UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FISICAS Y MATEMATICAS
DEPARTAMENTO DE INGENIERÍA ELÉCTRICA

CLOCK-GATING FOR LATCH BASED DESIGNS

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL ELECTRICISTA

JOAQUÍN RUBÉN JESÚS FIGUEROA ÁLVAREZ

PROFESOR GUÍA:
VICTOR GRIMBLATT HINZPETER

SANTIAGO DE CHILE
JULIO 2012

UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FISICAS Y MATEMATICAS
DEPARTAMENTO DE INGENIERÍA ELÉCTRICA

CLOCK-GATING FOR LATCH BASED DESIGNS

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL ELECTRICISTA

JOAQUÍN RUBÉN JESÚS FIGUEROA ÁLVAREZ

PROFESOR GUÍA:
VICTOR GRIMBLATT HINZPETER

MIEMBROS DE LA COMISIÓN:
HÉCTOR AGUSTO ALEGRÍA
NICOLÁS BELTRÁN MATURANA

SANTIAGO DE CHILE
JULIO 2012

RESUMEN DE LA MEMORIA
PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL ELECTRICISTA
POR: JOAQUÍN RUBÉN JESÚS FIGUEROA ÁLVAREZ
FECHA: JULIO 2012
PROF. GUÍA: VICTOR GRIMBLATT HINZPETER

CLOCK-GATING FOR LATCH BASED DESIGNS

Digital circuits, whose play a crucial role in everyday life, consume large amounts of power which is undesirable as a general rule, and specially for battery power devices such as cellphones, thus circuit designers and automatic synthesis tools aim to reduce power consumption of such circuits using different techniques.

One of the most successful power reduction techniques is clock-gating used in *Flip-Flop* based designs, which aims to reduce the power consumption caused by the transitions in the **clk** signal. The power reduction is achieved by the insertion of clock-gating cells which keeps the **clk** signal of reaching the *Flip-Flops* when they aren't expected to modify their output signal.

Latch based designs being used less that *Flip-Flop* based design and with additional complexities are still largely used for some benefits of the *Latch* timing restrictions, but no automatic synthesis tool provides an automatic clock-gating insertion feature therefore *Latch* based design circuit designers are forced to perform clock-gating by hand which is far from efficient.

The present work focus on clock-gating and the requirements to allow its use in *Latch* based designs from the automatic synthesis tool perspective, while providing theoretical discussion on the differences between *Latches* and *Flip-Flops* and how these differences force the requirements of a clock-gating insertion engine

Considering the restrictions that should apply for an automatic clock-gating insertion engine focused on *Latch* based designs and using the development environment provided by *Synopsys* as well as the code base existent in the synthesis tool developed by them, a prototype of clock-gating insertion for *Latches* is developed as part of *Design Compiler*®

The prototype embedded in *Design Compiler*® is tested against several small designs created for this purpose, and a larger design provided by a *Synopsys* customer and used in actual circuit development, which allows to test the tool robustness against large designs.

RESUMEN DE LA MEMORIA
PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL ELECTRICISTA
POR: JOAQUÍN RUBÉN JESÚS FIGUEROA ÁLVAREZ
FECHA: JULIO 2012
PROF. GUÍA: VICTOR GRIMBLATT HINZPETER

CLOCK-GATING FOR LATCH BASED DESIGNS

Los circuitos digitales, que juegan un papel crucial en la vida cotidiana, consumen grandes cantidades de potencia lo que es considerado como una situación no deseada, lo que es particularmente cierto para equipos que dependen de baterías como celulares, es por esto que los diseñadores de circuitos así como las herramientas de síntesis utilizan diferentes técnicas con el fin de reducir su consumo de potencia.

Una de las técnicas de reducción de potencia mas exitosas es *clock-gating* cuyo objetivo es reducir el consumo de potencia generado por las transiciones debidas a la señal de **clk** . La reducción de potencia se logra mediante la inserción de *clock-gating cells*[1] que impiden que la señal de **clk** llegue a los *Flip-Flop* cuando el valor de la salida de estos no se espera que cambie.

Los diseños basados en *Latch,* que si bien no son tan utilizados como los diseños basados en *Flip-Flop* debido a sus complejidades adicionales, todavía son utilizados gracias a ciertos beneficios que presentan las restricciones de *timing*[2] de los *Latch,* sin embargo ninguna de las herramientas de síntesis existentes permite la inserción automática de *clock-gates* para diseños basados en *Latches,* por lo que los diseñadores de circuitos se ven forzados a insertar las *clock-gates* de forma manual lo que es ineficiente.

El presente trabajo se enfoca en los mecanismos de *clock-gating* y los requisitos que se deben cumplir para permitir su uso en diseños basados en *Latches* desde la perspectiva de una herramienta de síntesis, al tiempo que provee de una discusión teórica sobre las diferencias entre *Latches* y *Flip-Flops* y como estas diferencias fuerzan los requerimientos de una herramienta de inserción de *clock-gates*

Considerando las restricciones que debieran aplicar para una herramienta de inserción de *clock-gates* automática enfocada en *Latches* y utilizando el entorno de desarrollo provisto por *Synopsys* así como el código existente en la herramienta de síntesis desarrollada por ellos, se desarrolla un prototipo de inserción de *clock-gates* para *Latches* como parte de *Design Compiler®*

El prototipo una vez embebido en *Design Compiler®* es probado en diversos diseños creados con este propósito y un diseño de mayor envergadura provisto por uno de los clientes de *Synopsys* y que es utilizado durante el desarrollo de circuitos reales, lo cual permite verificar la robustez de la herramienta desarrollada en diseños grandes.

[1] celdas de *clock-gating*
[2] timing o sincronización

A mi familia...

Agradecimientos

A mi familia por su apoyo incondicional.

A mi profesor guía por su ayuda.

A los miembros de la comisión por ayudarme a mejorar la redacción.

A los ingenieros de *Synopsys* por ayudarme en los inicios de la investigación y en especial al grupo de clock-gating que me ayudo a comprender mejor el código para generar el prototipo adecuadamente.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

Digital circuits, a key element in modern life, embedded in almost any device nowadays, are built using *CMOS* transistors for their speed, small power consumption and constantly shrinking size. Usually those transistors are grouped in functional cells in order to take advantage of the higher abstraction level in circuit designs; among those designs sequential circuits are of special interest since they allow complex decision making and signal processing algorithms to be part of the circuit which allow the fastest processing by keeping computations in the hardware instead of using software. This allows the existence of *ASIC*[1] and microprocessors.

Sequential circuit are built using sequential cells which are divided in two mayor families *Flip-Flops* and *Latches. Flip-Flops* are used in most circuit designs for their simpler timing properties allowing more robust designs with less effort, while *Latches* are used in less designs since requires more effort to ensure proper behavior, their less restraining timing characteristics allow faster designs[2].

Unfortunately as complexity and requirements of digital circuits their power consumption rise accordingly thus most circuit designers have to make an additional effort in order to reduce power consumption of digital circuits. Most synthesis tools have embedded power reduction techniques. One of the most used power reduction techniques is clock-gating which currently is available only for *Flip-Flop* based designs

The present work studies the possibility of allowing automatic clock-gating insertion in *Latch* based designs using the *Synopsys* synthesis tool *Design Compiler*® ; to do so this work provide information about *Flip-Flop* and *Latch* based designs, discussed the differences between the sequential elements and uses them to provide support for the clock-gating insertion for *Latches.*

[1] *ASIC* stands for Application specific integrated circuit

[2] Some *Latch* based designs are faster thanks to time-borrowing and usually *Latches* are used in other designs which are inherently faster, which is briefly discussed in chapter 2

Finally a prototype of clock-gating insertion for *Latches* is provided and the results of its application in different designs is discussed.

1.2 Objectives

The present work was defined with specific goals which are now introduced.

1.2.1 General Objective

The general objective for this work is to provide a prototype of automatic clock-gating insertion for *Latch* based designs during the synthesis process of the circuit using the *Design Compiler*® software developed by *Synopsys* .

1.2.2 Specific objectives

- Investigate tools and existent papers about the subject in order to insert clock-gates in *Latch* based designs
- Understand the differences between *Latches* and *Flip-Flops* in order to provide a correct solution for *Latch* based design clock-gating insertion
- Understand or develop theoretical support on the conditions for clock-gating insertion in *Latch* based designs
- Analyze possible problems that might arise in a clock-gating insertion for *Latches*
- Analyze possible clock-gating insertion mechanisms available for *Latch* based designs
- Develop a prototype of clock-gating insertion for *Latch* based designs

1.3 Structure

The present work is divided in several chapters that should be read in sequence in order to better understand it, as each chapter depends on the information provided by the former. Thus the following is a short description of the contents of each chapter.

Literature Review: The literature review chapter, provides insight in the theoretical fundamental and state of art of digital circuits and power reduction techniques. This chapter begins by providing a short introduction to the digital circuits theory to refresh some key concepts needed later, then the description of *Flip-Flop* and *Latch* based designs theory is visited as

the understanding of the key differences between both sequential elements is crucial for the prototype development. The power reduction techniques are discussed next, they provide enough background on the power consumption of digital circuits and the different techniques used to reduce it, here is the clock-gating technique introduced, to later be explained in further detail showing the basic principle and why it's one of the most used techniques to reduce power consumption along with some advanced features available.

Tool Development: This chapter realize the description of the steps to develop a prototype for clock-gating insertion for *Latch* based designs. The chapter starts by making a brief introduction to the current implementation in *Design Compiler*® for *Flip-Flop* clock-gating, this is performed by making a rough deception of the algorithm used to insert *CG* cells, and which parts of the code should be modified. Afterwards the description of the different auxiliary tools developed is made and finally some details on the modifications performed to the existent code are provided.

Analysis of results: This chapter performs the analysis of the results of running the prototype tool in different designs, from small tests used for early verification to larger and more complex designs, analyzing the quality of the obtained results. Also performs a brief description of future work related to the tool and requirements for the prototype to enter to the *Production* status

Conclusions: This is the final chapter of this work, provides the final round up for the different concepts treated here and summarizes the strengths and weakness of this feature.

Chapter 2

Literature Review

2.1 Digital circuits

Electronic circuits can be divided in 2 great families analog and digital circuits. The analog circuits are based on continuous quantities [1]; in this case these kind of circuits are based on voltage, and current so they can be interpreted intuitively by people, this way in a given circuit if the input voltage varies, then it's expected that the current must vary in a related way.

On the other hand, digital circuits are based on discrete quantities so they can't be interpreted with the same degree of intuition as analog circuits, because there is a threshold that must be surpassed in order to observe a difference in the behavior of the circuit given a change in the operating conditions of the circuit, in the same way a light's button must be pressed with a minimum force in order to turn-on or turn-off the light.

From a strict electric point of view there are more important difference between analog and digital circuits, as discussed by Roth on [1] and Floyd on [2]. Digital designs have several advantages over analog designs, being information compression, codification and reliability of the transferred information over a noisy channel, still analog circuits remain unbeaten in high fidelity audio systems where any compression would degrade the received audio quality.

The information compression is an immediate consequence of the quantification of the measured quantities and a sampling process, for example, in an analog temperature measurement scenario the temperature is fully measured continuously, while in a discrete scenario only fractions of the temperature are measured and at given intervals this way if the temperature were to be 35.342178 degrees, the analog circuit would measure the full temperature and a discrete would only measure the 35.3 degrees.

The codification in a digital system allows different ways of transmitting the data; in the temperature example the measured data could be transmitted as a voltage related proportionally

[1] Common examples of continuous quantities are temperature or distance

to the measured temperature (which is called a discrete circuit) or codify the data in a different way, being one of the most common ways using a binary code transmitting a train of pulses directly related to the temperature.

When choosing a binary representation of the discrete data in a quantified circuit it's called a digital circuit, therefore in the following only binary data representation is used whenever a circuit is considered digital; in a digital circuit the following terms are used [3].

Logic level A voltage level that represents a defined digital state in an electronic circuit.

Logic HIGH (or *Logic* 1) The higher of two voltages in a digital system with two logic levels.

Logic LOW (or *Logic* 0) The lower of two voltages in a digital system with two logic levels.

Positive logic A system in which *Logic LOW* represents binary digit 0 and *Logic HIGH* represents binary digit 1.

Negative logic A system in which *Logic LOW* represents binary digit 1 and *Logic HIGH* represents binary digit 0.

The reliability is a consequence of the data codification, by choosing a binary codification where a certain voltage represents the *Logic HIGH* and another the *Logic LOW* , there is a threshold where any signal above a threshold can be interpreted as *Logic HIGH* and any signal below another threshold represents the *Logic LOW* with no mistake implies high tolerance to noise during data transmission, as shown in the figure 2.1

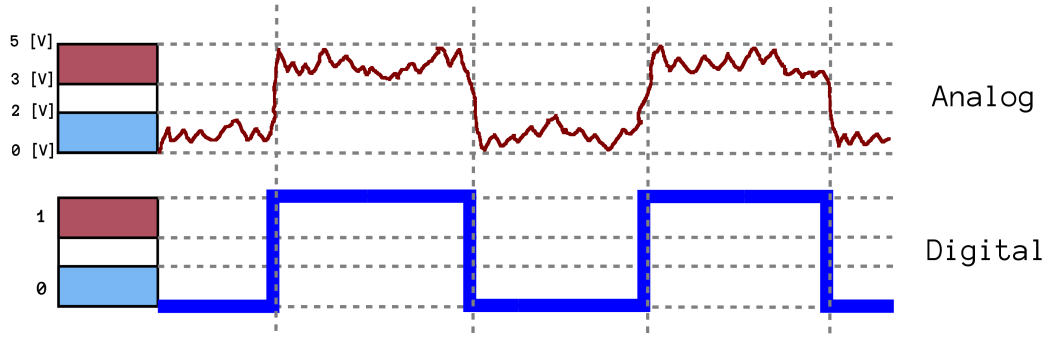


Figure 2.1: The figure illustrates the reliability of a digital circuit. For a given circuit the *Logic HIGH* signal was defined as 5[V] with a threshold of 3[V], meaning that any voltage between 3[V] and 5[V] is recognized as a *Logic HIGH* signal; similarly the *Logic LOW* signal is defined as 0[V] with a threshold of 2[V] meaning that any voltage between 0[V] and 2[V] is recognized as *Logic LOW* .With these thresholds, the analog signal in the upper part of the figure is correctly recognized as the digital signal showed in the lower part of the image regardless of the associated noise in the analog transmission providing the robustness of the digital representation.

There is another advantage in digital circuits, also discussed in [4] is abstraction, which is the key for designing complex systems because all digital systems are built over analog circuits, yet digital circuits aren't designed transistor by transistor but over more complex units like logic gates[2]. The figure 2.2 [4]shows different levels of abstraction in a digital system design based on computer hardware and software.

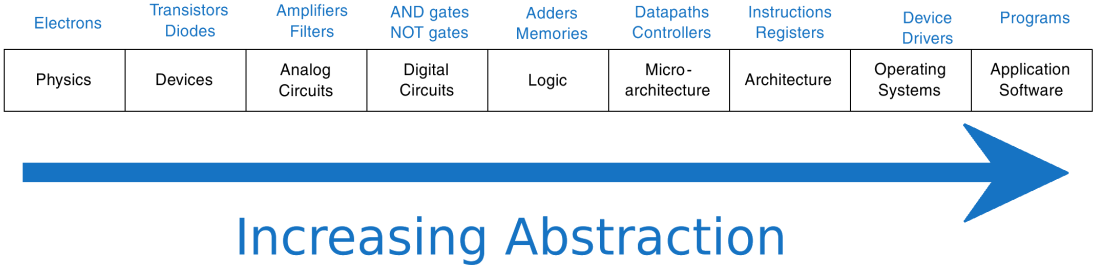


Figure 2.2: This figure illustrates different abstraction levels in digital design [4]

2.1.1 Digital circuit design advantages

From a design perspective, digital circuits have many advantages over analog circuits as discussed on [5]

Reproducible results: For a given set of digital inputs, a digital circuit produces the same digital results every time, in contrast to analog circuits where there is no way to guarantee the same input[3], and even if the exact same inputs are give, the result is not exactly the same because it's sensitive to different noise sources such as temperature

Programmable: Digital circuits can be designed using a programmable approach using HDLs[4] therefore the circuits can be designed using a behavioral (or functional) description which is later translated to the corresponding netlist. Also the netlist can be simulated so the behavior of the circuit is verified prior actual implementation which is done automatically using special hardware and software, while analog circuit design doesn't have such features.

Flexibility and functionality: Digital circuits can be designed to perform specific with high levels of complexity and perform such functions efficiently. Also the same circuit can perform different tasks depending on the inputs; this is the case of programmable circuits like micro-controllers and processors

[2]Logic gates are circuits built over transistors(typically) which operates on *Logic levels* like boolean functions (AND , OR)

[3]Remember that in a analog circuit the input is sensitive to noise, while a digital circuit isn't as sensitive

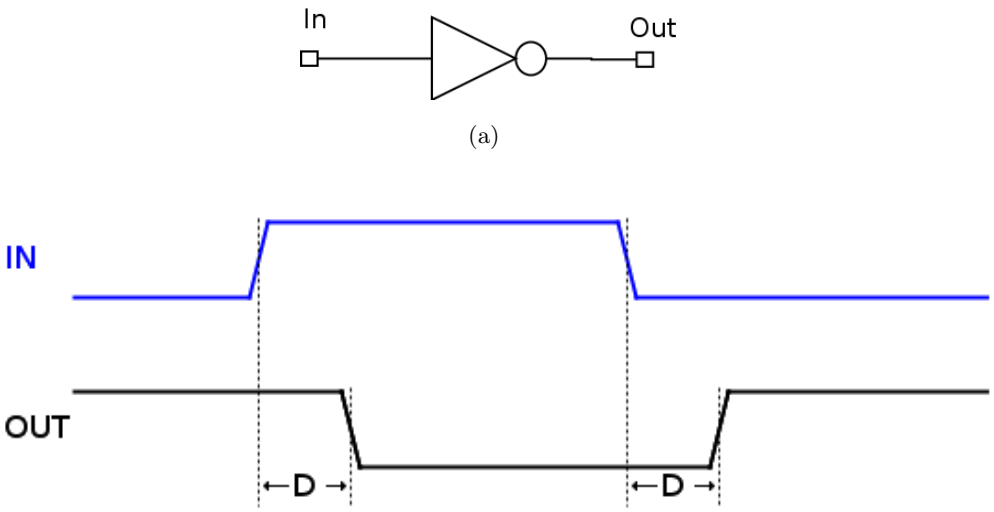
[4]HDL: Hardware description language

Economy: Digital circuits can be easily mass-produced in integrated circuits once the circuit has passed the prototype stage, making the mayor component of the circuit cost it's design time and prototyping. [6]

2.1.2 Timing analysis on digital circuits

Logic cells and wires composing a digital circuit are far from ideal from the timing perspective because the time for a signal to traverse the circuit isn't 0, in fact every wire and logic gate in the signal path introduces finite delays to the data transmission, regardless of how small these delays can be, they should be taken into consideration in digital designs as they can affect the outputs [1]. Wire related delays are usually neglected, in comparison to the delays introduced by logic gates in timing analysis. [5]

The best way to understand the timing of a simple design is using a timing diagram as the one shown in figure 2.3



(a)

(b)

Figure 2.3: (a) Shows the schematic of a simple inverter. (b) Shows the timing diagram of the inverter in (a) where the delay introduced by the inverter is shown as **D**

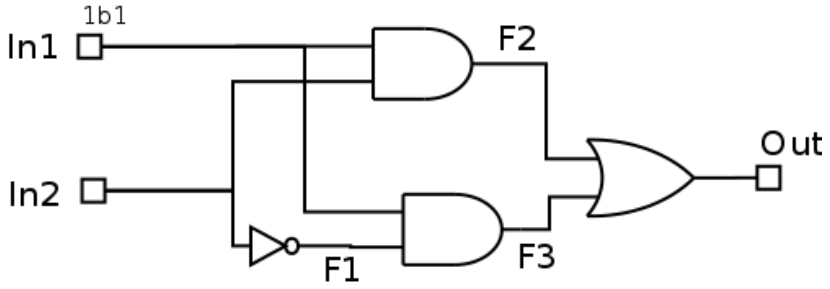
Timing hazards

Different timing delays through the datapath introduced by logic gates delays, causes dynamic behavior through the circuit when the input changes as the signal is propagated. This dynamic behavior may cause a difference between the expected result obtained by static analysis of the circuit and the observed output which behaves dynamically thus producing momentary differences

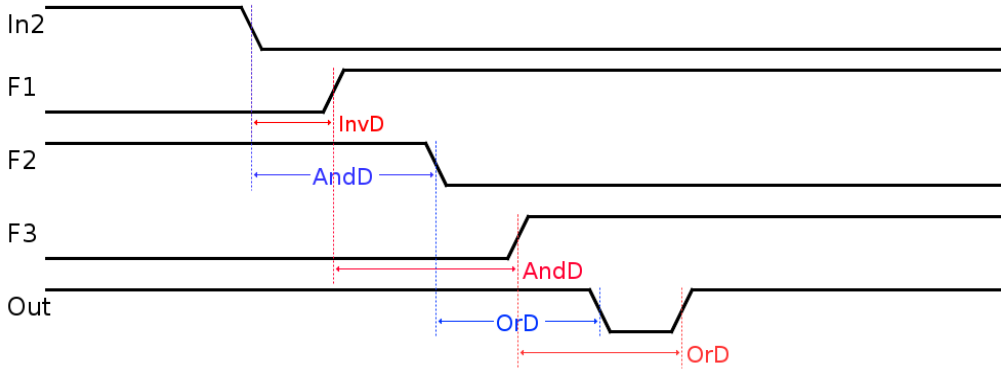
[5] Currently as logic gates and transistors shrink, the delays of the logic gates becomes smaller closing the gap between both delays

between the expected result and the observer. This difference is known as *glitch* which corresponds to a quick transition between 2(or more) different outputs states when the static analysis predicted a different behavior. The timing hazards are divided in two categories [5].

Static Hazards: A static hazard is a pair of input combination that differ only in one variable while both yield the same output signal, thus the output is expected to stay constant however the output changes momentarily before stabilizing to the expected result. An example of this is shown in figure 2.4



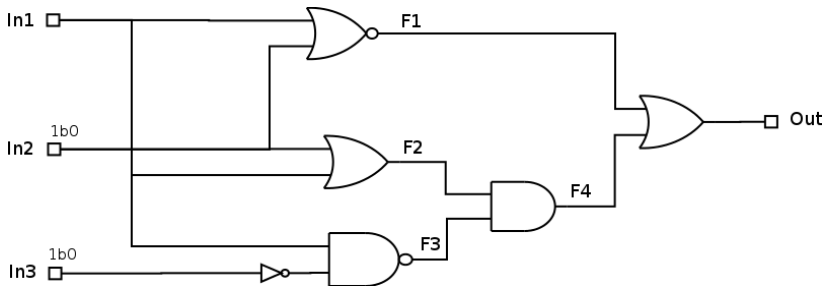
(a)



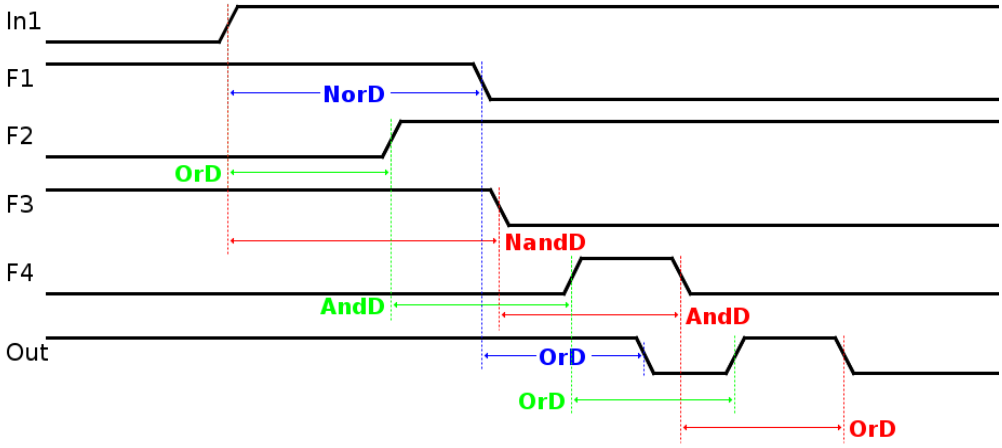
(b)

Figure 2.4: (a) Shows the schematic of a circuit susceptible to a static hazard. (b) Shows the timing diagram of the circuit shown in (a) where the signal *In1* stays in *Logic* 1 , while the signal *In2* changes from *Logic* 0 to *Logic* 1 , thus the timing diagram shows the signal propagation across the different logic stages in the circuit where the digital delays are illustrated as : **InvD** for the delay introduced by the inverter, **AndD** for the delays introduced by the *AND* gated and **OrD** for the delays introduced by the *OR* gates. The glitch is observed by the successive transitions in the output signal compared to the static analysis which predicted no transition in the output.

Dynamic hazards: A dynamic hazard a the result of multiple transitions in the output signal as result of a single input variable transition where the output was expected to change only once. An example of this is shown in the figure 2.5



(a)



(b)

Figure 2.5: (a) Shows the schematic of a circuit susceptible to a dynamic hazard. (b) Shows the timing diagram of the circuit shown in (a) for the transition of the input signal *In1* from *Logic* 0 to *Logic* 1 , while the signals *In2* and *In3* stay as *Logic* 0 ; the timing diagram illustrates the propagation of the signal across the logic to the output signal where the delays of the different logic gates are as follows: **NorD** is the delay of a *NOR* gate, **OrD** is the delay of a *OR* gate, **NandD** is the delay of a *NAND* gate and **AndD** is the delay of an *AND* gate.The glitch is observed by the successive transitions of the output signal while the static analysis predicts only the transition from *Logic* 1 to *Logic* 0

Designing hazard free circuits requires the designer to include redundant logic to the minimized design, as the inclusion of additional logic stabilizes the output signal for conflicting transitions. This can be achieved using a Karnaugh map or even including the full list of redundant functions, however this method is area expensive and such logic insertion should be avoided.

2.2 Sequential Circuits

Digital circuits are divided in 2 categories: combinational and sequential circuits as discussed by Duek on [3] and Harris on [4].

Combinational circuits: The main characteristic of combinational circuits it's that its output depends only of it's current input, therefore they are effective in direct operation such as real-time audio and video filtering and high-speed calculations.

Sequential circuits: The output of a sequential circuit can't be analyzed by considering only its current input, because the output depends on all the previous sequence of inputs which gives them their name.

2.2.1 Finite State Machines

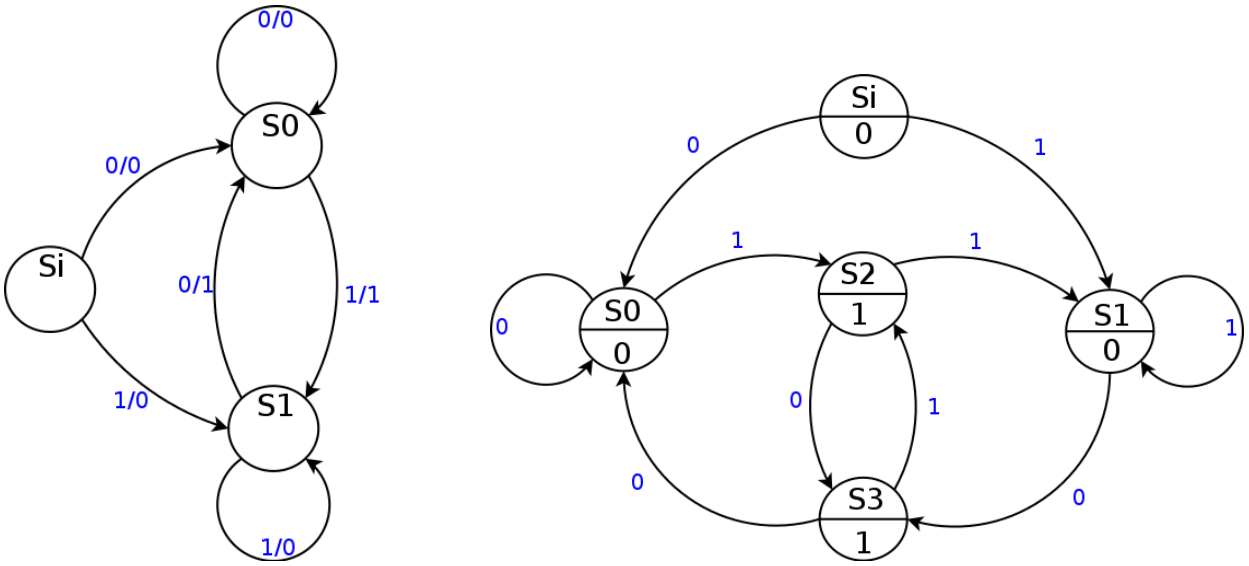
Digital circuits are widespread used mainly because of sequential circuits for they allow decision making which enables the construction of finite states machines *Finite State Machine* ; a *Finite State Machine* (as discussed in [7]) is a mathematical model composed of a finite number of states for representing the dynamic behavior of a complex system according to their sequence of inputs, where the definition of state is as follows. [5]

State : "The *state* of a sequential circuit is a collection of *state variables* whose values at any one time contain all the information about the past necessary to account for the circuit's future behavior"

There are two different kind of *Finite State Machine* , as discussed by Wakerly on [5] used for digital system modeling, the *Mealy* and *Moore Finite State Machine* .

***Mealy Finite State Machine* :** The output of a *Mealy* machine like the one shown on figure 2.6(a) depends on the current state of the machine and it's current inputs.

***Moore Finite State Machine* :** The output of a *Moore* machine like the one shown on figure 2.6(b) depends only on the current state of the machine.



(a) (b)

Figure 2.6: This figure shows the 2 mayor variants of *Finite State Machine* in digital system design. (a) Shows a basic *Mealy* machine. (b) Shows a basic *Moore* machine

Both types of *Finite State Machine* represent different approaches to solve the same problem, and both yield similar results, however a *Moore* machine is considered easier (and simpler) to implement in designs while a *Mealy* machine is usually considered more complex, but provides faster designs.

2.2.2 Sequential Cells

Sequential circuits are built using sequential elements and logic gates[6]; sequential elements are special cells with a bistable behavior, which, as discussed by Wakerly on [5], has two possible stable states and no other configuration is allowed, therefore if a bistable cell is in a given state, it can only go to the other state, there are no intermediate states allowed and as long as the cell is powered the cell won't change its state by noise. One of the most simple examples of a bistable cell is a *S-R* Latch shown in figure 2.7 [5]

[6]Logic gates is the name given to circuits that perform boolean operations over it's circuits broken down to the basic operations (like AND,OR,NAND,NOR,NOT,XOR)

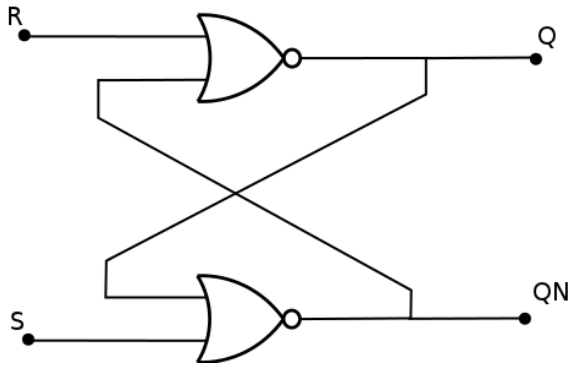
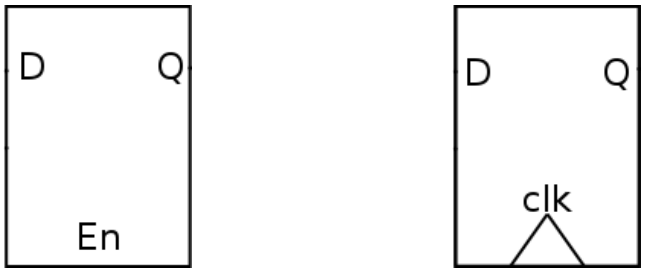


Figure 2.7: An S-R *Latch* as a collection of logic gates

There are two families of bistables in digital circuit designs, *Latches* and *Flip-Flops,* being the output response to the change of data input the mayor difference between them. Even though there are several kind of *Latches* and *Flip-Flops,* only (*Positive logic*) *D Latches* and *D Flip-Flops* shown in Figure 2.8, are treated in this text, for they are the easiest to understand and model, therefore the most used variant of either *Latch* or *Flip-Flop* in digital circuits design.



(a) (b)

Figure 2.8: This figure shows the most common representation of a *D Latch* and a *D Flip-Flops* (a) Shows the typical representation of a *D Latch.* (b) Shows the typical representation of a *D Flip-Flop* .

D Latch

The *D Latch* is a bi-stable element with 2 inputs (usually named **D** and **EN**) and typically 2 outputs (**Q** and **Q_n**)[7], the behavior of this component is dependent of the enable input (**EN**) as whenever this signal corresponds to a *Logic* 1 the output's (**Q**) value is exactly the same as the input's (**D**) value, while the **Q_n** output is always the logic inverse of the **Q** output. However as long as **EN** remains as a *Logic* 0 the outputs values won't change regardless of the input value. This behavior can be seen in the figure 2.9

[7]In the present document, for simplicity, the **Q** output will be used while the **Q_n** output is ignored unless necessary. Also some cells have only one of the outputs available from the *D Latch* cell in order to reduce the area of the cell; the same goes for a *D Flip-Flop*

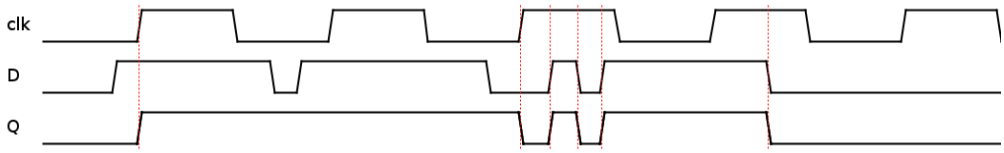


Figure 2.9: Timing diagram of the *D Latch.* For for simplicity, the **EN** pin is fed with a **clk** signal

D Flip-Flop

The *D Flip-Flop* behavior is very similar to the *D Latch,* as both components have the same inputs and outputs, but the main difference between them is the response of the element against the control signal **EN** . The outputs only change in a *Flip-Flop* if the control signal experiments a transition between a *Logic* 0 and a *Logic* 1 , keeping the stored value as long as there are no more transitions as seen on figure 2.10

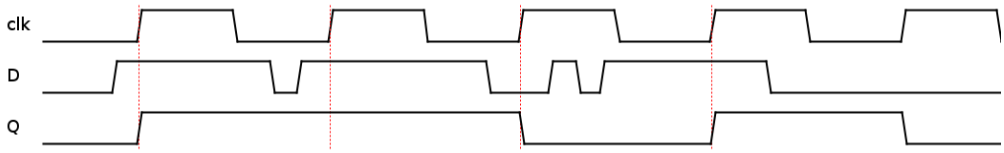


Figure 2.10: Timing diagram of the *D Flip-Flop* . For for simplicity, the **EN** pin is fed with a **clk** signal

From now on, a *D Latch* will be referred unambiguously as *Latch,* while a *D Flip-Flop* will be named as *Flip-Flop* or just *flop* for simplicity.

In most applications the control signal used for the sequential elements is an externally generated periodic signal, a clock signal o **clk** .

2.3 Synchronous and Asynchronous Circuits

Sequential circuits can be further divide in synchronous and asynchronous circuits, depending on how the sequential cell are connected. Most circuits can be designed as a synchronous circuit and have a similar design performing the same task built as an asynchronous circuit. Each paradigm has advantages and disadvantages and the decision over which type of design use is based on in parameters like complexity, speed and robustness.

Asynchronous Circuits: Are circuits that are not controlled nor synchronized to an external **clk** signal, even if such signal exists, therefore are called asynchronous since there is no synchronization possible between the different state transitions of the circuit given the absence

of the control signal. These circuits if fed a **clk** signal use it only as reference and not as a control signal as the internal state-changes are controlled only by the logic and sequential cells delay.

Synchronous Circuits: Are circuits controlled by an external **clk** signal and the outputs of the circuit, along with all the internal state transitions, are synchronized with it, this is accomplished by connecting every sequential cell to the clock signal source, however this is not enough if the sequential cell is a *Latch* given the latch transparency window[8]

Asynchronous circuits might be considered faster than synchronous circuits, given that synchronous circuits depend on the clock signal for state transition, however synchronous circuits have several advantages over asynchronous circuits for they are simpler to design and more robust, since synchronous circuits can be analyzed using the same models as asynchronous circuits, yet the synchronizing **clk** signal simplifies the analysis while improving the design robustness, this has as result that basically all digital designs are synchronous, however asynchronous circuit analysis is still fundamental for analyzing interfaces between 2 different designs with different clock signal and still having a working circuit. [4]

2.3.1 Synchronous Circuit advantages

Synchronous circuits are used because their timing analysis is simplified compared to the timing analysis of an asynchronous circuit, as the timing analysis is reduced to the analysis between the **clk** signal and the data signals, because the outputs and the internal states of the circuit are synchronized to the **clk** signal.

The timing analysis of a synchronous circuit is further simplified by using only *Flip-Flops* as sequential cells for their timing characteristics, previously discussed in 2.2.2, which allow the analysis to consider only a specific edge of the control signal, which in this case is the **clk** , and to compare it to the data arrival time, therefore as long as every the possible data signals arrive to the *Flip-Flop* before the **clk** signal for a given period the circuit will behave properly, therefore eliminating the need for analyzing the timing hazards of the circuits for most cases.

Since the timing analysis of a synchronous circuit is heavily simplified, circuits designers are able to focus on the functionality of the circuit, rather than preventing timing hazards, thus simplifying the design as the redundant logic needed to prevent the hazards is not necessary and leaving the timing analysis to a posterior stage. However timing analysis must not be overlooked as the maximum frequency of the circuit will be limited by the slowest logic path between 2 different sequential cells, therefore careless design negatively impacts the speed of the circuit.

[8]The transparency window is the time where the input signal of the *Latch* is propagated to the output. This will be discussed further in 2.4.3

2.4 Timing parameters for timing analysis

In order to design an synchronous circuit, the synchronous behavior of the *Flip-Flop* regarding it's control signal makes it the easiest choice of sequential cell, however with careful design *Latches* can also be used as sequential cell and still obtain a *synchronous* behavior. Even if *Latches* don't have the synchronous behavior that *Flip-Flops* have, their smaller size, thus requiring less area and power than a *Flip-Flop* and better timing parameters, as discussed in [8] makes *Latches* an interesting alternative to *Flip-Flops* when designing high performance circuits.

The timing analysis of the *Flip-Flop* or *Latch* datapath requires the understanding of the timing parameters of the *Latch, Flip-Flop* and the clock parameters.

2.4.1 Clock parameters

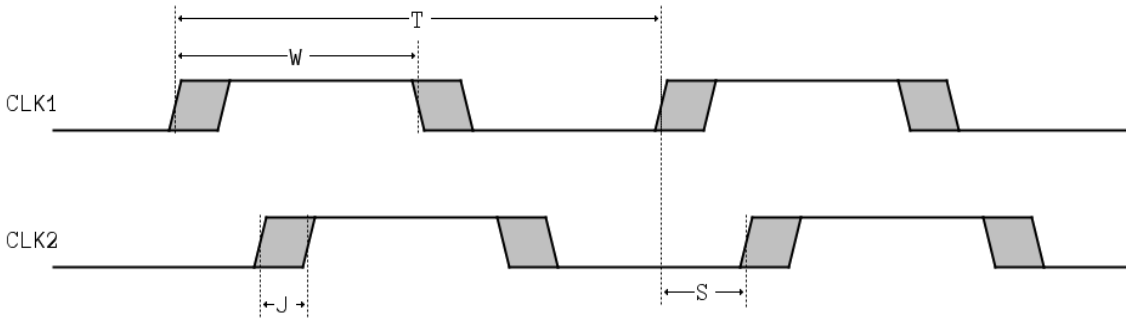


Figure 2.11: The clock parameters as discussed in [9], where T represents the period, W represents the duty cycle, J is the Jitter and S represents the clock-skew.

The clock signal used as reference in a synchronous design can be described using different parameters shown in figure 2.11, which are defined as following:

Period: Is the time between 2 consecutive rising edge of the **clk** signal.

Duty cycle: Is the time between the rising edge and the falling edge during a single period of the **clk** signal, also known as time window.

Jitter: Is the local deviation in the period of the **clk** signal as noise for the same location, therefore is considered as random local noise since it's different on every period and different for each location in the circuit.

Clock skew: Is time difference in the **clk** signal measured at 2 different locations of the circuit, therefore is considered a global noise and stable, since it depends on the circuit structure.

It becomes obvious that to describe an ideal **clk** signal is enough to define its period and duty cycle.

2.4.2 *Flip-Flop* timing parameters

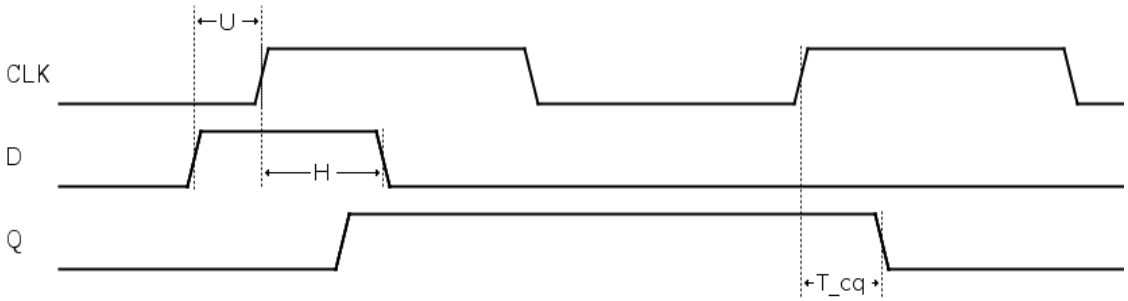


Figure 2.12: *Flip-Flop* timing parameters as discussed in [9], where U represents the set-up time, H represents the Hold time and T_{cq} represent the time clock-to-Q.

The *Flip-Flop* timing parameters as shown on figure 2.12 describe the behavior of the cell and its internal delays which are defined as following:

Set-up time: Is the minimum time that the data signal must be stable before the end of the data capture in order to the signal to be handled properly. In the case of the *Flip-Flop* the end of the data capture corresponds to the rising edge of the **clk** signal.

Hold time: Is the minimum time that the data signal must be stable after the end of the data capture of the cell in order to the signal to be handled properly. In the case of a *Flip-Flop* the end of the data capture corresponds to the rising edge of the clock signal.

Time clk-Q: Or t_{cQ} is the internal delay of the *Flip-Flop* which indicates the time it takes for the input signal to be available at the output pin of the cell after the rising edge of the **clk** signal

2.4.3 *Latch* timing parameters

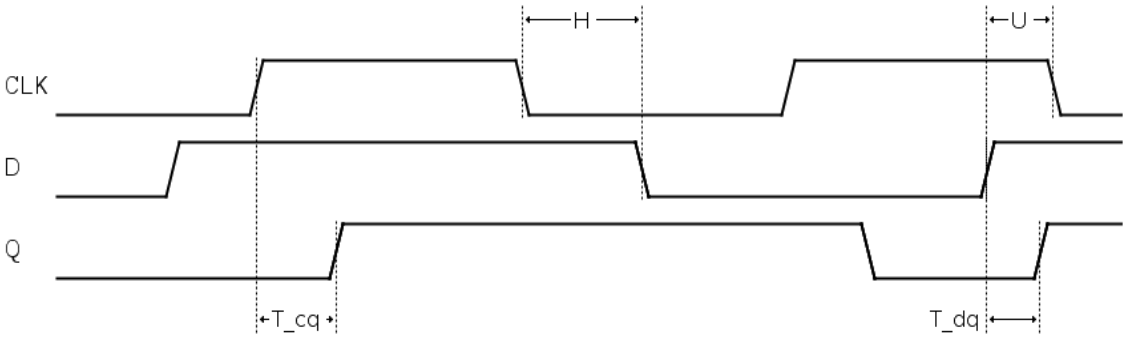


Figure 2.13: *Latch* timing parameters, as discussed in [9], where U represents the set-up time, H represents the Hold time, T_{cQ} represents the time clock-to-Q and T_{DQ}represents the time D-to-Q

The *Latch* timing parameters as shown on figure 2.13 describe the behavior of the cell and its internal delays regarding the control signal, which in this case is a **clk** for ease of comparison, which are defined as following:

Set-up time: Just as defined in 2.4.2 represent the minimum time that the data input must be stable before the end of the capture time, which in a *Positive logic Latch* is the falling edge of the control signal.

Hold time: Is the time that the data signal must be stable after the cell has stopped to capture data in order t handle it properly. The latch stops to capture data in the falling edge of the control signal.

Time clk-Q: Is the time it takes to the data signal to be available at the output of the cell after a rising edge of the control signal (in this case the **clk**).

Time D-Q: Is the time it takes to the input signal to be available at the output of the cell after a transition of the input signal while the control signal is in *Logic* 1 , which means that the *Latch* is transparent.

From the timing parameters of the *Latch* and *Flip-Flop,* it becomes clear that as long as all data signals are timed in such a way that every transition of the data inputs of the sequential cell occurs only while the **clk** signal is in *Logic* 0 , then there is no difference between a design where all the *Flip-Flop* cells are replaced with *Latches* as the *Latches* would behave as synchronously as *Flip-Flops* would. However this analysis is too simple and doesn't consider the timing differences between *Latches* and *Flip-Flops* .

2.5 Timing equations of *Latches* and *Flip-Flops*

The timing parameters of the *Latches* and *Flip-Flops* impose restrictions on the data arrival to a cell, therefore if the signal arrives after the set-up time of the cell, the cell is in a *Set-up timing Violation* and if the data input of a signal isn't stable during the Hold-time, then the cell is in a *Hold timing Violation*.

2.5.1 *Flip-Flop* set-up timing violation

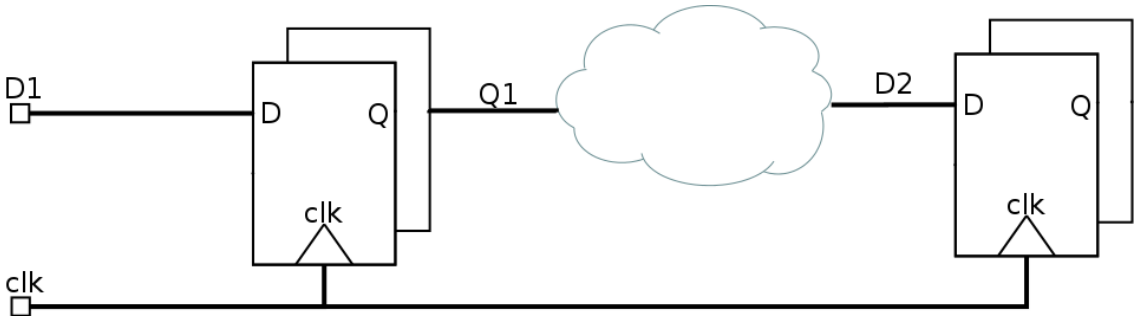


Figure 2.14: *Flip-Flop* schematic used as reference in the analysis of the set-up timing violations

$$T_{D_2} = R_1 + t_{DQ} + D_{L_M} \tag{2.1}$$

For the timing analysis of a *Flip-Flop* datapath, the circuit shown in figure 2.14 is used as reference, in the *Set-up* timing analysis, the data of the first *Flip-Flop* is assumed to have arrived at a proper time so the time it takes the signal to travel from the first *Flip-Flop* in the figure to the second is given for the equation 2.1 where :

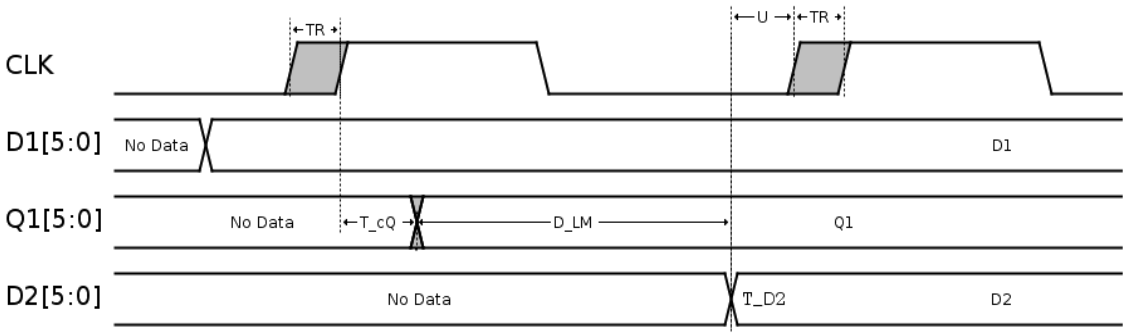


Figure 2.15: *Flip-Flop* timing diagram used as reference for the set-up timing violation analysis. This figure uses only one **clk** signal as reference for both *Flip-Flops* for simplicity, however in the analysis each bank is suposed to be fed with a slightly different **clk** signal derived from the same source.

T_{D_2} Is the time of arrival of the signal from the first *Flip-Flop* to the second *Flip-Flop* in the datapath

R_1 Is the rising edge of the **clk** signal perceived by the first *Flip-Flop*

t_cQ Is the *Time clk-Q* of the first *Flip-Flop*

D_{L_M} Is the maximum logic delay from the output of the first *Flip-Flop* to the second *Flip-Flop* .

And figure 2.15 is used to illustrate the scenario described.

$$T_{D_2} \leq R_2 - U \tag{2.2}$$

The data arrival to the second *Flip-Flop* must comply the set-up condition, shown in equation 2.2, where:

R_2**:** Is the rising edge in the **clk** pin of the second *Flip-Flop* .

U**:** Is the set-up time of the second *Flip-Flop*

Therefore the timing restriction is obtained by using equations 2.1 and 2.2 obtaining the equation 2.3.

$$R_1 + t_{cQ} + D_{L_M} \leq R_2 - U \tag{2.3}$$

The equation 2.3 is not complete as the information of the rising edges, which are defined by equations 2.4 and 2.5 where the value of " n" is assigned to match the corresponding edge, is not yet included in the equation.

$$R_1 = n \cdot T + t_r \tag{2.4}$$

$$R_2 = n \cdot T + t_r \tag{2.5}$$

By following a conservative approach for the rising edges defined by previous equations where T is the period of the signal and t_r is the uncertainty of the rising edge, equations 2.4 and 2.5 become 2.6 and 2.7 respectively, where T_R is the worst rising edge uncertainty.

$$R_1 = n \cdot T + T_R \tag{2.6}$$

$$R_2 = (n+1) \cdot T - T_R \tag{2.7}$$

Finally by using the equations 2.6 and 2.7 on the equation 2.3 with simple algebraic manipulation becomes the equation 2.8 which is the timing restriction of the *Flip-Flop* and the period given by the set-up time.

$$T \geq t_{cQ} + D_{L_M} + U + 2T_R \tag{2.8}$$

2.5.2 *Flip-Flop* hold timing violation

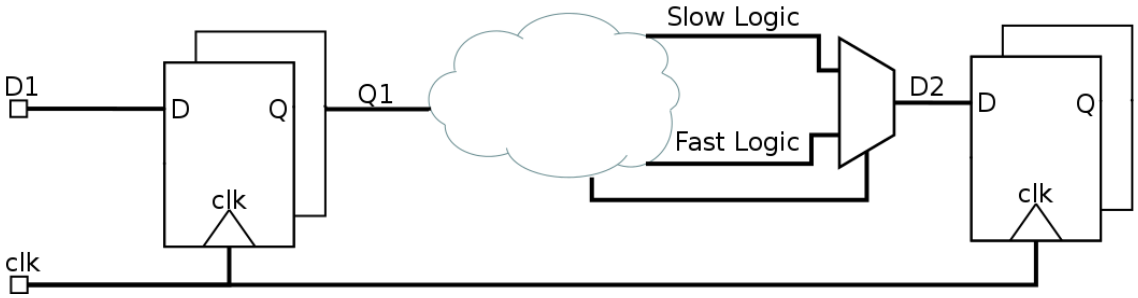


Figure 2.16: *Flip-Flop* schematic used as reference for the hold timing violations analysis. Shows a explicit difference between a slow (or regular) datapath and a faster datapath.

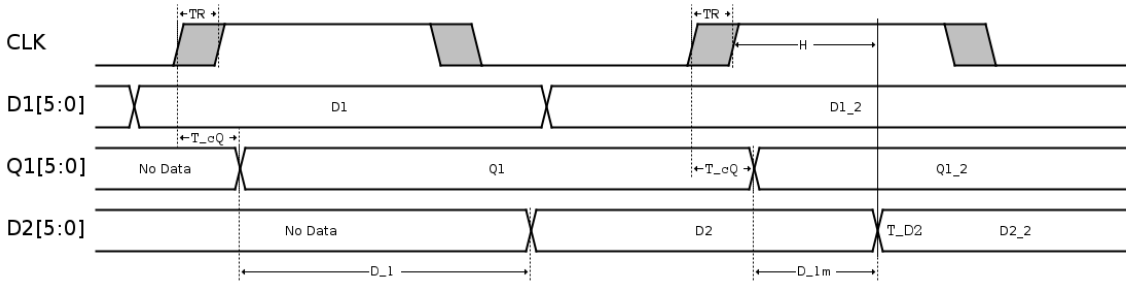


Figure 2.17: Timing diagram for the *Flip-Flops* used as reference for the hold-timing violation analysis. This figure uses only one **clk** signal as reference for both *Flip-Flops* for simplicity, however in the analysis each bank is suposed to be fed with a slightly different **clk** signal derived from the same source.

For the *Flip-Flop* hold timing violation analysis, the figure 2.16 is used as reference to better illustrate the circuits condition, also the figure 2.17 is used to illustrate the timing conditions analyzed.

$$T_{D_{2-2}} = R_1 + T_{cQ} + D_{l_m} \tag{2.9}$$

From figure 2.17 the arrival of the problematic signal $T_{D_{2-2}}$ is given by the equation 2.9 where:

R_1**:** Is the rising edge in the first *Flip-Flop*

D_{l_m}**:** Is the minimum logic delay given by the datapath between the first and second *Flip-Flop*

t_{cQ}**:** Is the *Time clk-Q* of the first *Flip-Flop*

$$T_{D_{2-2}} \geq R_2 + H \tag{2.10}$$

However for the signal to comply the hold restriction, the time of arrival of the signal must follow the restriction set by the equation 2.10 where:

F_2**:** Is the rising edge of the **clk** signal received by the second *Flip-Flop* .

H**:** Is the hold time of the second *Flip-Flop*

$$R_1 + T_{cQ} + D_{l_m} \geq R_2 + H \tag{2.11}$$

From equations 2.9 and 2.10 the hold timing restriction for a *Flip-Flop* is given by the equation 2.11, where the rising and falling edge are given by the equation 2.4 and 2.5 respectively.

$$R_1 = n \cdot T - T_R \tag{2.12}$$

$$R_2 = n \cdot T + T_R \tag{2.13}$$

By considering the worst case scenario for the **clk** signals arriving each *Flip-Flop,* the rising edge of the cells are given by the equations 2.12 for the rising edge of the first cell where the clock arrived the earliest possible time and the equation 2.13 for the rising edge of the second cell where the **clk** signal arrived at the latest possible time. Finally by combining equations 2.11, 2.12 and 2.13; using simple algebraic manipulation the restriction over the fastest logic in the design based on the hold timing requirement of a *Flip-Flop* cell is represented by the equation 2.14.

$$D_{l_m} \geq 2T_R + H - t_{cQ} \tag{2.14}$$

2.5.3 *Latch* set-up timing violation

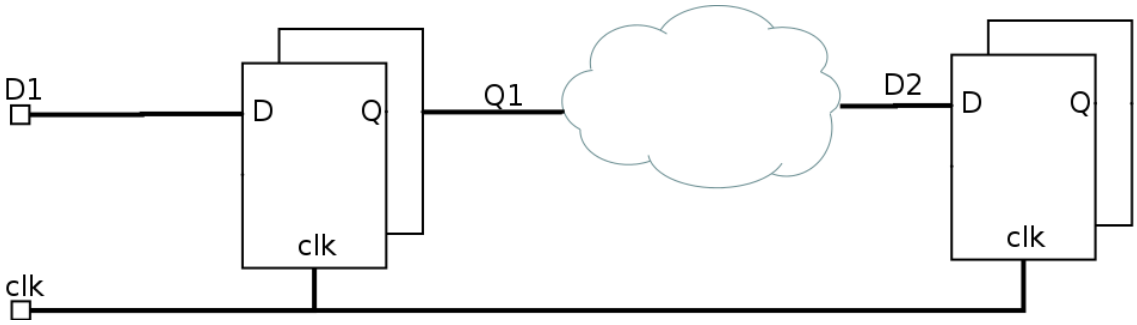


Figure 2.18: *Latch* schematic used as reference in the analysis for the set-up timing violations

Figure 2.18 shows the schematic of a circuit composed of 2 banks of latches which is used as reference for the timing analysis of the set-up condition. Given the *Latches* timing characteristics, there are 2 possible scenarios for a set-up timing restriction.

Latch set-up timing: late clk signal

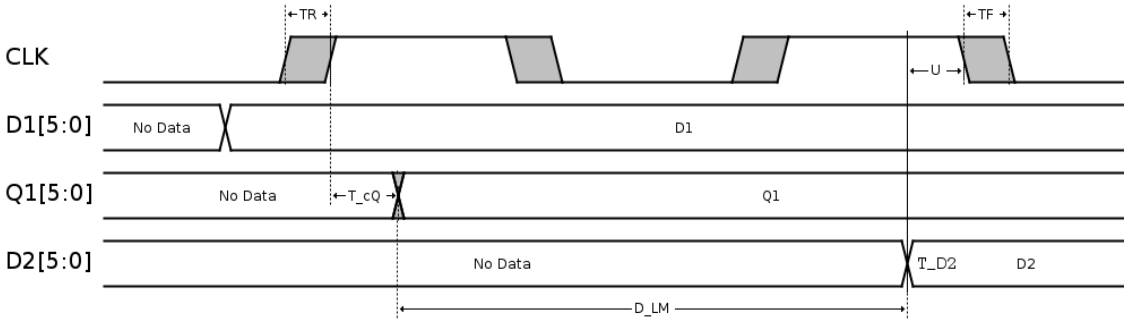


Figure 2.19: Timing diagram for the *Latches* used as reference for the set-up timing violation analysis, in the case when the data in the first *Latch* arrives before the **clk** signal which arrives at the latest possible time. This figure uses only one **clk** signal,for simplicity, as reference for both *Latches,* however in the analysis each bank is suposed to be fed with a slightly different **clk** signal derived from the same source.

$$T_{D_2} = R_1 + t_{cQ} + D_{L_M} \tag{2.15}$$

When the data of the first *Latch* arrives before the **clk** signal, as shown in the figure 2.19 the data arrival to the second latch is given by the equation 2.15 where:

R_1**:** Is the rising edge of the first *Latch*

t_{cQ}**:** Is the *Time clk-Q* of the first *Latch*

D_{L_M}**:** Is the maximum logic delay of the circuit.

$$T_{D_2} \geq F_2 - U \tag{2.16}$$

However the set-up requirement given by the equation 2.16, where:

F_2**:** Is the falling edge of the **clk** signal in the second *Latch*

U**:** Is the set-up time requirement of the *Latch*

$$R_1 + t_{cQ} + D_{L_M} \leq F_2 - U \tag{2.17}$$

$$F_2 = n \cdot T + W + t_f \tag{2.18}$$

Now using the equations 2.15 and (2.16), the set-up timing restriction for the case when the data arrives to the first *Latch* before the **clk** signal is represented by the equation 2.17, in which the rising edge of the first *Latch* is given by the equation 2.4, meanwhile the falling edge of the **clk** arriving the second latch is given by the equation 2.18, where the value of the parameter n is determined accordingly to the case of analysis.

$$R_1 = n \cdot T + T_R \tag{2.19}$$

$$F_2 = (n+1) \cdot T + W - T_F \tag{2.20}$$

By using the worst-case scenario approach for the timing analysis in the scenario, the rising and falling edge of the **clk** signal are given by the equations 2.19 and 2.20 where:

W**:** Is the nominal duty-cycle of the **clk** signal

T_F**:** Is the worst **clk** uncertainty in the falling edge of the **clk** signal in the whole design.

Finally by combining using the equations 2.19 and 2.20 in the equation 2.16 and using simple algebraic manipulations the set-up timing requirement of the *Latch* for the current scenario is given by the equation 2.20

$$T \geq D_{L_M} + t_{cQ} + T_F + T_R + U - W \tag{2.21}$$

***Latch* set-up timing: late data signal**

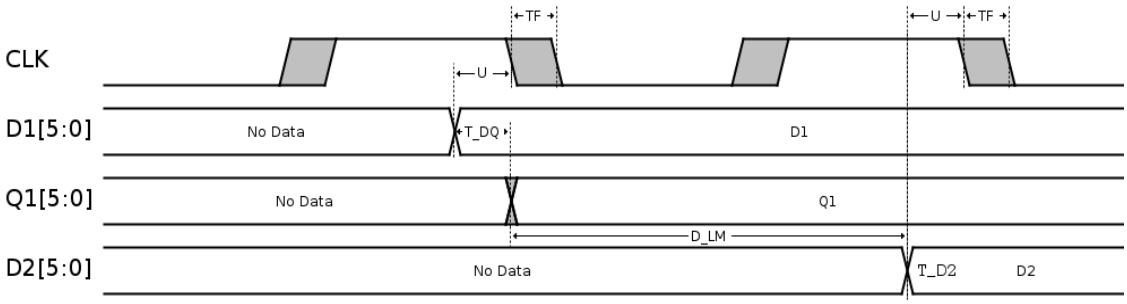


Figure 2.20: Timing diagram for the *Latches* used as reference for the set-up timing violation analysis in the case when the data signal on the first *Latch* arrives at the latest possible time. This figure uses only one **clk** signal,for simplicity, as reference for both *Latches,* however in the analysis each bank is suposed to be fed with a slightly different **clk** signal derived from the same source.

$$T_{D_2} = T_{D_1} + t_{DQ} + D_{L_M} \tag{2.22}$$

When the data signal in the first *Latch* arrives during the transparency window, as shown in the figure 2.20, then the arrival of the data signal to the second *Latch* is given by the equation 2.22 where:

T_{D_1}**:** Is the arrival of the data signal to the first *Latch* .

t_{DQ}**:** Is the time *Time D-Q* of the *Latch*

D_{L_M}**:** Is the maximum delay of the circuit in the figure 2.18.

$$T_{D_1} + t_{DQ} + D_{L_M} \leq F_2 - U \tag{2.23}$$

However the set-up timing requirement of the *Latch* is given by the equation 2.16, therefore by using the equation 2.16 and 2.22 to get the equation 2.23 which represents the timing requirement for the *Latch* in this scenario.

(2.24) $$T_{D_1} = R_1 + W - U$$

(2.25) $$T_{D_1} = n \cdot T + T_F + W - U$$

Now by using the worst-case scenario analysis for the falling edge of the *Latch* given by the equation 2.20 and considering the latest possible time arrival of the data signal in the first *Latch* given by the equation 2.24 where the rising edge of the **clk** signal received by the first *Latch* R_1 given by the equation 2.19, which used in the equation 2.25 yields the worst possible data arrival for the first *Latch* given by the equation 2.25.

(2.26) $$T \geq t_{DQ} + D_{L_M}$$

Finally by using the equations 2.23 and 2.25 by using simple algebraic manipulations the set-up timing restriction of the *Latch* given by the latest possible data arrival is given by the equation 2.26.

The equations 2.21 and 2.26 represent the timing restrictions over the period given the maximum logic delay of the circuit. Since the internal delays of the *Latch* are similar as discussed in [9], in a first order analysis, they can be considered as equal, thus the timing restriction that apply for most analysis is the most restrictive one, which is equation 2.21.

2.5.4 *Latch* hold timing violation

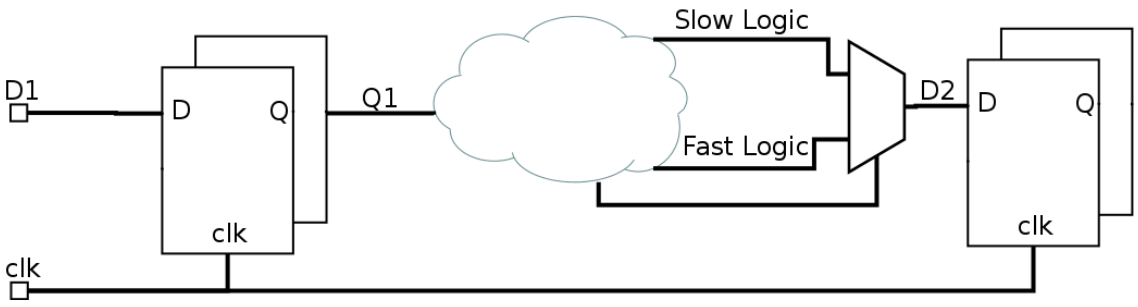


Figure 2.21: *Latch* schematic used as reference in the analysis for the hold timing violation

Figure 2.21 shows the schematic of a circuit composed of 2 banks of latches which is used as reference for the timing analysis of the hold condition. Given the *Latches* timing characteristics, there are 2 possible scenarios for a set-up timing restriction.

***Latch* hold timing: data after clk**

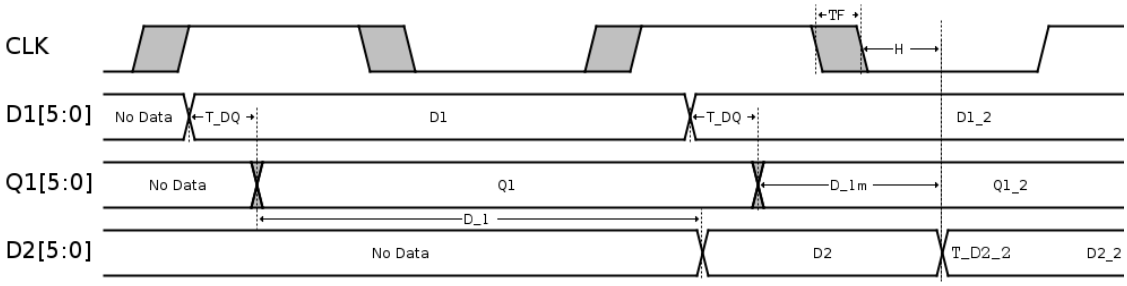


Figure 2.22: Timing diagram for the *Latches* used as reference for the hold timing violation analysis in the case when the data signal on the first *Latch* arrives at the latest possible time. This figure uses only one **clk** signal,for simplicity, as reference for both *Latches,* however in the analysis each bank is **clk** signal is slightly different.

The first scenario for the timing analysis is that the data arrival in the first *Latch* occurs after the **clk** signal arrival, which is shown in figure 2.22.

$$T_{D_{2-2}} = T_{D_{1-2}} + t_{DQ} + D_{l_m} \tag{2.27}$$

From figure 2.22 the arrival time of the signal in the second *Latch* is given by the equation 2.27 where:

$T_{D_{1-2}}$**:** Is the time of the second data arrival to the first *Latch*

t_{DQ}**:** Is the time *Time D-Q* of the *Latch*

D_{l_m}**:** Is the minimum delay between the banks of *Latches*

$$T_{D_{2-2}} \geq F_2 + H \tag{2.28}$$

Still the hold timing requirement of the latch is given by the equation 2.28 where:

F_2**:** Is the falling time of the **clk** signal in the second *Latch*

H**:** Is the hold time of the second *Latch*

(2.29) $$T_{D_{1-2}} + t_{DQ} + D_{l_m} \geq F_2 + H$$

By using the equations 2.27 and 2.28 the restriction over the logic delays in the circuit can be calculated as shown in equation 2.29.

(2.30) $$F_2 = n \cdot T + T_F + W$$

Considering, for analysis purposes, the worst-case scenario for the falling edge in the second *Latch* given by the equation 2.30 and using it on the equation 2.29, through simple algebraic manipulations the timing requirement of the *Latch* over the minimum logic delay in the datapath can be written as in the equation 2.31.

(2.31) $$D_{l_m} \geq W + TF + H - t_{DQ} - T_{D_{1-1}}$$

***Latch* hold timing: data before clk**

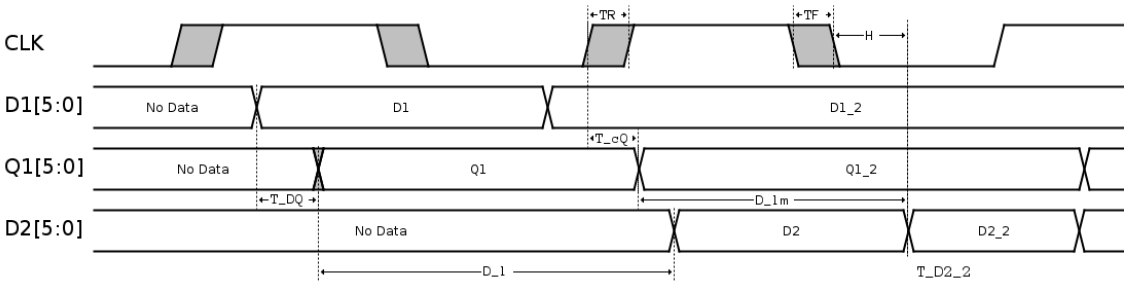


Figure 2.23: Timing diagram for the *Latches* used as reference for the hold timing violation analysis in the case when the data signal on the first *Latch* arrives before the **clk** signal which in turn arrives at the latest possible time. This figure uses only one **clk** signal for simplicity as reference for both *Latches,* however in the analysis each bank is **clk** signal is slightly different.

(2.32) $$T_{D_{2-2}} = R_2 + t_{DQ} + D_{l_m}$$

Figure 2.23 shows the timing diagram of figure 2.21 for the case in which the second data signal to the first *Latch* arrives before the **clk** rising edge. In this scenario, the arrival time of the second data signal to the second *Latch* is given by the equation 2.27 where:

R_2**:** Is the time of the second rising edge of the second *Latch*

t_{cQ}**:** Is the time *Time clk-Q* of the *Latch*

D_{l_m}**:** Is the minimum delay between the banks of *Latches*

$$R_1 + t_{cQ} + D_{l_m} \geq F_2 + H \tag{2.33}$$

The timing requirement of the datapath is given by the equation 2.28, therefore by using the data arrival time given by the equation 2.27 over the equation 2.28 the restriction can be re-written into equation 2.33.

$$D_{l_m} \geq W + TF + T_R + H - t_{cQ} \tag{2.34}$$

The worst-case scenario analysis implies the rising edge of the first *Latch* arrives at the earliest possible time which is shown by equation 2.12, similarly the worst possible falling edge for the **clk** signal in the second *Latch* is as late as possible which is shown in the equation 2.30. Therefore by using equations 2.12 and 2.30 in equation 2.33 the timing requirement over the logic delay can be written as shown in equation 2.34.

Further analysis is required regarding the hold timing condition for *Latches,* because equations 2.31 and 2.34 represent different race conditions not entirely compatible since equation 2.31 is calculated over a specific arrival time for the data signal in the first *Latch,* therefore if the data to the first *Latch* were to arrive before the time used as reference to calculate the hold condition, then for that path, the data arrival to the second *Latch* wouldn't comply the hold time restriction. Thus in order to ensure no problems arise for the logic delay, the condition shown in 2.31 must be calculated for the worst possible scenario which means that the earliest possible data arrival must coincide with the earliest possible **clk** for that datapath.

$$D_{l_m} \geq W + TF + H - t_{DQ} + T_R \tag{2.35}$$

Using the previous analysis and by using the rising edge is given by the equation 2.12, which applied to the equation 2.31 yields the equation 2.35, however the difference between equations 2.34 and 2.35 is the internal delay of the latch whose corresponds to the time *Time clk-Q* and time *Time D-Q* , yet since the data for the data-after **clk** signal scenario must arrive at the exact same time as the **clk** signal, the parameter triggered is the the time *Time clk-Q* and not the time *Time D-Q* , therefore the only valid equation for the hold timing requirement is equation 2.34.

2.6 Latch-based Designs

Section 2.4 argued that *Latches* were an interesting alternative to *Flip-Flop* based designs, now thanks to the equations from section 2.5 it's possible to analyze the effect of switching *Latches* and *Flip-Flops* in a designs.

2.6.1 Replacing *Flip-Flop* with *Latch*

Consider a *Flip-Flop* based design, similar to the one used in the section 2.5.1, in this scenario timing restriction over the period of the **clk** signal is given by the equation 2.8. If one were to replace the *Flip-Flops* with *Latches,* the period restriction would be given by equation 2.14.

For analysis purposes, consider the maximum delay in the circuit isn't affected by this operation, also given the **clk** parameters aren't related to sequential cells, thus it's parameters wouldn't be affected. Finally the only parameters that are still different are the internal delays of the sequential cells which for the analysis can be forced to be equal.

$$T_{ff} \geq T_{la} + (T_R - T_F) + U + W \tag{2.36}$$

So by choosing the equality in the equation 2.21 and by sorting the terms in the equation the a relation between the period of a *Flip-Flop* and the period of a *Latch* which is shown by the equation 2.36 where:

T_{ff}**:** Is the period of the *Flip-Flop* given by equation 2.8

T_{la}**:** Is the period of the *Latch* given by the equation 2.21 where the equality has been chosen.

T_R**:** Is the worst uncertainty in the rising edge of the **clk** signal.

T_F**:** Is the worst uncertainty in the falling edge of the **clk** signal.

U**:** Is the set-up timing requirement of either the *Latch* or *Flip-Flop* (given that they have been supposed to be the same)

W**:** Is the duty-cycle of the **clk** signal.

Equation 2.36 can be used to conclude that given *Latches* and *Flip-Flops* with the exact same parameters, the period requirement of the *Flip-Flop* is larger than the period requirement for the *Latch,* thus the same design based with *Latches* can operate at a higher frequency than if it were *Flip-Flop* based.

2.6.2 Latch based design advantages

Flip-Flop based designs have the advantage of being easy to design thanks to their hold timing condition. The *Flip-Flop* hold timing restriction imposes a minor restriction over the minimum delay of the logic in the circuit compared to the restriction imposed by *Latches,* still *Latch* based designs have definitive advantages over *Flip-Flop* as discussed previously in section 2.4.

Latches have less area

The number of transistor used to build a digital cell is directly related to it's complexity, since the number of transistors required to build a *Latch* are less than the number of transistors required to build a *Flip-Flop,* a *Latch* cell requires less area than a *Flip-Flop* performing the same function built using the same transistors [9]

Latches consumes less power

The power consumption of a circuit which will be discussed deeply later, like the area is directly related to it's complexity which means the number of transistors used to build the cell. Since *Latches* are built using less transistors than *Flip-Flops,* they require less energy to keep each transistor powered up. Also the state transitions of the circuit consume power therefore by having less transistors *Latches* consumes less power than *Flip-Flops* in every state transistor, thus reducing the total power consumption of the circuit.

Clock uncertainty absorption

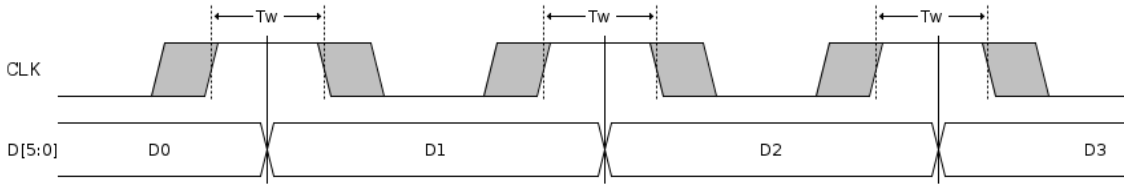


Figure 2.24: Timing diagram showing clock uncertainty absorption for latch based design.

Thanks to the timing requirements of the *Latch,* there is a *transparency window*, illustrated in figure 2.24 as T_w, in which the arriving data signal will be propagated through the cell to its output. This transparency window starts with the rising edge of the **clk** signal and ends a set-up time before the falling edge of it.

Considering the clock uncertainties for each edge, still there is a completely deterministic time frame during which the *Latch* is transparent, therefore by designing the logic of the circuit in such

a way that every transition at the input of every *Latch,* the clock uncertainty won't be relevant for the analysis. [9]

Figure 2.24 shows the timing diagram of a latch based design in which the clock-uncertainty absorption becomes clear.

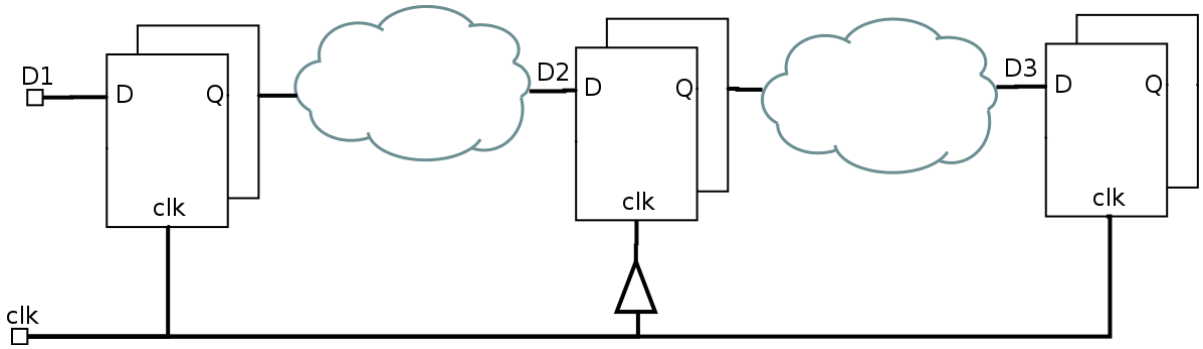
Time borrowing

Time borrowing also known as *Cycle stealing* is a design technique in which some of the logic of the circuit is allowed to be slower than the **clk** signal with the restriction that the slow logic is compensated in further stages with fast logic, thus in this operation slow logic stages *borrow* time from faster stages.

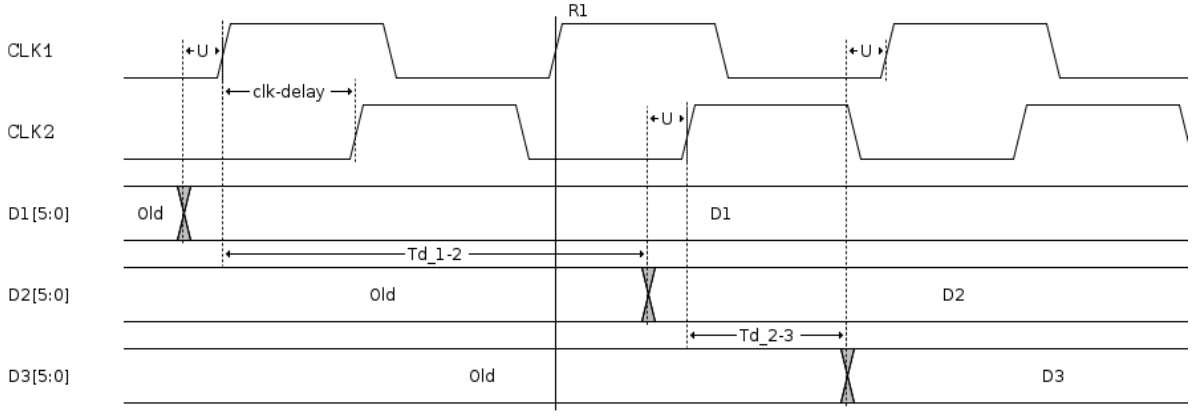
The theory behind time borrowing discussed in [9] states that there are 2 ways of achieving time borrowing:

Static Time borrowing: Static time borrowing is achieved using a variation over a standard clock-tree synthesis[9] in which the **clk** signal for a specific group of sequential cells is delayed using buffers in order to allow the slow logic to finish its evaluation, before the arrival of the clock signal, while the next group of sequential cells is fed with the non-delayed clock. Static time borrowing gives the path the same average **clk** signal while the slowest logic delay is longer than the time period which is compensated borrowing time from the subsequent stages. This situation is illustrated by figure 2.25.

[9] Clock tree synthesis is the process of creating the wiring of the **clk** signal in order to minimize the clock-skew over the design. Since this process isn't part of the focus of this work, no further information will be provided, still the reader is free to search for further information in [5] near the **clk** skew discussion and [10]



(a)



(b)

Figure 2.25: Static time borrowing phenomena. (a) Shows the schematic of a datapath implementing static time borrowing, the schematic shows an additional buffer for the **clk** signal of the second group of cells, while the third group receives the same **clk** signal as the first group, thus allowing the time borrowing. (b) Shows the timing diagram of figure (a) which shows the time borrowing behavior by allowing the maximum delay being larger than the period of the reference **clk** signal.

The static time borrow technique depends on the modifications to the **clk** tree therefore can be used with *Flip-Flops* and *Latches* .

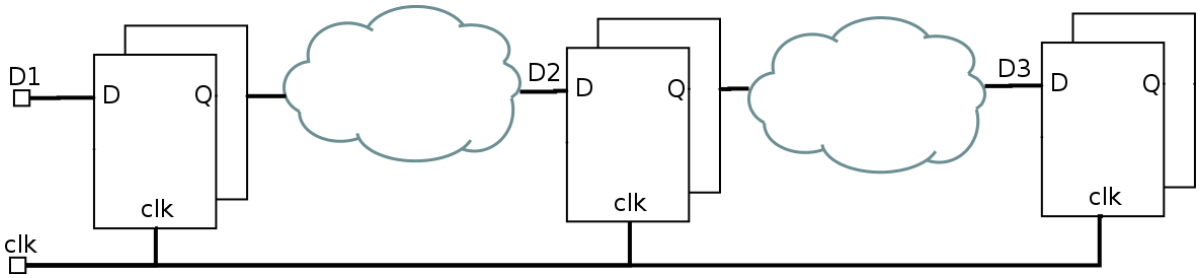
Dynamic time borrowing: Dynamic time borrowing contrary to the static time borrowing can only be achieved by using *Latches* in the design, this is possible thanks to the *Latch* set-up timing restriction given by equation 2.21 which can be re-written as equation 2.37 where:

D_eff**:** Is the total delay in path corrected by **clk** uncertainties

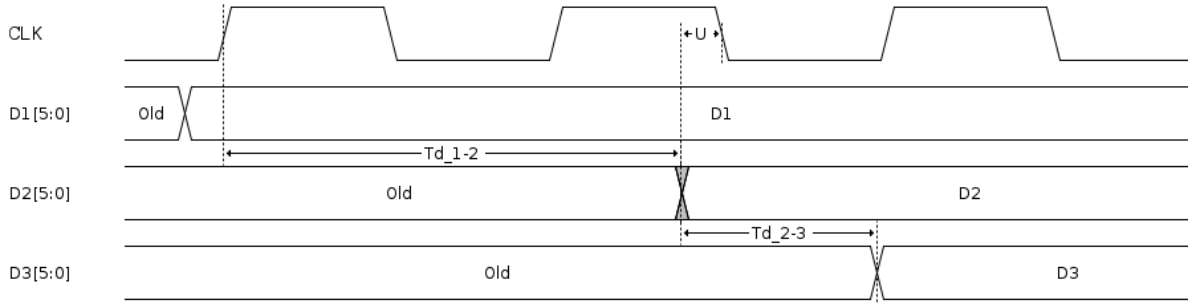
$$T \geq D_{eff} - W \tag{2.37}$$

Thus from equation 2.37 can be deducted that for a given time window W it is possible for the logic delay of the circuit to be longer than the **clk** period. Oklobdzija in [9] makes a full

analysis over time borrowing over a datapath by modifying the set-up timing condition of a group of *Latches* to take into consideration the time borrowing.



(a)



(b)

Figure 2.26: Dynamic time borrowing phenomena. (a) Shows the schematic of a datapath where dynamic time borrowing is possible, in this schematic, the delay is characterized by the slow logic between the first and second group of *Latches.* (b) Shows the timing diagram for the circuit in figure (a), in this timing diagram is possible to observe that the delay between the rising edge of the **clk** signal in the first group of *Latches* and the data arrival on the second group is longer than the **clk** period, still the data si propagated correctly over the design because the delay between the second and third group of cell is smaller.

Figure 2.26 shows the timing diagram of a circuit in which dynamic time borrowing is observed.

2.6.3 Domino circuits

Domino circuits are a special kind of digital circuits that uses *Latches* as sequential elements and not *Flip-Flops,* used specially in high performance systems. To understand why *Latches* are used in domino circuits, dynamic logic must be discussed first.

dynamic logic

Dynamic logic is a family of logic circuits that include in their input list the **clk** signal of the circuit, this is because the dynamic logic is not designed in the same way as standard logic cell using the

CMOS technology.

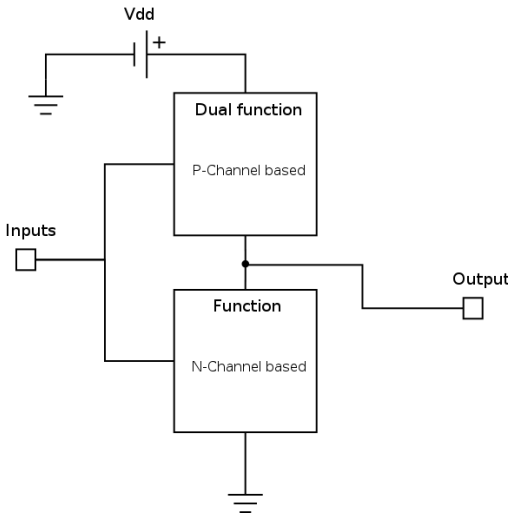


Figure 2.27: Cmos basic design principle

CMOS logic design uses complementary *MOSFet* transistors to realize the logic design. The desired logic function is computed using *N-channel* transistors and connected to the dual version of the desired logic, built using *P-channel* transistors, as shown in figure2.27.

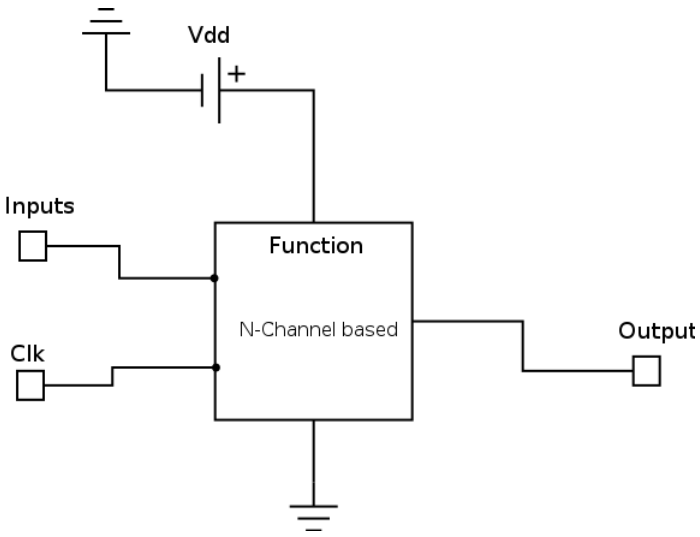


Figure 2.28: Dynamic logic basic design principle

The problems with *CMOS* logic lies in the fact that doubles the number of required transistors for a given logic function, however this is required for reliable functionality, also it's slow as for every transition the transistor must charge the signal (specially true for rising transitions) [11]. Also *P-channel* transistors are slower than *N-channel* [11].

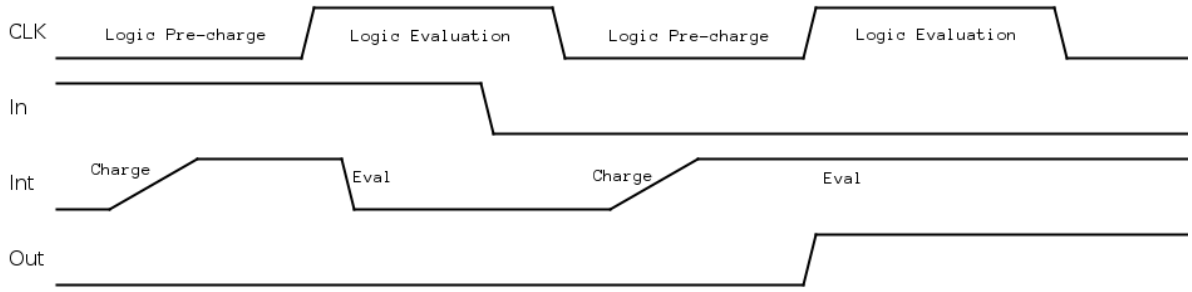


Figure 2.29: Dynamic inverter timing diagram

Dynamic circuits solve that problem by using only one kind of transistors for logic functions and feeding them with the **clk** signal as shown in figure 2.28, so during the time while the **clk** signal is in *Logic* 0 , the transistors are charged to *Logic* 1 so when the rising edge of the **clk** arrives to the logic cells the signal is evaluated at the output effectively halving the number of transistors this design strategy is shown in figure 2.29 which shows the timing diagram of a dynamic inverter. Also given the fact that falling transitions are faster than rising transitions, the circuit is faster than conventional *CMOS* circuits. However this strategy still requires *P-channel* transistors for the signal to be propagated correctly through different stages of evaluation.

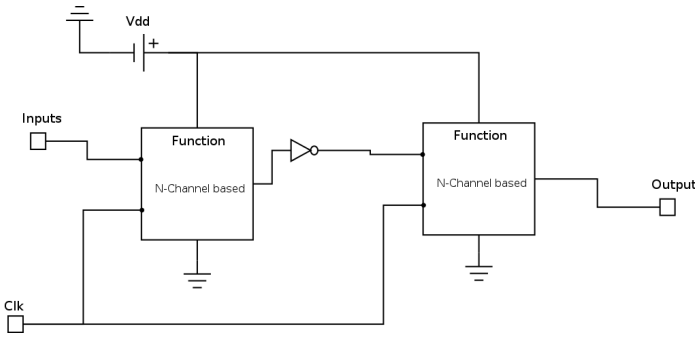


Figure 2.30: Domino Logic design principle

In order to avoid the use of *P-channel* transistors, an inverter logic cell is used between consecutive stages allowing them to be based on *N-channel* transistors only which is known as domino logic [12] and it's illustrated in figure 2.30.

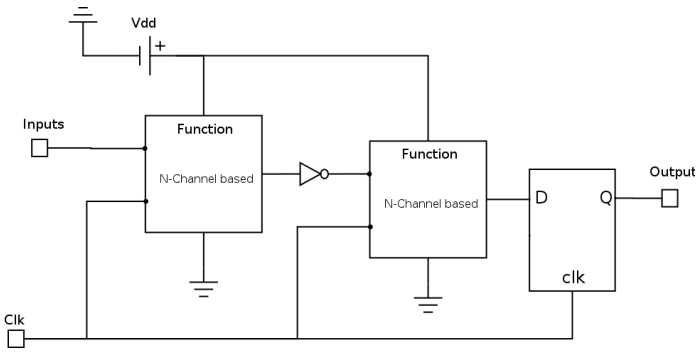


Figure 2.31: Domino Logic stages using a *Latch* to avoid signal degradation

One of the problems that arise with the successive domino logic stages is the progressive signal degradation [12], so in order to avoid this issue sequential cell are required, but given the superior area and power characteristics of *Latches* added to the fact that signal evaluation is bound to a common **clk** signal, which makes the strict synchronous timing behavior of a *Flip-Flop* cell meaningless for the timing analysis, *Latches* are used in a domino logic design as shown in figure 2.31.

2.6.4 Pulsed *Latch*

Even using the analysis made in 2.6.1 circuit designers prefer using *Flip-Flop* over *Latches,* thanks to the better hold timing characteristics of *Flip-Flop* allowing for correct behavior of the circuit with little effort compared to a *Latch* design where the timing must be adjusted taking in consideration the hold timing requirement. The major problem for circuits using *Latches* lies in the hold timing requirement defined by equation 2.35 where the restriction over the fastest logic compared to the *Flip-Flop* hold equation (2.14) is given by the time window during which the *Latch* is transparent caused by the duty cycle of the **clk** signal.

In order to simplify the hold timing analysis of a *Latch* based design, the obvious path is to reduce the duty cycle of the **clk** signal, from equation 2.35 there is no restriction over how much the duty cycle of the **clk** signal can be reduced however the *Latch* set-up timing constraints also depend on the time window, thus equation 2.21 and 2.26 yield the real constrain over the time window.

$$t_{DQ} + D_{L_M} = D_{L_M} + t_{cQ} + T_F + T_R + U - W \tag{2.38}$$

In order to determine the minimum allowable time window both set-up timing restriction must be honored, thus to determine the minimum time window both restrictions are made equivalent, this is because for this analysis, the timing restriction given by equation 2.21 can't be less restrictive that equation 2.26, in other words the minimum period given by the timing restriction from equation 2.21

can't be smaller than the smallest period given from equation 2.26, thus for the smallest possible duty cycle both minimum periods are equal which yields equation 2.38

$$W_{min} = U + (t_{cQ} - t_{DQ}) + T_F + T_R \tag{2.39}$$

Finally by using simple algebraic manipulation over equation 2.38 the minimum time window is given by equation 2.39, from which it can be easily concluded that the minimum duty cycle needed as **clk** signal for a successful *Latch* based design is very small, because all the parameters in the equation are small, which leads to the possibility of using a train of pulses instead of a simple 50% duty cycle **clk** signal in a latch based design.

$$D_{l_m} \geq U + H + (t_{cQ} - 2 \cdot t_{DQ}) + 2 \cdot (T_F + T_R) \tag{2.40}$$

The strategy of using a minimum duty cycle **clk** signal in a *Latch* based design is known as pulsed *Latch.* . For a pulsed *Latch* the hold timing analysis becomes fairly similar to the hold timing analysis of a *Flip-Flop.* When comparing the hold timing requirement for a *Latch* based design for the minimum duty cycle shown in equation 2.40, it's possible to observe that it has the same form as the hold timing requirement for *Flip-Flops* given by equation 2.14; then the minimum logic delay for pulsed *Latches* is about twice the minimum delay for *Flip-Flops* considering the set-up and hold time of the cells being similar.

2.7 Power Consumption in CMOS circuits

The standard technology used in the integrated circuits are CMOS transistor, which is known for having negligible losses in simple circuits, however when the transistor density increases inside an integrated circuit this losses become relevant as they are the primary power consumption inside these circuits heating the device and therefore decreasing their performance, reliability and lifetime [13].

It's important to reduce the power consumption of a circuit, not only because it will reduce its power demands, also because circuits with high power requirements might need ceramic casing in order to allow a more efficient cooling which are much more expensive than plastic ones, also currently many devices operate using battery so reducing the power consumption effectively increases their battery life [14]. High power circuits such as computer microprocessors (50 W or more) are harder to cool requiring active cooling systems (fans, liquid refrigeration) with no proper power management would heat so much that is very likely that today's home computers would be either to slow (in order to allow standard cooling systems) or necessarily require liquid refrigeration in order to function [15].

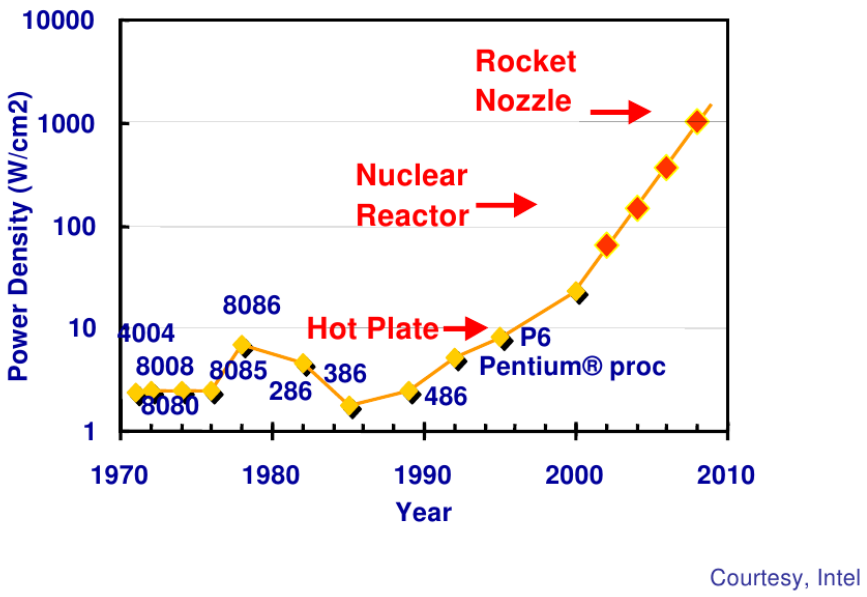


Figure 2.32: Power density increase through time [16]

Nowadays some of the most demanding circuits are microprocessors such as Intel or AMD for desktop and server computers, this chips have been growing their power necessities regardless of the advances in architecture and design techniques whose allow the decrease in the power consumption surpassing the 100 W consumption and the advancements in the technology allowing smaller transistors, thus enabling higher density inside the circuits increases significantly beyond the power density of nuclear reactors as shown in the figure 2.32 [16].

The power consumption of the CMOS transistors comes from 2 main sources:

1. Dynamic power
2. Static leakage power

2.7.1 Dynamic power

Dynamic power consumption which account the largest portion of power consumption in a *CMOS* based circuit, occurs at every input signal transition which means that there are losses for the rising and falling transitions, but still these transitions don't accurately account for the full dynamic power consumption given that the transitions are never sharp enough there will always be a small current for every *intermediate voltage* [14]

Rising and falling transition power consumption

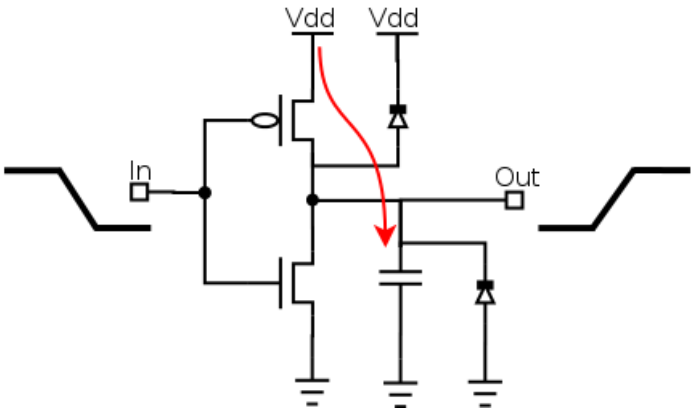


Figure 2.33: Rising edge transition currents, for simplicity of the diagram a simple Inverter i shown

Figure 2.33 shows the current flow inside a *CMOS* based gate during the rising transition and the equivalent capacitor of the gate (and connected wires). During the transition the capacitor must be charged in order to complete the transition in the output signal which represents the effective power consumption of the gate.

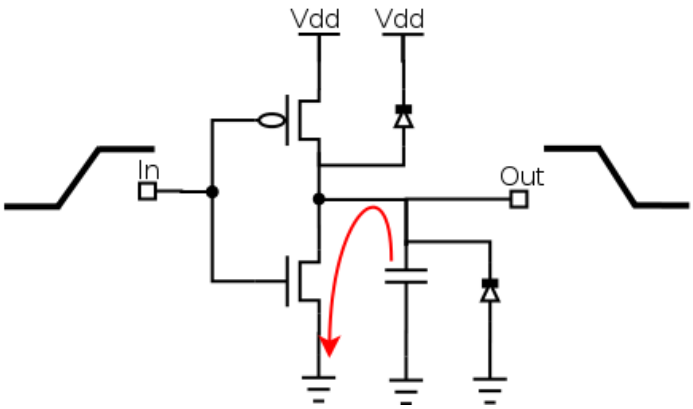


Figure 2.34: Falling edge transition currents, for simplicity of the diagram a simple inverter is shown

On the other hand figure 2.34 shows the flow of currents during a falling transition which shows the discharge of the equivalent capacitor consolidating the power loss of the transistor array. [13]

$$E_{trans} = C_L \cdot V_{dd}^2 \tag{2.41}$$

Thus the energy consumption during a rising/falling transition cycle is given by equation 2.41 where:

C_L**:** Capacitive load of the transistors and wires

V_{DD}**:** Supply voltage

$$P_{trans} = C_{eff} \cdot V_{dd}^2 \cdot f_{clk} \tag{2.42}$$

Still it's necessary to address the power consumption of the gate during it's normal operation which can be expressed as equation 2.42 where:

f_{clk}**:** **clk** signal frequency. This parameter assumes that the entire circuit is controlled by a **clk** signal which is a good assumption in almost every scenario as discussed previously.

C_{eff}**:** Effective capacitance calculated as $C_L \cdot \alpha$

α**:** Is the probability of an output transitions during a clock cycle

V_{DD}**:** Supply voltage

Crowbar current effect

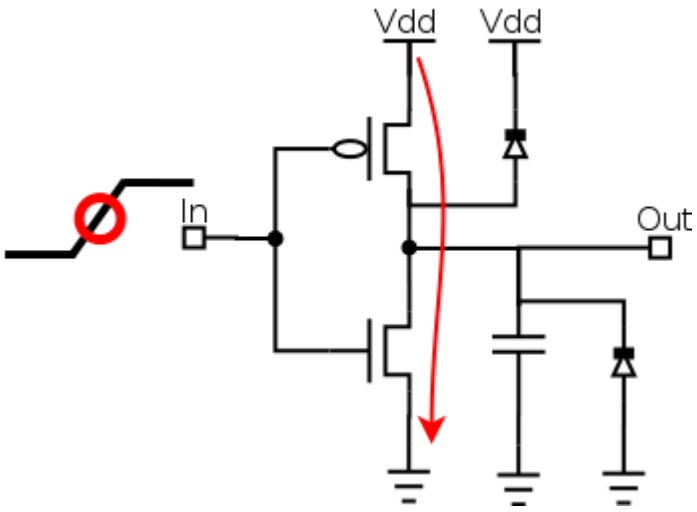


Figure 2.35: *Crowbar-current* during the transition, for simplicity of the diagram a simple inverter is shown

Previously it was mentioned that there was an additional dynamic power consumption source for a *CMOS* gate for an *intermediate voltage* known as the *Crowbar current* which is shown in figure 2.35.

$$P_{crow} = t_{sc} \cdot V_{dd} \cdot I_{peak} \cdot f_{clk} \tag{2.43}$$

The *Crowbar current* occurs during rising and falling transition because the transition is not instantaneous, thus there is a small time during which the all the transistors in the *CMOS* gate are conducing which produces a short circuit current, producing a power consumption given by the equation 2.43 where:

V_{DD}**:** Supply voltage

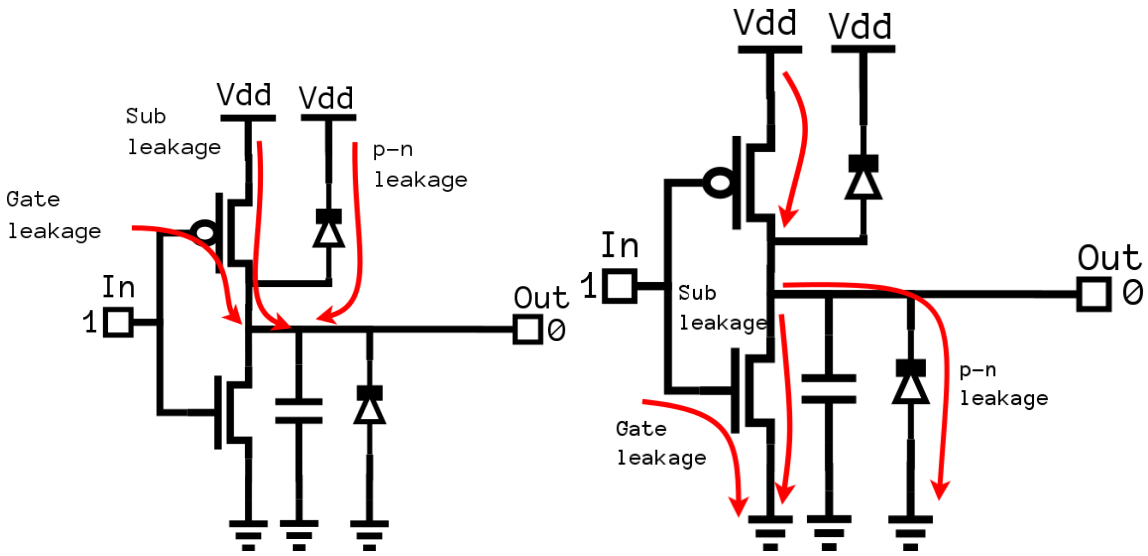
t_sc**:** Time duration of the short-circuit

I_{peak}**:** Total current during the short circuit

f_{clk}**:** **clk** signal frequency. This parameter assumes that the entire circuit is controlled by a **clk** signal which is a good assumption in almost every scenario as discussed previously.

The *Crowbar-current* effect is considered negligible compared to the transition dynamic loss since the short circuit time is short as the edges are expected to be sharp [13], also it's possible to designs transistors in order to fully prevent this effect by setting the transistor parameters accordingly [17]

2.7.2 Static Leakage Power



(a) (b)

Figure 2.36: Leakage currents flow where a simple inverter is used as example to simplify the diagram, however the analysis is valid for any circuit. 2.36(a) Shows the leakage currents that appear in the inverter for the static *Logic* 0 at the input while the output is static in *Logic* 1 2.36(b) Shows the leakage currents that appear in the inverter for the static *Logic* 1 at the input while the output is static in *Logic* 0

When a CMOS gate is in a powered stable state there are small leakage currents considered negligible as power consumption sources when compared to the dynamic losses, however these currents add as the number of transistor increases in a single circuit and eventually lead to a significant portion of the total power consumed by the circuit. These currents shown in figure 2.36 are [14]:

reverse bias p-n junction diode leakage: This leakage occurs in the reverse-bias p-n junction inside the *CMOS* pair, this loss is equivalent to the losses in a diode in the same condition which usually is small. In a *CMOS* pair there are 2 junctions affected.

1. n-type drain in the NMOS to the grounded p-mos sustrate
2. n-well (VDD) to the p-type drian in the PMOS transistor

$$I_{sub-threshold} \approx \mu \cdot C_o x \cdot \frac{W}{L} \cdot V_{th}^2 \cdot e^{\frac{V_{gs} - V_t}{n \cdot V_{th}}} \tag{2.44}$$

Sub-threshold leakage: When a transistor is held in "off" state, but still powered, there is a small current which flows from source to drain terminal which can be approximated by equation 2.44 where:

μ: Electron mobility.

C_ox: Oxide capacitance by area.

V_{th}: Thermal voltage [10]

V_{gs}: Voltage between the gate and source of the transistor. (Tied to the V_{DD})

V_t: Threshold voltage

W: Depth of the channel

L: Length of the channel

Gate leakage: The gate leakage current comes from the tunnel effect of the electrons through the insulating layers. This effect is more relevant with new technologies where the transistors are smaller leading to smaller isolation layers between the gate and drain increasing the possibility of the quantum current [14]

2.8 Low-Power design techniques for Integrated Circuits

Taking in consideration the necessity of reducing the power consumption of an integrated circuit and the mayor sources of the losses, there are several strategies to reduce the power consumed by a circuit at RTL[11] and gate-level with different levels of complexity and success rates using different automation tools [14]

2.8.1 Supply Voltage Reduction

Perhaps the most direct way to reduce the power consumption in *CMOS* devices is to reduce the supply voltage since the dynamic losses which can be explained mainly by the transition losses given by the equation 2.42 where the dependency over the voltage is clearly quadratic, thus reducing the supply voltage is highly effective to reduce the power consumption.

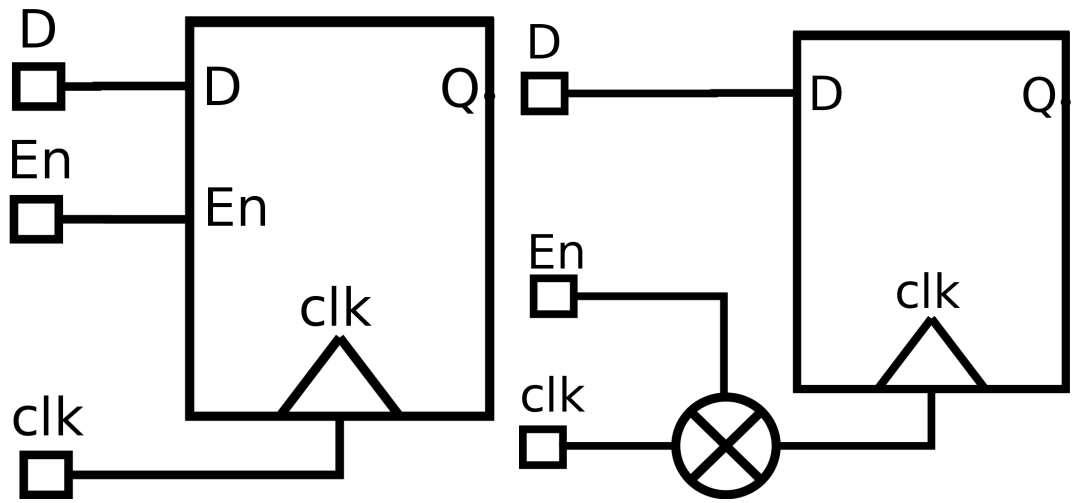
$$I_{ds} \approx \mu \cdot C_{ox} \cdot \frac{W}{L} \cdot (V_{gs} - V_t)^2 \tag{2.45}$$

The mayor issue when lowering the supply voltage of the cell is that it forces the output current to a smaller value which impacts the overall speed of the circuit, since the output current I_{DS} can be written approximately as shown in equation 2.45 , and since the V_{gs} voltage is tightly tied to the input voltage the output current is lowered significantly. In order to keep the overall circuit speed the silicon designers can lower the threshold voltage, which in turn increases the static power consumption of the circuit, which wasn't an issue for the first transistors but now the static power consumption is becoming more important [13]

[10] The thermal voltage is given by $V_{th} = \frac{kT}{q} \approx 25.9mV$ at room temperature

[11] Register Transfer Level

2.8.2 Clock-gating



(a) (b)

Figure 2.37: Basic clock-gating transformation. (a) Shows a *Flip-Flop* with enable pin candidate for clock-gating insertion (b) Shows the same *Flip-Flop* as figure (a) after the clock-gating insertion; clock-gating insertion doesn't modify the output of the cell greatly reducing it's power consumption with a minor impact on the timing of the affected cell.

Clock-gating is the primary method discussed in this work whose aim is to reduce the dynamic losses by disabling the clock input to the sequential element [12] when their input values wouldn't change or when the sequential cell activity depend on a certain enable signal which activates the cell when is set to *Logic* 1 . [13] The cell dependent of the **EN** signal is allowed to modify it's output as long as the **EN** signal stays at *Logic* 1 ; similarly the cell will not be allowed to modify it's output if the **EN** signal is *Logic* 0 . Clock-gating strategies use the enable signal associated to the register in order to insert clock-gating cells, like the one shown in figure 2.37, in order to capture the **clk** signal an keep it constant as long as the enable signal is in the inactive state reducing the power consumption of the cell affected by clock-gating

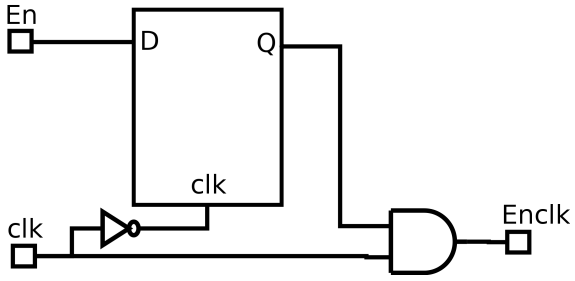


Figure 2.38: A simple clock gate design

[12] Clock-gating is usually performed over *Flip-Flops,* but other sequential elements may be used like *Latches* where the designer must perform the clock-gating insertion manually

[13] Although certain cells can be made in order to be active when the enable signal is set to *Logic* 0

With the insertion of the clock gate many unnecessary transitions in the clock signal applied to the *Flip-Flop* that would force dynamic losses in the cell yet it wouldn't modify it's output thus wasting power. Figure 2.38 shows the most common design of a clock-gate for a positive logic triggered *Flip-Flop.* Naturally the power reduction achieved with the clock-gating strategy scales with more cells gated by the clock-gating cell.

2.8.3 Multivoltage design

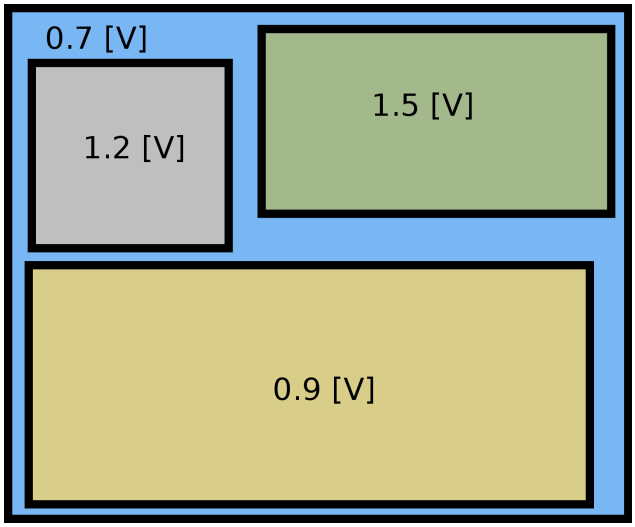


Figure 2.39: Multivoltage design principle

Since lower supply cell consume less power, but are slower than cells which are feed with higher supply voltage, a compromise between performance and low power is achieved using multivoltage designs where inside a single chip blocks with different supplies levels like the one shown in the figure 2.39. This way higher performance cells work located in the high supply voltage power domain[14] at full speed while the lower voltage power domain hosts the lower performance circuits dropping the entire circuit power requirements.

[14]Power Domain: In a multivoltage design the sub-circuit with a specific voltage level is called power domain

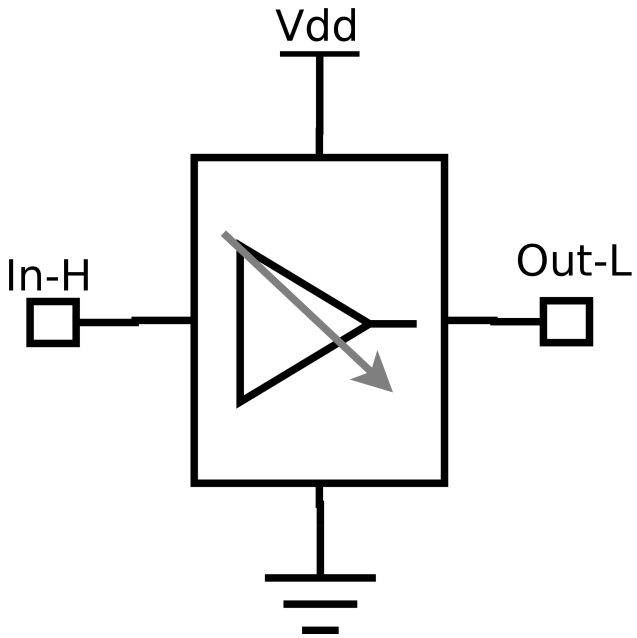


Figure 2.40: Low to high level-shifter [13]

The main disadvantage of the multivoltage design comes when signals travel between different power domains the signal must be adapted to protect the lower voltage circuits and to allow proper behavior in the higher voltage domain, to achieve this a special block is used called level-shifter [13]. These level-shifters can be inserted automatically since they don't alter the logic behavior, however the *Low to high* level-shifter like the one shown in the figure 2.40 introduces delays that should be taken into account

2.8.4 Multiple-Vt library cells

Some CMOS devices can be manufactured with different threshold voltages allowing multiple V_t libraries for the automatic synthesis tool to choose according to the design requirements allowing minimum losses and functional correctness without the intervention of the circuit designer. Different V_t cells have different uses since low V_t cell have high speed and low delays, but they have higher sub-threshold voltage thus making them undesirable when designing low power devices, while a high V_t cell is significantly slower, has lower sub-threshold losses making them ideal when there are no critical path involved in a low power environment.

2.8.5 Power Switching

The power switching technique is performed by shutting of portions of the circuit save power, effectively reducing dynamic and static power consumption of the subcircuits that are not being used. Thanks to this power switching greatly reduce power consumption of the circuit with low

impact on the performance of it.

Power switching techniques require power gates to isolate the target block, a set of specific register to store the data from the circuit at the shut down moment to restore the data in the circuit when the block is turned on and allow the rest of the chip to operate regardless of the power state of the target block.

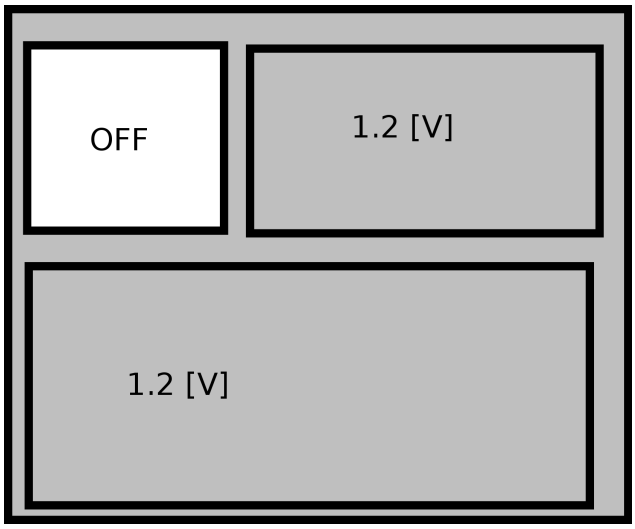


Figure 2.41: Power switching design principle

The power switching design principle shown in figure 2.41, as discussed by Keating on [13] and also treated in [14] keeps the power consumption of an entire block to 0 assuming the area cost that is required to introduce the additional cell which allow the rest of the circuit to keep functioning while the block is turned off. The power switching technique can be applied to a high level of granularity as far as a few cells each time turned off, but to a high area cost. [14]

2.8.6 Dynamic Voltage and frequency scaling

The dynamic voltage and frequency scaling expands the multivoltage approach allowing the supply voltage and the clock frequency of a single power domain to adapt during the operation of the device to match the workload needed from the chip saving power when not needed without sacrificing the maximum performance when required.

Obviously this technique requires the design of a variable frequency clock and a multivoltage supply to match the different workload levels, but the most difficult part it's the proper design and verification for every voltage and frequency combination available since some might change the timing and the behavior of the circuit

Finally it must be noted that all the power reduction techniques described here can be applied simultaneously to a given circuit in order to reduce it's power consumption to the minimum, as long

as the circuit designer is prepared to accept the area or speed cost of such techniques.

2.9 Clock-gating for *Flip-Flop* based designs

Clock-gating which was briefly introduced in 2.8.2, is one of the most widely used techniques to reduce dynamic power consumption since it has only a minor impact on the timing of the circuit [13], it's simple by concept, can be implemented with minor impact on the rest of the circuit and can be easily automated, thus performed by software, while leaving the circuit designers free to develop the circuit without worrying about the clock-gating insertion.

2.9.1 Types of clock-gating cell

Previously 2.9 showed one of the most frequently used clock-gating (*CG*) cell for positive clock-edge triggered *Flip-Flop,* however the same *CG* cell can't be used with a negative edge triggered *Flip-Flop* as it would lead to incorrect behavior of the circuit. Also it's possible to define clock-gating cell without a latch which yields to a total of 4 basic clock-gating types of cells, all of which consider a *Logic* 1 as the active **EN** signal, as another possibility can be easily adjusted using inverters.

Latch-based positive edge triggered

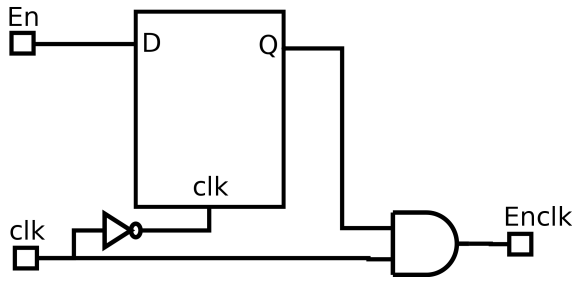


Figure 2.42: Latch based positive edge triggered clock-gating cell

Latch based clock-gating cell are the most common type of *CG* used by circuit designers, these cells are built using a single *AND* logic gate coupled with a *Latch* and an inverter gate as shown in figure 2.42, this configuration protects the rising edge of the **clk** signal which means that anytime that the **EN** signal is active the **clk** signal received by the *Flip-Flop* from the *CG* cell it's the same rising edge as it would see if there is no clock-gating cell, this behavior is shown in figure 2.43

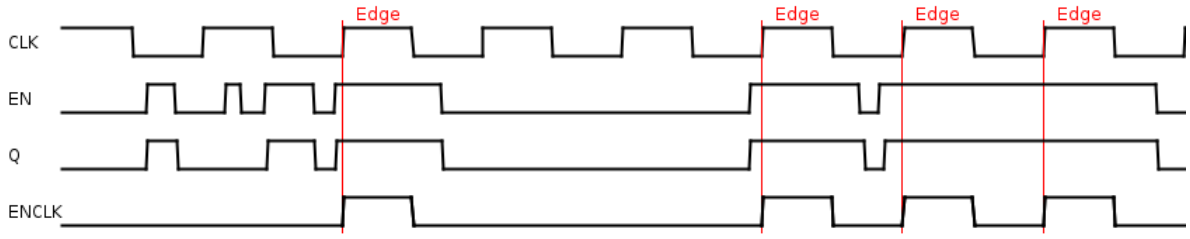


Figure 2.43: Timing diagram for a latch based positive edge clock-gating cell

It's important to note that given the presence of a *Latch* in the *CG* cell the enable signal must comply the set-up and hold conditions for the *Latch* .

Latch-based negative edge triggered

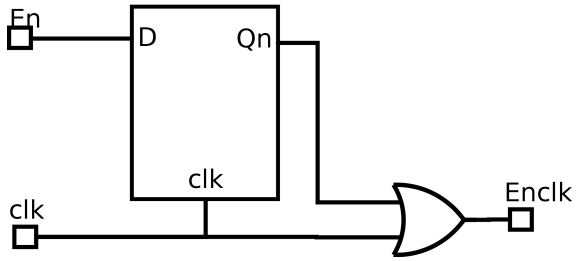


Figure 2.44: Latch based negative edge triggered clock-gating cell

For negative edge triggered *Flip-Flops* the clock-gating cell used is similar to the one shown in figure 2.44 where the *OR* gate allows the protection of the falling edge transition of the **clk** signal against violation in the set-up restriction on the *Latch* in the *CG* cell.

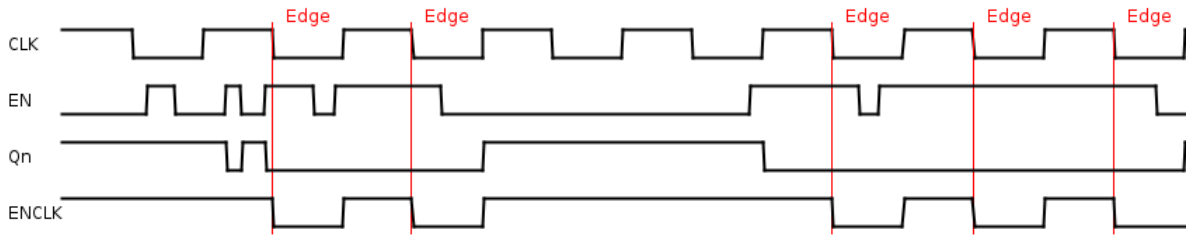


Figure 2.45: Timing diagram for a latch based negative edge clock-gating cell

The latch-based *CG* cell for negative edge triggered *Flip-Flops* timing diagram is shown in figure 2.45 which shows how the configuration of *Latch* and *OR* gate keep the falling edges of the **clk** signal while the **EN** signal is at *Logic* 1 , regardless of how unstable is the enable signal and thus reducing the number of transitions in the clock pin of the gated registers.

Latch-free positive edge triggered

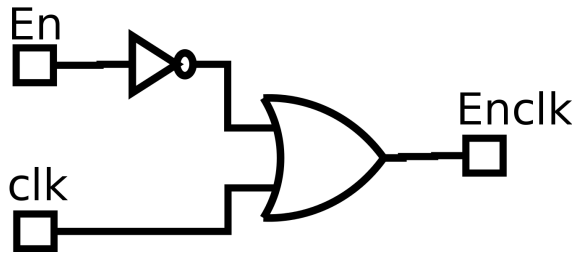


Figure 2.46: *Latch* free positive edge triggered clock-gating cell

The latch-free clock-gate for positive edge triggered *Flip-Flop* shown in figure 2.46, carries an inherent risk for the circuit behavior given the lack of the *Latch* cell since now many transitions in the **EN** signal will go to the **clk** signal of the gated *Flip-Flop* therefore an additional condition is imposed on the enable signal timing when a latch-free *CG* is requested. This restriction imposes that no transition of the enable signal must occur as long as the **clk** signal is in *Logic* 1 therefore the transitions of the enable signal can only occur while the **clk** signal is at *Logic* 0 .

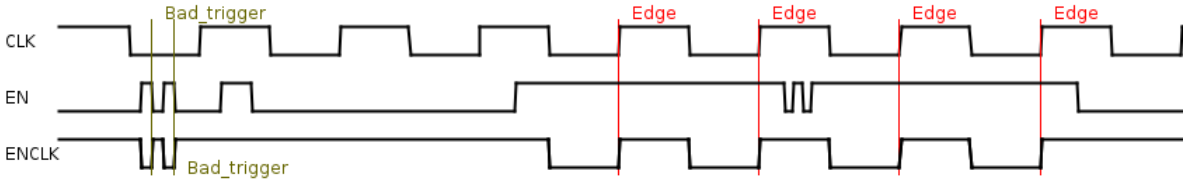


Figure 2.47: Timing diagram for a *Latch* free positive edge clock-gating cell

The timing diagram of the *CG* cell shown in figure 2.47, illustrates the effect of the transitions of the enable signal while the **clk** signal is at *Logic* 1 and how that produces incorrect activation of the gated cells, however if the restriction imposed is honored the rising edge transitions of the **clk** signal are correctly available in the output of the clock-gating cell.

Still it must be answered if such a rigid restriction is imposed over the enable signal, why is the *OR* gate used instead of an *AND* gate since they would yield to the similar results by modifying the condition over the enable signal, which would be completely valid, however the reason lies in the corner cases of the transitions of the **clk** and **EN** signals when both transitions are nearly simultaneous. In such scenario the *OR* gate provides better results for the output signal of the *CG* cell. The timing analysis of the corner cases is left as an exercise to the reader given that is very simple but doesn't provide more useful information for understanding the following parts.

Latch-free negative edge triggered

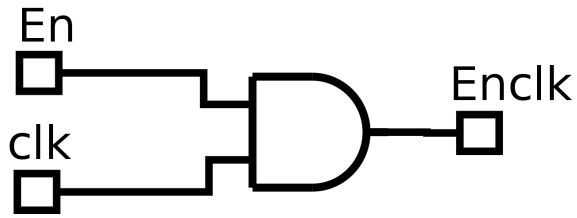


Figure 2.48: latch free negative edge triggered clock-gating cell

The latch-free clock-gating cell for negative edge triggered *Flip-Flops* ,shown in figure 2.48, suffers from the problems as its positive edge counterpart, and it's also limited by a restriction on the **EN** signal which is to allow transitions of the enable signal only when **clk** signal is at *Logic* 0 .

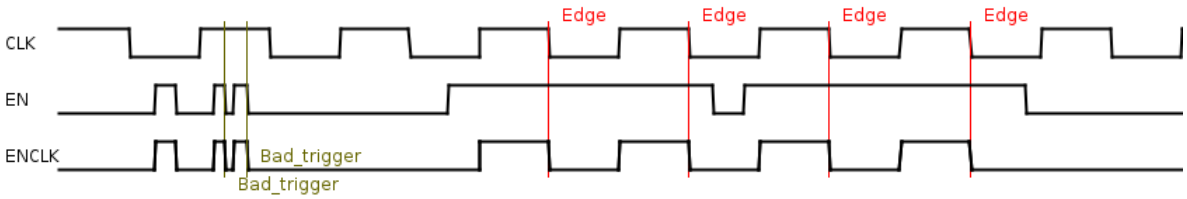


Figure 2.49: Timing diagram for a latch free negative edge clock-gating cell

The timing diagram of the latch-free for negative edge triggered *Flip-Flops* is shown in figure 2.49, where it can be seen the effect of allowing transitions of the enable signal while the **clk** signal is at *Logic* 1 which yield to incorrect trigger of the **clk** signal received by the gated cell thus producing bad results for the circuit. Still the same diagram shows how the *CG* cell is able to provide at the output of itself a the correct falling edge transitions of the **clk** signal.

In the same way as discussed in 2.9.1 the latch free negative edge *CG* cell is built using an *AND* gate instead of an *OR* gate for the protection that the *AND* gate provide against the corner cases of the transitions of the **EN** and **clk** signals.

Despise the problems that might arise when using latch-free clock-gating cell, they are still used by circuit designers since they can provide, good results with a smaller area cost for the circuit.

2.9.2 Basic clock-gating insertion

The basic clock-gating insertion, already introduced in 2.8.2 is based in the identification of the enable signal of the different registers[15] and use it as the enable signal of the clock-gating cell to be introduced. There are 2 basic mechanisms used to identify the enable signal of a candidate cell.

[15]Registers is just another term used to identify *Flip-Flop* cells

Dedicated enable pin in the cell

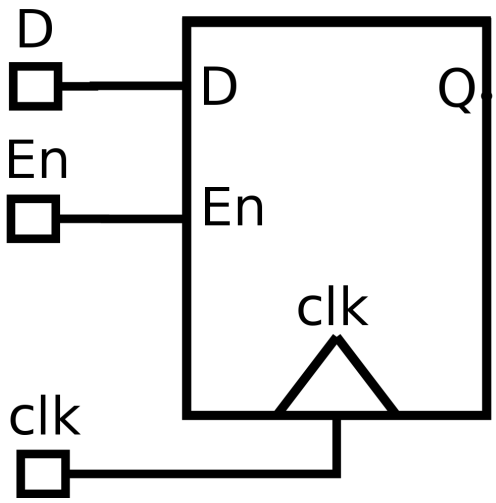


Figure 2.50: *Flip-Flop* with en pin

Some *Flip-Flop* cells have a dedicated **EN** pin which is identified and used as the source of the enable signal to be connected in the clock-gating cell. Figure 2.50 shows a register with an enable pin, these registers still capture the data coming from their **D** pin at the positive rising edge of the **clk** signal, however thanks to the enable pin, the registers only capture data for the next signal if the signal in the **EN** pin is already at *Logic* 1 during the rising edge transition of the **clk** signal. Therefore it becomes clear that the latch based clock-gating cell showed in figure 2.42 whose timing diagrams is shown in figure 2.43 reproduces the expected behavior of the *Flip-Flop* if no clock-gating insertion were performed.

Feedback-loop reduction

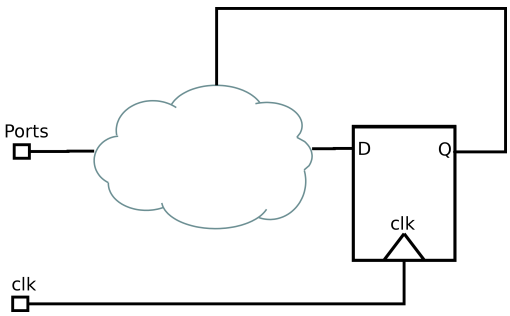


Figure 2.51: *Flip-Flop* with feedback loop

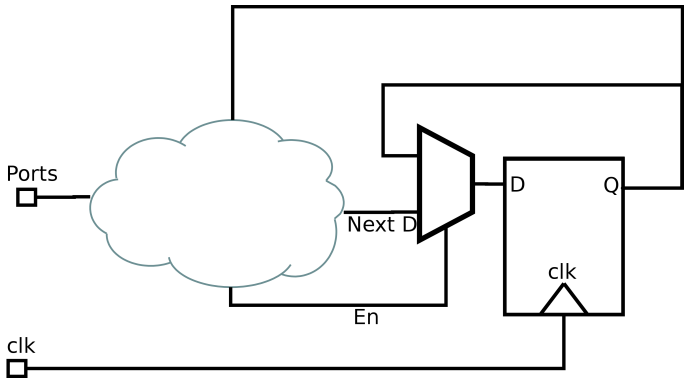


Figure 2.52: *Flip-Flop* with feedback loop with a multiplexer representation

Clock-gating insertion is possible for *Flip-Flops* that don't have an **EN** pin available, if they have a feedback-loop between the registers inputs and outputs, as shown in figure 2.51, in such scenario it might be possible to restructure the combinational logic to represent is as shown in figure 2.52.

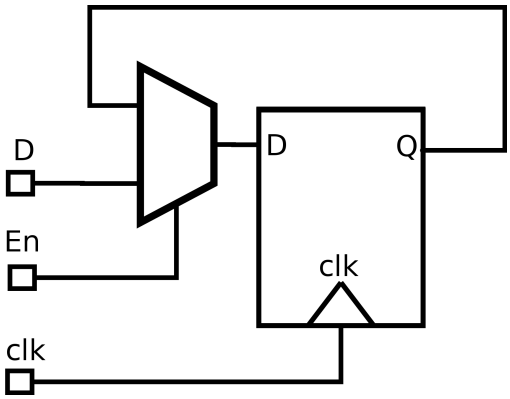


Figure 2.53: *Flip-Flop* with multiplexer and enable signal

The advantage of the representation of the feedback loop with the multiplexer incorporated is the coupling of the multiplexer and the *Flip-Flop* as shown in figure 2.53 because it has the exact same behavior as the *Flip-Flop* with enable pin, therefore it can be gated using the enable signal of the **EN** as the enable signal of the clock-gating cell and removing the multiplexer leaving only the input from the combinational logic.

2.10 Advanced clock-gating insertion for *Flip-Flop* based designs

From now on the clock-gating insertion analysis will consider only positive edge triggered *Flip-Flops* with **EN** pin, unless otherwise stated, as the results and algorithms are equally applicable to the other *Flip-Flop* cell possibilities using the strategies previously discussed. The following advanced clock-gating insertion strategies are state of art, available through *Design Compiler*®

2.10.1 Enhanced clock-gating insertion

So far the clock-gating insertion analysis realized have only considered single registers which, as the reader might have already guessed, is nearly useless and counterproductive when trying to reduce the power consumption of the circuit, therefore circuit designers tend to enforce certain restrictions over the minimum number of cells sharing the same enable condition in order be gated using the clock-gating insertion strategies.

Sometimes banks of registers that aren't big enough in order to be gated given the minimum bank size restriction, can still be gated using only *partially* their enable condition. This process is known as enhanced clock-gating, or partial clock-gating (as the enable condition is used partially)

Use existing clock-gates

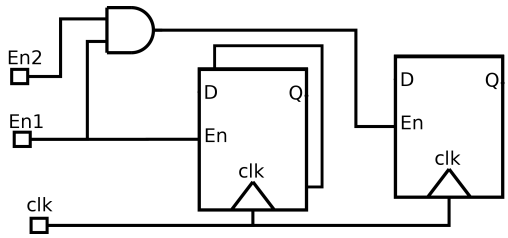


Figure 2.54: A bank of registers sharing the enable condition and a single registers with a more complex enable condition

To understand this clock-gating insertion mechanism, it better to work it through and example, so considering the scenario shown in figure 2.54 where a bank of registers sharing the same enable condition *En1* is shown, also in the same design a single register whose enable condition is given by the logic *AND* between the input signals *En1* and *En2*.

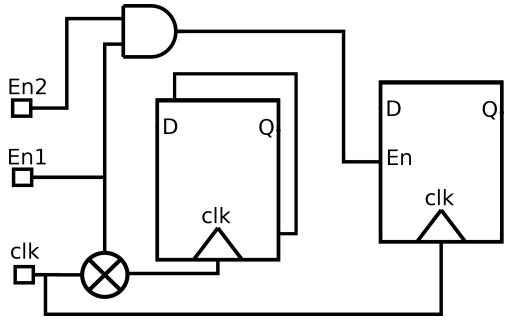


Figure 2.55: Gated bank of registers while a single register isn't gated

Then as discussed previously there is no reason to insert a clock-gate for the single register, while the bank gets a clock-gating cell using the enable condition *En1* as shown in figure 2.55

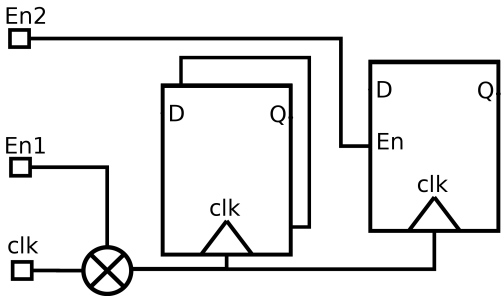


Figure 2.56: Bank of registers sharing the clock-gate with the single register

Finally using the existent clock-gate, the single register can be gated by connecting the output signal of the clock-gate to the **clk** pin of the single register, now the register still retains it's full functionality as if the **clk** signal were connected, but now the cell wastes less dynamic power since it's connected to an existent clock-gate, and the signal *En1* it's redundant for the cell, so it's removed as well, obtaining the circuit shown in figure 2.56

Create new clock-gates

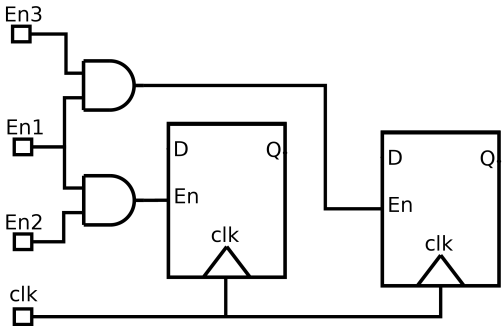


Figure 2.57: Two single registers with their enable conditions shown

Figure 2.57 shows 2 single registers, with different enable conditions, clearly as previously stated, makes little sense to insert a clock-gate for each register

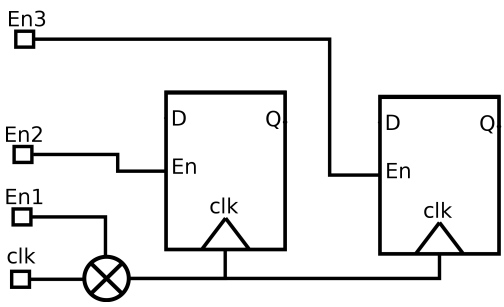


Figure 2.58: Two registers sharing a clock-gate but not completely gated

Even if each *Flip-Flop* have a different enable condition, both registers share the same signal as part of their enable condition (*En1*) therefore using a similar logic as used in 2.10.1 a new clock-gating cell can be created using the signal *En1* as enable condition and use it to gate both registers, therefore the signal *En1* and the *AND* gates driving the enable conditions of the different registers becomes redundant, thus they are modified in the circuit leading to the final result shown in figure 2.58

2.10.2 Multistage clock-gating insertion

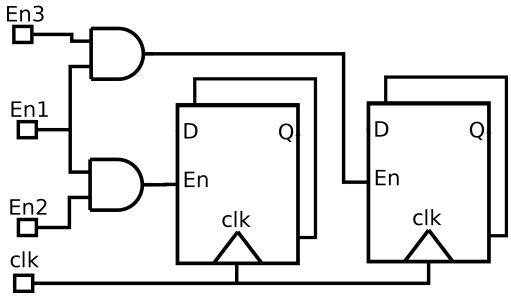


Figure 2.59: Two banks of *Flip-Flops* with their enable conditions, Note the signal *En1* is shared between the banks

Multistage clock-gating insertion is a more aggressive power reduction technique, in which a group of clock-gates sharing partially their enable conditions are gated together using another clock-gate for them, available for banks of *Flip-Flops* with different enable conditions, when at least one component of the enable condition of each bank is shared between the banks as shown in figure 2.59.

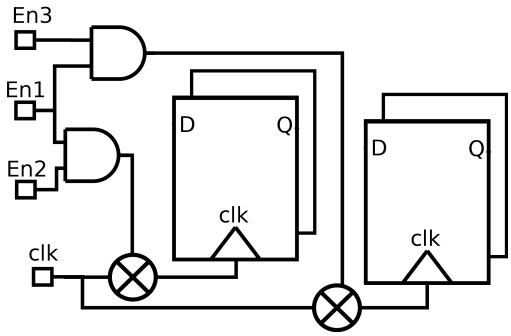


Figure 2.60: Two banks of gated *Flip-Flops* using their respective enable conditions

Unlike the case discussed in 2.10.1 each *Flip-Flop* bank is large enough to be gated by itself using their respective enable condition as shown in figure 2.60, but considering only the clock-gates it's possible to observe the analogy between the scenario of figure 2.60 and the scenario shown in figure 2.57

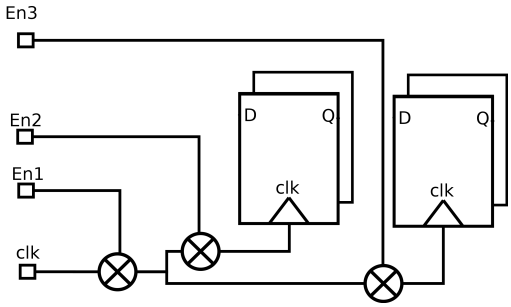


Figure 2.61: Two registers sharing a clock-gate but not completely gated

Therefore by taking advantage of the fact that the clock-gates share the signal *En1* as part of their enable conditions another clock-gate can be inserted using the signal *En1* as enable signal, and connect to it's output the other clock-gates as shown in figure 2.61. Using this strategy the energy consumption of the *CG* cells now that they have a **clk** signal a gated **clk** signal.

2.10.3 XOR Self-Gating

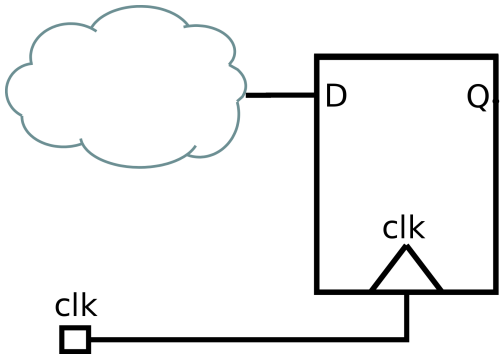


Figure 2.62: Single register candidate for XOR Self-gating, note that the register has no enable condition

The XOR Self-gating technique is a clock-gating strategy where a register without enable condition is gated, such as the one shown in figure 2.62. The idea behind XOR Self-gating as discussed in [18] is to generate the enable condition of the clock-gate using the information of the activity of the data signal arriving to the register, then if that signal has low activity, which means that doesn't change often, it means that the register is candidate for XOR Self-gating.

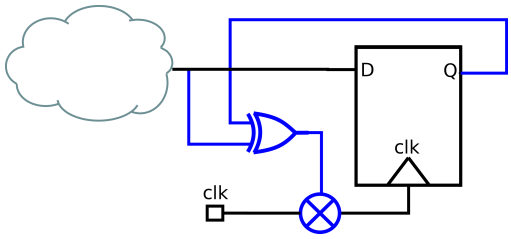


Figure 2.63: Single register gated using XOR Self-gating, note the inclusion of the feedback-loop and the *XOR* logic gate to allow clock-gating insertion

The enable condition for the *Flip-Flop* is created by comparing the input and output of it using a *XOR* logic gate, therefore if both signals are equal, there is no need to update the register, thus the output of the *XOR* gate is the enable signal of the register which is used to fed the enable input of the new *CG* cell inserted for the register, as shown in figure 2.63

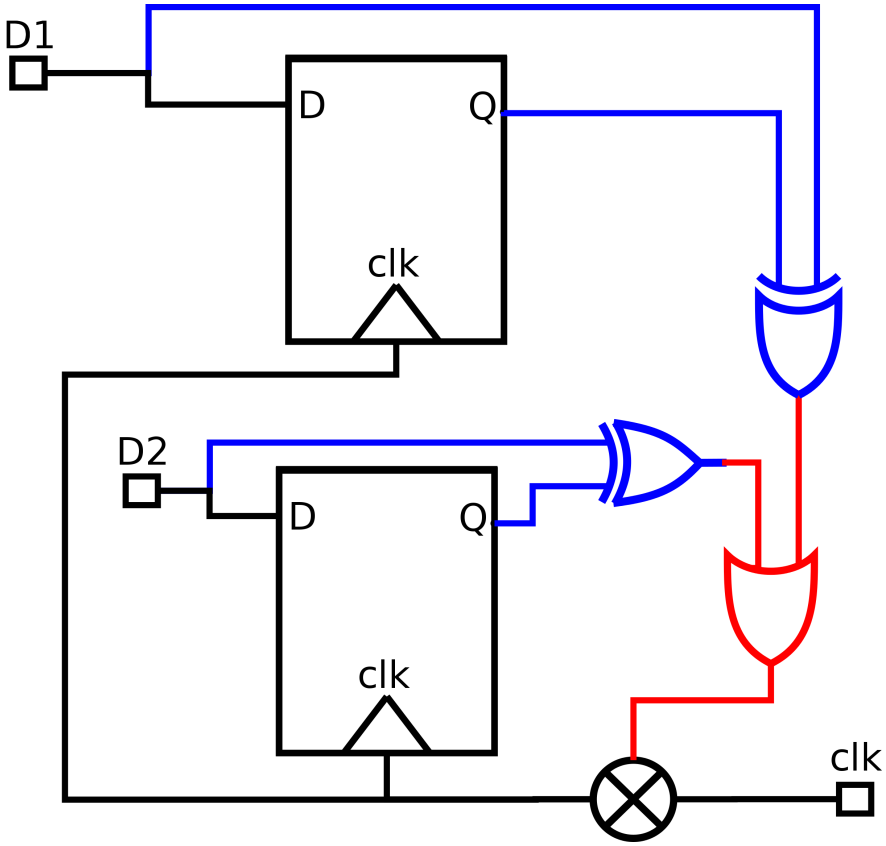


Figure 2.64: Two registers gated using XOR Self-gating

As discussed previously, there is little meaning to insert a clock-gate for a single register, therefore candidate registers for XOR Self-gating are grouped together and activated only if at least one of them must be activated, to do so, *OR* gates are used as shown in figure 2.64.

2.10.4 Other clock-gating operations

Clock-gate insertion is not the only operation performed automatically related to clock-gating cells since circuit designers may have inserted their own clock-gates or users of automatic synthesis tools may want to modify the circuit, in those scenarios the synthesis tool should be able to modify the circuit to fit the new requirements.

Remove clock-gating cells

The clock-gating cells removal is a straightforward operation as it's the reverse operation to the clock-gating insertion so to remove a clock-gate the gated register must be replaced by a register with an enable pin and connect the enable pin of the clock-gate to the enable pin of the new representation of the cell, finally the **clk** pin of the *CG* is connected to the **clk** pin of the *Flip-Flop.* If an adequate replacement for the gated cell can be found, then a multiplexer is inserted.

Balance clock-gating cells load

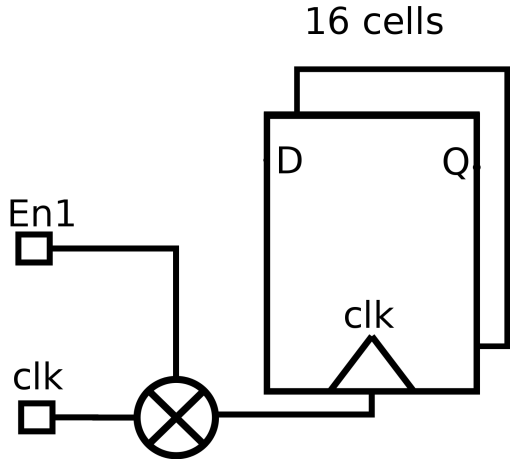


Figure 2.65: Large bank of gated registers for the figure consider the maximum for the *CG* cell to be 12

The clock-gating balance is no other thing that the balance of the register gated by a clock-gating cell or a group of them (that share the same enable condition) but the number of registers gated by a single clock-gate are either too large or to small[16] as shown in figure 2.65 where the number of gated registers is too large for the output of the clock-gating cell so the number of registers connected to each cell must be balanced; if the number of registers by cell is still too large then new clock-gating cells may be created.

[16]Having too many registers gated by the same clock-gating cells may introduce timing problems as the length of the number of wires becomes large increasing the capacitive load in the output of the clock-gating cell which modifies the speed of the rise and fall times of the signal

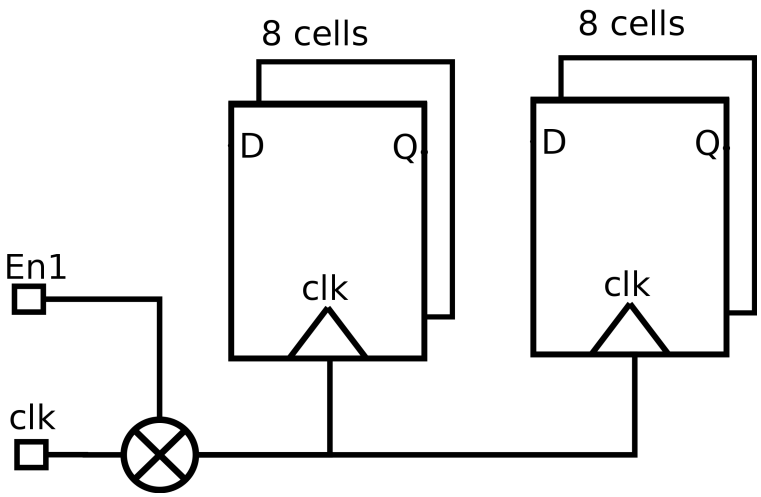


Figure 2.66: Balanced group of gated banks, now the bank is splitted in 2 smaller banks of 8 registers

Figure 2.66 shows the result of a balanced clock-gating cell, now each clock-gating cell gates 8 registers while the maximum is 12 so there are no violations to the circuit requirements and since the number of registers in each *CG* cell is equal the impact of the load on each cell is reduced by the better balance.

2.11 Aproaches to clock-gating for *Latches*

Clock-gating insertion for *Latches* is not currently supported by synthesis tools so when circuits designers need *CG* in their circuits with *Latches* they have to insert them manually which is obviously time consuming and error prone, and even if there is no mistake, an automatic tool should be able to find even more cases where to insert *CG* cells for latches.

2.11.1 Pulser-gating

Pulser gating, as discussed in [19] is the technique in where a clock-gating cell is inserted for pulsed *Latches* [17], because *Latches* consume less power than *Flip-Flop* and as discussed in 2.5 the timing equations for *Latches* depend on the time window or duty cycle of the control signal, therefore by having a very short time window *Latches* timing is close to *Flip-Flop* timing, thus can be replaced directly.

The paper suggest performing regular synthesis of clock-gating using *Flip-Flops* as sequential elements and then replacing the *Flip-Flop* with *Latches* by using a pulse generator in the clock tree, however the paper works using a different clock-gating insertion engine than the discussed here and

[17] A pulsed *Latch,* as it names suggest is a simple *Latch* which is fed with a very short pulse train, or small duty cycle clock, as opposed to the 50% duty cycle used in most circuits

also given the noisy characteristics of the pulse generators, the pulsed *Latch* approach requires the cells to be physically close together.

Chapter 3

Tool Development

3.1 Relevance of the work

Previously it was mentioned that clock-gating is one of the most effective and widely used techniques for reducing dynamic power consumption, however automatic synthesis tool support clock-gating for *Flip-Flop* based designs only, for most designs use *Flip-Flops* as sequential elements given that *Flip-Flop* timing is simpler than *Latch* timing as discussed in 2.5.

Latch based designs do exist since *Latches* have certain advantages over *Flip-Flop* in circuit designs as discussed in 2.6.2, therefore circuit designers are bound to have *Latch* based circuit to consume large quantities of dynamic power, and in order to reduce they have to insert clock-gates manually a task time consuming and clearly error prone while circuits designer would probably skip many gating opportunities or even worse, by inserting the clock-gates prior synthesis might prevent optimizations in the circuit.

Therefore circuit designers often request support for automatic clock-gating insertion for *Latch* based designs to companies that develop the automatic synthesis software such as *Synopsys* developer of the *Design Compiler*® tool. In order to realize this work a *Synopsys* customer allowed *Synopsys* employees access to a *Latch* based design of their own as a testing platform for the feature in development

3.2 Auxiliar tools developed

One of the first challenges faced, during the development of automatic clock-gating for *Latches* in the *Design Compiler*® synthesis tool, was the fact that such feature never existed, thus no tools were available for analyzing the results of the clock-gating insertion in a *Latch* based design, if

anything the current report tools[1] would show an increase in the number of clock-gating cells, but wouldn't provide any information regarding if the clock-gates are actually gating *Latches* or if they are properly inserted, and any other information should be extracted using the graphical user interface (*GUI*) of *Design Compiler®* .

Obviously in order to allow ease of verification on the correctness of the tool in development some auxiliary tools had to be developed, but this tools couldn't be embedded in the code for the feature being a prototype and implementing a report would be disruptive, therefore a set of scripts were created using the scripting language used in *Design Compiler®* , *Tcl* coupled with some already existent commands embedded in *Design Compiler®*

3.3 Timing analysis of a gated *Latch*

In 2.4 and 2.5 the timing analysis of *Latches* and *Flip-Flops* was introduced, also 2.9 realized a brief description of the clock-gates available for *Flip-Flops* and how they reproduced the behavior of the *Flip-Flop* depending on the clock-gating cell and certain restrictions. The same patterns can be applied for *Latches* in order to define the adequate clock-gating cells.

Let's begin the analysis by considering the behavioral description of a *Flip-Flop* with enable signal and comparing it to the behavioral description of a *Latch* with the same pins, for this purpose the *Verilog HDL* is used.

```
1  module EnFFD
       (D,En,Q,clk);
     input [0:0] D;
     input [0:0] En,clk;
     output reg [0:0] Q;

6    always@(posedge clk)
       begin
       if (En == 1'b1)
         begin
         Q<=D;
       end
     end

11
   endmodule;
```

Listing 3.1: Verilog description of a *Flip-Flop* with **EN** pin

```
   module EnLatch (D,En,Q,clk);
2    input [0:0] D;
     input [0:0] En,clk;
     output reg [0:0] Q;

     always@(clk,D,En) begin
7      if ((En && clk) == 1'b1) begin
         Q<=D;
       end
     end

12 endmodule;
```

Listing 3.2: Verilog description of a*Latch* with **EN** pin

From the behavioral description it becomes evident the fact that the difference between the **EN** and **clk** pin of a *Latch* is only by their name making them indistinguishable, which means that the obvious choice of a *CG* cell for a *Latch* based design is a latch-free clock-gate *AND* based, as

[1]The report tools here described are a series of *Synopsys* command starting the *report* prefix that print in the screen useful information about the cells or the designs according to the information requested. In this case mainly reports about clock-gating

it reproduces perfectly the behavior described by the code fragment 3.2, thus requires no further analysis.

The testcase provided by *Synopsys* customers, which already had some gated *Latches,* showed that some of them shared their *CG* cell with similar *Flip-Flop,* and the shared *CG* was a Latch-based one, which means that these kind of *CG* cells are expected in a *Latch* based design, thus they must be analyzed.

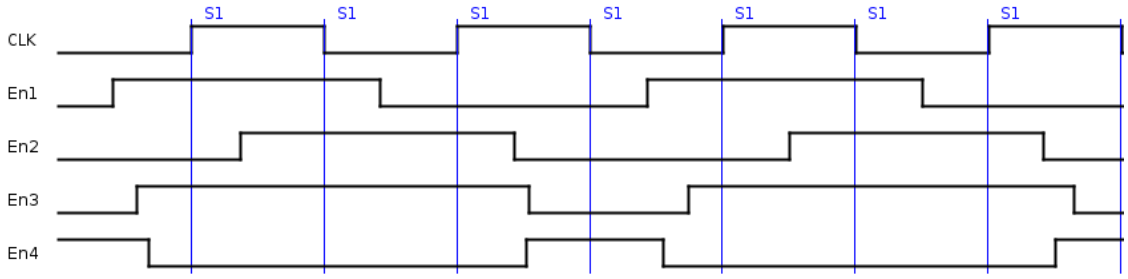


Figure 3.1: Reasonable possible **EN** signals as input in a *Latch* based design

When analyzing a *Latch* based design, the discussion over the different types of digital designs in 2.3 should be taken in consideration, as it states that basically most (if not all) complex digital circuits are synchronous, therefore a *Latch* based design is most likely to behave in a synchronous[2] fashion, which implies that random states transition are unlikely and mostly undesired, which can be summarized in the following statements:

1. For a positive-edge *Latch* the state should be stable as long as the sequential elements are inactive.
2. The enable condition of a *Latch* in the design should be stable during a clock period.
3. Therefore the enable condition of the *Latch* should be stable during any clock-edge.

The previous statements provide a restriction over the enable condition, therefore only the enable conditions shown in figure 3.1 represent the expected "behaviors" in a *Latch* based design.

En_1**:** The transitions of the enable signal are allowed only while the **clk** is at *Logic* 0 so the signal is stable as long as the clock is at *Logic* 1

En_3**:** The transitions of the enable signal are allowed only while the **clk** is at *Logic* 1 so the signal is stable as long as the clock is at *Logic* 0

En_3**:** The enable signal can only stabilize itself at *Logic* 1 if the **clk** signal is at *Logic* 0 , and can only stabilize to *Logic* 0 if the clock is at *Logic* 1

[2]In this case, the *Latch* based design is considered *synchronous* because all the state transitions are bound to the clock signal even if the *Latch* has an asynchronous behavior.

En_4**:** The enable signal can only stabilize itself at *Logic* 0 if the **clk** signal is at *Logic* 1 , and can only stabilize to *Logic* 1 if the clock is at *Logic* 0

This analysis was verified with the customer's own engineers and designers which confirmed the analysis by saying that in a *Latch* based design they expected the enable signal to behave as En_1 for a positive driven *Latch* .

Using this information and comparing it with the state of art of clock-gating for *Flip-Flop* based designs, it became evident that the relation between clock-gates and enable condition was very similar between both cases, but reversed, as for *Flip-Flops* the latch-based *CG* cells reproduce the behavior of the cell perfectly and the latch-free clock-gating required the verification of the characteristics of the enable condition, meanwhile for *Latch* based designs, latch-free clock-gating cells reproduce the behavior of the cell and the latch-based *CG* cells required a timing analysis on the enable condition for it to preserve the behavior of the circuit.

Therefore automatic clock-gating insertion for *Latch* based designs is feasible and similar enough to the *Flip-Flop* based designs scenario, as to recycle the algorithms used in the tool.

3.4 Analysis of the clock-gating insertion algorithm

Design Compiler® provides automatic clock-gating insertion as part of the synthesis process during the execution of the `compile` or `compile_ultra` commands which differ on the optimization algorithm used, but from the clock-gating insertion perspective there is no significant difference, at least for the clock-gating insertion algorithm discussed in this work.

3.4.1 *Design Compiler*® `compile` command flow

The present works assumes that the reader has some level of familiarity with the *Design Compiler*® tool developed by *Synopsys* , therefore no tutorial or introduction will be presented about this tool.

The compile flow corresponds to all the process realized by *Design Compiler*® during which the circuit design is logically optimized, mapped and constraints are applied to the cells in the design; also depending on the mode, it performs a basic routing of the cells in order to improve the timing optimization. The design once compiled though the `compile` or `compile_ultra` commands is ready to be processed by a place and route tool which is the following step in a circuit design.

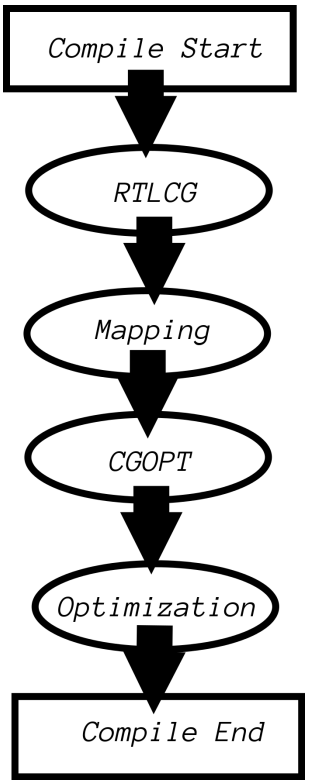


Figure 3.2: Rough representation of the compile flow performed by *Design Compiler*®

The compile flow can be understood, for clock-gating insertion perspective, as shown in figure 3.2 where each step represents:

1. *Compile Start* At the moment of compile start the design is already loaded in the tool and all the configurations and variables necessary during compile are set.

2. *RTLCG* Is the first clock-gating insertion engine used during compile, only works with RTL designs before mapping, thus its name.

3. *Mapping* In the mapping stage the combinational logic is optimized, the feedback loops removed and cell are mapped, which means that the generic representation of the cell is replaced by their specific representation according to the configuration and which has the full information about the building of the cells (area capacitance etc).

4. *CGOPT* The second clock-gating insertion engine, this one works over RTL and mapped cells so it's used to finish insert clock-gates in any instance left behind by the *RTLCG* engine and to perform further optimizations on the already gated cells.

5. *Timing* This stage performs the timing and area optimizations which makes it the longest part of the process.

3.4.2 *CGOPT* flow

The clock-gating insertion for *Latches* prototype feature is developed only in *CGOPT* since in *Design Compiler*® only mapped *Latches* have an **EN** pin aside the **clk** pin, because there is no way to distinguish the **EN** and **clk** pin in a *Latch* from their behavioral description so *Design Compiler*® makes no difference between both pins, thus it doesn't have a model for the *Latch* with **EN** and **clk** pins, until the cells are mapped, therefore providing a prototype using the *RTLCG* insertion engine would require an additional amount of effort only to distinguish the enable and clock signals which are already available in the mapped cell, thus available in during the *CGOPT* engine.

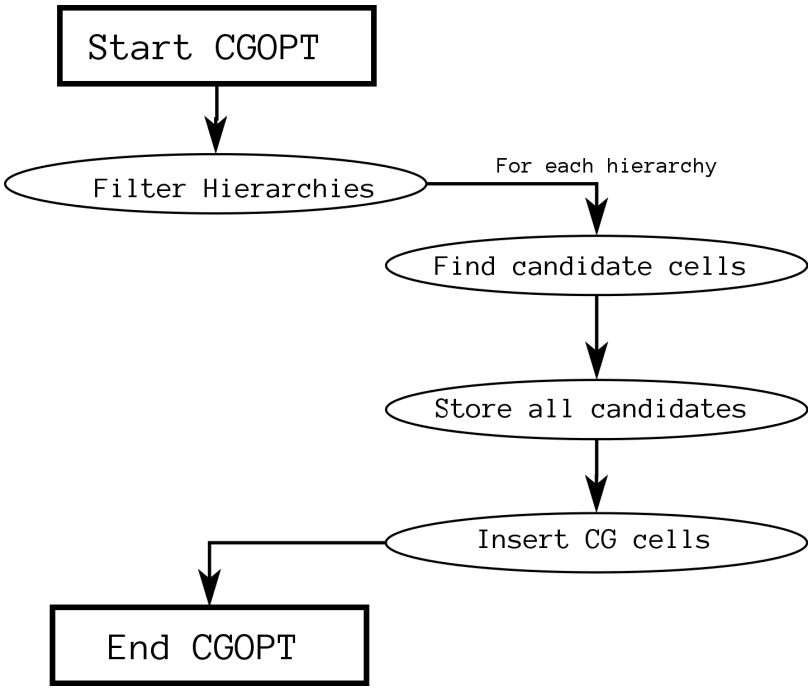


Figure 3.3: Rough representation of the *CGOPT* insertion engine performed during *Design Compiler*®

The *CGOPT* insertion engine roughly follows the algorithm showed in figure 3.3 where:

Start *CGOPT* : This is the stage of the flow, where the *CGOPT* insertion engine is started, the user configurations are loaded and most variables required thought the process are set

Filter hierarchies: Each hierarchy (or module using verilog notation) may or may not have candidate cells for clock-gating insertion, so all the hierarchies that don't have sequential cells, are *CG* cell are filtered in this stage to avoid unnecessary calculations later on.

Find candidate cells: For every hierarchy with possible candidate cells, each cell is verified to determine if it's possible to gate the cell, at this stage all the cells that aren't registers (*Flip-*

Flops) are filtered out, also any register that has already been gated is filtered, the only exception to this rule are *CG* cells as they could be used to gate more registers.

Store candidate cells: Each candidate cell, is processed before being stored, during this process the enable condition of the cell is calculated and then the candidate cell along with the enable condition and other parameters are stored. Finally when the once all the cells are stored, they are grouped by enable condition and **clk** signal in banks.

Insert *CG* cells: For every bank, using the enable and **clk** signal stored and comparing it to the same signals of the already exiting *CG* cells to check if an already existing *CG* can be used, if no *CG* cell exists then a new cell is created and all the cellls in the bank are connected to the *CG* cell.

3.5 Clock-gating for *Latches* implementation

Considering the already existing code dealing with clock-gating insertion for *Flip-Flop* based designs, and the availability of *CG* cells types the following restrictions were applied to the prototype implementation:

1. Reuse the largest possible amount of code and existing algorithms for the *Latches* case
2. Consider only latch-based cells for insertion.

Considering only *Latch* -based *CG* cells may not be as correct as considering latch-free cells, but in order to provide support for *Latch* free cells the whole *CG* cell selection engine would have to be re-implemented which escapes the purpose of generating a prototype.

3.5.1 New Variables

Perhaps the most important modification made to the code is the inclusion of a variable to activate the feature, and another variable to print helpful debug messages. This variables were created using an already existent mechanism for increased speed of execution at variable setting and reading.

The new variables are:

power_cg_allow_cg_for_latches: When this variable is turned on, the clock-gating for latches is enabled.

```
boolean cgopt_get_var_power_cg_allow_cg_for_latches(void)
{
 return power_cg_allow_cg_for_latches;
```

```
}
```

Listing 3.3: Relevant fragment of code that recovers the value of the variable *power_cg_allow_cg_for_latches*

pwr_cg_print_msg_in_cg_for_latch: When this variable is turned on, debug message printing is allowed during clock-gating for *Latches,* therefore this variable hold no meaning unless *power_cg_allow_cg_for_latches* is turned on.

```
boolean cgopt_get_var_pwr_cg_print_msg_in_cg_for_latch(void)
{
  return pwr_cg_print_msg_in_cg_for_latch;
}
```

Listing 3.4: Relevant fragment of code that recovers the value of the variable *pwr_cg_print_msg_in_cg_for_latch*

3.5.2 Functionality modifications

In order to provide the prototype for clock-gating for *Latches* different parts of the code had to be modified, in different parts of the *CGOPT* insertion flow.

Filtering hierarchies

True to the spirit of recycling as much as possible the already existent algorithms in clock-gating for *Flip-Flops,* the modifications to the code are mostly to allow *Latches* to get past the filters when the variable *power_cg_allow_cg_for_latches* is activated.

```
dc_for_all_child_cells(alt, cell) {

  /* Check if the cell has valid atributes */
  if (dc_cell_is_sequential_or_has_sequential_arcs(cell) &&
      !dc_cell_is_master_slave(cell) ) {
    /* add flops */
    if (clkgt_dc_cell_is_register(cell)) {
      found_seq_cell = TRUE;
      break;
    } else if(cgopt_get_var_power_cg_allow_cg_for_latches() &&
              dc_cell_is_latch(cell)) {
      /* Adding latch if not inside a cg cell */
      if (!clkgt_parent_cell_is_cg_cell(cell)){
        found_latch = TRUE;
        break;
      }
    }
  }
} dc_end_for;
```

Listing 3.5: Relevant fragment of code that allows hierarchies with latches not to be discarded during the hierarchies filtering

The code fragment 3.5 shows the implemented code that allow hierarchies compromised only with *Latches* to not be filtered out of the clock-gating insertion mechanism. In this code fragment, there are functions already implemented which have to be explained in order to understand the code fragment.

`dc_for_all_child_cells` : Variant of the well known *while* in the *C* programming language.

`dc_cell_is_sequential_or_has_sequential_arcs` : This function verifies if the cell in question has sequential attributes (like *Flip-Flops* and *Latches* have).

`dc_cell_is_master_slave` : This function verifies that the cell in question doesn't have a Master/Slave configuration

`clkgt_dc_cell_is_register` : Verifies if the cell is a *Flip-Flop*

`dc_cell_is_latch` : Verifies if the cell is a *Latch*

`clkgt_parent_cell_is_cg_cell` : Verifies if the *Latch* is not inside a *CG* cell hierarchy.

`found_latch` : Is a parameter used to not remove the hierarchy from the hierarchies to verify when searching for candidate cells later on.

Find candidate cells

The process of finding candidate cells, is actually the process of discarding progressively cells that cannot be gated, either for not being supported (such as *Latches*) of for the cells simply can't be gated (like being defined by the user as not a candidate cell for clock-gating), therefore this process had to be intervened to allow clock-gating for *Latches.*

```
1  if(!only_cgs &&
      dc_cell_is_sequential_or_has_sequential_arcs(cell)) {

     if(clkgt_dc_cell_is_register(cell) &&
      can_gate_reg(cell)) {
6
     cgopt_add_candidate_reg(cell, TRUE /* for cgopt */);

     } else if(cgopt_get_var_power_cg_allow_cg_for_latches() &&
               dc_cell_is_latch(cell)&&
11             can_gate_reg(cell)) {

       if(cgopt_get_var_pwr_cg_print_msg_in_cg_for_latch()){
         msg_printf("Hidden variables succsesfull \n");
   }
16
       /* add latches to candidate register */
       cgopt_add_candidate_latch(cell, TRUE /* for cgopt */);
     }
```

```
}
```

Listing 3.6: Relevant fragment of code that allows *Latches* to be considered as candidate cells, instead of discarding them.

The first filtering process, allows only *Flip-Flops* to be gated, thus this filter had to be modified accordingly, which is shown in the code fragment 3.6, where additional parameters and functions are introduced.

`only_cgs` : This boolean variable is used to identify hierarchies that only have clock-gating cells.

`can_gate_reg` : This function makes further filtering on the registers to be gated verifying among other thing if the user used setting to avoid the gating of a certain cell.

`cgopt_add_candidate_latch` : This function stores a valid candidate *Latch.*

`cgopt_add_candidate_reg` : This function stores a valid candidate *Flip-Flop.*

```
/* Skip latches whose co pin can't be found by rtdc */

if (ret && dc_cell_is_latch(reg) &&
    cgopt_get_var_power_cg_allow_cg_for_latches() &&
    ((dc_pin)rtdc_cell_get_co_pin(reg) == NIL(dc_pin)))
  {
    ret = FALSE;
    if (cgopt_get_var_pwr_cg_print_msg_in_cg_for_latch()) {
      msg_printf("skipping latch because the clock_pin couldn't be found \n");
      msg_printf("skipped latch %s \n",(string)dc_object_get_name(reg));
      msg_printf("skipped libcell %s \n",
                 (string)dc_object_get_name(dc_lib_cell_of_cell(reg)));
    }
  }


/* Check if the register has multiple clock pins, only for mapped regs */
if(ret &&
   !dc_cell_is_seqgen(reg)) {
  clk_pins = clkgt_dc_cell_clock_pin_count(reg);
  if(clk_pins == 0) {
    ungated_reason = PWR_CG_UNGATED_REGISTER_NO_CLOCK;
    ret = FALSE;
  } else if(clk_pins > 1) {
    if(!dc_cell_is_latch(reg) || !cgopt_get_var_power_cg_allow_cg_for_latches())
      {
      ungated_reason = PWR_CG_UNGATED_REGISTER_MULTIPLE_CLOCKS;
      ret = FALSE;
    }
  }
}
```

Listing 3.7: Relevant fragment of code that include verification to allow only suitable *Latches* to be added as candidate cells.

The second filter is obviously the function `can_gate_reg`, which had to be modified in order to allow *Latches* to be considered as valid candidate registers. Since this function was originally developed considering no *Latches* to arrive this function ever, there was no explicit filter for *Latches,* however there was an indirect filter in the function, because the function would filter any sequential cell with more than 1 pin with **clk** pin functionality and, as previously stated, the functionality of the **clk** and **EN** pins isn't distinguishable in a *Latch,* thus *Latches* were filtered out.

In later stages of the development, a problem arose with certain types of *Latches* making the tool crash in certain circumstances, thus a filter had to be added to kept problematic *Latches* out of the possible candidate cells. The code 3.7 shows the relevant fragments of the code modified to get the expected functionality.

`rtdc_cell_get_co_pin` : This function finds unequivocally the **clk** pin of a cell, which is particularly useful for *Latches,* therefore if that function can't find the **clk** pin of the cells (regardless of the cell actually having one) the *Latch* isn't considered as a valid candidate[3]

Storing the cells

As previously discussed the function `cgopt_add_candidate_latch` stores a *Latch* as a valid candidate cell, this function was created to mimic its counterpart `cgopt_add_candidate_reg`, but taking in consideration some subtleties in the *Latch* cell handling.

```
/* specific clk pin -> latches */
specific_clk_pin = rtdc_cell_get_co_pin(reg);
specific_clk_net = specific_clk_pin?
  dc_net_of_pin(specific_clk_pin):
  NIL(dc_net);
specific_clk_is_inverting = rtdc_cell_co_pin_is_inverting(reg);

/* specific en pin => enable pin in latch */
specific_en_pin = rtdc_cell_get_sl_pin(reg);
specific_en_net = specific_en_pin?
  dc_net_of_pin(specific_en_pin):
  NIL(dc_net);
specific_en_is_inverting = rtdc_cell_sl_pin_is_inverting(reg);

if(specific_en_net) {
  pwrbdd_add_target_net(pbm, specific_en_net);
}

specific_clk_bdd = pwrbdd_formula_of(pbm, specific_clk_net);

/* taking into consideration we are storing the external pins */
cand = candidate_latch_obj_create(reg, pbm, specific_clk_bdd, en_bdd, ns_bdd,
                                  fb_is_external, FALSE, specific_clk_pin,
```

[3]Patching this function to find the **clk** pin in some problematic cells is completely out of the scope of the prototype

```
                                    specific_clk_net);
```

Listing 3.8: Relevant fragment of code that corresponding to the storing of *Latches* as candidate cells.

The function `cgopt_add_candidate_latch` calculates the feedback-loop (if any) and the representation of the logic functions connected to the **clk** and **EN** pins (if the **EN** pin exists). These computations are performed in a slightly different way as they are done in the `cgopt_add_candidate_reg` function and the code fragment 3.8 shows the code lines in which both functions differ.

```
static cgopt_candidate_reg
candidate_latch_obj_create(dc_cell reg,
                           pwrbdd_manager pbm,
                           bdd_formula clk_bdd,
                           bdd_formula en_bdd,
                           bdd_formula ns_bdd,
                           boolean has_fb,
                           boolean has_non_standard_sync_pins,
                           dc_pin clk_pin,
                           dc_net clk_net)
{
  cgopt_candidate_reg cand;

  cand = candidate_reg_obj_create(reg, pbm, clk_bdd, en_bdd, ns_bdd, has_fb,
                                  has_non_standard_sync_pins);

  cand->reg_clk_pin=clk_pin;

  cand->reg_clk_net=clk_net;
  return cand;
}
```

Listing 3.9: Function `candidate_latch_obj_create` as an extension of the function `candidate_reg_obj_create`

```
typedef struct cgopt_candidate_reg_s cgopt_candidate_reg_t, *cgopt_candidate_reg;
struct cgopt_candidate_reg_s {
  dc_cell reg;
  pwrbdd_manager pbm;
  bdd_formula reg_clk_bdd;
  bdd_formula reg_en_bdd;
  bdd_formula reg_ns_bdd;
  bdd_formula clk_bdd;
  bdd_formula en_bdd;
  unsigned int has_fb : 1;
  unsigned int has_non_standard_sync_pins : 1;
  unsigned int has_non_standard_ext_fb_loop : 1;
  unsigned int has_non_standard_int_fb_loop : 1;
  dc_pin reg_clk_pin;
  dc_net reg_clk_net;
};
```

Listing 3.10: Modified data structure, candidate_reg

The introduction of the function `candidate_latch_obj_create`, shown in the code fragment 3.9, as an extension of the already existent function `candidate_reg_obj_create`, which creates a data structure holding all the necessary information required in the further stages of the clock-gating insertion algorithm, was necessary because the data structure shown in the code fragment 3.10 had to be modified incorporating the fields `reg_clk_pin` and `reg_clk_net` required in further stages of the algorithm to allow clock-gating insertion for *Latches.*

Clock-gating insertion

The clock-gating insertion part of the algorithm required few modifications thanks to the previous modifications, still 2 functions had to be modified.

```
            if (dc_cell_is_latch(reg)) {
              clk_pin = cand->reg_clk_pin;
              clk_net = cand->reg_clk_net;
            }
```

Listing 3.11: Relevant code fragment in the function that verifies if and inverter is needed between the clock-gate and the gated cell

The first modification is related to the modification in the `candidate_reg` data structure, this modification, whose lines are shown in the code fragment 3.11, is necessary to obtain the **clk** pin and net of the *Latch* in the function that realizes the connection between the *CG* cell and the gated cell, because it needs to verify if an inverter should be connected between the *CG* and the cell.

```
    if(dc_cell_is_sequential_or_has_sequential_arcs(cell)) {
      if(dc_pin_is_clock(pin) &&
         ( clkgt_dc_cell_is_register(cell)||
           (cgopt_get_var_power_cg_allow_cg_for_latches()
            && dc_cell_is_latch(cell) ) ) )
        hash_put_kptr_vint(gated_pin_hash, pin, 1); /* register or latch*/
      return TRUE; /* sequential */
    }
```

Listing 3.12: Relevant code fragment in the function that verifies the presence of a gated cell (*Latch* or *Flip-Flop*) connected to a clock-gate

The second modification to the clock-gating insertion mechanism, shown in the code fragment 3.12, is necessary to allow the correct identification of the sequential cells connected to the *CG* cell, because prior the prototype, any *Latch* that would be connected to a *CG* cell, wouldn't be considered as a sequential cell, thus isn't used to verify the maximum number of connected cells to the clock-gate.

3.6 Unit tests for the code

As part of the clock-gating for *Latches* prototype development some unit tests had to be created. The unit test are verified automatically every night to ensure that the implemented functionality wouldn't be broken by other developers working on the code.

Four unit test were created to verify different aspect of the clock-gating for *Latches* insertion engine, and ensure that some bugs that were in the code during the first development stages of the code will no longer be reproducible. These unit test are available in appendix B.

Chapter 4

Analysis of Results

The clock-gating for *Latches* prototype feature was successful by providing basic clock-gating functionality in *Latch* based designs, similarly some of the advanced available for *Flip-Flop* based designs worked properly, mainly because such features work over existent clock-gates without double-checking the connected cells, still those features are not considered as part of this project because there are still some basic features that should be improved first and full functionality of the advanced features is not guaranteed.

4.1 Unit test results

The prototype clock-gating for *Latches* engine was able to insert clock-gates in *Latch* based design, which is the main purpose of this work, in fact the software was effective enough for unit test to be developed as discussed in 3.6. Regarding the functionality implemented only the unit tests B.1 and B.2 are interesting as the others only verify that some early bugs are not longer reproduced.

4.1.1 Mapped *Latch* clock-gating

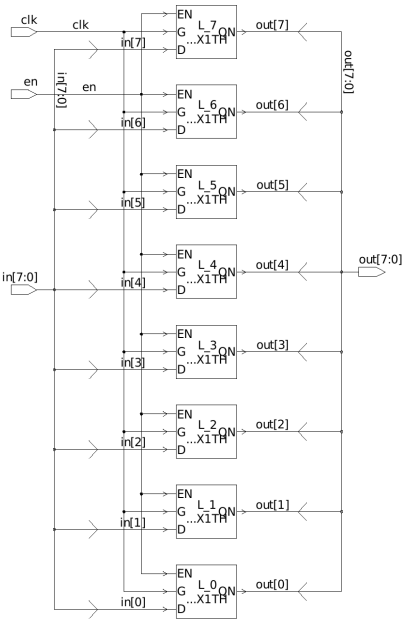


Figure 4.1: Bank of 8 mapped *Latches* with **EN** pin used as staring point in the testcase

Figure 4.1 shows the bank of *Latches* as displayed in the *Design Compiler® GUI* used as starting point in the unit test B.1 for the clock-gating for *Latches* feature.

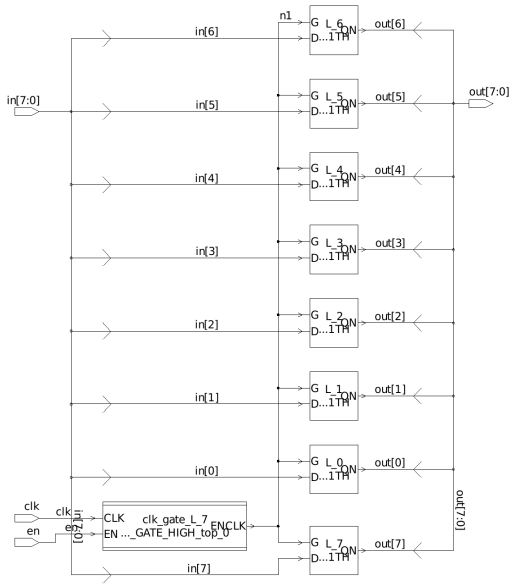


Figure 4.2: Bank of 8 mapped *Latches* gated using *Design Compiler®* and the clock-gating for *Latches* feature

At the end of the `compile` command with the clock-gating for *Latches* feature turned on, the

mapped *Latches* are gated with a newly created clock-gating cell and the gated cells are remapped into simpler version of the original *Latches* without their enable pin, which is the exact same behavior the clock-gating engine would have if the cells were *Flip-Flops.* This test illustrates the successful implementation of the feature in the *Design Compiler*® tool.

4.1.2 *Latch* with feedback loop clock-gating

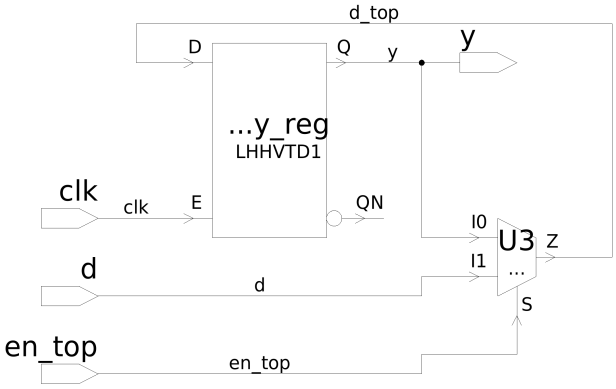


Figure 4.3: Single *Latch* with feedback loop after the first compile in *Design Compiler*® used as staring point for the clock-gating for *Latches* feature

The other type of clock-gating insertion developed for *Latches* requires a feedback loop to be present in the cell to be gated such as the one shown in figure 4.3 where the cell has a clear multiplexer-based feedback loop in the same way as the feedback loop required for *Flip-Flop* based designs, without an enable pin for the cells, to be gated. The diagram in figure 4.3 is extracted from a middle point in the unit test B.2 for comparison.

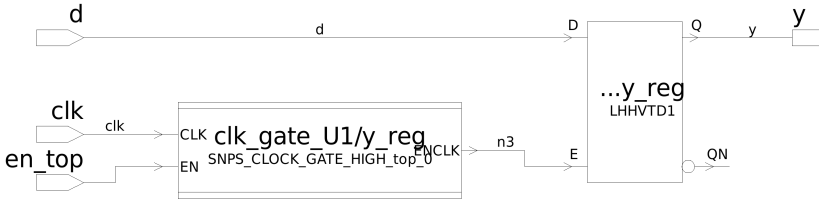


Figure 4.4: Gated *Latch* using *Design Compiler*®

Again the clock-gating for *Latches* feature is able to perform clock-gating in a cell as expected, by connecting the enable pin of the multiplexer to the newly created clock-gating cell and disarming the already existent feedback loop, as shown in figure 4.4 taken directly from the *GUI* of *Design Compiler*® after the unit test B.2 is run.

4.2 Customer testcase results

One of the *Synopsys* customers provided access to one of their designs to be used as a testing platform for the clock-gating or *Latches* feature. This testcase has to be accessed through an external server under a strict non disclosure agreement, thus very little details can be given about this project without violating those agreements.

4.2.1 The testcase

The testcase design has over ten thousand *Latches* and over fifteen thousand *Flip-Flops,* and the design already has several cells mapped and clock-gating cells inserted, in fact, some *Latches* are already gated with the same latch based *CG* cells than the *Flip-Flops* and some *Latches* share their clock-gating cell with *Flip-Flops,* thus the testcase provides support for some of the assumptions made during the feature development and discussed in 3.3.

The script used to run the testcase performs 2 successive `compile` commands, with slightly different configurations in order to obtain the desired results, a full run takes more than 4 hours.

4.2.2 The results

In order to obtain the results several copies of the testcase folder were made in order to store the results from different configurations and to avoid conflicting file access during the runs.

There are 2 flows run to validate the results of the clock-gating for *Latches* feature, the first flow is to stop the script at the end of the first `compile` command referred as the *Short flow*, while the second flow is to run the full script, which is as the *Full flow*. Each flow is run twice, using the same binary of *Design Compiler*® , one of the runs is performed as reference while the other is performed using the variable `power_cg_allow_cg_for_latches` set to true.

In order to analyze the results, the *Design Compiler*® commands and the scripts designed to measure the results were used.

The short flow

Data	**Reference**	***Latches***	**Improvement (%)**
Inserted *CG*	958	1023	6.78
Flip-Flops	32983	32983	0.00
Latches	9799	9799	0.00
Gated *Flip-Flops*	26282	26282	0
Ungated *Flip-Flops*	6701	6701	0
Gated *Latches*	2473	5398	118.27
Ungated *Latches*	7326	4401	-39.92
Gating *Latch* (%)	25.24	55.09	29.84

Tabla 4.1: Comparison between the reference results and the clock-gating for *Latches* results in the short flow

The results of the short-flow run in table 4.1 show that the design already had some *Latches* gated by the designers, still the number of gated *Latches* by the clock-gating for *Latches* feature allowed a 118.28% of gated *Latches* increase compared with the designers original effort, which given the lack of such feature is probably their best effort, thus the prototype shows promising results for the feature.

The full flow

Data	**Reference**	***Latches***	**Improvement (%)**
Inserted *CG*	777	907	16.731017
Flip-Flops	32981	32980	N/A
Latches	9799	9799	0
Gated *Flip-Flops*	26539	26527	-0.05
Ungated *Flip-Flops*	6442	6543	1.57
Gated *Latches*	2392	5365	124.29
Ungated *Latches*	7407	4434	-40.14
Gating *Latch* (%)	24.410654	54.750485	30.34

Tabla 4.2: Comparison between the reference results and the clock-gating for *Latches* results in the full flow

The full-flow results in table 4.2 showed even better results for the number of gated *Latches* compared with the designers original effort, regardless of the reduction in the number of *CG* cells which can be explained by further optimizations performed by the tool in the second run of the full-run and the slight reduction in the number of gated *Flip-Flops* which can be understood as a conflict in the number of gated cells for a given *CG* .

From the *Latches* that couldn't be gated in any flow there were 587 *Latches* that were skipped because their **clk** pin couldn't be determined reliably, which is not a problem in the feature but a problem in the cells used by the customer that couldn't be analyzed correctly by the tool.

Chapter 5

Conclusions

5.1 Conclusions

The goal for this works is to implement a prototype of clock-gating insertion for *Latch* based designs in the *Design Compiler*® tool developed by *Synopsys* , which was performed successfully as discussed in Chapter 4.

Unfortunately the clock-gating for *Latch* based design literature was scarce, since very little could be found through papers and much less was available in book abut clock-gating, which is a consequence of the little development in the area and that a small percentage of commercial designs make heavy use of *Latches,* making the knowledge of clock-gating implementations not widespread. The closest approach to clock-gating for *Latch* based design in the literature was the pulser-gating approach discussed in [19]

Still circuit designers use clock-gating for their *Latch* based circuits, but when they do they make sure by hand that the designed behavior of the circuit is not perturbed by the *CG* cell insertion because they tend to use the same cells for *Latches* and *Flip-Flops* .

Considering the differences between *Flip-Flops* and *Latches,* it's clear that the differences can be classified in area; power and timing differences. The area and power consumption differences are not relevant for clock-gating, while the timing differences are fundamental. The most interesting timing characteristics of *Latches* are dynamic time borrowing and clock uncertainty absorption, and any clock-gating scheme implemented should not break these characteristics.

During the present work, the analysis of clock-gating insertion for *Flip-Flops* provided 2 different strategies for gating *Flip-Flops.* The first strategy is based on *Latch* based clock-gates which provide a functionality invariant transformation on the circuit, while the second strategy based on *Latch* free clock-gates provide an unsafe clock-gating insertion which must be dealt with care in order to honor the designed functionality. Using the previous information and establishing an analogy between the *Latches* and *Flip-Flops* timing, it's possible to provide support for clock-gating

insertion for *Latches* as long as the appropriate clock-gating cells are used and some restrictions are observed.

Even if clock-gating for *Latches* is properly supported, the timing differences between *Latches* and *Flip-Flops* require the utilization of different *CG* cells between *Latches* and *Flip-Flops,* which raise the issue of not having the capability of selecting the *CG* cells for *Latches.* Also since clock-gating for *Latches* is only at experimental stages and not supported by any commercial tools, the process is not supported by formal verification tools[1].

Considering the similarities between *Latch* based designs and *Flip-Flop* based designs, it became clear that the candidate cells selection and some of the clock-gating mechanisms used in *Flip-Flop* based designs could be used in a *Latch* based designs; this means that the existent code base used by *Design Compiler®* during clock-gating insertion can be used in *Latch* based designs with minor modifications in order to produce a prototype. Still some extra modifications are required to consider border cases and ensure complete correctness in a *Latch* based design, but no redesign of the existent algorithms is required.

Finally the prototype developed showed promising results in the unit tests developed and in the testcase provided by *Synopsys* customer the significant increase in the number of gated *Latches* is even more relevant as the circuit designers already had gated the *Latches* in the design to their best effort, so every additional *Latch* gated is an improvement over the best possible result up to date.

5.2 Future Work

Despite the success of the clock-gating for *Latches* prototype, there is still much work to do in order to the feature to be considered as "Production Ready", thus here is presented a list of the required work to be done in this feature:

5.2.1 Fix Remove Clock-gating

The remove clock-gating is partially supported for *Latches,* because even if the tool is able to remove the *CG* cell, the tool is still unable to correctly map the *Latch* to a version with **EN** and **clk** pin, and it's also unable to properly create a feedback loop, or to connect the enable signal and the clock signal with a logic *AND* gate, thus the current state of the feature would introduce an inconsistency between the original design and the implemented one.

Although the remove clock-gating feature was initially implemented, no further efforts were made to fix the remapping of *Latches,* because there was an ongoing development conflicting with the required modifications so they would have to be made after the other development was stable.

[1]Formal verification tools allows the verification that the functionality of the circuit is not affected by the optimizations performed

5.2.2 New configuration of the clock-gating requirements

Currently the clock-gating specification set through the `set_clock_gating_style` has no means of specifying the clock-gating requirements for *Latches* being the inability to configure the type of clock-gating cell the most critical feature to be implemented.

The reason for this requirement is already discussed in 3.3, as the best clock-gating cell is the latch-free clock-gate, and in the current scheme setting a latch-free *CG* cell would implement the latch-free clock-gating cell for *Flip-Flop* based designs whose timing diagram, shown in figure 2.47, would introduce erroneous signal propagation through the *Latch* .

5.2.3 Separate *Latches* and *Flip-Flop*

To separate *Latches* and *Flip-Flops,* means to avoid using the same *CG* cell to drive a *Latch* and a *Flip-Flop,* this feature is a must in order to consider the feature as "Production ready" because the user would need it, and providing such a basic option, which means to be able to allow or forbid the same *CG* cell between *Latches* and *Flip-Flop* at user request.

5.2.4 Support some advanced features

As discussed in 2.10 the clock-gating insertion is a complex feature that has already reached a high degree of sophistication, thus some of the previously discussed advanced features, are expected, not only at user request, but to be used at every possible scenario (such as the enhanced clock-gating features) where a clock-gate is inserted. Therefore a the clock-gating insertion for *Latches* must be able to support at least some of the advanced clock-gating features before it's considered as available for *Synopsys* customers. Obviously the ultimate goal for a clock-gating for *Latches* feature is to support the same features as the *Flip-Flops* counterpart.

5.2.5 *Formality*® support

With the advance of automatic synthesis tools, and the increase of the transformations they perform on the circuit designs by optimizing the logic and features like clock-gating that actively modify the circuit, circuit designers require a tool that allows them to verify that the functionality of the circuit isn't modified by the synthesis process.

Although one option is to simulate the circuit for every possible input combinations to verify the functionality of the circuit, such strategy is not practical, therefore the verification relies in the formal verification of the circuit by analyzing the combinational logic of the original design and the synthesized circuit and demonstrate the logic equivalence using mathematical functions. *Synopsys* provides *Formality*® as a formal verification tool, however the clock-gating transformation requires additional instructions to *Formality*® in order to be accepted as a valid transformation as

it performs a non mathematical equivalent transformation, similarly the clock-gating for *Latches* feature requires additional instructions in order to *Formality*® to validate it.

Currently there is no implementation in the *Formality*® tool to support clock-gating for *Latch* based design, because this feature is still in the prototyping stage. There is little to no information on the required instruction (if any) to validate clock-gating for *Latches* as *Formality*® could still validate the transformation, based on the instructions that are already valid in clock-gating for *Flip-Flop* based designs.

Bibliography

[1] C. Roth, *Fundamentals of Logic Design.* Cengage Learning, 2009. [Online]. Available: http://books.google.cl/books?id=EHN8VrpC21cC

[2] T. Floyd, *Digital fundamentals.* Prentice Hall, 2003. [Online]. Available: http://books.google.cl/books?id=gjAfAQAAIAAJ

[3] R. Dueck, *Digital design with CPLD applications and VHDL.* Thomson/Delmar Learning, 2005. [Online]. Available: http://books.google.cl/books?id=1eO7kLWUmYIC

[4] D. Harris and S. Harris, *Digital design and computer architecture*, ser. Morgan Kaufmann. Morgan Kaufmann Publishers, 2007. [Online]. Available: http://books.google.cl/books?id=5X7JV5-n0FIC

[5] J. Wakerly, *Digital design: principles and practices.* Prentice Hall, 2001. [Online]. Available: http://books.google.cl/books?id=RHv6ewEACAAJ

[6] P. D. Franzon, "Introduction to asic design," Synopsys University Courseware; Lecture 1, 2008.

[7] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Programming*, pp. 231–274, 8. 1987.

[8] H. Lee, S. Paik, and Y. Shin, "Pulse width allocation with clock skew scheduling for optimizing pulsed latch-based sequential circuits," in *Computer-Aided Design, 2008. ICCAD 2008. IEEE/ACM International Conference on*, nov. 2008, pp. 224 –229.

[9] V. Oklobdzija, *Digital System Clocking: High-Performance and Low-Power Aspects*, ser. Wiley-Interscience.

[10] C.-M. Chang, S.-H. Huang, Y.-K. Ho, J.-Z. Lin, H.-P. Wang, and Y.-S. Lu, "Type-matching clock tree for zero skew clock gating," in *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, june 2008, pp. 714 –719.

[11] "Dynamic logic gates." [Online]. Available: http://bwrc.eecs.berkeley.edu/classes/icdesign/ee141_f09/Lectures/Lecture19-Dynamic-2up.pdf

[12] "Ee241 - spring 2011: Advanced digital integrated circuits." [Online]. Available: http://bwrc.eecs.berkeley.edu/classes/icdesign/ee241_s11/Lectures/Lecture21-Domino.pdf

[13] M. Keating, D. Flynn, R. Aitken, A. Gibbons, and K. Shi, *Low Power Methodology Manual For System-on-Chip Design*. Springer, 2008.

[14] Synopsysy, *Synopsys Low-Power Flow user Guide*. Synopsys, 2010.

[15] P. D. Franzon, "Low power design," Synopsys University Courseware; Lecture 9, 2008.

[16] D. Z. Pan, "Low power design and challenges in nanometer multicore era," Aug 2009. [Online]. Available: http://www.ieeevic.org/docs/slides/Victoria_DavidPan.pdf

[17] A. Chandrakasan and R. Brodersen, "Minimizing power consumption in digital cmos circuits," *Proceedings of the IEEE*, vol. 83, no. 4, pp. 498 –523, apr 1995.

[18] L. Li and K. Choi, "Activity-driven optimised bus-specific-clock-gating for ultra-low-power smart space applications," *Communications, IET*, vol. 5, no. 17, pp. 2501 –2508, 25 2011.

[19] S. Kim, I. Han, S. Paik, and Y. Shin, "Pulser gating: A clock gating of pulsed-latch circuits," in *Design Automation Conference (ASP-DAC), 2011 16th Asia and South Pacific*, jan. 2011, pp. 190 –195.

Appendix A

TCL Scripts

```
##########################################################################
#                                                                        #
#                                                                        #
# Title: CG_counters_for_latch.tcl                                       #
# Author: Joaquin Figueroa                                               #
#                                                                        #
# Abstract:                                                              #
#     This script is the wrapper of the other scripts included in the    #
#     folder,with the purpose of having tools to count latches that      #
#     were clock_gated or to know how many were not gated, also          #
#     allows to do the same with flops without using the report,         #
#     however the results are not exactly the same between               #
#     both methods.                                                      #
#                                                                        #
#     A good example of usage for these commands is in  latches          #
#     Regressions located in syn/unit/power/cg/cgc/latch                 #
#                                                                        #
##########################################################################


##############################################################################
#                                                                            #
#                                                                            #
# Some examples on how to use the commands and intended use:                 #
#                                                                            #
# # get the sets                                                             #
#                                                                            #
# set latches [all_registers -level_sensitive]                               #
# set flops [all_registers -edge_triggered]                                  #
#                                                                            #
# # get all clock-gated cells                                                #
# set all_gated_cells [gated_pins_collection \                               #
#                         [output_pins [all_clock_gating_cells]]]            #
#                                                                            #
# # get all clock_gated registers and latches                                #
# set gated_flops Inter_collections $all_gated_cells $flops                  #
# set gated_latches Inter_collections $all_gated_cells $latches              #
#                                                                            #
```

```
#                                                                              #
## get all_ungated cells                                                       #
#                                                                              #
# set ungated_flops Inter_collections ($flops-$gated_flops) $flops             #
# set ungated_latches Inter_collections ($latches-$gated_latches) $latches     #
#                                                                              #
#                                                                              #
#                                                                              #
#                                                                              #
################################################################################

# This command intersect 2 collections
proc Inter_collections {A B} {\
set interAB [remove_from_collection $A [remove_from_collection $A $B]];\
return $interAB;\
}

################################################################
#                                                              #
# Example:                                                     #
#                                                              #
# set A = all_registers -level_sensitive                       #
# set B = all_clock_gating_cells                               #
#                                                              #
# Inter_collections $A $B                                      #
# Gives all the latches inside an CG cell whose are not ICG    #
################################################################


#Joaquin Figueroa: Command created to get all the cell inside a given hierarchy, for
#                  example to get the internal structure of a ICG.
proc Hier_inter {cells} { \
set int_cell {}
  foreach_in_collection a_cell  $cells { \
    append_to_collection int_cell [get_cells [get_attribute $a_cell full_name]/*]; \
    }
return $int_cell;\
}

################################################################################
#                                                                              #
#                                                                              #
# Title: gated_pins_collection.tcl                                             #
# Author: Joaquin Figueroa                                                     #
#                                                                              #
# Abstract:                                                                    #
#     This script find all the cells conected though an input port to          #
#     a specific pin, either directly or at most though buffers or             #
#     inverters thus the main use for this function is to find all             #
#     the registers or latches connected to a clock_gating cell.               #
#                                                                              #
#     The output of this procedure is a collection, which means                #
#     that this result can be used with other commands like                    #
#     all_registers or report_cell                                             #
#                                                                              #
################################################################################
```

```
proc gated_pins_collection {pin} { \
set CG {}
  set pins [filter_collection [all_connected [all_connected [get_attribute $pin full_name]]
      -leaf] "pin_direction == in"]; \
  foreach_in_collection gated_pin $pins { \
    set cell [get_cells -of_objects $gated_pin]; \
    set num_in_pins [sizeof_collection [filter_collection [get_pins -of_objects $cell]
        "pin_direction == in"]]; \
    set num_out_pins [sizeof_collection [filter_collection [get_pins -of_objects $cell]
        "pin_direction == out"]]; \
# if the clock_enable signal goes to a buffer/inverter, skip the cell and go to the next
    cell
   if {$num_in_pins == 1 && $num_out_pins == 1} { \
    set out_pin [filter_collection [get_pins -of_objects $cell] "pin_direction == out"]; \
    foreach_in_collection recur_pin $out_pin { \
      append_to_collection CG [gated_pins_collection $recur_pin]; \
    } \
   } else {
    append_to_collection CG $cell; \
   }
  }
return $CG;\
}
```

Listing A.1: Tcl functions developed to measure the gated *Latches*

Appendix B

Unit Tests

```
##/-----------------------------------------------------------------------------
##
##    Test:   gate_mapped_Elatch
##    Author: Joaquin Figueroa
##    Date:  Oct 13, 2011
##
##    Abstract: This test verifies the insertion of clock gates in a latch
##    based design, with mapped latch. The test is an 8 mapped-latches bank
##
##    Checkpoints:
##      CP0: General errors
##      CP1: verifies no CG insertion in compile
##      CP2: Verifies the insertion of CG in compile -gate_clock
##      CP3: Check all 8 latches are gated
##      CP4: Check there are no latch ungated
##      CP5: Verifies 2 CG inserted with maxfanout 4
##      CP6: Check all 8 latches are gated
##      CP7: Check there are no latch ungated
##      CP8: Verifies 8 CG inserted with maxfanout 1
##      CP9:  Check all 8 latches are gated
##      CP10: Check there are no latch ungated
##      CP11: Verifies 1 CG inserted with min bit = max fanout = 5
##      CP12: Check only 5 latches are gated
##      CP13: Check 3 latches left  ungated
##
##
##      There is no formality run, because this operation is not supported by
##      formality, and (in this scenario) never will
##
##    History:
##       Tue Oct 25 06:27:51 PDT 2011 - Joaquin Figueroa: as I've uploaded the
##       scripts to count gated latches, I'm including those tests.
##
##
##   *** TODO ***
##     As functionality is added to the clock_gating for latch based designs
##     project and the UI (and other tool like report_cg) start to suppot
##     Latches add these verifications to the regression.
##
```

```
##-------------------------------------------------------------------------/


##########################################################################
#
# For Now, we disable pass1, pass2 partial cg, since the insert CG
# And the fanout of a CGcell doesn't count latches.
#
##########################################################################


# CP0:
### Set library info
## source [format "%s%s" [getenv POWER_SCRIPTS] /alib_path.tcl.include]
set search_path [concat $search_path [list \
  [format "%s%s" [getenv REGRESSO_DESIGN] "/power/Latch_cg"] \
  [format "%s%s" [getenv REGRESSO_LIB] "/seqmap"] \
  [format "%s%s" [getenv REGRESSO_LIB] "/retime"] \
  [getenv REGRESSO_LIB] \
  ]]


### set cgopt_do_partial1 FALSE
### set cgopt_do_partial2 FALSE


### source the utilities I uploaded for CG for latches
source [format "%s%s" [getenv POWER_SCRIPTS] "/Latch_counters/CG_counters_for_latch.tcl"]

set target_library [list artisan_addhvt_tts.db lsi_10k.db]
set link_library "* $target_library"

##  set alib_library_search_path
### We read a design

read_file -f verilog latch_bank.v
current_design top
link

### activate clock gating for latch based designs using the hidden variables

set power_cg_allow_cg_for_latches TRUE
set pwr_cg_print_cg_for_latches_messages TRUE

### From now several compile will be made with different clock_gating results
### expected from the clock_gating_style.


##########################################################################
##########################################################################


### First no style, no gating.

compile
```

```
# CP1:
report_clock_gating
# CP1:FIND_REG_EXP "Number of Clock gating elements\s+\|\s+0\s+\|"


### Now make an incremental compile, default style, 1 expected CG
compile -incremental -gate_clock

# CP2:
report_clock_gating
# CP2:FIND_REG_EXP "Number of Clock gating elements\s+\|\s+1\s+\|"


### Now we count how many latches were gated.


### Define all latches in design
set latches [all_registers -level_sensitive]
### Get all clock gates
set cgs [all_clock_gates]
### get all latches inside a cg cell
set latches_in_CGs [Inter_collections $latches [Hier_inter $cgs]]
### Use this to get all latches that are not inside a CG cell
set latches_not_in_CG [remove_from_collection $latches $latches_in_CGs]

### Get all gated cells
set gated_cells [gated_pins_collection [all_clock_gates -output_pins]]
### get which  of the gated cells are latches
set gated_latches [Inter_collections $latches_not_in_CG $gated_cells]
### Get which of the latches were not gated
set ungated_latches [remove_from_collection $latches_not_in_CG $gated_latches]

### Finally we define the checkpoints needed.
### we should have 8 gated latches and 0 ungated
# CP3:
sizeof_collection $gated_latches
# CP3:FIND_STRING "8"

# CP4:
sizeof_collection $ungated_latches
# CP4:FIND_STRING "0"


### Now, free desing, set style, and compile again, 2 CG expeted
free


##############################################################################
##############################################################################

read_file -f verilog latch_bank.v
current_design top
link
```

```
set_clock_gating_style -minimum_bitwidth 1 -max_fanout 4
compile -gate_clock

# CP5:
report_clock_gating
# CP5:FIND_REG_EXP "Number of Clock gating elements\s+\|\s+2\s+\|"


## Define all latches in design
set latches [all_registers -level_sensitive]
### Get all clock gates
set cgs [all_clock_gates]
### get all latches inside a cg cell
set latches_in_CGs [Inter_collections $latches [Hier_inter $cgs]]
### Use this to get all latches that are not inside a CG cell
set latches_not_in_CG [remove_from_collection $latches $latches_in_CGs]

### Get all gated cells
set gated_cells [gated_pins_collection [all_clock_gates -output_pins]]
### get which  of the gated cells are latches
set gated_latches [Inter_collections $latches_not_in_CG $gated_cells]
### Get which of the latches were not gated
set ungated_latches [remove_from_collection $latches_not_in_CG $gated_latches]

### Finally we define the checkpoints needed.
### we expect 8 gated latches and 0 ungated

# CP6:
sizeof_collection $gated_latches
# CP6:FIND_STRING "8"

# CP7:
sizeof_collection $ungated_latches
# CP7:FIND_STRING "0"



### Now, free desing, set style, and compile again, 8 CG expeted
free

##############################################################################
##############################################################################

read_file -f verilog latch_bank.v
current_design top
link

set_clock_gating_style -minimum_bitwidth 1 -max_fanout 1
compile -gate_clock

# CP8:
report_clock_gating
# CP8:FIND_REG_EXP "Number of Clock gating elements\s+\|\s+8\s+\|"
```

```
## Define all latches in design
set latches [all_registers -level_sensitive]
### Get all clock gates
set cgs [all_clock_gates]
### get all latches inside a cg cell
set latches_in_CGs [Inter_collections $latches [Hier_inter $cgs]]
### Use this to get all latches that are not inside a CG cell
set latches_not_in_CG [remove_from_collection $latches $latches_in_CGs]

### Get all gated cells
set gated_cells [gated_pins_collection [all_clock_gates -output_pins]]
### get which  of the gated cells are latches
set gated_latches [Inter_collections $latches_not_in_CG $gated_cells]
### Get which of the latches were not gated
set ungated_latches [remove_from_collection $latches_not_in_CG $gated_latches]

### Finally we define the checkpoints needed.
### we expect 8 gated latches and 0 ungated

# CP9:
sizeof_collection $gated_latches
# CP9:FIND_STRING "8"

# CP10:
sizeof_collection $ungated_latches
# CP10:FIND_STRING "0"



### Now, free desing, set style, and compile again, 1 CG expeted
free

#################################################################
#################################################################


read_file -f verilog latch_bank.v
current_design top
link

set_clock_gating_style -minimum_bitwidth 5 -max_fanout 5
compile -gate_clock

# CP11:
report_clock_gating
# CP11:FIND_REG_EXP "Number of Clock gating elements\s+\|\s+1\s+\|"



## Define all latches in design
set latches [all_registers -level_sensitive]
### Get all clock gates
set cgs [all_clock_gates]
### get all latches inside a cg cell
```

```
set latches_in_CGs [Inter_collections $latches [Hier_inter $cgs]]
### Use this to get all latches that are not inside a CG cell
set latches_not_in_CG [remove_from_collection $latches $latches_in_CGs]

### Get all gated cells
set gated_cells [gated_pins_collection [all_clock_gates -output_pins]]
### get which  of the gated cells are latches
set gated_latches [Inter_collections $latches_not_in_CG $gated_cells]
### Get which of the latches were not gated
set ungated_latches [remove_from_collection $latches_not_in_CG $gated_latches]

### Finally we define the checkpoints needed.
### we expect 5 gated latches and 3 ungated

# CP12:
sizeof_collection $gated_latches
# CP12:FIND_STRING "5"

# CP13:
sizeof_collection $ungated_latches
# CP13:FIND_STRING "3"




##############################################################
##############################################################


# CP0:DONT_FIND_REG_EXP "(?<!MEM )Error:"
```

Listing B.1: Unit test used to verify proper clock-gating insertion in *Latches* with enable pin

```
##/-----------------------------------------------------------------------------
##
##    Test:   cg_latch_with_FB_loop
##    Author: Joaquin Figueroa
##    Date:   Oct 17, 2011
##
##    Abstract: This testcase verifies the insertion of a clock gate in a
##    latch based design, with mapped latch and a small feedback loop though
##    a mux with enable.
##
##    Checkpoints:
##      CP0: General errors
##      CP1: Verifies no insertion is made when the parameter is FALSE
##      CP2: Verifies the insertion of 1 CG for the design
##      CP3: Verifies the only latch in the design in gated
##      CP4: Verifies that there is no "misterious latch" which is gated
##
##      There is no formality run, because this operation is not supported by
##      formality, and (in this scenario) never will
##
##      The design is hosted in $REGRESSO_DESIGN/power/Latch_cg
##
##    History:
##       Tue Oct 25 06:27:51 PDT 2011 - Joaquin Figueroa: as I've uploaded the
```

```
##       scripts to count gated latches, I'm including those tests.
##
##
##   *** TODO ***
##     As functionality is added to the clock_gating for latch based designs
##     project and the UI (and other tool like report_cg) start to suppot
##     Latches add these verifications to the regression.
##
##------------------------------------------------------------------------/

# CP0:
### Set library info
### source [format "%s%s" [getenv POWER_SCRIPTS] /alib_path.tcl.include]
set search_path [concat $search_path \
  [format "%s%s" [getenv REGRESSO_DESIGN] "/power/Latch_cg"] \
  [format "%s%s"  [getenv REGRESSO_LIB] "/power"] \
  [getenv REGRESSO_LIB] \
  ]

### source [format "%s%s" [getenv POWER_SCRIPTS] /alib_path.tcl.include]

### source the utilities I uploaded for CG for latches
source [format "%s%s" [getenv POWER_SCRIPTS] "/Latch_counters/CG_counters_for_latch.tcl"]


set target_library "tcbn90ghvtbc.db"
set link_library "* tcbn90ghvtbc.db"
### set_power_alib_library_search_path

read_verilog latch_with_feedback_loop.v
ungroup -all
link


###  Configure parameters of the regression
set_clock_gating_style -minimum_bitwidth 1

###   Now we make 2 compiles, one without clock-gating, and one without CG

compile -gate_clock

# CP1:
report_clock_gating
# CP1:FIND_REG_EXP "Number of Clock gating elements\s+\|\s+0\s+\|"


### activate clock gating for latch based designs using the hidden variables

set power_cg_allow_cg_for_latches TRUE
set pwr_cg_print_cg_for_latches_messages TRUE


### Now make an incremental compile, defined style, 1 expected CG
compile -incremental -gate_clock
```

```
# CP2:
report_clock_gating
# CP2:FIND_REG_EXP "Number of Clock gating elements\s+\|\s+1\s+\|"

### Now, I define some useful sets.

### Define all latches in design
set latches [all_registers -level_sensitive]
### Get all clock gates
set cgs [all_clock_gates]
### get all latches inside a cg cell
set latches_in_CGs [Inter_collections $latches [Hier_inter $cgs]]
### Use this to get all latches that are not inside a CG cell
set latches_not_in_CG [remove_from_collection $latches $latches_in_CGs]

### Get all gated cells
set gated_cells [gated_pins_collection [all_clock_gates -output_pins]]
### get which  of the gated cells are latches
set gated_latches [Inter_collections $latches_not_in_CG $gated_cells]
### Get which of the latches were not gated
set ungated_latches [remove_from_collection $latches_not_in_CG $gated_latches]

### Finally we define the checkpoints needed.

# CP3:
sizeof_collection $gated_latches
# CP3:FIND_STRING "1"

# CP4:
sizeof_collection $ungated_latches
# CP4:FIND_STRING "0"


### This ends the testcase


# CP0:DONT_FIND_REG_EXP "(?<!MEM )Error:"
```

Listing B.2: Unit test used to verify proper clock-gating insertion in *Latches* with feedback loop

```
## /-----------------------------------------------------------------------
##
##    Test:   no_cg_for_latch_inside_cg
##    Author: Joaquin Figueroa
##    Date: Wed Oct 26, 2011
##
##    Abstract: This test check that there is no insertion of cg for
##    latches inside a CG-hierarchy, to do this, a manually inserted cg is
##    used, and the parameter power_cg_all_registers is set to true.
##
##    When the CG is left unidentified, the latch inside the hierarchy is
##    gated, otherwise, is not.
##
##    Also this test allows slight global verifications, but there will
##    be another regression for global.
##
```

```
##
##    The design is located in $REGRESSO_DESIGN/power/Latch_cg
##
##    Checkpoints:
##      CP0: General errors
##      CP1: verifies no CG insertion in compile
##      CP2: Verifies the insertion of CG in compile -gate_clock
##      CP3: Check all 8 latches are gated
##      CP4: Check there are no latch ungated
##      CP5: Verifies 2 CG inserted with maxfanout 4
##      CP6: Check all 8 latches are gated
##      CP7: Check there are no latch ungated
##      CP8: Verifies 8 CG inserted with maxfanout 1
##      CP9:  Check all 8 latches are gated
##      CP10: Check there are no latch ungated
##      CP11: Verifies 1 CG inserted with min bit = max fanout = 5
##      CP12: Check only 5 latches are gated
##      CP13: Check 3 latches left  ungated
##
##
##      There is no formality run, because this operation is not supported by
##      formality, and (in this scenario) never will
##
##    History:
##       Tue Oct 25 06:27:51 PDT 2011 - Joaquin Figueroa: as I've uploaded the
##       scripts to count gated latches, I'm including those tests.
##
##
##   *** TODO ***
##     As functionality is added to the clock_gating for latch based designs
##     project and the UI (and other tool like report_cg) start to suppot
##     Latches add these verifications to the regression.
##
## ------------------------------------------------------------------------/
##

### Set the search path
# CP0:

set search_path [concat $search_path \
  [format "%s%s"  [getenv REGRESSO_DESIGN] "/power/Latch_cg"] \
  [format "%s%s"  [getenv REGRESSO_LIB] "/power"] \
  ]

set target_library "sc_adv12_rvt_cln45gs_tt_typical_max_0p90v_25c.db"
set link_library "* $target_library"


### source the utilities I uploaded for CG for latches
source [format "%s%s" [getenv POWER_SCRIPTS] "/Latch_counters/CG_counters_for_latch.tcl"]


set power_cg_allow_cg_for_latches TRUE
set pwr_cg_print_cg_for_latches_messages TRUE

set_clock_gating_style -minimum_bitwidth 1
```

```
73
   ##########################################################################
   ##########################################################################

   ### The first test, will verify no insertion of cg
78
   read_verilog no_cg_for_latch_inside_cg_remade.v

   compile -gate_clock

83 # CP2:
   report_clock_gating
   # CP2:FIND_REG_EXP "Number of Clock gating elements\s+\|\s+0\s+\|"

   ##########################################################################
88 ##########################################################################

   ### Second test, we focre cg_insertion on all sequential (latch,flop,cg)
   free

93 read_verilog no_cg_for_latch_inside_cg_remade.v

   ### Set cg_all_registers to true -> force clock gating on all (latches or
   ### flop) cell regardles of their enable condition
   set power_cg_all_registers TRUE
98
   compile

   compile -gate_clock


103
   ### Now we expect 4 cg inserted, one for each hierarchy
   # CP3:
   report_clock_gating
   # CP3:FIND_REG_EXP "Number of Clock gating elements\s+\|\s+4\s+\|"
108


   ### Now we check the manually inserted CG is not identified
   # CP4:
   report_cell [all_clock_gates]
113 # CP4:DONT_FIND_REG_EXP "cg\s+clk_gate\s+\d+\.\d+\s+cg"


   ##########################################################################
   ##########################################################################
118
   ### third test, we focre cg_insertion on all sequential (latch,flop) After
   ### the identificatin of the clock-gate
   free

123 read_verilog no_cg_for_latch_inside_cg_remade.v

   compile

   ### Identify CG
128
```

```
create_clock -period 10 [get_ports clk]

identify_clock_gating -gating_element cg

compile -gate_clock


### Now we expect 3 cg inserted, one for each latch or flop, and
### the identified CG.

# CP5:
report_clock_gating
# CP5:FIND_REG_EXP "Number of Clock gating elements\s+\|\s+3\s+\|"


### Now we check the manually inserted CG is correctly included
# CP6:
report_cell [all_clock_gates]
# CP6:FIND_REG_EXP "cg\s+clk_gate\s+\d+\.\d+\s+cg"




################################################################################
################################################################################

### Fourth test, no identification of the CG, global On, so 2 expected
### CG for all hierarchies, plus the CG inserted for the flop.

free

set compile_clock_gating_through_hierarchy TRUE

read_verilog no_cg_for_latch_inside_cg_remade.v

compile

### Identify CG

create_clock -period 10 [get_ports clk]

compile -gate_clock -boundary_optimization

### Now we expect only 2 CG for all hierachies

# CP7:
report_clock_gating
# CP7:FIND_REG_EXP "Number of Clock gating elements\s+\|\s+2\s+\|"


### Now we check the manually inserted CG is not included
# CP8:
report_cell [all_clock_gates]
# CP8:DONT_FIND_REG_EXP "cg\s+clk_gate\s+\d+\.\d+\s+cg"
```

```
### Now we check all cells are gated.
# CP9:
sizeof_collection [gated_pins_collection [all_clock_gates -output_pins]]
# CP9:FIND_STRING "4"


################################################################################
################################################################################

### Fifth test, Global on  and identified CG.
free

read_verilog no_cg_for_latch_inside_cg_remade.v

compile

### Identify CG

create_clock -period 10 [get_ports clk]

identify_clock_gating -gating_element cg


compile -gate_clock -boundary_optimization


### Now we expect only 2 CG for all hierachies

# CP10:
report_clock_gating
# CP10:FIND_REG_EXP "Number of Clock gating elements\s+\|\s+2\s+\|"


### Now we check the manually inserted CG is included
# CP11:
sizeof_collection [gated_pins_collection [get_pins -of_objects [get_cell clk_gate_q_reg]
    -filter "pin_direction == out"]]
# CP11:FIND_STRING "2"

### Now we check all cells are gated.
# CP12:
sizeof_collection [gated_pins_collection [get_pins -of_objects [get_cell f2/clk_gate_q_reg]
    -filter "pin_direction == out"]]
# CP12:FIND_STRING "1"

### END OF TESTCASE

# CP0:DONT_FIND_REG_EXP "(?<!MEM )Error:"
```

Listing B.3: Unit test used to verify that no *Latches* that would be inside of a *CG* cell are gated

```
##/-----------------------------------------------------------------------
##
##     Test:    fix_non_latch_candidate.tcl
##     Author: Joaquin Figueroa
##     Date:    Thu Oct 20, 2011
##
```

```
##    Abstract: Test fix a fatal when trying to insert a CG to a cadidate
##    cell which is not a FF, nor latch (in cg for latch based designs),
##    so the filter was improved and now only latches are allowed to become
##    candidate cell as it should have been from the begining.
##
##    This regression is very simmilar to
##    prevent_active_low_inverter_removal_for_flat_icg.tcl, only that in this
##    case we activate the insertion of clock gates in latch based design.
##    Also it uses the same test design
##
##    Checkpoints:
##      CP0: General errors
##      CP1: Checks successfull compile -gate_clock
##
##
##
##   *** TODO ***
##     As functionality is added to the clock_gating for latch based designs
##     project and the UI (and other tool like report_cg) start to suppot
##     Latches add these verifications to the regression.
##
##-------------------------------------------------------------------------/

# CP0:
set search_path [concat $search_path \
  [format "%s%s"  [getenv REGRESSO_DESIGN] "/power"] \
  [format "%s%s"  [getenv REGRESSO_LIB] "/power"] \
  ]

set target_library "sc_adv12_rvt_cln45gs_tt_typical_max_0p90v_25c.db"
set link_library "* $target_library"


set power_cg_allow_cg_for_latches TRUE
set pwr_cg_print_cg_for_latches_messages TRUE

read_verilog flat_active_low_TE_ICG.v
current_design top

# CP1:
compile -gate_clock
# CP1:COMMAND_SUCCEEDED


# CP0:DONT_FIND_REG_EXP "(?<!MEM )Error:"
```

Listing B.4: Unit test used to verify that a bug in which a cell that isn't a *Latch* nor a *Flip-Flop* would be gated with the feature is no longer reproducible