Asynchronous FIFO
using Gray Code Counters

Dhruva S H (213079022)

Introduction

First In First Out Buffers (FIFOs) are commonly used in electronic circuits for buffering and flow control between different hardware components or between hardware and software.
A FIFO primarily consists of a set of read and write pointers, storage and control logic. A dual-port SRAM is usually used, where one port is dedicated to writing and the other to reading.

There are 2 types of FIFOs :

1. Synchronous FIFO
2. Asynchronous FIFO

Synchronous FIFO is a buffer where the read and write ports are operating at the same clock frequency. Such a FIFO is typically used to match the data rate between two blocks of hardware.

Asynchronous FIFO is a buffer where the read and write ports are operating at different (unrelated) clock frequencies. **Clock Domain Crossing** is the terminology used to describe the situation where data has to be written using one clock and read using another clock i.e data is being passed from one clock domain to another.

Design of Synchronous FIFO is much easier when compared to design of Asynchronous FIFO for obvious reasons. When crossing between two unrelated clock domains, the issue of meta-stability becomes a major concern. This document addresses design of Asynchronous FIFO using dual flip-flop synchronizer and gray code counter (elaborated in detail later).

Meta-stability

Meta-stability is a condition that can occur in digital circuits when setup or hold time violations occur. A signal is said to be in meta-stable state if it is residing at a logic level that can neither be interpreted as a 0 (low) nor a 1 (high). Such signals, if propagated, can cause catastrophe in the circuit.

When working with a single clock, timing checks are implemented to ensure that setup and hold time constraints are met and hence it can be guaranteed that meta-stability will never occur. However, when switching between clock domains, it is not that simple. Assuming the two clock domains are unrelated, a signal that passes all timing checks in one clock domain can violate setup or hold time constraints of the other clock. This will inevitably cause meta-stable signals to appear, hence creating issues.

To safely pass a signal from one clock domain to another, synchronizers must be used. A synchronizer is just a flip-flop running on the required clock.

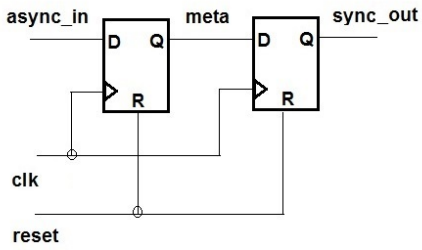

Figure 1: Synchronizer for CDC

This sort of synchronizer works only for single bit data change. If the data has multiple bits and all are potentially changing, then the synchronizer doesn't work because different bits can get resolved in different clock cycles.

In Asynchronous FIFO design, the read and write pointers are running on different clocks and hence they need to be synchronized (i.e read pointer must be synchronized with write clock and write pointer must be synchronized with read clock). Gray-code counters are used for synchronization because it ensures that when pointers are being incremented, only one bit will changed and hence the mentioned technique will work.

Asynchronous FIFO Design

Problem specifications :

1. Write clock frequency is $50MHz$
2. Read clock frequency is $10MHz$
3. Both clocks have 50% duty cycle
4. Data width is 16-bits
5. Maximum burst size is 50

It is assumed that data will be written once at every write clock edge and read once at every read clock edge.

Calculation of FIFO Depth:
Time required to write one data item = $1/50MHz = 20ns$
Time required to write all the data in the burst = $50 \times 20ns = 1000ns$.
Time required to read one data item = $1/10MHz = 100ns$
For every $100ns$, one data from the burst will be read. Hence, in a duration of $1000ns$, 50 data items can be written into the FIFO and $1000/100 = 10$ data items can be read from the FIFO.
The remaining number of data items must be stored in the FIFO, which is $50 - 10 = 40$. Therefore, the FIFO must be capable of storing 40 data items, which is the **minimum depth**.
Since gray-code counter technique is used for ease of synchronization, the depth must be a power of 2. The best option is to choose depth = 64.

The Asynchronous FIFO consists of the following building blocks:

1. A memory of depth 64 and data width 16-bits.
2. A dual-flop synchronizer to synchronize read address with write clock.
3. A dual-flop synchronizer to synchronize write address with read clock.
4. Read pointer management unit which generates empty condition.
5. Write pointer management unit which generates full condition.

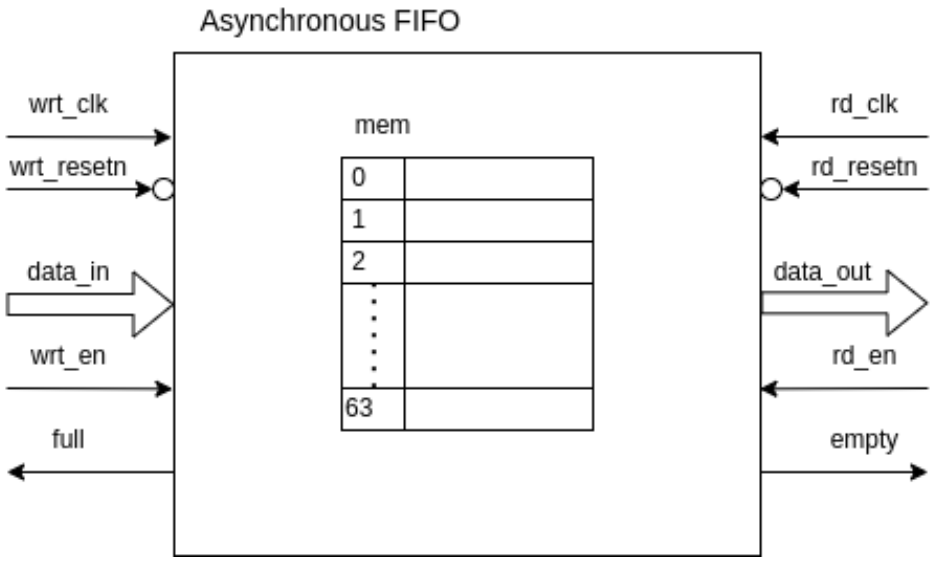

Figure 2: Asynchronous FIFO Module

In the above module, `data_in` is the data to be written into the FIFO, which comes in bursts of at most 50 at a time, at the rate of `wrt_clk` signal. The signal `wrt_en` indicates when the data coming in has to be written into the FIFO, hence it is used to increment the write pointer address.
Similarly, `data_out` is the data to be read from the FIFO at the rate of `rd_clk` signal. The signal `rd_en` indicates when the data stored in the FIFO has to be read, hence it is used to increment the read pointer address.

Empty and Full Conditions:
Since the depth of the FIFO is chosen to be 64, it means that 6 bits are required for addressing the memory.
However, if only 6 bits are used, then it will be impossible to distinguish between full and empty conditions. This is because the condition of FIFO to be full and empty are the same i.e read and write pointers should be equal.

That is why an extra bit is used as an indicator. If the extra bit is different in read and write pointers, it means the write address has wrapped around and caught up to the read pointer i.e the FIFO is full. If the extra bit is same for both, then obvious the FIFO is empty.

This is illustrated in the following figure.

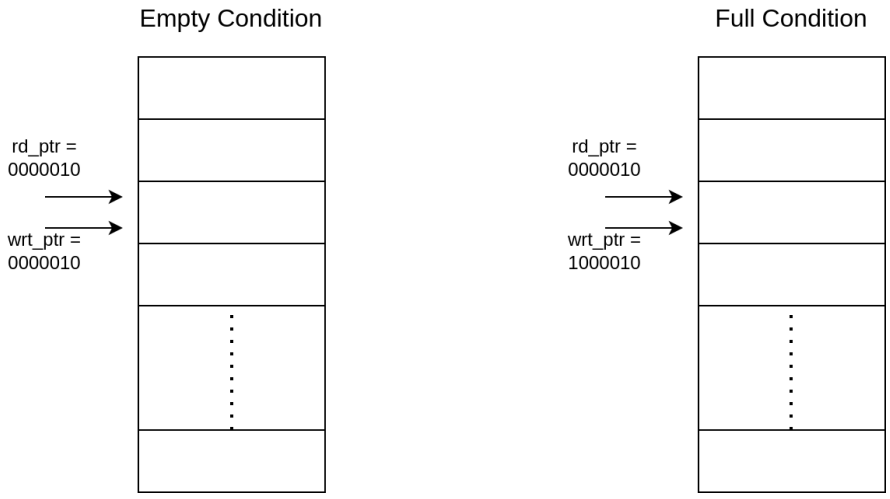

Figure 3: Empty and Full Condition Illustration

General expressions for empty and full conditions, if binary coded pointers were used are as follows.

* Empty condition :

```
rd_ptr[6:0] == wrt_ptr[6:0]
```

* Full condition :

```
(rd_ptr[5:0] == wrt_ptr[5:0]) && (rd_ptr[6] != wrt_ptr[6])
```

Binary code itself is used for addressing the memory but gray-code counter is used for synchronization with the opposite clock domain, hence generation of full and empty conditions must be done using gray-code counter.

Note that in gray counter, wrapping around check is different from that in binary counter. Pattern identification is fairly simple. The two MSBs of the read pointer must be complement to the two MSBs of the write pointer, while the rest of the bits must be same.

Gray-Code Counter Design:
As mentioned earlier, both binary code counter and gray-code counter are needed (for addressing and synchronization respectively). Hence, this design used 2 registers, an incrementer and a binary to gray code converter.

Width of both binary and gray code registers is 7.

- Binary register is incremented every time enable signal comes and the FIFO is neither full nor empty. 6 LSB bits of this binary counter are directly used for addressing the FIFO.

- Gray register is updated with the converted value of the incremented binary code value. 7 LSB bits of this gray counter are used for synchronization with the opposite clock domain and for generating full and empty conditions.

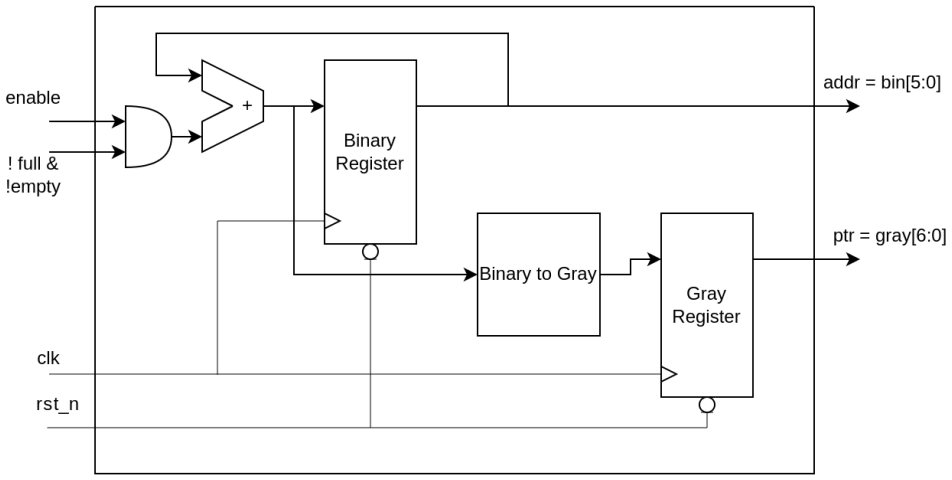

Figure 4: Binary and Gray pointer circuit

Note that the same structure is used for both read pointer and write pointer. Hence, 4 pointer registers are used in the entire FIFO design.

The `rd_ptr` and `wrt_ptr` (i.e gray pointers) are passed through dual-flop synchronizers that are clocked with `wrt_clk` and `rd_clk` respectively (opposite clock domains).

Generating Empty Condition:
The FIFO is empty when the synchronized read pointer (with respect to write clock) is equal to the write pointer. Note that gray code pointers are used here.

$$\texttt{empty} = (\texttt{rd\_ptr[6:0]} == \texttt{wrt2rd\_ptr[6:0]})$$

Generating Full Condition:
The FIFO is full when the synchronized write pointer (with respect to read clock) has wrapped around and reached the read pointer.

$$\texttt{full} = (\texttt{rd\_ptr[6:0]} == \{!\texttt{wrt2rd\_ptr[6:5]}, \texttt{wrt2rd\_ptr[4:0]}\})$$

Illustration to help understand full condition in gray-code pointers:
Assume binary code for read pointer = 0010101 and write pointer = 1010101. This should result in full condition.
Gray codes for read pointer = 0011111 and write pointer = 1111111. Note that the 2 MSBs are complements whereas the rest of the bits are same.

Complete Asynchronous FIFO is illustrated in the following figure.

Figure 5: Asynchronous FIFO interal circuit

Verilog Modeling and Verification

The Asynchronous FIFO described earlier is modeled using Verilog. The FIFO is synthesized using Intel Quartus Lite (18.1) and implemented on MAX-10 FPGA. The RTL view of the synthesized FIFO is shown below.

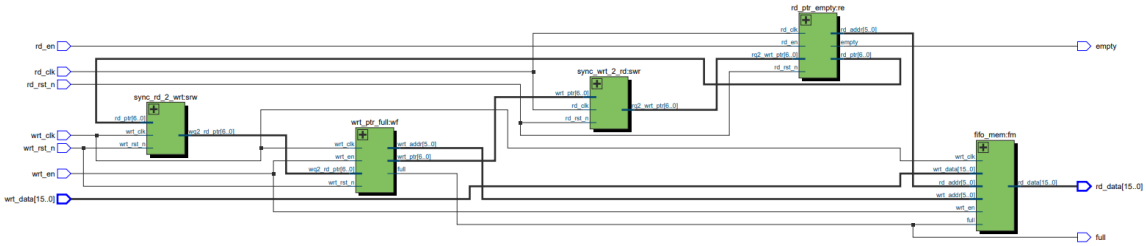

Figure 6: RTL View of Asynchronous FIFO (Intel Quartus)

Verilog testbench is used to check the working of the FIFO. Initially both pointers are reset. Then data is written into the FIFO in bursts of 50 at 50 MHz frequency and the same data is read out at 10 MHz frequency. Result of Gate level simulation using ModelSim Altera is shown below.

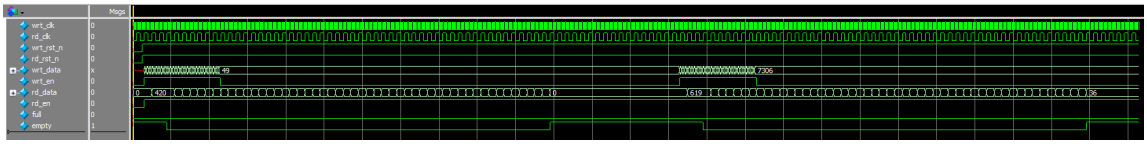

Figure 7: Gate Level Simulation (ModelSim Altera)

Note that the FIFO is empty when all the data has been read out, but is never full because the depth is 64, but it only needed a depth of 40 to function properly for the given specifications.

This FIFO should ideally be able handle a write data burst of up-to 80 (if data is also continuously being read). But due to synchronizer delays, it can handle a write data burst of 74 before becoming full.