Asynchronous FIFO Architectures (Part 3)

Vijay A. Nebhrajani

In the first article of this series we saw the general architecture of a FIFO and analyzed the trivial case with one clock. The second part of the series described one possible architecture for a dual clock design. In this third part we will explore an alternative architecture for a dual clock FIFO; this alternate architecture is not necessarily better — it is just another way of implementation.

1 Operating Principles

By now we have figured out that any calculation involving multi-bit binary quantities from different clock domains requires that they be Gray coded. This architecture is no different; it differs from the previous architecture only in one aspect — that of figuring out the condition that caused the read and write pointers to be equal.

If you remember the previous article, the equality of the read and write pointers implies either the full condition or the empty condition, depending on whether a write or a read (respectively) caused the equality. In the first case of a synchronous FIFO, this was easy to determine, since both operations happened with respect to a common clock. In the second architecture, this information was encoded in the pointers themselves. We now explore a second method for dual clock designs.

1.1 Direction flags

In this architecture we maintain a flag that keeps track of what caused the pointers to become equal. Let us call this flag "`direction_flag`". This flag tells the status circuit the direction in which the FIFO is "currently headed". This is based on the postulation that writes cause the FIFO to be "headed" in the direction of becoming full, and reads cause the FIFO to be headed in the direction of becoming empty.

Needless to say, each side must keep individual copies of the `direction_flag` and maintain it in a pessimistic fashion. Thus the write side would have its own `direction_flag` which would be maintained pessimistically — i.e., it may see delayed reads and the read side would maintain a `direction_flag` that would be computed based on delayed writes. This would, as in the previous dual clock architecture, make sure that the FIFO does not eat or regurgitate data but do this at the expense of dynamically shrinking the FIFO size a little.

The `full` and `empty` flags of the FIFO are calculated based on these `direction_flag`s. The idea is that if the FIFO was headed towards becoming full and the pointers did become equal, it is indeed full. If the FIFO was headed towards becoming empty and the pointers became equal, the FIFO is now empty.

1.2 Implementation of the direction flags

There are different ways in which the direction flags could be implemented: The general idea is that when the word count of the FIFO crosses a certain pre-determined threshold, it is deemed to be "going towards full" and when its word count falls below some other pre-determined threshold, it is deemed to be "going towards empty".

Some designers choose the "going towards full" threshold to be 75% of the FIFO size, and the "going towards empty" threshold to be 25% of the FIFO size. Other designers simply choose 50% for both thresholds. Yet others choose 80% and 20%. The choice of the threshold is really yours and you can choose what fits your needs best. You could derive a relationship between the clock speeds and the threshold values so that flag thrashing is minimized, but I am not sure that this will make your design more robust. I believe that some hysteresis is probably good (hysteresis means that the thresholds differ and that the "going full" threshold is greater than the "going empty" threshold).

Let us arbitrarily choose the thresholds to be 75% and 25% of the FIFO size. This is somewhat efficient since you only need to compare the upper two bits of the pointers to determine threshold crossings. With other threshold values, you would have to compare the full-width pointers, potentially slowing down your design. As before, the write side sees the write pointer and a synchronized version of the read pointer, both in Gray. It then converts both pointers to binary and figures out how much data there is in the FIFO. If the amount of data in the FIFO is greater than the "going full" threshold, it sets its `direction_flag`. When the amount of data falls below the "going empty" threshold, it clears its `direction_flag`.

Similarly, the read side sees the (Gray) read pointer and a synchronized version of the (Gray) write pointer. After performing a conversion to binary it computes the word count in the FIFO; if this word count is less than the "going empty" threshold, it sets its `direction_flag` (which is now in the opposite sense as the `direction_flag` on the write side), and when the word count increases so that it is above the "going full" threshold it clears its `direction_flag`.

Remember that the computations described above do not need to be performed on the full-width pointers if you choose the 75% and 25% thresholds. It is sufficient to merely use the upper two bits of the pointers.

1.3 Computing `full` and `empty`

On the write side, if the pointers become equal and the `direction_flag` was set, the `full` flag of the FIFO is set. Similarly, on the read side, if the `direction_flag` was set and the pointers become equal, the FIFO's `empty` flag is set. Notice that this means that we do not exclude the possibility that the `full` and `empty` flags are set simultaneously. Although it sounds counter-intuitive, this is a valid condition for the FIFO. You may think that it is not possible for a FIFO to be both full and empty at the same time; however if you analyze further, you will realize that "full" is merely a flow control mechanism for the write side and "empty" is a flow control mechanism for the read side. It is quite all right if the FIFO blocks both flows — this does not corrupt the memory or the pointers. It is dangerous if the FIFO does not report that it is full when it really cannot accept more data or if it does not report that it is empty when it really cannot supply more data. A careful analysis of the previous architecture will demonstrate to you that this possibility is not excluded in that architecture either.

Here are the general equations for calculating the direction flags on either side (note that the pointers in the equations are synchronized as applicable and converted to binary):

$$\texttt{word\_count} = \begin{cases} \texttt{wr\_ptr} - \texttt{rd\_ptr} + 1 & \textit{if } \texttt{wr\_ptr} > \texttt{rd\_ptr} \\ \texttt{fifo\_size} - (\texttt{rd\_ptr} - \texttt{wr\_ptr}) + 1 & \textit{if } \texttt{rd\_ptr} > \texttt{wr\_ptr} \end{cases}$$

$$\texttt{direction\_flag}_{wr} = \begin{cases} 1 & \textit{if } \texttt{word\_count} > \texttt{going\_full\_threshold} \\ 0 & \textit{if } \texttt{word count} < \texttt{going\_empty\_threshold} \end{cases}$$

$$\texttt{direction\_flag}_{rd} = \begin{cases} 1 & \textit{if } \texttt{word\_count} < \texttt{going\_empty\_threshold} \\ 0 & \textit{if } \texttt{word\_count} > \texttt{going\_full\_threshold} \end{cases}$$

This is shown in Figure 1, with the special case of 75% and 25% thresholds. In that special case the word count equation above really requires only the two uppermost bits of the binary pointers, and it is not really necessary to add 1 to get a precise word count. You really just need to know if the threshold has been crossed or not.

Also keep in mind that it is not necessary that the thresholds on the write and read sides be the same; you could tune your threshold values based on the frequency of the write and read clocks to optimize performance.

2 Conclusion

This architecture is a variation on the theme presented in the case of a synchronous FIFO. The synchronous FIFO is a special case of this architecture where the thresholds equal `(fifo_size - 1)` and `1` for the "going full" and "going empty" thresholds respectively.

Does this architecture represent specific advantages over the previous asynchronous architecture? In my opinion: Not necessarily. There are borderline cases where this architecture may be an advantage — wherever your timing is so tight that converting N-bit Gray to binary is OK, but N+1 bit Gray (required in the previous architecture) is not. Or when your area is so tight that the extra area for the N+1 bit conversion is not feasible. In my opinion, these are rare cases indeed, and therefore which architecture you choose really depends on your preference.

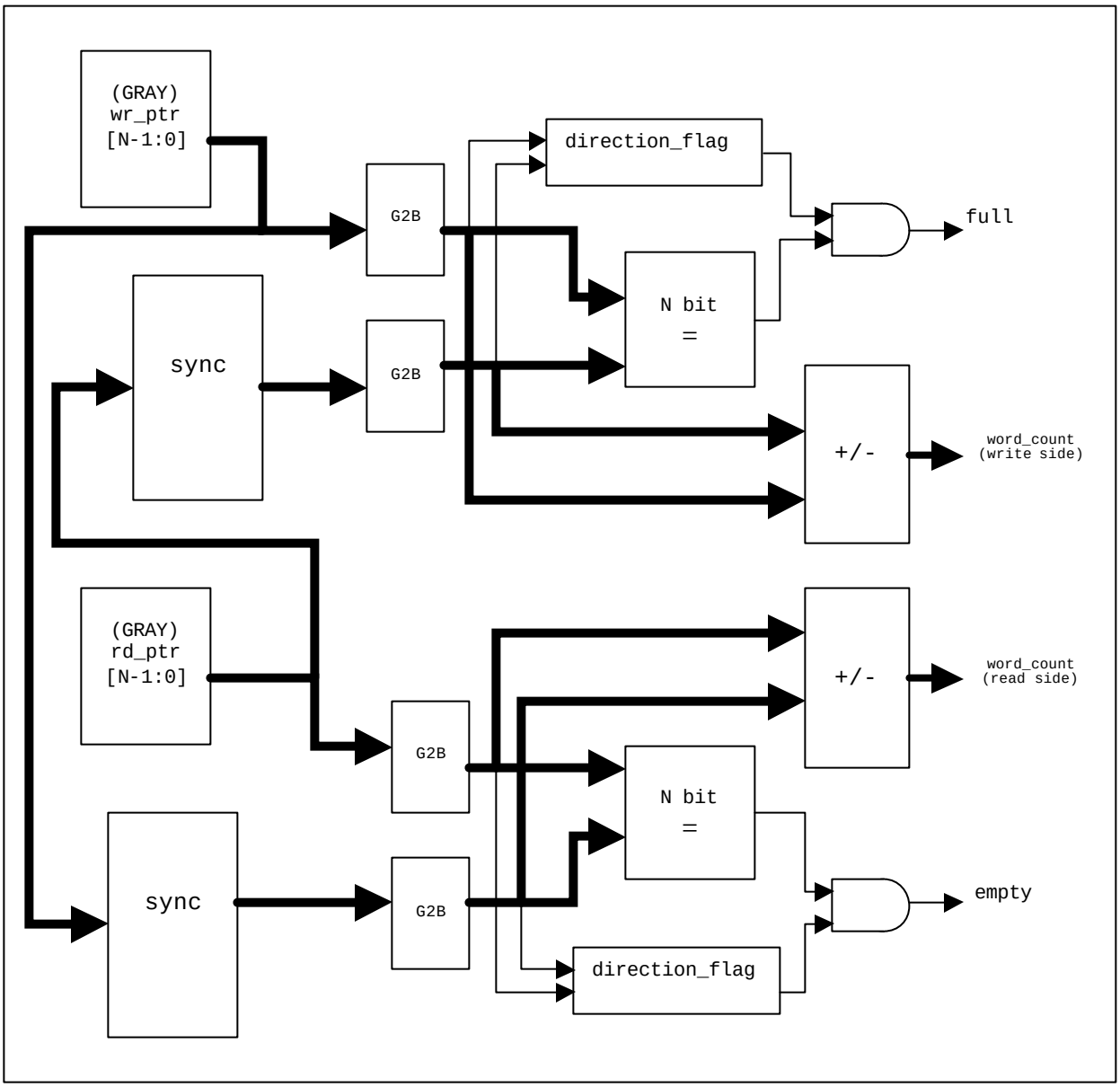

Figure 1 Status block for second dual clock FIFO architecture.