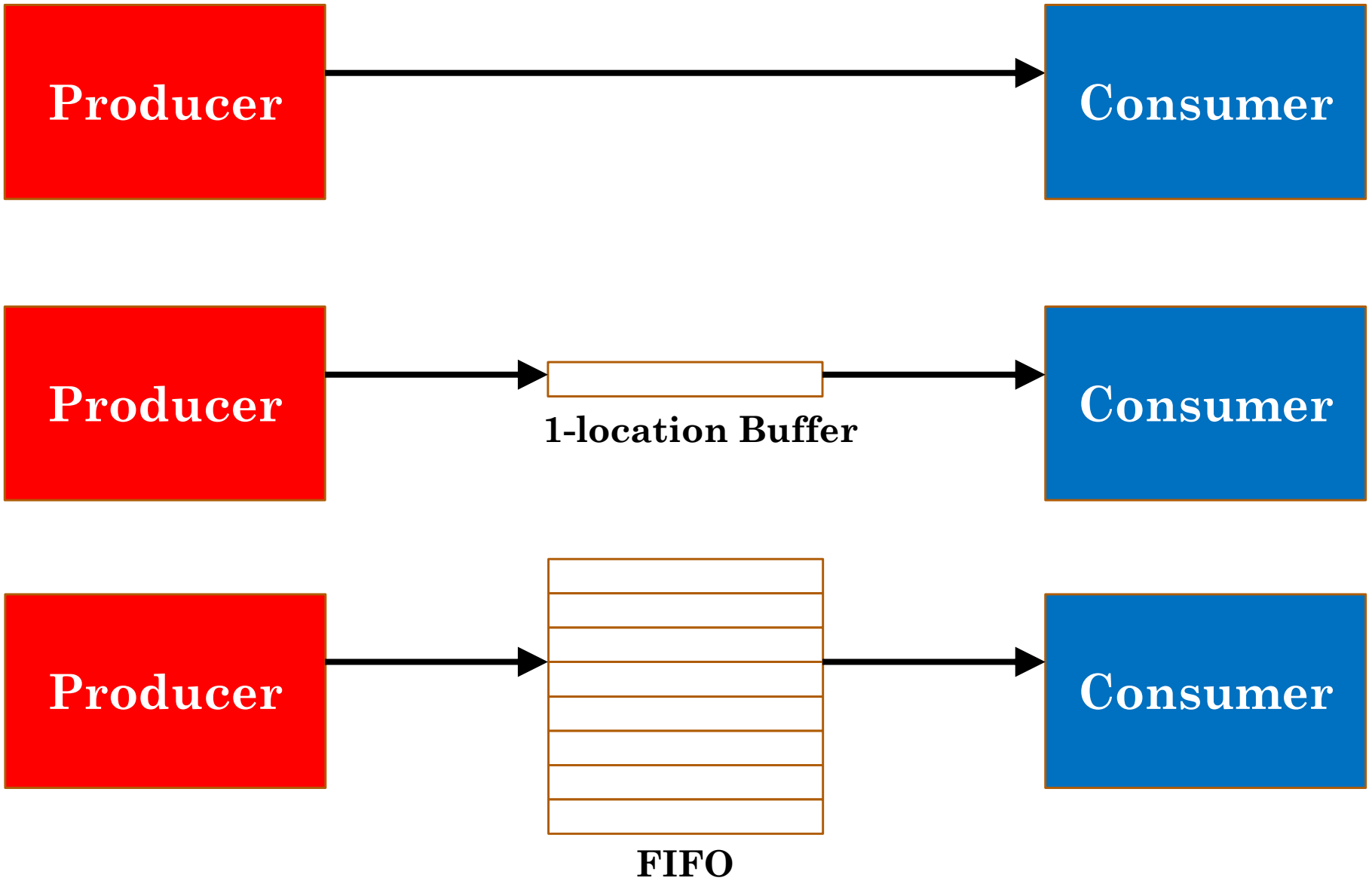
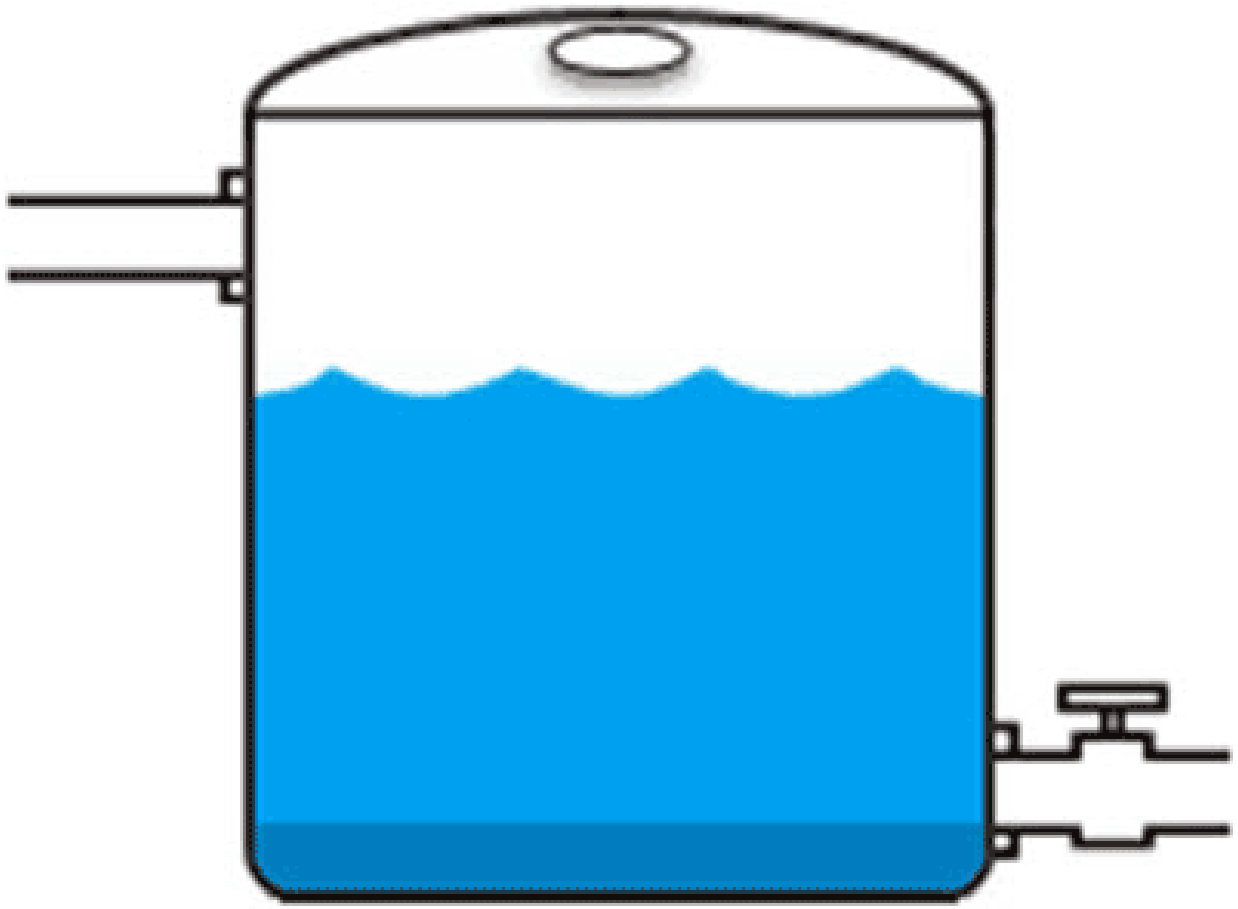
FIFO:

1-clock FIFO
and
2-clock FIFO


Producer
Consumer
Producer
1-location Buffer
Consumer
Producer
FIFO
Consumer

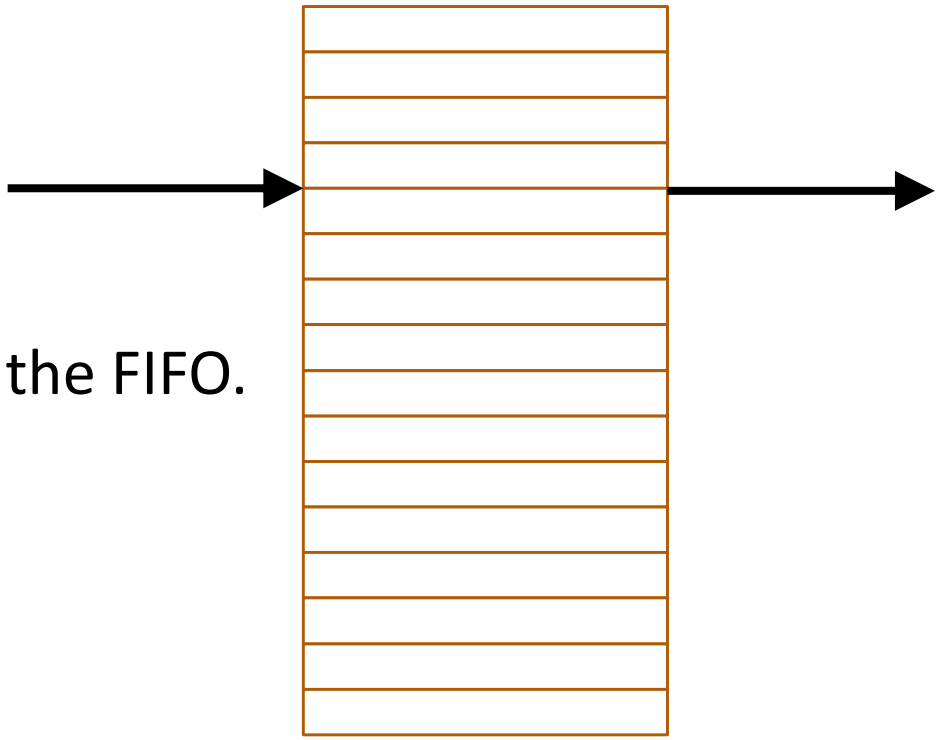


A water tank delinks the pumping of water by the city and consumption of water by the residents.

In the same fashion a FIFO delinks the producer of data and the consumer of data by holding the excessive production in the FIFO.

FIFO

Delinks the producer and the consumer



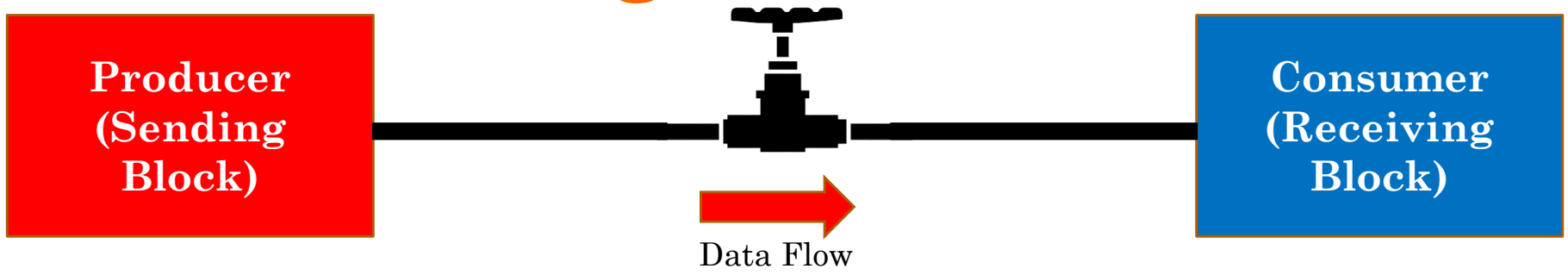
PRODUCER

- Writer
- Deposits Data into the FIFO.

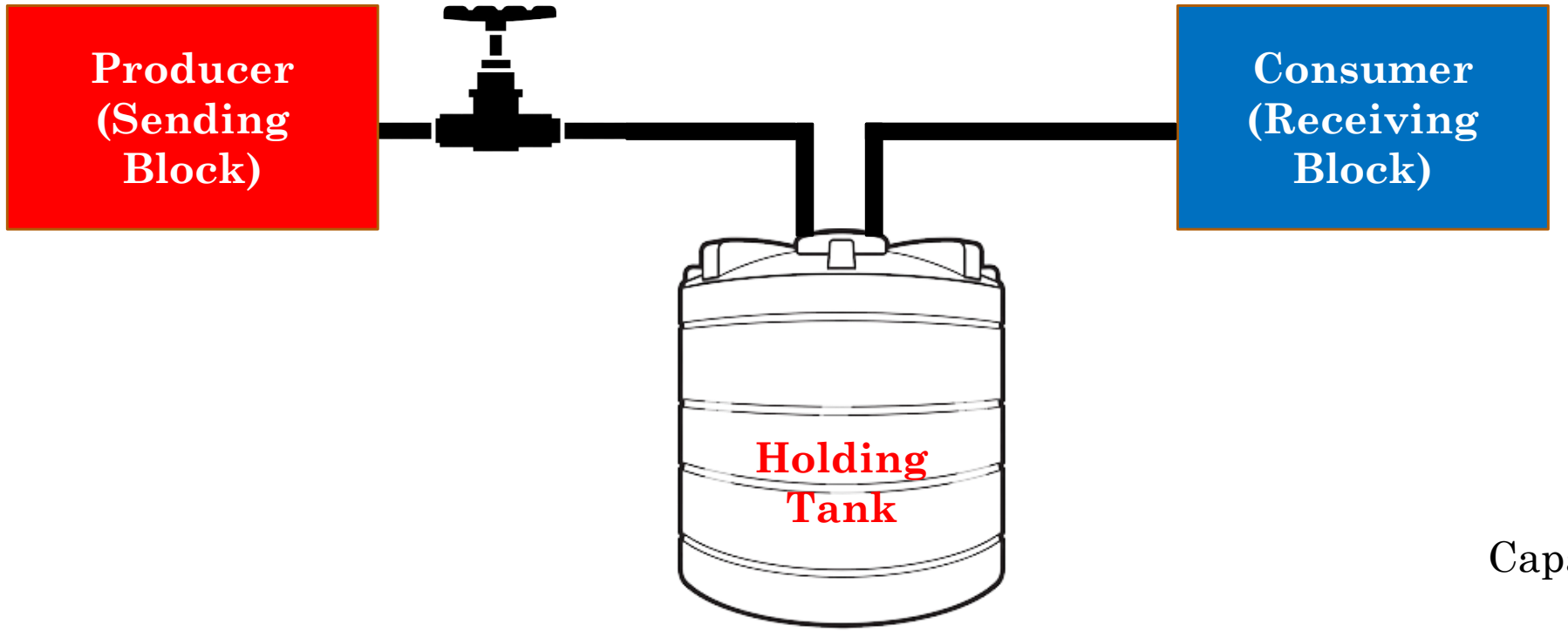
CONSUMER

- Reader
- Reads Data from the FIFO.

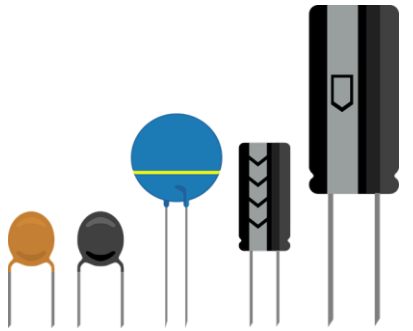
Data Plumbing:



Matched flow schedules require no buffering



Unmatched flow schedules require buffering



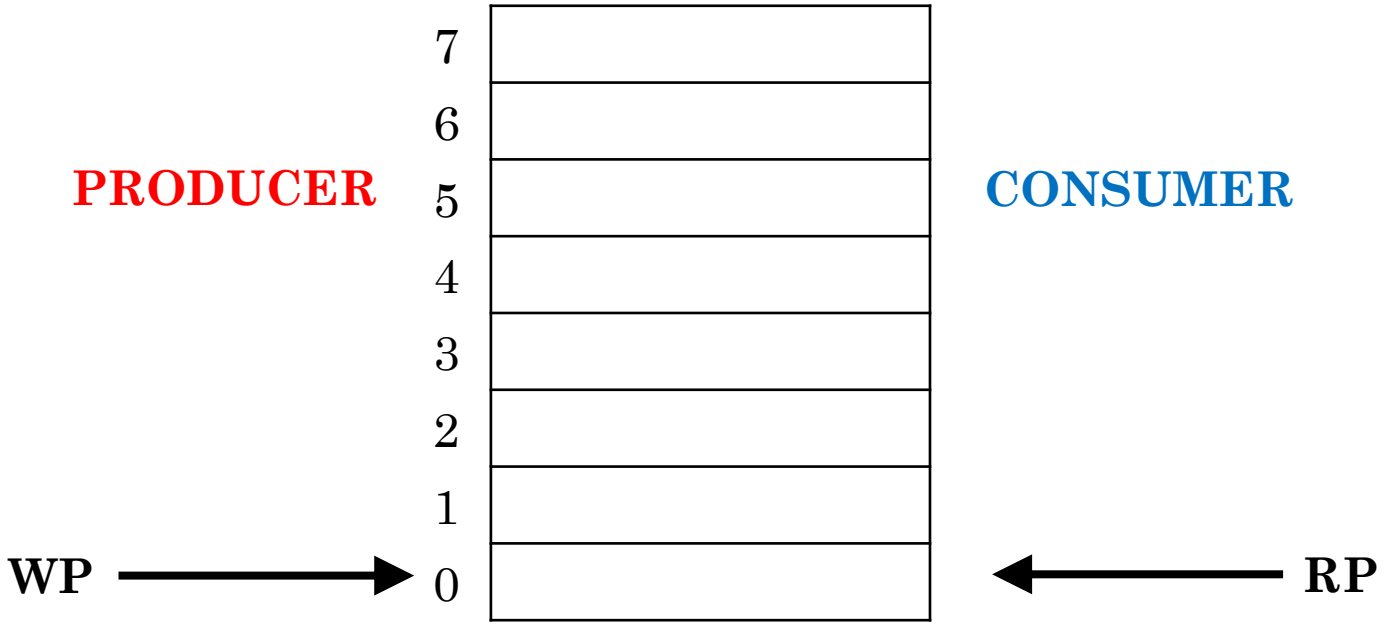
Capacitor on circuit boards

Connecting data with different bit-widths is like trying to drink water from a fire hose.

We need an adapter.

FIFOs with “aspect ratio” are used to go from, say, a 32-bit data producer to an 8-bit data consumer. Here, we are covering FIFOs with same size data.

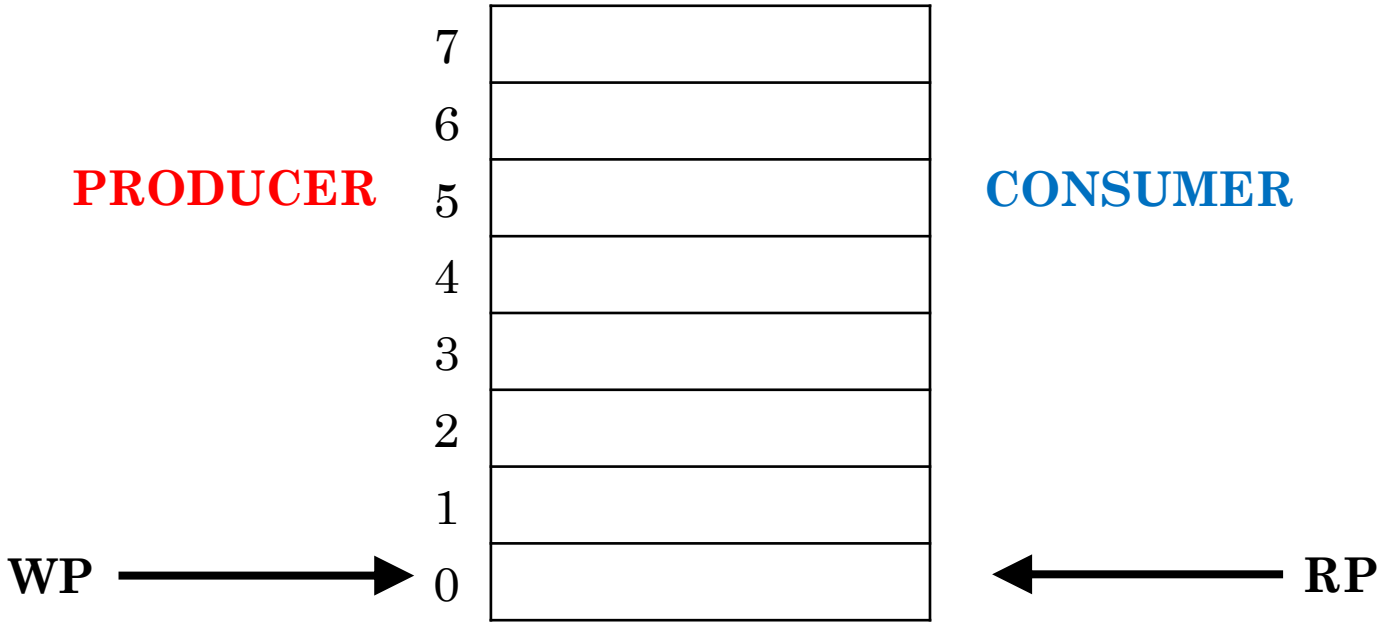
Let us use an 8-location FIFO for our design example



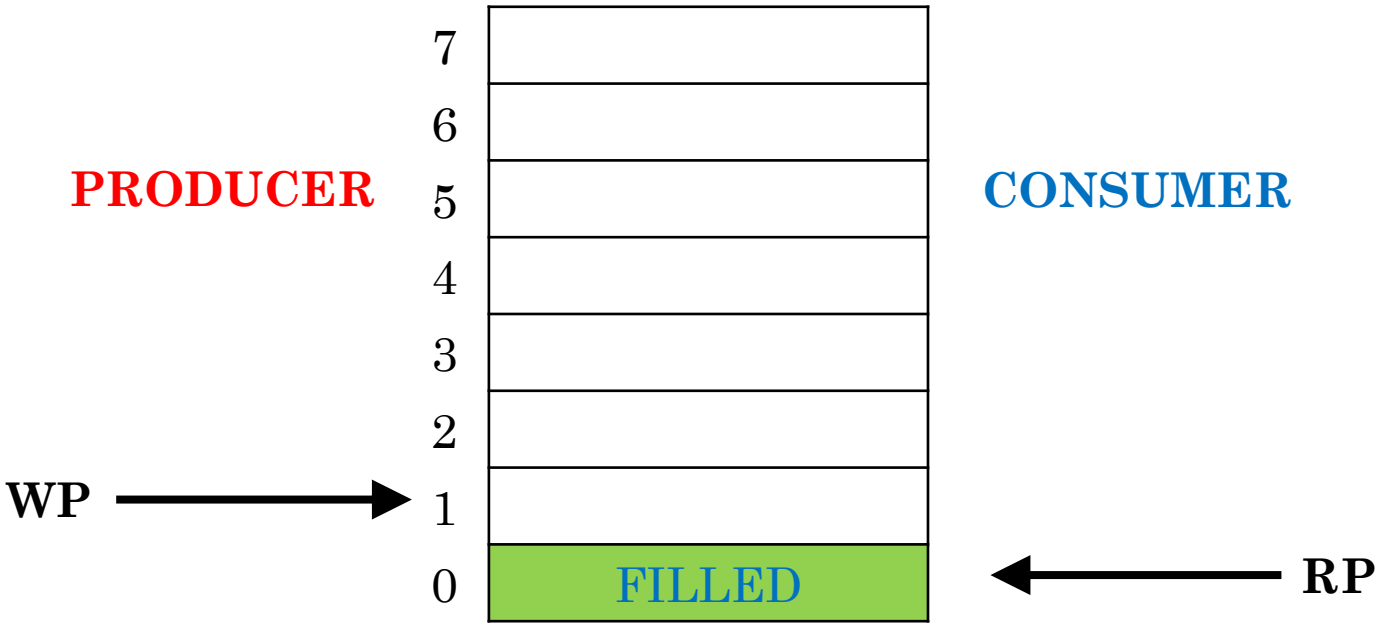
Initially

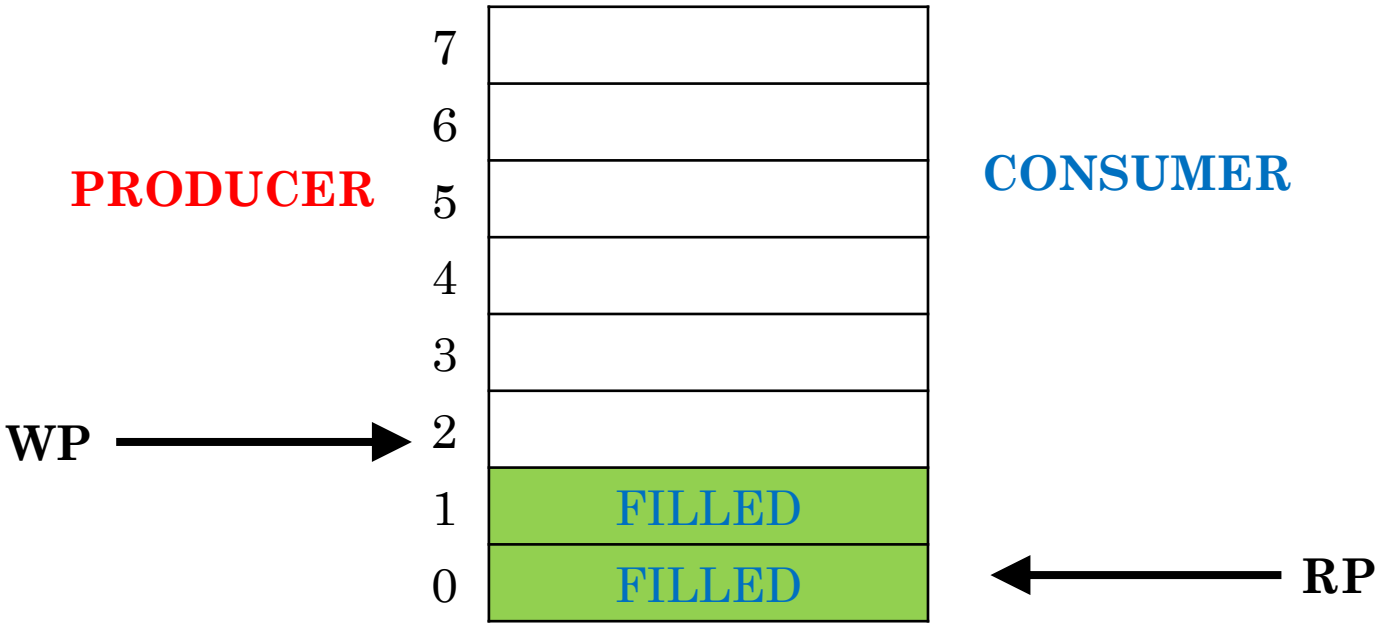
- The FIFO has two pointers to denote the locations to write and read:
  - Write Pointer (WP): The location pointed to by the WP is where producer deposits data.
  - Read Pointer (RP): The location pointed to by the RP is from where consumer reads data.

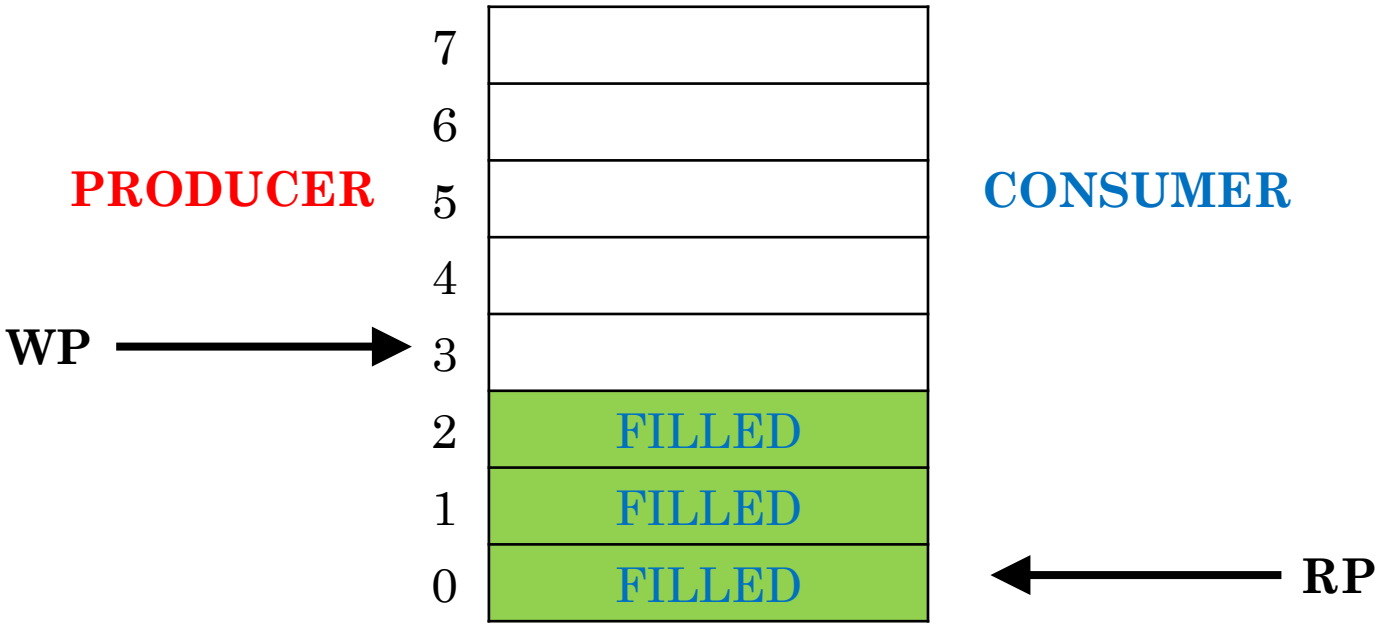
- Initially the FIFO is empty. Both WP and RP point to location 0 of the FIFO.

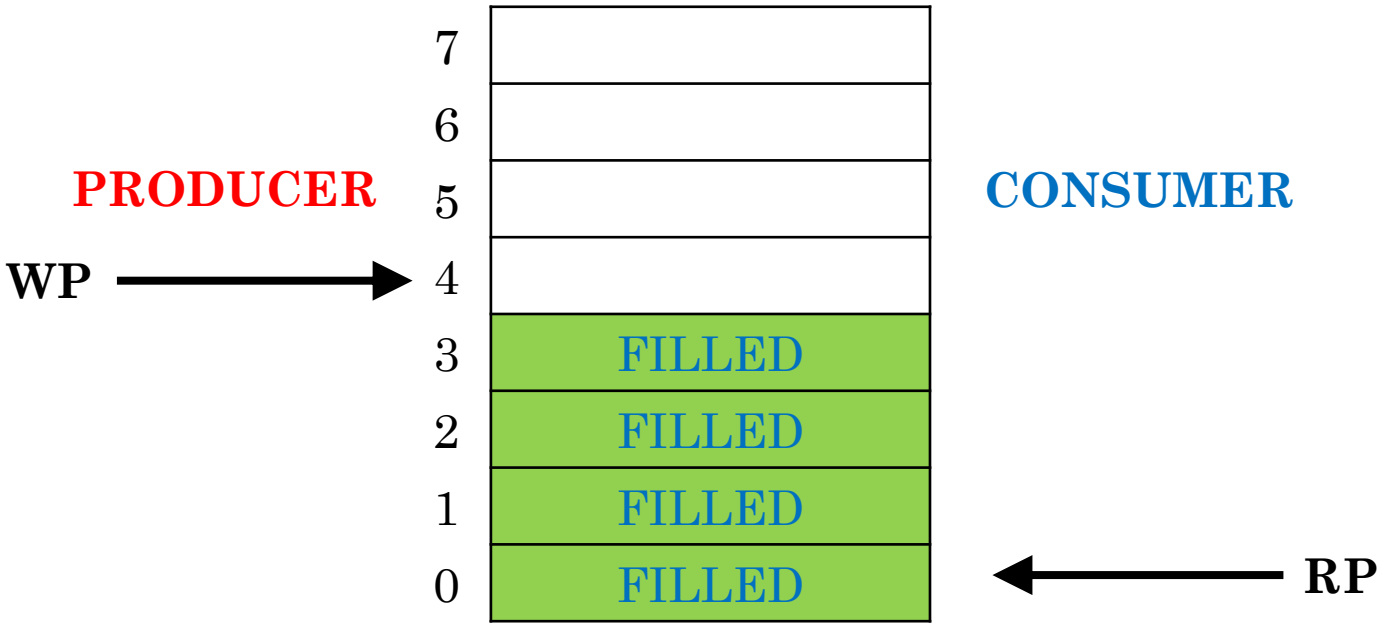


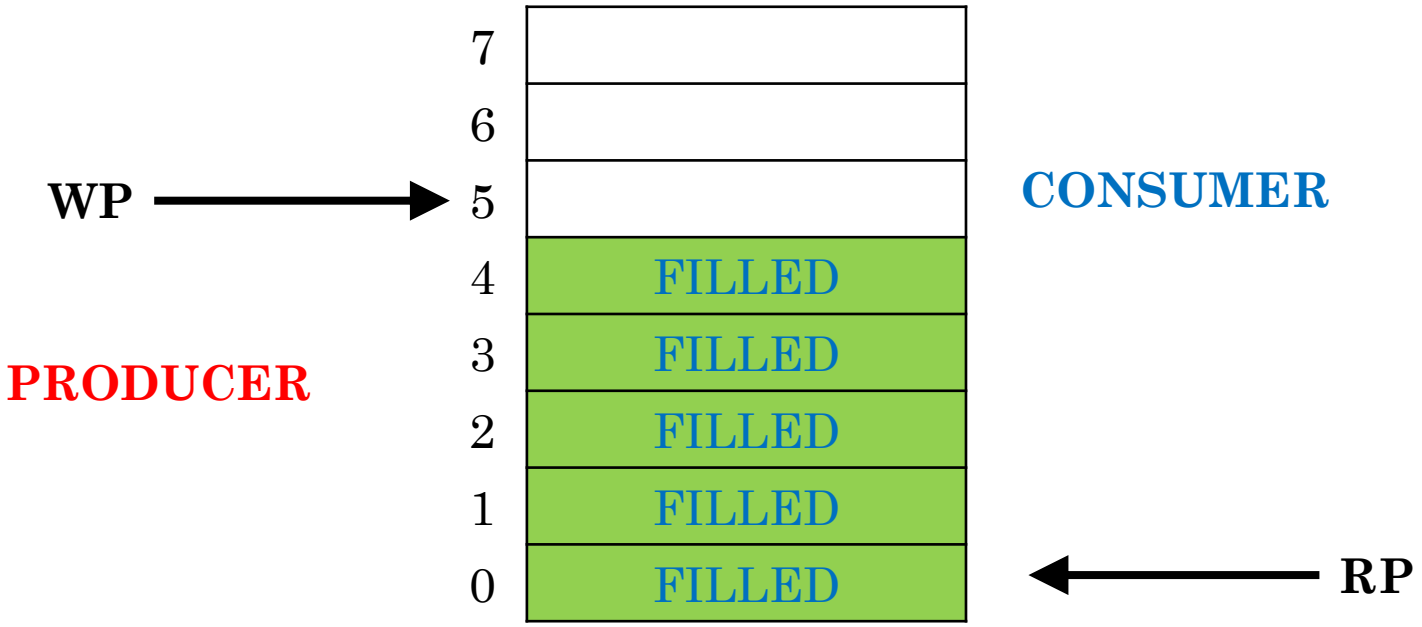
Say the producer starts depositing the data.


PRODUCER
CONSUMER
7
6
5
4
3
2
1
0
FILLED
WP
RP

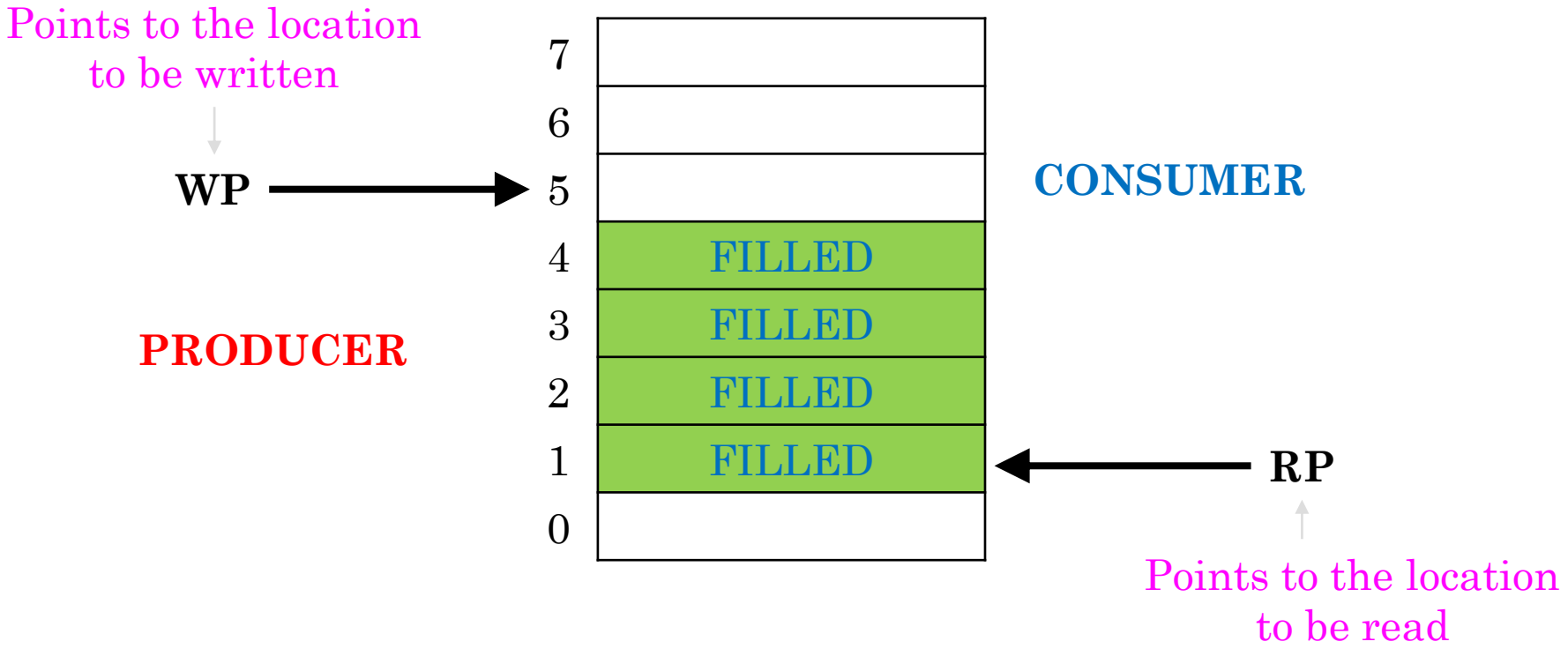

PRODUCER
CONSUMER
7
6
5
4
3
2
1
0
FILLED
FILLED
WP
RP

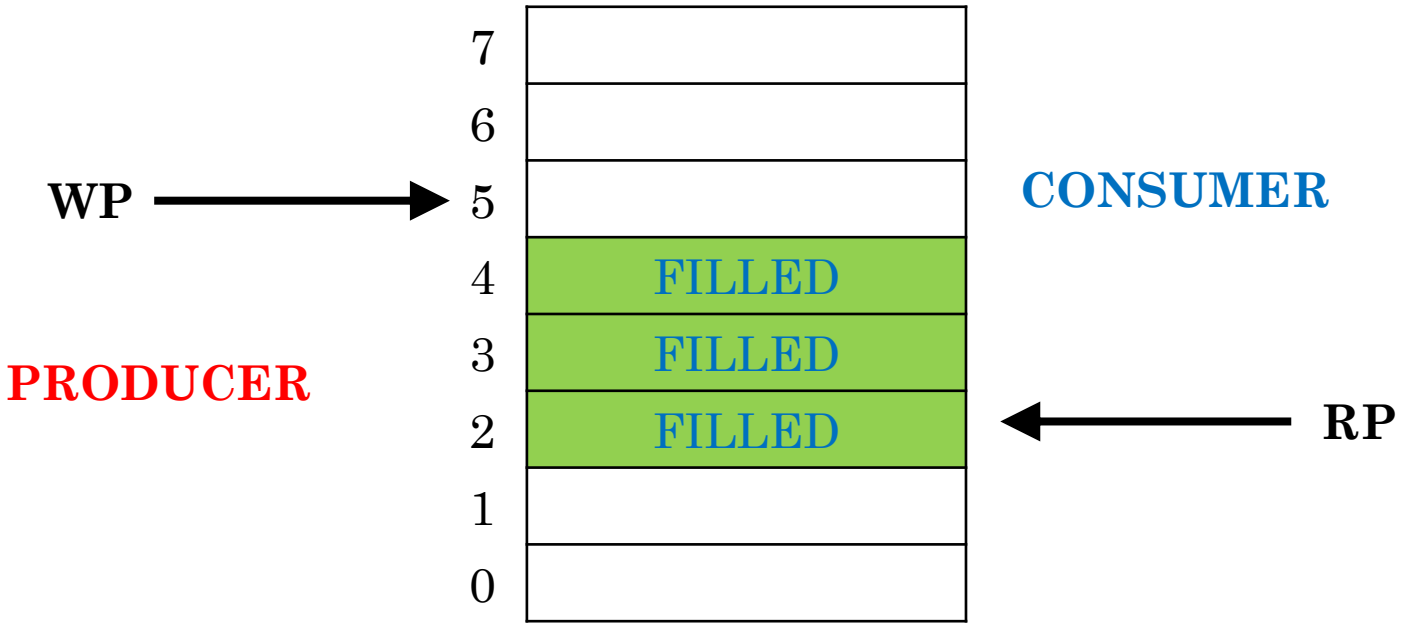

PRODUCER
CONSUMER
WP
RP
7
6
5
4
3
2
1
0
FILLED
FILLED
FILLED

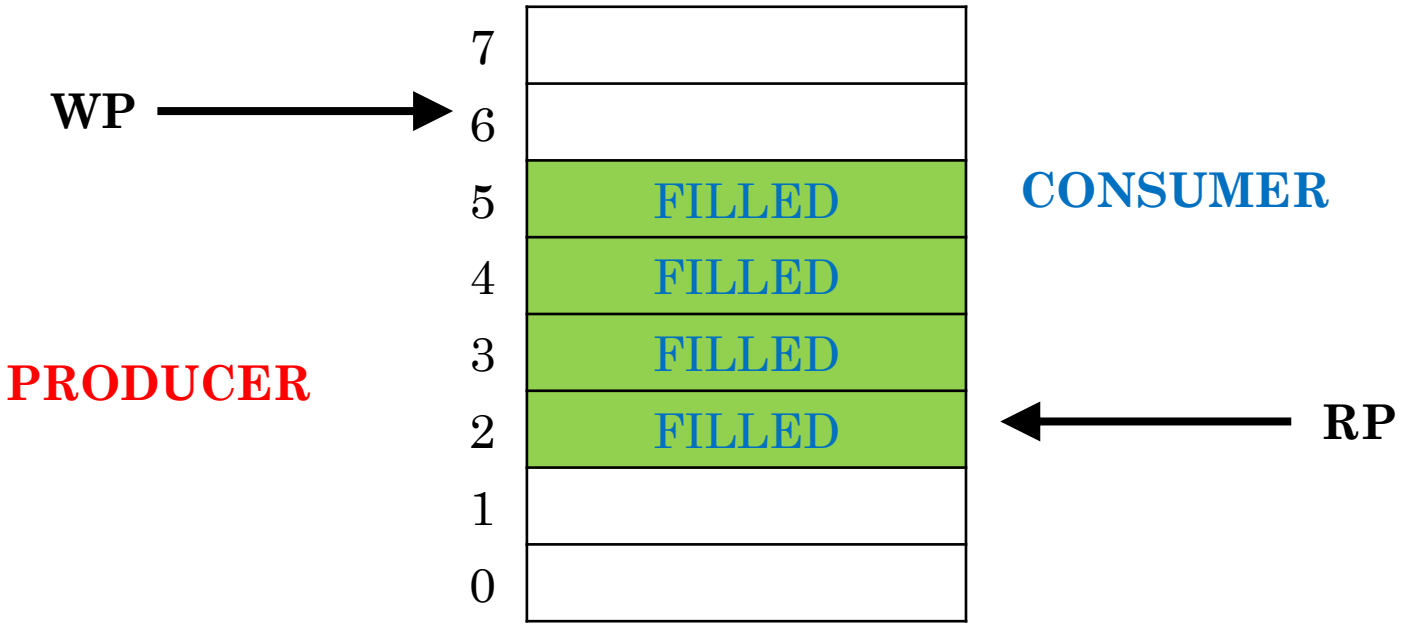

7
6
5
4
3
2
1
0
PRODUCER
CONSUMER
WP
FILLED
FILLED
FILLED
FILLED
RP

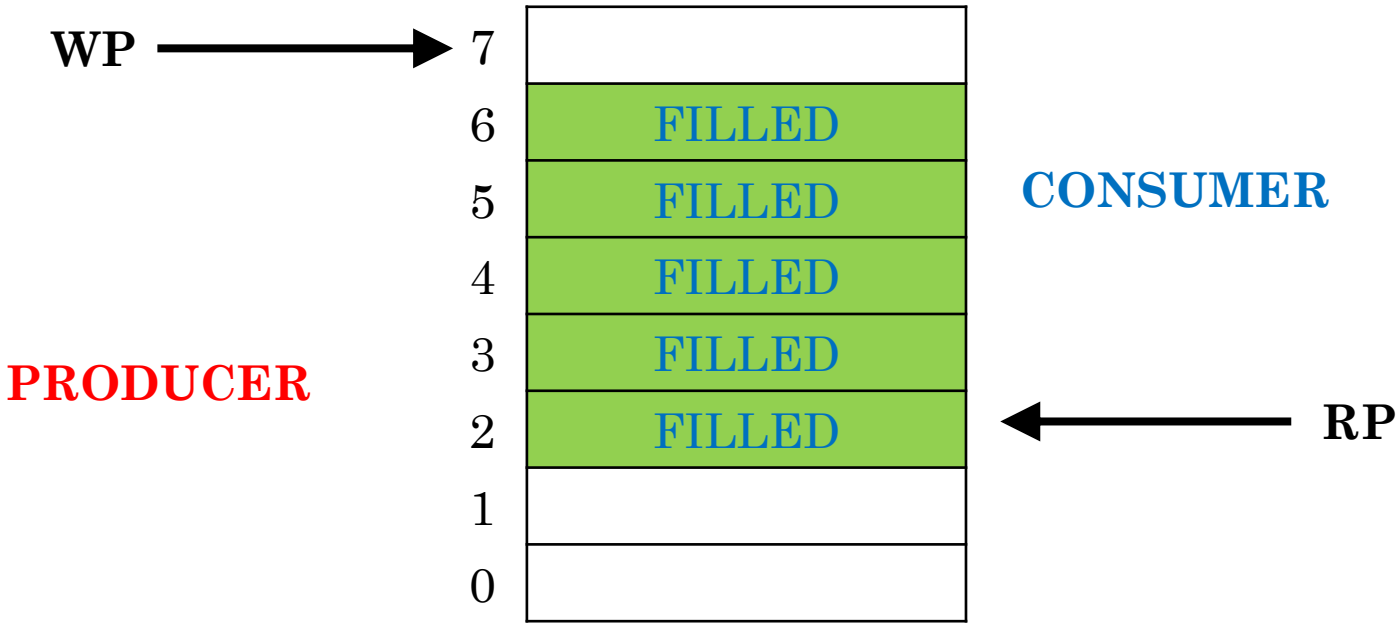


Now if the reader starts reading from this cycle.


Points to the location
to be written
WP
PRODUCER
7
6
5
4
FILLED
3
FILLED
2
FILLED
1
FILLED
0
CONSUMER
RP
Points to the location
to be read

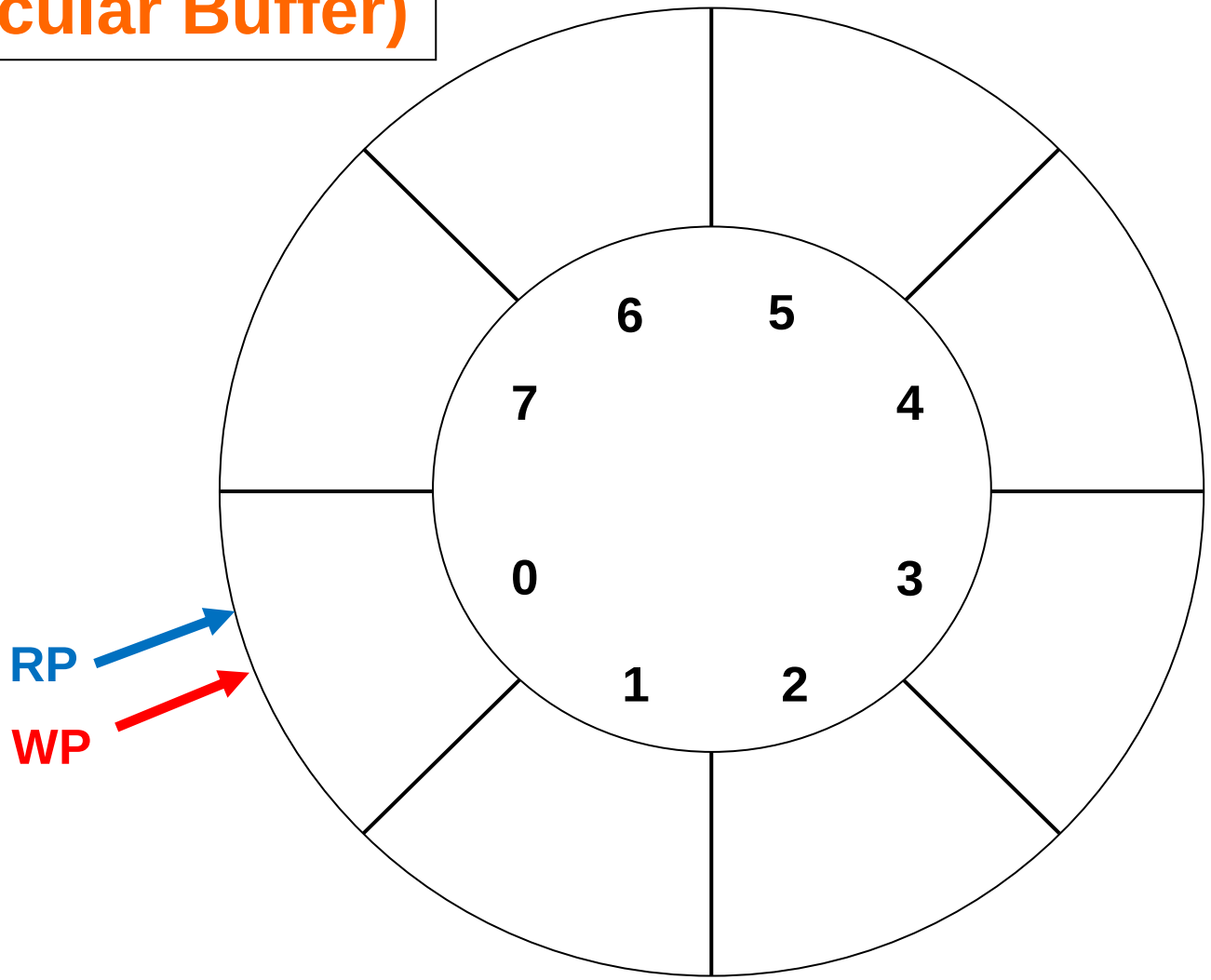

7
6
5
4
3
2
1
0
WP
CONSUMER
FILLED
FILLED
FILLED
PRODUCER
RP


WP
7
6
5
FILLED
4
FILLED
3
FILLED
2
FILLED
1
0
CONSUMER
PRODUCER
RP



Now if we write to location 7, should the WP be incremented and be allowed to point to location 0?

FIFO (= a Circular Buffer)

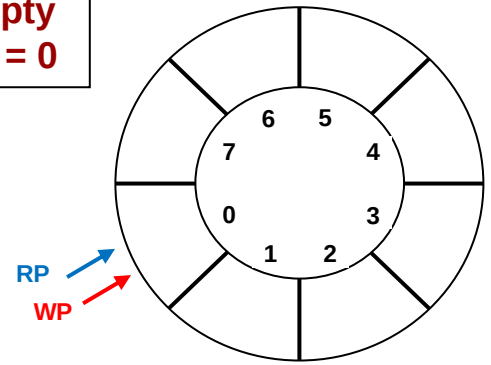


Depth (i.e. # of filled locations) = WP - RP

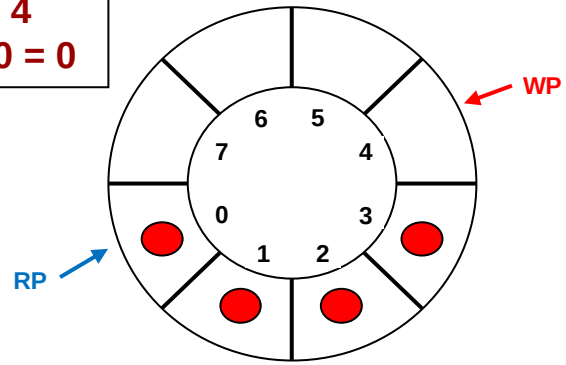
Computing Depth (# of filled locations)

- Depth = (WP – RP) mod 8

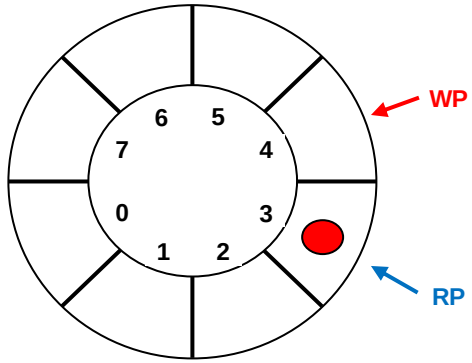
FIFO Initially Empty
D = WP-RP = 0-0 = 0



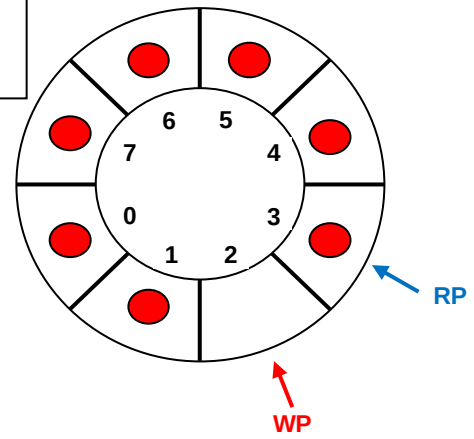
FIFO Depth = 4
D = WP-RP = 4-0 = 0



FIFO Depth = 1
D = WP-RP = 4-3 = 1

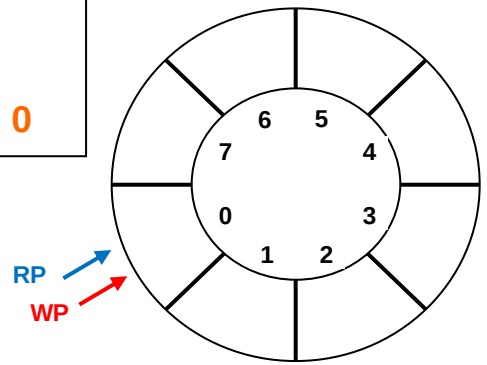


FIFO Depth = 7
D = WP-RP
= $(2-3)_{mod\ 8}$ = 7



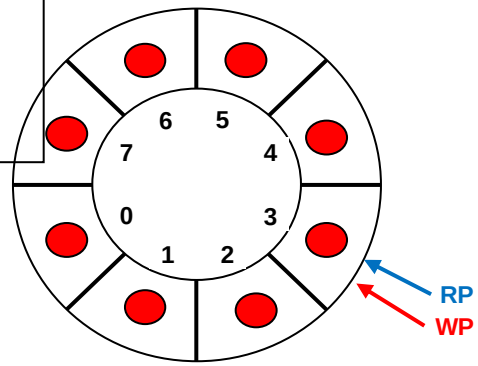
Initially Empty

FIFO depth = 0
D = WP-RP = 0-0 = 0



FIFO Full

FIFO Depth = 8
But
D = WP-RP = $(3-3)_{mod\ 8}$ = 0



WP – RP = 0 → **EMPTY** (if most recently it was almost empty)

WP – RP = 0 → **FULL** (if most recently it was almost full)

Example Thresholds for Almost Empty and Almost Full:

- Almost Empty if depth is less than or equal to 2
- Almost Full if depth is greater than or equal to 6

We need a flip-flop to record whether most recently it was running **almost empty or almost full**.

Otherwise, when WP=RP you can't tell if the FIFO is Empty or Full!!

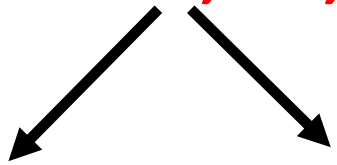
FOR AN 8-LOCATION FIFO:

POSSIBLE DEPTH VALUES ARE 9 (NOT 8):

0, 1, 2, 3, 4, 5, 6, 7, 8

BUT $|WP - RP|_{MOD\ 8}$ EXPRESSION YIELDS ONLY 8 VALUES

0, 1, 2, 3, 4, 5, 6, 7



EMPTY
(If it was seen running
almost empty most recently)

FULL
(If it was seen running
almost full most recently)

MOD SUBTRACTOR

How do we design a MOD Subtractor?

We are all taught in our elementary school how to do subtraction but do you think subtracters are built using that very technique?
What is a posted-borrow subtraction?

MOD SUBTRACTOR

EXERCISE

A digital SUBTRACTOR uses a

POSTED BORROW Technique


```
 4 0 0 2
-1 2 3 4
---------
 2 7 6 8
---------
```

```
 4 0 0 2
-1 2 3 4
---------
 2 7 6 8
---------
```

```
 4 0 0 2
-7 2 3 4
---------
 6 7 6 8
---------
```

SUMMARY

An ordinary n-bit subtractor with its borrow output ignored is a Mod-2^n subtractor.

MOD SUBTRACTOR

SOLUTION

A digital SUBTRACTOR uses a

POSTED BORROW Technique

```
    9  9
    3 10 10 10
   4 0 0 2
  -1 2 3 4
  ---------
   2 7 6 8
  ---------
```

```
      10 10
   -0 -1 -1 -1 10
    4 0 0 2
   -1 2 3 4
   ---------
    2 7 6 8
   ---------
```

```
          10 10 10
   -1 -1 -1 -1 10
    4 0 0 2
   -7 2 3 4
   ---------
    6 7 6 8
   ---------
```

SUMMARY

An ordinary n-bit subtractor with it's borrow output ignored is a Mod-2^n subtractor.

FIFO is like a queue:

Next Empty Slot
in the queue.
(Tail of the queue)

Similar to WP

WP

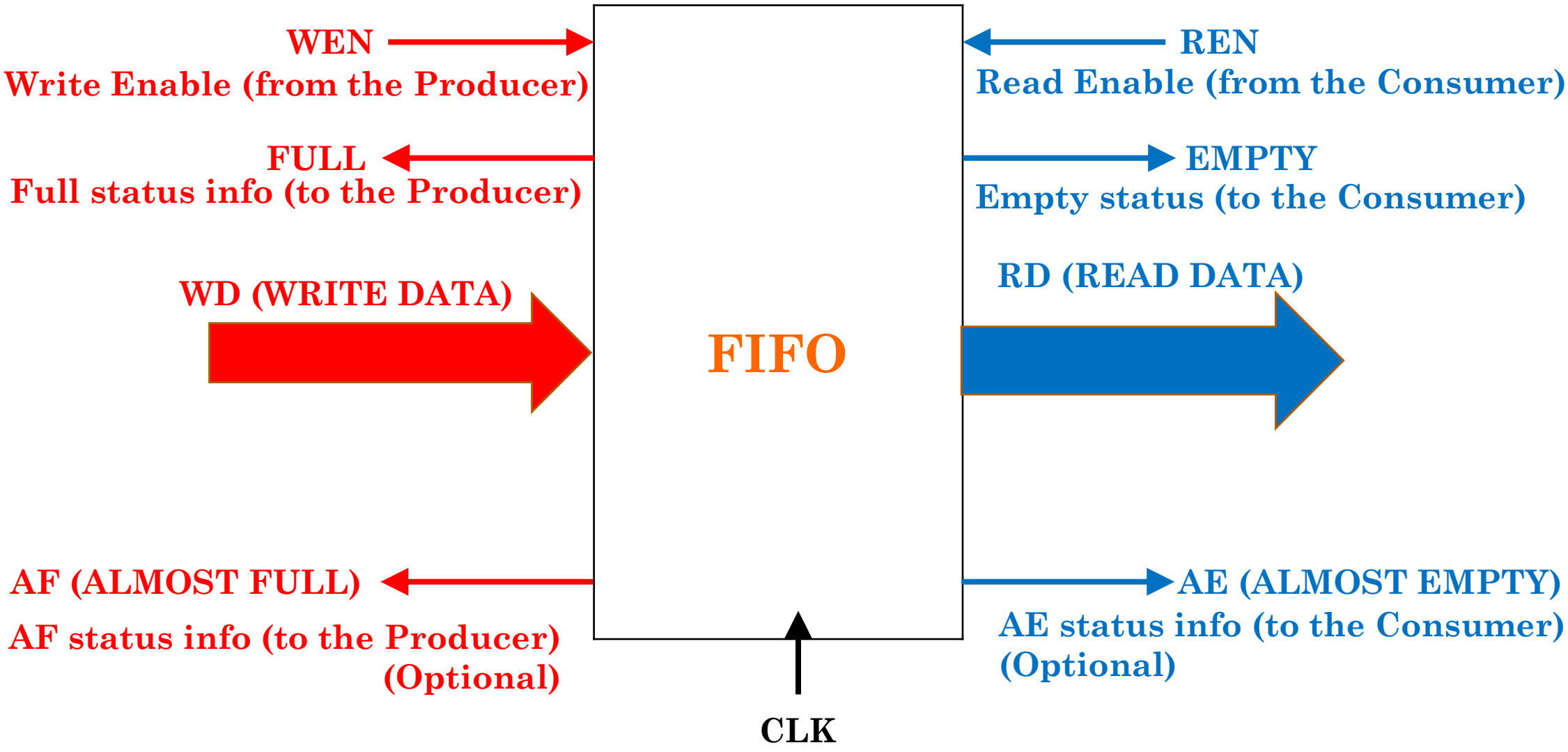
Senior-most person in
the queue
(Head of the queue)

7	
6	
5	
4	FILLED
3	FILLED
2	FILLED
1	FILLED
0	FILLED

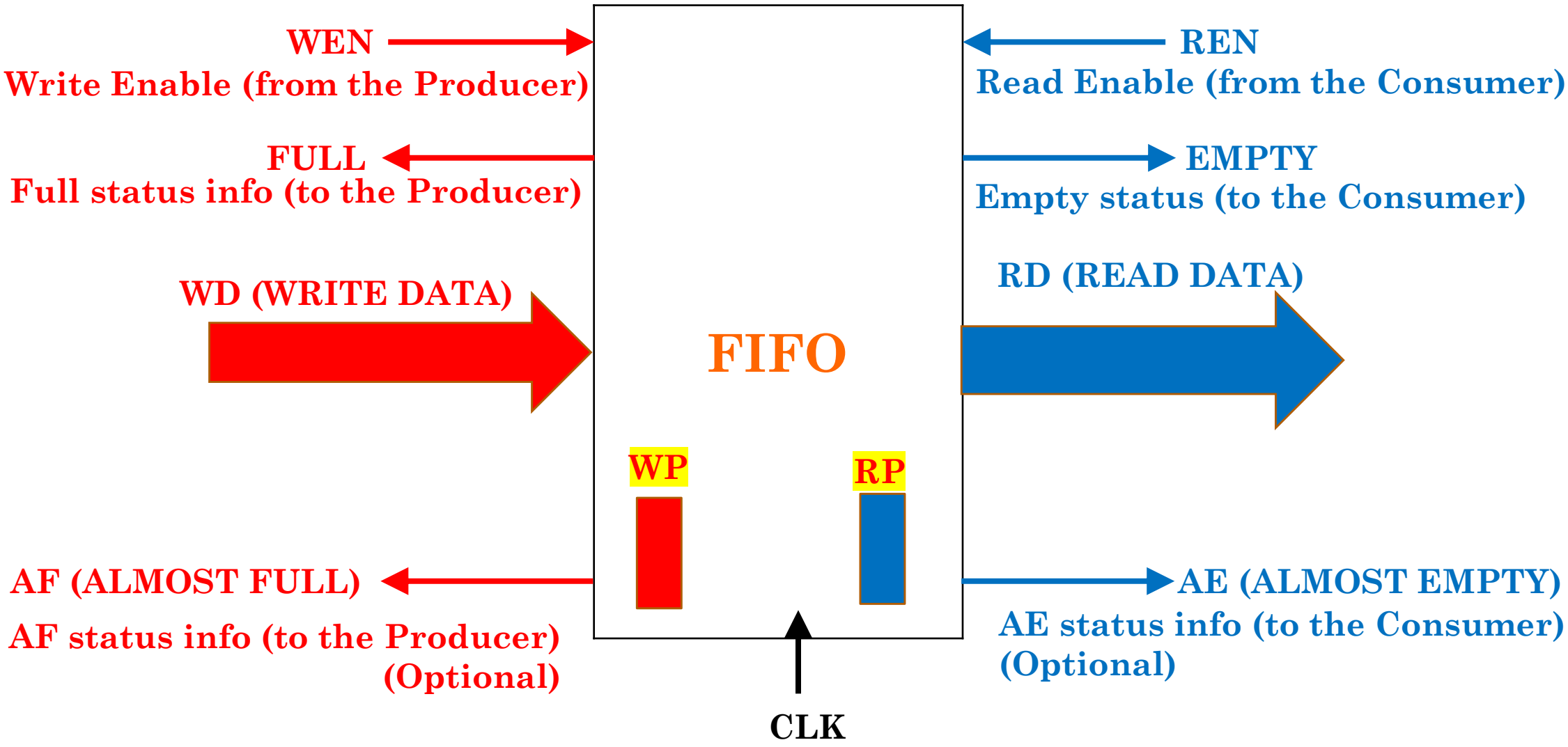
Similar
to RP

RP

FIFO PIN-OUT


WEN
Write Enable (from the Producer)
FULL
Full status info (to the Producer)
WD (WRITE DATA)
AF (ALMOST FULL)
AF status info (to the Producer)
(Optional)
FIFO
CLK
REN
Read Enable (from the Consumer)
EMPTY
Empty status (to the Consumer)
RD (READ DATA)
AE (ALMOST EMPTY)
AE status info (to the Consumer)
(Optional)

Since the WP and RP are part of the FIFO, the FIFO PIN-OUT does not change if we change the FIFO depth from 8 locations to 8K locations!

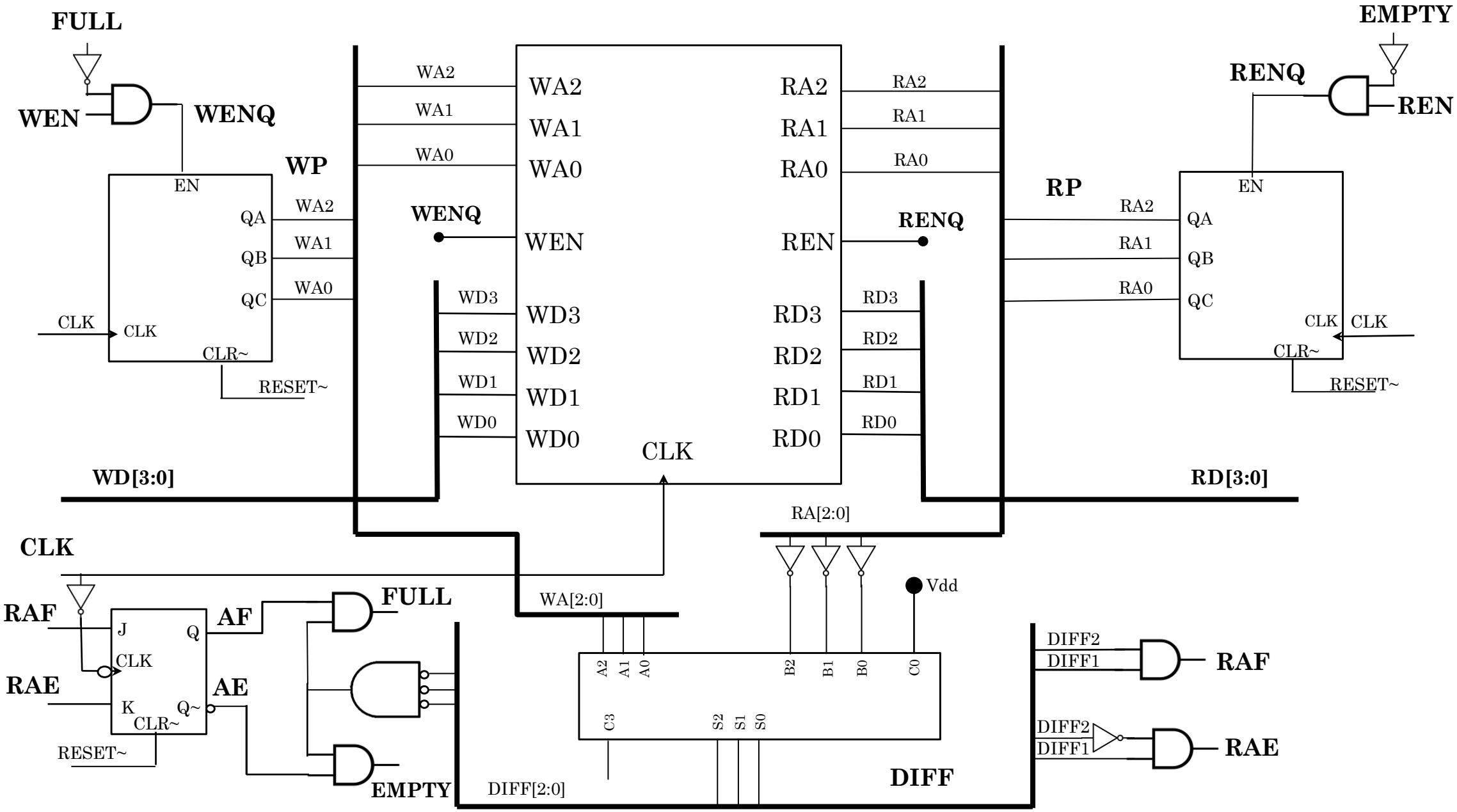


DETAILED DESIGN OF A SINGLE-CLOCK 8X4 FIFO

We illustrate a detailed design using schematic components .

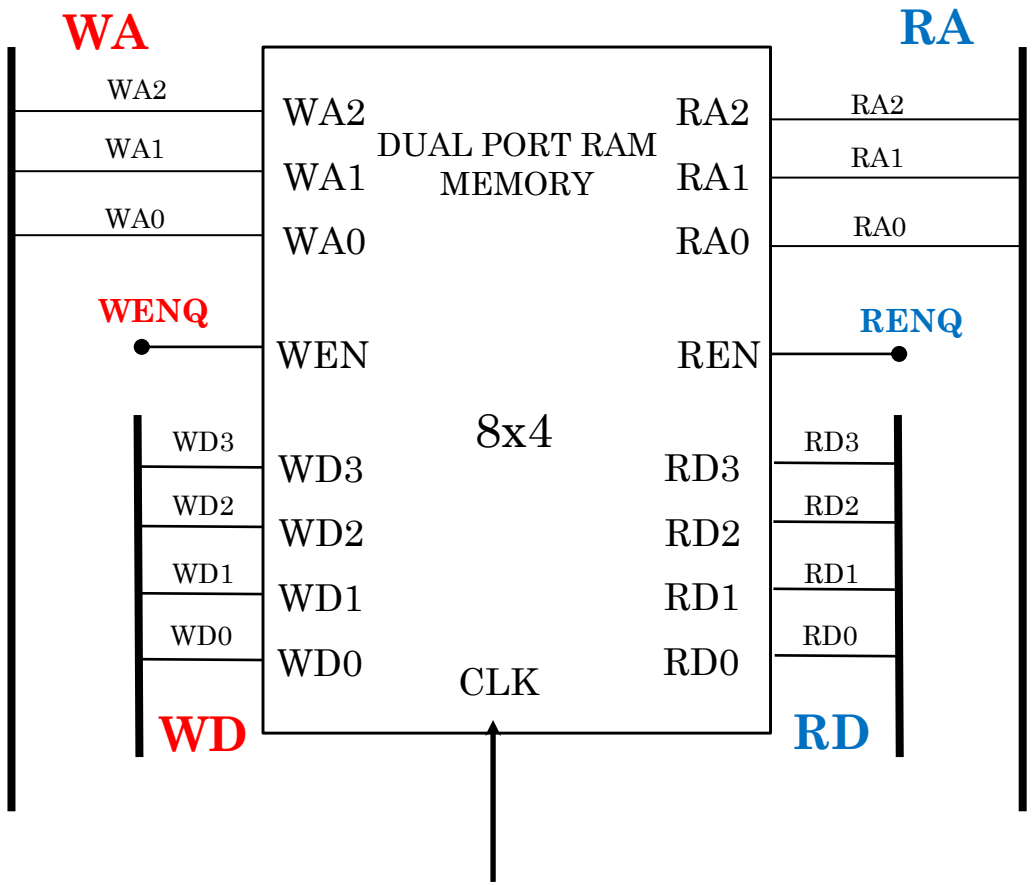
We use here
-- an 8x4 register array for the FIFO storage,
-- two 3-bit counters for the WP and RP pointers, and
-- a 3-bit subtractor.

Synchronous FIFO (Common clock for WRITE & READ)



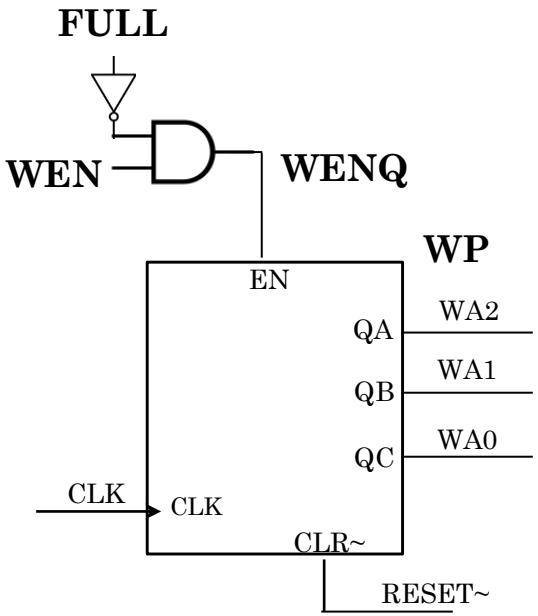
FIFO Storage

Register Array acting as a Dual Port Memory (a write-only (WO) port and a read-only (RO) port)

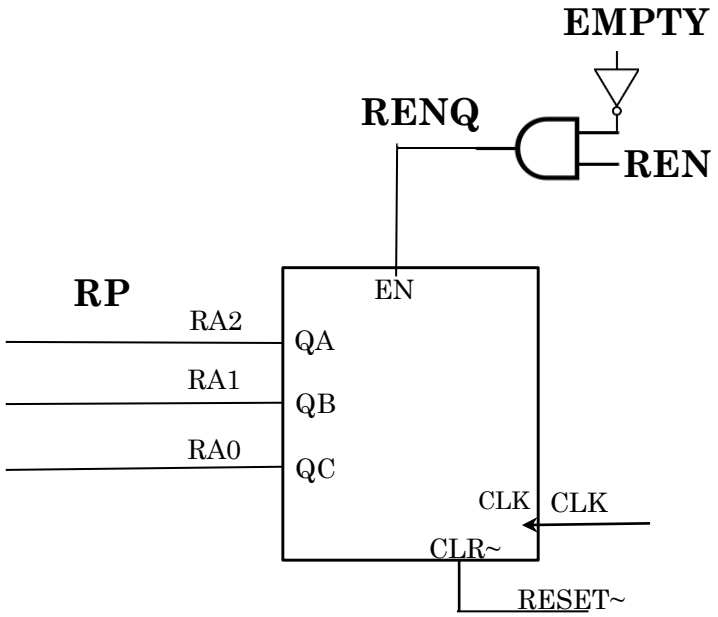


WP and RP pointers – when to increment them

WP



RP

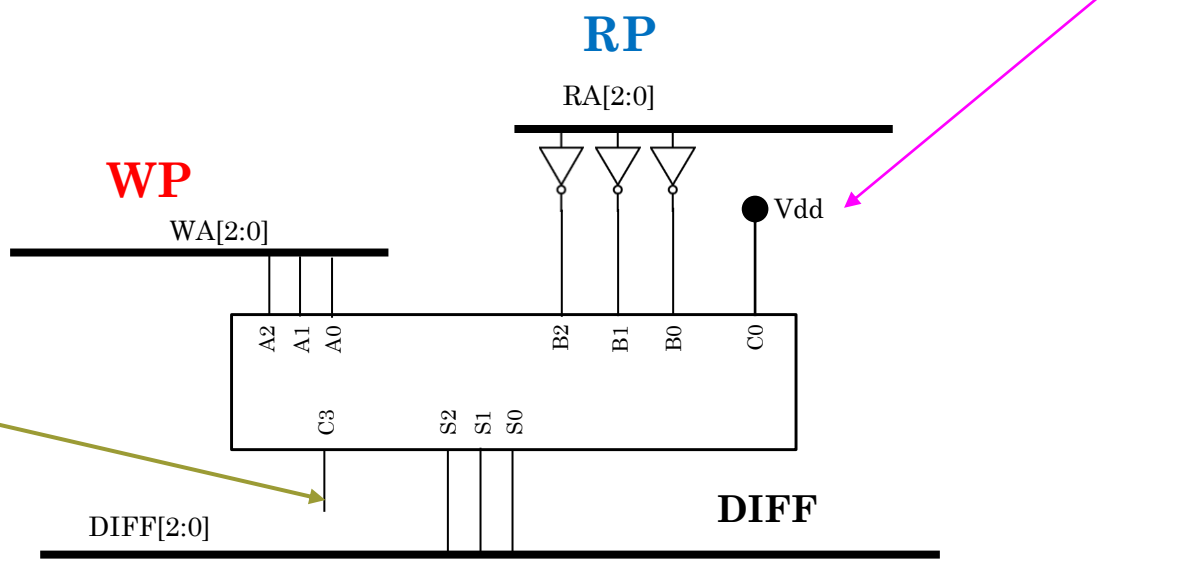


Computing Depth

A ripple carry adder turned into a subtractor

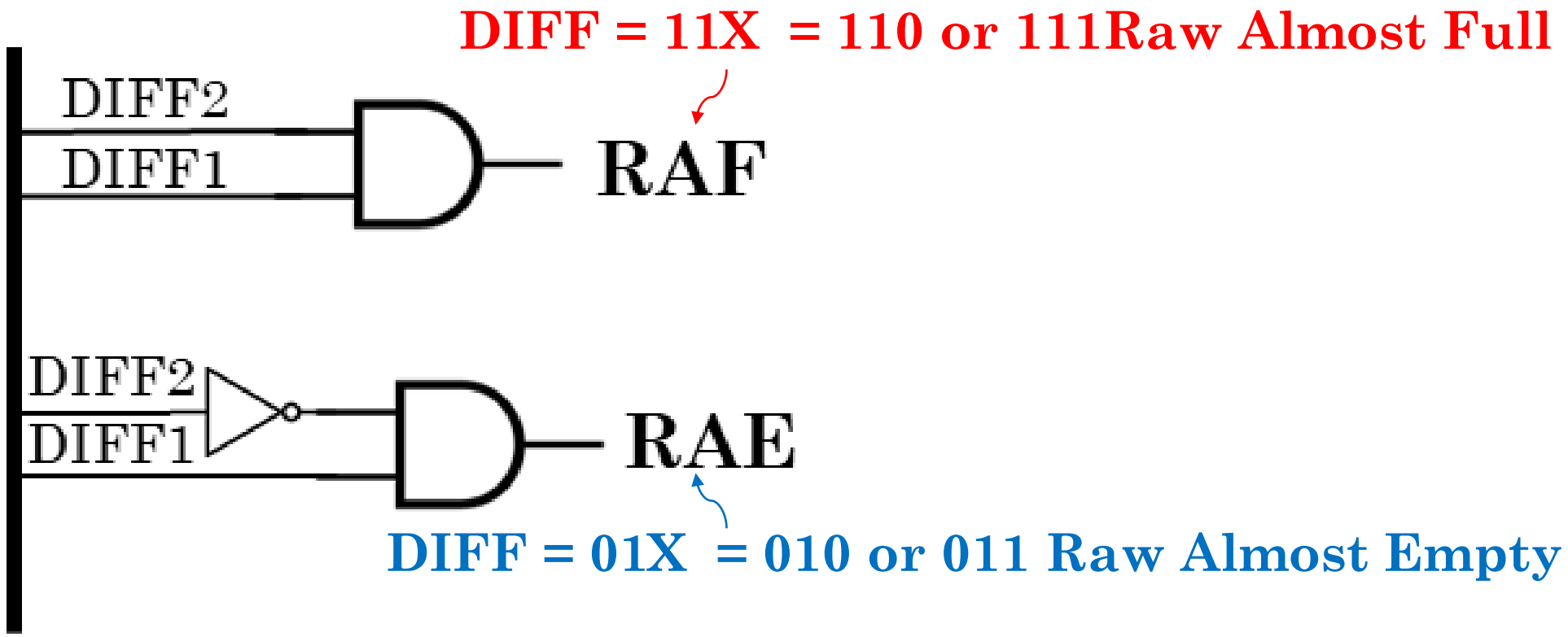
$$X - Y = X + Y' + 1$$

WA_2 WA_1 WA_0 - RA_2 RA_1 RA_0



Ignore this Borrow~ as we are building a Mod-8 subtractor

RAF and RAE



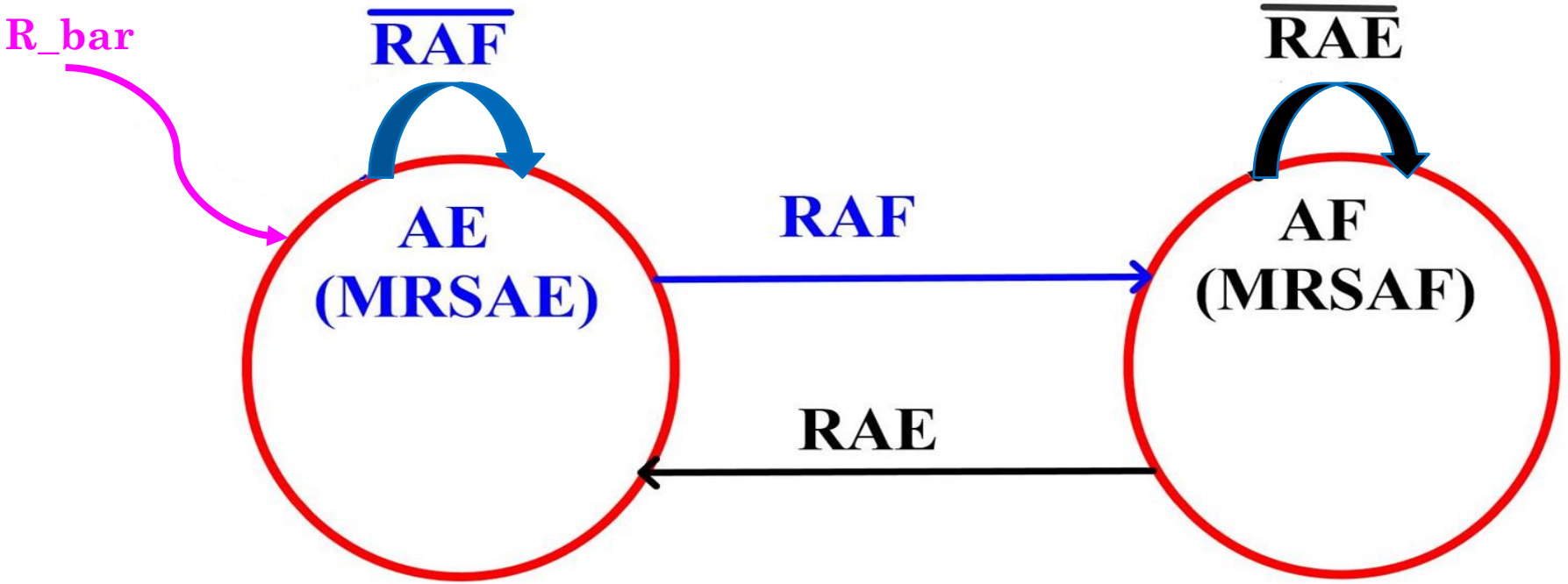
Notice that "000" is excluded from activating RAF or RAE, as DIFF = 000 is the ambiguous situation that we are trying to disambiguate.

- **We need to record (register) the RAF and RAE to form registered AF and registered AE.**

- **AF = Almost Full = MRSAF =**
  Most Recently Seen the FIFO running Almost Full
  AE = Almost Empty = MRSAE =
  Most Recently Seen the FIFO running Almost Empty

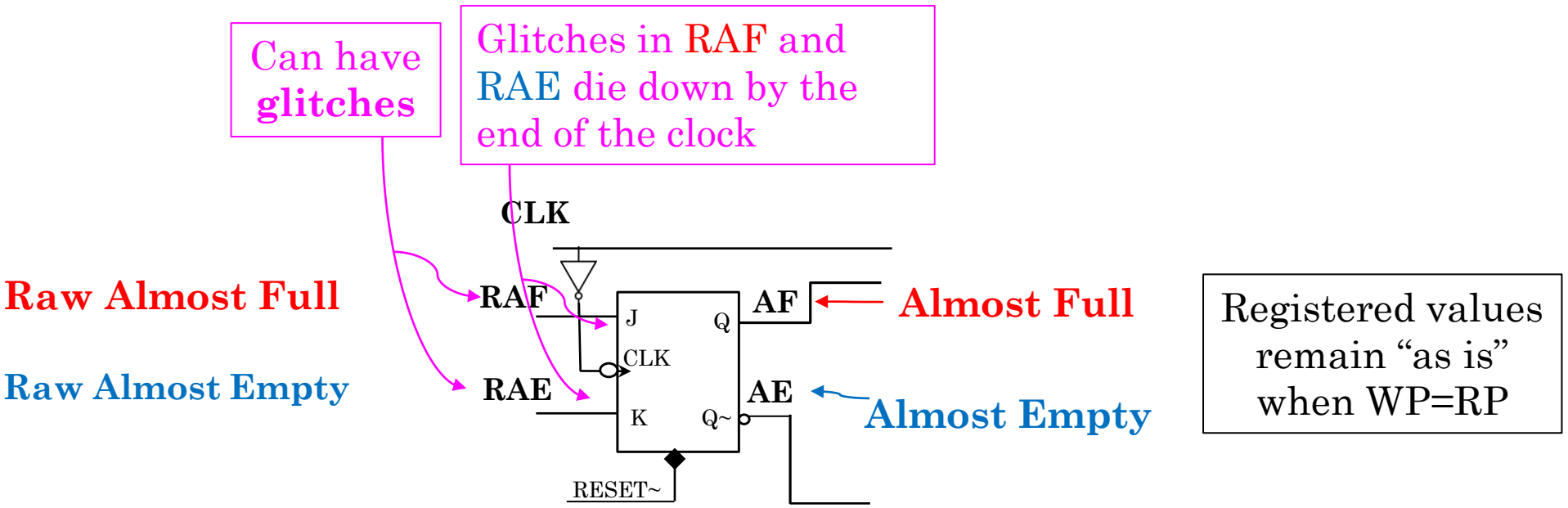
- **We need a 2-state state machine, which can be implemented using either one FF (using the encoded state assignment method) or two FFs (using the one-hot coded state assignment method).**

The state diagram is the same irrespective of the implementation method.



Please notice that the FIFO may be half-full and it is possible that you had seen most recently the depth crossing (lingering around) the low-threshold or the high-threshold. This (in itself) does not create any ambiguous situation. We make use of AE/AF information when the ambiguous depth situation (WP-RP = 0) arises.
Before the FIFO becomes FULL, it would be almost full a little before.
Before the FIFO becomes empty, it would be almost empty a little before.

By using a JK FF and the encoded state assignment method, we can arrive at the NSL heuristically. It's very simple. Just connect RAF to J and RAE to K.



N-bit pointers
vs
(N+1)-bit pointers

Another method [called the (n+1)-bit method] to remove (instead of solving) the ambiguity caused by WP-RP=0

- Instead of remembering whether most recently the depth was lingering around the almost empty threshold or the almost full threshold, in order to disambiguate the ambiguous situation caused by WP-RP = 0, we can avoid the ambiguous situation totally as follows.
- For the 8-location FIFO, we used 3-bit pointers for the WP and the RP. And the 3-bit subtraction (WP-RP)mod_8 produced an 8-valued result [0, 1, 2, 3, 4, 5, 6, 7], where as the depth has nine values, namely [0, 1, 2, 3, 4, 5, 6, 7, 8]. This led to the ambiguous situation.
- Now suppose we deliberately use a 4-bit pointer for the WP and a 4-bit pointer for the RP. Then the 4-bit subtraction (WP-RP)mod_16, which can potentially produce 16 values (0-15), will produce all the 9 legal values (0-8) and will never produce the 7 illegal values (9 through 15). There is no ambiguity to be resolved now!

Mod_16 or Mod_8 for the 8-location FIFO?

- To understand the need to do mod_16 subtraction, consider a slow producer and a fast consumer. The consumer would wait for the producer to deposit one item and he would consume it in the very next clock. So, the WP would be one step ahead of the RP for just one clock. Most of the time the WP is equal to the RP. A few examples are given below.

WP = 1; RP = 0; WP-RP = 1 - 0 = 1;
WP = 1; RP = 1; WP-RP = 1 - 1 = 0;

WP = 5; RP = 4; WP-RP = 5 - 4 = 1;
WP = 5; RP = 5; WP-RP = 5 - 5 = 0;

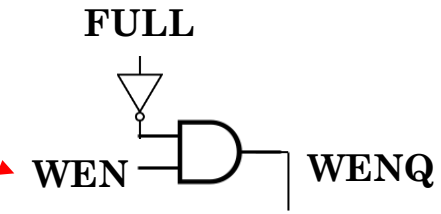
WP = 8; RP = 7; WP-RP = 8 - 7 = 1;
WP = 8; RP = 8; WP-RP = 8 - 8 = 0;

WP = 13; RP = 12; WP-RP = 13 - 12 = 1;
WP = 13; RP = 13; WP-RP = 13 - 13 = 0;

WP = 15; RP = 14; WP-RP = 15 - 14 = 1;
WP = 15; RP = 15; WP-RP = 15 - 15 = 0;

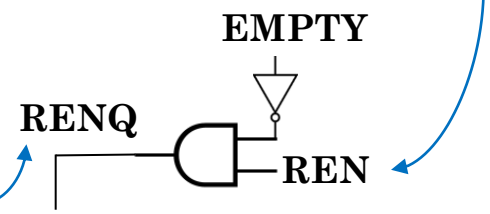
WP = 0; RP = 15; WP-RP = 0 – 15|mod16 = 1; **Note! Hence, we need to perform mod 16.**
WP = 0; RP = 0; WP-RP = 0 - 0 = 0;

WEN = Write Enable



WENQ = Write Enable Qualified
(Qualified by FULL=0)

REN = Read Enable



RENQ = Read Enable Qualified
(Qualified by EMPTY=0)

It looks like we the FIFO-designers, do not have faith in the producer designer and the consumer designer.

We provided the producer the FULL information. So, when he says "write" (by activating WEN), he must have checked to see that we are not running FULL.

Similarly, since we provided EMPTY information to the consumer designer, a responsible consumer designer would have checked to see that we are not running EMPTY before he tells us to "read" (by activating REN).

If so, why are we double-checking Full and EMPTY as shown above before activating WENQ and RENQ?

So, should we require that

the producer requests to write (makes WEN=1) only after ascertaining that FIFO is not full (FULL=0)?

And similarly, should we require that

the consumer requests to read (makes REN=1) only after ascertaining that the FIFO is not empty (EMPTY=0)**?**

YES and NO!!

YES and NO

CRITICAL PATH

At the beginning of a clock, DIFF is produced, and FULL & EMPTY are updated by the FIFO. This info is to be relayed to the PRODUCER and CONSUMER who will then activate WEN and REN. Let us try to avoid this round trip.

Because if FIFO is FULL it should not be written.
And if FIFO is EMPTY it should not be read.

Because we can do better in Timing Design.

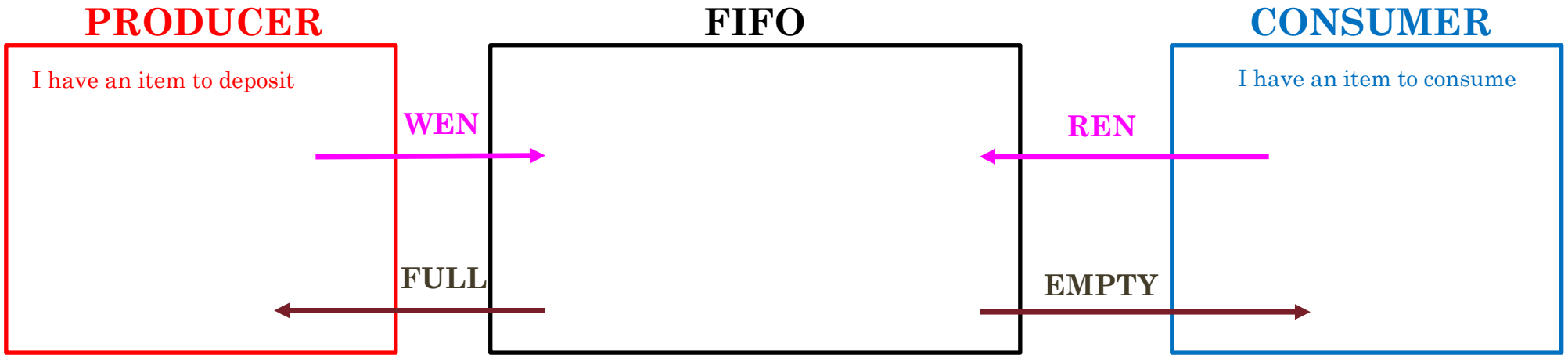
Suggestion for a better timing outcome

The producer sends a request to write (WEN=1) if he has something to write without waiting to check to see if FULL=1. Later in the clock if he finds that FULL is true, he will **RETRY** to write the same data again.

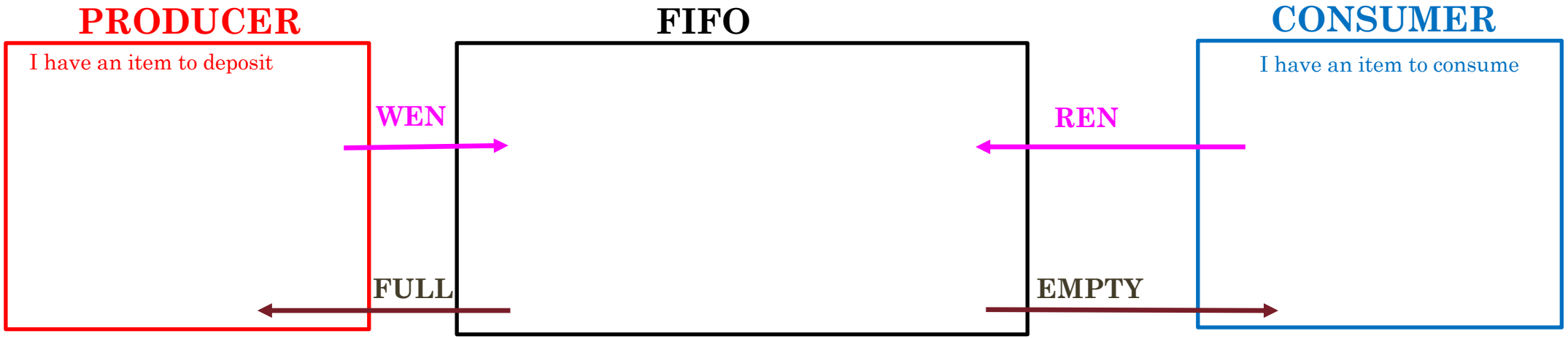
Avoiding LONG critical path (round-trip) by breaking it into two one-way trips

LONG round Trip

★ EXERCISE

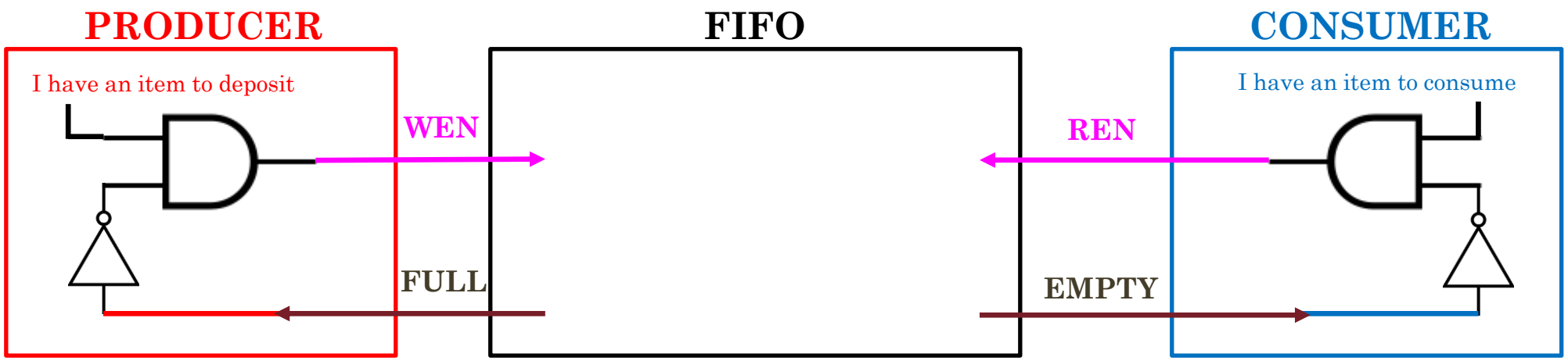


Two one-way SHORT Trips



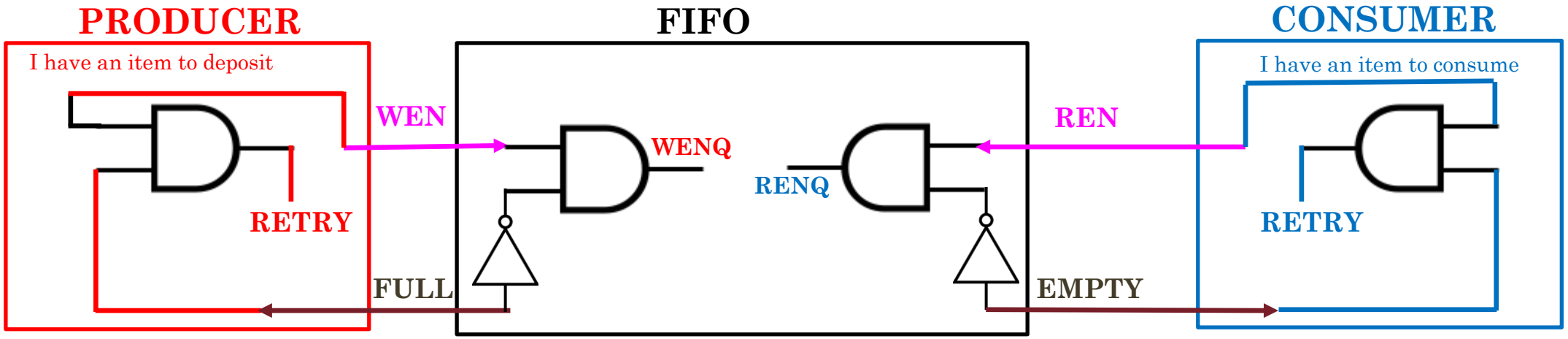
Avoiding LONG critical path (round-trip) by breaking it into two one-way trips

LONG round Trip



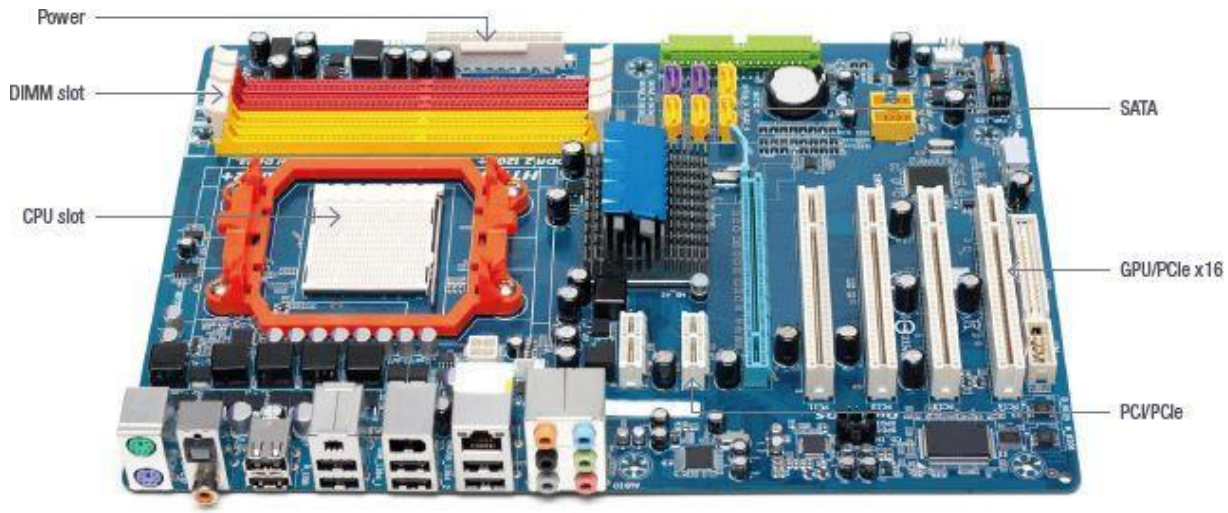
SOLUTION

Two one-way SHORT Trips

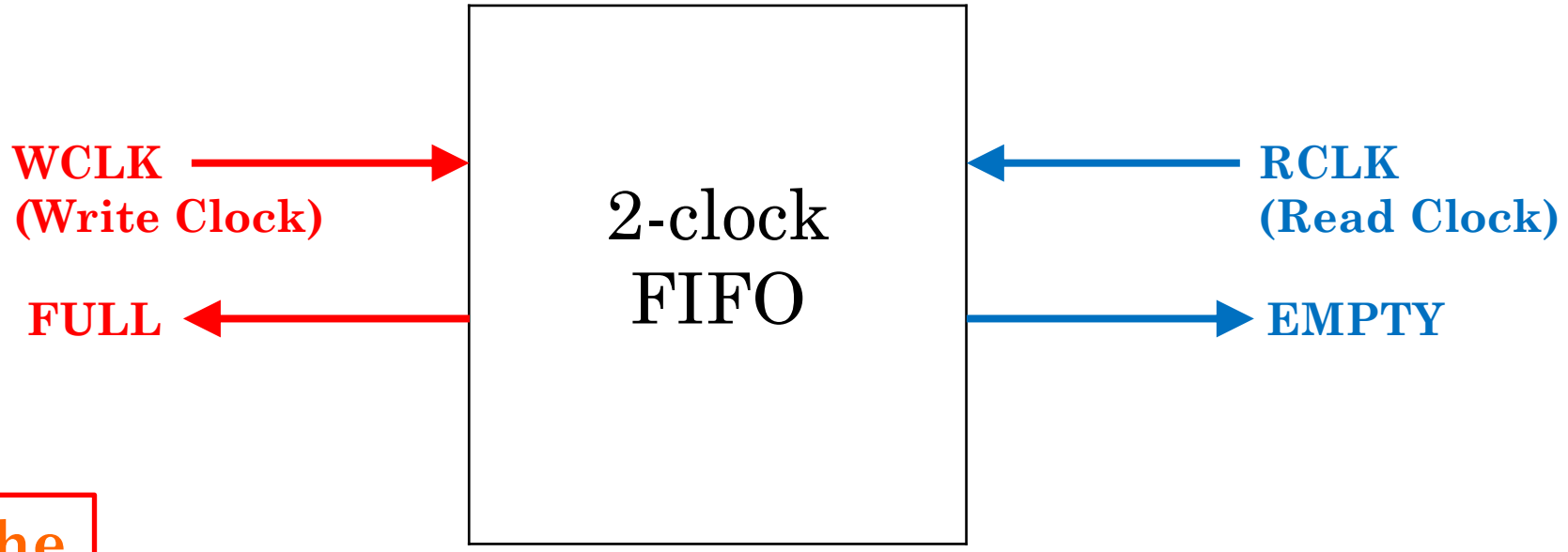


CDC (Clock Domain Crossing)

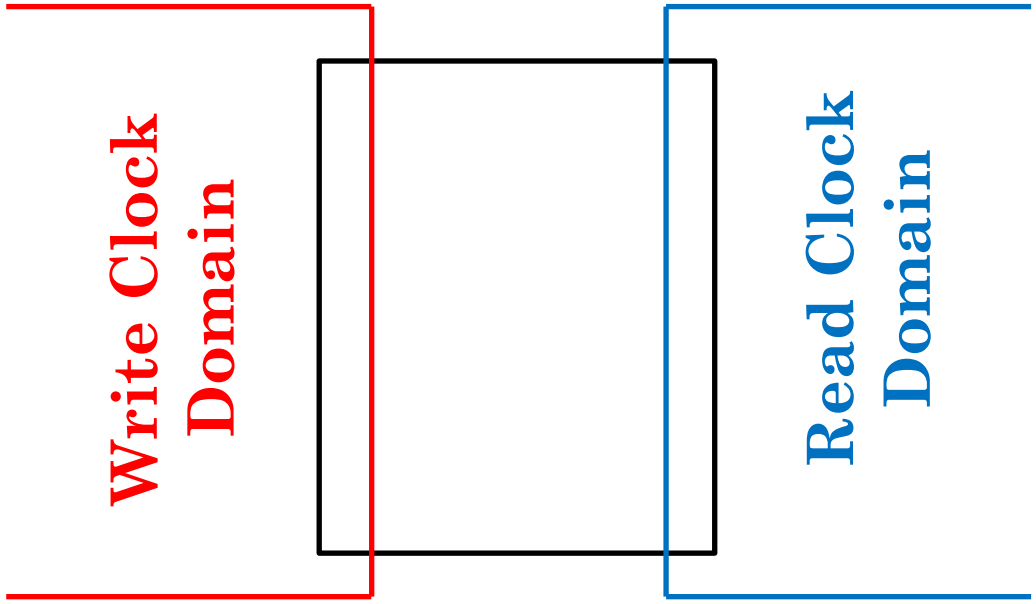
- Clock Domain Crossing (CDC) is a very common situation every digital designer encounters.
- On a mother board of a computer, you can easily find 4 to 10 different subsystems working on different clocks. Example: A processor may be working at 3 GHz, DDR4 at 1GHz, EPROM at 400MHz and the PCI Bus may be working at 66MHz etc.



- The **Robust** method of CDC is to use a 2-clock FIFO.

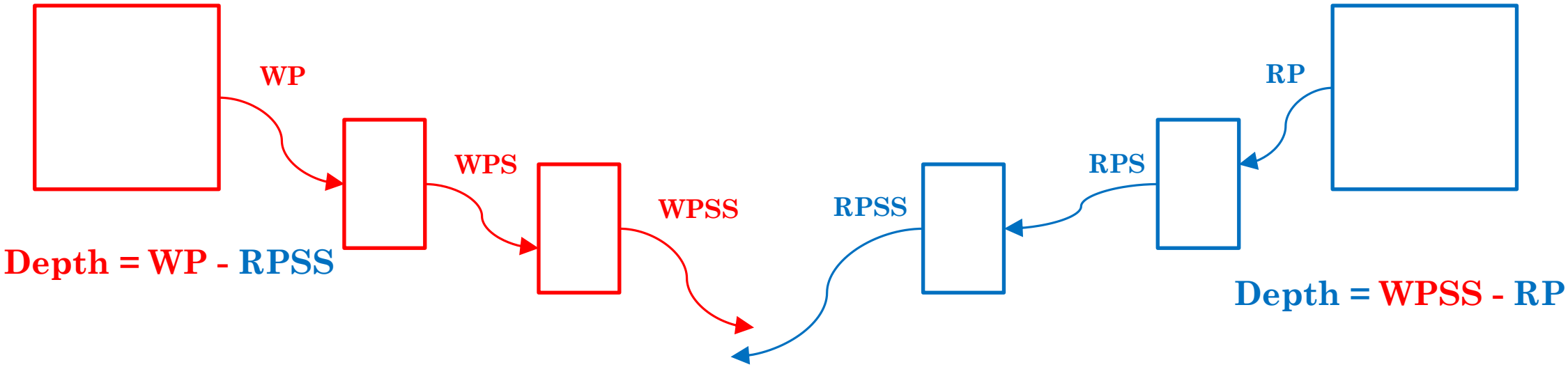


FIFO is used to bridge the separately clocked domains
An asynchronous FIFO = A two-Clock FIFO



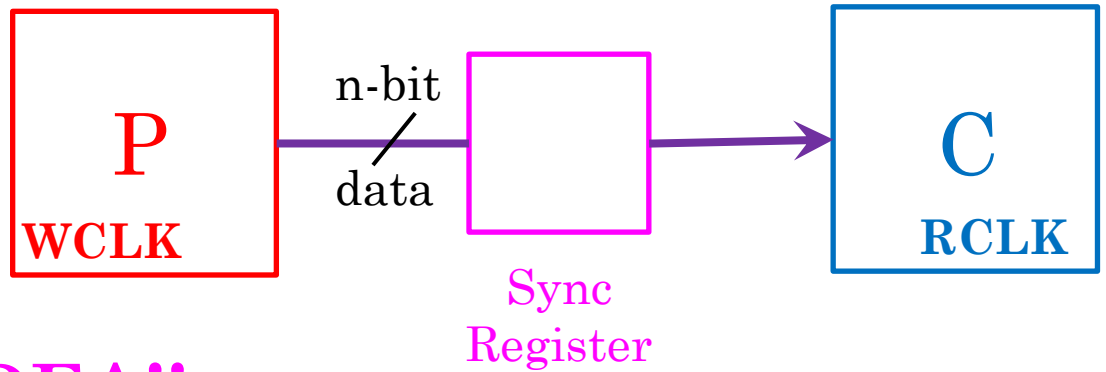
Depth Calculation for 2-clock FIFO

We just can't do $(WP-RP)_{mod <FIFO\ Depth>}$ to generate depth value as they are generated in different clock domains. Also, we can't say when (even for a short moment) the WP and RP values are stable and valid together.

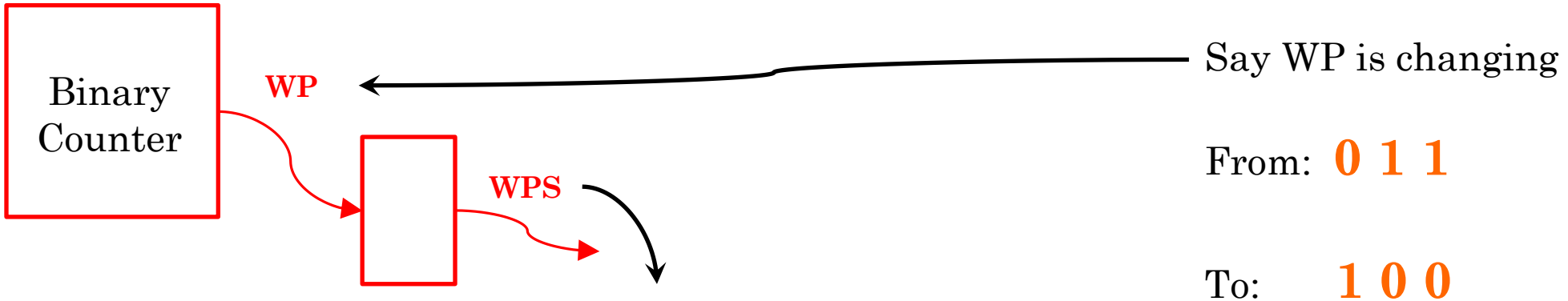


DELAY: Safe or Unsafe?

Never ever try to synchronize a multibit data item when you are crossing a clock domain

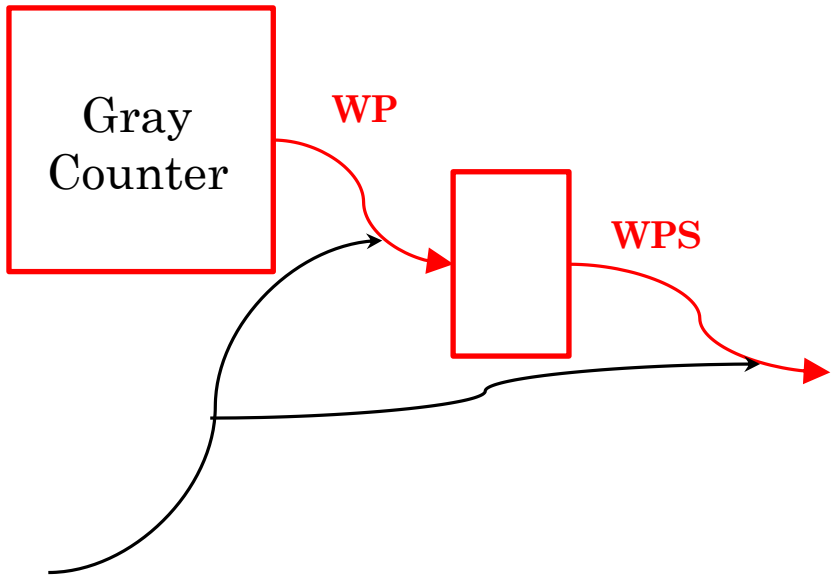


BAD IDEA!!



Can potentially be any value from 0 (000) to 7 (111) due to metastability of the flip-flop.

For SEQUENTIALLY changing data such as WP and RP use GRAY CODE

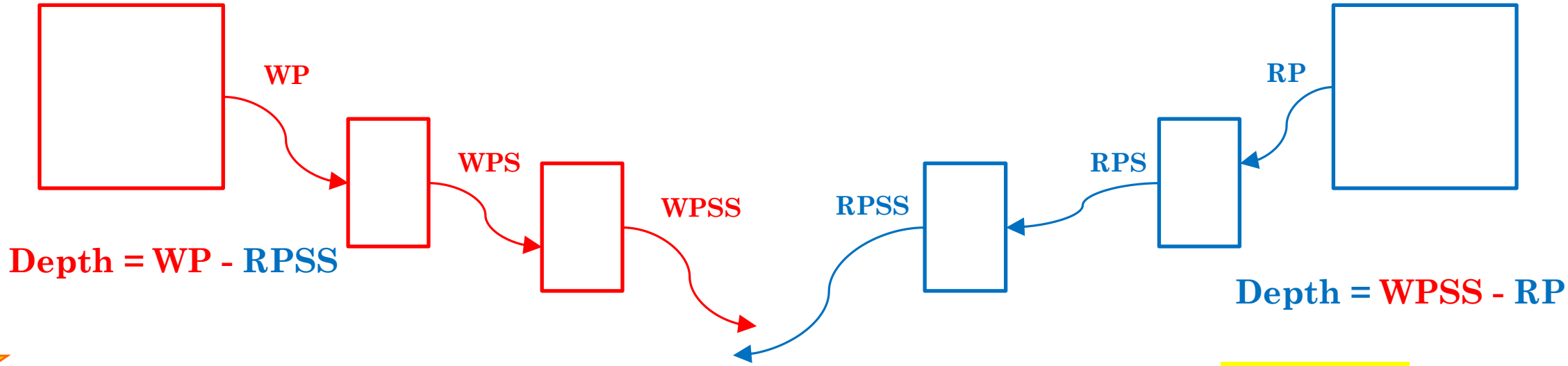


Since only 1-bit changes at most in **WP**, the **WPSS** will either be the **old WP** or the **new WP** and **will never be an absurd value!**

GRAY CODE:

0	0	0
		1
	1	1
		0
1	1	0
		1
	0	1
		0

SYNCHRONIZATION using GRAY CODE



EXERCISE:

Try to write a similar paragraph for a faster writer.

DELAY: Safe or Unsafe? → Safe!!

Say the consumer is faster and the FIFO is running empty for a long time. Now the WRITER just wrote a data item

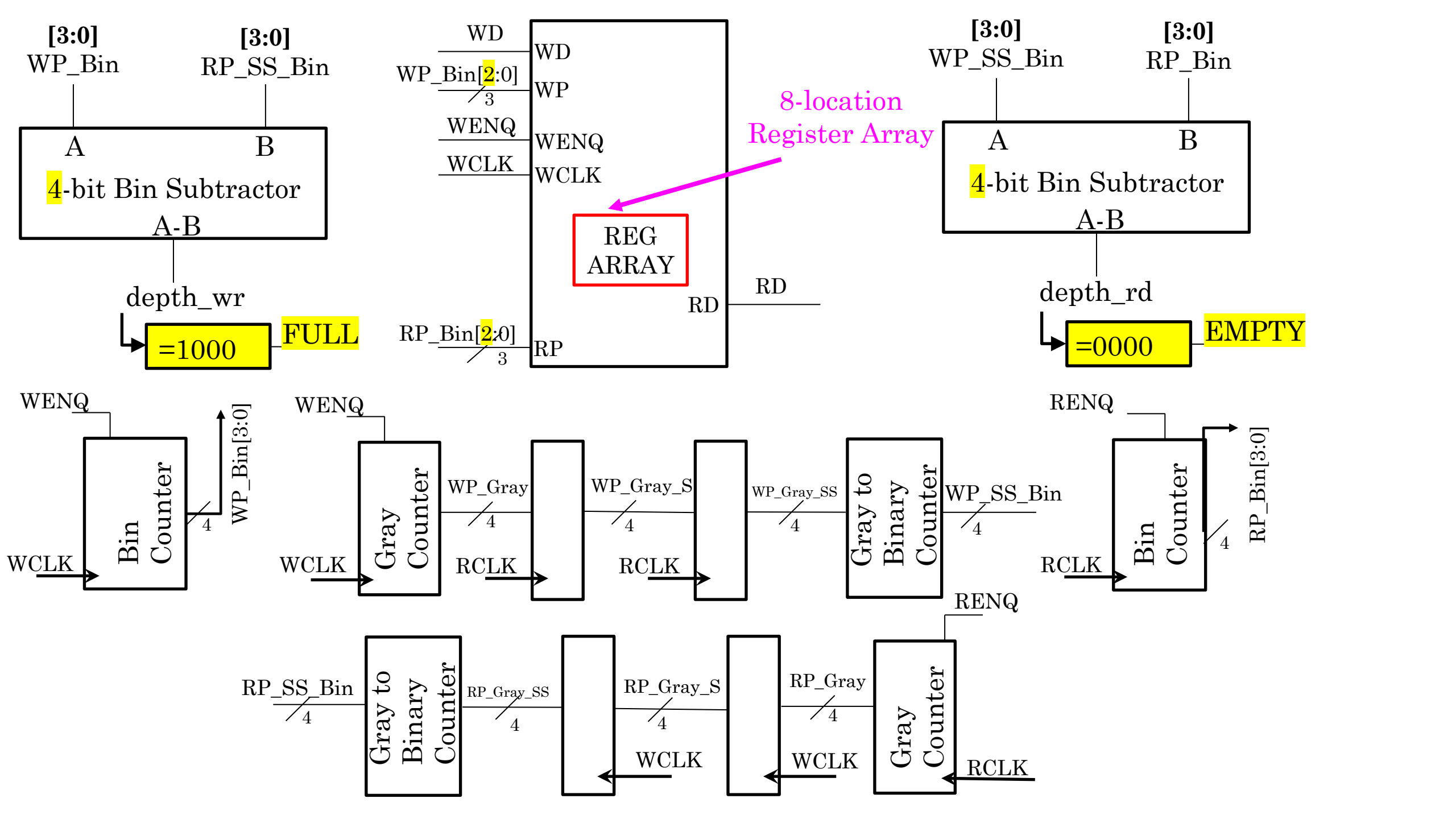
The incremented WP takes 1 or 2 clocks to reach the Read Clock domain and until that time the reader continues to believe that the FIFO is still empty.

So, he will delay consuming the data and this is safe! As long as the consumer does not consume from an empty FIFO, it is safe!

The (n+1)-bit pointer method for the 2-clock FIFO

On the next slide, we have an 8-location 2-clock FIFO with two 4-bit gray-code counters (WP_G and RP_G), two 4-bit binary counters (WP_Bin and RP_Bin), and two depth producing 4-bit subtractors (one in each of the two clock domains).

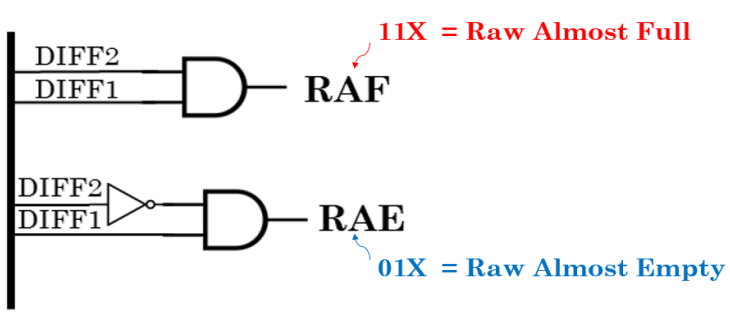
Note: The depth inferred in the write-clock domain (depth_wr) and the depth inferred in the read-clock domain (depth_rd) can differ substantially *tentatively* because of the lag in pointer exchange, but it is all on safe side!

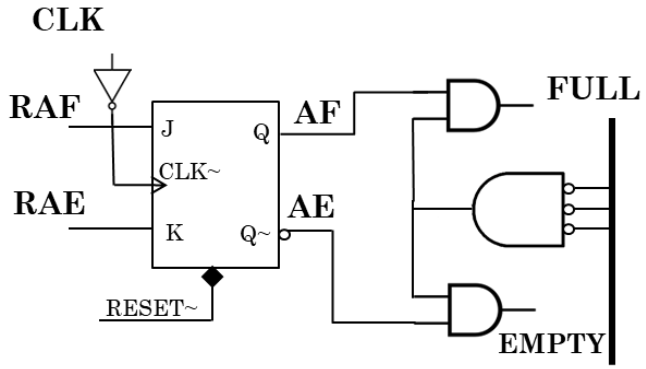

[3:0]
WP_Bin
[3:0]
RP_SS_Bin
A
B
4-bit Bin Subtractor
A-B
depth_wr
=1000
FULL
WD
WP_Bin[2:0]
3
WENQ
WCLK
WD
WP
WENQ
WCLK
8-location
Register Array
REG
ARRAY
RD
RD
RP_Bin[2:0]
3
RP
[3:0]
WP_SS_Bin
[3:0]
RP_Bin
A
B
4-bit Bin Subtractor
A-B
depth_rd
=0000
EMPTY
WENQ
Bin
Counter
4
WP_Bin[3:0]
WCLK
WENQ
Gray
Counter
WP_Gray
4
WP_Gray_S
4
WP_Gray_SS
4
Gray to
Binary
Counter
WP_SS_Bin
4
WCLK
RCLK
RCLK
RENQ
Bin
Counter
4
RP_Bin[3:0]
RCLK
RENQ
RP_SS_Bin
4
Gray to
Binary
Counter
RP_Gray_SS
4
RP_Gray_S
4
RP_Gray
4
Gray
Counter
WCLK
WCLK
RCLK

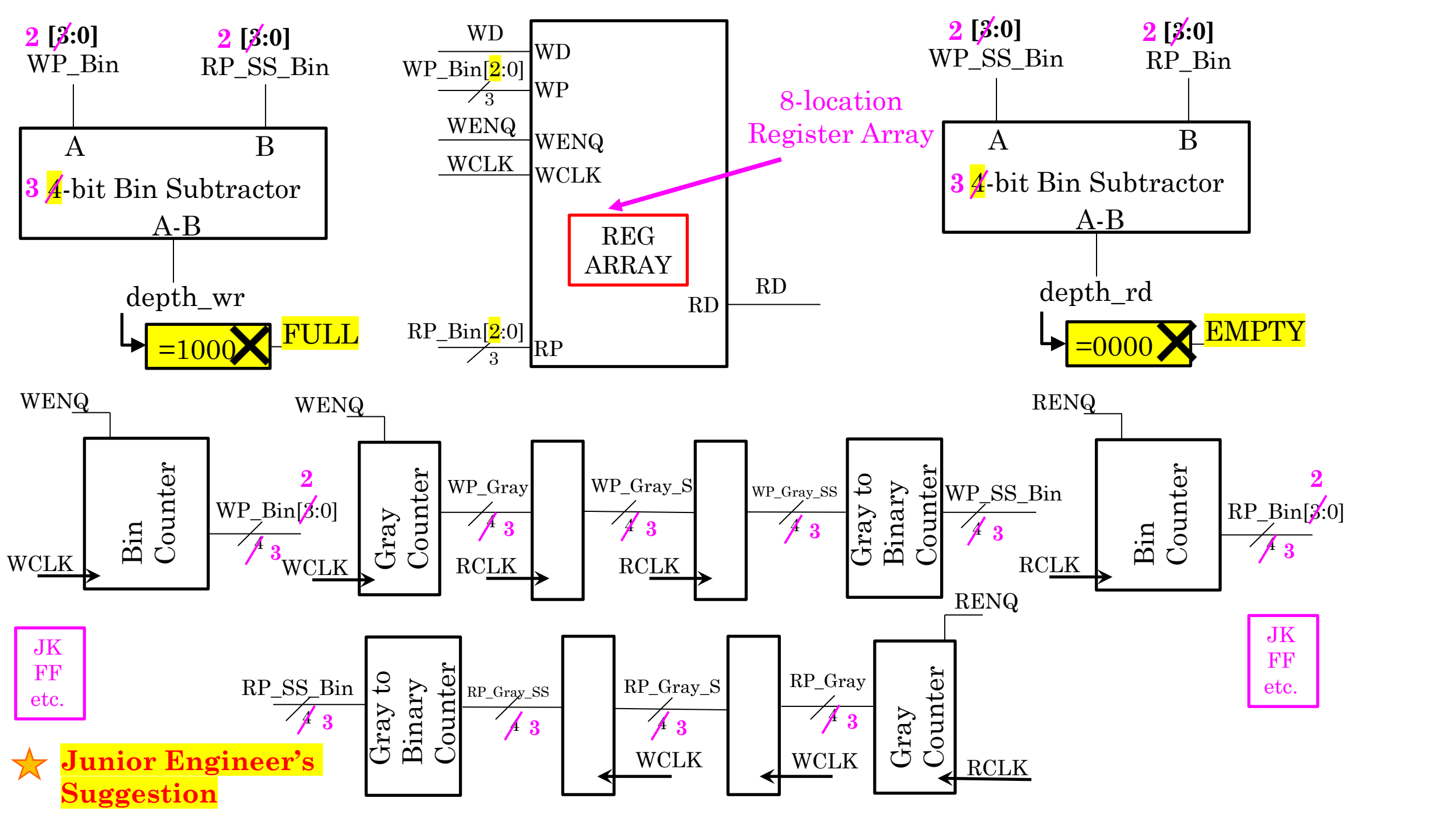
A junior engineer offers an alternative design. He says he would change all the 4-bit items to 3-bit (i.e., the counters, synchronizing registers, code converters, subtractors, etc.). In each of the two clock domains, he will have a JK flip-flop (or something similar) to remember if most recently the FIFO was running almost empty (AE) or almost full (AF) and accordingly interpret a depth of 000 as zero (Empty) or eight (Full), like what was done in a single-clock FIFO.

He says that this is cheaper so why not use this alternative?

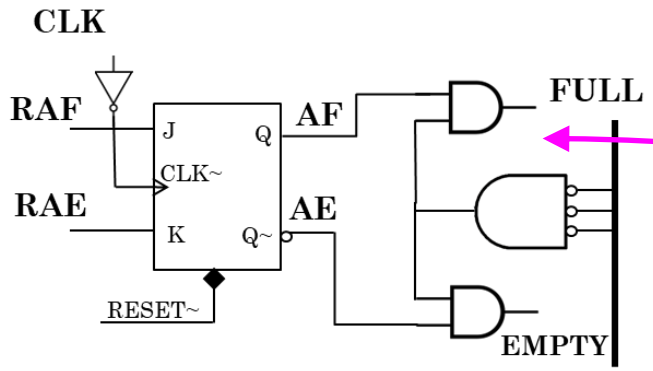
The junior engineer changed all [3:0] on the previous page to [2:0] as shown on the next slide and is about to add on each side one set of "JK FF and related circuitry" **(Similar to the single clock FIFO).**






2 [3:0]
WP_Bin
2 [3:0]
RP_SS_Bin
A
B
3 4-bit Bin Subtractor
A-B
depth_wr
=1000
FULL
WD
WP_Bin[2:0]
3
WENQ
WCLK
WD
WP
WENQ
WCLK
REG
ARRAY
RD
RD
RP_Bin[2:0]
3
RP
8-location
Register Array
2 [3:0]
WP_SS_Bin
2 [3:0]
RP_Bin
A
B
3 4-bit Bin Subtractor
A-B
depth_rd
=0000
EMPTY
WENQ
WCLK
Bin
Counter
2
WP_Bin[3:0]
4 3
WENQ
WCLK
Gray
Counter
WP_Gray
4 3
RCLK
WP_Gray_S
4 3
RCLK
WP_Gray_SS
4 3
Gray to
Binary
Counter
WP_SS_Bin
4 3
RENQ
RCLK
Bin
Counter
2
RP_Bin[3:0]
4 3
JK
FF
etc.
RP_SS_Bin
4 3
Gray to
Binary
Counter
RP_Gray_SS
4 3
RP_Gray_S
4 3
WCLK
RP_Gray
4 3
WCLK
RENQ
Gray
Counter
RCLK
JK
FF
etc.
Junior Engineer's
Suggestion

The senior engineer told him not to do that as that would create a deadlock!



DON'T DO THIS

WP – RP = 0

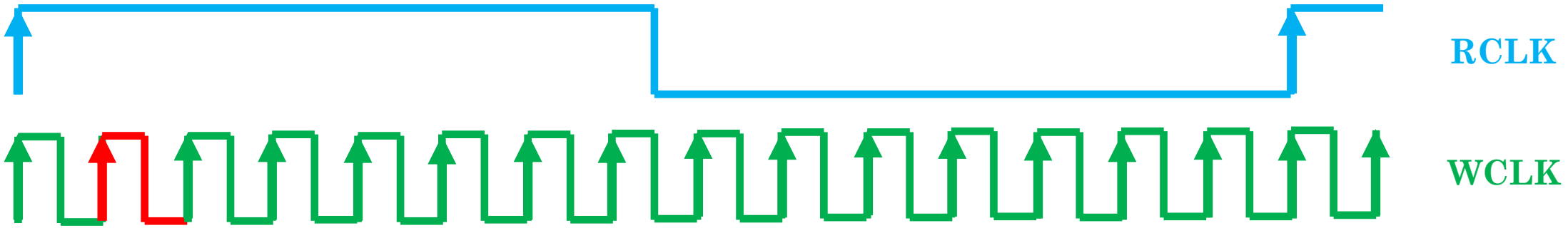
EMPTY (if most recently it was almost empty)

FULL (if most recently it was almost full)

For a 2-clock FIFO, use the (n+1)-bit counter method only!

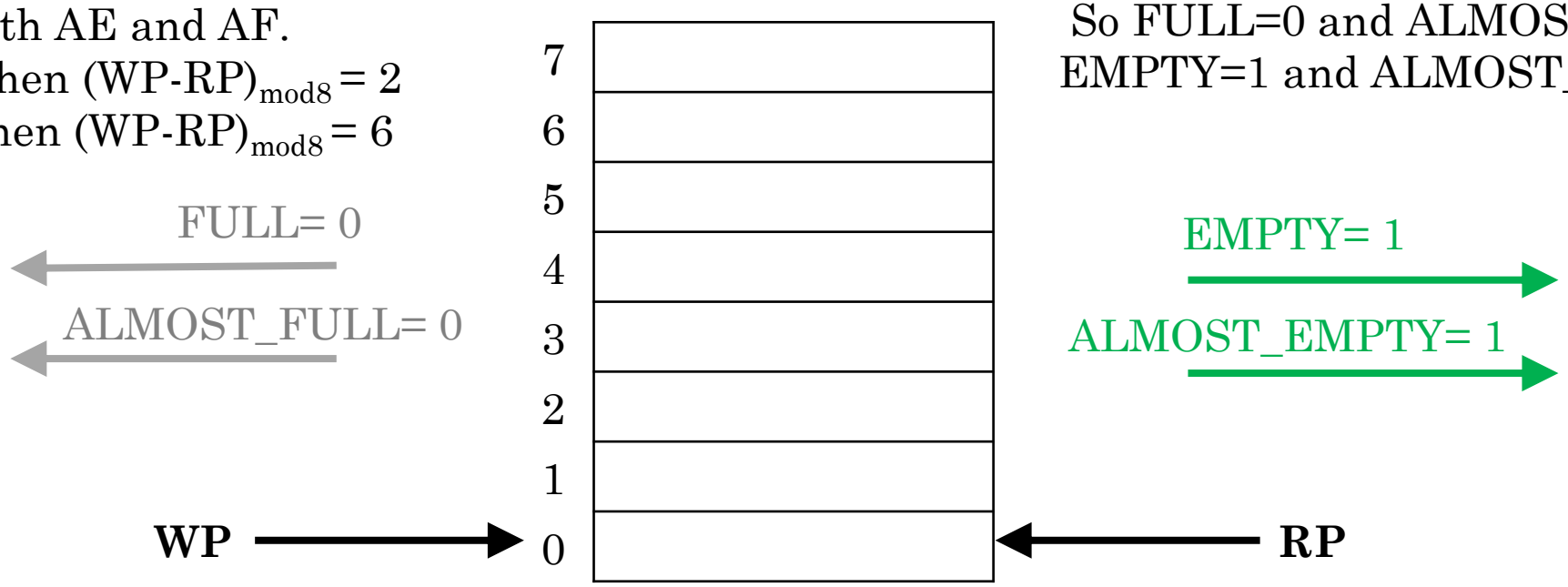
But WHY?! How can it cause a deadlock?

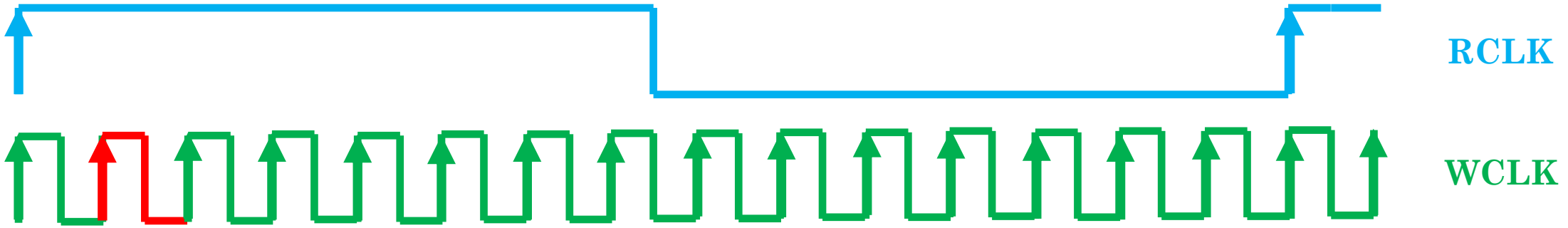
Say $F_{WCLK} >> F_{RCLK}$



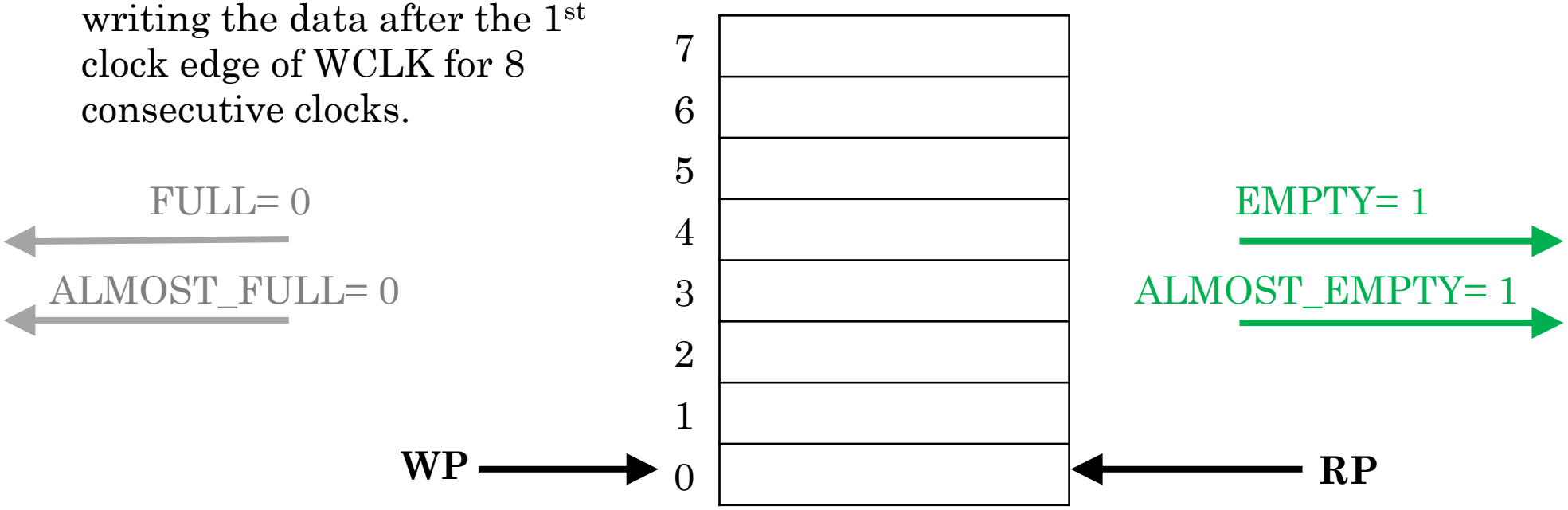
- Consider the junior engineer's design with AE and AF.
- AE==1 when $(WP-RP)_{mod8} = 2$
  AF==1 when $(WP-RP)_{mod8} = 6$

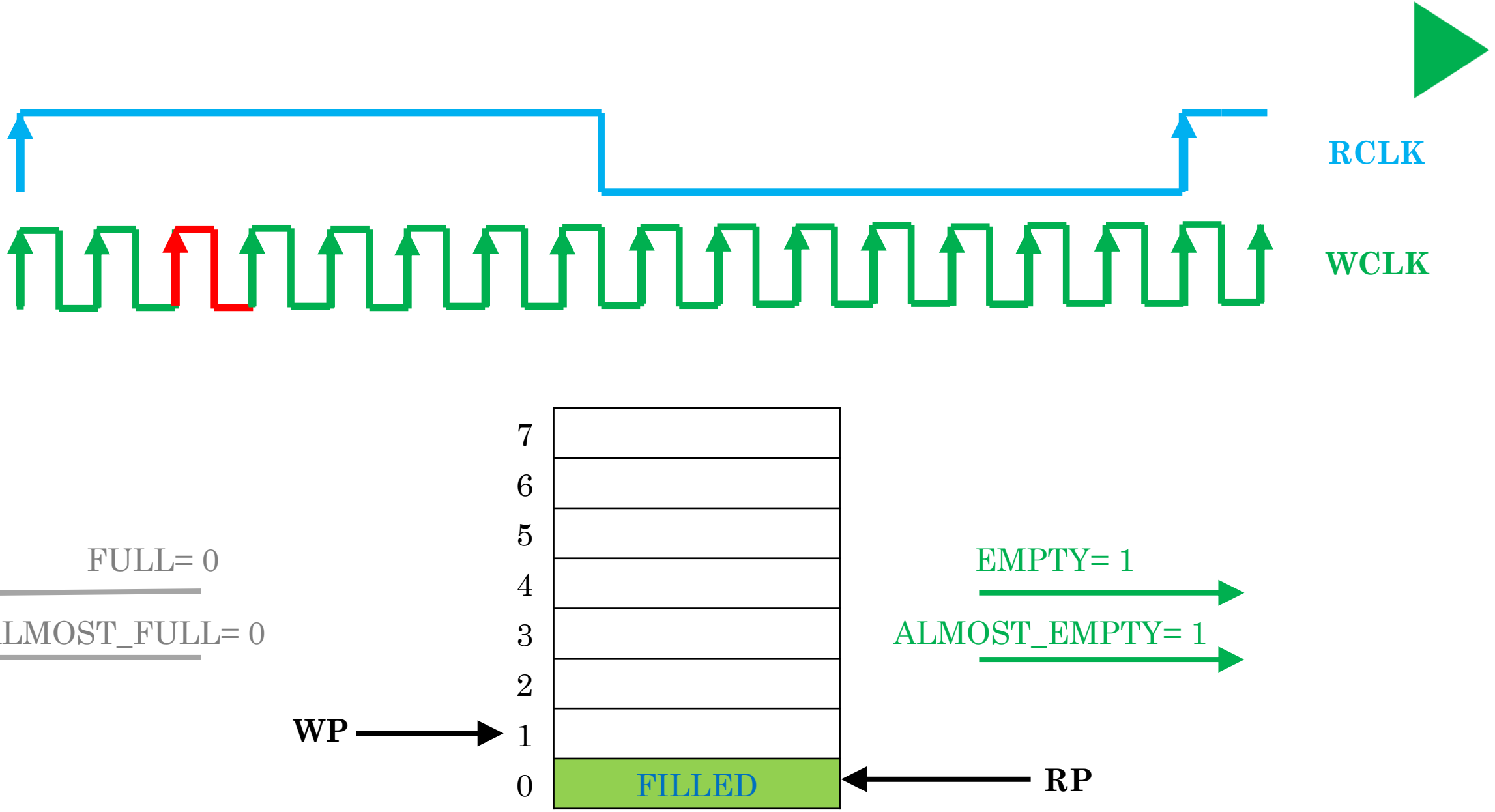
- FIFO is initially empty.
  So FULL=0 and ALMOST_FULL=0;
  EMPTY=1 and ALMOST_EMPTY=1

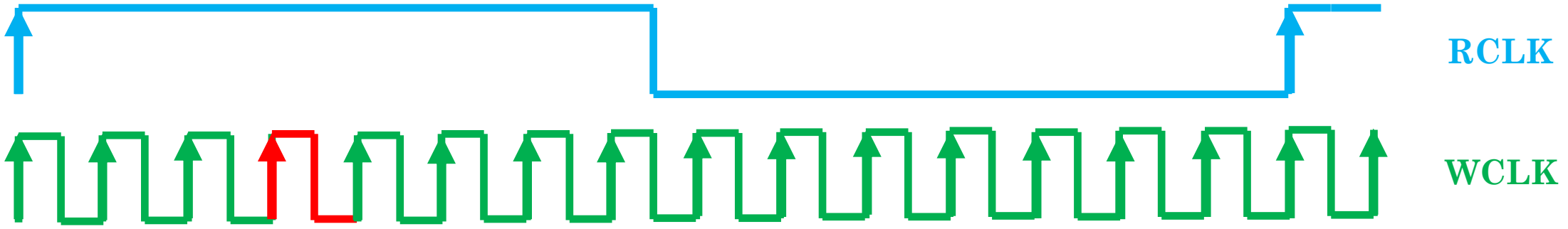


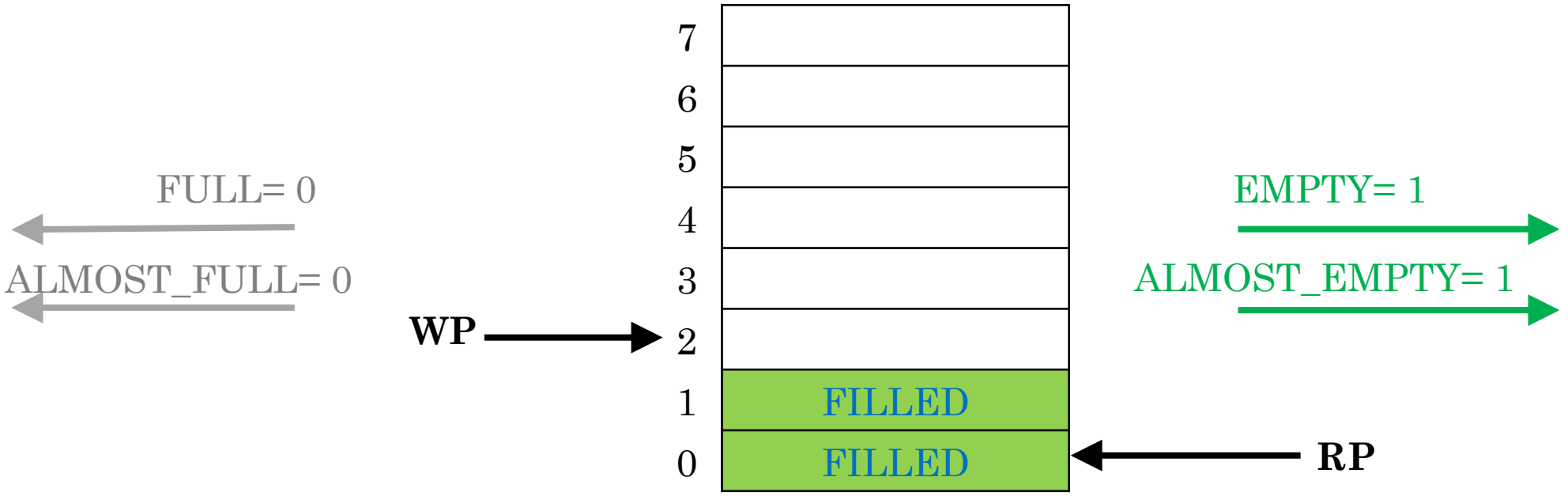


- Suppose the producer starts writing the data after the 1st clock edge of WCLK for 8 consecutive clocks.

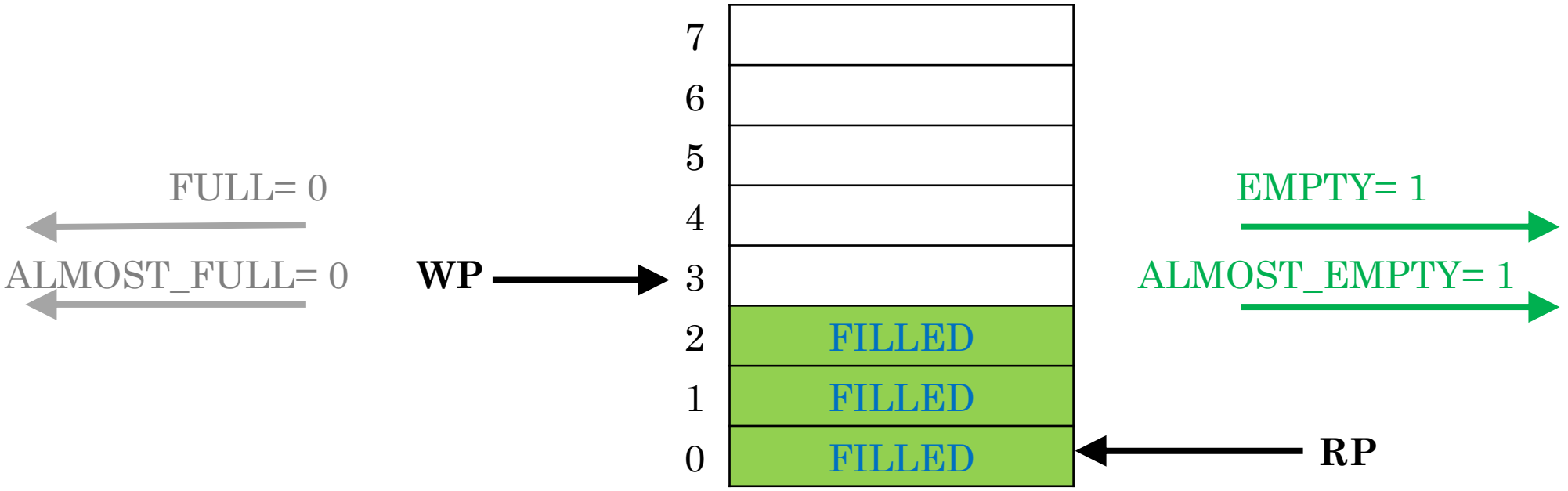


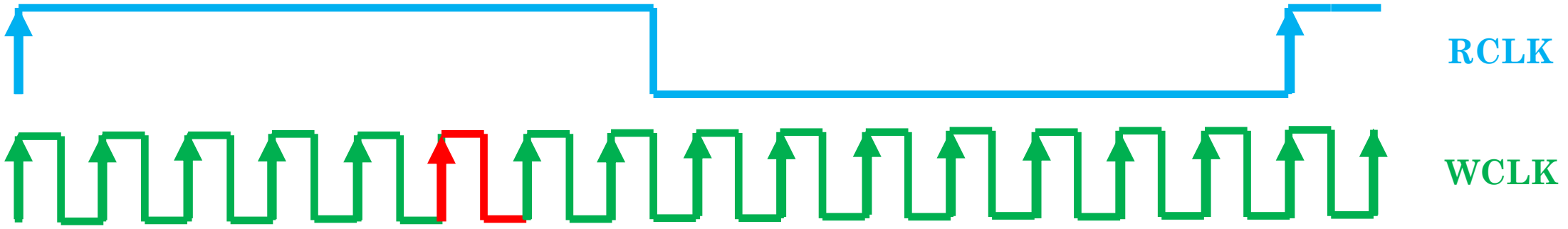

RCLK
WCLK
7
6
5
4
3
2
1
0
FILLED
WP
RP
FULL= 0
ALMOST_FULL= 0
EMPTY= 1
ALMOST_EMPTY= 1

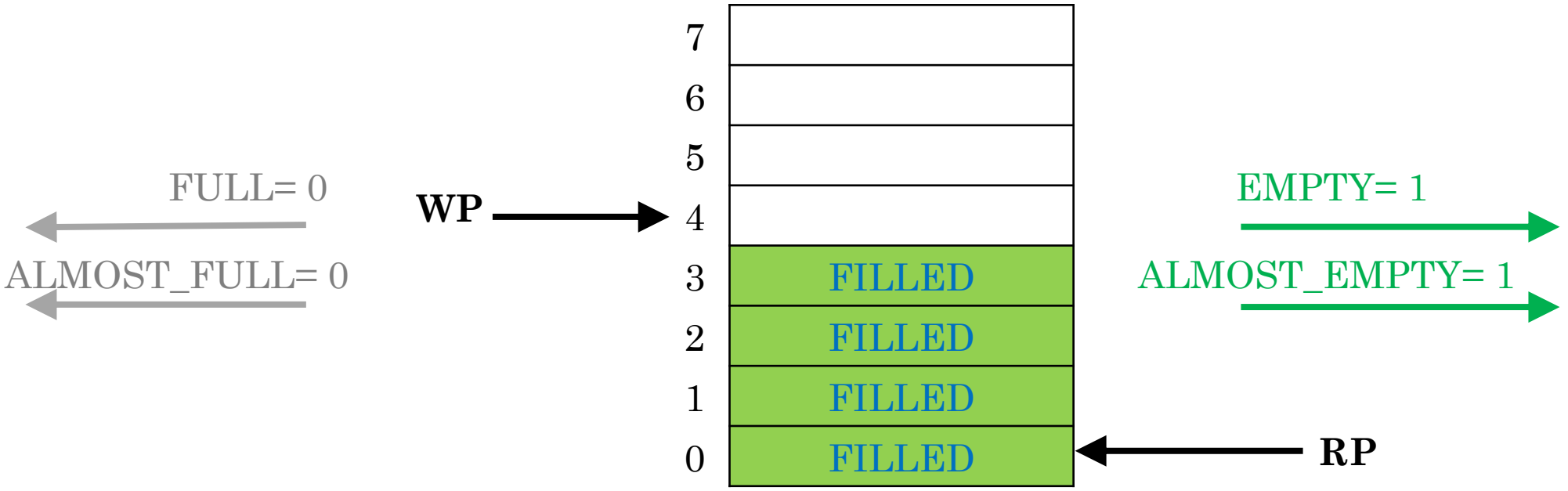

RCLK
WCLK

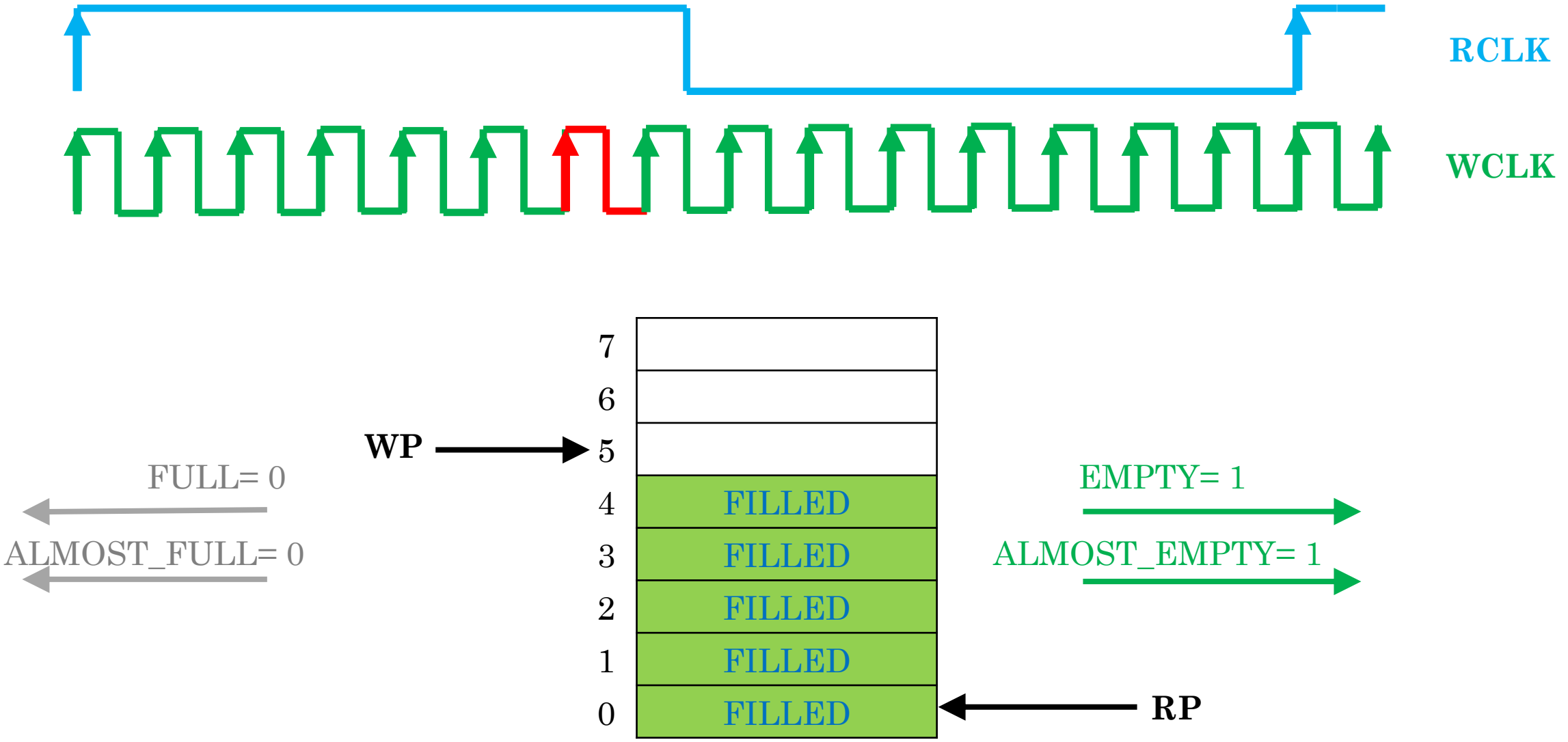

7
6
5
4
3
2
1
0
FILLED
FILLED
FULL= 0
ALMOST_FULL= 0
EMPTY= 1
ALMOST_EMPTY= 1
WP
RP

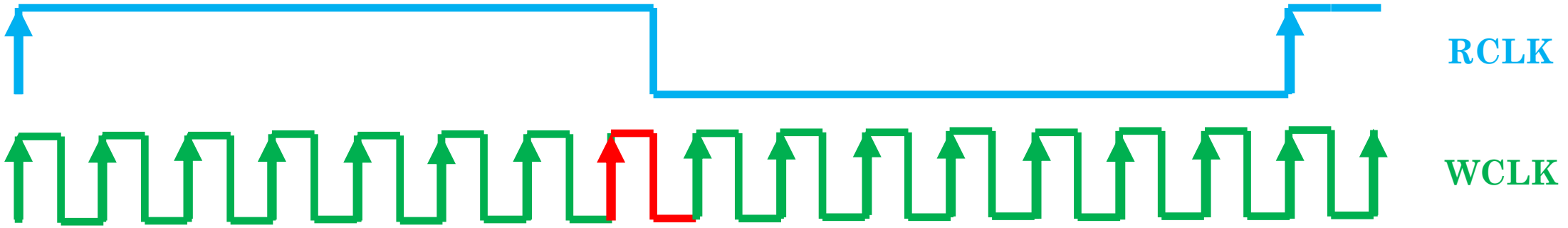

RCLK
WCLK


7
6
5
4
3
2
1
0
FILLED
FILLED
FILLED
FULL= 0
ALMOST_FULL= 0
WP
EMPTY= 1
ALMOST_EMPTY= 1
RP

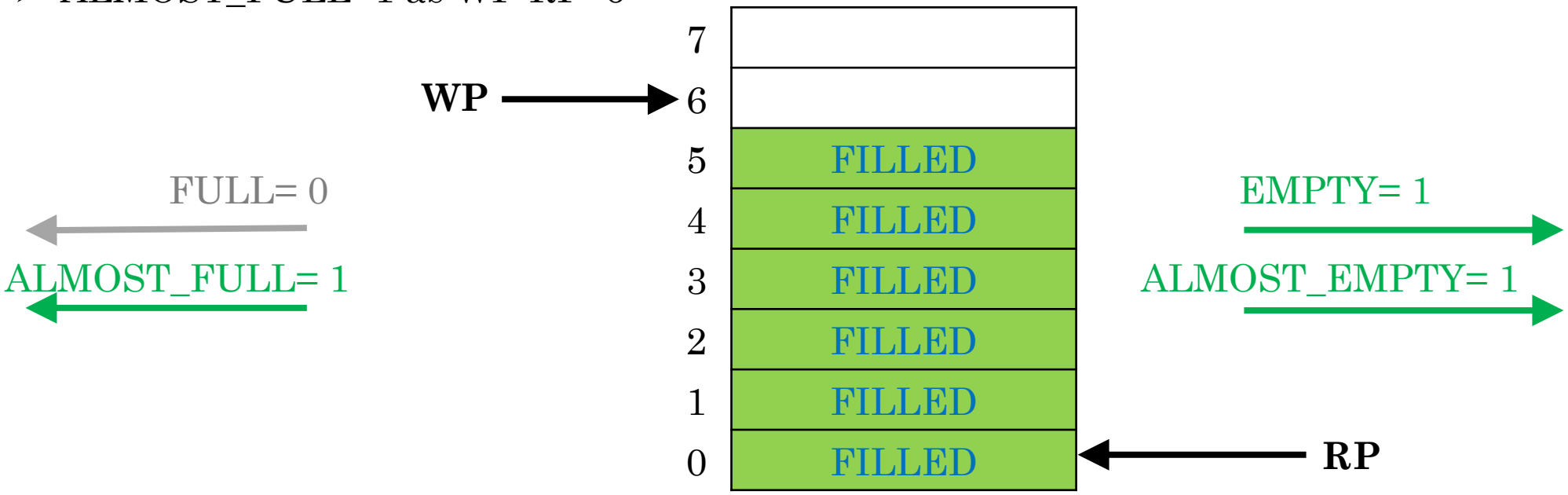

RCLK
WCLK

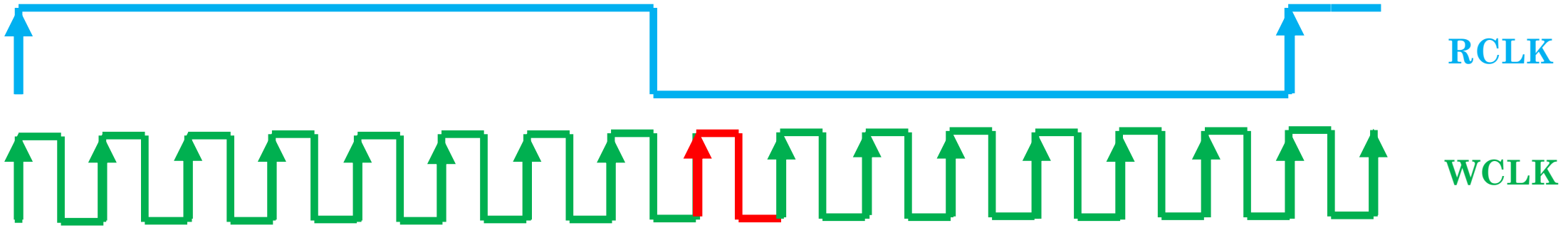

7
6
5
4
3
2
1
0
FILLED
FILLED
FILLED
FILLED
WP
RP
FULL= 0
ALMOST_FULL= 0
EMPTY= 1
ALMOST_EMPTY= 1

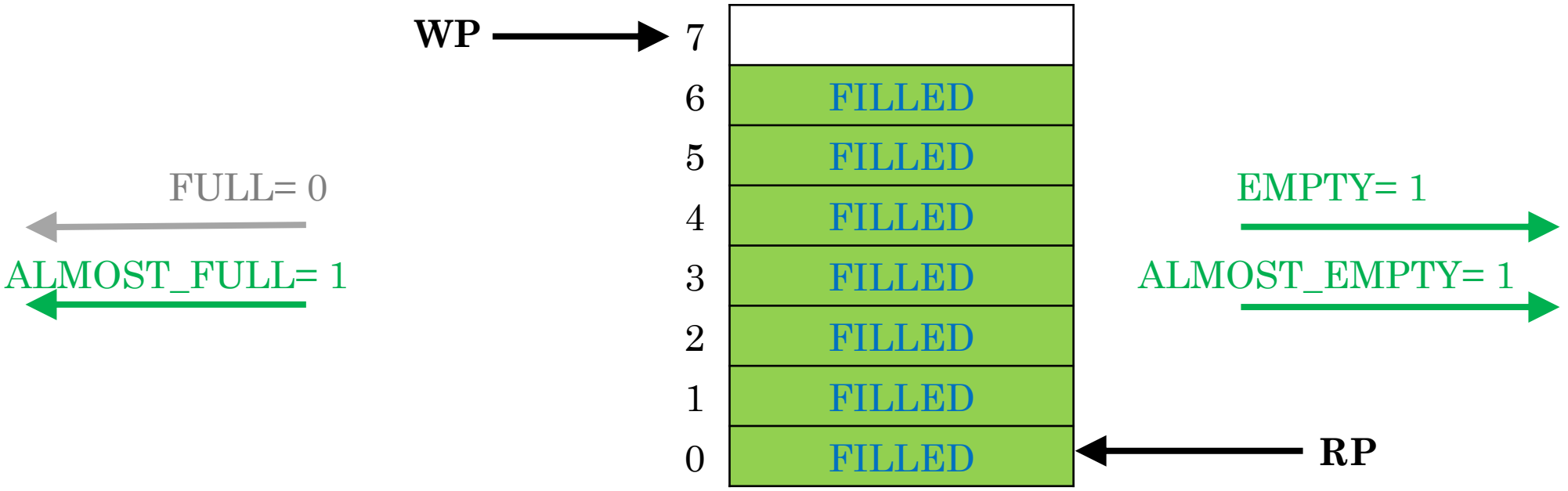

RCLK
WCLK
7
6
WP
5
4
3
2
1
0
FILLED
FILLED
FILLED
FILLED
FILLED
FULL= 0
ALMOST_FULL= 0
EMPTY= 1
ALMOST_EMPTY= 1
RP

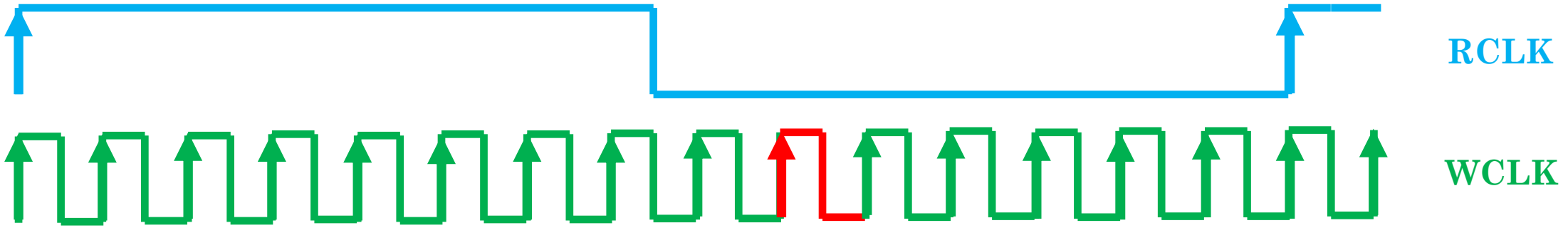


- ALMOST_FULL=1 as WP-RP=6

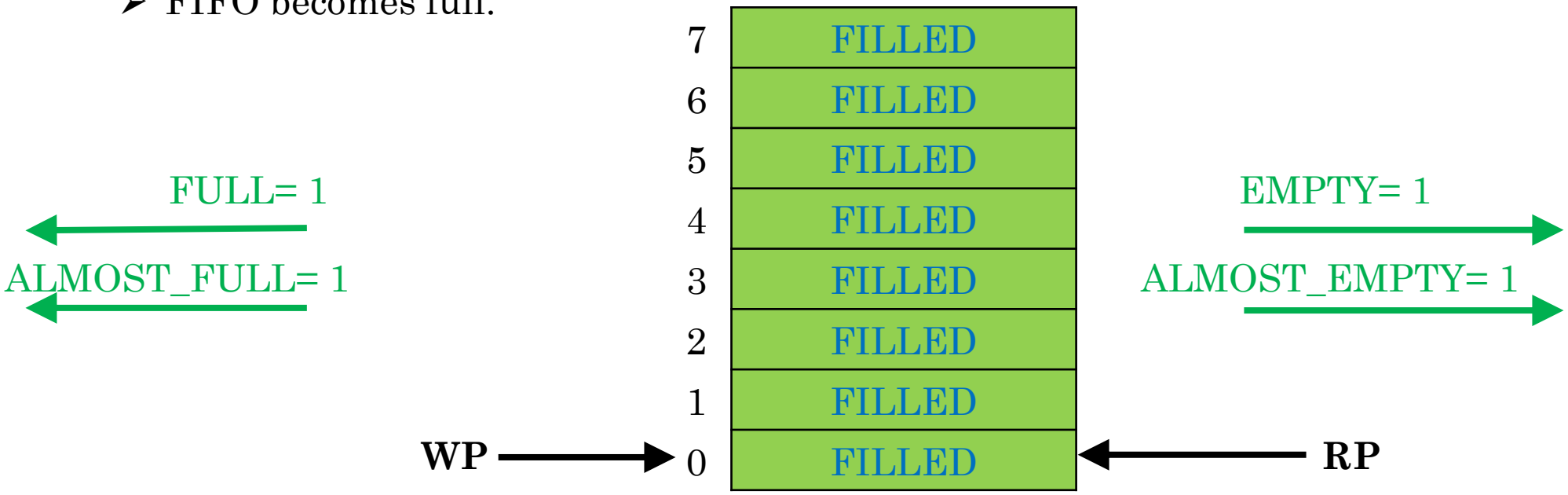



RCLK
WCLK

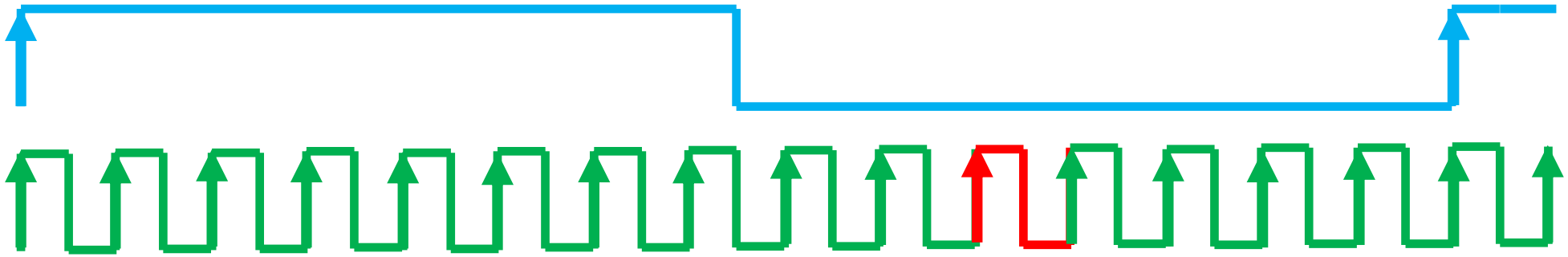

WP
7
6
5
4
3
2
1
0
FILLED
FILLED
FILLED
FILLED
FILLED
FILLED
FILLED
RP
FULL= 0
ALMOST_FULL= 1
EMPTY= 1
ALMOST_EMPTY= 1



- FIFO becomes full.



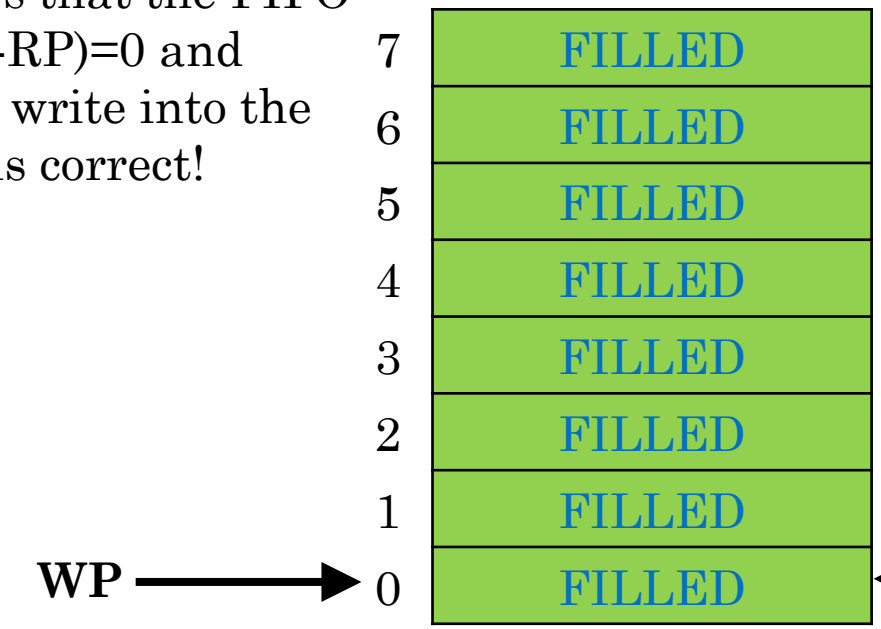
What happens here?!



The producer sees the AF and Full flag asserted. He infers that the FIFO is FULL because (WP-RP)=0 and AF=1. So, he wouldn't write into the FIFO any more. This is correct!

FULL= 1

ALMOST_FULL= 1



The consumer sees the old (WP-RP)=0 and doesn't see the updated WP or updated depth as he samples at a very slow rate. As a result, he continues to believe that the FIFO is empty. And he doesn't read either.

EMPTY= 1

ALMOST_EMPTY= 1

This is a deadlock as both the Full flag and the Empty flag are high together and neither the producer nor the consumer is going to take a step!!

The vice versa holds true as well if $F_{RCLK} >> F_{WCLK}$