TECHNION-
DEPARTMENT OF ELECTRICAL ENGINEERING
DEPARTMENT OF COMPUTER SCIENCE

The VLSI systems research and laboratories

# Out-Of-Order Asynchronous Synchronizer
# Design, syntheses, implementation and performance investigation

Ameer Abdel-Hadi & Rami Busool
Supervisor: Rostislav (Reuven) Dobkin

Technion, March 2006

THE VLSI SYSTEMS RESEARCH LABORATORIES

# OOO Asynchronous Synchronizer Design, syntheses, implementation and performance investigation

Ameer Abdel-Hadi and Rami Busool
Supervised by:
Rostislav (Reuven) Dobkin

## ABSTRACT

Shrinking device sizes and increasingly complex designs have created multimillion transistor systems running with multiple asynchronous clocks with frequencies as high as multiple gigahertz. SoC (System on Chip) systems have multiple interfaces, some using standards with very different clock frequencies or even asynchronous. Several modern serial interfaces are inherently asynchronous from the rest of the chip or interfaced to a variety of external busses ticking at different frequencies or different asynchronous protocols **GALS (globally asynchronous, locally synchronous)** systems. There is also a trend toward designing major sub-blocks of SoCs to run on independent clocks to ease the problem of clock skew across large chips besides the gain of area, power and design time. Synchronization is needed for crossing clock domain or passing asynchronous events to synchronous domain, for that we use a synchronizer. The basic synchronizer consumes high throughput and latency; many academic researches have been done in order to investigate and invent new synchronization methodologies, our research is one among these, which investigates a parallel out-of-order synchronizer for packed data (data bus, group of bits) synchronization. The main idea is to synchronize each coming data synchronization request in different basic synchronizer; the data will be committed to the synchronous domain at the same order of its coming. A full specification, design, and simulation of the proposed synchronizer are presented. We compare and contrast our implementation with existing synchronous versions, the FIFO synchronizer and the basic brute-force two flops synchronizer.

## תקציר

מזעור מימדים של התקנים אלקטרוניים בנוסף למורכבות שהולכת וגוברת הובילה למערכות עם כמה מיליוני טרנזיסטורים שרצה עם מספר גדול של שעונים לא מסנכרנים בתדרים של כמה גיגה-הרץ.
למערכות SoC (מערכת על שבב) כמה ממשקים שעובדים עם תקנים שונים שלכל אחד תדר שעון שונה ויתכן שהוא לא מסנכרן לשעון כלל. ממשקים טוריים מודרניים לא מסנכרנים לשאר השבב או שכל ממשק מסנכרן לאפיק נתונים שונה שעובד בתדר שונה או פרוטוקול תקשורת לא סינכרוני שונה (מערכות **GALS**, גלובלי לא סינכרוני, מקומי סינכרוני). בנוסף יש נטייה לכיוון תכנון של יחידות עיקריות של SoC כך שירוצו עם שעונים בלתי תלויים כל אחד, בכדי להתגבר על בעיות סטייה בשעונים לאורך שבבים גדולים, בנוסף לרווח בשטח, הספק וזמן תכנון. בשביל מעבר בין תחומי שעונים שונים , וגם למעבר מתחום לא סינכרוני לתחום אחר סינכרוני, חייבים סינכרוניזציה, משימה זו מבצע המסנכרן. מסנכרן בסיס צורך זמני תפוקה ואחזור גבוהים, הרבה מחקרים אקדמיים מתעסקים בחקר והמצאה של מתודולוגיות סנכרון חדישות, המחקר שלנו הוא אחד מאלה שחוקר מסנכרן שמסנכרן נתונים (בקבוצות) במקביל לא דווקא לפי סדר הופעתם בכניסה (out-of-order). העיקרון המרכזי הוא סנכרון כל בקשת סנכרון של מידע חדש שמגיע במסנכרן בסיסי נפרד, המידע יימסר לתחום הסינכרוני לפי סדר הופעתו בכניסה. במחקר מוצגים מפרט מלא, תכנון וסימולציה של המסנכרן המוצע. בנוסף הצגנו השוואה של המסנכרן המוצע מול מסנכרנים אחרים: מסנכרן FIFO (ראשון נכנס, ראשון יוצא) ומסנכרן שני-דלגלגים בסיסי.

**Index**


1. Introduction……10
2. Theoretical background……12
   2.1. Synchronization……12
      2.1.1. Overview……12
      2.1.2. Classification of signal-clock synchronization……14
   2.2. Metastability……14
   2.3. Mean Time Between Failures (MTBF)……17
   2.4. Brute-Force (Waiting) Two flops synchronizer……20
      2.4.1. Overview……20
      2.4.2. Push Synchronizer……20
      2.4.3. Brute-Force synchronizer failure……22
      2.4.4. Two Synchronizer Calculation Examples……24
   2.5. FIFO synchronizer……25
      2.5.1. Overview……25
      2.5.2. Plesiochronous FIFO Synchronizer……26
      2.5.3. Flow control……27
      2.5.4. General Purpose Asynchronous FIFO Synchronizer……27
3. Out-of-Order Synchronizer……28
   3.1. Overview……28
   3.2. RTL and circuit design……30
      3.2.1. Definitions package (DEF)……30
         3.2.1.1. Functional description……30
         3.2.1.2.RTL description……30
      3.2.2. Asynchronous controller (AC0/ACi)……32
         3.2.2.1. Functional description……32
         3.2.2.2. RTL description……38
         3.2.2.3. Circuit decription……38
      3.2.3. N Flip-flop (NFF)……39
         3.2.3.1. Functional description……39
         3.2.3.2. RTL description……39
         3.2.3.3. Circuit description……39
      3.2.4. Out-of-Order (OOO)……40
         3.2.4.1. Functional description……40
         3.2.4.2. RTL description……40
         3.2.4.3. Circuit description……41
         3.2.4.4. Bit-Slice implementation……41
      3.2.5. Synchronizer (SYN)……44
         3.2.5.1. Functional description……44
         3.2.5.2. RTL description……44
         3.2.5.3. Circuit description……45
         3.2.5.4. Bit-Slice Implementation……45
      3.2.6. Test-Bench (TBN)……47
         3.2.6.1. Functional description……47
         3.2.6.2. RTL description……47
         3.2.6.3. Circuit description……48
      3.2.7. Checker (CHK)……49
         3.2.7.1. Functional description……49
         3.2.7.2. RTL description……49

3.3. Implementation........................................................................50
3.3.1. Platform..................................................................................50
3.3.2. Tools.......................................................................................50
3.3.2.1. Asynchronous Syntheses tools..........................................50
3.3.2.2. Synchronous design tools................................................51
3.4. System verification and simulation...............................................52
3.4.1. Verification.............................................................................52
3.4.2. Simulation...............................................................................52
4. Results..................................................................................................53
5. Conclusions and further work...............................................................56
6. Appendixes...........................................................................................57
6.1. Asynchronous Design Flow.............................................................57
6.2. Synchronous Design Flow...............................................................59
7. References.............................................................................................63
7.1. Articles...........................................................................................63
7.2. Web Sites........................................................................................63
7.3. Books..............................................................................................63

**Figures Index**

Figure 1: synchronization.................................................................. 12
Figure 2: Output at Intermediate Level Due to Data Edge Within t0 Aperture......14
Figure 3: D Flip-Flop timing Scheme..................................................15
Figure 4: Output of Flip-Flop oscillation at metastable state........................ 15
Figure 5: Quasi-Stable sate on the top of the hill.......................................16
Figure 6: Typical MTBF of a flip-flop...................................................18
Figure 7: Two flip-flops in serial........................................................19
Figure 8: Brute-Force (Waiting) Two flops synchronizer..............................20
Figure 9: A push synchronizer...........................................................20
Figure 10: Four-phase handshake push synchronization protocol STG...............21
Figure 11: Push synchronizer logic and protocol FSM.................................21
Figure 12: Synchronization failure, still in metastable state..........................22
Figure 13: Aperture time ta, clock period tcy..........................................22
Figure 14: Initial voltage difference ΔVs, and final voltage difference ΔVf..........23
Figure 15: FIFO synchronizer scheme...................................................25
Figure 16: Plesiochronous FIFO synchronizer scheme................................26
Figure 17: General Purpose Asynchronous FIFO Synchronizer........................27
Figure 18: Out-Of-Order asynchronous synchronizer...................................29
Figure 19: Wave form for important interface signals and timing constants......... 31
Figure 20: "AC0/ACI" – Asynchronous Controllers STG................................33
Figure 21: 4-phase handshaking with outer interface user waveform/simulation.....34
Figure 22: Cyclic trigger protocol waveform/simulation................................34
Figure 23: REn, gating requests signal, with Prv and RqO waveform/simulation....35
Figure 24: Latching signals waveform/simulation.......................................35
Figure 25: RqI/AcI 4-phase protocol/simulation.........................................35
Figure 26: "AC0/ACI" – Asynchronous Controllers STG – critical path..............36
Figure 27: "AC0" – critical transmissions on "AC0" - waveform/simulation.........37
Figure 28: "ACI" – critical transmissions on "ACI" - waveform/simulation.........37
Figure 29: NFF – N Flip-Flops Synchronizer............................................39
Figure 30: OOO – Out Of Order Unit.....................................................42
Figure 31: OOO – Out Of Order Unit – Bit-Slice Design...............................43
Figure 32: SYN – Synchronizer...........................................................46
Figure 33: CHK – Checker................................................................ 49
Figure 34: Top simulation with Clk @ 1.23GHZ (max freq)............................52
Figure 35: Top simulation with Clk @ 100MHZ.......................................... 52
Figure 36: Top simulation with Clk @ 10MHZ........................................... 52
Figure 37: ac0.g/ac0.g.ps................................................................. 57
Figure 38: aci.g/aci.g.ps...................................................................57
Figure 39: File→read.......................................................................59
Figure 40: Choose file......................................................................60
Figure 41: Command window............................................................. 60
Figure 42: set_dont_touch..................................................................60
Figure 43: Window→Hierarchy, Wave, Source...........................................61
Figure 44: All Group........................................................................61
Figure 45: Choose trace signals...........................................................62
Figure 46: Step Time & Run...............................................................62
Figure 47: Waveform........................................................................62

**Tables Index**


Table 1: Classification of Signal-Clock Synchronization……13
Table 2: "DEF" Constants……30
Table 3: "DEF" Data types……30
Table 4: "DEF" Functions……30
Table 5: Critical path delay by arcs……36
Table 6: "AC0/ACI" Interface signals……38
Table 7: "NFF" Interface signals……39
Table 8: "NFF" Internal Signals……39
Table 9: "OOO" Interface signals……40
Table 10: "OOO" Internal signals…… 40
Table 11: "SYN" Interface signals……44
Table 12: "SYN" Internal signals……44
Table 13: "SYN" Components……44
Table 14: "TBN" Interface signals……47
Table 15: "TBN" Internal signals……47
Table 16: "CHK" Internal Signals……49
Table 17: "CHK" Components……49
Table 18: OOO/FIFO Area comparison……55
Table 19: Results……55

## 1. Introduction

Designers of digital systems are constantly confronted with the problem of synchronizing two systems that operate at different frequencies. The problem is usually resolved by synchronizing one of the signals with the local clock generator using a flip-flop. But such a solution, of necessity, leads to a violation of the operating conditions for the flip-flops, in these cases, the setup time and hold time are not maintained. Therefore, a flip-flop can go into a metastable state, endangering the operability of the circuit and, thereby, the reliability of the whole system.
Synchronization is a challenging topic that has been investigated intensively; large VLSI chips tend to employ asynchronous intermodule timing due to two principal reasons. First, it is sometimes more economical (in terms of area, power and design time) to break a large synchronous chip (or section of a chip) into multi-synchronous modules, which use the same basic clock frequency but do not require the exact same phase of the clock. Multi-synchronous timing can be based on thrifty clock distribution networks, which avoid the heavy area and power penalty of assuring minimal skew across a large chip. Second, interfacing the chip to a variety of external busses ticking at different frequencies imposes a requirement for the chip to contain multiple unrelated clock domains. Both types of multiple-clock domain chips are sometimes termed **GALS (globally asynchronous, locally synchronous)** systems.
A **clock domain** is defined as that part of the design driven by either a single clock or clocks that have constant phase relationships. A clock and its inverted clock or its derived divide-by-two clocks are considered a clock domain (synchronous). Conversely, domains that have clocks with variable phase and time relationships are considered different clock domains.
In synchronous design, the master clock acts as a timing reference signal for all basic modules of the design. A welldesigned global clock distribution network, with additional circuitry for keeping clock skew under reasonable limits is required to make sure that local clock signals reaching different computational blocks are synchronized. However, due to the increasing number of transistors and complexity of today's designs, a continuous reduction in clock skew is possible only by careful design and simultaneous consideration of global interconnect delay increase.
In recent years, the GALS approach has been explored to tackle this problem. Such a solution eliminates the requirement of a global reference clock signal by assuming that the system is comprised of several synchronous blocks communicating asynchronously.
Two separate clock domains are **'mesochronous'** if they are clocked at the same frequency but at a fixed relative phase difference. If the phase difference drifts over time, they are called **'multi-synchronous'**. If the clock frequencies are close but different, they are **'plesiochronous'**. **Periodic** signals have arbitrary frequencies but the phase difference is periodic. This information can be used to predict which signal events occur during the unsafe portion of the clock.
**Asynchronous** signal is not periodic, which means that the signal events may occur at arbitrary time, this compels full brute-force (two flops) synchronizer.
In multi-synchronous GALS systems, all modules receive the same clock frequency. The design of inter-module communications can take advantage of that

fact and employ mesochronous synchronizers for higher bandwidth than possible with the more general two-flop synchronizers.
However, relative clock phases drift over time (typically due to intra-die temperature and voltage temporal variations) thus requiring adaptive multisynchronous synchronizers that either periodically or continuously adapts to the varying phase differences. Similar conditions often arise among separate chips on a board, where the chips are clocked by the same system clock. The analysis of synchronizers for on-chip cross-clock domain communications is quite difficult. Circuit simulations of synchronizers only provide a partial characterization. Typical logic validation tools are totally ignorant of synchronization failures. Postproduction testing also provides very little help. The only useful metric proposed in the literature is that of MTBF, which is only indirectly driven out of approximately defined parameters.

# 2. Theoretical background
## 2.1. Synchronization
### 2.1.1. Overview

Large systems on chip (SoC) typically contain multiple clock domains. Inter-domain communications require data synchronization, which must avoid metastability while typically facilitating low latency, high bandwidth, and low power safe transfer. The synchronizer must also prevent missing any data or reading the same data more than once. Communicating clock domains can be classified according to the relative phase and frequency of their respective clocks.

**Heterochronous** or **periodic** domains operate at nominally different frequencies, **plesiochronous** domains have very similar clock frequencies, **multi-synchronous** domains have the same clock frequency but a slowly drifting relative phase, and **mesochronous** domains have exactly the same frequency.

The simplest solution for inter-domain data transfer is the **two-flip-flop** synchronizer. The main problem with that synchronizer is its long latency (1-2 cycles): Typically, a complete transfer incurs waiting about one to two clock cycles at each end. Although it is a very robust solution, it is sometimes misused or even abused in an attempt to reduce its latency.

Another commonly used synchronizer is based on dual-clock **FIFO**. In certain situations, especially when a complete data packet of a pre-defined size must be transferred, this may be an optimal solution. Another advantage is that synchronization is safely contained inside the FIFO, relieving designers of the communicating domains of this delicate design task.

The main drawback of FIFOs is their one-to-two cycle latency that is incurred when the FIFO is either full or empty, and that scenario is highly typical with periodic clock domains where the clock frequencies are different.

**Mesochronous** synchronizer is a rational clocking synchronizer for a special case of periodic domains in which the two clocks are related by the ratio of two small integers.

Another synchronizer for a limited cases of periodic the **plesiochronous** synchronizer. It incorporates an exclusion detection controlling a multiplexer that selects either the data or a delayed version of it. While having a low latency and a "duplicate and miss" algorithm for the plesiochronous case, it is inapplicable to periodic domains.

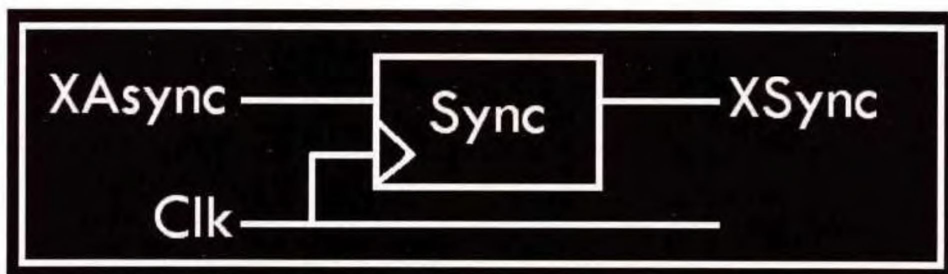



**Figure 1: synchronization**

### 2.1.2. Classification of signal-clock synchronization

**Synchronous**
Signal has the same frequency and phase as the clock.
It is safe to sample the signal directly with the clock (synchronizer is not needed)

**Mesochronous**
Signal has the same frequency as the clock but is potentially out of phase, with a phase difference φc.
It is safe to sample the signal if the clock or signal is delayed by a constant amount

**Plesiochronous**
Signal and clock have nearly the same frequency, and hence the phase difference varies slowly.
It is safe to sample the signal if the clock or signal is delayed by a variable amount.

**Periodic**
Signal and clock have arbitrary frequencies but the phase difference is periodic.
This information can be used to predict which signal events occur during the unsafe portion of the clock, i.e., close to the sampling edge of the clock

**Asynchronous**
Signal is not periodic, which means that the signal events may occur at arbitrary times.
Full brute-force synchronizer is needed!

| Type | Frequency | Phase |
|---|---|---|
| Synchronous | Same | Same |
| Mesochronous | Same | Constant |
| Plesiochronous | Small Difference | Slowly Varying |
| Periodic | Different | Periodic Variation |
| Asynchronous | N/A | Arbitrary |

**Table 1: Classification of Signal-Clock Synchronization**

## 2.2. Metastability

**What is metastability?**

Metastability in digital systems occurs when two asynchronous signals combine in such a way that their resulting output goes to an indeterminate state. A common example is the case of data violating the setup and hold specifications of a latch or a flip-flop.

In a synchronous system, the data always has a fixed relationship with respect to the clock. When that relationship obeys the setup and hold requirements for the device, the output goes to a valid state within its specified propagation delay time. However, in an asynchronous system, the relationship between data and clock is not fixed; therefore, occasional violations of setup and hold times can occur. When this happens, the output may go to an intermediate level between its two valid states and remain there for an indefinite amount of time before resolving itself or it may simply be delayed before making a normal transition. In either case, a metastable event has occurred.

Metastable events can occur in a system without causing a problem, so it is necessary to define what constitutes a failure before attempting to calculate a failure rate. For a simple D Flip-Flop, as shown in Figure 2, valid data must be present on the input for a specified period of time before the clock signal arrives (setup time) and must remain valid for a specified period of time after the clock transition (hold time) to assure that the output functions predictably (see figure 3). This leaves a small window of time with respect to the clock (t0) during which the data is not allowed to change. If a data edge occurs within this aperture, the output may go to an intermediate level and remain there for an indefinite amount of time before resolving itself either high or low, as illustrated in Figure 2. This metastable event can cause a failure only if the output has not resolved itself by the time that it must be valid for use (for example, as an input to another stage); therefore, the amount of resolve time allowed a device plays a large role in calculating its failure rate.

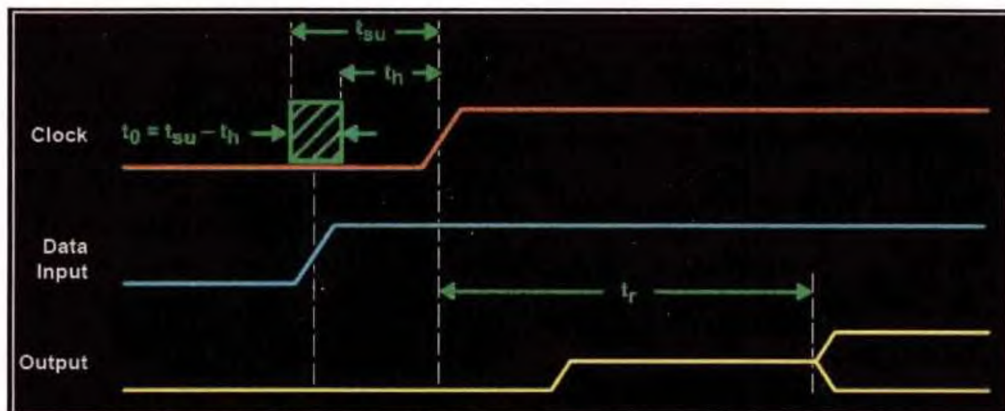



**Figure 2: Output at Intermediate Level Due to Data Edge Within t0 Aperture**

When ever there is setup and hold time violations in any flip-flop, Flip-flop enters a state, where its output is unpredictable, this state is know as metastable state (quasi stable state), at the end of metastable state, Flip-Flop settles down to either '1' or '0'. This whole process is known as metastability. In figure 3 below Tsu is setup time and Th is hold time. When ever the input

signal D does not meet the Tsu and Th of the given D Flip-Flop, metastability occurs.

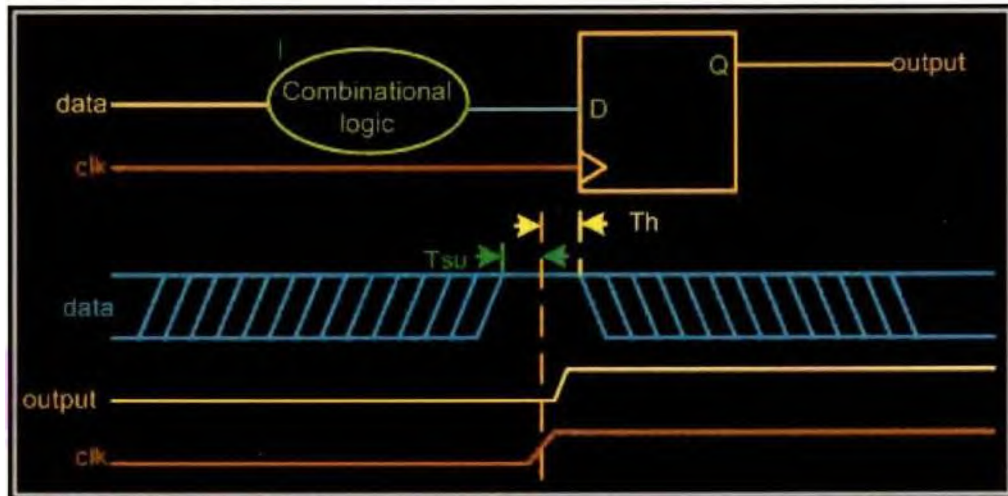



**Figure 3: D Flip-Flop timing Scheme**

When Flip-Flop is in metastable state, the output of the flip-flop oscillate between '0' and '1' as shown in figure 4 below (here the flip-flop output settles down to '0') . How long it takes to settle down, is depending on the technology of the flip-flop

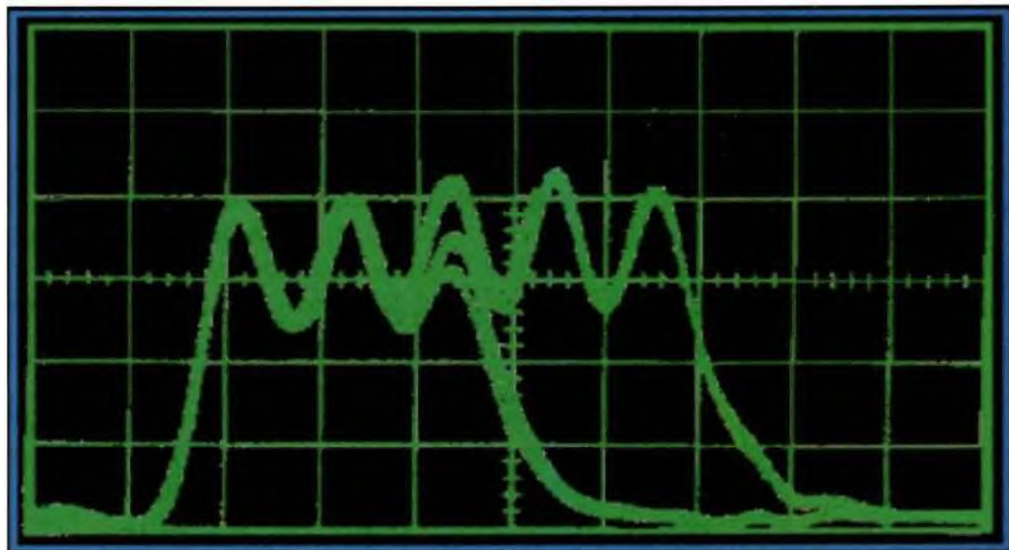


**Figure 4: Output of Flip-Flop oscillation at metastable state**

If we look deep inside the flip-flop we see that, the quasi-stable state is reached when a flip-flop's setup and hold times are violated. Assuming the use of a positive edge triggered "D" type flip-flop, when the rising edge of the flip-flop's clock occurs at a point in time when the D input to the flip-flop is causing its master latch to transition, the flip-flop is highly likely to end up in a quasi-stable state . This rising clock causes the master latch to try to capture its current value while the slave latch is opened allowing the Q output to follow the "latched" value of the master. The more perfectly "caught" quasi-stable state (on the very top of the hill, figure 5) results in the longest time required for the flip-flop to resolve itself to one of the more stable states.

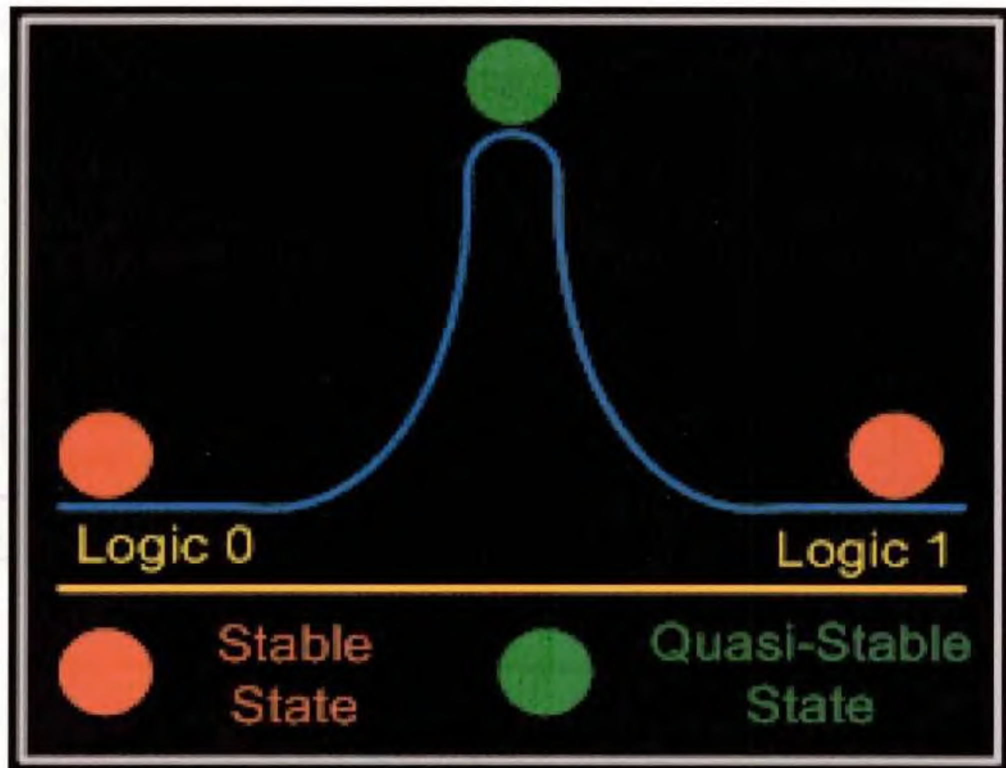



**Figure 5: Quasi-Stable sate on the top of the hill**

**How long does it stay in this state?**
The relative stability of states shown in Figure 5 shows that the logic 0 and logic 1 states (being at the base of the hill) are much more stable than the somewhat stable state at the top of the hill. In theory, a flip-flop in this quasi-stable hilltop state could remain there indefinitely but in reality it won't. Just as the slightest air current would eventually cause a ball on the illustrated hill to roll down one side or the other, thermal and induced noise will jostle the state of the flip-flop causing it to move from the quasi-stable state into either the logic 0 or logic 1 state.

**What are the cases, when metastability occurs?**
As we have seen, when ever setup or hold time violation occurs, metastability occurs, so when does this signal violate this timing requirement?

- The input signal is an asynchronous signal
- The clock skew (rise time and fall time) is more than the tolerable values).
- Interfacing two domains operating at two different frequency or same frequency but different phase.
- The combinational delay is such way that, data input of flip-flop changes in the required window (setup+hold window)

### 2.3. Mean Time Between Failures (MTBF)

**What is MTBF?**

MTBF is Mean time between failures, what does that mean?
MTBF gives us information on how often a particular flip-flop will fail, in other words, it gives the time interval between two successive failures.
Figure 6 below shows a typical MTBF of a flip-flop and also it gives the MTBF equation.
The probability of a metastable state persisting longer than a time $t_r$ decreases exponentially as $t_r$ increases. This relationship can be characterized by equation 2.1:

(2.1) $$f(r) = e^{(-t_r/\tau)}$$

Where the function $f(r)$ is the probability of non-resolution, as a function of resolve time allowed $t_r$ and circuit time constant $\tau$.

For a single-stage synchronizer with a given clock frequency and an asynchronous data edge that has a uniform probability density within the clock period, the rate of generation of metastable events can be calculated by taking the ratio of the setup and hold time window previously described to the time between clock edges and multiplying by the data edge frequency. This generation rate of metastable events coupled with the probability of non-resolution of an event as a function of the time allowed for resolution gives the failure rate for that set of conditions. The inverse of the failure rate is the mean time between failures (MTBF) of the device and is calculated with the formula shown in equation 2.2:

(2.2) $$\frac{1}{failure\ rate} = MTFB_1 = \frac{e^{(t_r/\tau)}}{t_0 f_c f_d}$$

Where:

| | |
|---|---|
| $t_r$ | = resolve time allowed in excess of the normal propagation delay time of the device |
| τ | = metastability time constant for a flip-flop |
| $t_0$ | = a constant related to the width of the time window or aperture wherein a data edge triggers a metastable event |
| $f_c$ | = clock frequency |
| $f_d$ | = asynchronous data edge frequency |

The parameters $t_0$ and $t$ are constants that are related to the electrical characteristics of the device in question. The simplest way to determine their values is to measure the failure rate of the device under specified conditions and solve for them directly. If the failure rate of a device is measured at different resolve times and plotted, the result is an exponentially decaying curve. When plotted on a semi-logarithmic scale, this becomes a straight line the slope of which is equal to $\tau$ therefore, two data points on the line are sufficient to calculate the value of $\tau$ using equation 2.3:

(2.3)
$$\tau = \frac{t_{r_2} - t_{r_1}}{\ln(N_1 - N_2)}$$

Where:

$t_{r_1}$ = resolve time 1

$t_{r_2}$ = resolve time 2

$N_1$ = number of failures relative to $t_{r_1}$

$N_2$ = number of failures relative to $t_{r_2}$

After determining the value for $\tau$, $t_0$ may be solved for directly. The formula for calculating the MTBF of a two-stage synchronizer, equation 2.4, is merely an extension of equation 2.2:

(2.4)
$$MTFB_2 = \frac{e^{(t_{r1}/\tau)}}{t_0 f_c f_d} \times e^{(t_{r2}/\tau)}$$

Where:

$t_{r_1}$ = resolve time allowed for the first stage of the synchronizer

$t_{r_2}$ = resolve time allowed in excess of the normal propagation delay

$f_c$, $f_d$, $t$, and $t_0$ are as previously defined, with $t$ and $t_0$ assumed to be the same for both stages.

The first term calculates the MTBF of the first stage of the synchronizer, which in effect becomes the generation rate of metastable events for the next stage. The second term then calculates the probability that the metastable event will be resolved based on the value of $t_{r_2}$, the resolve time allowed external to the synchronizer. The product of the two terms gives the overall MTBF for the two-stage synchronizer.

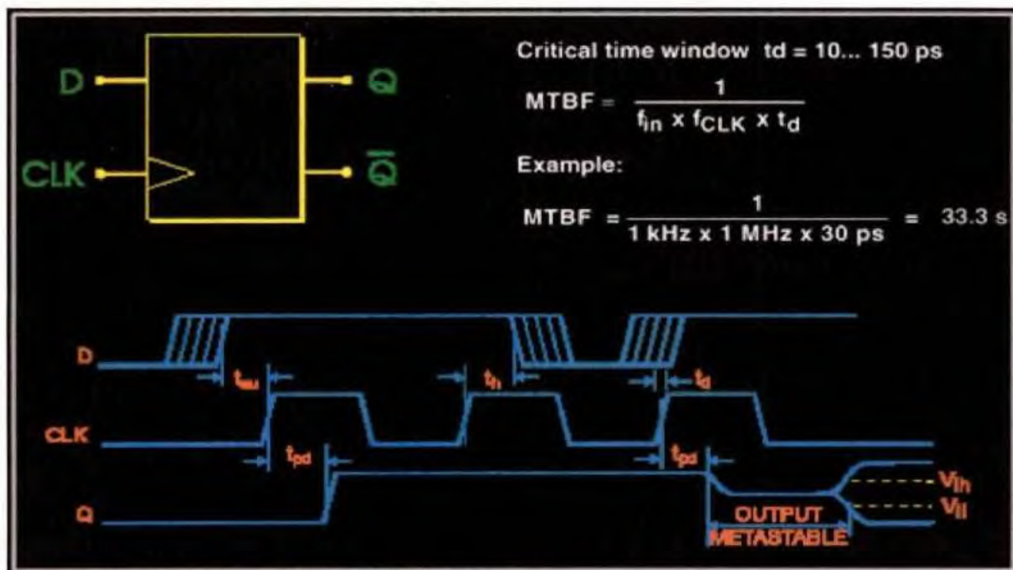



**Figure 6: Typical MTBF of a flip-flop**

**How do I avoid metastability?**

In reality, one cannot avoid metastability and increase clock-to-Q delays in synchronizing asynchronous inputs without the use of tricky self-timed circuits. So a more appropriate question might be "How do you tolerate metastability?"

In the simplest case, designers can tolerate metastability by making sure the clock period is long enough to allow for the resolution of quasi-stable states as well as whatever logic may be in the path to the next flip-flop. This approach, while simple, is rarely practical given the performance requirements of most modern designs.

The most common way to tolerate metastability is to add one or more successive synchronizing flip-flops to the synchronizer. This approach allows for an entire clock period (except for the setup time of the second flip-flop) for metastable events in the first synchronizing flip-flop to resolve themselves. This does, however, increase the latency in the synchronous logic's observation of input changes.

Neither of these approaches can guarantee that metastability cannot pass through the synchronizer; they simply reduce the probability to practical levels.

In quantitative terms, if the Mean Time Between Failure (MTBF) of a particular flip-flop in the context of a given clock rate and input transition rate is 33.33 seconds then the MTBF of two such flip-flops used to synchronize the input would be (33.33* 33.33) = 18.514 Minutes. Well I have taken a worst flip-flop ever designed in history of man kind. Figure 7 below shows a how to connect two flip-flops in series to achieve this and also the resultant MTBF.

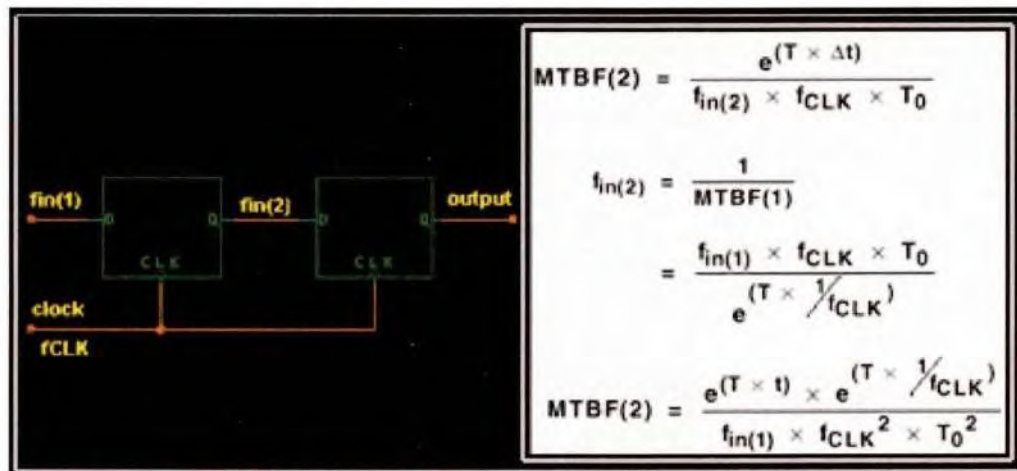



**Figure7: Tow flip-flops in serial**

Normally

- We can use a metastable hardened flip-flop
- Cascade two or three D-Flip-Flops (two or three stage synchronizer).

## 2.4. Brute-Force (Waiting) Two flops synchronizer

### 2.4.1. Overview

The second flip-flop receives the output signal of the first stage one clock period later and can go into a metastable state only if its input conditions are also violated. That is, the output of the first flip-flop is still metastable during its setup and hold time.
At Figure 7, the Brute-Force synchronizes the asynchronous input A to the clock Clk, First flip-flop FF1 samples A (may go into a metastable state, depending on the timing of A and Clk) then we await the possible metastable state to end for a waiting period tw. Usually the waiting time is one clock cycle, which means that the output of FF1 is sampled by the second flip-flop FF2 to generate the final synchronized signal AS. In general, to implement an N-cycle waiting period, we need N cascaded flip-flops in addition to the sampling flip-flop FF1; Total synchronization latency is $N$+1 clock cycles.

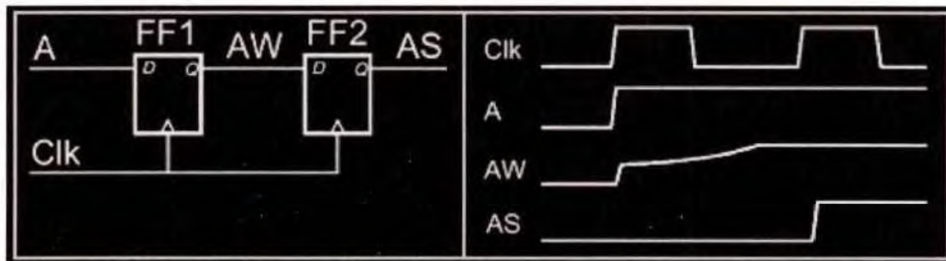



**Figure 8: Brute-Force (Waiting) Tow flops synchronizer**

### 2.4.2. "Push" Synchronizer

A "push" synchronizer is shown in Figure 9, but the principles apply also to pull, push-pull, and control-only synchronizers.

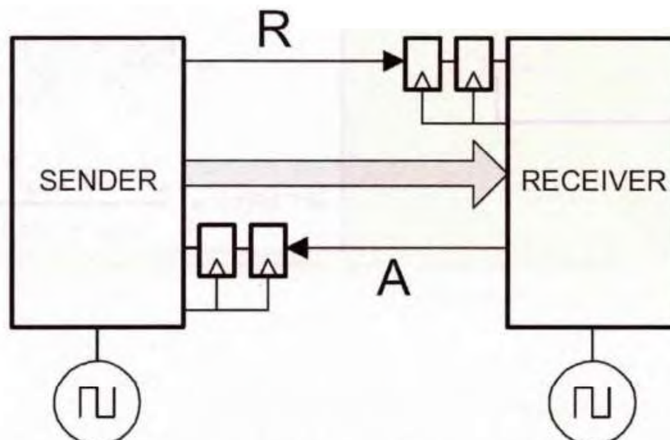



**Figure 9: A push synchronizer**

Bundled data is employed. The "synchronizer" actually comprises two synchronization circuits that envelope the data lines, implementing a complete handshake protocol. The Request (R) and Acknowledge (A) lines are synchronized by the receiver and sender, respectively.

The two synchronizers connect two simple finite state machines that implement the required protocol. A fourphase protocol is specified by means of a generalized STG in Figure 10, where "DD" means that the data is available (at the sender), "UU" means that it may be removed, and "LL" means data latched by the receiver. (A two-phase protocol may also be employed; the circuits are a bit more complex, and this is typically used in order to minimize latency on long lines.) The complete logic and FSM are shown in Figure 11. A send request (V, true for a single cycle) latches data into REGs and starts the sender's FSM. The synchronized request (R2) latches the data into REGR and triggers the receiver's FSM. The receiver is given a single-cycle "data received" (D) signal. The protocol is sometimes modified so that A is set as soon as the received data are latched, but removed only after the receiver has had an opportunity to use the data.

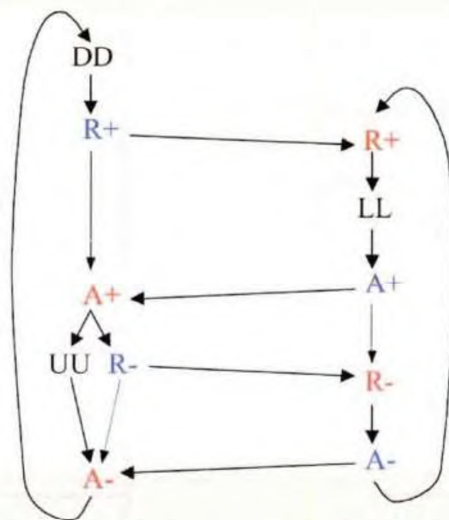



**Figure 10: Four-phase handshake push synchronization protocol STG**

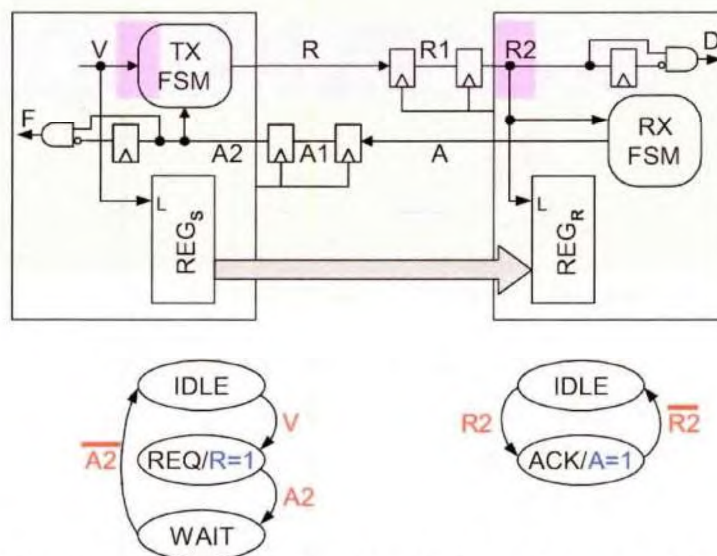



**Figure 11: Push synchronizer logic and protocol FSM**

### 2.4.3. Brute-Force Synchronization Failure

If the first flip-flop FF1 is still in a metastable state after the one-cycle waiting period (figure 12), flip-flop FF2 samples a metastable state, in general, the probability of synchronization failure can be calculated as follows:

(2.5) $P(failure) = P(enter\ metastable\ state) \cdot P(still\ in\ metastable\ after\ t_w)$ Or $P_F = P_E \cdot P_S$

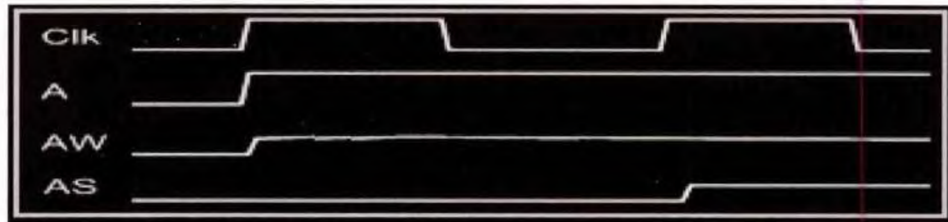



**Figure 12: Synchronization failure, still in metastable state**

Flip-flop can enter a metastable state, when its data input D changes the state during the aperture time or sampling window of the flip-flop. Probability of an input transition to occur during the sampling window is computed by dividing the aperture time ta by the clock period tcy (figure 13).

(2.6) $$P_E = \frac{t_a}{t_{cy}} = t_a f_{clk}$$

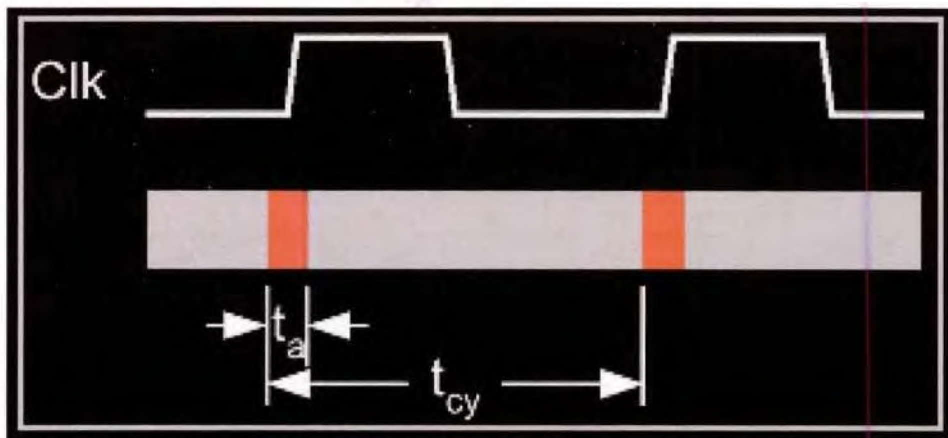



**Figure 13: aperture time ta, clock period tcy**

Flip-flop is still in the metastable state after the waiting period $t_w$, if the initial voltage difference $\Delta V_S$ was too small to be exponentially amplified to the full value 1 during the waiting period. Hence, the final value $\Delta V_F$ is smaller than 1.

Probability of this to happen is defined to be the ratio of such a small initial difference

$\Delta V_S$ to the corresponding final value $\Delta V_F$ (figure 14).

(2.7) $$\Delta V_F = \Delta V_S \cdot e^{t_w/\tau_S}, P_S = \frac{\Delta V_S}{\Delta V_F} = e^{t_w/\tau_S}$$

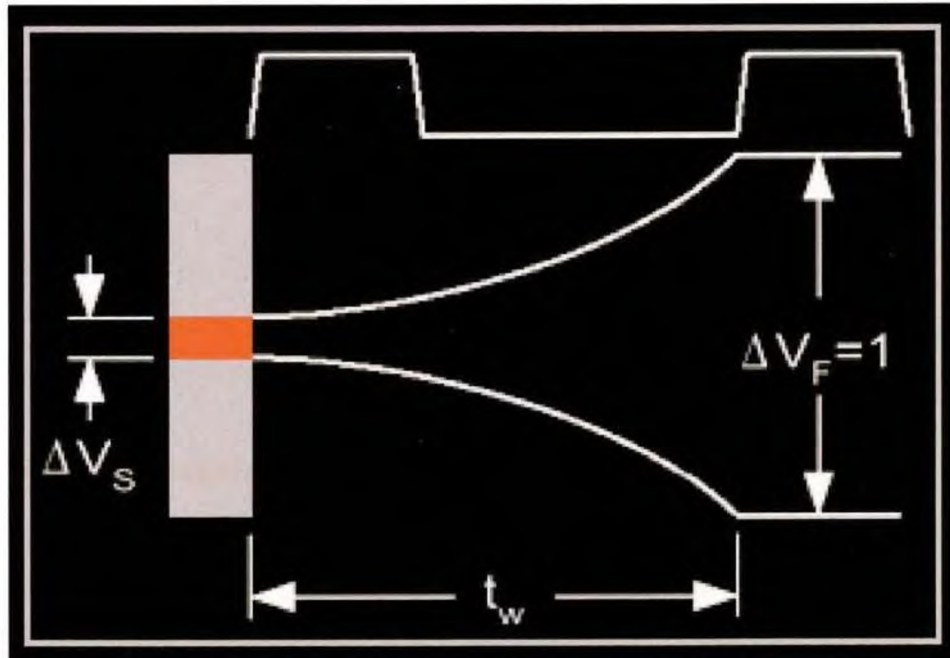



**Figure 14: initial voltage difference ΔVs, and final voltage difference ΔVf**

Synchronization failure probability $P_F$ is the product of the probabilities of entering and staying in the metastable state $P_E$ and $P_S$.
Potentially, every event at the flip-flop data input D can cause a synchronization failure with the probability $P_F$.
The synchronization failure frequency (synchronization error rate) $f_F$ is calculated by multiplying the failure probability $P_F$ by the event frequency $F_D$.

(2.8) $$P_F = P_E P_S = t_a f_{clk} e^{-t_w/\tau_S}, f_F = f_D P_F = t_a f_D f_{clk} e^{-t_w/\tau_S}$$

### 2.4.4. Two Synchronizer Calculation Examples

Example 1:

- Assume that a 1 MHz data signal (fD) is sampled with a 100 MHz clock signal (fClk)
- Aperture time ta and the regeneration time constant τs of the sampling flip-flop are both 200 ps.
- Waiting period tw is one clock cycle (we have the second flip-flop): tw = 1/ fClk = 10 ns
- We get:
  PE = 0.02
  PS = 1.94·10 -22
  PF = 3.88·10 -24
  fF = 3.88·10 -18 Hz
  MTBF = 1/ fF = 2.58·10e17 s
  = 8.2 ·10e9 years
  ("Mean Time Between Failures")

Example 2:

- Assume that a 10 MHz data signal ($f_D$) is sampled with a 500 MHz clock signal ($f_{Clk}$)
- Aperture time $t_a$ and the regeneration time constant $\tau_s$ of the sampling flip-flop are both 100 ps
- Waiting period $t_w$ is one clock cycle (we have the second flip-flop): $t_w = 1/f_{Clk} = 2$ ns
- We get:
  $P_E = 0.05$
  $P_S = 2.07 \cdot 10^{-9}$
  $P_F = 1.04 \cdot 10^{-10}$
  $f_F = 1.04 \cdot 10^{-3}$ Hz
  MTBF = $1/f_F$ = 961 s
  = 16 min !!

## 2.5. FIFO synchronizer

### 2.5.1. Overview

FIFO synchronizer is built usually from three sampling flip-flops per data bit (3- place FIFO buffer), FIFO buffer is filled using the transmitter clock xclk and the associated counter which provides the transmit pointer xp, FIFO buffer is emptied using the receiver clock rclk and the associated counter which provides the receive pointer rp, receive pointer follows the transmit pointer one step behind, i.e., when xp is 0,1,2 , rp is 2,0,1, respectively.
Each flip-flop output x0, x1, or x2 is updated on every 3rd clock cycle and passed to the MUX output xs as a clock-cycle-wide sample during each 3-cycle round, this gives a clock-cycle-wide timing margin on both sides of each sample.
The 3-place FIFO synchronizer provides a stable and correct output as long as the phase difference between xclk and rclk is between – tcy and tcy.

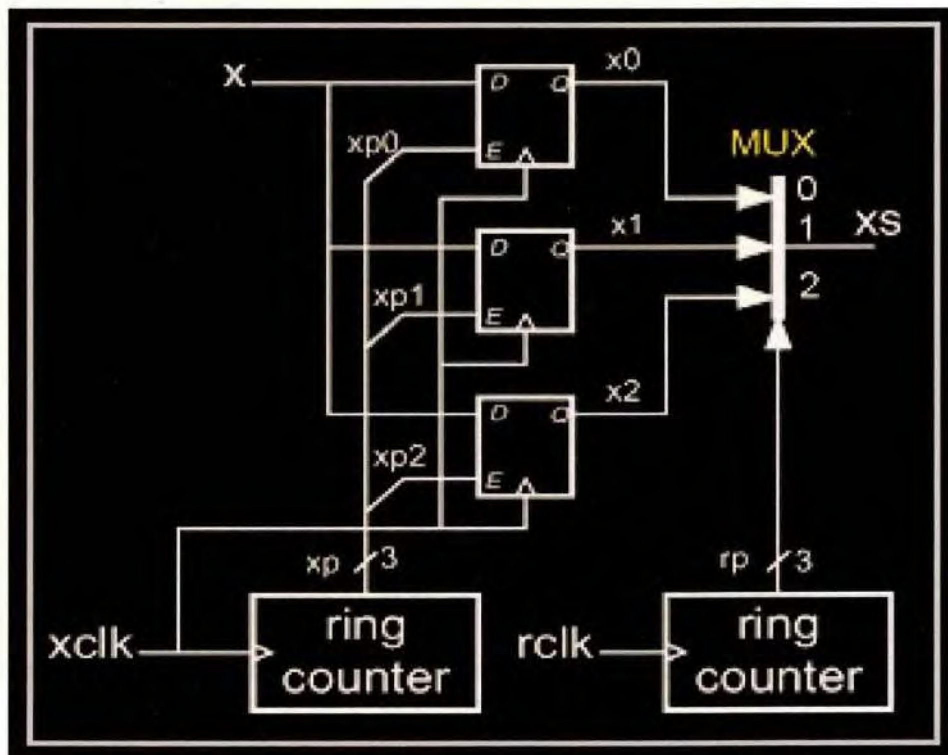



**Figure 15: FIFO synchronizer scheme**

### 2.5.2. Plesiochronous FIFO Synchronizer

Data is inserted into the FIFO buffer with the transmitter clock xclk, Data is removed from the FIFO buffer with the receiver clock rclk.
Receive pointer rp is periodically updated by synchronizing the transmit pointer xp to the receiver clock rclk, rp is replaced with the previous xp whenever the control signal resync is high.
rp is incremented normally whenever resync is low.
resync signal is driven by a controller which is designed to activate a resynchronization cycle whenever the phase difference between xclk and rclk wraps.
These updating events keep the counters in a correct phase, but lead to data dropping or replication.

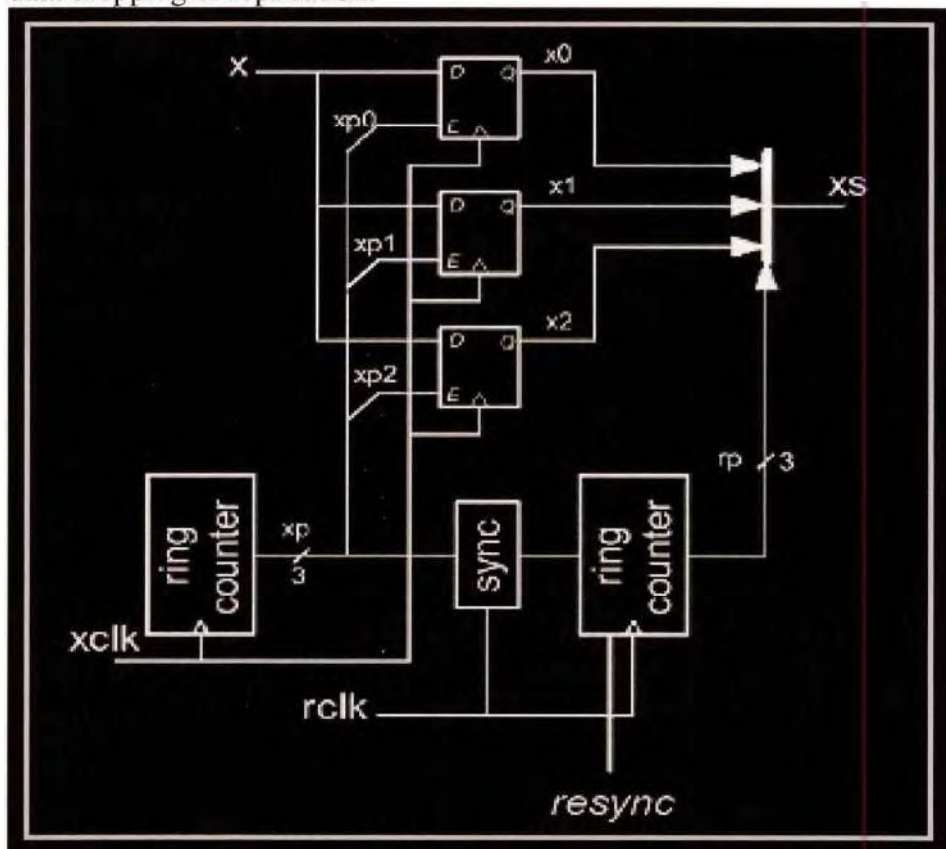



**Figure 16: Plesiochronous FIFO synchronizer scheme**

### 2.5.3. Flow Control

- Whenever the phase difference between the transmitter clock *xclk* and the receiver clock *rclk* wraps, the transmitter
  - *overruns* the receiver, if *xclk* has a *higher* frequency than *rclk*
    - a data symbol is dropped (lost)
  - *underruns* the receiver, if *xclk* has a *lower* frequency than *rclk*
    - a data symbol is replicated
- Data-rate mismatch problem is handled by inserting *null* into the data stream
  - extra signal — a *presence bit*
  - reserved bit pattern
- If the receiver is overrun, the resynchronization event is performed only when a null symbol is received
  - a null symbol is dropped, not a data symbol
- If the receiver is underrun, it inserts a null symbol instead of duplicating a data symbol after the resynchronization event
- In the *open-loop flow control*, the transmitter inserts nulls in the data stream with a frequency high enough to meet the *maximum update latency* constraint on the receiver
  - rate of actual data symbols in the stream should be less than the clock frequency of the receiver
- In the *closed-loop flow control*, the eceiver *requests* a null symbol from he transmitter when it is about to be overrun

### 2.5.4. General Purpose Asynchronous FIFO Synchronizer

Data in inserted into the FIFO buffer with the transmitter clock *xclk* keeping the control signal *shiftIn* high.
Data in removed from the FIFO buffer with the receiver clock *rclk* keeping the control signal *shiftOut* high.
No actual synchronization delay or failures in the data path.

- Provides inherent flow control via the *full* and *empty* signals
  - when the FIFO is about to be overrun, *full* is asserted, and the transmitter pauses its data sending process by setting *shiftIn* low
  - when the FIFO is about to be underrun, *empty* is asserted, and the receiver pauses reading by setting *shiftOut* low

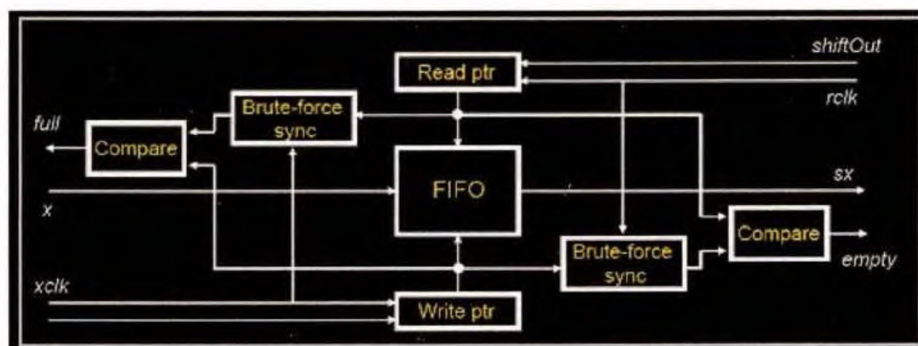



**Figure 17: General Purpose Asynchronous FIFO Synchronizer**

# 3. Out-of-Order synchronizer

## 3.1. Overview

The Out-of-Order synchronizer, synchronizes each user's request separately, independent on the previous request.
The main issue is that each request is routed to an individual slot, this is done in serial, i.e. each request is gated (AND) with an internal enable, which enables the current slot and generates the internal requests. At the next request cycle, the next slot (cyclic counting) will be enabled.
Each internal request (at each slot accordingly) will trigger the input data, and send the request as a "data valid" signal to the Out-of-Order unit, which is responsible of the serial commitment of the data.
When the data is committed by the Out-of-Order, the input may be changed at the same slot; therefore we send a "data written" signal back to the asynchronous controller, which translated it an acknowledgment to the user.
The Out-of-Order unit commits the data depending on one-hot cyclic counter which controls the data mux and enabled by the "data valid".

**Request path flow**

- o A request comes from a user
- o For each slot the request input is gated, only one controller is enabled, which will get the request.
- o The asynchronous controller translated the request, to an internal system requests.
- o The internal system request enables the data latch.
- o The internal system request enters the synchronizer in order to be synchronized to the system clock; then it translated as a "data valid" signal.
- o A "data valid" signal enables the Out-of-Order counter.

**Acknowledge path flow**

- o Starts from "data written" signal, which indicated that the data was written into the Out-of-Order.
- o Enters the asynchronous controller and translated to a user acknowledge.
- o Each acknowledge form any slot will be translated as a user acknowledge.

**Data path flow:**

- o The data starts from the data latch which is enabled by the internal request signal.
- o The stored data muxed by the Out-of-Order mux, which is controlled by the Out-of order counter.
- o Exits from the mux as the system's output.

**Figure 18: Out-Of-Order asynchronous synchronizer**

## 3.2. RTL and circuit design

### 3.2.1. Definitions package (refer to def.vhd):

#### 3.2.1.1. Functional description:

Contains all the common definition, constants, data types, function of the whole system.

#### 3.2.1.2. RTL description:

| Constant | Type | Brief description (Figure 19) |
|---|---|---|
| DBW | INTEGER | Data bus width |
| SLN | INTEGER | Slots number |
| FFN | INTEGER | N-flops sync (Brute-Force) |
| PHS | TIME | One pahse time duration (1/2 cycle) |
| RSD | INTEGER | Reset duration, in clock phases |
| SRQ | INTEGER | Start requests after SRQ clock phases |
| A1R0 | TIME | Users delay from acknowledge rise to request fall |
| A0R1 | TIME | Users delay from acknowledge fall to request rise |
| VSS | STD_LOGIC :='0' | Logical '0' |
| VCC | STD_LOGIC :='1' | Logical '1' |

**Table 2: "DEF" Constants**

| Data Type | equals to type | Brief description |
|---|---|---|
| DataBus | STD_LOGIC_VECTOR (DBW-1 DOWNTO 0) | Data bus vector |
| DataBusArr | ARRAY (NATURAL RANGE <>) OF DataBus | Data bus vectors array |
| SlotBus | STD_LOGIC_VECTOR (SLN-1 DOWNTO 0) | Slot bus vector |

**Table 3: "DEF" Data types**

| Function | Arguments | Returns | Brief description |
|---|---|---|---|
| inc_cnt | inp_cnt: SlotBus | SlotBus | Increase the cyclic counter value, equal to shift left operation |
| cnt2int | inp_cnt: SlotBus | INTEGER | One-hot value to integer convert, equal to the leading 1 index |
| OrSlot | inp_slt: SlotBus | STD_LOGIC | Logical or among all vector's bits |

**Table 4: "DEF" Functions**

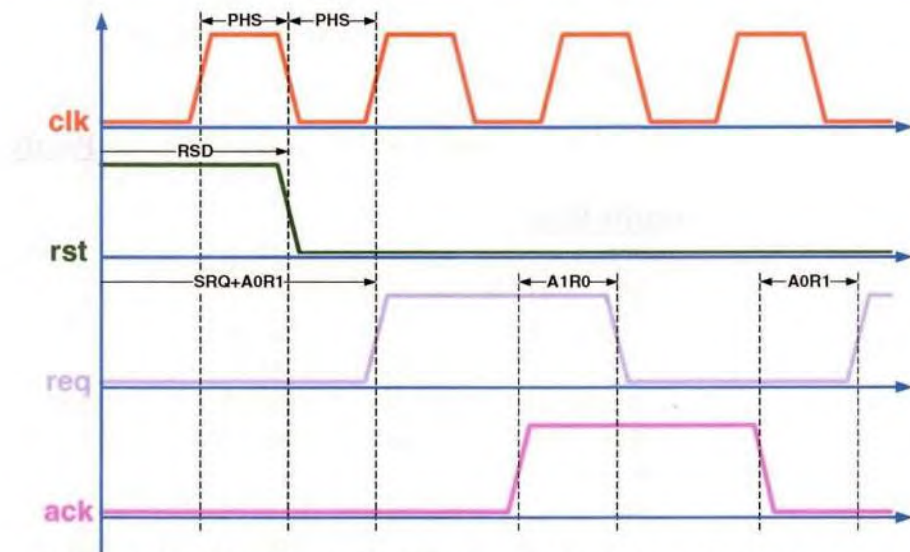



**Figure 19: Wave form for important interface signals and timing constants**

### 3.2.2. Asynchronous Controller (ac0.vhd/aci.vhd):

#### 3.2.2.1. Functional description:

This is an asynchronous block designed with Petrify, refer to the theoretical section in order to see the asynchronous design using petrify flow.

This block takes user's request (RqO), Acknowledge (AcI) and trigger from previous controller (Prv) as inputs, and produce output request (RqI), Acknowledge (AcO), trigger to the next controller (Nxt), and Request enable (REn) as outputs.

The controller makes four handshakes, with its user's input and its output (OOO), with the previous controller and the subsequent controller.

When the user send request (RqO) to the controller, that means that the input data is ready, the controller sends request to the ooo (RqI), via the n-flops synchronizer) and the rising of that request triggers the flip-flop on the data, which means that the data is locked for the ooo use.

The controller send acknowledge to the user (AcO) to inform him that the data may be changed, When the ooo finished with the data, is sends (wrt) signal to the controller which translated as acknowledge (AcI), and the controller May proceed with the next shaking cycle.

Each controller is triggered by the falling of (Prv) input. When a controller is triggered, it rises (Nxt), when it finishes, it falls (Nxt), and thus the (Prv) input of the next controller is triggered. It doesn't need any acknowledgement from the next controller because it will keep the input until it take control again, then it will rise (Nxt) signal, thus we have full control cycle (among all the controllers), we assume that this time is enough for the controllers to get the inputs, so we need no acknowledgment.

When the current controller is triggered by it'S (Prv) input from the previous control it rises (REn), request enable input which permits outer requests (RqO) to enter the controller.

**Figure 20: "AC0/ACI" – Asynchronous Controllers STG**

**Interface Protocols:**

- Handshaking with outer user:
  regular 4-phase handshaking, user sends request (RqO) which indicated ready data at input and waits to the rise of acknowledge (AcI), means that the data is latched and may be changed, then the user falls (RqO) and waits to (AcO) fall.

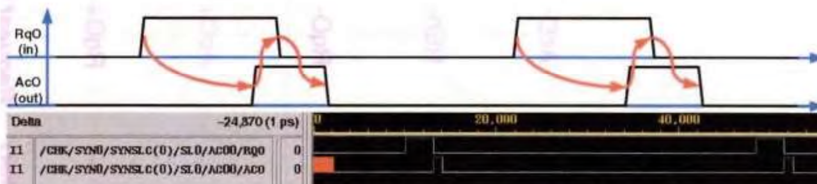



**Figure21: 4-phase handshaking with outer interface user waveform/simulation**

- Cyclic triggers:
  When the controller starts to work it send Nxt+ for the subsequent controller, when it finished, it sends Nxt-. Each controller waits to the subsequent signals Prv+ then Prv- in order to take control. It doesn't need any acknowledgement from the next controller because it will keep the input until it take control again, then it will rise (Nxt) signal, thus we have full control cycle (among all the controllers), we assume that this time is enough for the controllers to get the inputs, so we need no acknowledgment.

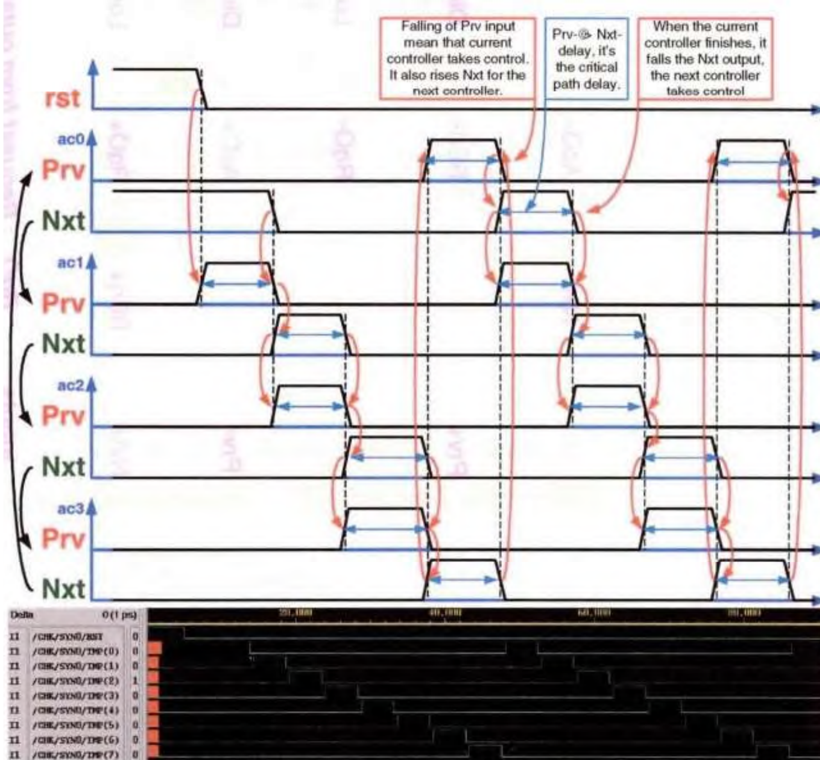



**Figure 22: Cyclic trigger protocol waveform/simulation**

- Gating requests:
  When the current controller is triggered by it'S (Prv) input from the previous control it rises (REn), request enable input which permits outer requests (RqO) to enter the controller.

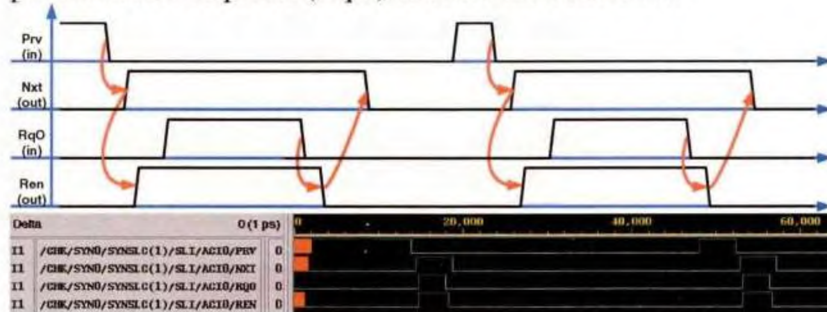



**Figure 23: (REn), gating requests signal, with (Prv) and (RqO) waveform/simulation**

- Latching data:
  the controller rises (Lod) signal to enable the latch, (Din) is an input to the controller which equals to Lod after "delay ( $d_{buf}$ )" and means that the data is inside, the "delay" is a time period needed to latch the data.
  the protocol is 4-phase, thus when (Din) goes high, (Lod) drives '0', and wait to the fall of (Din)

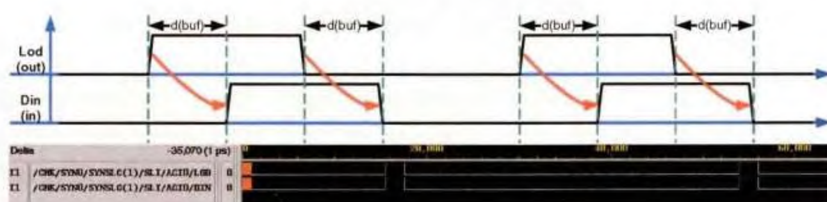



**Figure 24: latching signals waveform/simulation**

- Handshaking with inner ooo unit:
  The asynchronous controller drivers request (RqI) high to the ooo unit which says that the data is latched and may be used, this signal should be synchronized (we use n-flops synchronizer) since it goes from asynchronous domain to synchronous (ooo), when the ooo finished with the latched data, it sends acknowledge (AcI) high back to the controller, the protocol is 4 phase, so after we get (AcI) low, (RqI) goes low, synchronized, and (AcI) goes low.

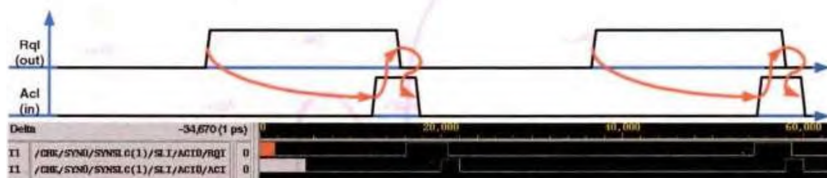



**Figure 25: RqI/AcI 4-phase protocol/simulation**

**Controller critical path:**
the critical path of the asynchronous controller start when the controller takes control from the previous controller, when Prv falls, and ends when the current controller gives control to the next controller, when Nxt falls. It contains the data latching controls, because we must wait to the data latching before proceeding to the next data, besides the 4-phase handshaking with the user (interface), and the cyclic trigger of previous/next controller. At the following table, you may see each transition delay, it's clear that the arc Din+→Lod- is the delay heavy consumer.

| arc | Delay (ns) |
|---|---|
| Prv-→Nxt+ | 0.54 |
| Nxt+→REN+ | 0.4 |
| REn+→RqO+ | 0 |
| RqO+→Lod+ | 0.77 |
| Lod+→Din+ | negligible |
| Din+→Lod- | 2.02 |
| Lod-→AcO+ | 0.31 |
| AcO+→RqO- | negligible |
| RqO-→REn- | 0.39 |
| REn-→Nxt- | 0.38 |
| Total | 4.81 |

**Table 5: Critical Path delay by arcs**

**Could we improve the critical path more?** From the STG point of view, it seems to be that we have the most optimized STG, at the beginning we tried handshaking 4-phase protocol to move control between controllers, but this consumes too much delay, so we moved to the new cyclic trigger as shown above, more ideas to boost the STG may arise, like parallelize Nxt+ and Ren+ (add Prv-→REn+, delete Nxt+→REn+, add Nxt+→REn-).
Using faster technology than the one we have (0.35um) would make significant improve, we used "Petrify" to synthesize the asynchronous controllers, the circuit may be more optimized with better synthesizer or relative user constrains.
From table 5 above the arc Din+→Lod- is the heaviest delay consumer, so it's worth to try improving this transition.

**Figure 26: "AC0/ACI" – Asynchronous Controllers STG – critical path**

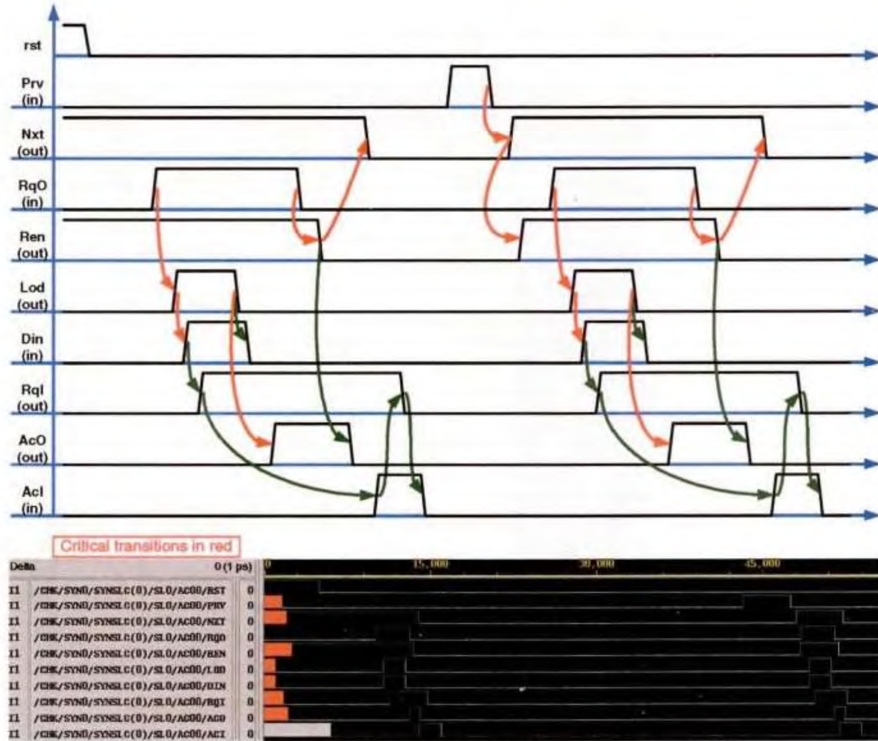



**Figure 27: "AC0" – critical transmissions on "AC0" - waveform/simulation**

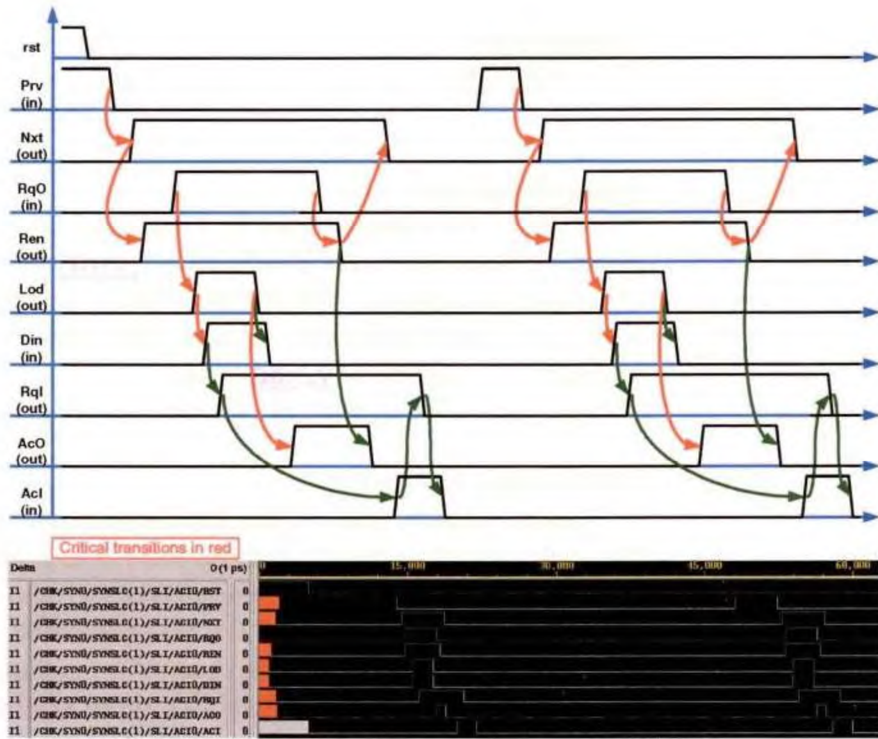



**Figure 28: "ACI" – critical transmissions on "ACI" - waveform/simulation**

#### 3.2.2.2. RTL description:

| Signal | Direction | Type | Brief description |
|---|---|---|---|
| Rst | Input | STD_LOGIC | System global reset |
| RqO | Input | STD_LOGIC | Request from Output (user/interface) |
| Prv | Input | STD_LOGIC | Trigger from previous controller |
| Din | Input | STD_LOGIC | The data is latched |
| AcI | Input | STD_LOGIC | Acknowledge from Input (OOO) |
| AcO | Output | STD_LOGIC | Acknowledge to Output (user/interface) |
| Nxt | Output | STD_LOGIC | Trigger to the next controller |
| Lod | Output | STD_LOGIC | Load latch with data |
| RqI | Output | STD_LOGIC | Request to Input (OOO) |
| REn | Output | STD_LOGIC | Request enable |

**Table 6: "AC0/ACI" Interface signals**

**Internal signals:**
Internal signals/mapping/states generated by petrify.

**Components:**
CSX_HRDLIB components, 0.35um library.

**The architecture:**
The RTL architecture of this block contains only the mapping of a library cells as they were generated by petrify by using –tm and –lib option for mapping library cells, Petrify generated a Verilog file with reset using the options –vl and –rst1, The Verilog file was translated to vhdl using v2vhd script.
Please refer to the section which talks about using Petrify.

#### 3.2.2.3. Circuit description:

No Special implementation, just the library cells.Please note the "set_dont_touch" attribute on those cells in at syntheses in order to have the same design after syntheses (no hazards).
SDF "Standard Delay Format" file was generated "make_sdf" in order to have the controller delays modeled at simulation.
Please refer to the section which talks about syntheses.

### 3.2.3. N flip-flops (nff.vhd):

#### 3.2.3.1. Functional description:

Serial rising edge flip-flops with reset.
This block behaves as n-flops synchronizer, it's the basic synchronization unit at our design; you may read about its synchronization function at the theoretical part.
It's simply built out of serial flip-flops; each flip-flop adds one more cycle of latency on the output.
We intend to synchronize the Request signal (RqO), with the system global clock, in order to inform the ooo unit that the data is ready to be read.

#### 3.2.3.2. RTL description:

| Signal | Direction | Type | Brief description |
|---|---|---|---|
| clk | Input | STD_LOGIC | System global clock |
| rst | Input | STD_LOGIC | System global reset |
| inp | Input | STD_LOGIC | Input to serial flip-flops |
| otp | Output | STD_LOGIC | Output from serial flip-flops |

**Table 7: "NFF" Interface signals**

| Signal | Type | Brief description |
|---|---|---|
| fft | STD_LOGIC_VECTOR (FFN-1 DOWNTO 0) | Temporal signal; holds the internal value ofd the serials ff's |

**Table 8: "NFF" Internal Signals**

**The architecture:**
The flip-flop is sensitive to rest and clock signals, so we put then at its process sensitivity list.
On reset mode, each flip-flop gets zero as output (fft=0).
When the block exits the reset mode then it functions on each rising edge clock, the first flip-flop of the chain gets the input, the last flip-flop on the chain generates the output, and the other flip-flops at the middle, each one get the output of it's previous flip-flip as inputs and drives the input of the next flip-flop on the chain.

#### 3.2.3.3. Circuit description:

No special implementation, it's implemented as a serial flip-flops chain with shared clock and reset.

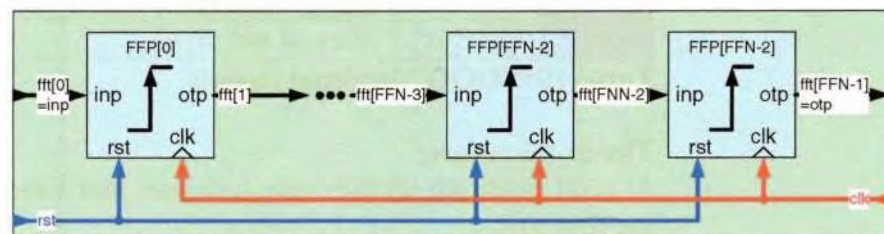



**Figure 29: NFF – N Flip-Flops Synchronizer**

### 3.2.4. Out Of Order (ooo.vhd):

#### 3.2.4.1. Functional description:

The purpose of this unit is to re-arrange all the synchronized data; it's a synchronous unit, which have a clock input.
The design has one-hot cyclic counter (cnt) which points to the current data to commit (inp[cnt]), when that data is ready to commit, it is muxed out, and the counter is increased to point to the next data (slot), on the rising clock.
The data is ready to commit when the input to the current slot we are pointing at is valid, that's to say: cnt(slt)=1 and vdi(slt)=1, (vdi) indicated that the input data is valid at a specific slot.
The system gives also a feed-back to the inputs to indicate that the data has been written and you may change your data (wrt), this is equal the writing enable for each slot: cnt(slt)=1 and vdi(slt)=1.
The (ovd) output valid signal, indicates that the data on the output bus is valid, it sets up when cnt(slt)=1 and vdi(slt)=1 for any slot (OR on all the slots), it's also the same indicator for increasing the counter.
Each data is valid for a one cycle, it's sure that the data is stable due that cycle due to the synchronization operation which takes more than one cycle.

#### 3.2.4.2. RTL description:

| Signal | Direction | Type | Brief description |
|---|---|---|---|
| clk | Input | STD_LOGIC | System global clock |
| rst | Input | STD_LOGIC | System global reset |
| inp | Input | DataBusArr(SLN-1 DOWNTO 0) | Input data to OOO, SLN-1 data busses |
| vdi | Input | SlotBus | Valid input,for each slot. |
| wrt | Output | SlotBus | Written? Indicates that the data was written, for each slot. |
| otp | Output | DataBus | Output data to OOO |
| ovd | Output | STD_LOGIC | Output valid? |

**Table 9: "OOO" Interface signals**

| Signal | Type | Brief description |
|---|---|---|
| cnt | SlotBus | One-hot cyclic counter, points to the current data to commit |
| wrti | SlotBus | Internal wrt signal |

**Table 10: "OOO" Internal signals**

**The architecture:**
At reset mode we set the counter to one, that's mean, its points to the first slot.
The sequential logic of this unit is the counter, it increases if we have (wrt(slt)=vdi(slt) and cnt(slt)) for any slot, that's mean the

current pointed slot is valid.
The combinatorial logic of this part includes the (wrt(slt)) generation, which indicates that the current slot valid data has been written and the user may change the input data on that slot.
The combinatorial logic also contains the muxing of the input data, it's a pass-gates passed mux, which the control must be strongly mutex, like the one-hot counter at our case.
We have (ovd) – output valid, if any slot has a valid data which has been written out, in other words, any slot have (wrt) sat on, therefore that signal is logically or for (wrt(slt)) for each slot.

#### 3.2.4.3. Circuit description:

The one-hot cyclic counter is implemented from rising edge serial cyclic flip-flops.
Only the first flip-flop have a "set" input, all the others have "reset" inputs, thus the counter will be initialized to one.
Each clock, if the counter is enabled, the counter content is shifted right cyclic, and this implements one-hot cyclic counter.
The counter output controls one big pass-gate based mux, the input to that mux should be strongly mutex.
(wrt(slt)) is implemented for each slot with logical AND between (vdi(slt)), valid input, and the counting bit value for this slot.
The counter enable, and the (ovd), out valid, which have the same logical value, are implemented with a logical OR among all the (wrt(slt)) for each slot.

#### 3.2.4.4. Bit-Slice implementation:

The above implementation is suitable for bit slices, but the first slice will differ, because it has a flip-flop with set instead of reset at the other flip-flops.
The wide SLN-1 inputs OR gate can be replaced with SLN-1 OR gates, each with two inputs, which have serial connection one gate output to the next gate input (serial), the first OR must be connected to Zero on one of it's inputs.
This implementation may be a delay consumer due to the serial OR gates.

**Figure 30: OOO – Out Of Order Unit**

Figure 31: OOO – Out Of Order Unit – Bit-Slice Design

### 3.2.5. Synchronizer (syn.vhd)

#### 3.2.5.1.Functional description:

Top hierarchy of our Out-of-Order Synchronizer, includes the above blocks as components (please see the components list below also).

#### 3.2.5.2. RTL description:

| Signal | Direction | Type | Brief description |
|---|---|---|---|
| clk | input | STD_LOGIC | System global clock |
| rst | input | STD_LOGIC | System global reset |
| req | input | STD_LOGIC | Request, mean that user put data on input bus and requests to write it |
| ack | output | STD_LOGIC | Acknowledge; mean that the data has been written |
| inp | input | DataBus | Input Bus |
| otp | output | DataBus | Output Bus, the synchronized data |
| ovd | output | STD_LOGIC | Output valid? Indicates that the output is valid |

**Table 11: "SYN" Interface signals**

| Signal | Type | Brief description |
|---|---|---|
| vdi | SlotBus | Internal valid |
| ini | DataBusArr (SLN-1 DOWNTO 0) | Internal input |
| Aci | SlotBus | Internal acknowledge |
| Rqi | SlotBus | Internal request |
| wrt | SlotBus | Indicates that the current slot has been written to the OOO. |
| RqO | SlotBus | Request, output of act, triggers the data ff and enters the n-flops sync. |

**Table 12: "SYN" Internal signals**

| Component | Brief description |
|---|---|
| RRA | Round Request Acknowledge |
| AC0 | Asynchronous Controller 0 |
| ACi | Asynchronous Controller i |
| LCH | Latch |
| NFF | n flip-flop synchronizer |
| OOO | Out-of-Order |

**Table 13: "SYN" Components**

**The architecture:**

The systems gets request from the user, and valid data in the input bus, request is enabled (AND gate) at each slot, only one slot enables the request and sends tit to the suitable AC0/ACi (Asynchronous Controller), the AC0/ACi send request the OOO (Out-of-Order) via the n-flops synchronizer, and latches the data at the input latch, the OOO commits the suitable data and send Acknowledge back to the AC0/Aci which send it back to the user.

3.2.5.3. **Circuit description:**

The circuit contains only mapping and connection to other components.

3.2.5.4. **Bit-Slice Implementation:**

This circuit may be implemented as bit-slice design, because each component may be implemented as bit-slice design.
We mentioned before how to implement OOO unis as bit-slice design (the slice will be differ than the others).
The AC0/Sci, LCH and NFF appear once for each slice.

**Figure 32: SYN – Synchronizer**

### 3.2.6. Test-Bench (tbn.vhd):

#### 3.2.6.1. Functional description:

A test-bench is a virtual user. It simulates the functionality of the user and checks the correctness of the results given the checked system depending on the inputs it delivers to the checked system. Our test-bench produced the clock and reset signals, depending on the user parameters. It also produced the request signals together with the input data; it waits to acknowledge form the user in order to make another request. When an "ovd – output valid" signal is detected it checks the output bus (of the synchronized data) and determined whether that results are correct.

#### 3.2.6.2. RTL description:

| Signal | Direction | Type | Brief description |
|---|---|---|---|
| Clk | input | STD_LOGIC | System global clock, initialized to '1' |
| Rst | input | STD_LOGIC | System global reset, initialized to '0' |
| Req | input | STD_LOGIC | Request, mean that test-bench put data on input bus and requests to write it, initialized to '0' |
| Ack | output | STD_LOGIC | Acknowledge, mean that the data has been written |
| Inp | input | DataBus | Input bus generated by test-bench, initialized to 0 |
| Otp | output | DataBus | Output bus, enters test-bench in order to be checked |
| Ovd | output | STD_LOGIC | Output valid? Indicates tha output bus is valid |

**Table 14: "TBN" Interface signals**

| Signal | Type | Brief description |
|---|---|---|
| Clki | STD_LOGIC | Internal clock, initialized to '0' |
| Rsti | STD_LOGIC | Internal reset, initialized to '1' |
| Inpi | DataBus | Internal input bus, initialized to 0 |
| Pot | DataBus | Previous output, temp. signal for checking output, initialized to 0 |
| Ppo | DataBus | Previous of previous output, temp. signal for checking output, initialized to -1 |
| Cct | INTEGER | Clock counter, counts the clock phases , initialized to 0 |

**Table 15: "TBN" Internal signals**

**The architecture:**

Clock generation process:

The process runs in an infinite loop, every iteration it waits for PHS (one phase time) then it inverts the clock and increases the clock counter (cct).

Reset generation process:

The reset initialized with 1, that's mean that the systems starts at reset mode.

After RSD (reset duration) clock phases, the reset goes 0, that's mean the system exits the reset mode.

Input generation process:

The process runs in an infinite loop, the data initialized to zero, it starts the first request after SRQ (start request time) clock phases, then it waits for acknowledge rise.

When the acknowledge rises it wait's for A1R0 then brings down the request (req <= '1' after A0R1). When the acknowledge gets down it increases the data by 1 and waits for A0R1 before rising the request again for the next iteration (req <= '0' after A1R0).

"Generate previous output" process:

This process produced the previous output "pot" and the previous of the previous of the previous output, in order to compare it with the current output.

It's actually passes the output through two flip-flops, if the output is valid (ovd).

"Check output" process:

Here we check that the previous output "pot" equals to the previous of the previous output +1 "ppo+1" when "pot" changes (ASSERT (pot=(ppo+1)) report "Error in results").

Because the test-bench applies the system with sequence of serial numbers, it should get also the same sequence synchronized to the clock at the same order.

3.2.6.3. **Circuit description:**

This block is not synthesizable; it's intended only for testing, not for implementation.

### 3.2.7. Checker (chk.vhd):

#### 3.2.7.1. Functional description:

This block connects out design top hierarchy – TOP, with the test-bench – TBN.
For wave-form simulation we form the internal signals between the two blocks.

#### 3.2.7.2. RTL description:

**Interface signals:**
No Interface signal, because we may sample the internal signals between the blocks.

| Signal | Type | Brief description |
|---|---|---|
| clk | STD_LOGIC | System global clock |
| rst | STD_LOGIC | System global reset |
| req | STD_LOGIC | Request, mean that user put data on input bus and requests to write it |
| ack | STD_LOGIC | Acknowledge; mean that the data has been written |
| inp | DataBus | Input Bus |
| otp | DataBus | Output Bus, the synchronized data |
| ovd | STD_LOGIC | Output valid? Indicates that the output is valid |

**Table 16: "CHK" Internal Signals**

| Component | Brief description |
|---|---|
| SYN | Synchronizer – our design top hierarchy |
| TBN | Test-bench – designed toimulate the user |

**Table 17: "CHK" Components**

**The architecture:**
Only components mapping for connecting between the SYN and TBN.

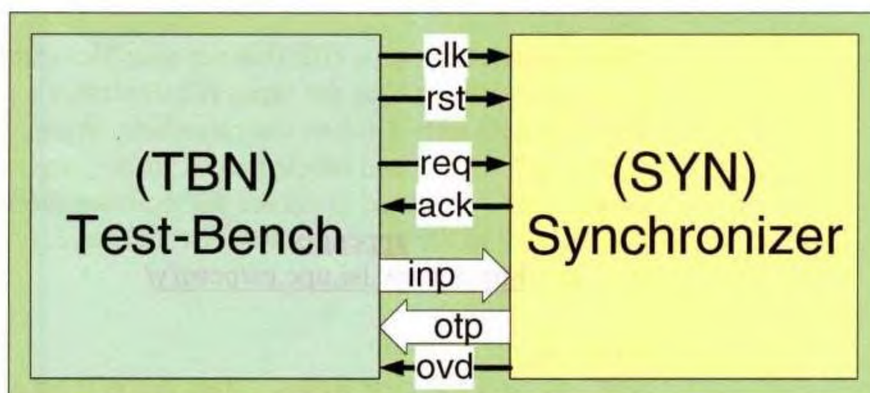



**Figure 33: CHK - Checker**

## 3.3. Implementation

### 3.3.1. Platform

The system was designed, synthesized (synchronous and asynchronous), and simulated at Sun Solaris OS, at the VLSI systems research and laboratories EE, Technion.

**Technology Process and cells library:**
The circuits were implemented in 0.35 CSX CMOS technology at 3.3V.

### 3.3.2. Tools

#### 3.3.2.1. Asynchronous Syntheses tools

**Petrify**
*Petrify* is a tool for synthesis of Petri nets and asynchronous controllers. *Petrify* reads a Petri net and generates another bisimilar Petri net which is simpler than the original description. Initially, *petrify* performs a token flow analysis of the initial Petri net and produces a transition system (TS). In the initial TS, all transitions with the same label are considered as one event. The TS is then transformed and transitions relabeled to fulfill the conditions required to obtain a Petri net. *Petrify* is able to obtain Petri nets with some specific properties: pure, free choice, unique choice, place irredundant, etc. The Petri nets accepted by *petrify* can also be interpreted as Signal Transition Graphs describing the behavior of asynchronous controllers. *Petrify* is able to solve the Complete State Coding problem and generate a speed-independent circuit. Petrify also includes another application called *draw_astg* to draw Signal Transition Graphs in several graphic formats.
Please refer to the appendix for the tool usage, or visit the following web site: http://www.lsi.upc.es/petrify/

**draw_astg**
*draw_astg* is a tool that draws a Petri net from a description is astg format (SIS compatible). It can generate descriptions in different formats: PostScript, MIF (FrameMaker graphics), HPGL (HP pen plotters),
PCL (Laserjet printers, GIF (bitmap graphics) and DOT (graph format for dot). In case the input file describes a state graph or a Petri net with only 1-token state machine, *draw_astg* can depict a state graph (nodes and labeled arcs). *draw_astg* calls dot, a preprocessor designed at AT&T for drawing directed graphs.
Please refer to the appendix for the tool usage, or visit the following web site: http://www.lsi.upc.es/petrify/

**write_sg**
write_*sg* is a tool that writes a state graph from a description is the astg format used by petrify.
Please refer to the appendix for the tool usage, or visit the following web site: http://www.lsi.upc.es/petrify/

### 3.3.2.2. Synchronous design tools

**v2vhd**
A Perl scrip for converting VERILOG RTL code into VHDL, we need such that script because petrify output is VERILOG but the whole system RTL is designed with VHDL.

**vhdlan**
*vhdlan* is the command to compile the VHDL source program and generate the simulation-ready files *.sim* and others in the "*work*" directory. Analyze/compile means to translate VHDL source program into another representation that can be simulated and analyzed.

**da, Design Analyzer**
Synopsys Design Analyzer is a widely used Logic Synthesis and Optimization tool. Logic synthesis translates textual circuit descriptions like VERILOG or VHDL into gate-level representations.
Optimization minimizes the area of the synthesized design and improves the design's performance.
The HDL description can be synthesized into a gate-level net-list composed of instances of the standard cells.
Actually we use the Design Analyzer for generating SDF (Standard delay Format) for the asynchronous controllers in order to simulate them afterward.
http://www.synopsys.com/products/logic/design_comp_ds.html

**Synopsys VirSim® VHDL Simulator**
VirSim® is the highest performance, highest capacity full-language VHDL simulator. It is the only VHDL simulator designed specifically to address the challenges of SoC verification. VirSim's unique architecture combines cycle-based, performance-driven optimization techniques with the flexibility of event-driven simulation to deliver superior performance and capacity for complete system verification. Scirocco and VCS work together to provide the same high performance and high capacity for mixed-HDL simulation.
http://www.synopsys.com/products/simulation/scirocco/scirocco_ds.html

## 3.4. System Verification and Simulation

### 3.4.1. Verification

The system is not formally verified, but the current verification is satisfying to cover all the corners on the system.
The system is fed with serial input data with request on each data change and waits for acknowledge from the system and before changing the data, the system assumed to produce the same serial data on the output.
For that purpose a test-bench is used. A test-bench is a virtual user. It simulates the functionality of the user and checks the correctness of the results given the checked system depending on the inputs it delivers to the checked system.
Our test-bench produced the clock and reset signals, depending on the user parameters. It also produced the request signals together with the input data; it waits to acknowledge form the user in order to make another request.
When an "ovd – output valid" signal is detected it checks the output bus (of the synchronized data) and determined whether that results are correct.
The special line:

```
ASSERT (pot=(ppo+1)) report "Error in results";
```

will report "Error in results" when the output is wrong.

### 3.4.2. Simulation

The system was simulated Synopsys VirSim® VHDL Simulator, VerSim® is the graphical cockpit front-end to both the Scirocco® VHDL and VCS® Verilog HDL simulators. input signals produced was a test-bench, that test-bench checks also the correctness of the output.
Please refer to the previous section to see how the simulator is activated.
The following are some screenshot from the system simulation.

**Figure 34: Clk @ 1.23GHZ (max freq)**

**Figure 35: Clk @ 100MHZ**

**Figure 36: Clk @ 10MHZ**

## 4. Results, conclusions and further work

In order to evaluate the performance of the OOO synchronizer design presented in this paper, the circuits were implemented in 0.35 CSX CMOS technology and simulated at 3.3V using Synopsys VirSim® VHDL Simulator.
We have simulated the behavior of an OOO synchronizer of capacity 8 (8 slots), with a data item of width 8 (data bus with 8).

- **Measured parameters, depends at implementation technology:**
  These parameters were measured with incremental simulation, thus simulations were run with various incremental values of the measured parameter until a fault appears, then the maximum parameter is determined.
  - **Handshaking delay from the synchronous domain side:**
    Handshaking delay is defined as the delay needed to repeat the handshaking protocol, thus the delay form request rising till acknowledge falling. This period of time depends on the asynchronous controllers design and directly on the critical path on these controllers as shown in the relevant section above. As simulated at VirSim, that maximum handshaking delay is: 4.8ns.
  - **Maximum clock frequency from the synchronous domain side:**
    The system functions reliably up to the local clock frequency of 1.23GHz (according to VirSim simulation) - the maximum clock rate is limited by the ring oscillator, not the plausible clocking control, because if the clock controls (enable in our case) has late rising or late enable, the ring counter will stay disabled until the enable is raised, it case it has late falling or late disable, the counter value will increased, but the output valid will be false, waiting to the correct data.
- **Architectural parameters:**
  These parameters have nothing to do with the technology process or the implementation; they depend on the basic design of the system.
  A comparison with general asynchronous FIFO synchronizer is shown beneath.
  - **Data latency from input to output:**
    In order to analyze the data throughput, depending on the amount of data in OOO/FIFO slots, two cases should be analyzed:
    - Amount of data is less or even than "almost full":
      Each input data accompanied with writing request from the asynchronous domain, the writing request must be synchronized, 2-flops synchronizer is used with latency of 2..3 cycles, for n-flops synchronizer we have n..n+1 cycles latency. When the writing request is synchronized, the latched data is declared as valid at the synchronous domain, the latched data has 0 cycles to be passed out (the output mux, combinatorial logic).
      Conclusion: OOO synchronizer has n..n+1 input to output latency for the n-flops synchronizer implementation.
      FIFO comparison: the FIFO synchronizer has the same latency with the n-flops synchronizer implementation.
    - Amount of data is more than "almost full":
      When all slots are full we must wait to catch an empty one for our data, the same scenario happened at FIFO synchronizer,

this may harm the data latency.
Conclusion: latency can't be determined in this case
FIFO comparison: FIFO synchronizer acts the same.

- **Output data throughput:**
  In order to analyze the data throughput, depending on the amount of data in OOO/FIFO slots, two cases should be analyzed:
  - At the OOO synchronizer, at "almost empty" region we shouldn't wait for the first synchronizer to complete synchronization, we may proceed with the next data, so the throughput is equal to the input rate.
    Conclusion: OOO synchronizer has throughput which is equals to the input data rate.
    FIFO comparison: At "almost empty" region, the FIFO throughput is n..n+1 cycles (for n-flops synchronizer) because the reading pointer won't proceed until the pointer is synchronized and compared to the current pointer.
    When we have an input data rate of 1 input data/1 cycle or slower rate, the output rate will 1 output data/n..n+1 cycles for the FIFO synchronizer, but it will be equal to the input data rate at the OOO synchronizer.
  - Amount of data is more or even than "almost empty":
    the throughput in this case is 1, because the data is ready and latched for the synchronous domain use, each read takes one cycle.
    Conclusion: OOO synchronizer has throughput of 1 each cycle, when the amount of the latched data is more or even than "almost empty".
    FIFO comparison: FIFO synchronizer acts the same.
- **Input data limitation:**
  In order to analyze the data throughput, depending on the amount of data in OOO/FIFO slots, two cases should be analyzed:
  - Amount of data is less or even than "almost full":
    At both OOO & FIFO, no limitation at input data rate because we have empty slots to fill.
  - Amount of data is more than "almost full":
    Both OOO and FIFO wait to have empty slot, OOO send writing acknowledge to indicate that the data in the latch may be rewritten, FIFO compares the current writing pointer to the reading pointer to detect if there's any empty slot, thus the input data rate will be limited to 1 data/1 cycle.

**Area comparison OOO and FIFO architectures:**
Area results need syntheses of the full design and are depending on the logic size, so I prefer to make block by block comparison between FIFO and OOO.

| n slots*m bits input bus FIFO | n slots*m bits input bus OOO |
|---|---|
| n*m wide latching elements | n*m wide FIFO memory elements |
| n brute-force synchronizers | n brute-force synchronizers |
| n-bits wide cyclic one-hot counter | n-bits wide writing pointer |
| n-buses (of m-bits) wide output mux | n-buses (of m-bits) wide output mux |
| 2*n two inputs AND gates<br>2 n-input OR gate | 2 n-bit comparators |
| n asynchronous controllers | n-buses (of m-bits) wide input mux<br>n-bits wide reading pointer |

**Table 18: OOO/FIFO Area comparison**

At the table 17 above, blocks at the left column are almost equal to the blocks at the right column, except of the asynchronous controllers which may be different. We may conclude that FIFO & OOO have areas from the same order.

**When the OOO synchronizer is better that FIFO?**
The main difference between the two architectures is that the FIFO synchronizer synchronizes the reading pointer, but the OOO synchronizes the requests.
At "almost empty" region, the FIFO throughput is n..n+1 cycles (for n-flops synchronizer) because the reading pointer won't proceed until the pointer is synchronized and compared to the current pointer.
At the OOO synchronize, at "almost empty" region we shouldn't wait for the first synchronizer to complete synchronization, we may proceed with the next data, so the throughput is equal to the input rate. (We still have n..n+1 latency)
When we have an input data rate of 1 input data/1 cycle or slower rate, the output rate will be 1 output data/n..n+1 cycles for the FIFO synchronizer, but it will be equal to the input data rate at the OOO synchronizer.
At "almost full" region both OOO and FIFO has a data at slots to commit each cycle.
Regarding input data rate limitation, both OOO and FIFO wait to have empty slot, OOO send writing acknowledge to indicate that the data in the latch may be rewritten, FIFO compares the current writing pointer to the reading pointer to detect if there's any empty slot, thus the input data rate will be limited to 1 data/1 cycle.

| Parameter | Value | Units |
|---|---|---|
| Handshake delay (request rise to acknowledge fall) | 4.8 | nsec |
| Maximum frequency | 1.23 | GHZ |
| Data latency | 2-3 | cycles |
| Data throughput | 1 | cycles |

**Table 19: Results**

## 5. Conclusions and further work

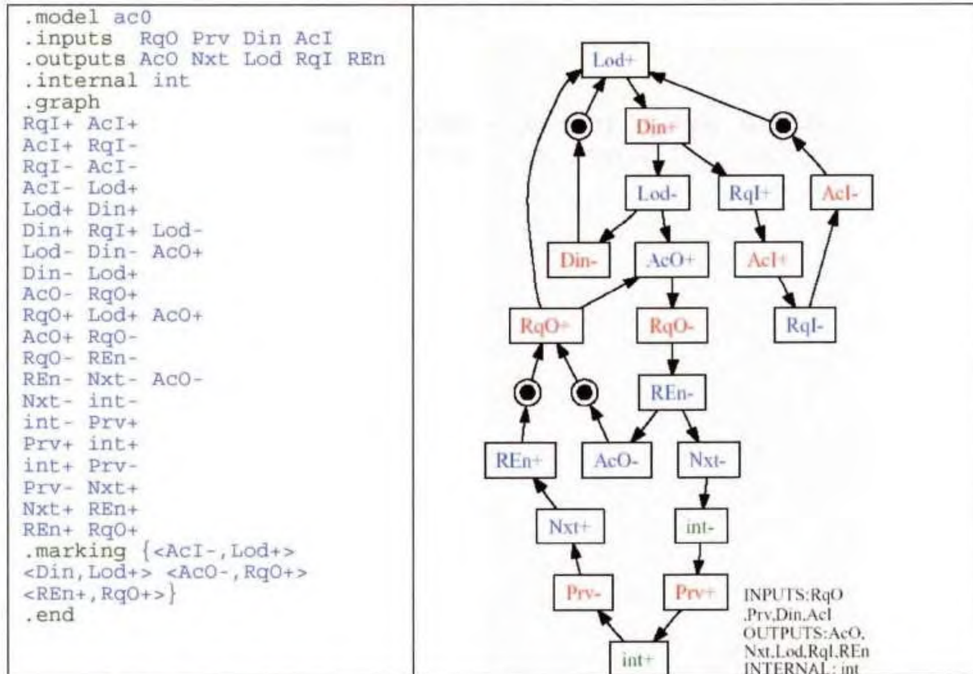
This paper has presented a new low-latency/high-throughput out-of-order synchronizer design which interfaces between asynchronous and synchronous subsystems.
The design is based on the idea of token passing, only the request asynchronous signal is synchronized. The control token shouldn't wait to complete the request synchronization before passing to the next controller, the input data is latched and then control token is passed to the next controller (cyclic mode).
In order to prove its feasibility, we constructed a simulation bed consisting of test-bench with asynchronous and synchronous modules, connected to the out-of-order synchronizer, all implemented on 0.35u CSX 3.3v CMOS technology and simulated with Synopsys VirSim® VHDL Simulator.
The resulting system functions reliably up handshaking data rate of xxx at the asynchronous domain it's limited by the controllers inner implementation (Petrify) and clock frequency of 220MHz at synchronous domain, the maximum clock rate is limited by the ring counter.
The system has the same architectural parameters as the FIFO synchronization system besides the high synchronization parallelism which keeps low writing latency by preventing deadlock on full slots system.
In the future, the asynchronous controllers may be optimized in order to gain low input interface handshaking delay, for large slots number, a wide input or gates shown on the design, they may b implemented with static CMOS logic by nor/nand tree, or with dynamic logic domino gates.
The output mux width also depends on the slots number, so we should thinks of another implementation in place the one-hot passgate mux, like a fully CMOS static mux.

# 6. Appendix

## 6.1. Asynchronous Design Flow

- **The STG design was written into .g files (ac0.g and aci.g) as petrify inputs.**
- **Drawn to .ps files (ac0.g.ps and aci.g.ps) with draw_astg:**

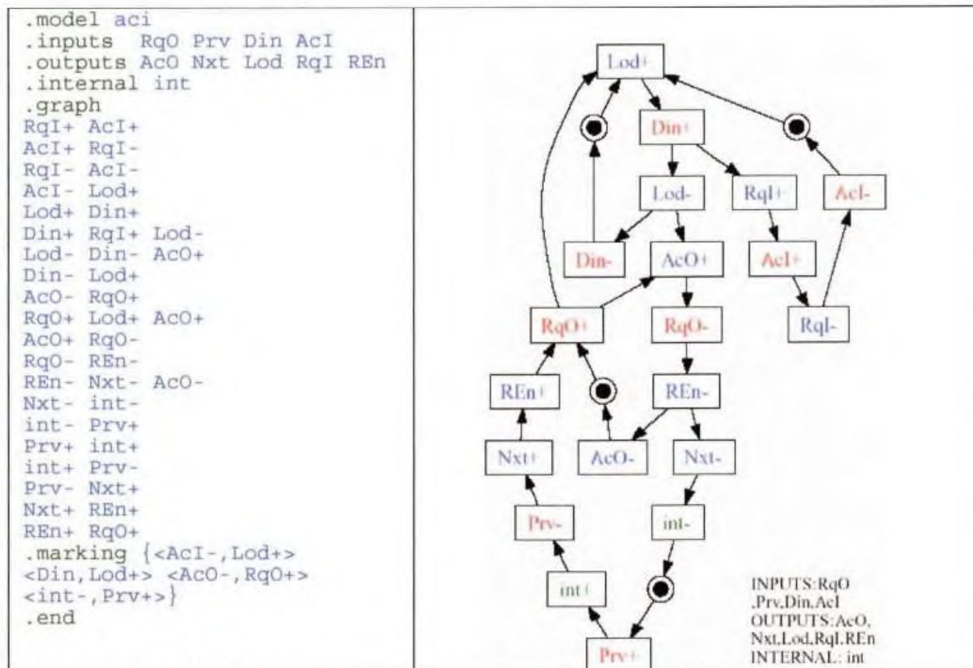
```
>draw_astg ac0.g > ac0.g.ps
>draw_astg aci.g > aci.g.ps
```



```
.model ac0
.inputs  RqO Prv Din AcI
.outputs AcO Nxt Lod RqI REn
.internal int
.graph
RqI+ AcI+
AcI+ RqI-
RqI- AcI-
AcI- Lod+
Lod+ Din+
Din+ RqI+ Lod-
Lod- Din- AcO+
Din- Lod+
AcO- RqO+
RqO+ Lod+ AcO+
AcO+ RqO-
RqO- REn-
REn- Nxt- AcO-
Nxt- int-
int- Prv+
Prv+ int+
int+ Prv-
Prv- Nxt+
Nxt+ REn+
REn+ RqO+
.marking {<AcI-,Lod+>
<Din,Lod+> <AcO-,RqO+>
<REn+,RqO+>}
.end
```



**Figure 37: ac0.g/ac0.g.ps**



```
.model aci
.inputs  RqO Prv Din AcI
.outputs AcO Nxt Lod RqI REn
.internal int
.graph
RqI+ AcI+
AcI+ RqI-
RqI- AcI-
AcI- Lod+
Lod+ Din+
Din+ RqI+ Lod-
Lod- Din- AcO+
Din- Lod+
AcO- RqO+
RqO+ Lod+ AcO+
AcO+ RqO-
RqO- REn-
REn- Nxt- AcO-
Nxt- int-
int- Prv+
Prv+ int+
int+ Prv-
Prv- Nxt+
Nxt+ REn+
REn+ RqO+
.marking {<AcI-,Lod+>
<Din,Lod+> <AcO-,RqO+>
<int-,Prv+>}
.end
```



**Figure 38: aci.g/aci.g.ps**

- **Run petrify with the previous .g files and the specified library, you may add the following lines to your ~/.aliases:**
  ```
  setenv petlib <YOUR CELLS LIBRARY>.lib
  alias petri 'petrify \!^.g -o \!^.o -eqn \!^.eqn -
  rst1 -vl \!^.v -lib $petlib -nmap 100 -tm10 -
  tm_ratio1'
  ```
  **then run petrify from terminal**
  ```
  >petri ac0
  >petri aci
  ```
  **the following files will be generated:**
  **ac0.o, aci.o, ac0.eqn, aci.eqn, ac0.v, aci.v**
  **run draw_astg for the previous .o files:**
  ```
  >draw_astg ac0.o > ac0.o.ps
  >draw_astg aci.o > aci.o.ps
  ```

### 6.2. Synchronous Design Flow

- Now, we have VERILOG module for the controllers, v2vhd script is used in order to translate them into VHDL.
```
>v2vhd ac0.v > ac0.vhd
>v2vhd aci.v > aci.vhd
```
- Compile the CSX_HRDLIB library:
This library is not compiled at the latest release of Synopsys at the VLSI lab, so we compile it separately.
```
>mkdir  src lib  lib/FTSM
>cp /hj/ams340/AMS_3.40_CDS/vital/csx/csx_HRDLIB_* src
>echo "csx_HRDLIB_FTSM: ./lib/FTSM" >> .synopsys_vss.setup
>echo "csx_HRDLIB > csx_HRDLIB_FTSM" >> .synopsys_vss.setup
>vhdlan -noevent -nc -work csx_HRDLIB_FTSM src/csx_HRDLIB_Vtables.vhd
>vhdlan -noevent -nc -work csx_HRDLIB_FTSM src/csx_HRDLIB_Vcomponents.vhd
>vhdlan -noevent -nc -work csx_HRDLIB_FTSM src/csx_HRDLIB_VITAL.vhd
```
- **Compile the system's components:**
```
>vhdlan -nc -noevent def.vhd
>vhdlan -nc -noevent buf.vhd
>vhdlan -nc -noevent lch.vhd
>vhdlan -nc -noevent nff.vhd
>vhdlan -nc -noevent ac0.vhd
>vhdlan -nc -noevent aci.vhd
>vhdlan -nc -noevent ooo.vhd
>vhdlan -nc -noevent syn.vhd
>vhdlan -nc -noevent tbn.vhd
>vhdlan -nc -noevent chk.vhd
```
- Use Design Analyzer (da) for creating SDF (Standard Delay Format) files:
```
>da&
```
File→Read

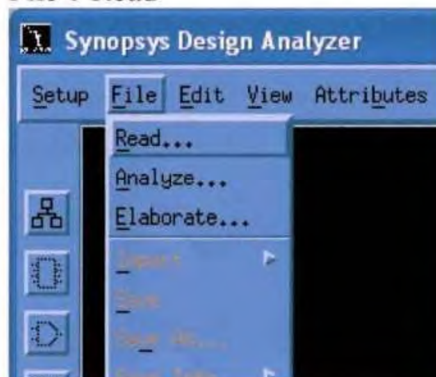



**Figure 39: file→read**

Choose ac0.vhd→Press OK

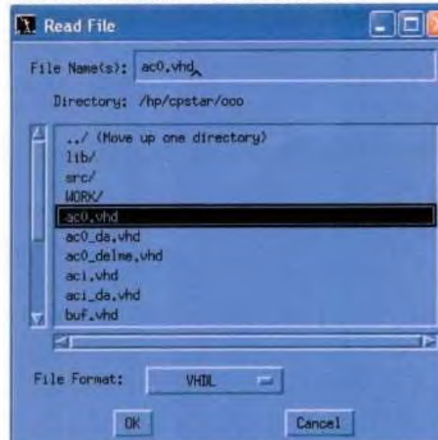



**Figure 40: choose file**

Setup→Command Window

**Figure 41: Command window**

- At command window type:
  Set dont-touch on all the components.
  `set_dont_touch find(cell, {*U*})`
  Write out the sdf file.
  `write_timing -format sdf-v2.1 -context vhdl -output ac0.sdf`

**Figure 42: set_dont_touch**

- Same operation to aci.vhd

- Simulation with SDF (Standard Delay Format):
  Run Scirocco:

```
>scs -ccpath /usr/local/bin/gcc -exe /tmp/scsim$$ chk_cfg
>scirocco +sim+/tmp/scsim$$ +simargs+"-debug_all -sdf
/CHK/SYN0/SYNSLC(0)/SL0/AC00:ac0.sdf -sdf
/CHK/SYN0/SYNSLC(1)/SLI/ACI0:aci.sdf -sdf
/CHK/SYN0/SYNSLC(2)/SLI/ACI0:aci.sdf -sdf
/CHK/SYN0/SYNSLC(3)/SLI/ACI0:aci.sdf -sdf
/CHK/SYN0/SYNSLC(4)/SLI/ACI0:aci.sdf -sdf
/CHK/SYN0/SYNSLC(5)/SLI/ACI0:aci.sdf -sdf
/CHK/SYN0/SYNSLC(6)/SLI/ACI0:aci.sdf -sdf
/CHK/SYN0/SYNSLC(7)/SLI/ACI0:aci.sdf" > &
/tmp/scirocco$$.log &
```

  At VirSim main window:
  Window→Hierarchy
  Window→Wave
  Window→Source

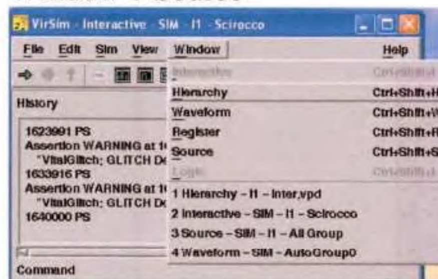


**Figure 43: Window→Hierarchy, Wave, Source**

- At Source window:
  Press the left upper icon, then press "All Group".
  Press the right yellow arrow.

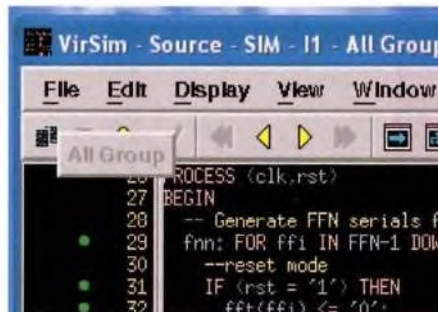


**Figure 44: All Group**

The RTL source code will be shown at "Source" window.
At Hierarchy window:
Click the top cell "CHK", click on any signal you want to trace then push "Add" button.

**Figure 45: Choose trace signals**

These signals will be added to the Wave Window

- To run the simulator go to the main window, at "Step Time" box, put ]the time period, then push "OK"

**Figure 46: Step Time & Run**

The simulation waveform result will be shown at the Wave window.

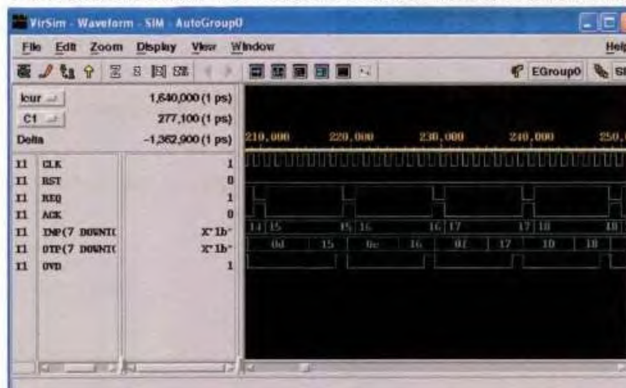



**Figure 47: Waveform**

## 7. References


### 7.1. Articles

- D. J. Kinniment, A. Bystrov, and A. Yakovlev,"Synchronization Circuit Performance,"IEEE Journal of Solid-State Circuits, vol.37, pp.202--209, 2002.
  http://www.staff.ncl.ac.uk/david.kinniment/Research/papers/Jssc2002.PDF
- David G. Messerschmitt,"Synchronization in Digital System Design," IEEE Journal of Selected Areas in Comunication, vol. 8, NO. 8, Octobor 1990
  http://www.eecs.berkeley.edu/~messer/PAPERS/IEEE/Oct90-1.pdf
- D. J. Kinniment and J. V. Woods,"Synchronization and Arbitration Circuits in Digital Systems," Proceedings of the IEE, vol.123, pp. 961--966, 1976.
  http://www.staff.ncl.ac.uk/david.kinniment/Research/papers/IEE1976.pdf
- Y. Semiat and R.Ginosar, "Timing Measurements of Synchronization Circuits," ASYNC 2003.
  http://www.ee.technion.ac.il/~ran/papers/SemiatGinosar-TimingMeasurements-Feb03.pdf
- R. Ginosar, "Fourteen Ways to Fool Your Synchronizer," ASYNC 2003
  http://www.ee.technion.ac.il/~ran/papers/Sync_Errors_Feb03.pdf
- R. Dobkin, R. Ginosar and C. Sotiriou, "Data Synchronization Issues in GALS SoCs," ASYNC 2004}
  http://www.ee.technion.ac.il/~ran/papers/gals-clocking.pdf
- U. Frank and R. Ginosar, "A Predictive Synchronizer for Periodic Clock Domains," PATMOS 2004
  http://www.ee.technion.ac.il/~ran/papers/FrankGinosarPatmos2004.pdf
- R. Ginosar, "MTBF of a MultiSynchronizer System on Chip," 2005
  http://www.ee.technion.ac.il/~ran/papers/MTBFmultiSyncSoc.pdf

### 7.2. Web Sites

- Publications - Ran Ginosar.
  Technion-Israel Institute of Technology
  http://www.ee.technion.ac.il/~ran/publications.html
- Petrify: a tool for synthesis of Petri Nets
  and asynchronous circuits.
  Universitat Politècnica de Catalunya, Barcelona, Spain.
  http://www.lsi.upc.edu/petrify/
- Visual STG Lab.
  Technical University of Denmark, DTU.
  http://vstgl.sourceforge.net/
- The Asynchronous Logic Home Page
  The University of Manchester, School of Computer Science
  http://www.cs.man.ac.uk/async/index.html

### 7.3. Books

- W.J. Dally and J.W. Poulton, "Digital Systems Engineering," Cambridge University Press, 1998.
  http://www.amazon.com/gp/product/0521592925/qid=1146910024/sr=2-1/ref=pd_bbs_b_2_1/103-9550353-9680653?s=books&v=glance&n=283155

- T. H.-Y. Meng, Synchronization Design for Digital Systems (Eds.): Kluwer Academic Publishers, 1991.
  http://www.amazon.com/gp/product/0792391284/qid=1146910643/sr=1-1/ref=sr_1_1/103-9550353-9680653?s=books&v=glance&n=283155

- D. M. Chapiro, "Globally-Asynchronous Locally-Synchronous Systems," Stanford University, 1984.
  http://www.amazon.com/gp/product/B00071483C/qid=1146910981/sr=1-1/ref=sr_1_1/103-9550353-9680653?s=books&v=glance&n=283155