

FULLY SYNCHRONOUS DESIGN

By Serge Mathieu

1- INTRODUCTION.

By the end of my 30 years career in electronic design, I designed a few complex ASICs, like this high performance Powerline transceiver ASIC. See :

http://www.arianecontrols.com/documents/AC-PLM-1_User_Manual.pdf

This digital ASIC represents 2-1/2 man-year of effort. It has been designed by a 2 people team, using a schematic capture software with simulation for debugging. Sections of the design were implemented with FPGAs, in order to prove the design in real environment, in real life. The completed FPGA design was transferred to an ASIC in order to reduce cost and increase performance.

During the design phase, I have developed what I believe are original circuits and methods to allow for **Fully Synchronous Designs (FSDs)**. I would like to share my experience with others, may be it could help someone.

DEFINITION OF "FULLY SYNCHRONOUS DESIGN."

Fully synchronous design refers to digital design, where all FFs (Flip-Flops) clock inputs are connected to a single clock source: the system clock. This is the case for the ASIC described above: the two parts of this design comprising 10,000 + FFs are clocked respectively by a single clock signal: the n MHz system clock. (This design uses 2 internal clock signals, and clock domain crossing is implemented.)

Fully synchronous designs have so many advantages that never again I would design any digital circuit without the principles and basic tools you will learn here. Advantages of fully synchronous design are, among others, :

- Combinatorial glitches have no consequences on the functionality of the system. Glitches need not to be analysed or even taken into account. Knowing that glitches may differ from one implementation to another (e.g. from Asic to FPGA or full custom), ignoring glitches is a major advantage.
- Timing analysis is simplified. One must simply make sure that set up and hold times for all FFs are met. (We will see later that path delays, the delay introduced by combinatorial logic between the Q output of the source FF to the D input of the destination FF, path delays may be longer than the clock period.
- FSDs (Fully Synchronous Designs) are portable, easily transferred from one technology to another, FPGA to ASIC, full custom, finer geometries, etc.
- FSDs are extremely robust.
- Adding a scan path (for test purpose) is easy since all FFs already have a single clocking source.
- Test vectors are more easily generated: vectors are compared once per clock, just before the active

edge of the clock. Glitches are ignored.

- Design is clean, easy to understand, modifications are easy.

- etc.

You will learn here how to design high performance, fully synchronous digital systems.

Serge Mathieu
smathieu13@hotmail.com

FULLY SYNCHRONOUS DESIGN

By Serge Mathieu

For full understanding, read the whole document once before skipping directly to a selected page.

- Rules for Fully Synchronous Designs (FSDs)	
- Conventions	
- Basics: C and S	
- Basics: Enable	
- Basics: Mixed C, L E	
- Basics: Toggle C, L, E	
- Basics: CS and SC	
- Example 1	
- Example 2	
- Event driven	
- Path delay	
- Clock skew	
- Events, events	
- Example 3	
- Dual clock domain crossing	
- Always enable them	
- Virtual clock domain crossing	
- To be continued	

FULLY SYNCHRONOUS DESIGN

By Serge Mathieu

These rules allow for the design of fully synchronous systems, easily portable from one technology to another. (From FPGA to ASIC or full custom, for example.) One should comply with these rules to get a robust and portable designs.

1- SINGLE CLOCK. Each and every flip-flop clock input of the whole design is driven by a single clock signal: all clock-input of FFs are connected to the system clock. Avoid ripple clocking, avoid gating clock. Do not insert clock delays using buffers. If possible, use a single clock buffer. If more than one clock buffer is used, minimize clock skew by careful physical design and placement.)

2- THE SAME ACTIVE EDGE. All FFs (Flip-Flops) in a design must use the same active edge of the clock. Do not mix FFs with positive and negative edge in a design.

3-NO RESET/SET. The asynchronous set/reset of FFs is never activated nor used at "run-time". FFs are asynchronously reset at power-on only. Resetting a FF or a function (resetting a counter for example) must be done synchronously at run-time.

4- LATCHES. Do not use latches. Always use registers. No R-S latches using cross-connected gates.

These rules lead to clean, portable designs. Scan paths are easily implemented.

FULLY SYNCHRONOUS DESIGN

By Serge Mathieu

CONVENTIONS.

To get highly portable design, use the simplest FFs. In my designs, 99,9%+ are D FFs with asynchronous reset. According to the rules for FSDs, (Fully Synchronous Designs) the asynchronous reset is never used at run time. The asynchronous reset is used only once, at power-on. Avoid using FFs with the set function: they are not really needed in most designs. (The function of R-S FF is implemented synchronously, without the use of the async. set/reset.)

A) In diagrams to follow, the R pin of FFs are always connected to the general power-on reset. I will omit drawing this connection.

B) In diagrams to follow, the clock input of FFs is always connected to a single clock signal: the system clock. I will not draw this connection. In diagrams to follow, it is assumed that all FFs use the same active edge for clocking. (Positive edge.)

C) In diagrams to follow, it is assumed that all FFs use the same active edge for clocking. (Positive edge.)

THE BASICS PART 1:

Sequential functions are built using combinatorial logic and FFs. (Flip-Flops) Any given FF in any given application can only take the following values:

- Its own value. (the FF is locked, or idles.)
- Its own complementary value. (Toggle function)
- A zero. (A constant data involving the reset or clear function)
- A one. (A constant data involving the set function)
- An external value (Variable).

FULLY SYNCHRONOUS DESIGN

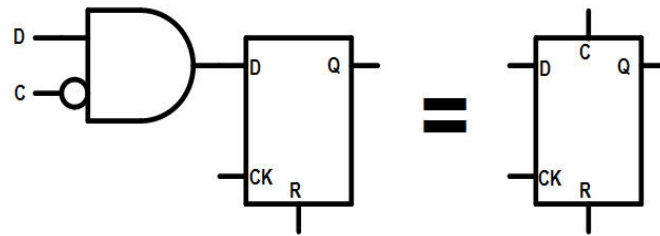
By Serge Mathieu

THE BASICS, PART 2.

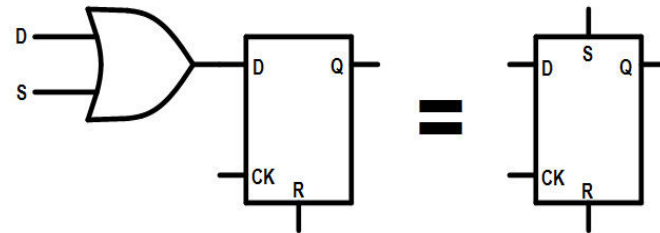
The synchronous set and reset.

FFs may need to be set or reset at run-time. The following diagram shows the basic synchronous set and clear FFs. I use the term "Clear" to make a distinction with the Reset function of FFs, which is asynchronous. In all diagrams to follow, Clear is always synchronous. In the diagram below, the implementation of the synchronous FFs is on the left, and the symbol on the right. Remember: R is always the asynchronous reset, C or S are synchronous.

D CLEAR



D SET



Most applications will use the D Clear FF almost exclusively. It may surprisingly happen that an application needs a FF with both set and reset facility. In which case, one must decide which function has priority over the other, should both the set and reset signal be true at the same time. To solve, simply connect serially either the and gate followed by the or gate, or the or gate followed by the and gate. Label the symbol as CS or SC, the first letter indicating the priority function.

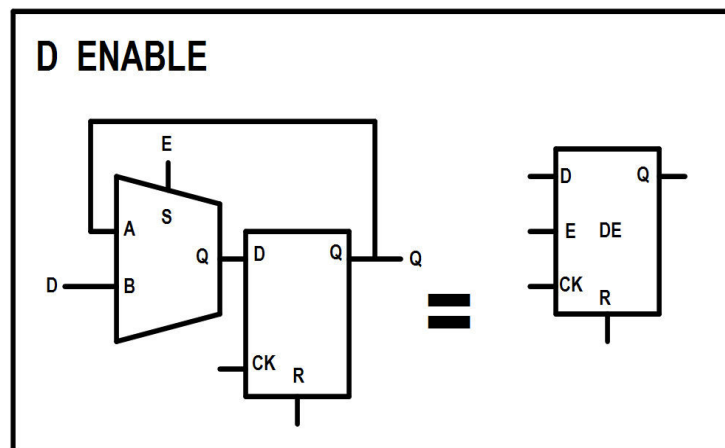
FULLY SYNCHRONOUS DESIGN

By Serge Mathieu

THE BASICS, PART 3.

THE ENABLE FF.

An enabled FF keeps its own value while being clocked. This is the essential "sine qua non" function in order to build fully synchronous systems. Fully Synchronous Designs would be impossible without it.



Some technologies implement the Enabled FF it at the transistor level. If not available, you can implement the function with a MUX (multiplexer) as shown above.

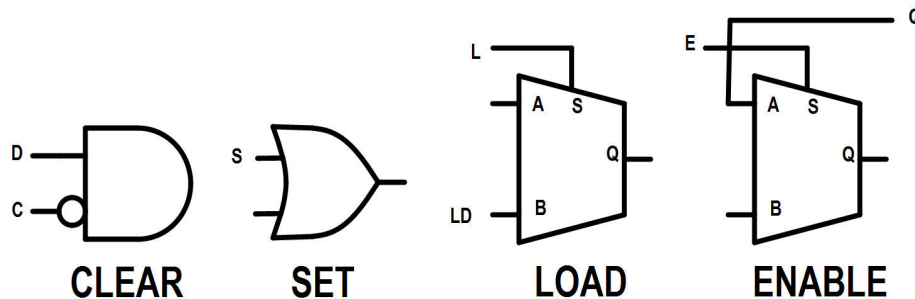
FULLY SYNCHRONOUS DESIGN

By Serge Mathieu

THE BASICS, PART 4.

MIXED FUNCTIONS.

It is possible to design basic flip-flops by mixing the basic functions Clear, Set, Load and Enable, (C, S, L, E). Building these FFs and deciding which function has priority over the others is simply done by cascading the required components shown below, the order in the cascade defining which function gets the priority.

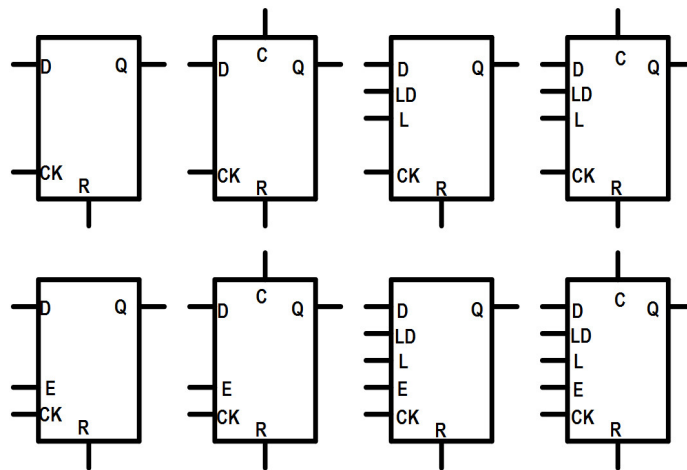


To design a clearable and loadable FF, clear having priority, simply cascade the Load MUX, the Clear gate and the D FF. To design a EC FF, (the FF is cleared only if the enable true, cascade the required components in "EC" reverse order .

All the possible combinations would lead to a large number of FFs. We can reduce the number of FFs by adopting the following conventions:

- 1- The set function is never used. (One can easily do without it.)
- 2- The priority order is always: Clear followed by Load, followed by Enable. In other words, the FF is cleared or loaded even if it is not enabled, and clear has priority over load.

This leads to the 8 basic FFs shown below . These 8 FFs will cover 99%+ of your needs. Should you need an exceptional FF, design it with the building blocks above.



THE 8 BASIC FFs USED IN A DESIGN

Most FFs you will use will be the enabled type, and among the 4 enabled FFs, loadable FFs are not many so that the E and CE FFs will cover 98%+ of your needs in a typical application. FFs without enable work at full clock speed, and are generally not legion.

Using these basic tools, you are now almost ready to design Fully Synchronous Systems.

FULLY SYNCHRONOUS DESIGN

By Serge Mathieu

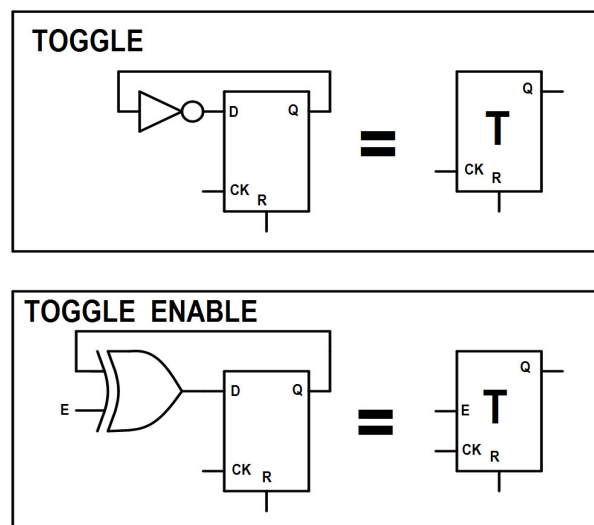
THE BASICS, PART 5.

THE TOGGLE FUNCTION.

You read previously that any given FF in any given application can only take the following values:

- Its own value. (the FF is locked, or idles.) By the enable (disable) function.
- Its own complementary value. (Toggle function) To be seen here.
- A zero. (A constant data involving the reset or clear function) The synchronous clear
- 1 one. (A constant data involving the set function) We will not use this function
- An external data (Variable). The load function, using a MUX.

The toggle function is used to build counters. Counters are so extensively used in designs that it is worth designing a T FF. Here is the extremely complex solution...



Using the basic Clear and Load components shown on the previous page, one can easily design the 8 T FFs required. (see an example below.)

C, clearable T FF

CE, Clearable with Enable T FF, clear having priority over the enable

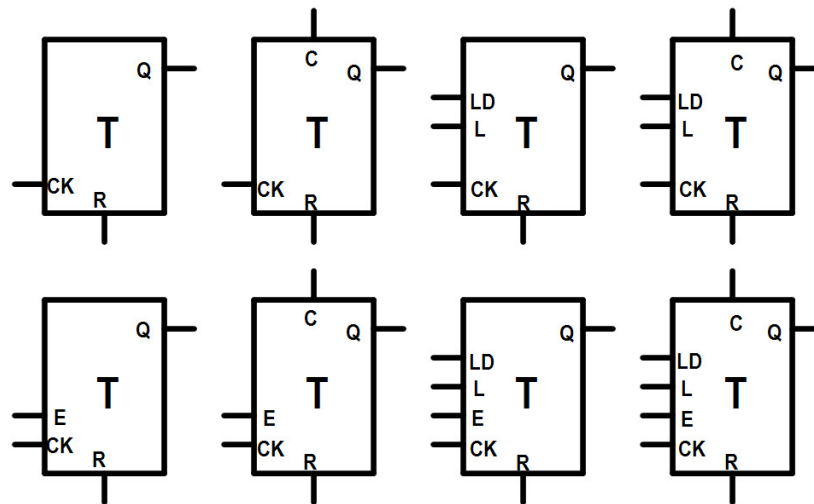
L, Loadable T FF

LE, loadable T FF with enable

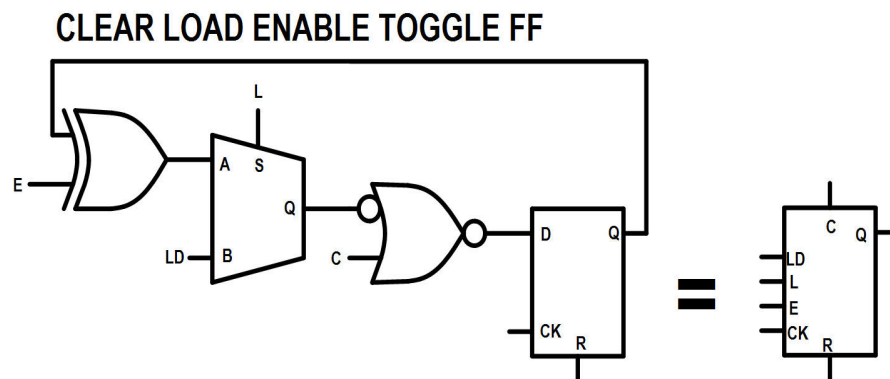
CL, Clearable and loadable T FF, with clear priority

CLE, Clearable and loadable T FF, with Enable.

Including the 2 T FF above, we now have 8 T FFs to put in our tool box that already includes the 8 corresponding D FFs.



Below is the design for the CLE toggle FF.



All sequential functions in your designs will be built from the 8 basic D FFs, and the 8 basic T FFs. Moreover, most functions will be the enabled type, because this principle is so extremely powerful and simplifies the design process.

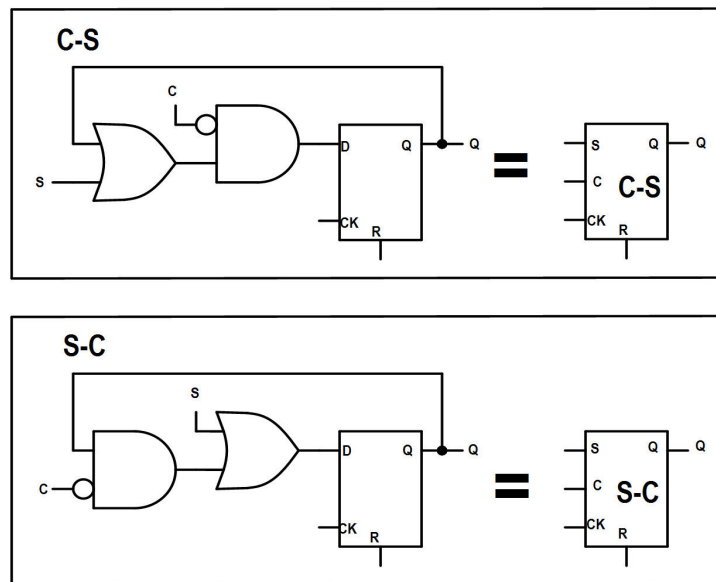
Should you want to design a second order filter with clear function, a clearable accumulator, a loadable-clearable BCD up/down counter with enable, simply chose the appropriate FF to buid the desired function. Examples will follow.

By Serge Mathieu

THE BASICS, PART 6.

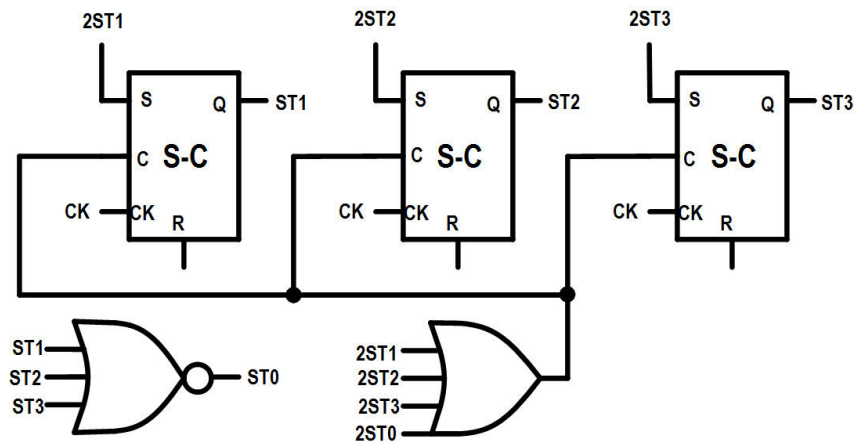
THE C-S FLIP-FLOP.

We will need one more tool in our tool box: the C-S FF (often called R-S). Of course, it must be a synchronous function if we want to retain the benefits of FSDs in the whole system. One thing is for sure: if one inserts a single asynchronous element in his design, the mess is coming!



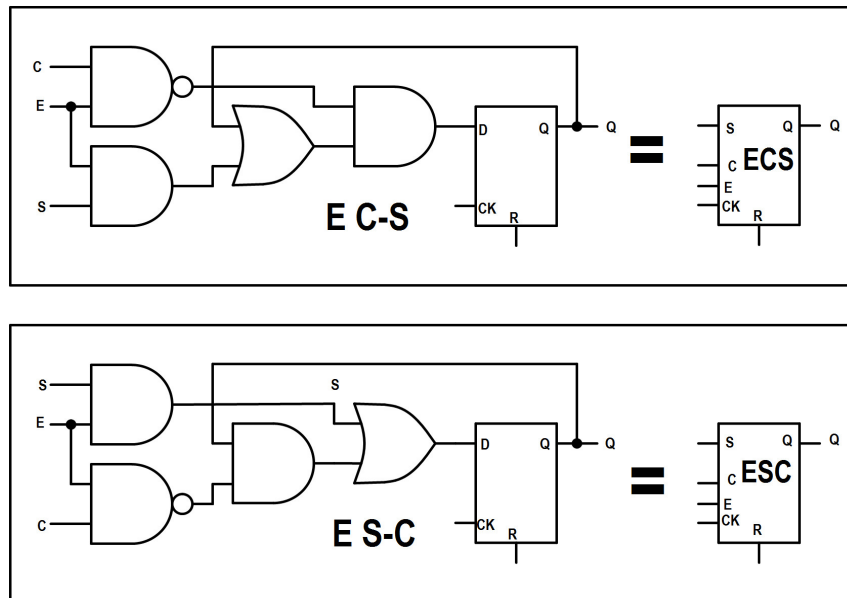
C and S signals will generally be events, (one system clock wide pulses). But we will talk about events later.

C-S FFs can be used as a one bit memory. They can also be used to build efficient state machines, a very useful function in digital designs. See the basic State Machine example below.



The 2ST equations are generally in the form $2ST1 = ST0 * [\text{some logic condition}]$. This makes the 2ST1 signal go back to false at the next system clock after [some logic condition] came true, making the 2ST signal an event. (But we will talk about events later...) The pulse can be used to initialize (start) the State1 work. (Resetting counters, initialize an embedded state machine, etc.) The same holds for all the 2ST. Note that the 2STn clears all the other STATE FFs, including itself! But conveniently, we chose the S-C FF, where Set is priority.

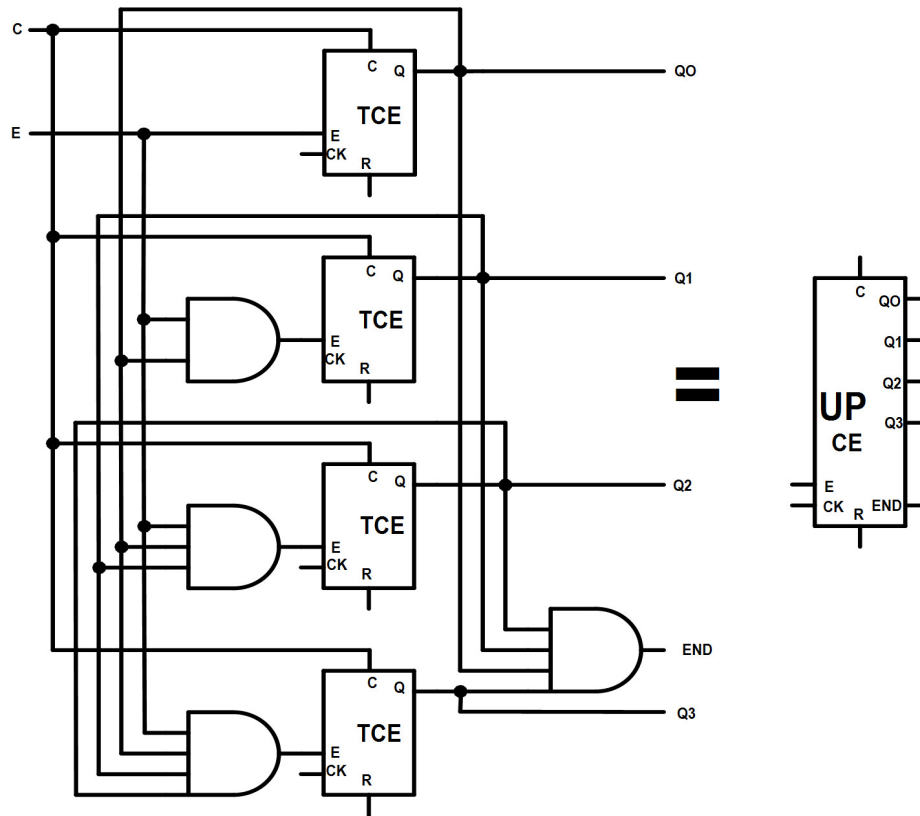
Should the C-S function needs an enable, use these enabled C-S FFs.



FULLY SYNCHRONOUS DESIGN

By Serge Mathieu

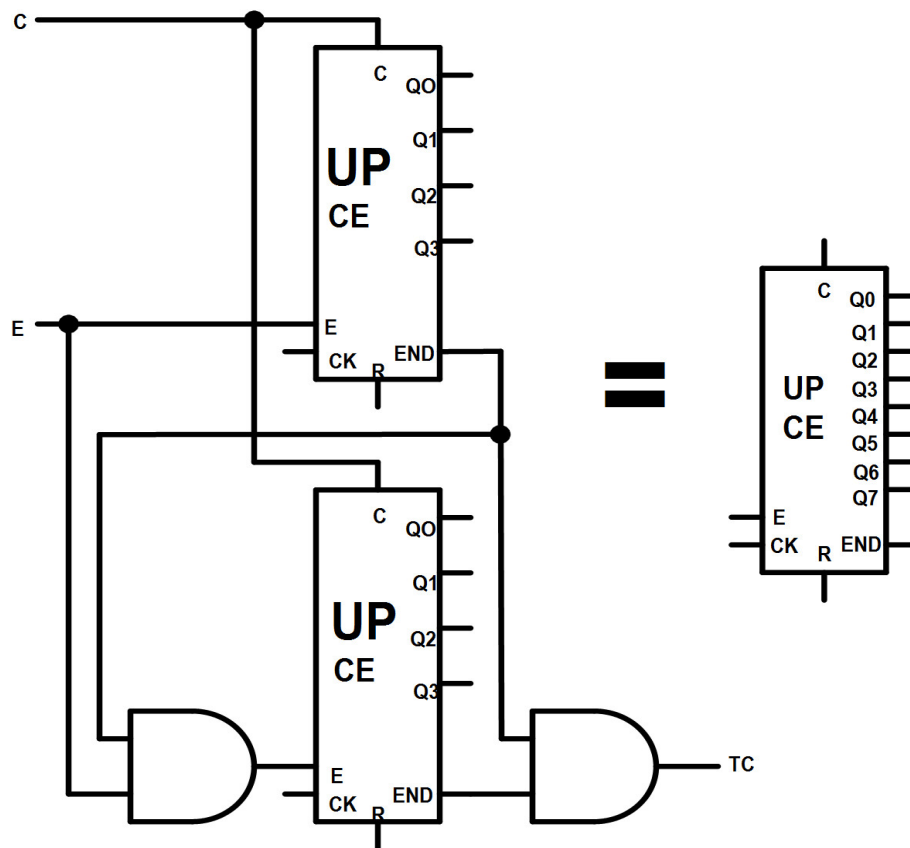
LET US START DESIGNING.



4 BIT UP COUNTER CE

Should you need a Loadable and Clearable 4 bit up counter, simply replace the TCE FFs by a TCLEs ones.

(Remember our convention: Clear has priority over Load, and both have priority over the Enable. In other words: Clear and Load are done even if the FF is not enabled, and if Clear and Load are simultaneously asserted, Clear has priority.



The figure above puts 2, 4 bit counters in cascade to get an 8 bit counter, This counter remains fully synchronous: all counters are clocked by the system clock.

Some will say that this is old stuff. But there is something new about this 8 bit counter:

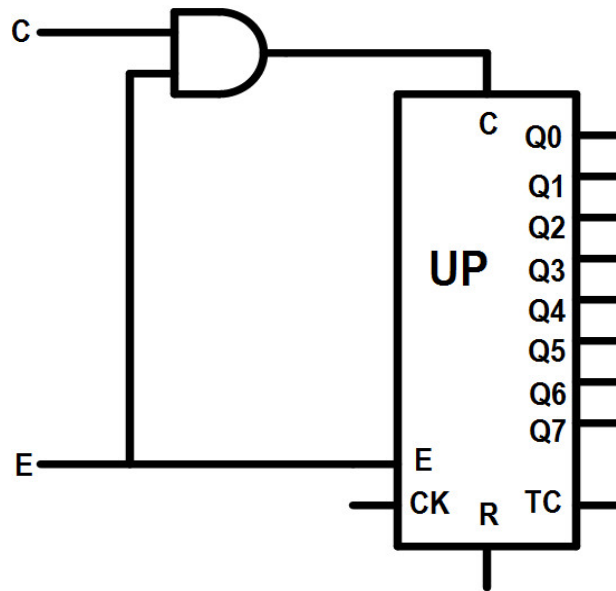
- 1- it is **cleared synchronously** by its clear input,
- 2- it is **enabled**: it does not increment when the enable is false, even if its clock is 10 MHz..

Even if you clock this counter at 10 MHz, it does not count at all if the enable is not asserted. The sole way to make this counter increment by one is to enable it. If one raises the enable hi, the counter starts incrementing at the system clock rate, until the enable goes low again, freezing the counter again. But this not the best idea : Counting the system clock is not quite useful, you rather want it to count something else, anything you might imagine. To do so, the idea is to raise the enable FOR ONE CLOCK PERIOD to make it count the thing you want it to count. Doing so, the counter increments only by one and freezes again.

This is how Fully Synchronous Designs work. Functions are created normally, (Counters, filters, accumulators, specific designs, etc.) using the appropriate enabled FFs, (Clearable and/or loadable) to build the required (clearable and or loadable) enabled functions. These functions idle, (they do nothing, are not alive, not active, are frozen at their actual state) even if they are clocked at x MHz. These function do their jobs only when their enable input is asserted. More examples later about how powerful this approach is.

Do you remember that the clear function has priority over the enable? What if we wanted that this counter counts from 0 to 59, and then rolls back to zero. You understand that to make it count the seconds, one just have to enable it with a **one clock wide pulse (OCWP)** every second. Now, if we

decode the count 59 with some gates, and connect the "=59" signal to the C input, the counter will increment by one every second (because it is enabled only once per second by a OCWP) up to count 59, and will roll back to 0 **the system clock after**, because it clears even if it is not enabled. In other words, the state 59 will last only one system clock period, and not one second. This is not what we want. We want that the counter to be cleared only when enabled, at the next second. We want the enable have priority over the clear, in other words, we want that the counter does not clear when not enabled. Doing so, the counter will roll back to zero at the next one second pulse. To do it, simply combine the enable and the clear signal through an and gate that drives the Clear input. You now have a fully synchronous counter that clears synchronously with its enabling signal.



One may think at this moment that there is nothing really new here, just some potential improvement to an old way of doing things. Just take a few minutes to read the following pages, you will see how powerful is the enable concept.

FULLY SYNCHRONOUS DESIGN

By Serge Mathieu

EVENTS.

Any given digital system comprises logic levels that represent some value. It may be a temperature, a combinatorial state, a status, a false or true, a negative or positive, a number, logic conditions, etc. These values are characterized by the fact that they may be stable for long periods, that is, many system clock periods. But how do we represent an event? Something that happens in a cinch and does not last? A change?

In typical old discrete CMOS logic designs, events were typically represented by the transition (level change) of a signal. This level transition was used as a clock applied to the clock input of a function. To count apples on a conveyor, the apple detector output (hopefully unglitched) was applied directly to the clock input of an apple counter. This approach is cumbersome and dangerous, because it leads to multiple clock signals within a system. Timing analysis becomes rapidly extremely complex. Such designs typically need adding delays on clock and/or signals, trying to resynchronize the system. These designs are hard to port over PVT (process-voltage-temperature) variation. Changing the process and/or, the voltage and/or the temperature could make the system crash. FSD solves these problems, while greatly simplifying the design process.

For example, a number may represent a pressure, but the fact that the pressure just passed over a given threshold (represented by another number) is an event. How do we represent an event in FSDs? How do we "clock" a sequential function, like a counter, when this event happens? We could compare (with a comparator) the actual pressure value with a threshold value, but the output of the comparator yields a status, not an event. A status that says that the actual pressure is above or under the threshold value. The **rising edge** of the comparator output would be an event. But using this rising edge to drive a FF clock is extremely dangerous, your system would probably crash: glitches, induced clock delay, timing problems in the whole system, multiple clocks within a system, etc.

FSDs are built around this dual concept :

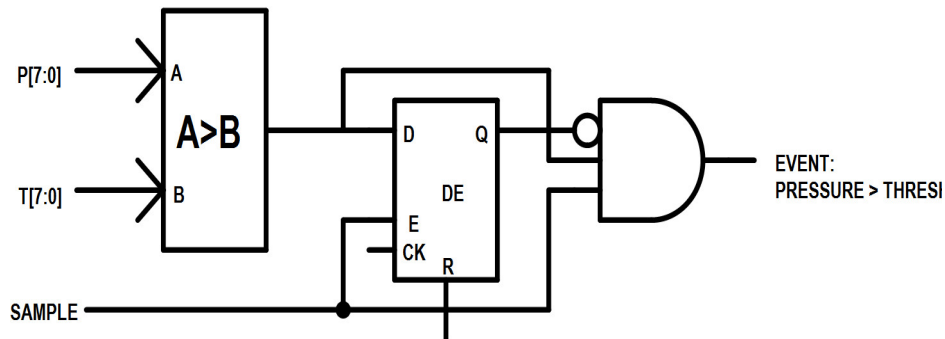
1- Numbers, values, state, status, give them the appropriate name you want. These logic levels may be stable for indefinite periods, (many system clock periods) often depending on the outside world or some internal or external changes.

2- Events : They are One clock wide pulses (**OCWP**) signalling that something happened and that it is time to do something. These pulses enable the sequential functions.

In FSDs, events are represented by OCWPs. These events are used to enable the functions in the system.

FSDs extensively use enabled FFs, **FSDs are event-driven.**

An event is represented by a pulse. See the pressure vs threshold example below. The output of the comparator $A > B$ is a status : actual pressure exceeds (or not) the threshold. Let the sample signal (see below) be a "one clock wide pulse" at suitable frequency for sampling the pressure, then the output of the And gate below is an event: a OCWP that means tells that the pressure just went above the threshold. This is an event. The pulse can be used to initiate (synchronously of course) a state machine, make a counter increment, etc.



SOME INTERESTING PROPERTIES OF THE ENABLE.

In FSDs, all FFs are clocked by a unique system clock. There is no clock delay insertion using buffers, FFs use the same edge for clocking, no gated clock, etc. (See the rules in Part 1.) When a FF is not enabled, it keeps its value, just like if it had no clock signal!

Hence one may think of an OCWP enabling signal as a clock. But the difference with a real clock is that the enable may have glitches, delays, etc. without consequences for the state of the FF. The system will perform properly even with glitched enables, as long as the set-up and hold time of FFs are not violated. And all FFs of the system change their state simultaneously, (supposing that clock skew does not exist) simplifying timing analysis.

In fully synchronous designs complying with the rules expressed above, combinatorial glitches are absolutely meaningless. They have no effect on system performance nor functionality, since they cannot affect the state of FFs. Nothing can change the actual state of a FF, except the clock or the asynchronous reset. Since all FFs are clocked by a single signal, the system clock, (hopefully unglitched!) and since the async. reset is never used to reset a FF at run-time, (Reset is used at power on only) then glitches may be discarded from the problem list, they must even be discarded from the test vectors. Simply make sure that setup and hold times requirements for FFs are not violated. Knowing that glitches are implementation dependant, ignoring glitches is a plus when passing from one technology to another. (FPGA to ASIC, FPGA to FPGA, etc.)

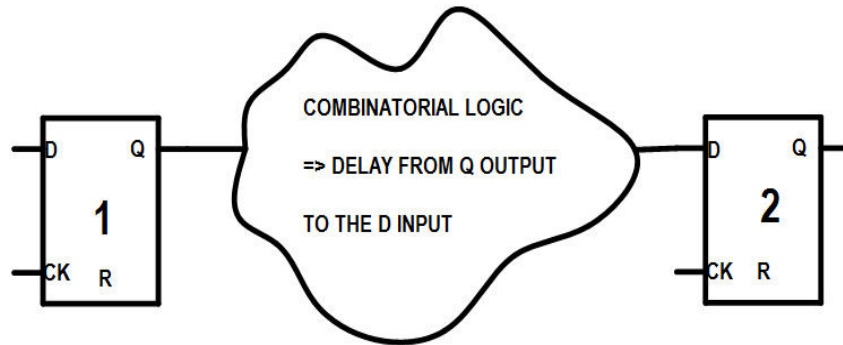
One question now arises. **Is it possible to have a functional system if a path delay is longer than the clock period?** See the answer on next page.

FULLY SYNCHRONOUS DESIGN

By Serge Mathieu

PATH DELAYS.

Synchronous designs can be modeled as:

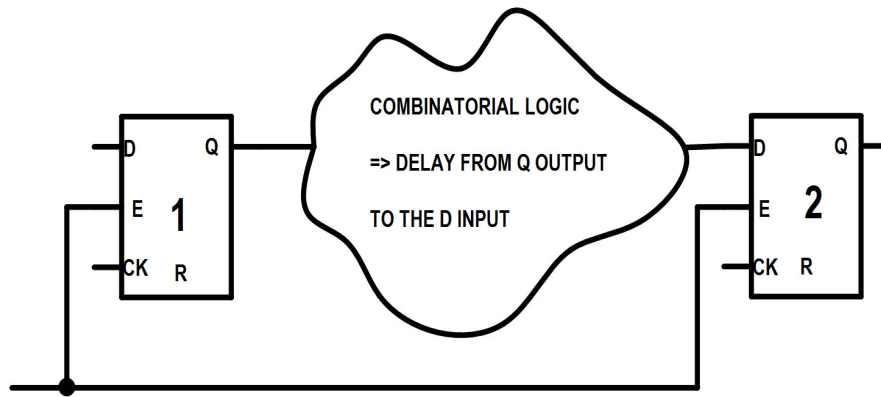


Where the clock signal is the system clock.

One question arises: Is it possible to have a fully functional system if the path delay from the combinatorial logic above is longer than the clock period?

In standard systems, the answer would be no. Designers would use pipe line FFs or some other trick. But in many systems, the information does not need to be processed by the sequential elements at the system clock speed! Nyquist–Shannon sampling theory still holds nowadays: it is not necessary to sample a signal a lot faster than its frequency content. It is not necessary to process data faster than the data rate. Only the fastest physical signals need to be processed at the system clock speed. In practice, large portions of most systems may process data at much lower rates than the system clock. Enabled FFs do exactly that job.

Suppose that data in the figure below could be processed at 10 MHz, (Thus the enable signal is 10 MHz) one could have a 100 MHz system clock (10 nS period), (used to process the fast portion of the design) and the path delay of the figure below could be roughly up to 100 nS, even if the clock period is 10 nS, (in other words, the path delay could be 10 times the clock period!) using an enabled design like this :



**PATH DELAY = 100 Nsec, CLOCK PERIOD = 10 Nsec,
AND THE SYSTEM IS FULLY FUNCTIONAL.**

This systems works even if the clock period is 10 nS and the path delay is +- 100 nS, as long as the enabling path meets the setup-hold time specification. In general, enabling paths are short.

FULLY SYNCHRONOUS DESIGN

By Serge Mathieu

CLOCK SKEW.EW.

In FSDs like in most systems, it is assumed that the Clock signal is perfect: no glitches (hopefully!), fast rise-time and fall-time, etc., but above all, it is assumed that the clock signal activates all the on die FFs at the same time. Unfortunately, this is not necessarily the case. One must be aware that clock skew may exist: FFs are not necessarily clocked at the same time, there could be some pico, even nanoseconds between the clock edge of one FF to the next. In FSDs, clock skew must be analysed between 2 adjacent FFs, adjacent FFs being those linked from the Q output of the source FF (through some combinatorial logic or directly) to the D input of the destination FF. Skew may be positive (the source FF is clocked before the source) or negative.

See http://en.wikipedia.org/wiki/Clock_skew

It is the semiconductor manufacturer's responsibility to implement a good clocking scheme. When you are converting from FPGA to ASIC or full-custom, make him aware that FSDs are clock skew sensitive. I had once problems with one semi manufacturer, the prototype ASIC didn't work because clock skew was too high. The manufacturer settled the issue by redesigning the clock, using a metal grid/tree for clock routing.

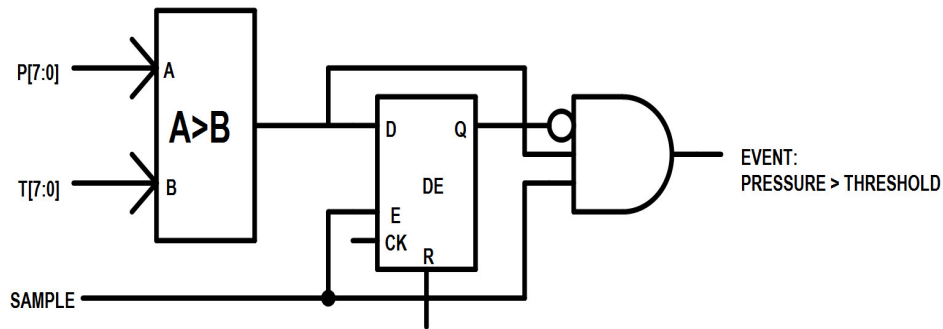
Very short paths could be problematic vs clock skew: they are highly clock-skew sensitive. Direct Q to D connections are a perfect example. Imagine a 4 bit shift register, in presence of clock skew...Potential disaster.

It is a good idea to add some delay to these very short paths: a couple of inverters from Q to D for example.

FULLY SYNCHRONOUS DESIGN

By Serge Mathieu

EVENTS, EVENTS, EVENTS...



Do you remember this diagram comparing the actual pressure to a given threshold? We said that the output of the and gate above is an event: a One Clock Wide Pulse (OCWP). How many events do you see in this diagram? The correct answer is two. Because the signal SAMPLE is also an event! It says that time has come to sample the pressure status.

NEW EVENTS ARE OFTEN GENERATED BY "ANDING" A FORMER EVENT (« Sample » in this case) WITH SOME NEW CONDITIONS.

Do you remember the Power Line Carrier Transceiver mentioned earlier? Let me simplify this ASIC: An event was generated in the FSK demodulator when: the FSK carrier input just did a positive transition. (Using exactly the diagram above) A certain number of carrier transitions would generate a bit ready-event, a certain numbers of bit-ready would generate a word-ready event, a certain number of word-ready events would generate a message-ready event, and so on. Finally, the "message-ready event" pulse comprises the initial carrier transition event.

Count 59 of a seconds counter is not an event. It is a status, since it is stable for one second. The useful event in this case would be: the seconds counter is about to roll back to zero. This new event will be created by anding the logic "seconds=59" with the own enable of the seconds counter. This new event would be used to enable the next counting stage that counts minutes. The event "One hour elapsed" would be the And of the minute counter with its own enable. etc.

An event is often initially generated by the transition of some signal, just like in the figure above. The transition detector above is quite useful. Events are generally not values, data, etc. Events are most often the result of some change, variation, in values, etc.

As an exercise, design a simple time clock. Suppose that the system clock frequency is conveniently 32 768 HZ. The system should use the system clock as reference for time. For the moment, ignore the required logic for the user to set the right actual time. (Note: this logic will use multiplexers on the enables of counters: the minute counter generally counts as described above, or from 4 Hz if the user is setting minutes, etc.)

All the FFs in this alarm clock design must be clocked by the system clock, no exception is allowed.
Do not reset counters using their async reset: always reset counters synchronously by using the basic clearable FFs.

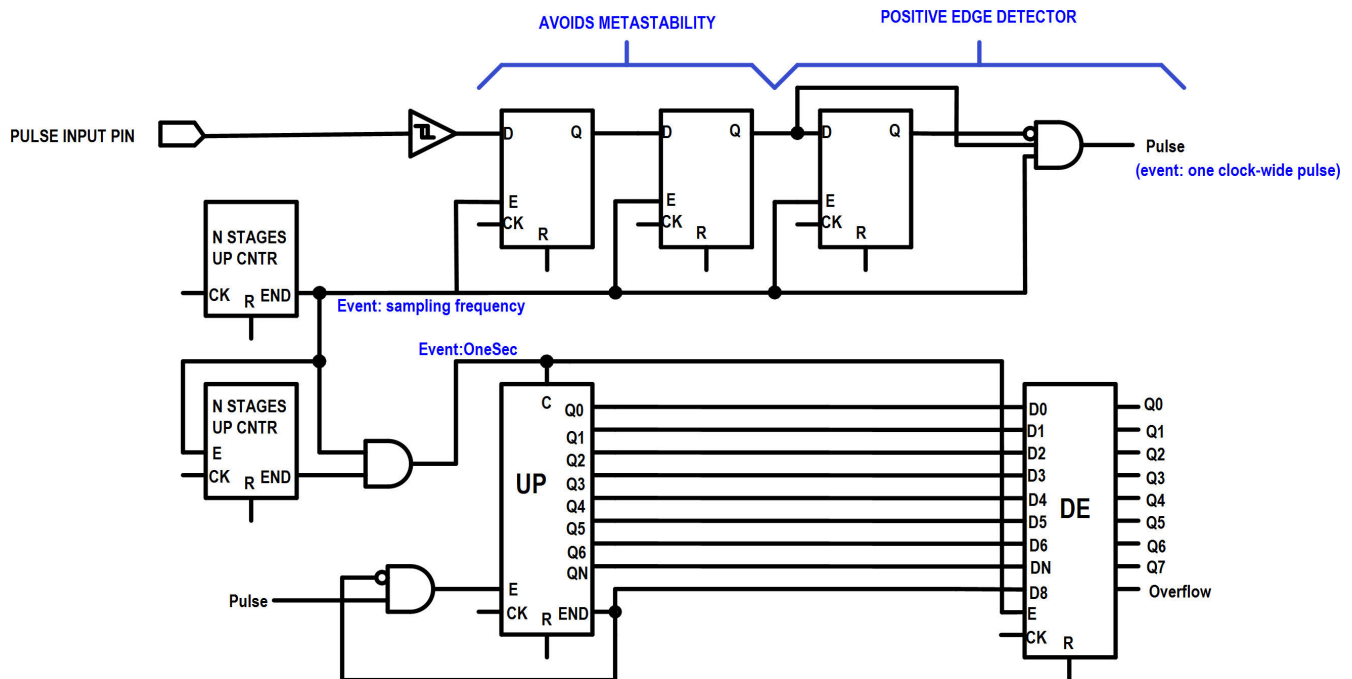
(Suggestion : the OneSec event is the decoding « = 32767 » of binary an UP counter that counts the system clock, the 4 Hz is the decoding of « =8191 » from the same counter. These last one system clock, so that they are OCWPs that can be used to generate events.)

You are just starting having fun with FSDs. See another design on next page.

FULLY SYNCHRONOUS DESIGN

By Serge Mathieu

ANOTHER EXAMPLE.



A RUDIMENTARY FREQUENCY METER OR TACHOMETER

Objectives of the design: count the pulses from the input pin for one second, then transfer the result to a register while resetting the pulse counter. Manage the overflow.

The first N Stages counter generates an event (sample) at a suitable frequency. For example, if the minimum Pulse Input Pin signal width is 100 uSec, then sampling it at, say, 40-50 kHz is ok. The "pulse" output (output of the and gate) increments the UP counter. The sampling signal is divided again to get a One second event, a one system clock wide pulse. When one second has elapsed, at the exact same system clock, the counter output is registered and the counter is cleared. (The DE function simply contains 9 DE FFs.) There is a simple overflow logic included.

Suppose that the 2, N Stages counters have 10 stages, the UP counter has 15 stages, and the DE register is 16 stages wide. Then we have a design comprising 54 FFs.

These 54 FFs, even if they perform various functions, are all clocked by the system clock. (Not shown) Counters are always synchronously cleared at run time. The async reset is used only at Power On. This is a true Fully Synchronous Design, complying with the rules given on page 2.

The OneSec event comes from some combinatorial logic. Consequently, this signal could be glitched. But glitches have no consequence at all. This is why glitches are never compared in test vectors: glitches are insignificant. Adding a series of inverters between the UP and the DE could be a good idea if clock skew is suspected.

The longest path delay goes from the first stage of the upper left N Stages counter, up to the QN stage of the UP counter (Passing through the second N Stages counter). For example, the UP counter could have, say, half a second internal path delays, (Combinatorial logic within the counter could take half a second to settle! Quite a long time!) and this would have no consequence, since this counter is enabled only once per second. However, the enabling path must be fast enough to comply with setup-hold times. Using the basic enabled FFs ensures the shortest possible enabling paths. As already said, enabling paths are generally much shorter than the "working" paths.

Since the clock input of our 54 FFs are all tied together, and since no combinatorial logic makes asynchronous resets, scan paths are easily inserted, should you contemplate an FPGA to full custom conversion. (Asynchronous resetting any FF with internal logic could prohibit scan path insertion.)

Since glitches may be discarded in test vectors because they cannot induce any FF change, the test vectors file contains one only vector per system clock. The sole thing to watch: setup and hold times, taking also into account the effect of potential clock skew.

Remember that "working" paths (those used to implement the designed function) delays may be longer than the system clock period. These paths should be analysed against the enabling path period, not the system clock period.

SECONDARY CLOCK GENERATION, AND CLOCK DOMAIN CROSSING

This design assumes that the LCK sub-system will use the positive edge of LCK to clock its FFs.

DATA HI IN is some data to be passed from the high frequency part of the system to the low frequency one. Analysing the design, you will see that the FF changes its state around the negative edge of LCK, (leaving space for LCK delay/skew setup time, etc.) so that LCK may use it without setup/hold violation. DATA LO IN is some data to be passed from the low frequency part of the design to the high frequency one. This data always changes at the rising edge of LCK, but since it is read by the HCK system near the falling edge of LCK, there is plenty space for set-up/hold times in this direction too.

That simple!

Thanks to enabled FFs, and to HCK-LCK synchronization.

FULLY SYNCHRONOUS DESIGN

By Serge Mathieu

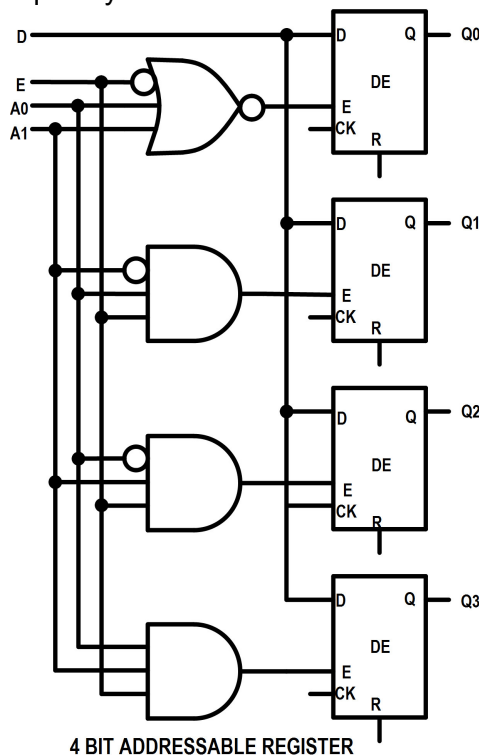
ALWAYS ENABLE THEM.

Always enable the functions you design: the enable principle is so powerful and extremely useful. In other words, always use enabled FFs to build functions, and design an enable for the whole function. Design your systems on the data/event principle. And of course, always design fully synchronous systems. Should you design

- an adressable register,
- a successive approximation register (for A/D conversion),
- a first order low pass filter,
- a BCD loadable, clearable up/down counter
- a thermostat,
- a toy,
- etc.

Enable the functions even at the highest level! If finally the enable input of a function is not required, just connect the enable input to Vcc (hi), and the development tools will remove all the unused gates. An enabled function is versatile. It is expandable (in counters for example). It is re-usable, whatever its operating frequency versus the system clock.

Always enable the whole function. Doing so, the function will perform at the pace required by the application, whatever the system clock frequency. Non-enabled function are limited: they can only work at the system clock frequency.



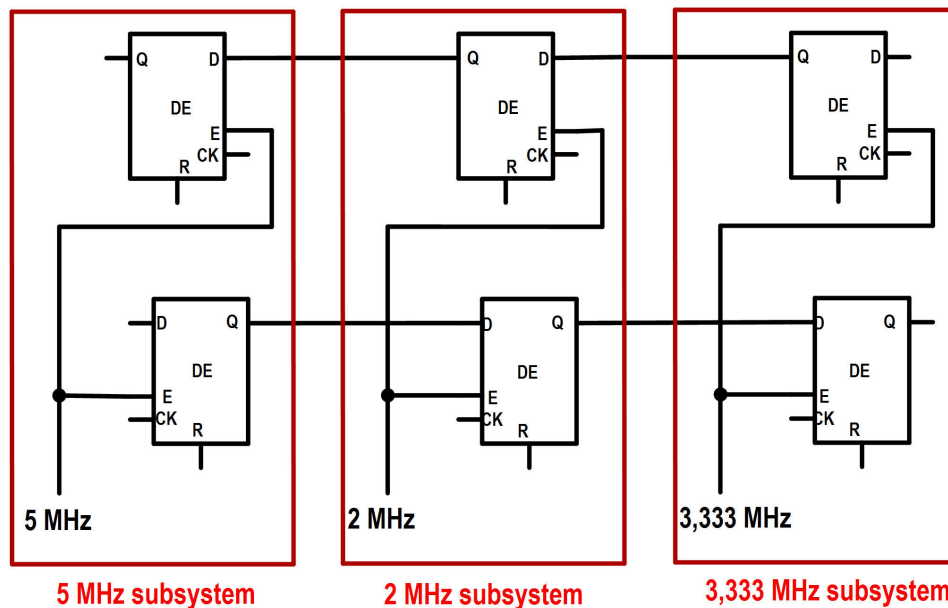
FULLY SYNCHRONOUS DESIGN

By Serge Mathieu

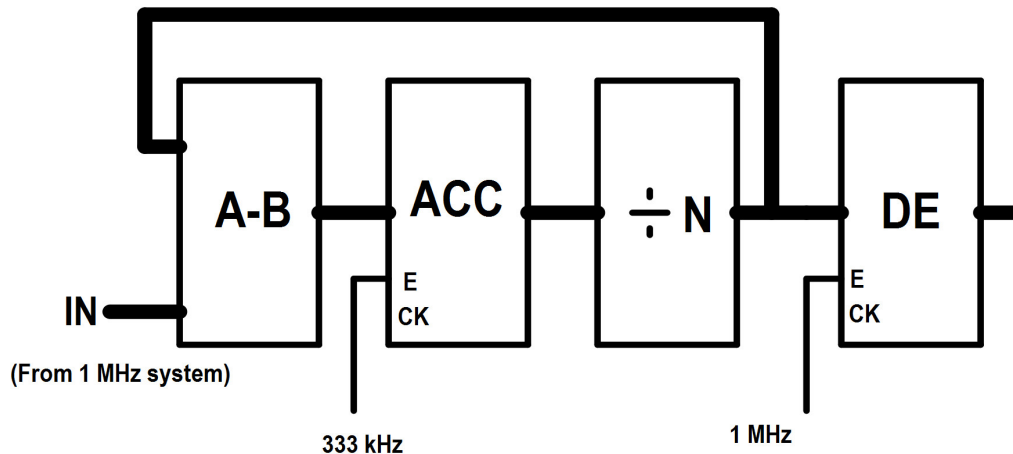
VIRTUAL CLOCK DOMAIN CROSSING.

We have already seen that a OCWP (One Clock Wide Pulse) enabling signal can be thought exactly as a virtual clock applied to a FF or to a whole subsystem. OCWP makes FFs change state, it makes FFs act as if they had received a normal clock pulse. A OCWP enabling signal is an event, just like the clock edge was an event in the old conventional CMOS-TTL circuits. Consequently, seen from a given subsystem perspective, an enabling OCWP signal applied to all FFs of this subsystem has exactly the same effect as if this subsystem had been clocked. In other words, remove the enabling OCWP (or tie the enable signal hi) and clock the subsystem each time there would have been an OCWP, and the behavior should be the same. OCPW are like « virtual clocks ».

A OCWP enabled subsystem performs exactly like if this subsystem were part of a new clock domain. However, there is a huge difference: if the various enabling OCWPs are all built from the same system clock, then clock domain crossing is no more a problem, since even if various subsystems work at various frequencies, they remain coordinated by the system clock. Hence the following "clock domain crossing" works without any special effort. This could be called some sort of "easy clock domain crossing."



In the lower part of the diagram above, data is passed from a 5 MHz subsystem to a 2 MHz one, which also passes some data (not necessarily the same!) to a 3,333 MHz subsystem, as the upper part of the diagram shows the reverse circulation. Data passes from one "virtual clock" domain to another without the mess of real clock domain crossing. Of course, one must assume that when data is passed from a high frequency domain to a lower frequency one, the source data is stable for at least one virtual clock of the receiving domain. Otherwise, data could be lost.



See above a low pass first order filter. The cutoff frequency is the operating frequency (333 kHz) divided by $(2 * \pi * N)$. If N is conveniently 16 (divide by 16 is a simple rounded shift right), then the cutoff frequency is 3,315 kHz. The function ACC is an accumulator. Also note that the output of the A-B block is the high pass first order filter.

Changing the enabling frequency changes the cutoff frequency. The input IN to the filter comes from the 1MHz subsystem, and the output is returned to this 1 MHz subsystem. Note that since all enabling pulses are built from the system clock, they are either coincident (the 333 kHz pulse coincides with the 1 MHz one), or separated by some phase difference. The smallest phase difference (depending on the generated frequencies) is one system clock. In such a case, the path delay from one system to another must be shorter than one system clock. In all other cases, it may be longer.

Another interesting property of OCWPs: These events can be directly applied to any other subsystem working at any frequency. This means that a 200 kHz subsystem event may reset a state machine in the 1 MHz subsystem, make this machine advance to another state, and the reverse direction is also true: the 1 MHz subsystem may reset counters within the 200 kHz subsystem, etc.

All these subsystems, operating at various frequencies, still make a FULLY SYNCHRONOUS DESIGN!

This is the power of enabled FFs. And that's just the beginning!