1) In order to synchronize the same bus with different bandwidths, we choose to use a FIFO.

(a) We begin with a depth calculation for the FIFO:

$$\begin{aligned}&F_{clka} = 80\text{ MHz} \;\Rightarrow\; T_{clka} = 12.5\text{ ns}\\ &F_{clkb} = 50\text{ MHz} \;\Rightarrow\; T_{clkb} = 20\text{ ns}\\ &\text{burst} = 20(\text{BlockA writes 20 words and then resets})\end{aligned}$$

We calculate the total write time of BlockA:

$$T_{\text{all-writes}} = \text{burst} \cdot T_{clka} = 20 \cdot 12.5\text{ ns} = 250\text{ ns}$$

Read time of BlockB:

$$T_{\text{read-one}} = T_{clkb} = 20\text{ ns}$$

Number of reads BlockB performs during the write phase:

$$\text{readings during writing} = \frac{T_{\text{all-writes}}}{T_{\text{read-one}}} = \frac{250\text{ ns}}{20\text{ ns}} = 12.5\text{ readings}$$

The number of words left in the FIFO after the write phase is:

$$\text{left to read} = \text{total readings} - \text{readings during writing} = 20 - 12.5 = 7.5 \;\Rightarrow\; 8$$

Since we are designing a FIFO, we must add 2 entries to handle cases where the FIFO is not being read.

Additionally, we take into account 2 extra entries to handle the initial synchronization phase, where data is being written but not yet read.

Therefore, the total required FIFO depth is:

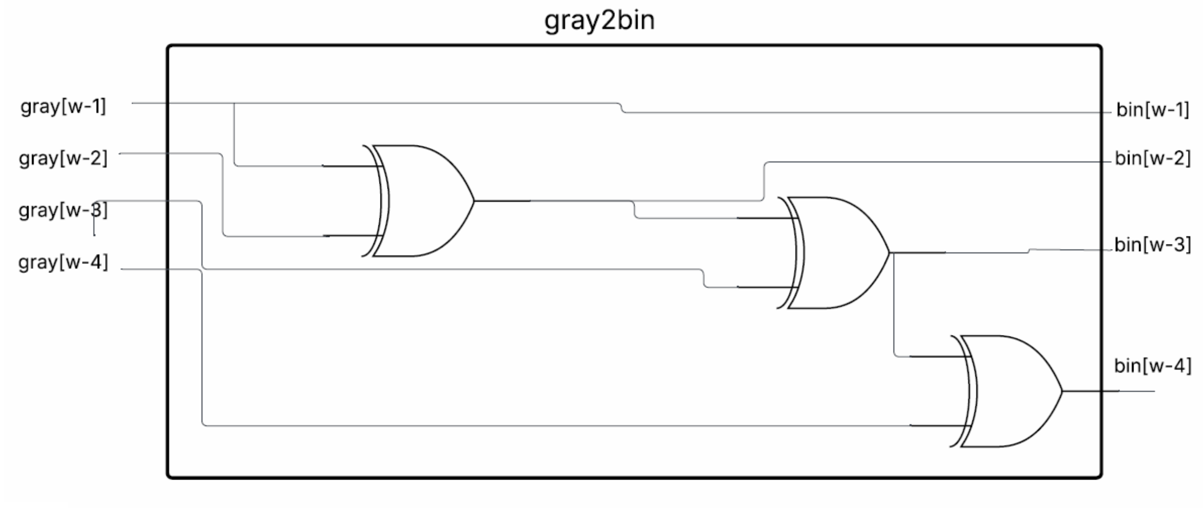
$$\text{FIFO depth} = 8 + 2 + 2 = 12$$

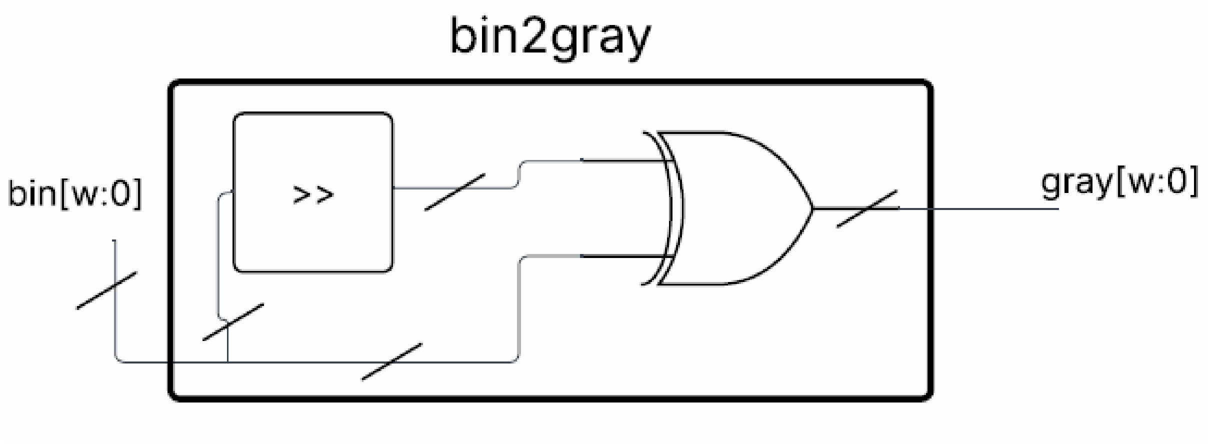
Since we are designing an asynchronous FIFO, the pointer size must include 2 additional bits to support Gray coding. Therefore, the final FIFO depth is:

$$\boxed{\text{FIFO depth} = 16}$$

b)

we took from internet the algorithms

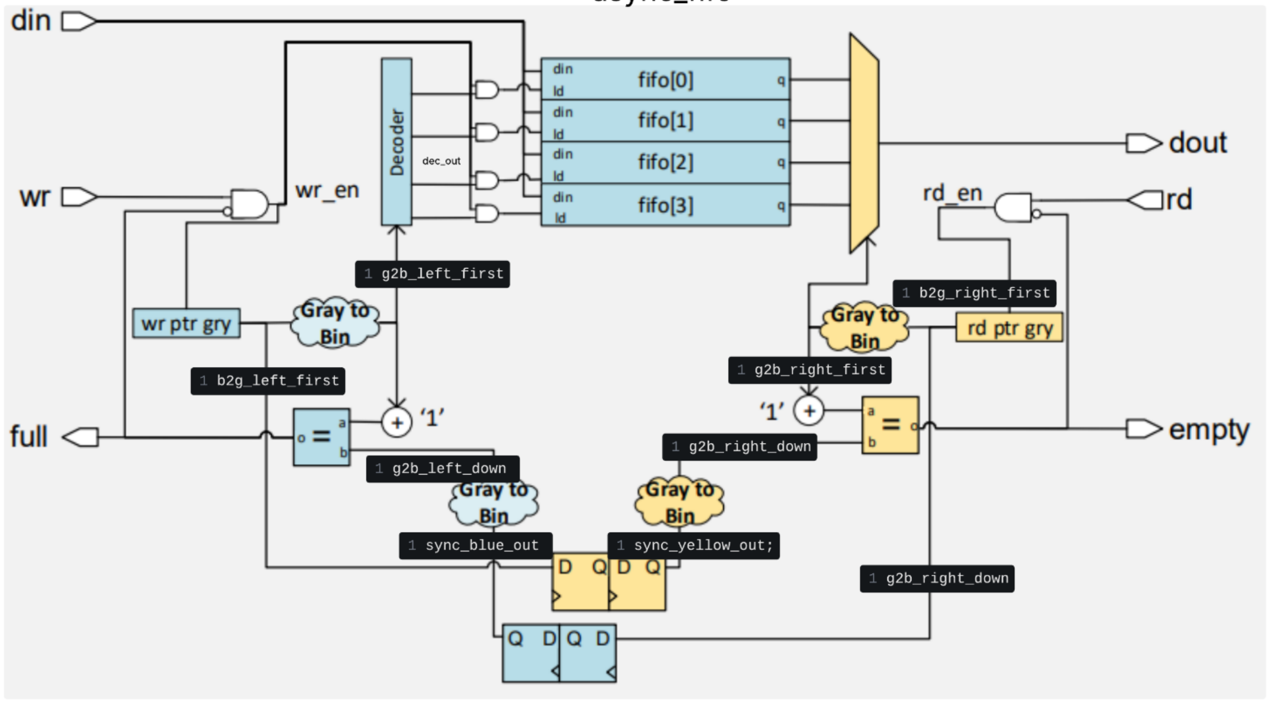




c)

scheme for async fifo as learned in class.

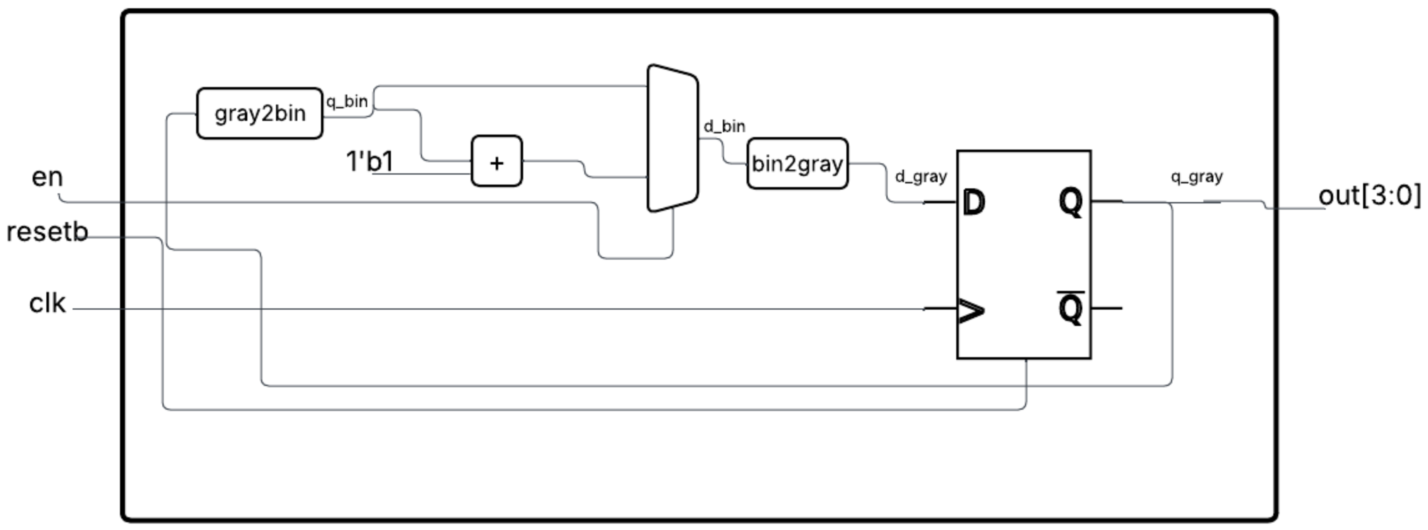
async_fifo

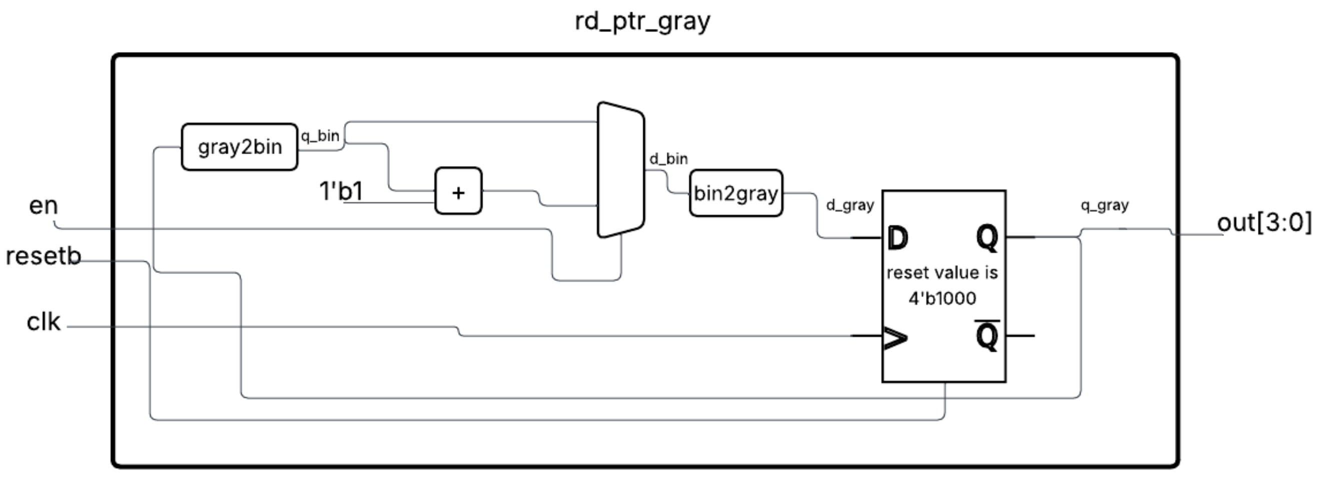


*All the black logic wires that we added to the class scheme are 4bits.

We also implement the counters in the scheme as we saw in class:

wr_ptr_gray





Note that we changed the value of the FF that synchronizes between the counter_rd and counter_wr counters.
We did this so that when a reset occurs on the write side, the synchronized value is **0** (across the entire bus).
In contrast, when a reset occurs on the read side, the synchronized value is set to the **maximum possible value**, in order to prevent overflow.

In the case of a Gray counter, the maximum value is 0…10, since the number of states is $N - 1$.
This ensures that the reset logic correctly preserves the required behavior.

In order to support both possibilities—whether the counter is used for writing or for reading—we introduced an additional parameter that controls the reset value.

**2) We now sketch a synchronization scheme for the sync bridge, in which the FIFO we designed is used.**

First, we would like to connect the inputs and outputs of the FIFO to the relevant logic and complete the synchronization mechanism.

**Inputs:**

- **clka** – connected to the clka clock, using the new connections as shown in the diagram.
- **clkb** – connected to the clkb clock, using the new connections as shown in the diagram.
- **reset_clkb** – connected to reset_b, and additionally synchronized using a reset synchronizer, connected to reset_a.
- **din_clka[7:0]** – connected to din, the FIFO input data bus.
- **data_req_clka** – the data request signal is transferred to domain B using a synchronizer (from domain A to domain B).
  Since one clock period of clka is 12.5 ns and one clock period of clkb is 20 ns, we get:

$$1.5 \cdot T_A = 18.75 \text{ ns} < 20 \text{ ns} = T_B$$

Therefore, even if the request pulse is asserted for only 1.5 clock cycles, it will still be detected in domain B.
After the request is synchronized, the data bus is transferred from domain A to domain B for 20 clock cycles,using module req_extenstion and finally data_req_clka is deasserted.

- **data_valid_clka** – connected to wr of the FIFO, meaning data is written into the FIFO.

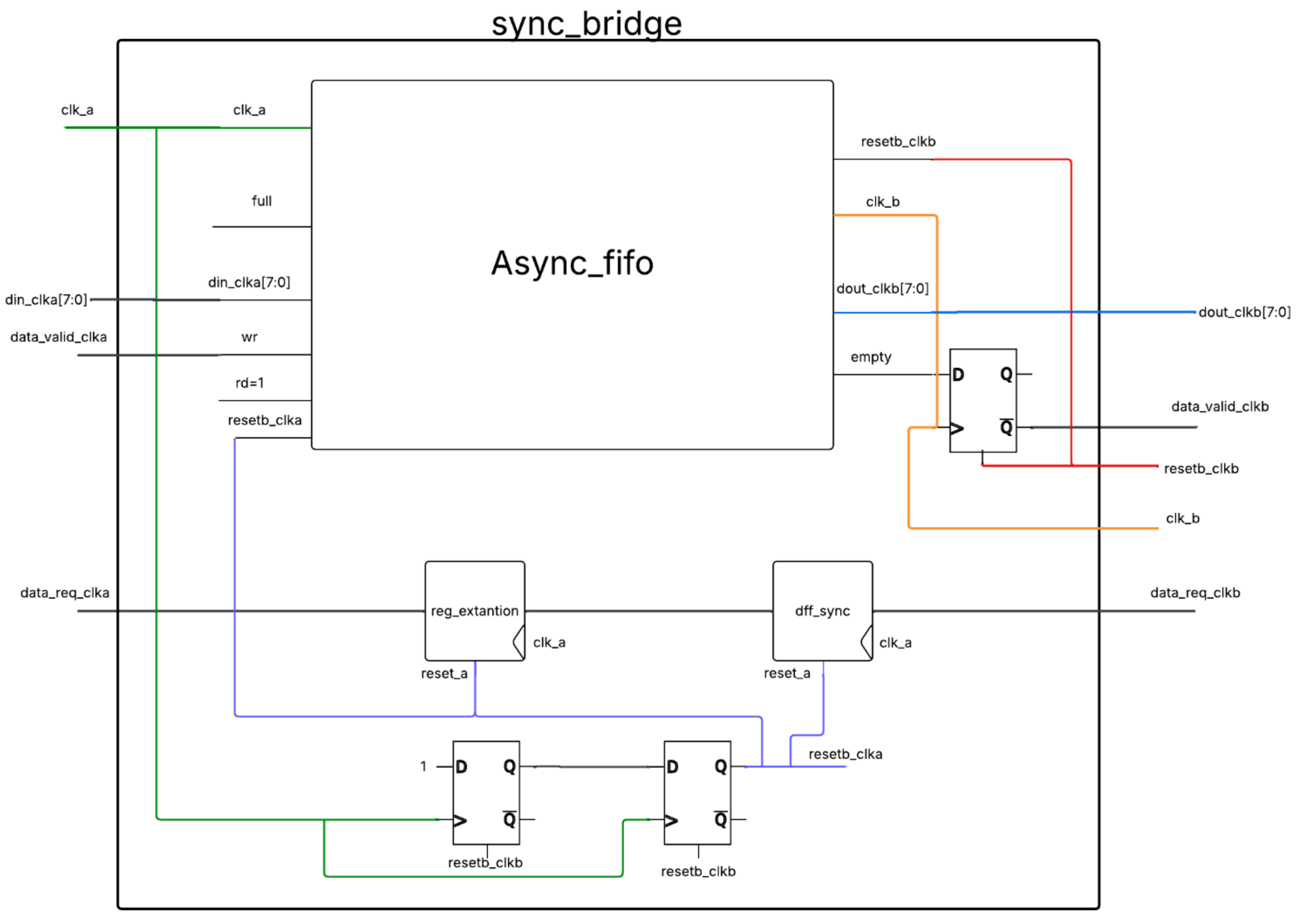
**Outputs:**

- **data_req_clkb** – connected to data_req_clka, as shown in the diagram.
- **data_valid_clkb** – asserted when the FIFO is not empty, meaning valid data is available.
  The read operation is therefore performed when empty = 0.
  The output empty is generated in domain B using clock clkb.
- **dout_clkb[7:0]** – connected to the FIFO output dout.

Note that there is no need to use the FIFO full signal, since we previously calculated the FIFO depth such that writing 20 words will never cause an overflow.

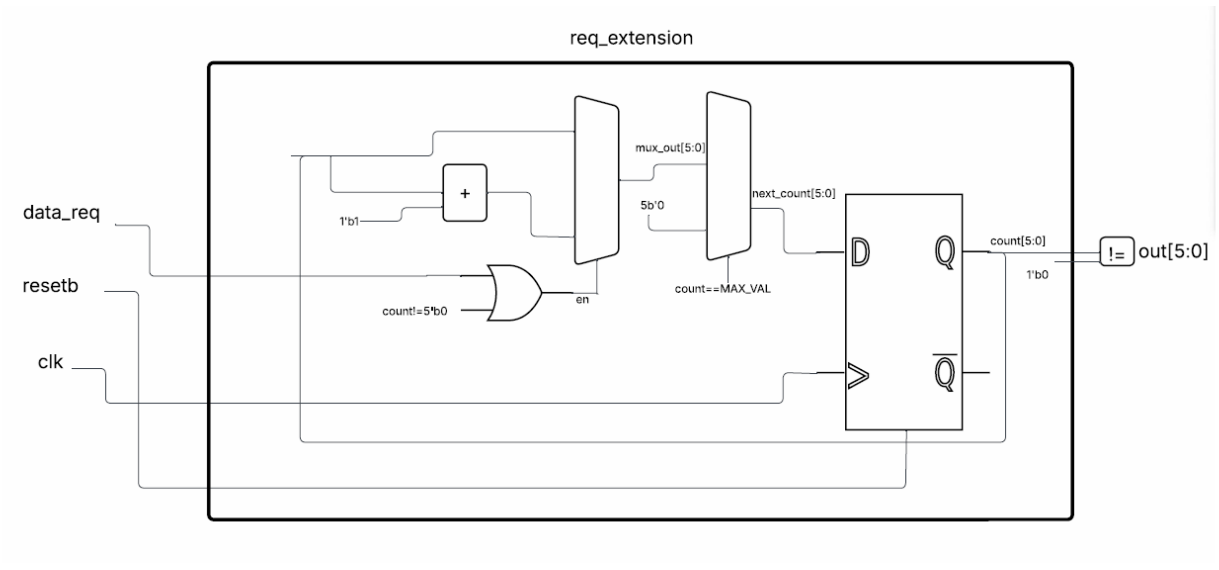
In addition, we always want to remain in read mode; therefore, the read enable (rd) is permanently set to logic '1'.

The full scheme is :



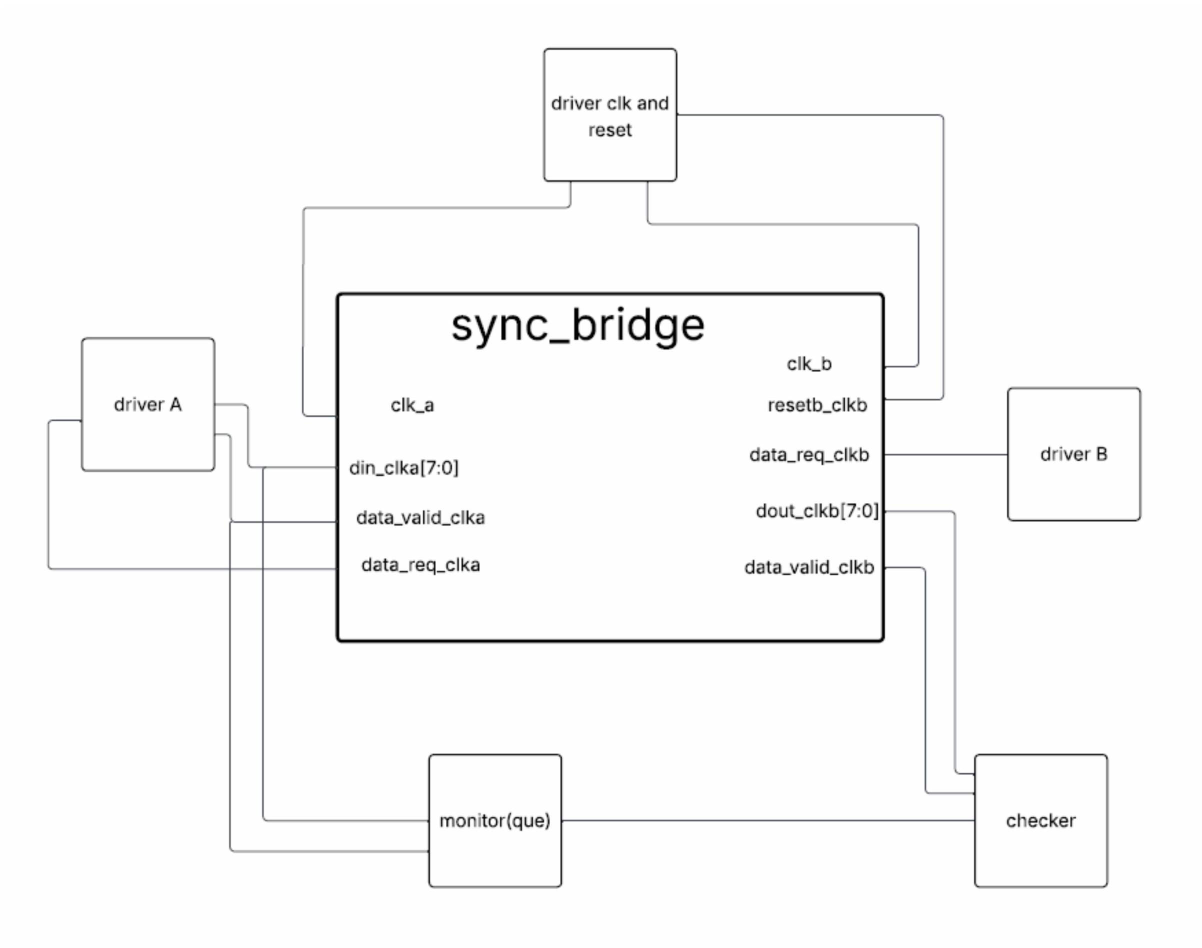
The resetb_clka scheme is  inside as we learn in class,

reg_extansion scheme ,req_exetension is counter until MAX_VAL which is 21 in our question,used to extend data_req_clkb to 21 cycles after synchronizer



4)

Test bench diagram:

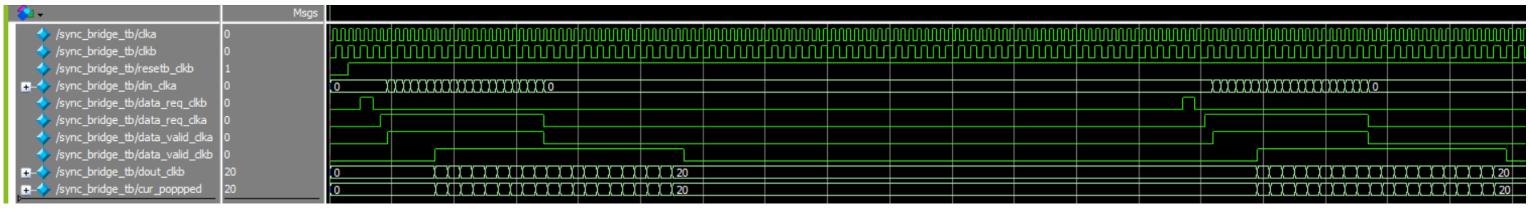


The top-level driver generates the clocks and resets according to the required specifications. After that, a data_req_clkb is issued through **driver B** in order to request words from **domain A**. The request propagates through the sync_bridge module according to the implementation described in the previous section and extended to data_req_clka.
Based on this behavior, the signals data_valid_clka and din_clka are driven into the sync_bridge by driver A, and on the output side. in **domain B**, the signals data_valid_clkb and dout_clkb are produced by sync_bridge.

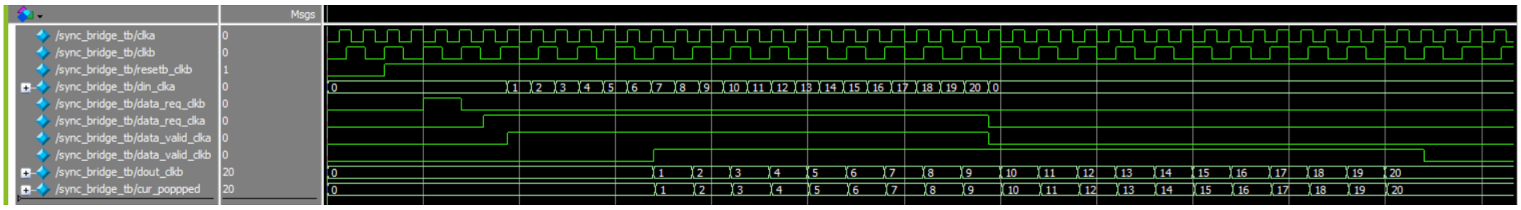
During this process, the **monitor** observes the signals generated by **driver A** and inserts the data into a queue according to the data_req_clka signal.
When the data reaches **domain B** and the signal data_valid_clkb is asserted, the **checker** compares dout_clkb[7:0] with the value at the front of the queue that was collected by the monitor.
Based on the comparison result, the checker prints whether the test passed or failed.

The driver for **domain A** was implemented as a **counter** that outputs values from 0 to 20 rather than using random data, in order to make debugging the module clearer and more intuitive.

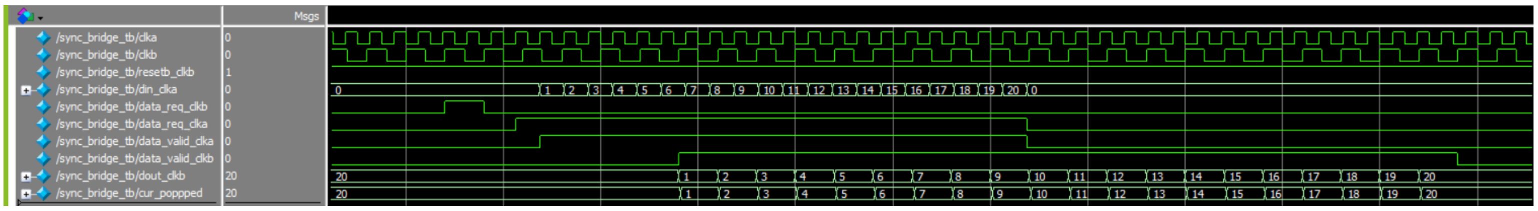
Here you cam see the full waveform of the tb:


Msgs
/sync_bridge_tb/clka 0
/sync_bridge_tb/clkb 0
/sync_bridge_tb/resetb_clkb 1
/sync_bridge_tb/din_clka 0
/sync_bridge_tb/data_req_clkb 0
/sync_bridge_tb/data_req_clka 0
/sync_bridge_tb/data_valid_clka 0
/sync_bridge_tb/data_valid_clkb 0
/sync_bridge_tb/dout_clkb 20
/sync_bridge_tb/cur_poppped 20

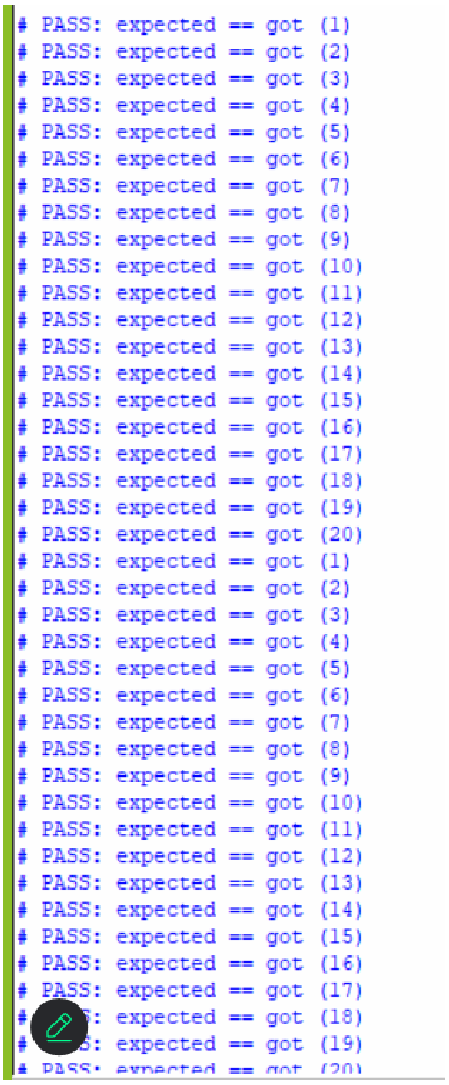
Here is zoom in on the left side of the waveform of the tb:


Msgs
/sync_bridge_tb/clka 0
/sync_bridge_tb/clkb 0
/sync_bridge_tb/resetb_clkb 1
/sync_bridge_tb/din_clka 0
/sync_bridge_tb/data_req_clkb 0
/sync_bridge_tb/data_req_clka 0
/sync_bridge_tb/data_valid_clka 0
/sync_bridge_tb/data_valid_clkb 0
/sync_bridge_tb/dout_clkb 20
/sync_bridge_tb/cur_poppped 20
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Here is zoom-in on the right side of the waveform of the tb:


Msgs
/sync_bridge_tb/clka 0
/sync_bridge_tb/clkb 0
/sync_bridge_tb/resetb_clkb 1
/sync_bridge_tb/din_clka 0
/sync_bridge_tb/data_req_clkb 0
/sync_bridge_tb/data_req_clka 0
/sync_bridge_tb/data_valid_clka 0
/sync_bridge_tb/data_valid_clkb 0
/sync_bridge_tb/dout_clkb 20
/sync_bridge_tb/cur_poppped 20
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 0
20 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
20 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Here you can see the comparison done in the tb



```
# PASS: expected == got (1)
# PASS: expected == got (2)
# PASS: expected == got (3)
# PASS: expected == got (4)
# PASS: expected == got (5)
# PASS: expected == got (6)
# PASS: expected == got (7)
# PASS: expected == got (8)
# PASS: expected == got (9)
# PASS: expected == got (10)
# PASS: expected == got (11)
# PASS: expected == got (12)
# PASS: expected == got (13)
# PASS: expected == got (14)
# PASS: expected == got (15)
# PASS: expected == got (16)
# PASS: expected == got (17)
# PASS: expected == got (18)
# PASS: expected == got (19)
# PASS: expected == got (20)
# PASS: expected == got (1)
# PASS: expected == got (2)
# PASS: expected == got (3)
# PASS: expected == got (4)
# PASS: expected == got (5)
# PASS: expected == got (6)
# PASS: expected == got (7)
# PASS: expected == got (8)
# PASS: expected == got (9)
# PASS: expected == got (10)
# PASS: expected == got (11)
# PASS: expected == got (12)
# PASS: expected == got (13)
# PASS: expected == got (14)
# PASS: expected == got (15)
# PASS: expected == got (16)
# PASS: expected == got (17)
# [illegible]: expected == got (18)
# [illegible]S: expected == got (19)
# PASS: expected == got (20)
```