Clock Management Tips on a Multi-Clock Design

Sylvain Haas

Motorola Semiconductor Products Sector

Sylvain.Haas@motorola.com

ABSTRACT

The ideal synthesizable design has only one clock and all its flip-flops are triggered on its positive edge. That makes the whole flow a lot easier, from synthesis to post-route timing analysis via DFT stages. Unfortunately, that dream is usually far from reality; the design uses both clock edges, has latches, several internal clocks, sometimes coming from different PLLs, and even asynchronous inputs that need to be used as edge triggers.

The purpose of this paper is to give a set of mechanisms and synthesis tips that should get the design as close as possible to the ideal case. The clock issues are progressively introduced; a solution is always proposed, either as a synthesizable positive edge-triggered mechanism, or as a constraint tip, with its application requirements and its drawbacks. Every proposition has already been used on a real design case.

1.0 Introduction

In current designs, multiple clocks are utilized. There are the clocks required to time external interfaces, internal clocks generated from different phase-lock-loops, etc.

There are a many problems due to complex clocking schemes. Such as inadequate false paths, some real paths are filtered out and thus not analyzed. The analysis itself becomes very complex because the false path definitions did not remove all the false path violations. It may require case analysis and even dynamic timing analysis with functional patterns to detect particular problems.

Moreover, the timing analysis at chip level is conducted by designers that have to merge several complex clocking schemes and analyze the module timings. Unfortunately they do not have complete knowledge of all the designs and thus have a difficult task. To ease design and integration, the best structure is one single clock and all the module registers are triggered from the same edge (usually positive).

The question becomes how to achieve that best design structure and still meet all the constraints that were solved by the presence of multiple clocks. To reach that goal or at least approach it, this paper proposes a two-step process:

- remove all the clock oddities,
- iffor the remaining clocks, provide tips to facilitate synthesis and reduce the negative impact of multiple clocks, asynchronous clocks or multiple clock edge utilization.

Solutions are fully described, including a schematic, a timing diagram and Verilog code sample; pros and cons are studied, with a special focus on power as it is often a good reason for having multiple clocks within a design.

All the solutions assume the existence of a relatively high frequency clock in the module, which is considered as the main clock in the following paragraphs.

2.0 Low frequency clock reception

Let us first consider a module that receives a low frequency clock in addition to its high frequency main clock. That low frequency clock is generated externally to the module and serves as reference for input and/or output data. This is typically the case with serial and parallel external interfaces like UART and JTAG.

The typical implementation uses the input low frequency clock to drive a few flip-flop clock pins, which requires a dedicated clock tree and multiple synchronization mechanisms everywhere data goes from that clock domain to the main chip clock domain.

The related issues are numerous: having asynchronous clocks makes verification, synthesis and timing analysis more complex, every additional clock requires a dedicated effort to setup the design constraints and during clock-tree synthesis.

2.1 Do you really need a clock?

To reduce the number of clocks, the first step consists in analyzing the functionality in order to find a synchronous mechanism that will be equivalent.

Sometimes, data signals are connected to flip-flop clock pins to implement a very particular behavior. A typical example is the detection of an input signal variation when the main clock of the module is gated off. Provided the signal toggle rate is low compared to the main clock frequency, it is usually possible to implement a synchronous mechanism that fits the need.

Below is an example that illustrates such a clock usage and proposes a simple mechanism to perform the same function. The goal of that paragraph is simply to demonstrate that it is sometimes possible to have a single clock solution and encourage designers to spend enough time searching for that kind of solutions before considering adding another clock to their module.

2.1.1 Problem description

The peripheral has to trigger an asynchronous wake-up command (*async_trigger_posedge*) to the CPU upon reception of the rising transition of an input signal (*async_trigger*). The wake-up command is cleared by the CPU by writing to a specific register that triggers the *clr_async_trigger* signal; that clearing is always performed with the main clock running.

That mechanism has to work when the main clock is not running. The resulting command does not need to be synchronous and it cannot be as the reference clock is not always available.

2.1.2 Former solution

The *async_trigger_posedge* signal is connected to a flip-flop clock pin which naturally detects its rising edge regardless of the main clock status. The corresponding Verilog code is available below:

```
reg async_trigger_posedge;
reg clr_async_trigger;
wire clr_async_trigger_reset_b = (~clr_async_trigger) & main_reset_b;

always @(posedge async_trigger or negedge clr_async_trigger_reset_b)
    if (!clr_async_trigger_reset_b)
        async_trigger_posedge <= 1'b0;
    else
        async_trigger_posedge <= 1'b1;

always @(posedge main_clk or negedge main_reset_b)
    if (!main_reset_b)
        clr_async_trigger <= 1'b0;
    else if (clear_async_trigger)
        clr_async_trigger <= 1'b1;
    else if (start_detecting_async_trigger)
        clr_async_trigger <= 1'b0;
```

2.1.3 New system

The exact functionality analysis shows that the system only goes to sleep once the *async_trigger* input signal is low. That means detecting its rising edge is similar to detecting when it is high. The proposition is to implement a small FSM (one flip-flop) the state of which is combined with the incoming signal to generate the wake-up command. Before going back to sleep, the system enables the propagation of the input signal high level to the wake-up command. The following Verilog code implements that single-clocked solu-

tion:

```
reg detecting;
wire async_trigger_posedge = (detecting == 1'b1)? async_trigger : 1'b0;

always @(posedge main_clk or negedge main_reset_b)
    if (!main_reset_b)
        detecting <= 1'b0;
    else if (clear_async_trigger)
        detecting <= 1'b0;
    else if (start_detecting_async_trigger)
        detecting <= 1'b1;
```

Simply by reconsidering the problem, it was possible to find a solution that does not require the addition of a clock with all its inherent drawbacks.

2.2 Clock sampling mechanism

We have seen that it is possible to remove some unnecessary clocks. The example above was a solution for a very particular case. What follows is a well-known general solution to connect an external interface with a low frequency reference clock, to a module that has a high frequency clock at its disposal. Yet, that type of solution is not always used despite of its simplicity. We will determine its real limitations with a special interest in power consumption that is a common argument against its usage.

The described system is implemented with clock and data separate. However, the same principles can be applied when clock and data are merged like in UARTs.

The incoming clock is sampled with the much faster main reference clock, which generates an enable when the desired clock edge is detected. Based on that enable, all inputs are sampled and all outputs are generated. The computational logic should work based on that enable to calculate the output data, which matches the interface throughput capability.

Figure 1 Low frequency clock synchronizer

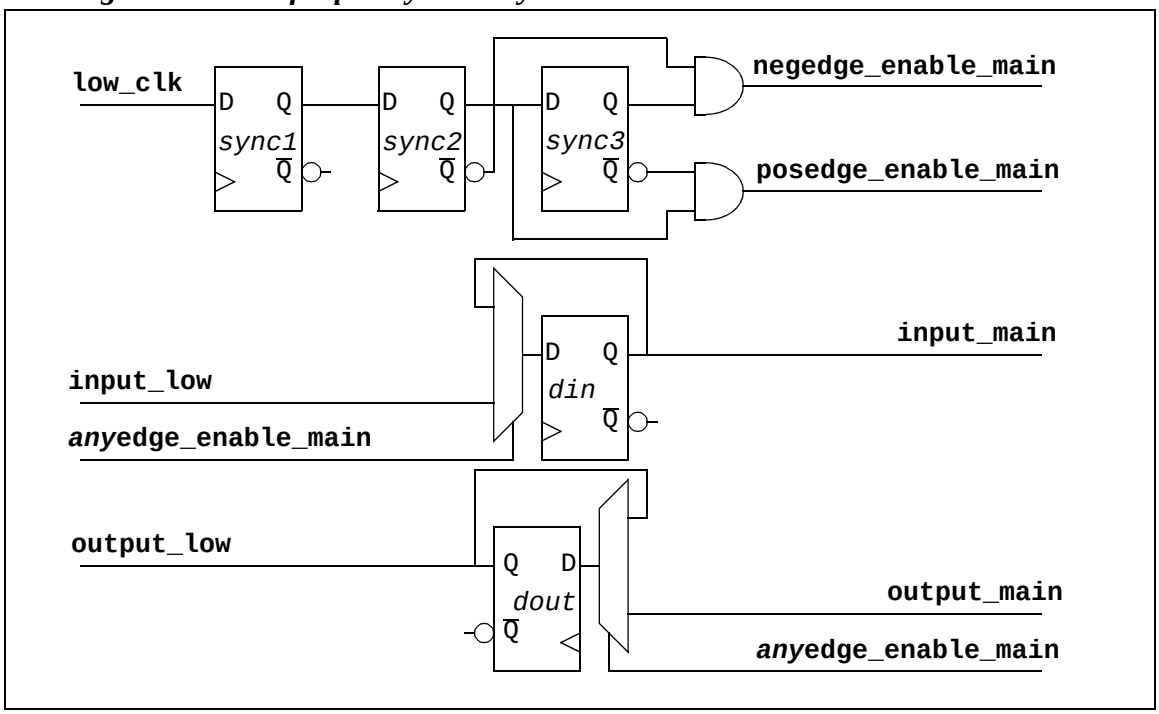


Figure 2 Timing diagram

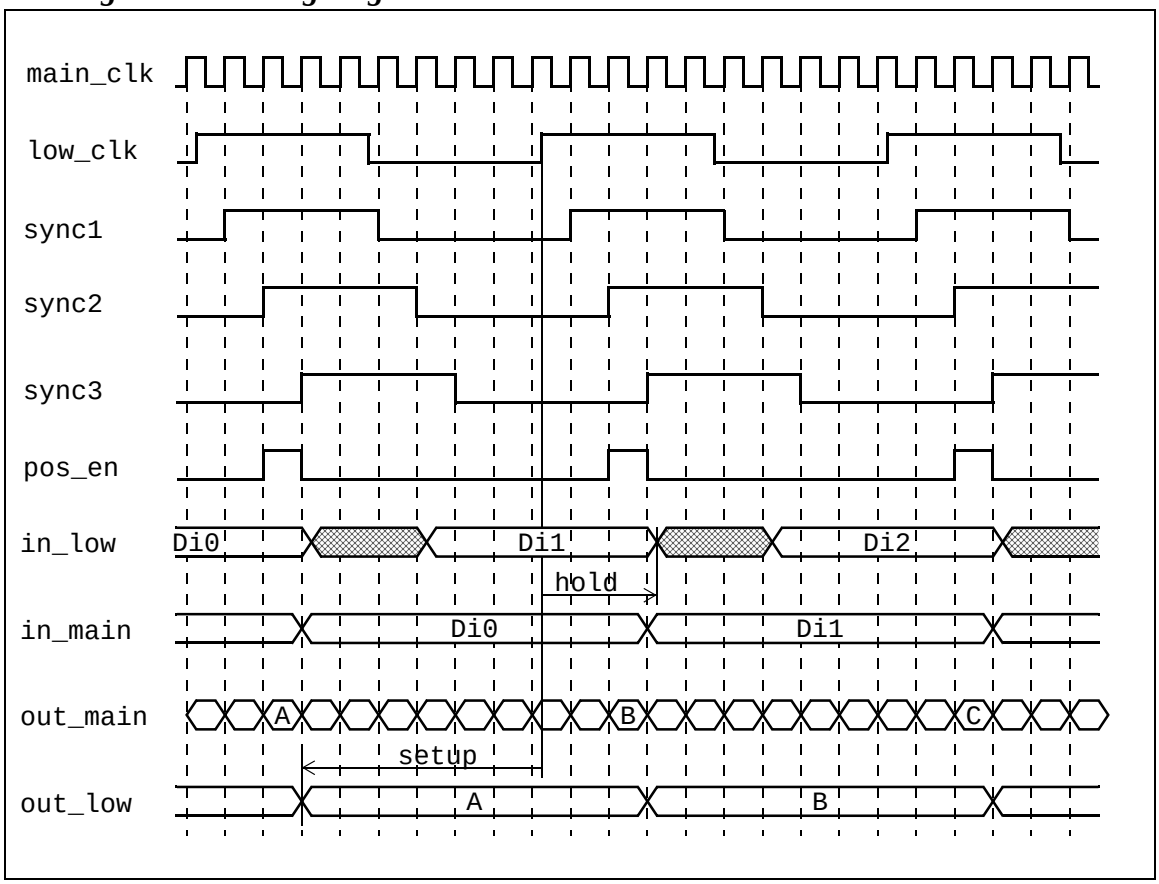


Figure 3 Verilog code

```
// inputs & outputs
input main_clk, main_rst_b, output_main;
output input_main;
input low_clk, input_low;
output output_low;
// FFs declaration
reg sync1, sync2, sync3;
reg din, dout;
// low_clk edges detection
wire posedge_enable_main = sync2 & !sync3;
wire negedge_enable_main = !sync2 & sync3;
// input & output
wire input_main = din;
wire output_low = dout;
// synchronization logic
always @(posedge main_clk or negedge main_rst_b)
    if (!main_rst_b)
        begin
        sync1 <= 0;
        sync2 <= 0;
        sync3 <= 0;
        din <= 0;
        dout <= 0;
        end
    else
        begin
        sync1 <= low_clk;
        sync2 <= sync1;
        sync3 <= sync2;
        if (posedge_enable_main)
            begin
            din <= input_low;
            dout = output_main;
            end
        end
```

2.3 Implementation requirements

The first sampling flip-flop (*sync1*) drastically reduces the risk of metastability propagation since its output has a full cycle to resolve to one or zero until it is sampled by the next flip-flop (*sync2*). That system assumes the sampled signal edges are sharp enough to prevent more than one metastability occurrence per edge because there would be difficulty detecting several edges.

The first stage should be removed if it can be guaranteed the flip-flop outputs will not become metastable if their D input changes during the capture window. That helps reducing the latency between the edge and its detection as an enable signal.

Another means to achieve better efficiency is the use of both clock edges for the synchronizer: a negedge flip-flop for *sync2* further reduces the detection latency. However, that last possibility should be reserved to cases when the posedge solution cannot be used, as using both edges negatively impacts the DFT, dual phase makes the design more complex and the enable is only available less than a half-cycle before the next rising edge.

The minimum sampling frequency is determined by the hold time of the input data from

the low clock reference edge and by the setup time required for the output data. The calculation for both constraints is detailed below and an application example is also proposed.

2.3.1 Hold time constraint

The worst case equation assumes the low clock reference edge happens at the exact end of the capture violation window and the sampling flip-flop state remains at the value before the edge.

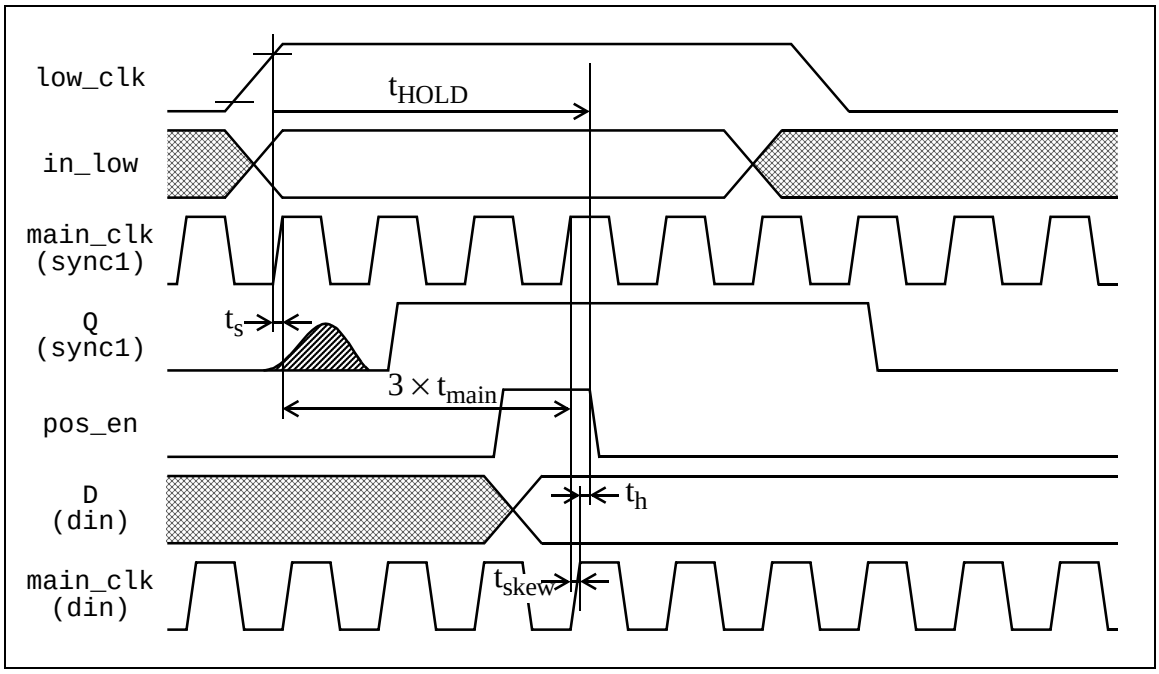
$$t_{HOLD} > 3 \times t_{main} + t_{ph} + t_{skew} + t_h + t_s$$

which becomes

$$t_{main} < (t_{HOLD} - t_{ph} - t_{skew} - t_h - t_s)/3$$

where t_{main} is the main clock maximal cycle duration, t_{ph} is the main clock maximal phase shift between two successive cycles, t_{skew} is the clock tree maximal skew, t_h and t_s are respectively the hold time of *din* and the setup time of *sync1*.

Figure 4 t_{HOLD} calculation



If the library flip-flop outputs cannot become metastable, the removal of *sync1* translates into the following equation:

$$t_{main} < (t_{HOLD} - t_{ph} - t_{skew} - t_h - t_s)/2$$

And using negative edge-triggered flip-flop for *sync2* respectively gives

$$t_{main} < (t_{HOLD} - t_{ph} - t_{skew} - t_h - t_s)/2.5$$

when *sync1* is kept and

$$t_{main} < (t_{HOLD} - t_{ph} - t_{skew} - t_h - t_s)/1.5$$

when *sync1* is removed.

As far as the input data is setup before the reference edge of its clock, it does not generate any particular constraint to the system. If this is not the case, there is a maximum sam-

pling frequency. It is however very easy to adapt the system by delaying the sampling.

2.3.2 Setup time constraint

The same type of calculation is performed to determine the frequency constraint derived from the required setup time of output data.

$$t_{SETUP} < t_{low} - t_{phlow} - 3 \times t_{main} - t_{skew} - t_s - t_{ck2q}$$

which becomes

$$t_{main} < (t_{low} - t_{phlow} - t_{SETUP} - t_{skew} - t_s - t_{ck2q})/3$$

where t_{low} is the low clock minimal cycle duration, t_{phlow} is the low clock maximal phase shift between two successive cycles, t_{main} is the main clock maximal cycle duration, t_{skew} is the clock tree maximal skew, t_{ck2q} and t_s are respectively the clock-to-Q delay of *dout* and the setup time of *sync1*.

If the library flip-flop outputs cannot become metastable, the removal of *sync1* translates into the following equation:

$$t_{main} < (t_{low} - t_{phlow} - t_{SETUP} - t_{skew} - t_s - t_{ck2q})/2$$

The output data hold constraint can create a maximum sampling frequency constraint if it is not null; it is easy to solve that issue by further delaying the *dout* update.

2.3.3 Example

Let us consider a JTAG interface with a 20 MHz TCK (50 ns cycle), 0.5% frequency drift, 1% phase shift, 10 ns setup required for TDO, and 40 ns hold for TDI and TMS.

The flip-flops have the following characteristics: $t_h = 0.1$ ns, $t_s = 0.4$ ns, $t_{ck2q} = 0.6$ ns.

The sampling clock worst case parameters are 0.5 ns skew, 0.5% phase shift and 0.3% frequency drift. We are looking for its maximal cycle delay t_{main}.

- the t_{HOLD} constraint gives:
  $40 > (3 + 0.5/100) \times t_{main} + 0.4 + 0.1 + 0.5$
  $t_{main} < 39.0/3.005$
  $t_{main} < 12.98$ ns
- and the t_{SETUP} constraint:
  $50 \times 0.995 - 50 \times 1/100 - 0.4 - 0.6 - (3 + 0.5/100) \times t_{main} - 0.5 > 10$
  $t_{main} < 12.56$ ns
- taking the tighter constraint and considering the frequency drift:
  $t_{main} < 12.56/(1 + 0.3/100)$
  $t_{main} < 12.52$ ns (79.8 MHz minimum)

So, the main frequency clock must be about 80 MHz to safely oversample the JTAG TCK. If the flip-flop outputs do not become metastable and *sync1* flip-flop is removed, the sampling frequency drops to 54 MHz.

If *sync2* is implemented as a negative edge-triggered flip-flop, the main clock frequency limits respectively become 67 MHz and 40 MHz.

2.4 Pros & Cons

Why is it so interesting to use that mechanism?

- That solution is safe, provided the sampling frequency constraints are met, which is easily determined by calculation as it was shown above.
- The design is simplified, which impacts all the stages; coding, verification, synthesis, timing analysis (dynamic analysis is no more necessary), layout and especially clock tree synthesis. It is merely impossible to find a simpler solution with the same level of safety; as soon as you introduce a new clock domain in a design, there needs a synchronization mechanism somewhere. The proposed solution has only one in the simplest form; there cannot be less.
- DFT compliancy is preserved. There is no need for additional logic to circumvent multiple clock issues, which tends to add complexity and area to the design. And over 99.5% of fault coverage can be achieved, which is a pipe-dream when you have to rely on functional patterns because some sections could not be easily made testable.

But there are limitations and drawbacks:

- There is not complete freedom in the clock ratio: the sampling clock frequency is constrained, thus preventing the use of that system in all cases.
- Synthesis is performed with tighter constraints; that causes an increase of the module area; it may be sometimes difficult to meet the timing goal and multi-cycle paths constraints should be considered. The area impact is further evaluated in the following paragraph.
- Power consumption? This is the most common objection to the principle of synchronizing the input clock as if it were data. As soon as you assume there is a synchronizer somewhere in the design, that part of the mechanism requires the same amount of power regardless of its location; the rest of the design takes advantage of existing clock-gating techniques to reduce power consumption to about the same level as systems that use the incoming clock as a clock. That assumption is also studied in the next paragraph.

2.5 Power and area impact

The study was performed on two designs. Several variations of the designs have gone through power conscious synthesis and power analysis using Design Compiler and Power Compiler.

The exact flow was taken from the Power Compiler user guide:

- RTL simulation to generate RTL activity file.
- Preliminary synthesis with clock gating insertion.
- Incremental compile with power optimization using backannotated RTL activity.
- Gate-Level simulation to generate post-synthesis activity file.
- Power analysis of backannotated GL activity. It was done for three running modes; in ‘sleep’ mode, there is no toggle on any low frequency input pin

including the clock; in ‘idle’ mode, the low frequency clock is running but the inputs are left idle; in ‘run’ mode, the system is stimulated by its low frequency inputs.

The design variations that went through the flow:

- Dual-clock design with minimized synchronization. The first clock is the input low frequency reference, the second clock is the main chip clock. There is a single synchronizer in the module. That design is taken as the reference in terms of power consumption.
- Dual-clock design with large synchronization. It is the same as above except several synchronizers are instantiated; the objective was simply to measure the impact on power when synchronizers are multiplied, as it is not guaranteed that a single synchronizer dual-clock solution is always achievable.
- Single-clock design with Power Compiler automatic clock gating.
- Single-clock design with two levels of clock gating; the first level was instantiated in the Verilog code, using the enable generated from the low frequency clock positive edge detection to gate the whole design logic except the synchronizer itself (the Verilog code was thus different from what is proposed in “Clock sampling mechanism” on page 4). The second level was automatically generated by Power Compiler.

It is important to understand that inserting clock gating makes the Gate-Level structure of the synchronization mechanism slightly different from what is presented in “Clock sampling mechanism” on page 4. The synchronizer and edge detection remain unchanged, but *din* and *dout* registers receive a clock gated by the enable generated from the synchronizer and the multiplexer at their input is removed. Power Compiler should automatically perform that modification.

Table 1: Power and area impact study of low frequency clock synchronization

design		**avg (µW)**	**sleep (µW)**	**idle (µW)**	**run (µW)**	**area (gates)**
A	dual-clock (1 sync.)	19.55	13.94	15.56	23.35	639
	dual-clock (5 sync.)	35.91	30.19	31.81	39.80	710
	single-clock (1 level)	39.04	28.82	33.17	41.87	723
	single-clock (2 levels)	19.61	13.90	15.52	23.59	646
B	dual-clock (1 sync.)	12.51	10.14	13.59	13.81	729
	dual-clock (3 sync.)	12.63	10.20	13.65	14.06	762
	single-clock (1 level)	26.13	23.75	27.20	27.43	731
	single-clock (2 levels)[a]	12.46	10.00	13.57	13.80	724

a. Manually instantiated clock-gating cells had to be put in a module wrapper because Power Compiler was unable to understand the cell logic when there was already clock-gating logic and thus refused to add another level of clock gates.

The major issue when designing the dual-clock solution was the capture of data clocked at the main frequency; it required a synchronized hand-shaking mechanism. As soon as several such synchronizers are used, not surprisingly, the power consumption increases significantly. It is interesting to notice that three synchronizers were necessary for functional safety in design B; the single synchronizer solution was only implemented for power comparison purpose. However, both additional synchronizers are clocked at the low frequency, so they have very little impact on the power consumption.

The single-clock solution, which only relies on Power Compiler for the clock gating, clearly shows the limits of the tool. As it is only able to generate one level of clock-gating cells, the enables are combined together to give a specific clock enable for every register bank (the size limit was set to eight). That logic toggles frequently, which costs power. Such a solution cannot compete for power or area with the dual-clock solution, even when the dual-clock solution uses multiple synchronizers.

Since Power Compiler is unable to manage several levels of clock gating, a first level is manually instantiated in a separate wrapper (see footnote a from table 1); it takes the enable active when the low-frequency-clock edge was detected to gate off all the flip-flops of the design. That makes it become similar to the dual-clock version with one synchronizer; the synchronizer is shifted at the low frequency clock port, one clock-gating cell drives the rest of the design and its enable is active when the low frequency clock rising edge is detected; which means all the design registers, except the synchronizer, receive the exact same amount of clock edges as the registers in the dual-clock version. The single-clock power consumption and area are thus similar to those achieved by the dual-clock with one synchronizer. The layout and clock tree impact was not evaluated; *a priori*, it can favor any solution. The DFT hardware cost was not considered either, but the dual-clock solution is necessarily more complex, thus requiring more DFT logic than the single-clock solution (e.g. clock muxes, etc.).

3.0 Low frequency clock generation

After considering the case when the design receives data and its reference clock, let us focus on the case when the low frequency reference is generated internally and does not need to be output. Typical examples are SSI, USB, etc.

The clock generators are often directly derived from the main reference clock PLL and sometimes from another PLL, using clock dividers. That creates several asynchronous clocks within the design. The drawbacks are similar to those listed in the previous paragraph; asynchronous clocks make verification, synthesis and timing analysis more complex; every additional clock requires specific design constraints and clock-tree synthesis; testability is lower because of the ATPG limitations.

It will sometimes be possible to implement a synchronous mechanism that yields the desired low frequency reference from the main clock. All the previous drawbacks are removed, but that kind of solution has its own limitations that will be analyzed. The kind

of synchronous mechanism proposed below should be the selected solution whenever possible (i.e. when the low reference constraints, such as the jitter or the frequency drift, are respected).

3.1 Mechanisms

Two mechanisms are proposed; the first one performs a simple clock division, whereas the second one can also be used to generate a fractional clock.

3.1.1 Synchronous frequency divider

When the desired frequency is a whole division of the main clock frequency, a simple clock divider can be used to generate an enable signal at the desired frequency. That enable is then used as a synchronous input or a clock-gating enable by all the flip-flops that need to run at the low frequency.

The exact mechanism consists of a counter modulo *n* clocked at the main frequency, where *n* is the frequency ratio. It can be initialized to any value, zero is usually a good choice. The enable is generated whenever the counter value is zero.

Ideally, the enable is used to synchronously enable the update of all the design flip-flops that have to run at the low frequency. The the clock gating is added by Power Compiler. Unfortunately, the current tool limitations may require a manual instantiation of the clock-gating cell for optimal performance.

Figure 5 Timing diagram (divide by 5 example)

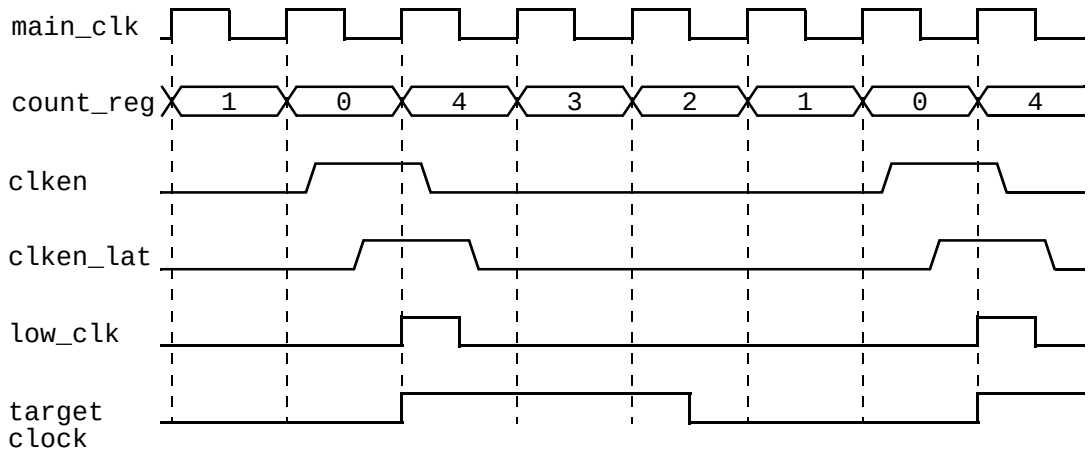


Figure 6 Enable generation Verilog code example

```
// counter declaration
parameter n = 2;
parameter divider = 5;
reg [n:0] count_reg;
// clock enable
wire clken = (count_reg == 0)? 1'b1 : 1'b0;
// counting
always @(posedge main_clk or negedge main_rst_b)
```

```
    if (!main_rst_b)
        count_reg <= 0;
    else if (count_reg == 0)
        count_reg <= divider - 1;
    else
        count_reg <= count_reg - 1;
```

Figure 7 Enable usage as synchronous input

Power Compiler builds the clock gating.

```
always @(posedge main_clk or negedge main_rst_b)
    if (!main_rst_b)
        <registers are reset here>
    else if (clken)
        <registers are updated here>
```

Figure 8 Enable usage via a clock-gating instance

```
wire low_clk;
clkgate low_clk_gen (
    .clkin(main_clk),
    .clken(clken),
    .clkout(low_clk));
always @(posedge low_clk or negedge main_rst_b)
    if (!main_rst_b)
        <registers are reset here>
    else
        <registers are updated here>
```

3.1.2 Synchronous frequency fraction generator

When the main clock frequency is not an exact multiple of the desired low frequency, there always exists a fraction that is close enough to the exact ratio. ‘Close enough’ means the error between the fraction and the ratio is lesser than a predetermined limit. Any limit can be chosen; there is always a solution since $\wp$ is dense in $\Re$. See below “Structural error due to the counter itself” on page 17 for the error calculation.

The fraction consists of a numerator and a denominator. The proposed solution works with a single counter that is decremented by the numerator modulo the denominator every cycle. Every time the counter wraps around its modulo value, the enable is activated.

The global frequency of the enable is exactly the fraction of the main clock frequency. Its jitter is one main clock cycle. If the numerator is a divider of the denominator (usually one since the fraction will be reduced), the system behaves like the divider and the enable is generated whenever the counter reaches zero.

As already mentioned, the major drawback of that solution is the jitter, or the phase error between the generated pseudo-clock and the ideal sub-frequency clock, which can be up to one main clock cycle.

A timing diagram example and the Verilog model are given below.

Figure 9 Timing diagram (multiply by 3/11 example)

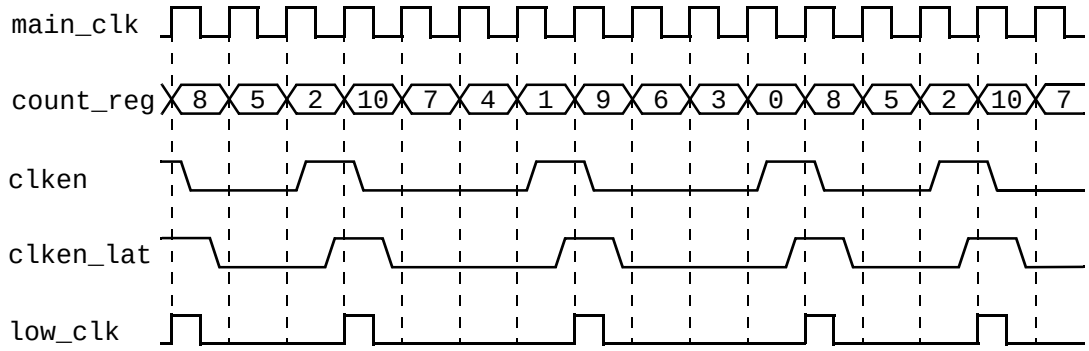
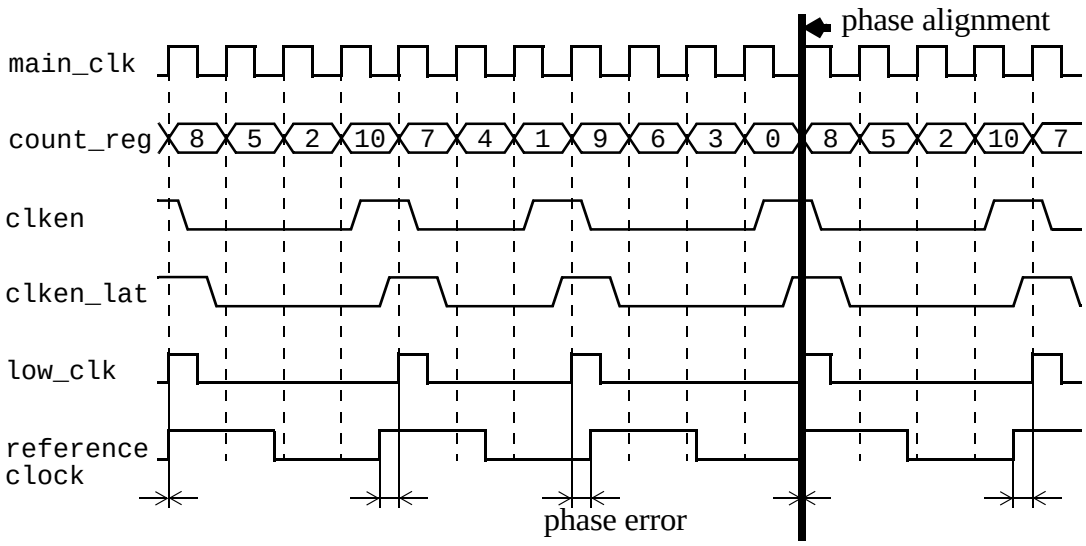


Figure 10 Fraction generator Verilog model

```
// counter declaration
parameter n = 3;
parameter numerator = 3;
parameter denominator = 11;
reg [n:0] count_reg;
reg [n:0] count_next;
// clock enable
reg clken;
// counter update
always @(posedge main_clk or negedge main_rst_b)
    if (!main_rst_b)
        count_reg <= denominator - 1;
    else
        count_reg <= count_next;
// counting
always @(count_reg)
    begin
    clken = 0;
    if (count_reg < numerator)
        begin
        count_next = count_reg + denominator - numerator;
        clken = 1;
        end
    else
        count_next = count_reg - numerator;
    end
```

Although the jitter between two successive enables is always zero or one main clock cycle, it is possible to adjust the phase error between the low frequency clock generated by the system and an ideal low frequency clock with a known phase. Starting from zero, the counter has a cyclic succession of values that goes through value zero periodically. It is possible to determine the counter values that trigger the enable in such a way that the phase error between the generated clock and the reference is always lesser than one half the main clock cycle. This is shown in the following timing diagram.

Figure 11 Timing diagram (multiply by 3/11 example with reduced phase error)



3.1.3 Dual edge-triggered synchronous frequency fraction generator

It is possible to halve the system jitter by using a dual phase system.

That solution is more complex and its advantages are less numerous; it should only be considered when the single phase solution is not applicable. It is however a better choice than having another PLL on-chip.

It consists of the same counter as the previous solutions, except it generates two different enables; one to gate the main clock in order to generate the positive sub-clock and the other to gate the inverted main clock in order to generate a negative sub-clock. Both sub-clocks are then ORed together to yield the desired low reference clock. The principle is described with the timing diagram and Verilog model below.

Figure 12 Timing diagram (multiply by 3/11 for dual edge-triggered system)

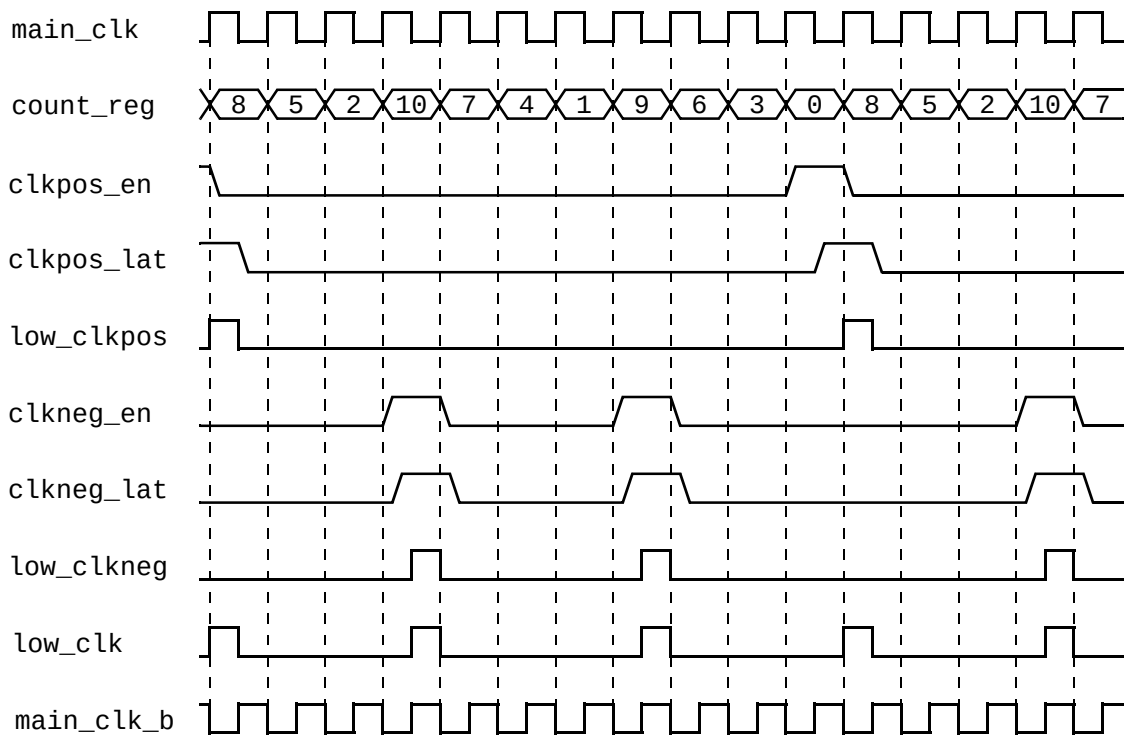


Figure 13 Dual edge-triggered fraction generator Verilog model

```
// counter declaration
parameter n = 3;
parameter numerator = 3;
parameter denominator = 11;
reg [n:0] count_reg;
reg [n:0] count_next;
// clock enable
reg clkpos_en;
reg clkneg_en;
// counter update
always @(posedge main_clk or negedge main_rst_b)
    if (!main_rst_b)
        count_reg <= denominator - 1;
    else
        count_reg <= count_next;
// counting
always @(count_reg)
    begin
    clkpos_en = 0;
    clkneg_en = 0;
    if (count_reg < numerator)
        begin
        count_next = count_reg + denominator - numerator;
        if (count_reg < numerator/2)
            clkpos_en = 1;
        end
    else
```

```
        begin
        count_next = count_reg - numerator;
        if (count_reg >= denominator - (numerator+1)/2)
            clkneg_en = 1;
        end
    end
// clock-gating cells
wire low_clkpos;
clkgate low_clkpos_gen (
    .clkin(main_clk),
    .clken(clkpos_en),
    .clkout(low_clkpos));
wire low_clkneg;
clkgate low_clkneg_gen (
    .clkin(~main_clk),
    .clken(clkneg_en),
    .clkout(low_clkneg));
wire low_clk = low_clkpos | low_clkneg;
```

The timing constraint for the logic that generates `clkneg_en` is one half the main clock cycle whereas it is a full cycle for `clkpos_en`. Moreover, the datapaths between the low frequency clock and the main clock are constrained to one half the main clock cycle length. Finally, the low frequency clock is rather tricky and has a complex logic on it, so clock tree synthesis as well as ATPG may prove to be difficult tasks.

3.2 Implementation requirements

The discussion is based on the positive edge-triggered fraction generator. The divider is a particular case of the fraction. The dual edge-triggered system has the same limits as the positive edge-triggered one running twice as fast.

3.2.1 Structural error due to the counter itself

For a desired clock ratio r and an m-bit counter, two cases occur:

- $r \in \wp$, r reduced expression is n/d and $d \leq 2^m$; the structural frequency drift $fd_{struct} = 0$ as the exact clock ratio can be generated;
- $r \notin \wp$, or $d > 2^m$; the structural frequency drift $fd_{struct} \leq 2^{-(m+1)}$ as the counter granularity is 2^{-m} so abs(“desired ratio” - “nearest possible ratio”) $\leq 2^{-(m+1)}$.

Once the best clock ratio has been selected, its value can be expressed as n/d with $d \leq 2^m$.

The jitter between two successive clock edges depends on n:

- if $n \neq 1$, its value is sometimes 1, sometimes 0 main clock cycle ($j_{struct} \in \{0,1\}$);
- if n = 1, which is the divider case, there is no jitter ($j_{struct} = 0$).

Jitter is by far the most limiting factor.

When considering the phase error between the generated clock and an ideal low frequency reference, we have seen it can be brought to less than half the main clock cycle.

3.2.2 Additional error due to the implementation limits

The minimal frequency for the main clock depends on the tolerance parameters for the generated clock. Those parameters are either explicit from the specification document when the low frequency clock itself is characterized, or they can be deduced from the constraints on the signals computed using that generated clock.

The resulting frequency drift and jitter depend on the structural error and on the main reference clock characteristics:

- $fd_{tot} \leq fd_{main} \times fd_{struct}$
- $j_{tot} \leq j_{main} + j_{struct} + j_{logic}$

where j_{logic} represents the maximal jitter due to the combinatorial logic that generates the low frequency output signals. Since that parameter is implementation dependent (combinatorial structure and layout) and relatively small compared to the other two parameters, it will be neglected in the following example.

3.2.3 Example

Let us calculate the minimal frequency of the main clock in several application cases. The structural jitter is the only parameter that will be taken into account; since this is the main system limitation, it allows a quick feasibility evaluation. Once the mechanism structural jitter is proved to respect the system tolerance, a further calculation based on refined parameters can be performed to draw the exact limitations of the mechanism.

- RS232 interface running at 28.8 kbauds, maximal distortion is 2%:
  the character average duration is
  $$char_{dur} = 1/28{,}800 = 34.7\ \mu s$$
  and thus the maximal allowable structural jitter is
  $$j_{struct} = 34.7 \times 0.02 = 694.4\ ns$$
  which gives the minimal frequency of the main clock
  $$f_{main} = 1.44\ MHz$$
- RS232 interface running at 115.2 kbauds with a maximal distortion of 1% gives, with a similar calculus:
  $$f_{main} = 11.52\ MHz$$
- Low speed USB interface running at 1.5 MHz with a maximal jitter of 10 ns:
  the minimal frequency is thus
  $$f_{main} = 100\ MHz$$
  which gets close to common peripheral frequencies; a dual edge-triggered solution might be necessary; the frequency requirement could then drop to
  $$f_{main} = 50\ MHz$$
  It is however possible to eliminate that issue by constraining the peripheral frequency to be an exact multiple of 1.5 MHz which removes the structural jitter.
- High speed USB interface running at 12 MHz with a maximal jitter of 1 ns:
  the minimal frequency reaches 1 GHz or 500 MHz for the dual edge-triggered solution, that are totally unacceptable running frequencies for peripherals. The

only solution to avoid any synchronization issue in the peripheral is to have the main frequency be a multiple of 12 MHz. Again, that constraint can be difficult to obtain and the module will require multiple clocks.

3.3 Pros & Cons

The advantages when using that type of mechanism are similar to those described in “Low frequency clock reception” on page 2 plus a couple more:

- Design simplification (coding, verification, synthesis, static timing analysis, clock tree synthesis, etc.).
- DFT-friendly solution.
- Synchronizer-free, since the whole design flip-flops receive gated clocks derived from the exact same base.

Drawbacks are also similar:

- Tighter constrained synthesis because the whole design receives the same reference clock; it is however possible to use multi-cycle paths constraints when timing is hard to meet, or when the impact on the design size cannot be disregarded.
- The system is not always able to provide an exact target frequency match.
- Jitter limits the range of applications to low frequency clocks or exact frequency division of the main clock.
- Power consumption impact is more complex to evaluate; it is discussed in the following section.

3.4 Power impact discussion

Power consumption comparison has to be considered for several aspects; some of them depend entirely on the design particularities (floorplan, clock tree constraints like the insertion delay, etc.) and thus need to be evaluated on a case basis to decide about the applicability of the proposed synchronous mechanism.

3.4.1 Low frequency running logic

Power consumption for the logic running at the low frequency is equivalent regardless of the system selection (low frequency clock or pseudo-clock generation with clock gating) as it was seen in “Low frequency clock reception” on page 2.

3.4.2 Clock tree

The pseudo-clock generation presents the advantage of a single clock routed between the clock controller and the module, instead of several clocks. However, the power gain mainly depends on the way clocks are gated on and off. When comparing the pseudo-clock generation system with a single high frequency clock input and the classical system with two clock inputs (the high frequency clock being a bus clock), several factors have to be taken into account:

- The activity of the low frequency clock that is routed to the module and thus

the average power consumption of its clock tree up to the module interface; that parameter only accounts for the dual-clock implementation.

- The activity of the high frequency clock that is routed to the module; when this is the peripheral bus clock, it is usually running according to the CPU power modes or, if its clock gating is smarter, when the CPU accesses one of the peripherals (so it is not necessarily the sole activity due to the peripheral we are considering). That parameter accounts for both solutions.
- The additional activity of the high frequency clock when the pseudo-clock generator is implemented in the module. That parameter only accounts for the pseudo-clock generator solution. However, if the bus clock is usually running when the module is working, there is little additional activity to be expected, which makes the pseudo-clock generator the best choice.

Let us consider a simple example with a peripheral that receives a bus clock and also needs a low frequency clock:

- the low frequency clock is running at 1 MHz and its clock tree consumes 1 µW/MHz;

- the bus clock is running at 90 MHz and the clock tree to the module consumes 5 µW/MHz when it is active, which happens 100% of the time in the first case and 15% of the time in the second case (smart clock management from the CPU).

In the first case, it is clear that the pseudo-clock generator solution brings a (very) little power consumption improvement of 1 µW.

In the second case, the dual-clock solution consumed power is $P = 1 + 90 \times 5 \times 0.15 = 68.5$ µW whereas it becomes $P = 90 \times 5 = 450$ µW for the pseudo-clock generation.

The degradation shown above also needs to be compared against the overall power consumption of the module.

3.4.3 Low frequency generator

A test case evaluation is proposed below with variations of the counter width for the pseudo-clock generator. In the case of a real clock generation, many different mechanisms are possible with as many varying power consumptions: crystal, PLL with a frequency divider, etc. Two dual-clock solutions using a clock divider from a 500 MHz PLL and from a 40 MHz PLL are also evaluated.

The main clock frequency is 42.63 MHz; the low frequency clock target is 1 MHz. The single clock solutions are implemented with both manual and Power Compiler automatic clock gating.

Table 2: Clock fraction generator evaluation

design	avg (µW)	sleep (µW)	idle (µW)	run (µW)	area (gates)
dual-clock (the reference)	5.86	4.47	6.35	6.76	751

Table 2: Clock fraction generator evaluation

design	avg (µW)	sleep (µW)	idle (µW)	run (µW)	area (gates)
dual-clock with 40 MHz clock divider	12.64	5.10	16.20	16.61	878
dual-clock with 500 MHz clock divider	96.67	14.80	137.4	137.8	939
single-clock (8-bit counter)	17.38	4.26	12.08	35.81	948
single-clock (12-bit counter)	26.56	4.26	15.77	59.65	1,173
single-clock (16-bit counter)	37.54	4.26	20.02	88.34	1,412

Sleep mode occurs when the main clock is shut off; in idle mode, clocks are running but the module is not stimulated; and in run mode, the module is working.

The table clearly shows the impact of the counter size. There is an obvious tradeoff between flexibility and accuracy on one side, area and power on the other side. The reference dual-clock design consumes far less power than single-clock solutions since it does not include any frequency divider. In the usual case when the low frequency clock is generated from a PLL, a clock divider is used to provide the module with its low frequency clock; two cases have been evaluated with the simple addition of a clock divider. The 500 MHz clock divider takes the direct output of the PLL while the 40 MHz clock divider is equivalent to the single-clock mechanism. Those clock dividers are very low power mechanisms optimized for one ratio and they do not respect clean synchronous design methodologies (output of flip-flop used as clock of next one). The results show that opting to single-clock solutions with reduced counter size stands comparison with other methods for power, but those systems outperform for simplicity, backend, integration, DFT, etc.

4.0 Multiple high frequency clocks

When the strategies above cannot be applied because of unacceptable drawbacks (clock ratio granularity, encapsulating IP with no resynthesis, too large frequency error, too much jitter, etc.), the design needs several clocks running at different frequencies. That means the use of synchronizers and more complex synthesis constraints.

Below is proposed a safe synchronizer design and synthesis tips to have the flow as smooth as possible.

4.1 Synchronizers

We consider a source and a destination clock domain; the source sends a command with data to the destination domain that sends an acknowledge back. The proposed mechanism follows a set of guidelines to guarantee the design safety and reduce the performance pen-

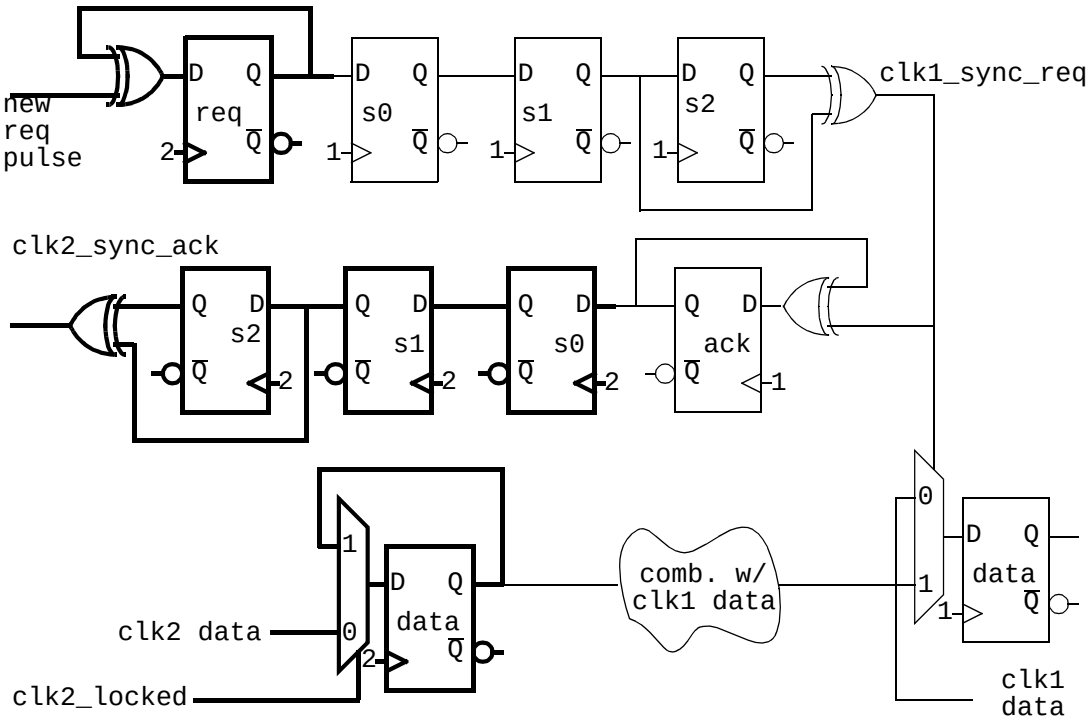
alty:

- There is one synchronized signal that serves as command enable; for a better efficiency, its transition is the enable and both transitions are used.
- The command and data signals are frozen as soon as the enable is toggled and until the acknowledge has been received from the destination domain; those signals can thus be used in the destination clock domain as ordinary data.
- The destination domain detects the transition on the enable; it can use the command and data information from the source domain the same way it uses its own clock domain data.
- The destination domain acknowledges by toggling its acknowledge signal which will release the frozen signals until the next transaction.

That type of mechanism works for all clock ratios; it also automatically adapts to varying clock ratios, which is an interesting property when clock frequencies are often modified.

Synchronizing packets of data can enhance throughput. The synchronization penalty is only paid once per packet, which reduces the average penalty per data but costs more hardware.

Figure 14 Synchronizer schematic



The *s0* stages can be removed if it can be guaranteed the flip-flops will never go metastable when their input changes during the capture window.

The principle of freezing source data allows complex transformations of those data before storing the results in the destination clock domain.

Figure 15 Synchronization timing diagram

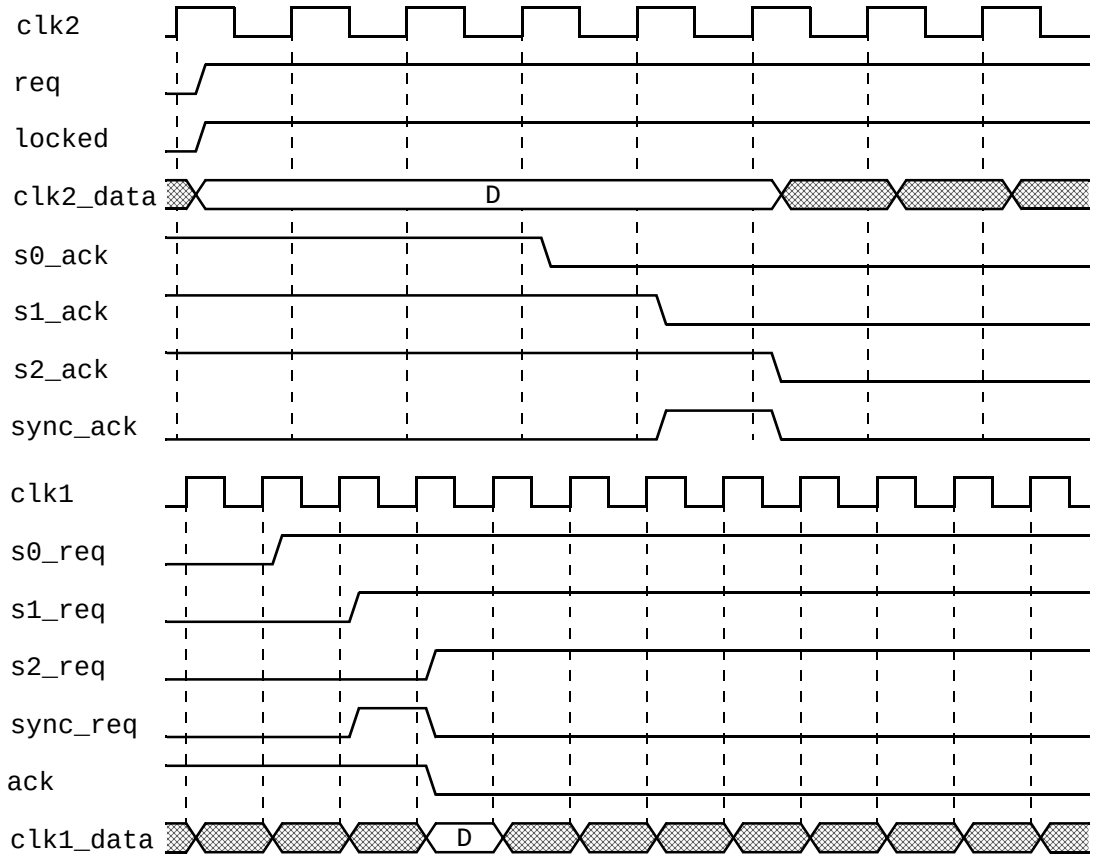


Figure 16 Synchronizer Verilog model

```
reg [n:0] clk1_D;
reg       clk1_ack;
reg       clk1_s0_req;
reg       clk1_s1_req;
reg       clk1_s2_req;
wire      clk1_sync_req = (clk1_s2_req != clk1_s1_req)? 1 : 0;

reg [n:0] clk2_D;
reg       clk2_req;
reg       clk2_synchronizing;
reg       clk2_s0_ack;
reg       clk2_s1_ack;
reg       clk2_s2_ack;
wire      clk2_sync_ack = (clk2_s2_ack != clk2_s1_ack)? 1 : 0;
wire      clk2_D_locked = clk2_synchronizing & ~clk2_sync_ack;

always @(posedge clk2 or negedge clk2_reset_b)
    if (!clk2_reset_b)
        <...>
```

```
    else
        begin
        // synchronizing clk1 acknowledge
        clk2_s0_ack <= clk1_ack;
        clk2_s1_ack <= clk2_s0_ack;
        clk2_s2_ack <= clk2_s1_ack;
        // unlocking clk2_D
        if (clk2_sync_ack)
            clk2_D_synchronizing <= 0;
        // requiring synchronization
        if (<...>)
            begin
            clk2_req <= ~clk2_req;
            clk2_synchronizing <= 1;
            end
        // modification of clk2_D is only possible when
        // no synchronization is ongoing
        if (!clk2_D_locked)
            clk2_D <= <...>;
        end

always @(posedge clk1 or negedge clk1_reset_b)
    if (!clk1_reset_b)
        <...>
    else
        begin
        // synchronizing clk2 side request
        clk1_s0_req <= clk2_req;
        clk1_s1_req <= clk1_s0_req;
        clk1_s2_req <= clk1_s1_req;
        // synchronizing registers and acknowledge
        if (clk1_sync_req)
            begin
            clk1_D <= clk2_D;
            clk1_ack <= ~clk1_ack;
            end
        // clk1_D can also be modified by a clk1 FSM
        else if (<...>)
            clk1_D <= <...>;
        end
```

4.2 Design constraints

Design simplification includes synthesis constraint simplifications. Since several asynchronous clocks cannot be avoided, the synthesis design constraints tend to get very complex with many false paths. False paths may be hard to propagate at chip level when they are numerous; they can hide violations and thus require dynamic timing verification through backannotated gate-level simulations and its well-known coverage.

The simplest solution is to use the same fastest clock frequency for all clocks. That obviously over constrains the design, but the size impact can be relatively negligible if the fastest clock is the main clock of the module and all other slower clocks are only used in

synchronizers. That assumption has been checked with two test cases:

Table 3: Single-clock vs multi-clock synthesis constraints

design	clocks	false paths	area
A	11 (90 → 10 MHz)	72	17,,009 gates
	1 (90 MHz)	0	18,211 gates
B	3 (80, 75, 9 MHz)	9	127,416 gates
	1 (80 MHz)	0	127,479 gates

It was pretty difficult to meet the 90 MHz target with design A and there was a lot of logic running at slower frequency clocks, which led to a 7% area increase.

Design B resembles more the typical case when the tip is best applicable; one main clock running at a high frequency and several interfaces at various frequencies that are directly connected to synchronization mechanisms. Since there is very little logic in the various clock domains except the main one, over constraining does not lead to a larger design size (below 0.05% area increase).

That simple method, when applicable at low cost, presents the advantage of drastically reducing the number of path groups, thus enhancing synthesis runtimes; it also simplifies constraints, thus analysis and reduces the risk of false paths that would hide real violations. However, if the strategy is not applied at chip level, it may still be necessary to write constraints with false paths. By over constraining the design, that method increases the module area and may even cause false timing violations.

To avoid false paths, another possibility is to define all clocks as multiples of each other; instead of defining 100 MHz, 90 MHz and 48 MHz, it is possible to define 100 MHz, 100 MHz and 50 MHz with no false paths. That over-constrains the design less than the previous method, but several clocks are declared.

5.0 One clock issues

The final chapter is only dedicated to address a few issues in the definition of clocks for synthesis. Those problems were encountered in some of our projects and the solutions we found are described below.

5.1 Over constraining a design using both edges of the clock

To over-constrain a design through its clock definition, one usually reduces the cycle time. That technique has a major drawback when both edges of the clock are used (clock-gating latches, particular bus protocols, DDR interfaces, latch-based designs, etc.).

For example, a design target frequency is 10 ns, it is synthesized with a 9 ns cycle time

and the synthesis yields a violation of 0.75 ns; both clock edges are used and some paths last half of the clock cycle (i.e. 5 ns target and 4.5 ns constraint). One of those paths has a violation of 0.7 ns after synthesis and was left unoptimized since its violation is less than the worst constraint violation. Unfortunately, once we switch back to the real cycle time target, there is now a 0.2 ns violation due to that path.

The suggestion is to use the clock uncertainty parameter instead of the clock cycle definition when over constraining the design. By keeping a cycle time of 10 ns and adding 1 ns to the clock uncertainty, all the paths will see the same absolute additional constraint and there won't be any surprises when the original constraint is restored.

5.2 Multiple insertion delays

In the case of manually instantiated clock-gating cells or when late/early clocks are used for synchronous hardmacros (hardIP, memories, etc.), there are multiple insertion delays for the same frequency clock and special care need be taken for paths starting from one insertion delay and finishing at another.

The problem arises when the insertion delay delta is not guaranteed as in the case of clock-gating cells. Let us consider such an example:

The clock-gating cells are instantiated in the middle of the clock tree, which means their latches receive an early clock compared to the clock used by the flip-flops to generate the enable that goes to the latch. With standard synthesis constraints defined at the clock port, the insertion delay seen by the clock-gating latches is equal to the insertion delay seen by the other flip-flops. When the clock tree has been inserted, the clock-gating latches are clocked earlier than the flip-flops that generate the enable, which means the enable has to be stable earlier than synthesis determined. If the enable is fairly complex to generate (i.e. with many levels of combinatorial logic), that can lead to large timing violations that are identified very late in the design cycle.

The solution that was selected requires the definition of multiple clocks running at the same frequency, but with different hook points and insertion delays:

```
create_clock -period $CLOCK_PER -name ROOT_CLK \
         [get_ports "*_clk"]
set_clock_latency  $INSERTION_DELAY [get_clocks {ROOT_CLK}]

create_clock -period $CLOCK_PER -name GATE_CLKOUT \
         [get_pins -hierarchical "*_clk_gate/clkout"]
set_clock_latency $INSERTION_DELAY [get_clocks {GATE_CLKOUT}]

create_clock -period $CLOCK_PER -name GATE_CLKIN \
         [get_pins -hierarchical "*_clk_gate/clkin"]
set_clock_latency 0 [get_clocks {GATE_CLKIN}]
```

The latches inside the clock-gating cells see a null insertion delay whereas the flip-flops that generate the enable that goes to the latch see a normal insertion delay; that over constrains the enable since the insertion to the latch is guaranteed to be greater than zero. If that constraint is too difficult to achieve, the zero value can be replaced by a larger value if timing cannot be met. However, that non-null value becomes a constraint for Clock Tree Synthesis since the clock-gating cell must receive a clock with more insertion delay than the synthesis constraint.

6.0 Conclusion

We have seen several simple mechanisms that can replace complex clocking schemes. They can drastically reduce the design effort in timing analysis, DFT and back-end. Their drawbacks have to be pondered with today's design constraints; in deep sub-micron technology, logic is cheap but time-to-market and bug-free designs are much more challenging than before. If some simplification helps you achieve those goals, do not be concerned too much about slight area increases; you need your design on schedule and with all the features.

Another reason why designs tend to use multiple clocks is the reuse. No designer wants to risk modifying a module that already works, even if that creates a lot of trouble because the module does not follow today's rules (DFT, edge triggered flip-flops, etc.). So, the chip design rules are adjusted and a lot of time is lost from integration to layout through functional verification and DFT because several small modules are sometimes five to ten years old.

Let us stop that. The only case when a designer should not use good synchronous design practice is when there is a real technical improvement (latches for speed, asynchronicity when drawbacks of synchronous systems are unacceptable, etc.), not because of reuse. Reuse is meant to reduce time-to-market, not to cost weeks or even months of additional work to integration, verification, backend and other teams.

Do not be afraid to rewrite completely old designs implemented with out-of-date techniques. They can be replaced by friendly clocking schemes, positive edge-triggered flip-flops, etc. There may be more time spent in solving issues due to old coding styles, than redesigning the whole thing from scratch. It is always possible to reuse the testbench and regression simulations to check the new design does not have more bugs than the old one. And it's more fun writing RTL than struggling with tools on a design you don't understand everything.

A last word about Power Compiler that helps to reduce power consumption of a design; there still needs manual instantiation of clock-gating cells to achieve an optimal power reduction since several levels of clock-gating cells are often needed. The numerous test cases studied when writing this paper showed that getting the best clock-gating cell structure is not an easy task. This is a job for Power Compiler; find common factors in the enables of registers, accept to have flip-flops with feedback loop and clock-gating cell, group them to reach the minimal bank size limit, explore multiple solutions with several levels of clock gating and determine the most efficient one.

7.0 Bibliography

Synthesis and Scripting Techniques for Designing Multi-Asynchronous Clock Designs, Clifford E. Cummings in SNUG San Jose 2001.