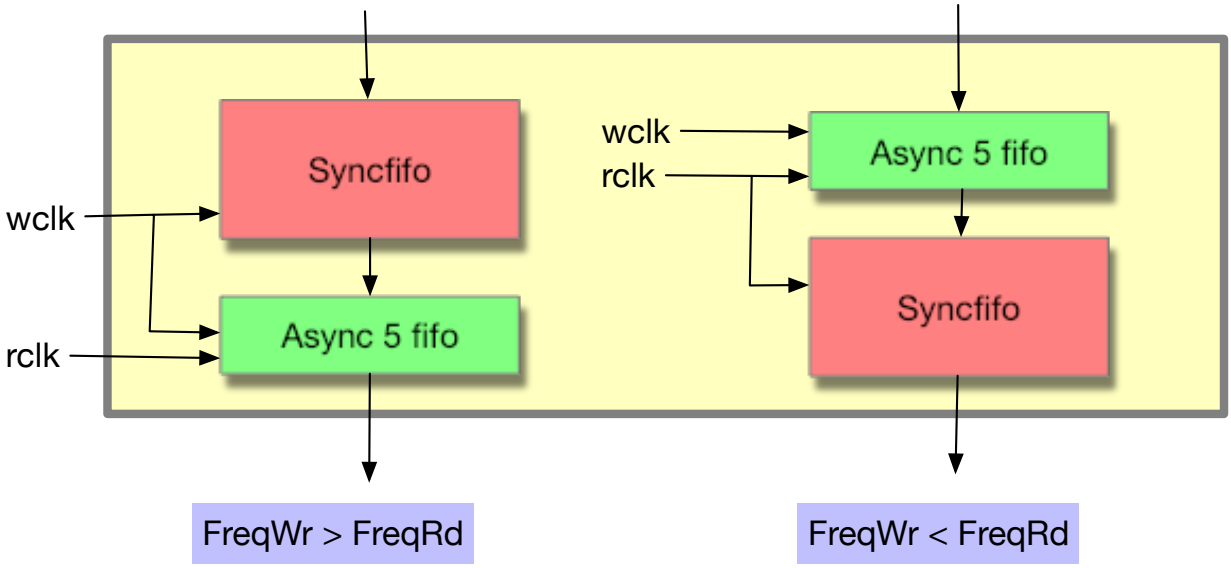


Here is the structure of each async_stage.v :

Entry is valid (full) when tokens are different and empty when tokens are equal. Any operation (read or write) flips the corresponding token.



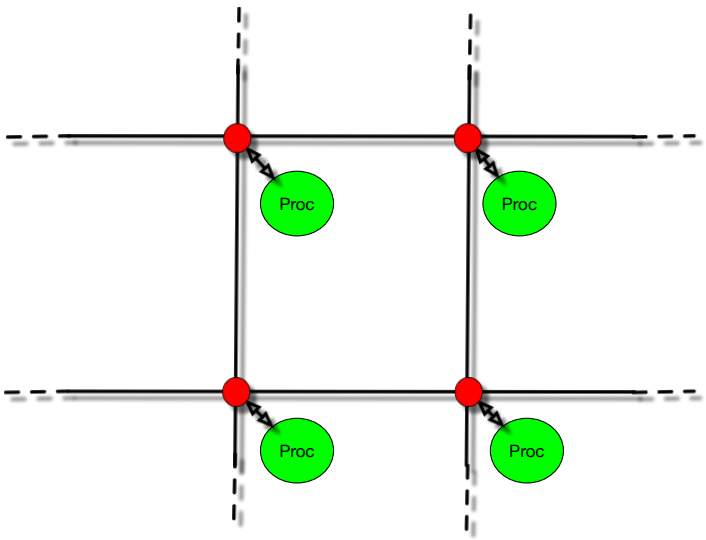
5 deep async fifo is enough. To enlarge the capacity of the fifo, put syncfifo (or syncramfifo) before or after this async5fifo. Additional syncfifo comes on the side of the faster clock.



syncfifo_shared

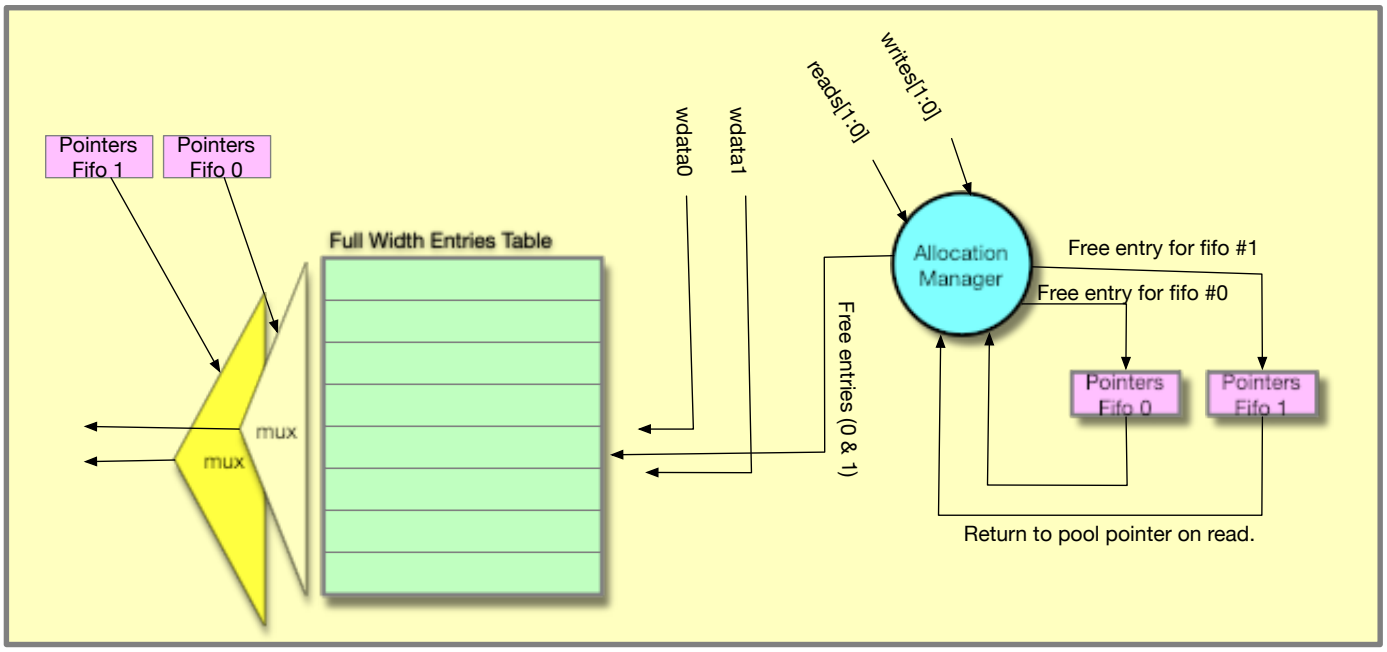
Here the scenario: we need to implement 5 sync fifos. Five is not a random choice. Grid arrangement of processing nodes likes it. Each red dot routes incoming traffic to any other four directions. (No u-turns).

We may allocate 5 indepenednt syncfifos for this task. But how do we determine the depth of each? When depth is incremented by one, it means 5 additional WID-wide flop entries.
It is more significant, when the WID is big. Suppose we keep ADDR+DATA+CONTROLS in each entry,
Width can easily go over 100 flops per entry.



Now, If Your chip needs not to be sellable (some startups aim to exit and be done with it) full 5 fifos is fine.
As it goes - no customers, no bugs.
But what about the rest?

The idea is to allocate number of full entries dictated by the anticipated traffic. And have 5 lean narrow fifos of pointers to manage the flow. The figure below shows 2-fifos version - reduced from five for clarity. Even this it is a bit complicated.



Pointer fifos are narrow ($clog2(NumberOfEntries)). Allocation manager is a vector with bit per entry, to signify occupied / free entries.

Finally, just for fun, tools used -

1. Typora to assemble the md file and pdf.
2. OmniGraffle to draw most of the drawings (Draw.io for sketching preliminaries)
3. Waveformer (my gitHub) to draw the waves.
4. zDraw to create picture of the instance.
5. VIM for writing text.
6. Coffee(s) to do something with my hands when thinking (No sugar!!).
7. M1-MacMini + Magic TrackPad + Magic Keyboard to have fun doing it.