Asynchronous FIFO Architectures
(Part 2)

Vijay A. Nebhrajani

In the previous part of this series we saw how a synchronous FIFO may be designed using a dual port, non-registered output RAM. This part examines how the same concept may be extended to yield a FIFO that has separate, free running read and write clocks. Having free running clocks simplifies some problems, but this leads to a case-specific solution. The most general case is one in which nothing is assumed about the clocks — not even that they are free running. I will present this in the concluding part of this series.

If you refer to the previous article, you will observe that other than the status block, nothing really needs to work off two clocks. The memory does not have registered reads, so it does not really use the read clock; and even if it did have registered reads, these could run on the read clock without problems. The status block fundamentally performs operations on the two pointers, and these run off two different clocks. This is what causes the real difficulty. If you were to sample the read pointer with the write clock (or vice versa), you would potentially run into a problem called *metastability*. This would cause you to calculate the empty and full flags incorrectly, and that would in a word, kill the design.

1 Metastability

So let us proceed systematically and tackle problems as they arise. The first problem is to understand metastability. Metastability is the name for the physical phenomenon that happens when an event tries to sample another event. Mathematically, the case is as follows: Assume that a signal changes instantaneously from `0` to `1` at time $t = 0$. What, then is the value of the signal *at* $t = 0$? Is it `0` or `1` or something in between? In mathematics, this problem is circumvented by defining two more instants, called `0-` and `0+`. At $t = 0-$, the signal is defined to have the value `0`, and at $t = 0+$ it is defined to have the value `1`. Of course, $0- = 0 - 0$ and $0+ = 0 + 0$. Note that this is a mathematical trick only, and if you did the same thing with a physical circuit, the output will either be a logical `0` (0 volts) or a logical `1` (5 volts, nominally) or anything in between (0-5 volts). In physical systems, as in mathematical ones, sampling an event by another event yields unpredictable results. Unpredictability also implies another phenomenon — and this is the real danger that metastability poses.

1.1 Resolution Time

When an event samples a static value, (or a value that has been stable for a while) the sampled value resolves itself to that of that static value. If you talk in terms of a `D` flip-flop,

Q resolves itself to the value of D. The time taken for this resolution, defined with respect to the sampling event is called resolution time. You are familiar with this term as the "clock-to-Q time", or t_{cq}. If you meet the setup time and hold time of the flip-flop, this is the time that the cell designer guarantees the output will be resolved by. Metastability affects the resolution time of the physical system, as well as the resolved value. Think of this in terms of "unstable equilibrium", or the ball on the hill. If you have a ball on a "hill" (another spherical surface, say) the ball is considered to be in a state of unstable equilibrium. If completely undisturbed, it may rest on the hill forever, but the tiniest perturbation will cause the ball to roll to one side of the hill or another. There is no way of calculating how long the ball will rest on the hill, or which side the ball will fall. This is the exact case with metastability — you cannot predict which value the physical system output will resolve to, and more dangerously, after how long. Put another way, there is a finite, nonzero probability that the output will remain metastable forever. In practice though this is rarely the case, and a figure of 20 times clock-to-q is taken as a reasonable figure for resolution. In theory, resolution time follows an asymptotic curve to infinity as the sampling instant comes closer to the event being sampled.

1.2 MTBF and Reliability

If you have an asynchronous element in your design, you will run into metastability whether or not you want to. There is absolutely no way of eliminating metastability completely, so what we do is calculate a "probability" of error and express this in terms of time. Let us say that there is a probability of a metastable failure of a physical system and that this is one in 1000. In other words, for every 1000 samples, one will cause a failure because of metastability. This also means that once every thousand times, the output does not resolve itself by the time the next clock edge comes along. If your clock frequency is 1 kHz, this implies a failure rate of once every second, and so the Mean Time Between Failures is calculated to be 1 second. This is of course, oversimplified; MTBF is a statistical measure of failure probability, and requires some more complex, empirical and experimental data to arrive at. For flip-flops, this relationship depends on a physical constant of the circuit itself, and on the clock frequency. It is important to remember that metastability itself has nothing to do with clock frequency — but MTBF does. We define a circuit with higher MTBF to be a more reliable circuit, naturally.

1.3 Synchronization

Since metastability is absolutely unavoidable, you must design the circuit so that it —

- Handles errors gracefully.
- The probability of occurrence of such errors is minimized.

The first requirement varies greatly from design to design, and is not in the scope of this brief introduction. For the second requirement, a technique called "synchronization" is used. This technique simply consists of two flip-flops as shown in Figure 1. Q2 will go metastable only when Q1 changes too close to the clock. If we take a 20x figure for t_{cq} as the time for resolution in the event of metastability, then the clock period would be given by

$t_{clk} = 20t_{cq} + t_{setup}$. What this really means that the probability that the output is still not resolved after 20 times the clock-to-q time is rather small. Thus, the probability of `Q2` not being resolved by the time a clock edge comes along is approximately p^2 where p is the probability that the first stage output does not get resolved by the time a clock edge comes along. This is called dual stage synchronization. When you use this probability with the clock frequency to calculate MTBF, you will find that the MTBF has increased greatly. If you want, you may increase the MTBF further by having a three-stage synchronizer, but this is rarely needed in practice.

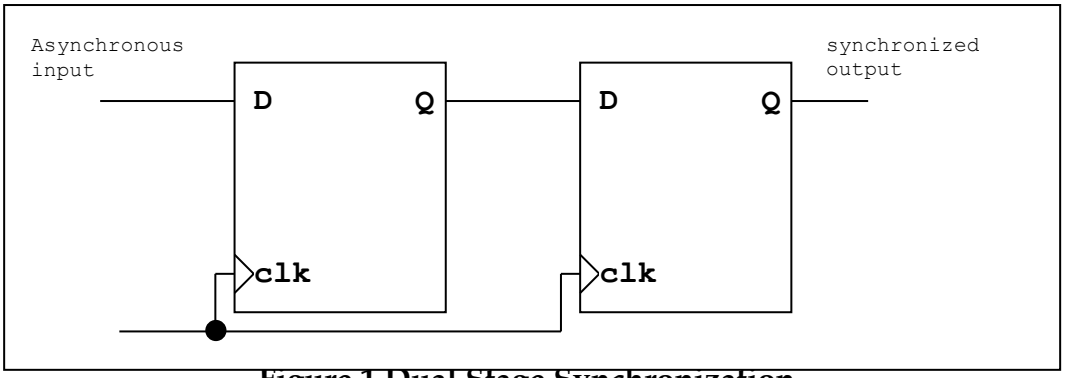

Figure 1 Dual Stage Synchronization

You can make the circuit more resistant to metastability by having redundant synchronizers as shown in Figure 2. The final output is calculated as the majority (2 out of 3) for triple redundancy and equality for double redundancy. In this case small routing and device differences imply that if one of the synchronizers has a metastable failure, the other two will, in all probability have resolved themselves. Once again, this technique is needed in only the most critical requirements.

For more on metastability and synchronization, see this:
Grosse, Debora; "Keep metastability from killing your digital design"
`http://www.ednmag.com/reg/1994/062394/13df2.htm`

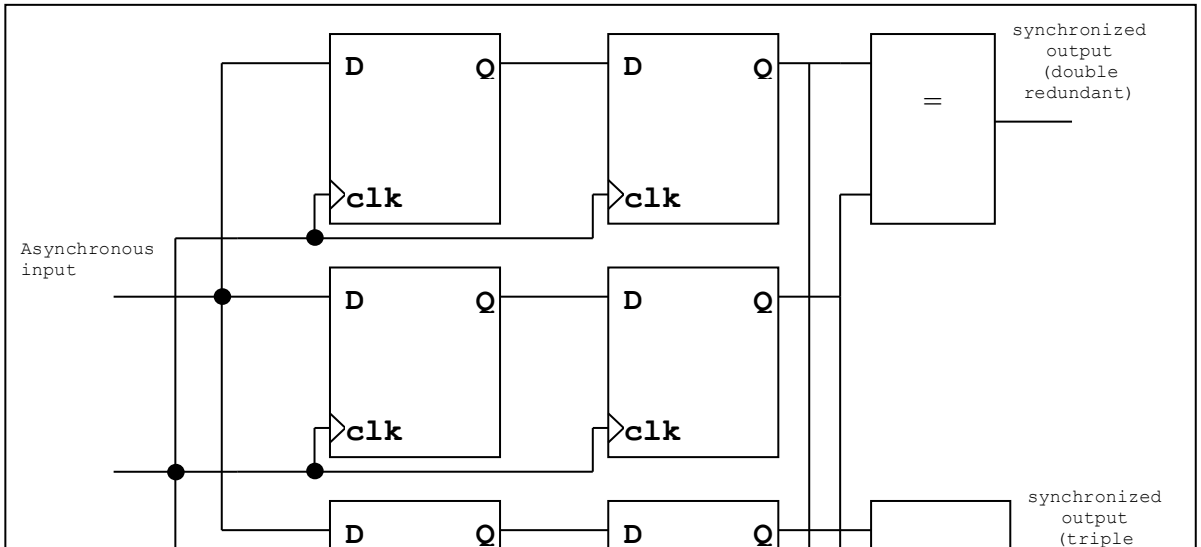

Figure 2 Double and Triple redundant 2-stage synchronization

2 Sampling counters

2.1 Synchronization: Solving the reliability problem

Now back to the FIFO problem. You need to sample the value of a counter with a clock that is asynchronous to the counter clock. Thus you could land up in a situation where the counter is changing from, say `FFFF` to `0000`, and every single bit goes metastable. This in turn means that you could potentially read any value between `0000` and `FFFF` (both included). Of course, this means that your FIFO will not work. Synchronization will save your counter samples from going metastable, but you still may get sampled values that are wildly off the mark. In other words, it is not sufficient to merely synchronize the counters.

The important thing that we must do is to make sure that not all bits of the counter can change simultaneously. In fact we have to make sure that precisely one bit changes every time the counter increments. This implies that if you catch the counter transitioning, only one bit may be in error. This is the best we can do, since we need at least one bit transition if the counter itself is to work. What we need therefore is a counter that counts in the Gray code. This is because the Gray code is a unit distance code; that is, every next value differs from the previous in only one bit position.

Let us now examine how this helps us. First, synchronization means that we will rarely have the sampled value of the counter go metastable, and second, the value that we do sample will at most have one bit error. This means that if the counter's actual value changed from $N - 1$ to N, you will read either $N - 1$ or N, but no other value. This is absolutely correct behavior for reading a counter, since at the time of change you need to make a decision about the value. As long as you decide that the value is the old value or the changed value, you are OK. Any other value is not OK. If you think about this further, you will realize that if you sample the counter at the instant it changes, either answer ($N - 1$ or N) is correct for the value of the counter.

2.2 Pessimistic Reporting: Handling errors gracefully

Knowing this much, let us now analyze how we can apply this to the read and write pointers of our FIFO. Let us say that we wish to know whether or not the FIFO is full. If it

is full, we must block further write accesses. This is critical because we must stop the write pointer from changing when the FIFO is full. We synchronize the (Gray coded) read pointer to the write clock. This means that we may have a stale value of the read pointer, since the actual read pointer may have changed to a different value while we were synchronizing it. If this is so, then the write side thinks that ***less*** reads have been performed (than actually have), and if conditions match, that the FIFO is full. In truth, the FIFO may not be full because reads may have taken place that the write side did not "see". However, we merely block additional writes and this is OK. It would be incorrect if we did not block writes when the FIFO was actually full.

Similarly for the read side — the read side sees "delayed" writes, and may decide that the FIFO is empty when it actually has some data. The effect of this is that reading will be blocked till the writes "become visible" to the read side. In the meanwhile, it will not allow further reads.

This is called pessimistic reporting. In short, reporting to the write side that the FIFO is full when it is not is fine, and so is reporting to the read side that the FIFO is empty when it is not. This acts as if the FIFO had dynamically shrunk a little, and is quite harmless.

In case of the word count, we use the same technique, providing a write-side word count and a read-side word count. The write-side word count is likely to be greater than the actual word count in the FIFO, and this is quite alright because the only effect it is allowed to have is to block further writes. Similarly the read side word count may be lesser than the actual word count, and that too is OK. Just make sure that you do not use the write-side word count on the read side and vice versa.

This mechanism of pessimistic reporting takes care of gracefully handling errors in the synchronized value. In fact even if the sampled read pointer value were to remain metastable for a while, the effect would be to block writes, causing the FIFO to "hang" for that period for writes, but not causing data errors. The same applies to reads.

3 Architecture 1

3.1 Creating the Empty and Full conditions

Remember from the last article that the pointers are not the only things that affect the empty and full flags. The empty condition is when a read caused the pointers to be equal and full is when a write caused the pointers to be equal. In other words, to generate the full and empty correctly, we need to sample the read and write signals themselves with respect to the other clock. This is a different ball game, since we do not wish to make assumptions about clock frequencies. Imagine sampling a 10 ns write signal (at 100 MHz) with a 1 kHz read clock. You can't do that without pulse stretching, and that implies that you know (or have assumed) a relationship between the clocks.

Of course, we do not want to assume any relationship between the clocks. This poses a problem around which are three ways, and this gives rise to the three different architectures that we will examine. The first architecture is described here and is most

elegant. The second is workable, but not very elegant and the third is the most robust but expensive in terms of area. Which architecture you choose is up to your requirement.

3.2 The First Solution

Since it is not possible to design one circuit that will satisfy pulse sampling regardless of frequency, we bypass the problem by encoding the read or write information in the pointers themselves. We keep a pointer width of $N + 1$ for a FIFO that has a depth of 2^N words. We also convert the Gray pointers to binary so that comparisons are easier.

The FIFO is deemed full when the most significant bits of the binary versions of the pointers (synchronized to the clock in question) differ and the remaining N bits are equal. The FIFO is deemed empty when the (binary converted) pointers are exactly equal. This may not be immediately obvious, so let us analyze it with an example.

Consider how this works for a 8 deep FIFO (we use the binary converted pointers). Initially both `rd_ptr_bin` and `wr_ptr_bin` are `"0000"`. Lets say that you write 8 words to the FIFO. This will mean that `wr_ptr_bin` is `"1000"` and `rd_ptr_bin` is `"0000"`. This is, of course the full condition. Now assume that you perform eight reads, causing `rd_ptr_bin` to be `"1000"`. This is the empty condition. Another 8 writes will make `wr_ptr_bin` equal to `"0000"`. But `rd_ptr_bin` is still `"1000"`, so this is the full condition, and so on.

It should be easy to see that the starting point need not be at `"0000"`. If it is at, say `"0100"` and the FIFO is empty, then eight writes will cause `wr_ptr_bin` to be `"1100"`, and `rd_ptr_bin` to remain at `"0100"`. This again implies full.

That is all there is to it! Did I mention that the first solution was most elegant? And you can use this technique with the synchronous FIFO as well. That would avoid the arithmetic operations and speed up the FIFO.

3.3 Implementation

We know that we need a gray counter. Not a binary counter whose value is converted to Gray (this will defeat the purpose of having one bit change per counter transition), but a true Gray counter. If you have tried implementing this, you would have found that it is not half as easy as it may seem. Of course, you can create a customized machine that does the job, but let me present a general solution to the problem. We know that we can convert from Gray to binary and binary to Gray using the simple equations:

$$g_n = b_n$$
$$g_i = b_i \oplus b_{i+1} \; \forall \; i \neq n$$

and

$$b_n = g_n$$

$$b_i = g_i \oplus b_{i+1} \; \forall \;\; i \neq n$$

In the equations above, the subscript refers to the bit number in an *n+1* bit binary or Gray value.

Also knowing that a counter is nothing more than a set of flip-flops and an incrementer, we do the following — convert the Gray value to binary, increment it, convert it back to Gray and store it. This is the general solution to the thorny problem of generalized n-bit Gray arithmetic (which has a complex, non-trivial solution). This generalized counter is shown in Figure 3.

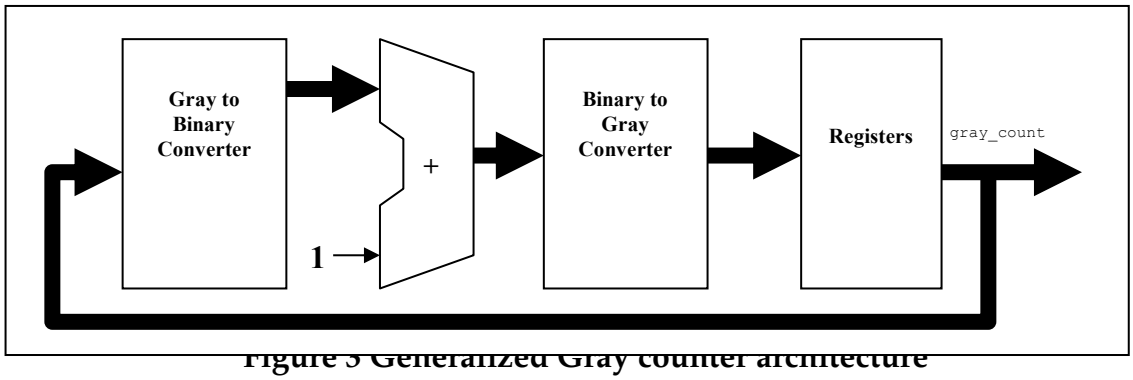

Figure 3 Generalized Gray counter architecture

When you optimize with a synthesis tool, I believe that the tool will provide you with a reasonably fast circuit for the Gray counter. Of course if you wanted a FIFO that was say 32 deep, you could hand code the counter in the form of a state machine with the states encoded in Gray. The states would then themselves be the count values.

The final FIFO design is shown in Figure 4. I am not providing code this time because I believe that it is simple enough to code in either VHDL or Verilog.

Let me note one thing here: If you observe Figure 4, you will notice that there are four Gray to binary converters, and that is not exactly not wasteful. It is possible to avoid these converters by keeping the lower N-bits of the pointers Gray, and the most significant bit binary. This would be a "hybrid" counter, and I leave that as an exercise to the reader.

3.4 Timing considerations

The primary timing condition that governs the operation of the FIFO is the maximum frequency of the clocks. In the case of a FIFO like this, you will have to meet several

parameters – the clock frequencies cannot be greater than what is required for the memory, and the metastable timing relationship $t_{clk} = 20t_{cq} + t_{setup}$ must be met. Of course, the factor of 20 in the equation is empirical, and you may choose any other, provided you have done your homework about MTBF. Other than that, you will have to worry about how fast your Gray counters will run, since one equation above requires chained `XOR` gates. Since we are not doing anything fancy here (except synchronization, and that is not really *fancy*) timing does not pose much of a problem.

The important consideration in this design is the avoidance of metastability related failure using Gray counters and synchronizing them. Do realize that if you have failure of the synchronizers, the entire FIFO could be corrupted (two bits in error could mean a completely different address to the DPRAM, since the addresses are also in Gray) – that is, it may eat or regurgitate data. Therefore I cannot lay enough emphasis on the fact that in your MTBF calculations you should be as pessimistic as you can afford to be.

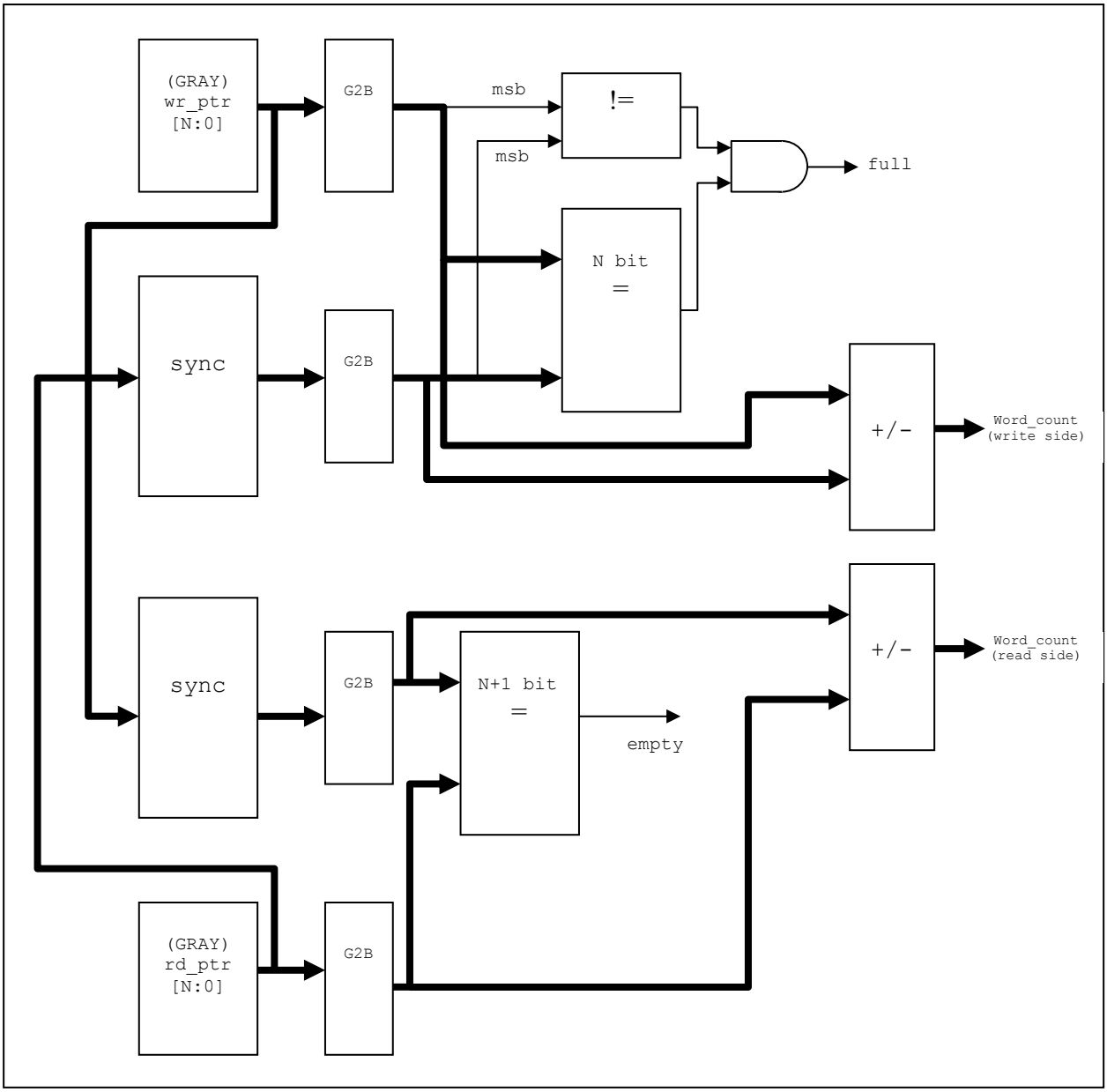

Figure 4 Status Block for Architecture 1