Extended Bubble Razor Methodology and its Application to Dynamic Voltage Frequency Scaling Systems

Martin Taugland Kollerud

Master of Science in Electronics
Submission date: June 2013
Supervisor: Snorre Aunet, IET
Co-supervisor: Johnny Pihl, Nordic Semiconductor

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Abstract

by Martin Taugland Kollerud

Increasing voltage and frequency margins in traditional worst-case designs will be more dominating as the process technology is scaled, where power is wasted in exchange for production yield. We have investigated a state-of-the-art DVFS method to eliminate all margins and still guarantee error-free operation, named Bubble Razor.

In the first part of the project did we investigate the methodology of automated conversion from a flip-flop design to a two-phased latch circuit and finally a complete Bubble Razor circuit.

The second part was investigating how Bubble Razor behaves in circuits with synchronous clock domain-crossings, and revealing a clock domain-crossing problem. Two new types of clock-gates are proposed, extending Bubble Razor and enabling it to operate in designs with clock-gates and multiple synchronous clock domains.

A conventional flip-flop design was converted to a two-phase latch design and got a Bubble Razor-circuit inserted. Bubble Razor enabled the design to operate at 80% of the flip-flop version's voltage, without any errors.

Sammendrag

av Martin Taugland Kollerud

Økende spenning- og frekvensmarginer i tradisjonelle worst-case design vil være mer dominerende ettersom prosess-teknologien blir skalert, hvor effekt er brukt i bytte for produksjonsgevinst. Vi har undersøkt en state-of-the-art DVFS metode for å eliminere alle marginer og samtidig garantere feilfri drift, kalt Bubble Razor.

I første del av prosjektet undersøkte vi metodikk for automatisert konvertering fra et flip-flop design til et to-fase latch-design for så til et komplett Bubble Razor-krets.

Den andre delen var å undersøke hvordan Bubble Razor oppfører seg i kretser med synkrone klokkedomene-kryssinger, og avslører et klokke domene-kryssnings problem. To nye typer klokkeporter er foreslått, dette utvider Bubble Razor slik at det kan operere i design med klokke-porter og flere synkrone klokkedomener.

Et konvensjonell flip-flop design ble omdannet til en to-fase latch design og fikk innsatt Bubble Razor. Bubble Razor lar kretsen operere p 80 % av flip-flop versjon sin spenning, uten noen feil.

Acknowledgements

I would like to thank my supervisors Johnny Pihl (Nordic Semiconductor) and Snorre Aunet (NTNU) for the guidance, discussions and for introducing me to a new and interesting field of study. I would also like to thank Nordic Semiconductor for allowing me to work at their office and giving me access to much needed tools and knowledge.

I would thank all my classmates, for five interesting years. I leave NTNU with mixed feelings, knowing that five good years is over. On the other hand, it will be nice to finally start working.

Contents

List of Figures

Abbreviations

DVFS	**D**ynamic **V**oltage **F**requency **S**caling
FF	**F**lip-**F**lop
ICG	**I**ntegrated **C**lock **G**ate
OCV	**O**n **C**hip **V**ariation
PoFF	**P**oint **o**f **F**irst **F**ailure
PVT	**P**rocess **V**oltage **T**emperature

Chapter 1

Introduction

To fulfil an everlasting demand for longer battery life, faster circuits and more functionality per area, parameters like voltage, frequency and process technology need to be scaled to even more extreme limits. However, production yield will decrease if not margins are added to guarantee error-free operation for every single PVT-corner. If a design is made for a given frequency and process, some margins need to be added when specifying the operation voltage. These margins cost power and will not contribute to any performance.

To overcome the increase of margins, the design-for-worst-case mentality must be reconsidered. This report is a study of a state-of-the-art method for making each single chip perform at its best at any condition, called Bubble Razor. It enables the circuit itself to give feedback about its status on the fly, giving the opportunity to scale voltage or frequency to the brink of failure. It will even let setup errors occur, due to slow propagation delay, correcting the errors with an error correcting bubble algorithm and tell the voltage/frequency controllers to speed up.

The project is mainly about the methodology of applying Bubble Razor to any sequential flip-flop design. If it will fit the normal design flow and if it performs as good as we hope. We do also look into how Bubble Razor will interact in a design with more than one clock domain. An extension to the bubble component, called Bubble ICG, is proposed, which will allow multiple clock domains and clock gating in a Bubble Razor design.

Regulators and the power-chain is not a part of this project. This is mainly about the error protection at register-to-register level and its methodology.

1.1 Layout of the Report

Chapter 2 contains a brief explanation of important therms and principles important for the understanding of Bubble Razor. It also includes motivation for DVFS. Further, two-phased latch design and Bubble Razor architecture are explained.

The first part of chapter 3 presents how we converted a flip-flop design to a two-phase latch design, implemented Bubble Razor and how it were verified. The second part is where the Clock Gate Problem explained and a proposed solution is presented.

The analogue simulation results are presented and explained in chapter 4. A discussion and further explanation of the power results are located in chapter 5.

Chapter 2

Theory and Background

Section 2.1 and 2.2 are based on similar sections from our previous work [Kollerud, 2012].

2.1 Power Dissipation in CMOS designs

Digital power dissipation is due to three main sources shown in equation 2.1 [Chandrakasan et al., 1992].

$$P_{total} = p_t(C_L \times V_{dd}^2 \times f_{clk}) + I_{sc} \times V_{dd} + I_{leakage} \times V_{dd} \tag{2.1}$$

Voltage is a part of all the terms and therefore is a good motivator for scale the voltage. The first term is the dynamic power and is a product of the switching factor, p_t, load capacitance, C_L, supply voltage squared, V_{dd}, and the clock frequency, f_{clk}. This term is very power consuming and as a result is clock-gating being more and more used to reduce the switching factor. However, voltage is squared and is a big contributor to this term.

The second term is the power due to short path current that arises when both NMOS and PMOS transistors are active. In addition, this is reduced by decreasing voltage.

The third term is the leakage power, this term is highly dependent on the manufacturing technology and is expected to be more dominant, or maybe the most dominant term as the technology is scaled. Leakage is also a dominating part in sub-threshold circuits [Blaauw et al., 2005].

2.2 Propagation Delay in Digital CMOS Circuits

Propagation delay is generally the time it takes for a signal to travel from launch point to the destination. In digital circuits, the delay of interest is often the delay through the combinatorial logic between two registers, and the delay in the clock network. These delays are composed of interconnect delay, delay through wires, and logic delay, delay through gates. Logic delay scale a lot compared to interconnect delay when voltage is reduced [Elgebaly and Sachdev, 2007].

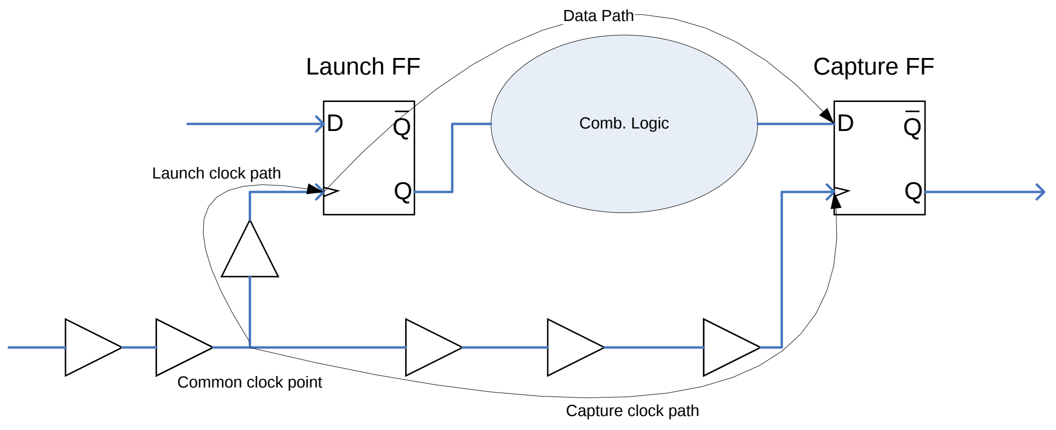


FIGURE 2.1: Illustration of the different paths between two registers.

Figure 2.1 shows the paths of interest between two registers. The data path is defined as the delay between the two registers plus the clock-to-Q time of the launch FF. The launch and capture clock paths are the delay from the common clock point to the clock pin on each FF. The clock paths are often composed of clock buffers, net delay and, if used, clock-gates.

$$T_{arrival} = T_{launch\_clock\_path} + T_{data\_path} \tag{2.2}$$

$$T_{reqired} = T_{capture\_clock\_path} + T_{capture\_clk\_period} - T_{setup\_time} \tag{2.3}$$

$$SetupSlack = T_{reqired} - T_{arrival} \tag{2.4}$$

Equation 2.4 shows some very important values when verifying the timing of a circuit. $T_{arrival}$ is the time a signal use from the common clock point, through the launch clock path and through the data path. $T_{reqired}$ is the deadline for when the instruction need to be stable at the capture registers input pin. It is the delay through the capture clock path plus the clock period and the capture registers flip flop. By combining these two values, $SetupSlack$ is derived. $SetupSlack$ tells how far away a path is from failing it's setup time constraint. The capture register will launch the wrong value, or maybe become metastable, and set the circuit in a wrong state. In the end, the setup slack is the limiting factor for the speed a circuit is able to handle. A circuit will stop working when the frequency is increased or voltage decreased to the point where the slack turns negative. This delay mentality is important when designing a DVFS system.

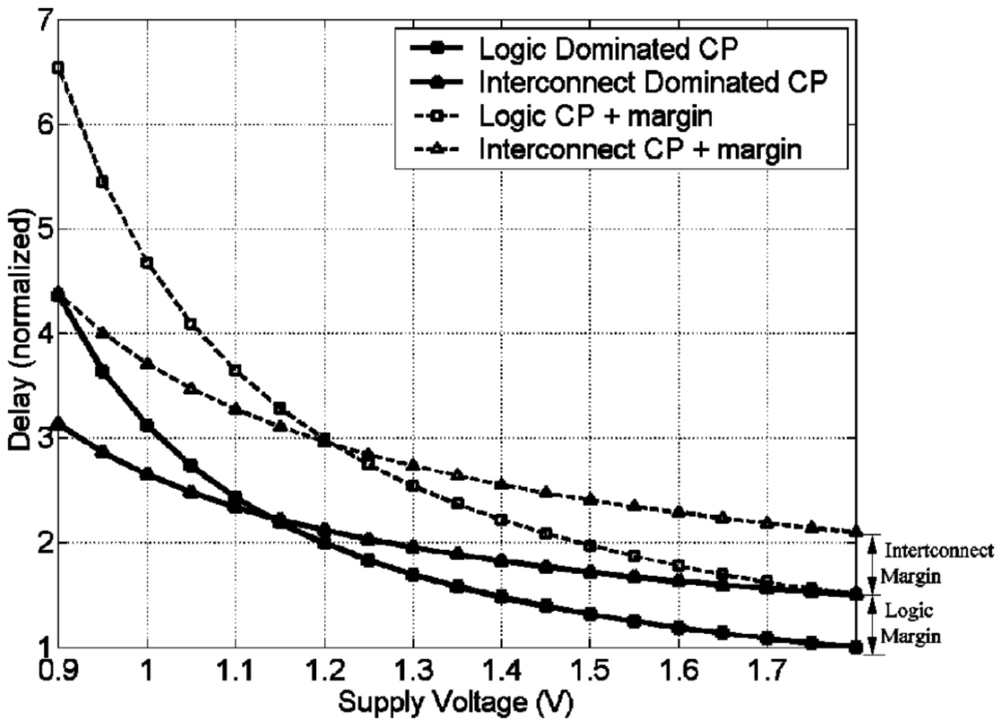


FIGURE 2.2: Illustration of one interconnect dominated path vs one logic dominated path in 180nm. [Elgebaly and Sachdev, 2007]

Logic delay scale a lot compared to interconnect delay [Elgebaly and Sachdev, 2007]. Over the typical voltage range in voltage scaling system is the scaling of the interconnect delay negligible and may be regarded almost constant. Figure 2.2 illustrate two paths being voltage scaled, one logic dominated and one interconnect dominated.

2.3 Dynamic Voltage Frequency Scaling and Error Resilience

Dynamic Voltage Frequency Scaling and design for error immunity are strongly connected. DVFS is all about tuning voltage and/or frequency down/up to the bare minimum/maximum. Some sort of feedback is needed to know when the circuit is operating at its limit. This is where error detection and recovery comes in handy. Traditionally were DVFS done by monitoring copies of a circuits most critical paths [Uht, 2005] [Park and Abraham, 2011]. These techniques are more error avoidance in the sense of measuring the circuits speed, for then to scale down accordingly. However, this is an indirectly method, where on chip variation need to be taken into account leaving some margins left making them pessimistic and not that efficient. More modern ways of monitoring, with DVFS in mind are in-situ monitoring, which detects setup errors and correct them. This will enable almost all margins to be cut away and, as explained later, even scaling beyond the Point of First Failure.

Figure 2.3 illustrates the voltage margins in different samples. Every circuit will have different delay properties, which will determine what frequency and voltage they need to meet all timing requirements. However, margins are added to the actual required values to get a good yield and guarantee error free operation. These margins increase as the technology is scaled, making power be wasted just to insure that most circuits will work in all process corners and under all temperatures, even if 90% of the chips are fast enough far within these margins.

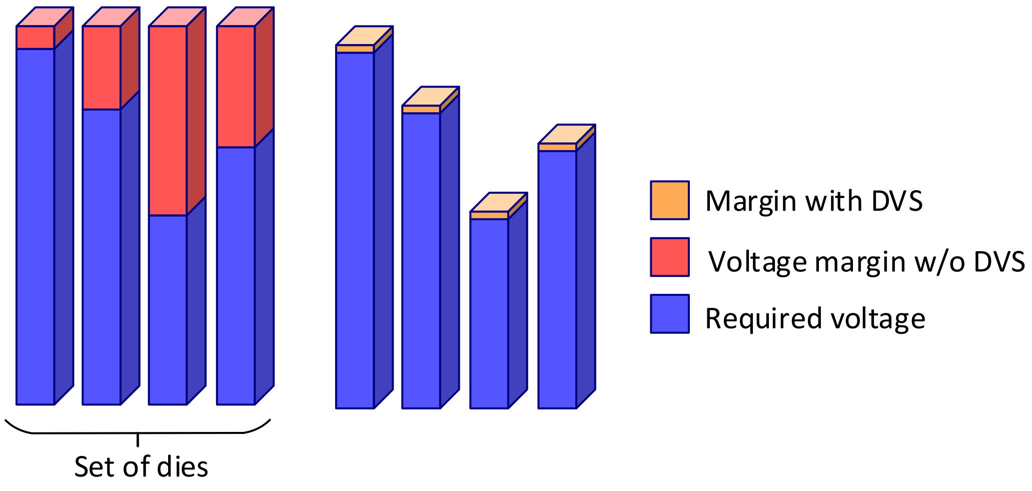


FIGURE 2.3: Illustration of operating voltage in four different dies. Design for worst-case at the left, voltage scaling to the left.

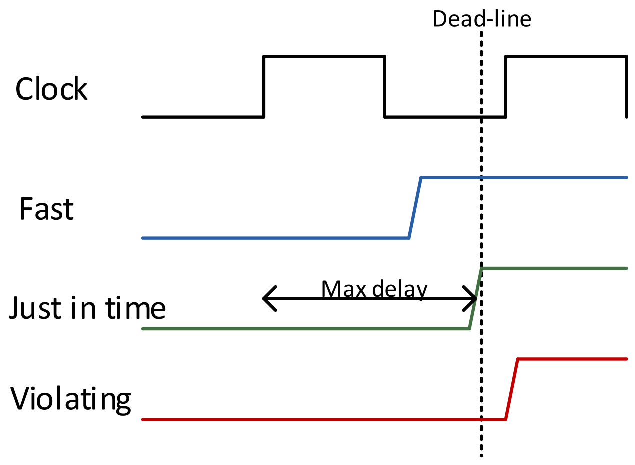


FIGURE 2.4: Just In Time principle.

The point is to get each individual chip to perform as good as it is capable of, and not let every chip perform as the worst-case corner. Figure 2.4 illustrates the "just-in-time" principle, which is the goal independent of what is being scaled. First when the most critical path barely reach its setup requirement, will the circuit operate at optimum voltage or frequency. The most critical paths are the bottlenecks, which mean these paths are the place to monitor. If the regulator control gets feedback about these paths' slack, it will be able scale the voltage to the "just-in-time"-point. Of course, the most critical path in a chip will wary with

OCV, different interconnect/logic-delay ratios and path activity, which means that multiple of paths need monitoring. The traditional critical-path copies needs to take all possible critical paths at every single PVT-variation into account, and guarantee that the copy is always the slowest, which leads to more safety margins.

It does not matter if it is voltage, frequency or even body-bias that is scaled. All of these variables will eventually make the slack negative if scaled too far. The system of detecting when to stop is the same, with some minor differences. Scaling voltage is more challenging than scaling the frequency due to the non-linear properties, which means the task of picking which paths to monitor is more complex. A good voltage scaling scheme will be able to also work for frequency.

This report is mainly about scaling the voltage to reduce power. However, frequency may also be scaled without any more modifications. The idea is to let voltage be scaled to a bare minimum at any time, but let the user scale the frequency accordingly to what throughput he or she need for the application. The voltage will automatically drop if the frequency is decrease, and boost if the frequency is increased for more throughput.

There are some different schemes when it comes to what kind of feedback the regulators get. The traditional methods often use an up/down-feedback, meaning speed up or down. The more modern in-situ methods only send a warning or error, meaning that the regulator need to stop scaling down the speed. Regulator will always decrease the speed with a slow rate until the circuit tells it to stop, then possible scale it a small amount back. The rest of this report is about this feedback, or monitors, and its error recovery capabilities.

2.4 Latch Based Design: Latch is Back?

This section is a short introduction to latches and latch design. For many people, latches are something not often used and are associated with poor tool support and bad verification methods. However, latch designs, if designed right, are faster than the normal flip-flop design [Chinnery et al., 2004]. This is due to the ability for time-borrowing, sometimes called time-stealing, which will be a vital part in the Bubble Razor design.

2.4.1 Registers

Latches and flip-flops are both registers and are used for storing a sate, either 0 or 1. There exist multiple different latches and flip-flops, however, the D-latch, D flop-flop and master-slave flip-flop will be the ones in focus and most important in this study.

In this report, a latch is defined as a level sensitive register and a flip-flop as an edge-triggered register.

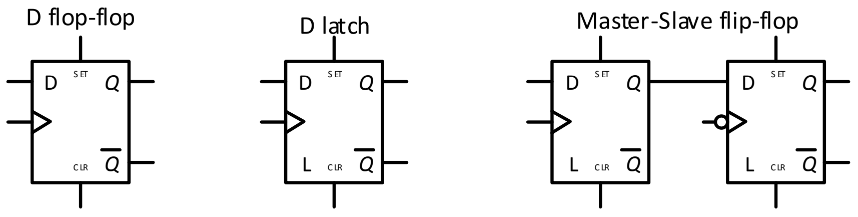


FIGURE 2.5: Symbols for D flip-flop, D latch and master-slave flip-flop.

Latches is transparent as long as the *enable* or *clock* signal is active, and its output will follow its input. The value is hold in the latch's opaque (closed) phase.

The principle of a master-slave flip-flop is important for a latch system to work correct, and behave just like its flip-flop counterpart. This basic element is what enables a flip-flop design to be converted to a two-phased latch design. Figure 2.5 shows a master-slave FF to the right. It's basically two latches, with opposite polarity, connected together. As a black box, this will behave just like a normal edge-triggered flip-flop, even though it is two latches. The first latch, the master, will open and let its input value through to its output in one of the clock phases, while the second latch, the slave latch, will open on the next clock phase.

Latch polarity is a way to distinguish which latches are active at which clock phase. Positive latches are transparent at high clock phase, while negative latches at low clock phase when using a root clock as reference. Active-low and active-high latches refer to the latches themselves when using their clock pins, or enable pins, as reference. A active-high latch is transparent when its clock pin is pull high and vice versa. Sometime are master and slave used instead of positive and negative latches. As seen later will all masters have the same polarity and all slaves the opposite polarity.

2.5 Two-phase Latch Design Principle

A two-phase latch design utilizes the principles of a master-slave flip-flop, that two latches with opposite polarity in series behaves like an edge-triggered flip-flop. By combining more master-slave latches to make up a sequential circuit, like a normal D-latch design, will it behave as a normal edge triggered design by observing its input and output ports. So, if two latches are connected directly together and behave edge triggered, why not balance the data-paths and take some of the logic between each master-slave pair, and put it inside the pairs themselves?

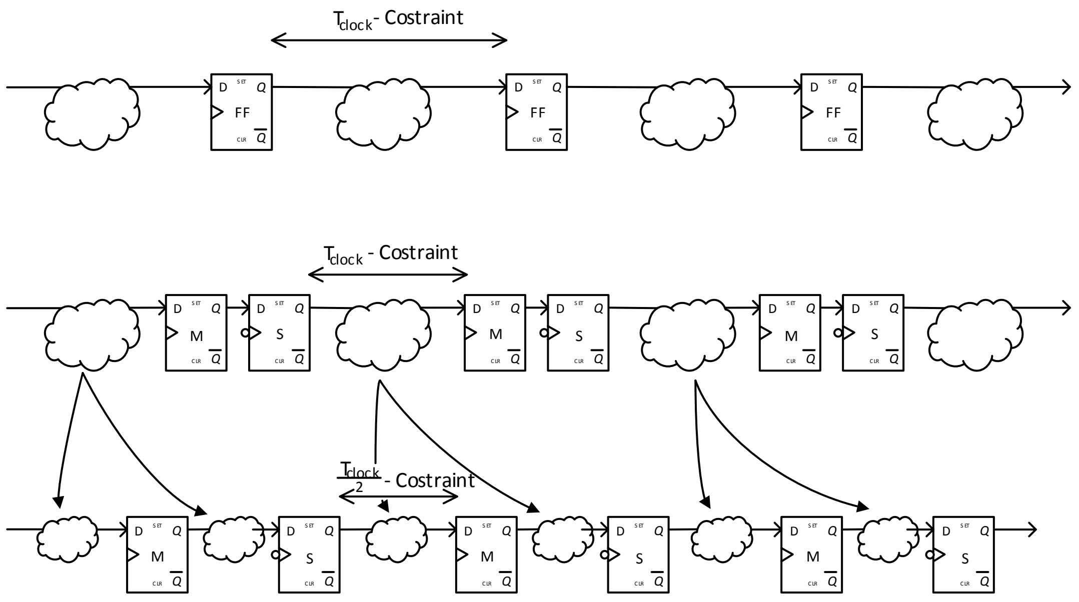


FIGURE 2.6: Illustration of the two-phase latch design principle.

Figure 2.6 illustrates the steps of how to convert a flip-flop design to a two-phase latch design. The most challenging step is the balancing of the paths, the last step in the figure. Modern tool do support retiming of latch circuits, making this step easier [Syn, 2011]. Prior to this tool support was balancing done by tricking the tools to believe they retimed a flip-flop circuit. Instead of inserting a master-slave pair, like in the first step, the flip-flop were swapped with a pair of flip-flops [Chinnery et al., 2004]. If paths then are constrained to half the clock cycle and retimed with a normal flip-flop synthesis tool, the output will be a balanced circuit. The last step is then to swap each flip-flop with a latch, and always let neighbouring latches be of opposite polarity. The downside by using this method instead of a purposely-made latch retiming tool, is that the circuit is not balanced for time-borrowing.

The two-phases, or clocks, should preferably be non-overlapping. It is vital that neighbouring latches are not transparent at the same time, which may introduce oscillating loops. However, the two phases may overlap a small amount, as long as the difference between the launch and capture clock path is not more than the length of the data-path.

2.5.1 Time-Borrowing

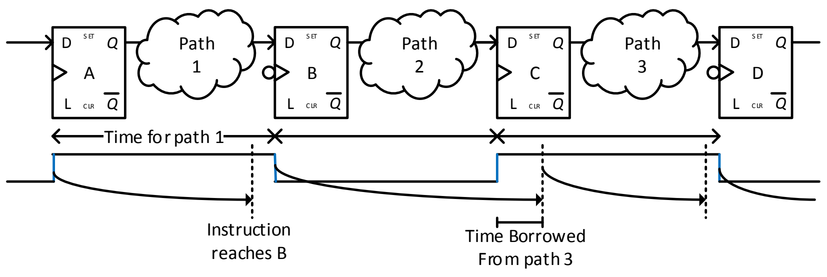


FIGURE 2.7: Illustration of the time-borrow principle.

Time-borrowing is one of the main benefits of a latch design. This enables a faster circuit compared to a flip-flop equivalent. Figure 2.7 illustrates the latch to latch timing. Each data-path is constrained to a half clock cycle, and this is the time instructions got to reach the next latch. The deadline is defined as the edge, which the capturing latch opens. However, as seen in data-path 2, does the instruction not arrive at time. Instead, it borrows some time from path 3 and since this path is much faster than path 2, does the instruction reach D before it opens and the path is again stable. The difficult part of verifying a latch design's timing, is the fact that every path's timing depends on all the upstream paths. Path 3 in the figure cannot be too fast, since it has already given some of it's time to path 2, meaning that this must be taken into account when deciding path 3's length.

2.6 History of Razor

First of all, is Bubble Razor a solution that solves many of the problems that its predecessors suffer from, Razor and Razor II [Ernst et al., 2003] [Das et al., 2006] [Das et al., 2009]. Therefore is Razor presented briefly before Bubble Razor. The principles of Razor, both Razor I and II, also apply to Bubble Razor. This was one of the subjects in our previous work, for a more in-depth discussion of Razor and other solutions please see [Kollerud, 2012].

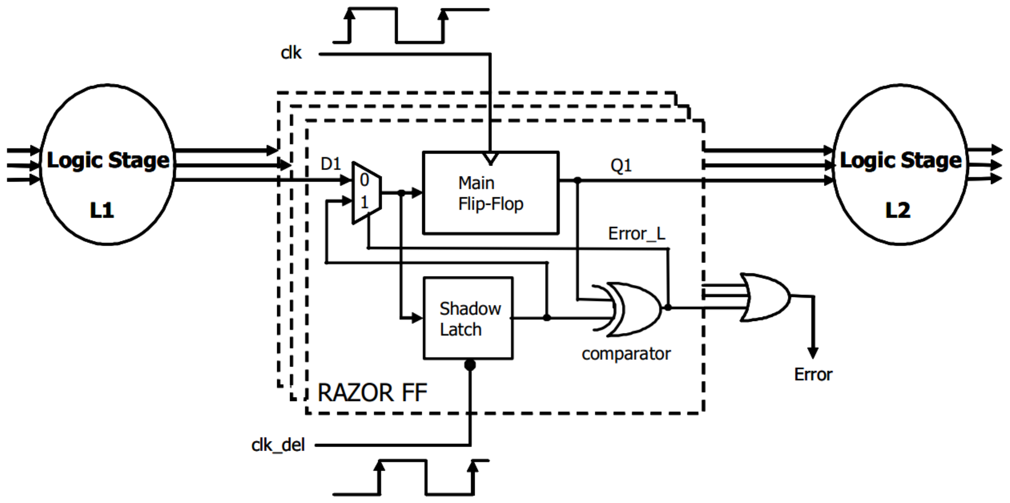


FIGURE 2.8: Razor I. [Ernst et al., 2003]

As mentioned in section 2.3, the first thing to fail as the voltage is scaled is the setup time constraint of the most critical paths. These errors need to be either predicted or detected. Razor is an in-situ error detecting technique that utilize the double sampling principle. Figure 2.8 shows the first version of Razor by [Ernst et al., 2003]. A RazorFF monitor is inserted at the endpoint of possible setup violating paths, most critical paths, to detect if a setup violation has occurred. By sampling the data at the endpoint twice, first at the positive clock edge (main flip-flop) and the some time after this edge (shadow latch), a compare between the two registers will reveal an error. The time difference between the two sample times works as an error detection window.

When the path-delay is too long and the main flip-flop latches the wrong value, the shadow latch, clocked by a delayed clock, will latch the right value. XORing the two stored values reveal an error and the main flip-flop need to be restored. Razor includes a local restore-function to latch the correct value from the shadow latch to the main flip-flop, done by the mux. The error is then used in the feedback to alert the voltage the voltage control.

In addition to the Razor flip-flops themselves, some error recovery is needed to prevent the invalid data propagating to the next stages, ultimately propagating through the circuit and possibly set it in a faulty state. This has been the biggest issue with all the Razor solutions. The Razor solutions are error-detection solutions, meaning the speed is allowed to be scaled to the point where paths fail the setup time and a faulty value is latched. Since error eventually will occur, some error recovery needs to handle this. Proposed error recovery solutions include pipeline flushing and stalling all other stages ones at the same cycle. This is not trivial and makes Razor tricky to apply for a general sequential circuit.

Another problem is the ”short path problem”. The detection window (a.k.a speculation window), the time between positive edge and the point in time when the shadow latch closes, constraints how fast a path is allowed to be. The Razor may issue a false error if the signal propagates through a data-path before the shadow latch latches the data from the prior cycle. A solution for this is to insert delay buffers at these fast paths, but could lead to a large area overhead.

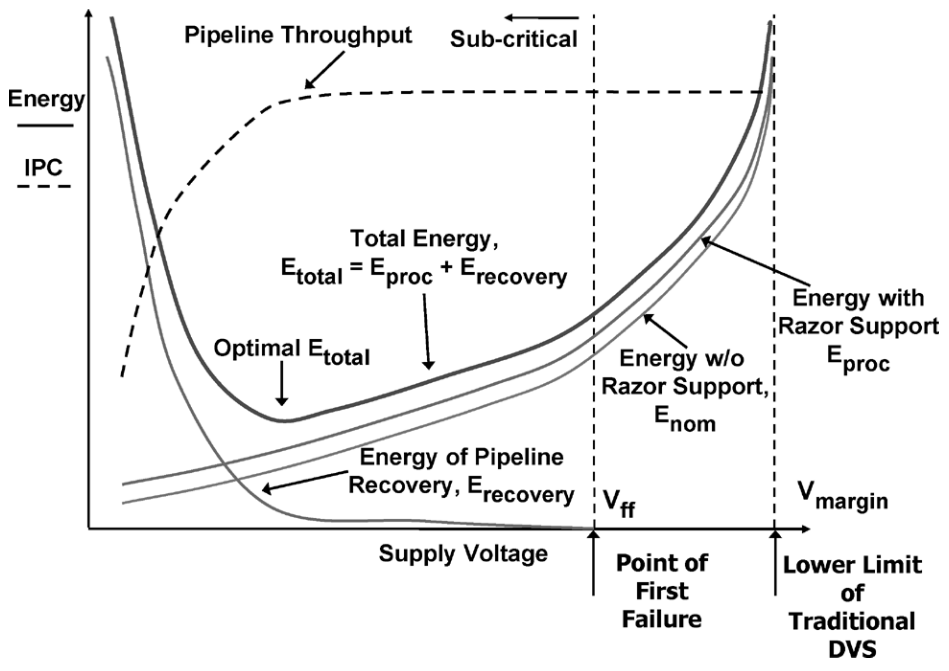


FIGURE 2.9: Illustration of energy saving with error detecting circuitry.[Ernst et al., 2003]

Another technique often mentioned is the Canary flip-flop [Sato and Kunitake, 2007]. The Canary flip-flop is very similar to Razor as it also utilizes an in-situ double sampling technique. However, instead of clocking the shadow register after the main flip-flop, as Razor, do the shadow register latch prior to the main flip-flop. This makes Canary an error predicting method and is not capable of detecting a real error. A real error is when the main flip-flop latches the incorrect value. Instead, Canary may only predict if a path is close to fail its setup time, and then warn the voltage control about it. If a real error should occur, somehow, it will go undetected. Predicting methods do not get rid of all the PVT-margins, as illustrated in section 2.3, it need some margin to guarantee that an actual error never occur.

Figure 2.9 shows the benefits of using an error detecting DVFS method compared to an error predicting method. Error detecting methods are capable of shaving away **all** voltage margins and scale the voltage down to right before failure. Because of error recovery does it even allow to scale even beyond PoFF and gain power savings in exchange for throughput.

2.7 Bubble Razor

Since the rest of the report really rely on the Bubble Razor paper by [Fojtik et al., 2013], will we summarize the main principles and work in this section. The cited figures is original figures from [Fojtik et al., 2013] work. We found these figures pedagogical and good for understanding the Bubble Razor principle. As far as we know, [Fojtik et al., 2013] is the only published article about this kind of Razor.

2.7.1 Basic Principle

Bubble Razor is a new DVFS method based on the same principles as Razor, being an error detection in-situ method. Bubble Razor solves the short path problem and the error-recovery challenge. Where Razor only specify the flip-flop itself and not a recovery architecture, does Bubble Razor include an error recovery algorithm based on a two-phase latch scheme. The idea is that with two phases, does the circuit get a phase extra giving a better time-resolution, or better aspect of time, to correct an error. Furthermore, the algorithm may be used in any design without much knowledge of the internal functionality [Fojtik et al., 2013]. The Bubble Razor algorithm recovers the datapath with only on cycle stall on the out and input port per error. Any Razor-style latch may be used, but it is not necessary with a local recovery in the monitor since this is handled by the Bubble-circuitry and time-borrowing.

2.7.2 Speculation Window and Error Correction on Latch Level

In the previous Razor architectures, did the minimum path delay constrain the width of the speculation window [Ernst et al., 2003]. Bubble Razor, on the other hand, enables a large speculation window of almost a half clock period, and no short path problems.

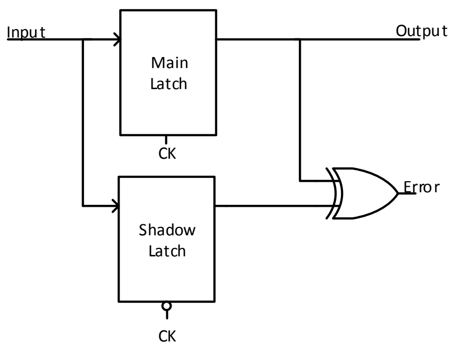


FIGURE 2.10: Illustration of a Razor-latch

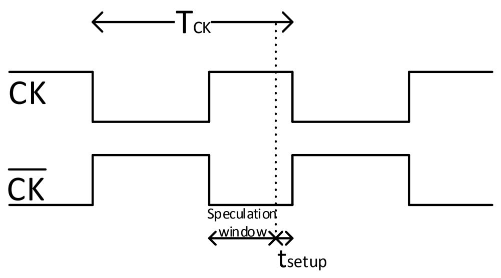


FIGURE 2.11: Illustration of the speculation window of the basic Razor latch

Figure 2.10 shows a basic Razor latch. Although the latch version does not have a local data-recovery multiplexer like the original Razor flip-flop. Except for that, is the error-signal generation the same, but with a latch as the main register instead of a flip-flop. As illustrated in figure 2.11 is the speculation window determined by the width of the main latch's clock pulse and setup time. This means that the most variation in delay allowed between two clock cycles must not exceed the speculation window length. An error is detected when the signal arrives inside this window. Note that the clock on the bottom in figure 2.11 is not the second phase, but an inverted version of the clock on the main latch which is locally generated inside the Razor latch.

An error is generated by the XORgate if the signal propagates too slow and reaches the main latch after it has opened. The deadline for the signal to arrive is the point when the shadow latch switches from opaque to transparent. The voltage cannot be scaled down to the point where a path takes more than a whole clock cycle minus the setup time. If the signal arrives after the speculation window closes, it will be interoperated as the next cycles instruction and will go undetected. Therefore must the lowest voltage allowed be restricted so this does never occur.

Metastability has been an issue in Razor, since it may occur in the main flop-flop, and propagate along the datapath. This is not a problem in Bubble Razor, since metastability can only occur in the shadow latch, reducing the risk of undesired behaviour.

So, when the deadline is violated and an error is issued, what is done to correct this locally? In contrast to the flip-flop Razor where, in case of an error, the instruction must be re-latched to the main flip-flop, does time-borrowing automatically insure the correct value to be latched in the latch Razor. An instruction arriving inside the speculation window will be stored in the main latch due to time-borrowing. A time-borrow will cause an error to be issued, but the datapath is kept intact, for now. However, the next stage has now given away some of its time and is not guaranteed to latch the right value. It has taken the punishment for the failing upstream path, and is itself prone to fail. This is where the clock control kicks in, a stall on the downstream stage/stages will give the instruction time to recover and reach this stage in time. A bubble of stalls is started along the datapath to let the next stages recover or keep them from latching the same value twice, hence Bubble Razor. This process is further explained in the next section.

2.7.3 Bubble Algorithm

A sequence of clock stalling, or bubbling, is started when a monitor issues an error. This sequence is the key to how Bubble Razor may be applied to large designs.

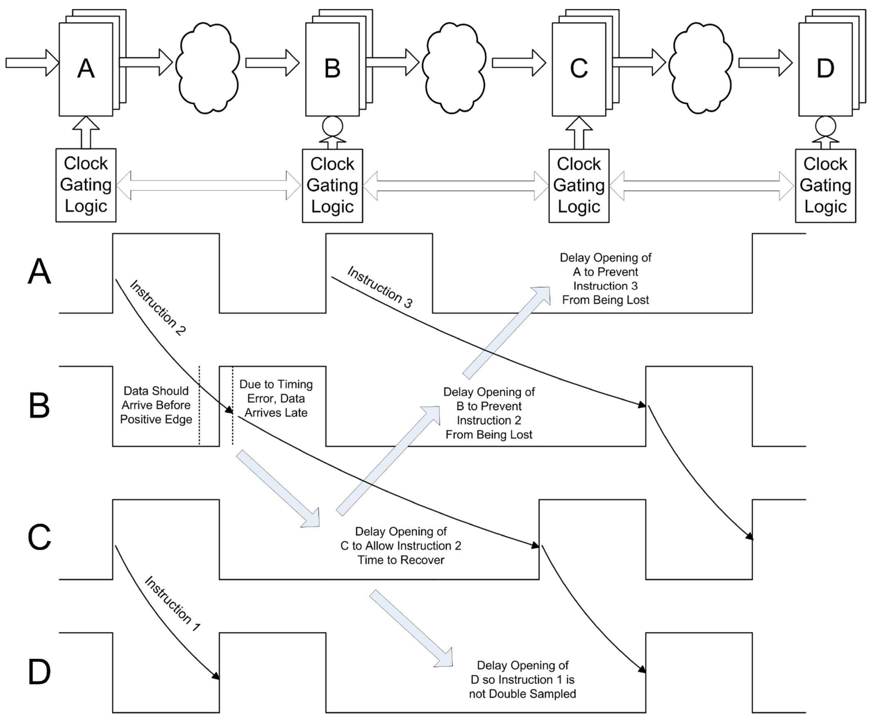

A
B
C
D
Clock Gating Logic
Clock Gating Logic
Clock Gating Logic
Clock Gating Logic
A
B
C
D
Instruction 2
Instruction 3
Instruction 1
Delay Opening of A to Prevent Instruction 3 From Being Lost
Data Should Arrive Before Positive Edge
Due to Timing Error, Data Arrives Late
Delay Opening of B to Prevent Instruction 2 From Being Lost
Delay Opening of C to Allow Instruction 2 Time to Recover
Delay Opening of D so Instruction 1 is not Double Sampled

FIGURE 2.12: Illustration of the recovery of a datapath. [Fojtik et al., 2013]

Figure 2.12 shows how the datapath is restored in case of an error. This is the point where the two-phase latch design comes in handy. The extra phase enable a stall without immediately losing any data in the neighbouring stages. A stall in a one-phased flip-flop system would make the flip-flop upstream to the error latch a new value and overwrite the old value before it is stored in the next stage.

Figure 2.14 shows *Clock Gate Control*, which controls the bubbling. The control logic follows a very simple set of rules. Each individual control is not aware of how many neighbours it has upstream, downstream or where it is in the system. This reduces the area and the complexity and makes it possible to apply without knowledge of the design. The Bubble algorithm given by [Fojtik et al., 2013] is as follows:

1 A latch that receives a bubble from one or more of its neighbours stalls and sends its other neighbours (upstream and downstream) a bubble one half-cycle later.

2 A latch that receives a bubble from all of its neighbours stalls but does not send out any bubbles, making the bubble process end.

3 Multiple errors at the same time are handled in the same way. Stages do not know how many errors there are in circulation, or where they originate from.

Figure 2.13 shows a bubble sequence in a simple test circuit. It is easier to see how the algorithm propagates through logic by following the rules in listed above. Each box, 1 to 8, is a monitor latch with a cluster control module. White boxes mean normal operation, solid red mean the data reaches the latch after it has opened and this latch issues an error. Solid blue is a stalling latch while red striped is a latch that stalled last cycle and cannot stall or send bubble at the current one.

An error is detected at step *1* in latch *6*. Step *2* are phase 2 latches supposed to latch incoming data, but due to the late data in latch *6*, *8* must stall to be ensure the instruction is recovered properly. This is the initialization of the bubble sequence. Next step, step *3*, do phase 2 latches open, and latch *8* sends bubbles both up- and downstream making latch *1*, *6* and *7* stall. Note that latch *8* do not stall in step *4*, since it stalled last time. The bubble sequence end when a latch gets bubbles in from all of its neighbours, like latch *3* in step *5*. Note that every latch only stalls once.

2.7.4 Bubble Circuitry: The Cluster Control

Figure 2.14 shows the Bubble Razor components. Cluster Control Logic is the same as Clock Gate Control logic, but as described later, are latches clustered into groups to reduce logic area from control logic. This means that multiple latches clocked by the same phased clock may share Cluster Control. The Cluster Control is identical for both phases; the only difference is which clock they run on.

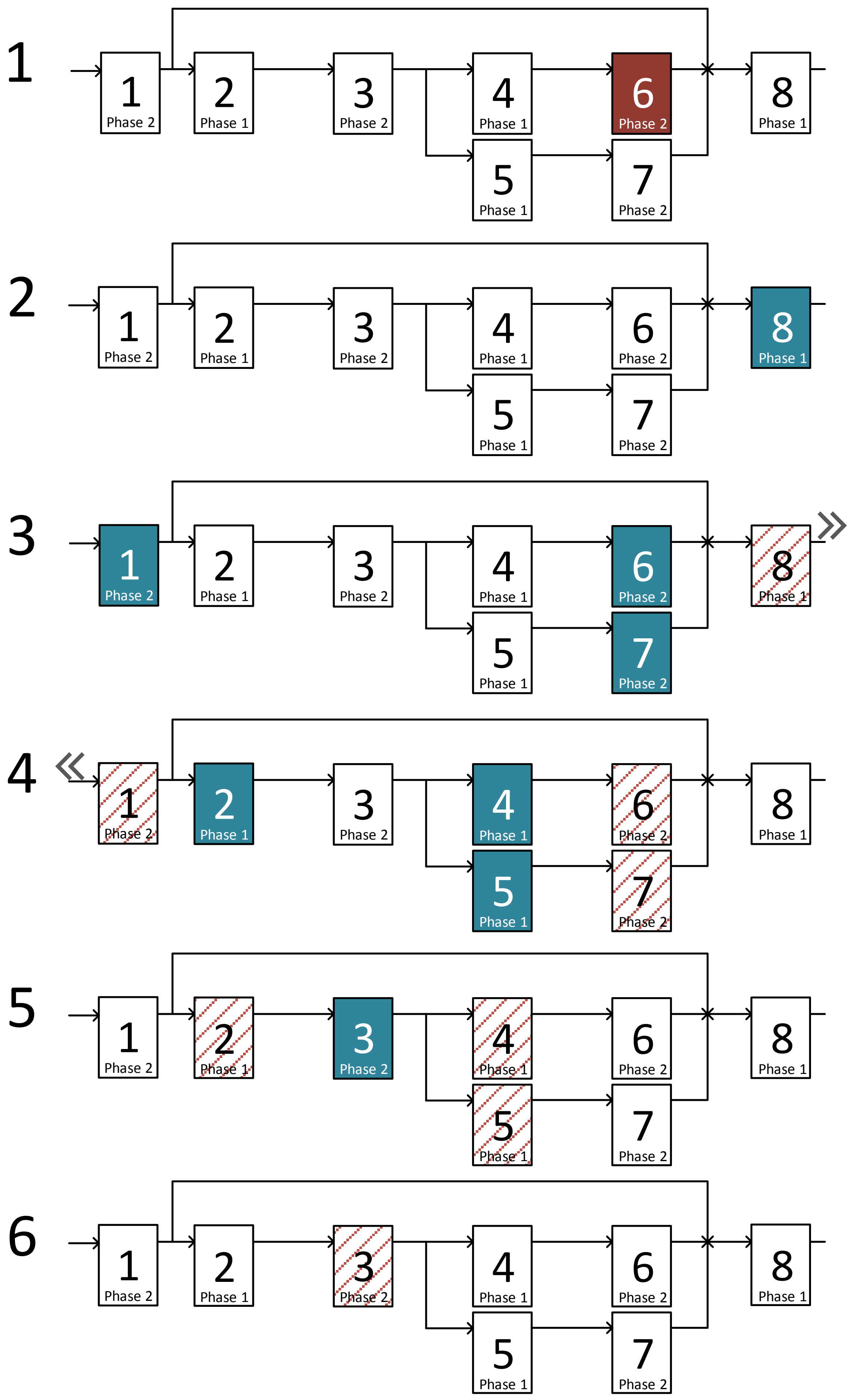


FIGURE 2.13: Example of the bubble algorithm.

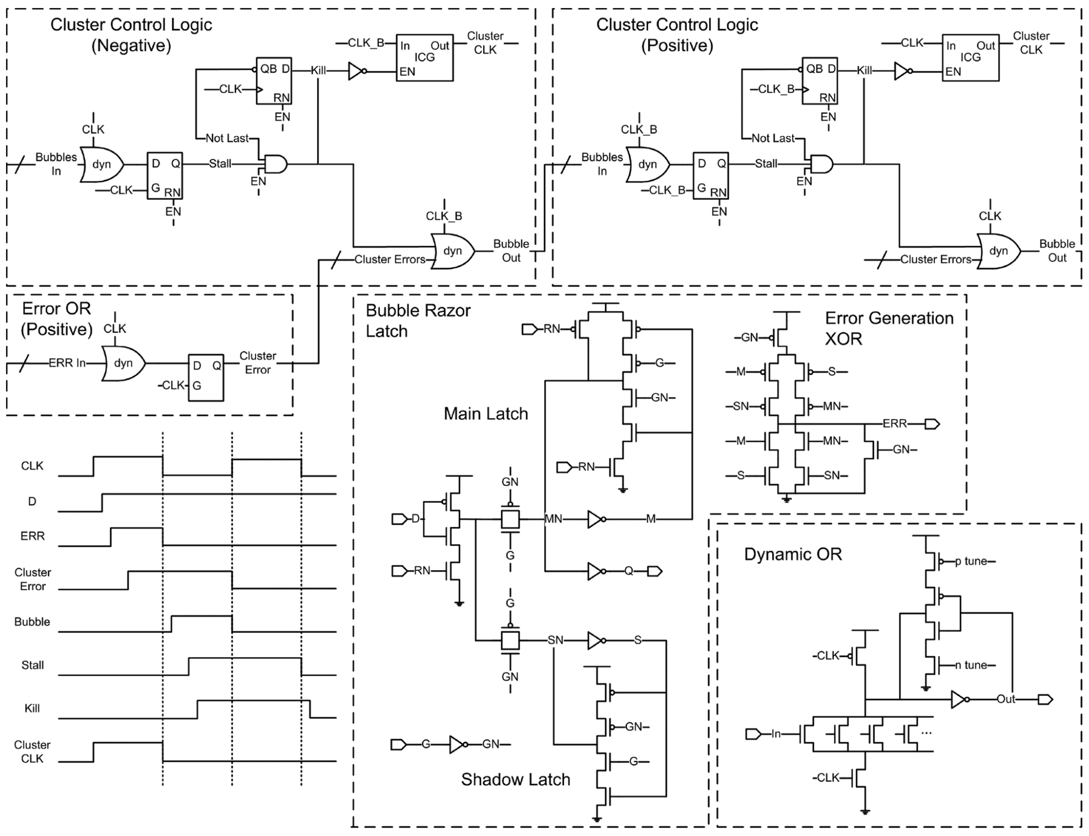


FIGURE 2.14: The Bubble Razor circuitry made by [Fojtik et al., 2013].

The Cluster Controls are the components that propagate the bubble signals and stalls the latches. Its main task is to gather bubble signals from neighbouring Cluster Controls and stall its member latches if it did not stall last cycle. Its other task is to gather all the error signals from its member monitors to a *Cluster Error*-signal. If a member monitor violate it's timing constraint and issues an error, this will be picked up by its Cluster Control and cause the bubble process to be initialized.

[Fojtik et al., 2013] uses dynamic OR-gate trees with maximum 16 inputs for both the *Bubble In* and *Cluster Error* signals. It is crucial that these error signal paths are fast. Delay through the OR-trees is makes the speculation window shorter. Dynamic gates are presumably used to decrease the delay. Why the delay through the OR-trees affect the speculation window is illustrated and explained in section 3.3.1.

The error signal from each Razor latch is only valid when the main latch is open and the shadow latch is opaque. This is the reason for the clock-controlled pull down in the XOR-gates. When the main latch is closed and the error signal is not valid, the shadow latch's input will toggle and glitch before stabilizing. This glitching would propagate through a standard static XOR-gate, but by pulling the XOR's output low, the signal will be stable zero until a valid error occur during high clock pulse. The pull down XOR-gate will be static zero as long as a valid error does not occur. Therefore, will there only be a small amount of toggling in the OR-trees, since an error is relatively rare.

Dynamic gates need to be clock with a frequency higher than a certain threshold to prevent the charge draining out. This is solved by a latch at the end of the *Cluster Error*-signal, making the OR-trees operate regardless of the clock frequency.

The bubble signals are then used in the feedback to the DVFS control logic. The control logic probes the bubble network at different intervals depending on how fast the regulation need to be. The voltage regulator control will increase the voltage if it picks up bubble activity somewhere in the bubble network. As mentioned in section 2.3, is there two kind of feedback to the voltage regulators, either up/down or only up, where Bubble Razor will only give an up feedback.

2.7.4.1 Clusters

To reduce area overhead from the cluster control logic, latches clocked by the same phase may share the same Cluster Control. Why not assigning all slave latches to one cluster and all masters to another? First of all would the OR-tree collecting Razors' error signal get a huge fan-in, thus be too slow. Another problem is the clock networks. It will be, if not impossible, very difficult to control half of the latches in a chip from one clock gate with single cycle precision. The delay through the clock network will possibly too large from buffering.

Cluster Controls are not aware of how many latches or monitors they control, and clustering do not change the bubble algorithm or the bubble modules. [Fojtik et al., 2013] cluster latches of same polarity with many common neighbours. There is a tradeoff between the size of OR-gates for the internal *Cluster Error* and the size of the OR-gates for the bubbles between clusters. They do the clustering by representing the circuit as a positive and a negative graph. Where the negative graph contain all the latches, and Razor monitors, clocked by the one of the phases, and positive graph of all clocked by the other phase. The edges between each vertices (latches & monitors), is weighted by the number of common neighbours between them. These graphs is than clustered by a hypergraph partitioning tool [Fojtik et al., 2013], with constraint on the size of the clusters to keep the OR-gate size down.

Chapter 3

Methodology

3.1 Setup

A good base design was needed to test the methodology of inserting Bubble Razor and its behaviour. The aim is to convert a competed sub-module to a fully functional Bubble-Razor system. [Fojtik et al., 2013] uses commercial tools and scripts, but do not say what is done by scripts and what is done by tools. We will investigate how to do the transformation with the tools available, and custom scripts. The test design need to fit some requirements:

1. **Flip-Flop based**

   One of the advantages of Bubble Razor is that it should be able to fit in any flip-flop design without the knowledge of the functionality. It should obviously fit in a dual-phase latch design as well, but the most common sequential architecture is the flip-flop design.

2. **Path Delay Distribution**

   The path delay distribution should represent a typical design. There should be critical paths as well as less critical paths.

3. **No Hard Macros**

   Hard macro cells do not report the correct timing in a Static Time Analysis.

The delays through these cells are often defined with a very large margin. Hard macros are not desired in this study.

4. **Testbench**

   Since the design will be attracted as a black box with no knowledge of the internals or the actual functionality, a good testbench written by the module designer must be available. This testbench will be used to confirm a correct operation between each conversion step.

5. **Size**

   The size of the module cannot be too large, but it must be big enough to get a good set of paths. Simulation time, particularly the analogue simulations, will be large if the design contains too many cells. From experience gives a size of 200-400 registers a good turnover time, and still large enough to test the methods and Bubble Razor.

6. **Clock Gates or multiple Clock Domains**

   Clock gates and/or more than one clock domain is something that is often used and is as far as we know not described in any Bubble Razor publication. If Bubble Razor really is applicable in any design, this is one of the things it should handle.

It was decided to use the same chip as in our previous study, a 180nm radio chip, in co-operation with Nordic Semiconductor. However, it will now only be one sub-module in compliance with the list above and not the whole chip.

With help from some of the designers, we landed on a sub-module believed to fit our requirements. This module is a digital filter and comes with a testbench. From now on, this module is named DigitalFilter.

3.1.1 Path Analysis of DigitalFilter

To verify that the paths in this module have a slack distribution that represents the whole design, were the path-analysis scripts from our previous work used to map the paths. This module runs at 52MHz and was therefore not part of the last study where only the 16MHz domain where analysed.

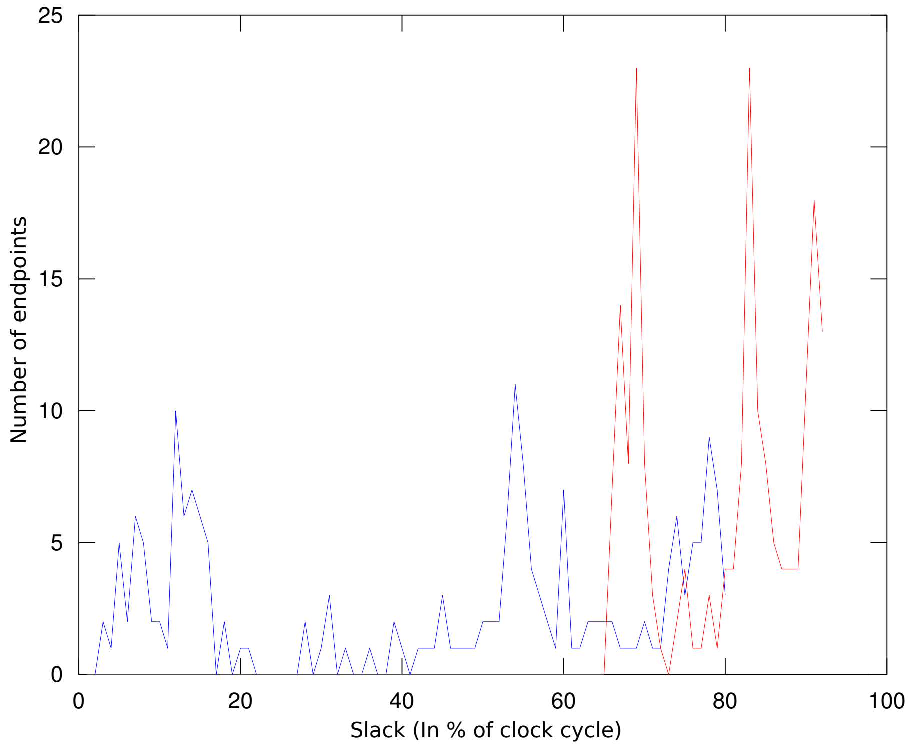


FIGURE 3.1: Plot of the slack distribution in DitalFilter. Blue is slow and red is fast corner.

Figure 3.1 shows a plot of the slack in DigitalFilter. There is a larger group close to zero slack, which tells that this module got a group of critical paths. The next plot, figure 3.2, displays how each individual path's delay change between the corners. There is no unusual incline which prove that the module do not contain hard-macros or special cells. DigitalFilter contains 243 flip-flops.

3.1.2 Verification

The module has a functional testbench to be used for verification of each step in the conversion between a flip-flop design to a Bubble Razor design. However, the testbench is not the typical GO/NO-GO testbench, but is instead a RTL-testbench with a MATLAB script to confirm the right filter response. Further do the testbench only connects to the ports of the module, and do not probe into the

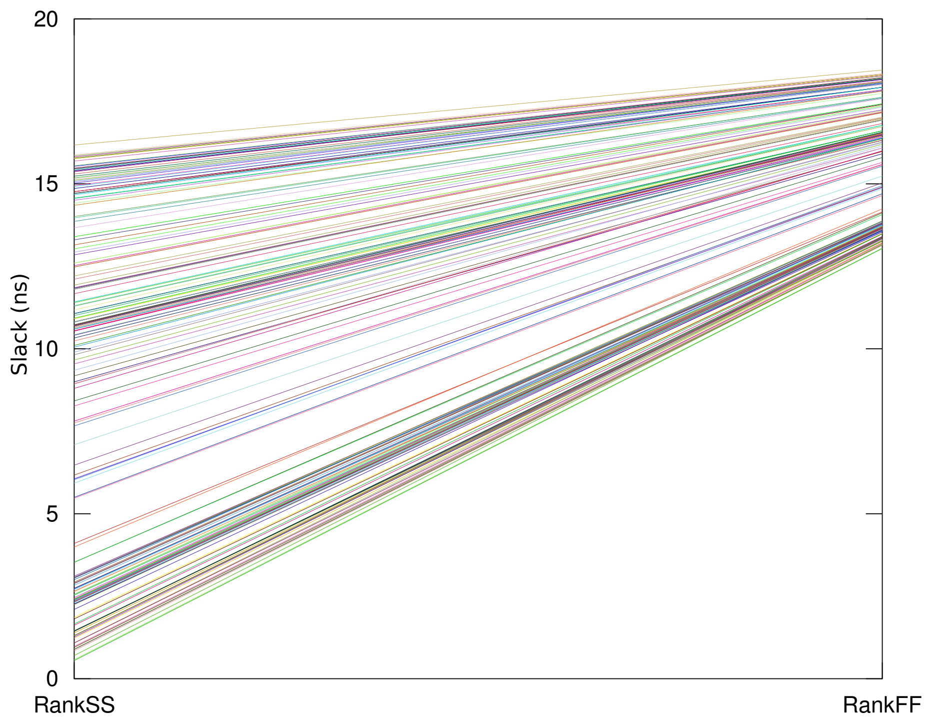


FIGURE 3.2: Slack movement over the slow and fast corner, where slack is Y-axis and corner is X-axis.

design. This enables the testbench to be used on any of the upcoming modified versions of the design as long as the interface is the same.

The testbench will be ran for each of the major modifications done to the design. Since the correct behaviour of the DigitalFilter is unknown, will the behaviour of the modified design be considered as correct if it matches the output from the original filter response.

3.1.3 Clock and Clock Gates in Case Module

DigitalFilter contains two clock gates. One of the clock gates is used to turn on or off a smaller section of the module while the second gate is used in a clock divider for a larger portion of the filter. This enables us to test how Bubble Razor behaves with more complex clocking and clock domain crossings. All clocks are synchronous.

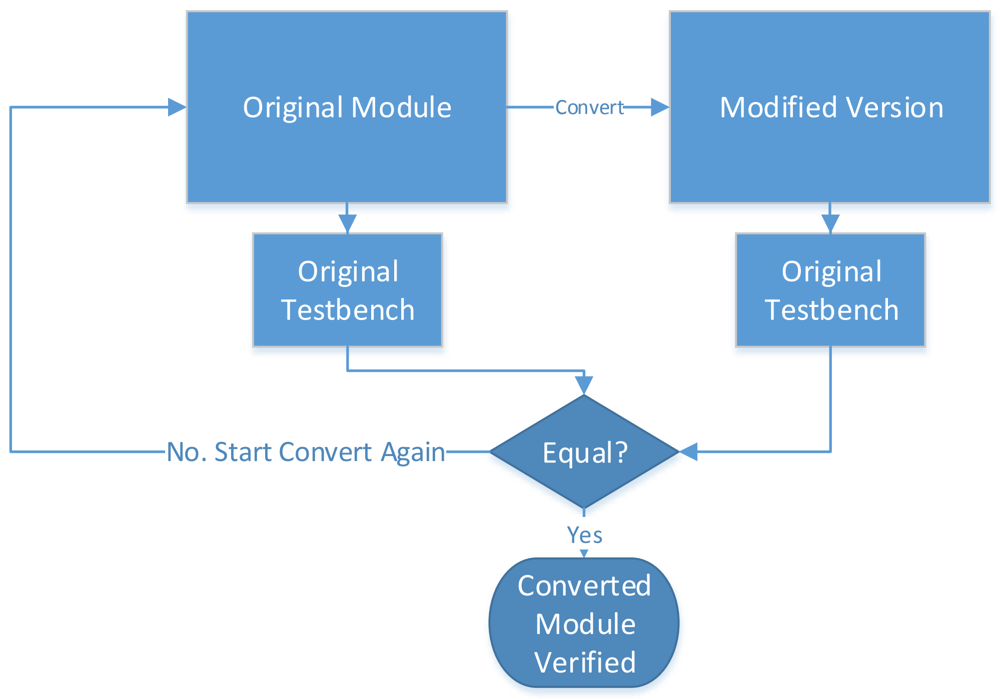


FIGURE 3.3: Verification of the RTL/netlist-changes.

3.2 Step 1: Converting to a Two-Phase Latch Design

This section explains how the DigitalFilter were converted to a two-phase latch design. These steps all depend on what software tools and what kinds of cell libraries are available. Most of the steps are done by scripts, but the vital re-timing step depends on a synthesis tool.

The original design is synthesized from RTL to a verilog netlist, which is the base for the conversion. RTL is left untouched; every modification is only done to the netslists. Every step, except retiming, were done by custom scripts.

One of the most characteristic things about Bubble Razor is the two-phase latch architecture. It is crucial to be able to easily convert any flip-flop design to a latch design for the Bubble Razor system to work at all, since most designs are based on the typical flip-flop architecture. The method used here is based on the latch rules from section 2.5.

3.2.1 Cell switch

Insertion of latches is based on the fact that the synthesis/retiming tool used in later stages is able to calculate the right output drive and balance the paths by adding or removing latches.

The initial move is swapping every flip-flop in the netlist with two latches, one for each phase, where the master is clocked by phase one and slave by phase two. This will make the module as a black box behave just like the original design. Next is the insertion of wires between the latches. The two latches are now connected directly together and appear as one master-slave flip-flop. It is important to use latches corresponding to the flip-flop being replaced. If the original flip-flop had asynchronous active-low reset or used the inverted output, should the master-slave replacement also be the same. Rest of the changes is inserting declarations based on the syntax of the netlist language, in this case verilog.

Script is found in appendix A.

3.2.2 Clock Tree

Unlike a flip-flop design, where only one clock tree is made, does a latch-design need two phases. On the other hand are the timing requirements for the two clock trees in a latch design less strict with skew in mind, which means that each tree is smaller and less power consuming than the tree from a flip-flop design.

This study is not going to be taken to the layout stage. The clock tree is often an ideal network in all stages before layout and the comparison would be more correct with an ideal clock for the BubbleRazor design, since the original design is not laid out.

The list below describes the two ways of setting up the clock in a latch design:

1. **Locally generated second phase**
   The basics behind the local generated second phase is in the name itself. One clock is routed throughout the design, where the second phase in each master-slave pair is made locally with inverters. This is possibly one of the best methods for generating the clock tree, but will introduce buffers in the clock tree which is not done in the original design. This buffer difference will introduce an error in the power estimation. Small local variation may introduce an error to the non-overlap constraint, but this will cause no troubles as long as the data signal uses longer time than the non-overlap.

2. **Generate a tree for each phase**
   Instead of locally generate the two phases with buffers and inverters, is the module granted the second phase from an external source. Both phases are then routed as two clocks throughout the design. This way no extra buffers need to be introduced for the upcoming power simulations. Another upside by having the two phases independent of each other is that this enables tweaking with the non-overlap time a duty cycle in later analogue simulations. Therefore is this clock solution used in the rest of the study.

The second option was chosen for the reasons given in the list. Script in appendix B sets up the second phase in each sub-module.

At this stage, the latches are inserted and a phase two is introduced throughout the design. However, as the next two sections explain, does the module contain clock gates that need to be modified to suit the active low latches and the two phases.

3.2.2.1 Active Low Clock Gates

The 180nm std. cell library used in this study did not contain active-high latches, which introduce some modifications to fit active-low latches. It did neither include an ICG (Integrated Clock Gate) for active-low clock setup. A ICG made of other cells shown in figure 3.4. This is only needed if active-low latches are used. It is important to note that a clock must always be set to the inactive phase when turned off. If it is not, oscillation loops may arise since both the master and slaves would be transparent.

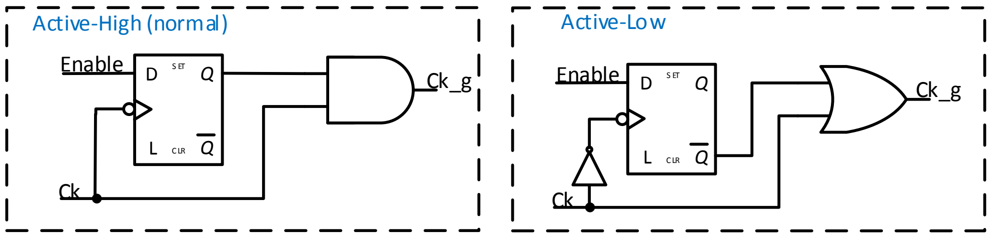


FIGURE 3.4: Clock gate for active low latches. Glitch free.

3.2.2.2 Two-Phase Clock Control

The clock gate must undergo further modifications. This clock control outputs a equal duty cycled pair. However, as explained later in the BubbleRazor section 3.4.2 is there a trade-off between equal or non-equal duty cycle between the two phases in the clock domains when inserting Bubble Razor.

Another aspect with clock-gating two phases is that for each time the gate is open, a pair of pulses needs to be gated, one for each phase. This gets important when the gates are used for more than turning clock domains on or off, but instead be used for more complicated gating such as clock dividing or pulse picking. Further, the masters need its pulse before the slaves gets its.

Figure 3.5 shows the clock gate used in this study, note that only the original gates from the flip-flop design are switched to this. Some different gates are introduced for the BubbleRazor logic, which is not switched to this gate. At first glance, the gate configuration in figure 3.5 may seem unsafe, and yes, it is not trivial to send the *enable*-signal directly to the OR-gate to generate the first gated phase. The *Enable*-signal is the same as the *Enable*-signal in figure 3.4.

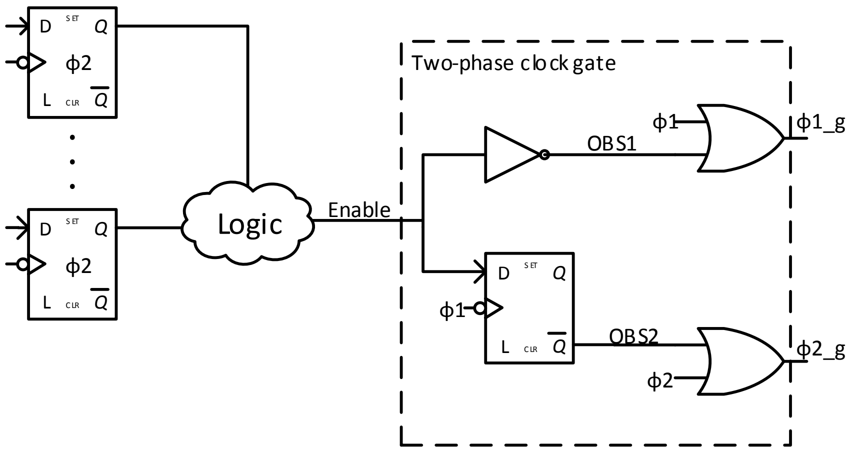


FIGURE 3.5: Illustration of a clock gate for both phases. Figure shows the active-low version.

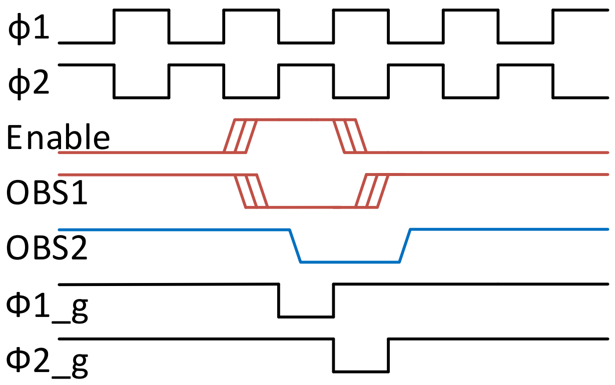


FIGURE 3.6: Illustration of the two-phase clock gate behaviour.

It is important to constrain the data-path from the phase-two latches to the OR-gate with good margin within a half period. If the *Logic* part generating *Enable* is too large, or has a too long delay, this gate cannot be used. However may the synthesis tool be able to retime and balance some of this logic upstream of the phase-two latches, giving more time for the *Enable*-signal to reach the first OR-gate.

The reason why this gate is being used, is to get the first, phase one, clock through before the phase two clock as shown in figure 3.6. As mentioned in section 3.2.1, the last of each inserted pair of latches, the slave latch, is clocked on the phase two clock. This means that the enable signal from the original design is now being clocked on phase two. Furthermore does the design rely on getting the first pulse after the *enable* is set.

By now, the design has a complete clock tree and it is run through the testbench with ideal timing. This confirms the logical behaviour of the circuit, though the circuit would not work in a simulation with real delay on the data paths due to timing violating paths, which is fixed in the next section.

3.2.3 Retiming

The retiming is a very important step to get the logically correct latch-design meet the timing requirements. The middle circuit in figure 2.6, section 2.5, shows the design at the current state. The goal is to balance all paths to a half clock cycle and end up with the last stage.

The tool used for retiming is *Design Compiler* from Synopsys. Design Compiler include a command, *optimize registers*, that retime circuits [Syn, 2011]. Fortunately do this command supports latch designs, it even supports time-borrowing. However, is time-borrowing something not wanted under the conditions in slow corner. Synthesis tools use libraries specifying cells at normal operation condition, but as [Fojtik et al., 2013] mention, is time-borrowing something not desired when voltage is not scaled. An error is issued when a path borrows time. At 1.2V, which is the slow corner defined in the cell library, should the circuit run without any time-borrowing.

The data-paths between each latch are constrained to half clock period minus setup time and clock uncertainty. Paths between latches and input/output ports are constrained to a quarter clock cycle. Time-borrowing is turned off and the clock-network is set as *don't touch* and ideal. DigitalFilter is now a complete latch design.

3.3 Step 2: Bubble Razor insertion

This section explains how we inserted a complete Bubble Razor system to the DigitalFilter, which at this point is a complete two-phase latch design. Section 3.3.1 explains how we made the components in figure 2.14 by [Fojtik et al., 2013] with our std. library. Section 3.3.2 to 3.3.6 explains how Bubble Razor is inserted to our test module, DigitalFilter. However, as mentioned do DigitalFilter include clock gates, which introduce some challenges. The steps in this section is enough for a design without clock gates. The clock-gate problem and a proposed solution for the clock gates is described in section 3.4.

3.3.1 Algorithm and Components

Before starting implementing Bubble Razor were the components from figure 2.14 made from the set of std. cells available. In addition, do these cells need to operate in an active-low system like the latch version of DigitalFilter.

Figure 3.7 and 3.8 are the active-low versions of the modules by [Fojtik et al., 2013], figure 2.14, the only difference being active-low latches and the clock gate. The state-holding flip-flop in the Cluster Control is now clocked by the other phased clock, since a negative edge flip-flip is not available. This does not change any behaviour.

A test circuit with a testbench was made in SystemVerilog to ensure the circuit behave as desired. The algorithm is already been confirmed to work by [Fojtik et al., 2013], but there were some confusions. Which cluster error is connected to which Cluster Control? "... it was noted that upon initiating the bubble propagation sequence after detecting a timing error, the first clock gating event is optional, so clock gating does not take place during the first bubble" [Fojtik et al., 2013]. Initially, this were interpreted to that the first downstream latch, after the monitor issuing the error, do not need to stall, but just send bubble up and downstream. This means that latch *8* in figure 2.13 do not stall. The way [Fojtik

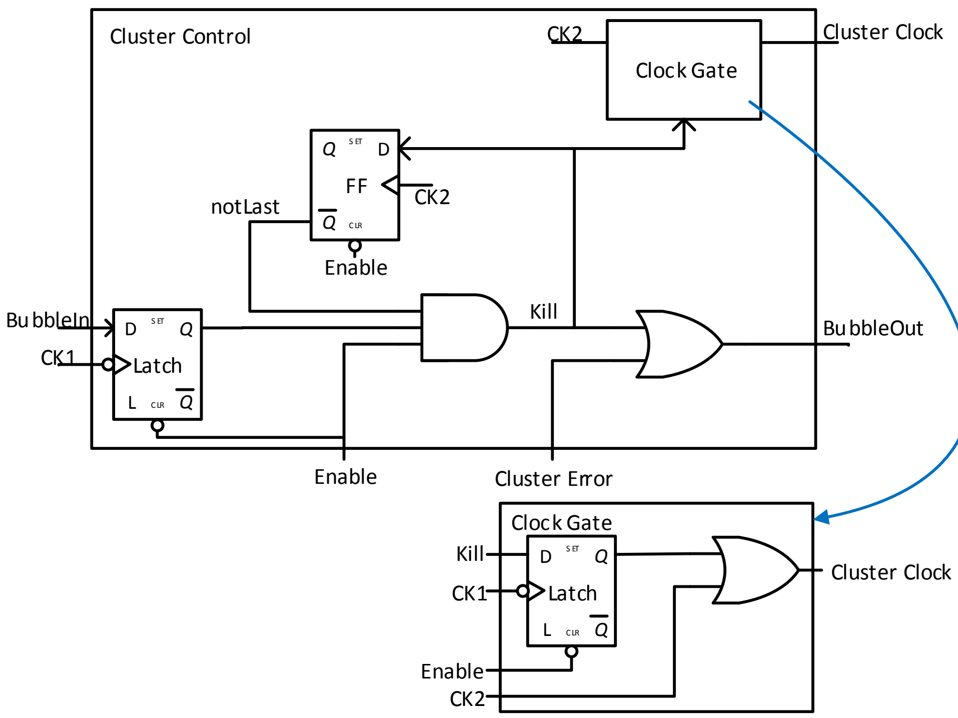


FIGURE 3.7: Illustration of the active-low cluster control.

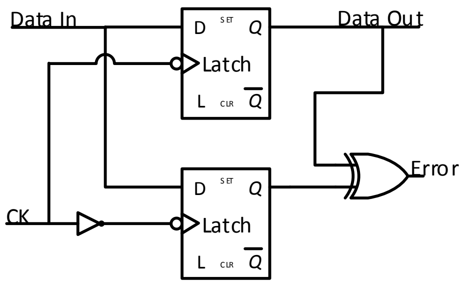


FIGURE 3.8: Illustration of the active-low Bubble Razor monitor.

et al., 2013] samples the error signal in the Error OR-tree, on the opposite clock of the cluster clock, also indicates that the error signals going in to the cluster control is not from the monitors within the cluster. The error signal from each monitor is valid only when the main latch is open. The Error OR, in figure 2.14, is the latch receiving *ERR In* clocked on *CLK* which implies this error signal is issued by a monitor clocked on the other phased clock. Without saying that this is wrong, we may consider another version of this algorithm.

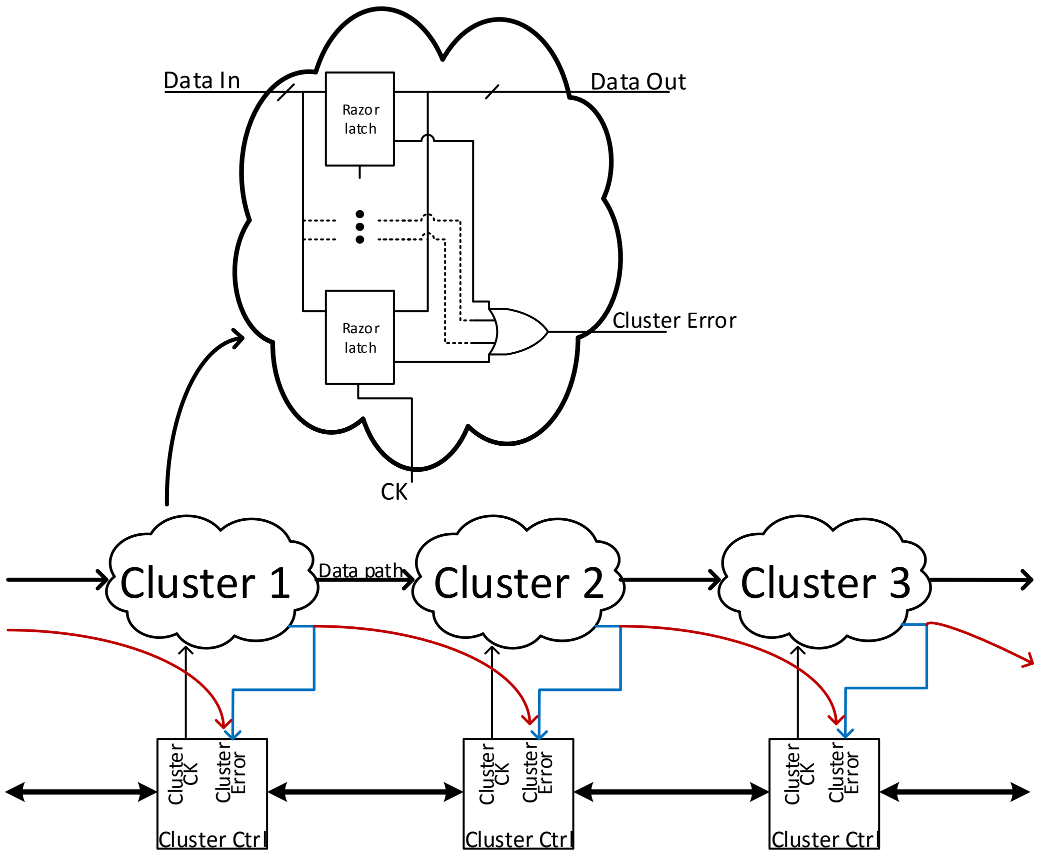


FIGURE 3.9: Alternative Cluster Error Routing.

Figure 3.9 shows the two *cluster-error* routing options considered. The red wires is the *Skip first stall* option, blue is a *dual error injection* version. In the *skip first stall* option are cluster errors from each latch connected to the downstream latches' cluster controls. The *dual error injection* is cluster error connected to the same cluster control controlling the monitor's clock.

The different behaviours of the two options are illustrated in figure 3.10 and is based on the example of the ideal algorithm in figure 2.13. The *Optional first stall* to the left shows an error in latch *6*, which sends the cluster error downstream to latch *8*. Latch *8* do not stall, unlike to the example in section 2.7.3, but instead sends bubbles both up and downstream. The behaviour is identically to the example, except latch *8* stalls in step *4* instead of *2*. If we have interpreted the "skip first stall" correct, do the wave at the left bottom illustrate the possible

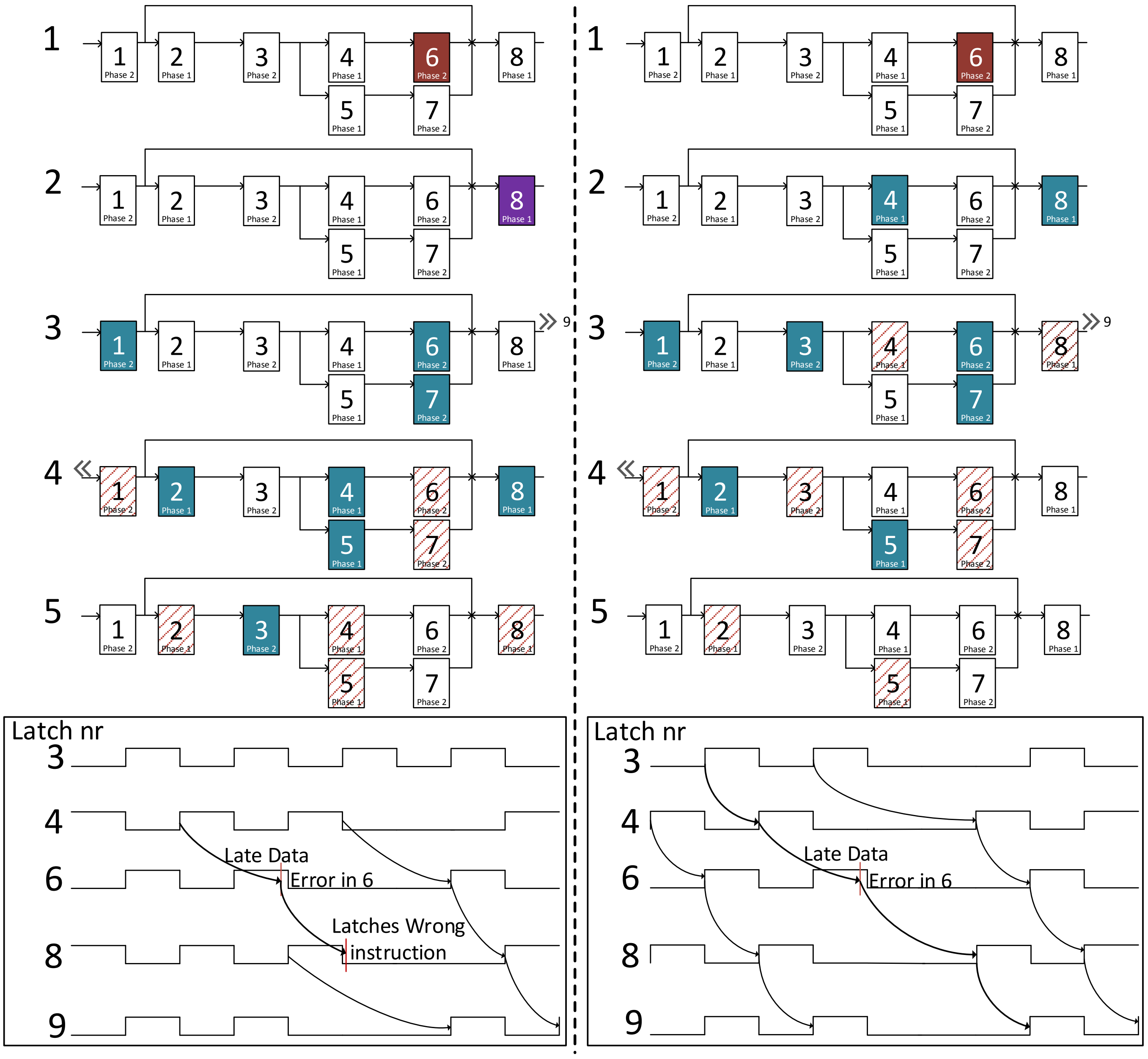


FIGURE 3.10: Illustration of the impact of the two cluster error routing alternatives. *Optional first stall* to the left, *dual error injection* to the right.

hazard with this method. The wave shows latches *3, 4, 6, 8* and *9*, which is in the correct order. This example shows the circuit under very slow, but legal condition, which makes the data arrive very late at latch *6*. This is not a problem in the example from section 2.7.3, where the data get an extra half cycle to reach latch *8*. However, if latch *8* do not stall, hence the optional first stall, does the data has only a little over half cycle to reach *8*. Under normal operation is this not a problem, since data paths use under a half cycle anyway. On the contrary, the paths are already operating slow when latch *6* fails with this margin, consequently is the path between *6* and *8* also slower than normal. There is no guarantee that this path is below a half cycle, under these conditions. If this instruction misses the deadline in latch *8*, will the instruction be lost since latch *6* will overwrite it before *8* opens again. This happens when two paths in succession combined is over 1.5 clock cycle long. If *optional first stall* is going to be used, path pairs must be constrained for this. We do not tolerate a latch error.

The upside of using *optional first stall* is less constrained OR-trees. *Cluster Error* do now have a half period extra to reach the recipients. As discussed later, do the delay through this OR-tree affect the speculation window.

Dual error insertion is the other way to connect the control circuitry. This is the method selected for the DigitalFilter. *Dual error insertion* is very similar to the ideal bubble initiation from 2.7.3. The different is that a bubble is not sent just downstream, but also upstream in the first step, as shown to the right in figure 3.10 (step *1* to *2*). This simplifies the automation done in the next chapters by routing the cluster error signal, as shown in blue in 3.9. However, won't inserting two errors result in two stalls in each latch? No, the circuit does not know where the bubbles originate from or how many are bubbling around. The two bubbles will cancel each other out in the cluster/cluster-control pair issuing the error and then propagating from this point out. This method do not have the problem where an instruction may be lost, however is this method dependent on a low delay through both *cluster error* and *bubble in* OR-trees. An error must propagate from the Razor monitor and be stable on the input of the clock gate in the neighbours' cluster control before the next clock pulse arrives. This reduces the speculation window by limiting how late an instruction can arrive, illustrated by the red line in the wave illustration in figure 3.10. This is the same constraint as the original algorithm.

The active low components and the *dual error injection* initiation where tested in ideal logic simulations and chosen for this project.

3.3.2 Analysing and Mapping DigitalFilter

The functionality of DigitalFilter is unknown, but the relationship between each individual latch needs to be mapped for the insertion of Bubble Razor. The routing of each signal introduced with Bubble Razor maps directly to the local connection between latches. Which clock the latches are clocked by and what slack each latch has on its input is also vital information.

Initially, a custom Verilog parser was considered. However, by using DesignCompiler instead enables getting more information without making a parser. By using DesignCompiler's *report_timing*-function on each latch and input-port in DigitalFilter, obtaining slack and downstream latches for every latch. This data were extracted and are the base for a lookup file of each latch in the design, intended for the insertion of Bubble Razor. DesignCompiler also include command for reporting clock information. It gives the full clock path to each latch allowing the script to get the last clock gate, or which clock domain the latches are clocked by.

It is possible to report the whole design in one command for timing reports and one command for clock tree reports, but the computer's memory were not large enough. To overcome this, a wrapper script written to report for each latch and input port and then dump it to a file, appendix C. This report file were then imported to another script, which generates the actual lookup of all the latches, in addition, this is also the script handling the cluster control lookup. There are no clusters for now, every latch have their own control module. The cluster part is described in sectin 3.3.6, but it is the same script performing the clustering.

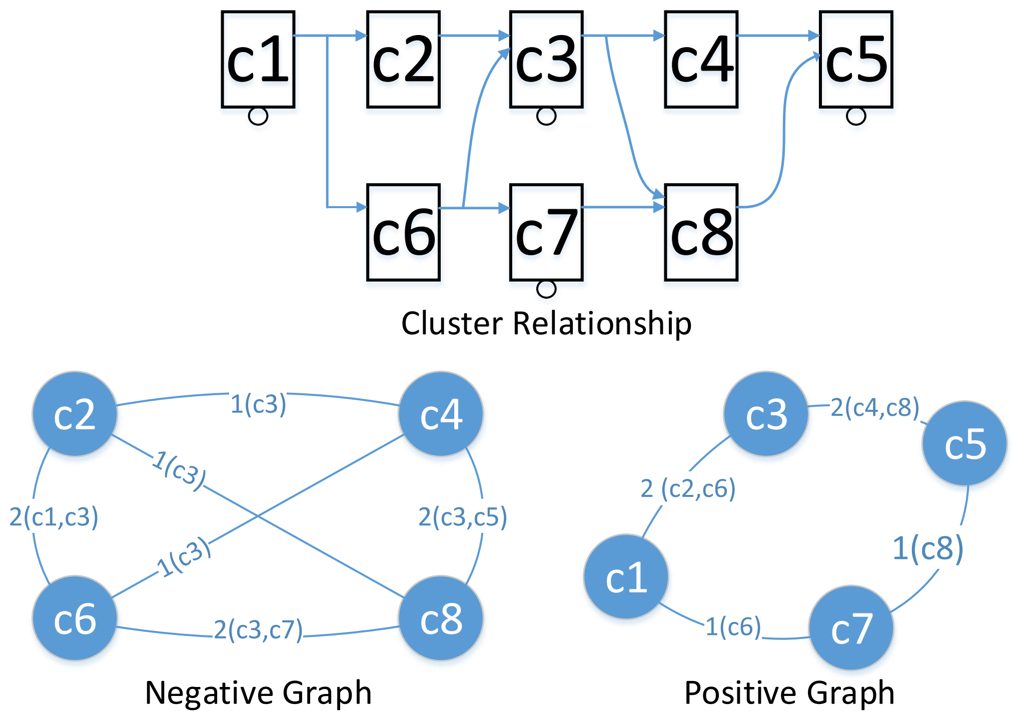


Figure 3.11: Cluster relationship graph example. Number is common neighbours and parentheses list the common neighbours.

Figure 3.11 shows an example of a mapped circuit. The script makes, just like [Fojtik et al., 2013], a positive and a negative graph showing how many common neighbours each latch of the same polarity share. This is used for the clustering algorithm. See section 3.3.6.

This step, like all the other steps, is important to automate. Notice that the runtime of the automated DesignCompiler wrapper script is significant. However, it is OK for the work in this report due to the size of DigitalFilter. A better solution is to write a Tcl script native for DesignCompiler, which will reduce the runtime for larger designs in the future.

An example of the lookup file may be seen in appendix F.

3.3.3 Deciding the Number of Monitors

Which latches should be replaced by monitors? [Fojtik et al., 2013] propose tree different methods of selecting paths, or latches: only negative latches, only positive latches or a subset of all latches. A 100% speculation window is achieved by replacing all latches, however do this increase the area a lot. By only applying Razor latches to only negative or positive latches will result in half the speculation window time, since only every other latch monitors, but half the number of extra Razor logic.

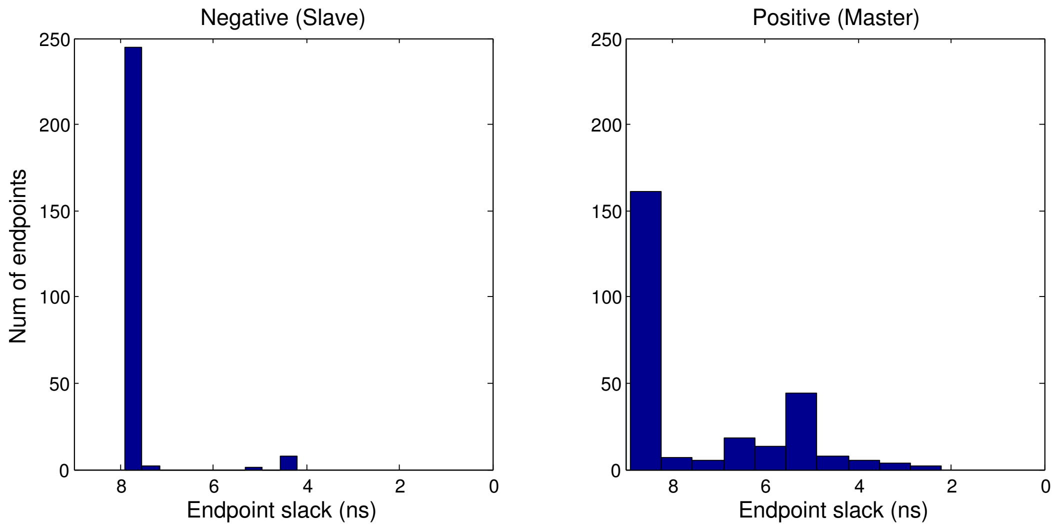


FIGURE 3.12: Histogram of endpoint slack at each path.

Figure 3.12 illustrates a histogram of endpoint slack at all latches. Endpoint slack is how far away a path is failing its setup time. When the latches were inserted was each flip-flop swapped by a pair of latches. There is the same amount of masters and slaves prior to retiming. However may retiming affect this ratio depending on area and timing constraints. DigitalFilter post retiming is the ratio almost 1, with 257 negative latches and 267 positive latches. Some area may be saved by only monitoring negative latches, but the gain is minimal.

A more important factor for deciding which phase to monitor, is how many critical paths they include, where the two phases differ a lot. The group of positive latches includes all the most critical paths. Exactly why it is like this is unknown, but a theory that the retiming tools do not change paths within it's time constraint, if not one of it's upstream or downstream paths violates it's timing constraint. A path violating it's timing constraint will be balanced, and some of it's logic will be pushed either to the previous stage or the next stage. Many paths between master-slave pairs are violating the timing constraints prior to retiming, since these paths are synthesized for a flip-flop circuit with a whole clock period constraint. The paths within each pair are almost zero, with no cells between them. With the new constraints, just below a half clock cycle between each latch, force retiming tools to balance the paths. However, the tool does not necessary balance paths to a 50%-50% ratio, but may tend to only move over the logic needed to meet requirements in all paths. This is why positive latches, master latches, are endpoints for more critical paths. The positive latches switched with Razor monitors since these include all critical paths.

Decision of only monitoring the positive latches and not only the critical paths may seem a little odd. By consistently monitor one phase, guarantees monitoring every other latch. This will break up time borrowing paths, which may accumulate to a point of failure if there are multiple unmonitored successive paths.

3.3.4 Applying Bubble Razor to DigitalFilter

DemodFilter is still a latch design at this point, but the design is mapped with all the information required for the Bubble Razor insertion stored in lookup files. The next step is to make scripts to automate the insertion of all the Bubble Razor logic based on these lookups.

No more synthesis was done from this point on. Due to some unexpected behaviour from the synthesis tools, the decision of not running the design through synthesis to keep a good overview of the circuit. Synthesis tools tend to rename signals and remove, add or move cells around which makes it a lot more difficult to debug. In addition, due to the strict timing requirements in the bubble network do we not want the synthesis tool to touch any of the cluster logic. The latch circuit itself is already synthesized and retimed from section 3.2.3. A Bubble Razor circuit can be attracted as two circuits, one bubble network of cluster controls, and the original lathes and datapaths. Only the latch-datapath part is retimed.

3.3.4.1 Script Inserting Bubble System

The script do only work on flatten hierarchy netlists, where the whole circuit is in one module. This is a constrained made when the circuit was retimed. The script goes through a latch design netlist and for each latch looks up in the latch lookup to see if this latch is being replaced by a Razor monitor. If it is not being replaced, it will only switch the clock port input to the clock provided by its new cluster control, also listed in the lookup. Otherwise, if swapped to Razor latch, this is inserted instead of the latch and the clock and error signal is connected. When the script reach end of module and all the latches are treated, the insertion of the cluster control takes place. These are inserted and connected according to the cluster lookup file. The OR-trees for the bubble and error signals is inserted, but only ideal assignments and not cells. This were done for testing purposes, OR-trees made by cells will be discussed in the next section, 3.3.5. At last, all the instantiations for the new signals were added accordingly to the Verilog syntax.

The script seen in appendix E executes the insertion.

3.3.4.2 Input and Output ports

The script inserting monitors and cluster control works only on flat hierarchy netlists. However, it is still possible to use it on individual modules in a hierarchical system. The input and output ports of each module are considered synchronous to one of the phases. Input ports are clocked on the opposite phased clock relative to its first downstream latch, while output ports are clocked on opposite phased clock relative to its first upstream latch. The script adds bubble signals, both in and out, for each port as in figure 3.13. By running each module creating a

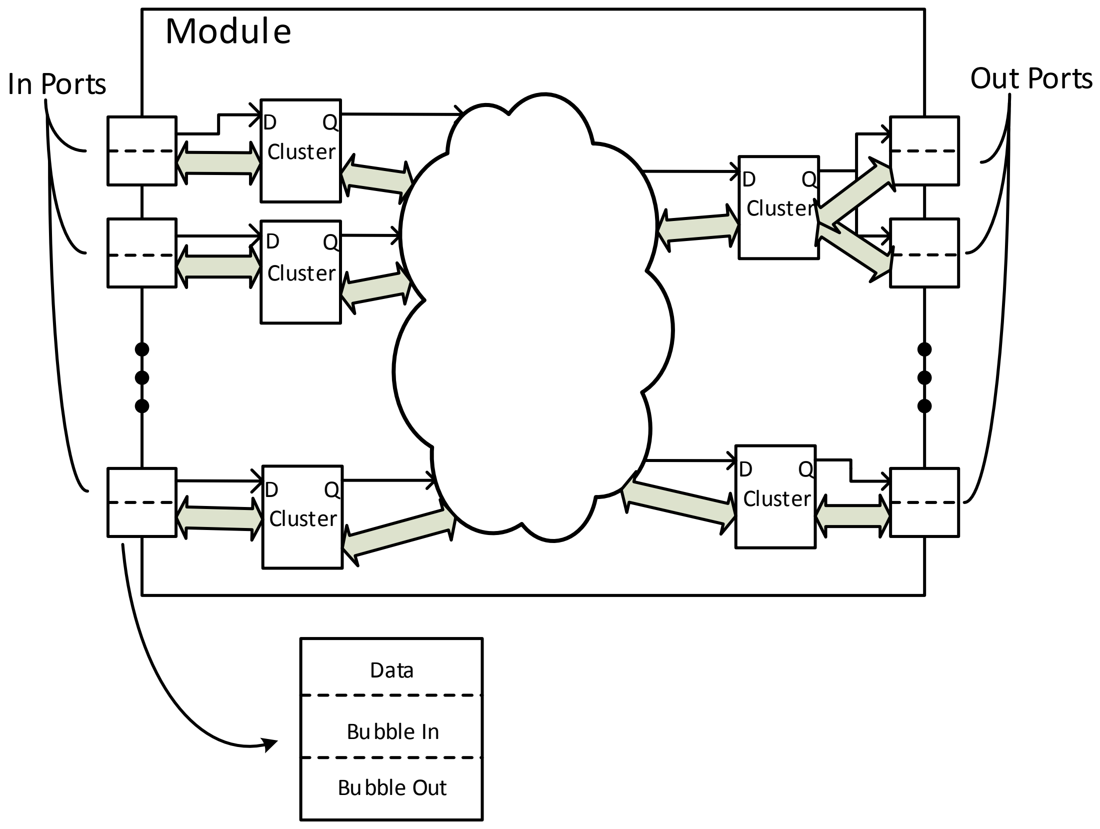


Figure 3.13: Illustration of the port interface.

hierarchy through the script individually, to then connect all the bubbles to/from the ports to other modules' ports or latches/cluster-controls.

3.3.4.3 Testbench Modifications

The testbench needs some minor modifications to test the Bubble Razor version of DigitalFilter. First of all, do it need to insert errors randomly in the circuit and then filter out the bubbles on the input and output ports.

The testbench is ideal, without cell delay, so it is not possible to tune the frequency or voltage to provoke errors. The solution is to probe down in the circuit and force-release random Razor latches' data-pins to hold its value into the speculation window. This will introduce errors randomly in the circuit.

When errors are inserted and the bubble system is working, these will propagate to the in and output ports. A cluster control is added to each port, making the ports themselves stall upon bubbles.

At this point, the testbench did not go through error free. The output is not the same as the reference design, as expected. The reason for this is the clock gate problem, or clock domain problem. This problem is elaborated in section 3.4 and a proposed solution is presented. However, by permanently open the clock gates, practically getting rid of all clock domains and letting all logic being clocked by the two root clocks, the testbench match the reference design, also with all gates opened. Note that the clock gates inside the cluster control is not opened, only the clock gates which were in the original flip-flop design.

3.3.5 OR-trees

The OR-trees introduced by the bubble logic are vital components. It is very important that these signals always reach their destination on time. In addition, the OR-trees generating the *cluster error*-signal affects the speculation window.

[Fojtik et al., 2013] use dynamic OR-gates in a tree structure, which has a speed advantages. Unfortunately are no dynamic gates available in the std. cell library. A decision of not making models of these cells taken made in co-operation with Nordic Semiconductor, due to limited time, but will be added in the future.

The custom XOR-gate, pulling down the error signal when it is not valid, was also not available. This will most likely increase the power consumption to a certain degree. Toggling and glitching will leak through along the error-paths because a normal static XOR-gate always will be open.

Due to the decision of only using available std. cells and the strict delay requirement in the OR-trees, modules of ORs are written in Veliog and synthesized with maximal time constraint for the maximum speculation window.

OR modules made up of std. cells, able to OR from 4-26 inputs, makes up the static OR-trees. The slowest one is the 23-input OR which has a delay of 0.77ns, or 4% clock period under worst-case library. These gates are mostly a tree of NORs, inverters and NAND gates. These cells have a strong output drive, due to the timing requirement.

3.3.6 Clustering

Clustering is an important step to reduce area added by Bubble Razor. DigitalFilter's latches have all their own cluster control, which increase the area by a lot. As mentioned, does the lookup-generating script also perform clustering. When clustering, it outputs a cluster lookup containing all cluster, its member latches and monitors, clock etc.

As mentioned in 2.7.4.1, is there a tradeoff between the size of *Cluster Error* OR-tree and *Bubble In* OR-tree. Bigger clusters give fewer neighbours, hence smaller *Bubble In* OR-trees. However, bigger clusters mean more Razor monitors per cluster and larger OR-trees for *Cluster Error* generation.

Figure 3.14 illustrates the main steps when clustering. The algorithm starts on graphs like the ones in figure 3.11. It selects the two clusters with most neighbours in common, and then check if these two clusters merged fulfils the OR-tree size constraints. If a pair of clusters do not meet the constraints, they will be added to a ban-list and will be ignored until some other cluster pair is merged. Otherwise, if fulfilling the constraints, they are merged together and the positive and negative graphs are updated. The merging is finished when there are no legal cluster pairs left.

There is only a constraint for how many monitors a cluster is allowed to contain. Due to the size of DigitalFilter, is there no need to constrain number of neighbours. The problem with this algorithm is that it cannot escape a local minimum if the number of neighbours is constrained, which will be a problem in larger designs. A

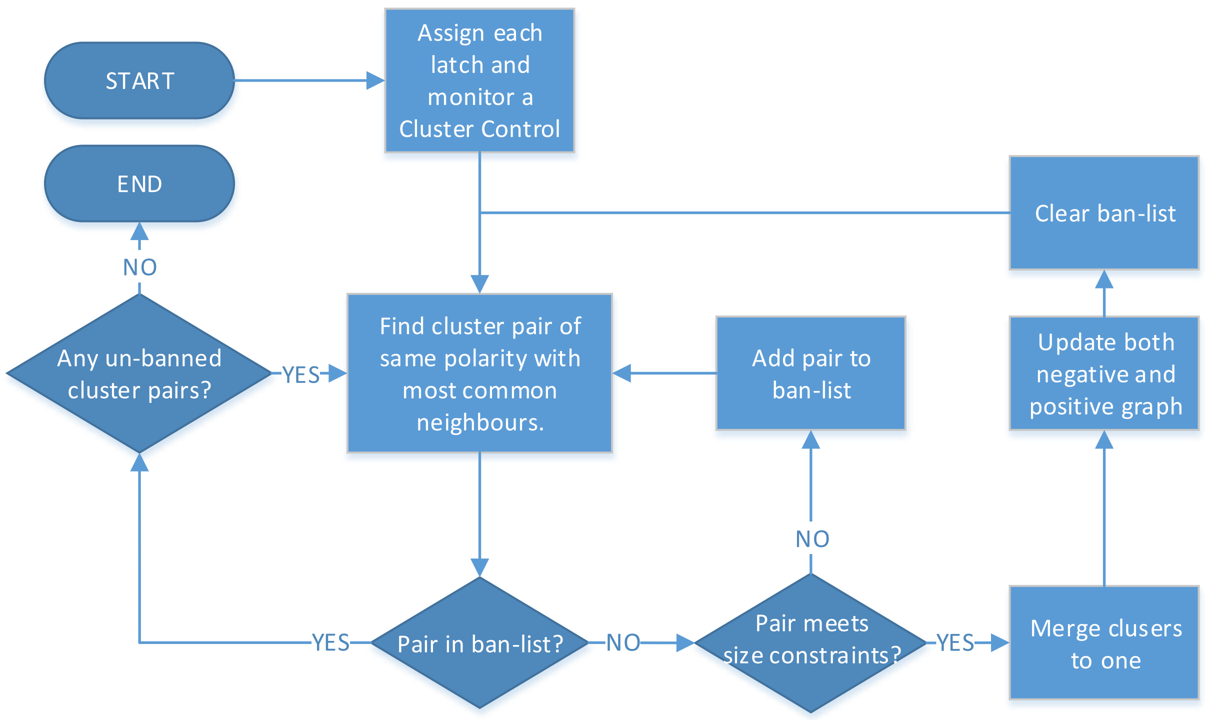

START
Assign each latch and monitor a Cluster Control
END
NO
Clear ban-list
Any un-banned cluster pairs?
YES
Find cluster pair of same polarity with most common neighbours.
Add pair to ban-list
Update both negative and positive graph
NO
YES
Pair in ban-list?
NO
Pair meets size constraints?
YES
Merge clusers to one

FIGURE 3.14: Illustration of how the clustering is done.

hypergraph partitioning tool will most likely be the best way to cluster latches in designs above a certain size.

The number of clusters is reduced from 524 to 19 clusters. The best result were achieved with constrain on number of monitors per cluster of 16. The largest *Bubble In* OR-tree is then a 26 input, which is the only larger one. The second largest has 5 inputs. Number of components added by Bubble Razor is found in appendix G.

3.4 Clock Gate Problem

All the steps in section 3.3 will work in designs with only one clock domain. A clock domain is here defined as groups of registers clocked by the same clock. The output of two clock gates are defined as different clock domains. However, the clock gate inside cluster controls does not define a new clock domain. In the context of latch design, are both the two phases in a two phased clock defined as one clock.

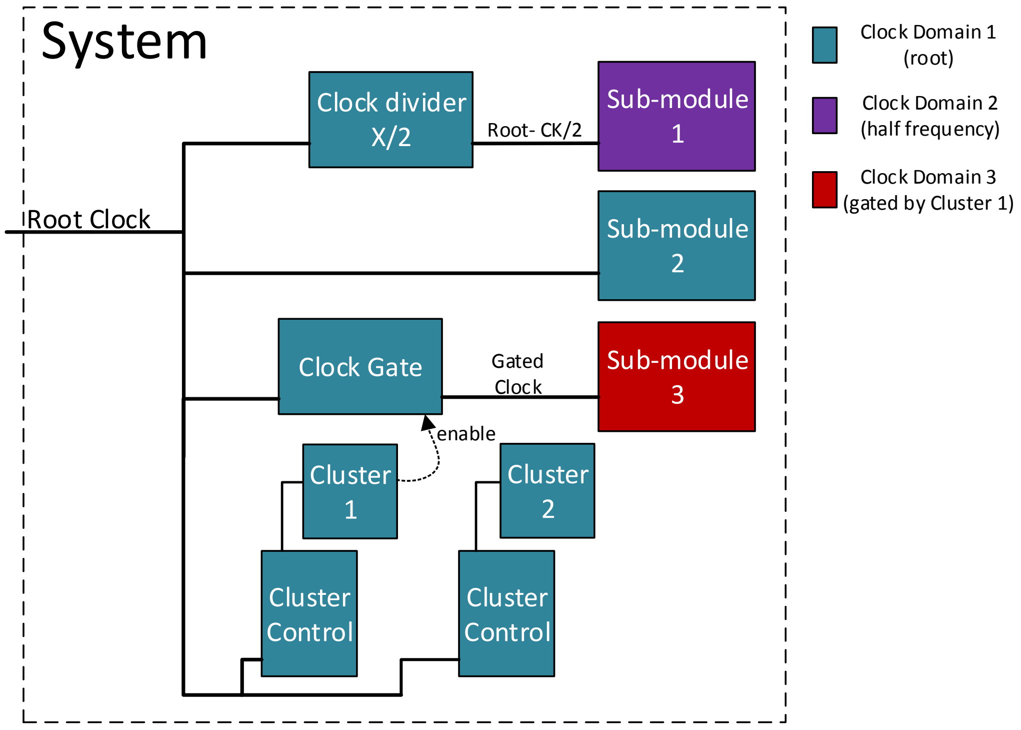


FIGURE 3.15: Illustration of different clock domains.

Figure 3.15 illustrates what is defined as a clock domain and what is not. The clock signal includes both phases. Note that all the different clocks are synchronous. Asynchronous clock domain crossings are a totally different problem.

As far as we know, is Bubble Razor not been applied to designs with a more complicated clock scheme. Why not remove clock gates? If Bubble Razor should work on ”any” design, this problem must be solved. Clock gates are widely used and is a way to reduce power. Many of the design which already are made will often include clock gates, either only to turn on and off sections of a chip or more complicated where it is used to control the behaviour of the circuit. One of the gated domains in Digital filter does the clock itself control the sampling rate, and cannot be removed without a larger modification.

3.4.1 Problem Description

The problem with clock domain crossings is how the bubble signal is treated in the domain with the slowest clock. Normal cluster controls will give an error time to recover, but will not stall the clock itself. This will lead to incorrect behaviour.

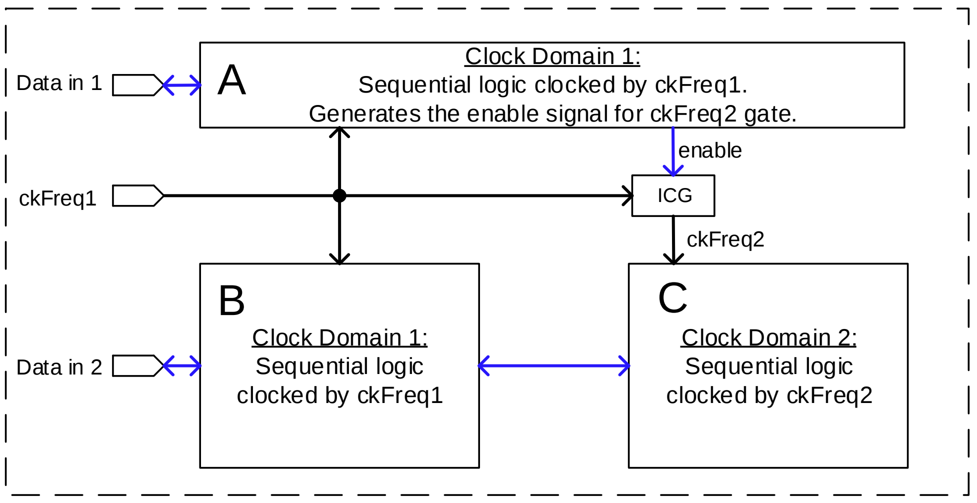
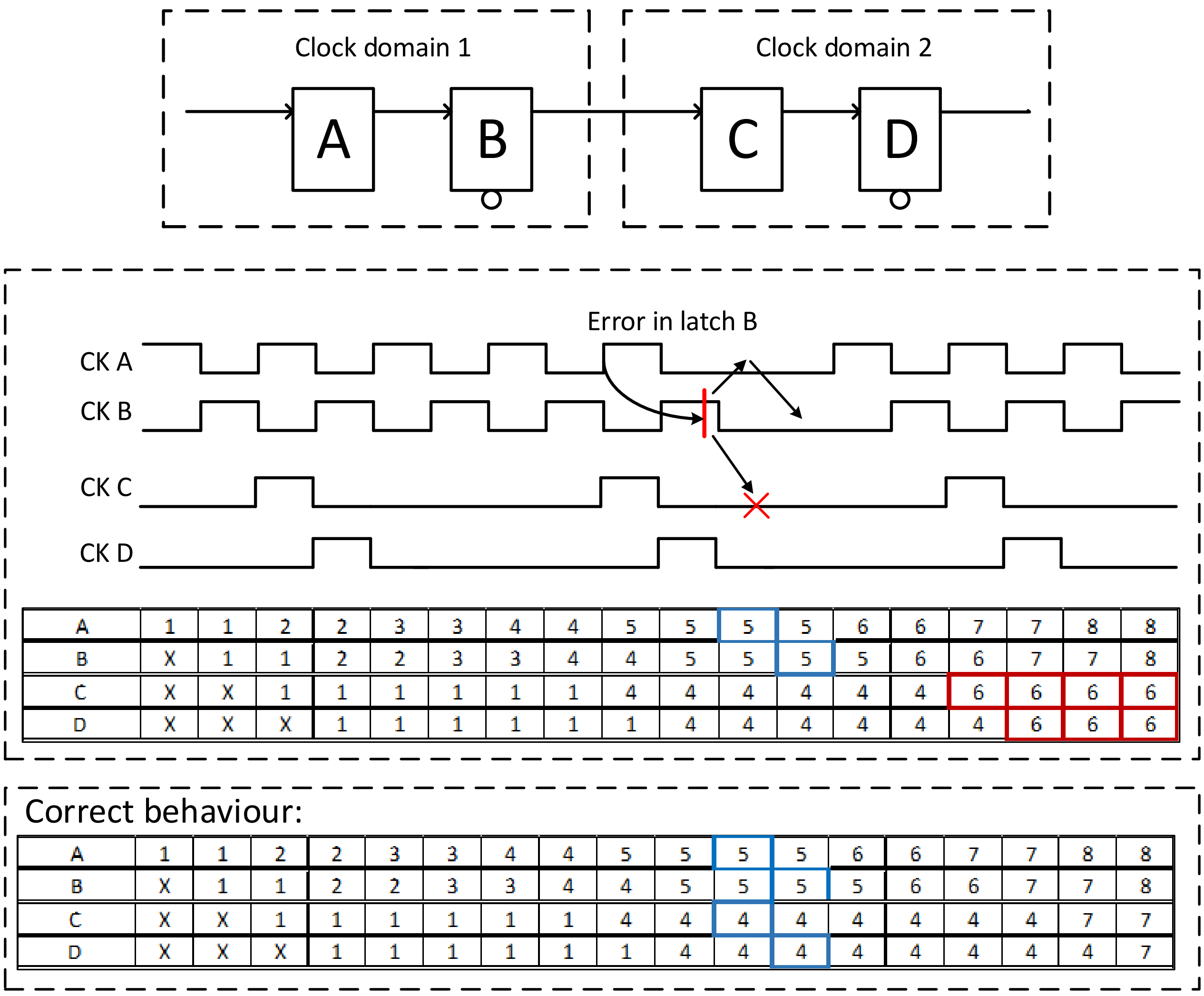


FIGURE 3.16: Illustration of the clock domains in DigitalFilter with its two different clocks, ckFreq1 and ckFreq2.

Figure 3.16 illustrate how the clock is routed in DigitalFilter. The module at top, A, is a control module for the clock *ckFreq2*. *ckFreq2* is a downscaled version of *ckFreq1*, where the frequency is controlled by module A. Although *ckFreq2* is a periodic clock with a controlled period, the problem is the same in modules clocked by pulse picking with no fixed period. There is no data paths between module A and the two others, so no bubbles will bubble between them and A.

Figure 3.17 shows how the error occur. A, B, C and D are four latches with their own cluster controls. Latch A and B are located in clock domain 1, C and D in domain 2. Throughout will these domains be referred to ”fast clock” and ”slow clock”. Technically is ”slow clock” the clock **pair** in the slowest clock domain containing both phases, and the same for fast clock.

When an error occurs in domain 1 and a bubble passes to domain 2, the cluster control in latch C prevent latch C to clock data. However, this makes no difference



A	1	1	2	2	3	3	4	4	5	5	5	5	6	6	7	7	8	8
B	X	1	1	2	2	3	3	4	4	5	5	5	5	6	6	7	7	8
C	X	X	1	1	1	1	1	1	4	4	4	4	4	4	6	6	6	6
D	X	X	X	1	1	1	1	1	1	4	4	4	4	4	4	6	6	6

Correct behaviour:

A	1	1	2	2	3	3	4	4	5	5	5	5	6	6	7	7	8	8
B	X	1	1	2	2	3	3	4	4	5	5	5	5	6	6	7	7	8
C	X	X	1	1	1	1	1	1	4	4	4	4	4	4	4	4	7	7
D	X	X	X	1	1	1	1	1	1	4	4	4	4	4	4	4	4	7

FIGURE 3.17: Illustration of how an error occur between clock domains. (Active high)

on the latches in domain 2. The slow clock will keep ticking in the same pace as before independent of errors in either domains. To illustrate this do the two tables show the instruction stored in each latch as the clock is ticking. Blue cells are stalling latches and red cells are incorrect behaviour. An even worst behaviour occur when a bubble reaches domain 2 at a positive edge on latch C's clock. The cluster controls will then stall both C and then D and cancel the pulses, which result in no new clock pulse until next scheduled edge. If it happens in this example to the middle pair of slow clocks, latch C will not latch any instruction between the first and last pulse in the figure. The slow clock itself needs to stall, as in the bottom table, for the correct behaviour. This complicated things a lot. The slow clock is global to all the latches inside its clock domain, so it is not possible to stall it along the datapath with the bubble algorithm. Instead, there need to exist a dependency between clock domain 2 and *ckFreq2's* control module.

The same things are true if an error occurs inside the Clock Control Logic itself. When a bubble reaches the latches controlling *Enable* and *Enable* stalls at a logic one, this will result in a double clock pulse on both phases in *CkFreq2*. *CkFreq2* will also be delayed by one cycle relative to module B.

Bubble Razor is based on propagating stalls along the data dependencies, but what happens in the case where one module is controlling another through its clock? The idea is that a gated clock contains information about the clock gate's enable signal. If so, and a gate clock could be seen as a data carrying signal, some bubble exchange must take place between these modules at the clock gate.

3.4.2 Proposed solution

The clock gate problem is not insignificant and, as far as we can see, introduce a whole set of constraints. The possible circuitry presented here is an extension and our contribution to Bubble Razor. The point is to give an idea of how the solution will affect the circuit that gets Bubble Razor. Two different Bubble Clock Gates are presented in section, one in 3.4.2.1 and one in 3.4.2.2.

Rules for domain crossings:

1. Every latch along a data path need to stall the same amount of time independent of its clock period.

2. The amount of time each latch stalls must be based on the fast clock: a stall takes one fast clock cycle. Throughput penalty will increase if it is based on the slow clock. This will especially be a problem where the time between two pulses on the slow clock is very long. To work independent of the slow clock period, all bubbles and cluster controls must be clocked by the fastest clock.

3. All the latches in the slow domain is clocked by a gated clock, thus do all latches have a data dependency to the clock gate. A stall on the slow clock itself is a stall on all the latches in the domain, making the domain itself act like a cluster.

4. When a bubble reaches the slow domain, or an error is issued from inside the domain itself, exactly before a pulse on the slow clock, do this pulse need to be cancelled and postponed to a fast clock cycle later. As shown in figure 3.20.

5. A bubble to the clock gate must propagate to the circuitry controlling the enable signal, delaying the next enable.

Constraint for clock gated domains:

1. The amount of monitors inside the slow domain is limited. Since the whole slow domain itself acts as a cluster, all *Cluster Error*-signals from all monitors must be OR'ed together. One OR-gate for each phase, which cannot be larger than the delay limit for the desired speculation window.

2. The amount of neighbouring clusters to the domain is limited. Every neighbouring cluster need to connect their *BubbleOut* signal to the clock gate: one OR-gate for upstream clusters and one OR-gate for downstream clusters.

3. If constraint 1. and 2. cannot be met, and the domain need to be split to two or more slow domains, each new slow domain need their own copy of the Clock Gate Control circuit, or a copy of part of the Clock Gate Control circuit. This may increase the area a lot, depending on the number of new domains and the size of the Clock Gate Control circuit.

Figure 3.18 shows a new component introduced to the circuit from figure 3.16, Bubble ICG. This component is a special cluster control, or more correctly domain

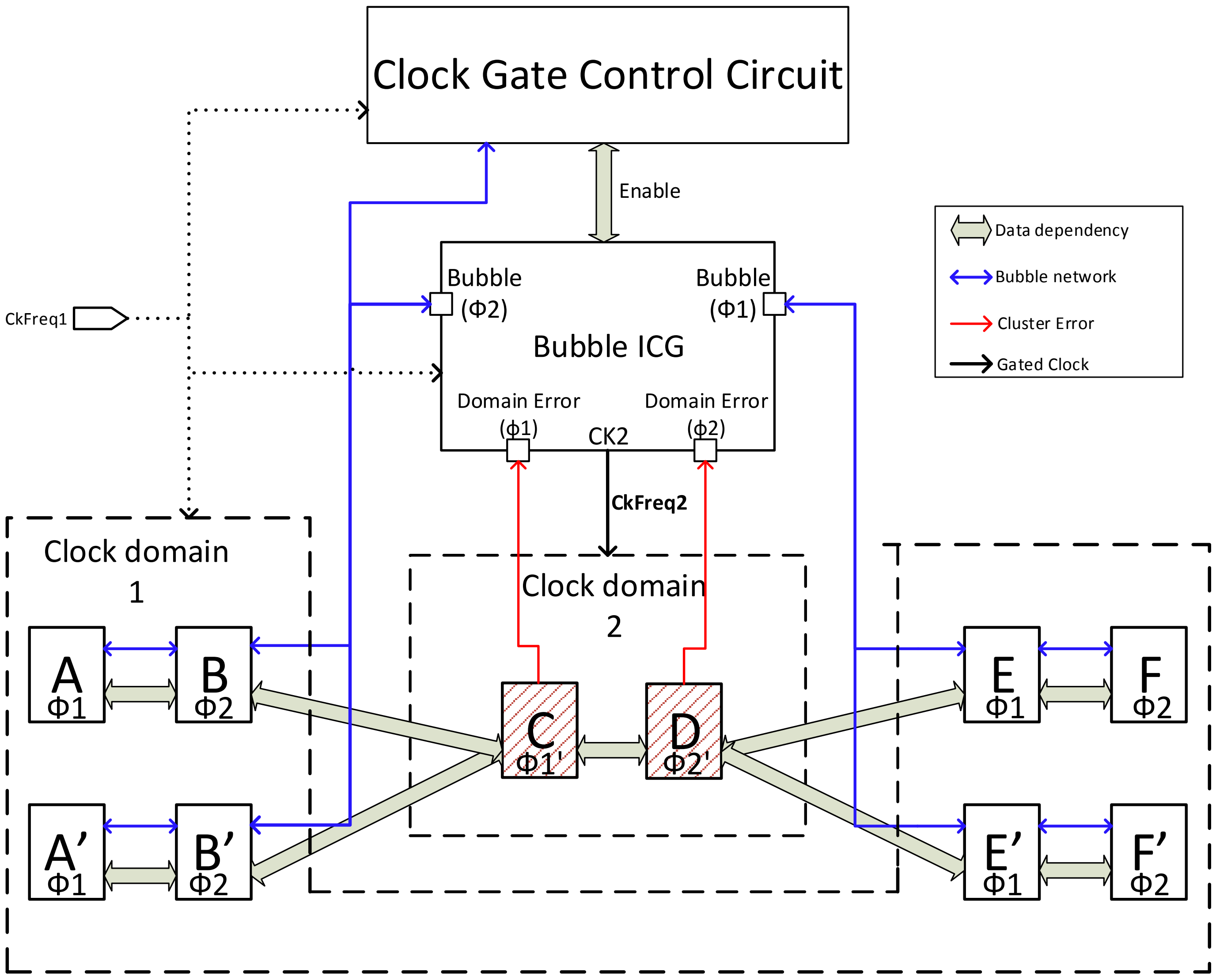


FIGURE 3.18: Illustration of DigitalFilter’s clock domains with a Bubble Clock Gate.

control. Bubble ICG works as a cluster control for a cluster containing both positive and negative latches by controlling both phases. It also does the task of gating *CkFreq2*, which means that this special cluster control get a datapath signal, *Enable*, and need a bubble signal to Clock Gate Control Circuit. Thick grey arrows are the datapaths between latches, blue connections are bubble signals and red are cluster error signals. A, A’, B, B’, E, E’, F and F’ are clusters in clock domain 1 with a cluster control each. While the patterned C and D are latches or monitors without cluster control. As mentioned is Bubble ICG their cluster control and therefore is C' and D's *Cluster Error* connected to Bubble ICG's Domain Error ports. Bubble ICG got a pair of BubbleIn, BubbleOut and Domain Error ports, one for each phase, because Bubble ICG boarders to each phase in the bubble network.

3.4.2.1 Bubble ICG: Equal Duty Cycle Version

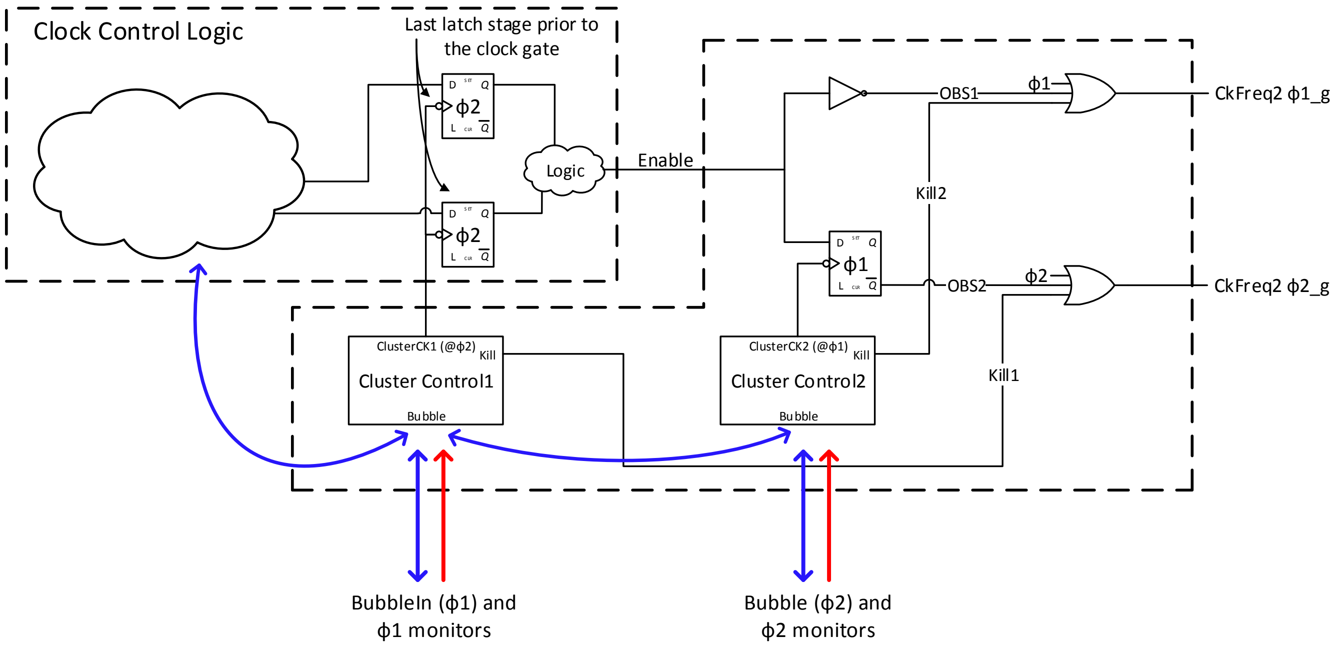


FIGURE 3.19: Bubble ICG, the clock domain bubble control. This version is for active low clocks. Blue arrows are bubble signals, and red are *Cluster error*-signal from the slow clock domain.

Figure 3.19 shows the Bubble ICG. It is based on the two-phased clock gate from section 3.2.2. However, it now includes two cluster controls, which is the standard cluster control with active low latches, from figure 3.7. These clusters provide clocks to the latches in the clock gate. Note that the phase two latches, $\phi 2$, is actually the last stage in Clock Control Logic. This means that Cluster Control Logic must get a clock, from Cluster Control 1 for these bordering latches.

The latches are clocked by cluster controls to delay *CkFreq2* and *Enable* signal upon an incoming bubble or error signal. If the clock gate is stalled when a

CkFreq2 pulse is being issued, the *OBS* signal will be frozen high one cycle extra, which will lead to two pulses in succession on *CkFreq2*. To prevent this, is the internal *Kill* signal from each cluster control cross connected and sent to the OR-gates. This may not be the best method to pull the *CkFreq2*-clocks down. An alternative method is to use *ClusterCK2* and *OBS2* to generate *CkFreq2* $\phi 2\_g$, and *ClusterCK1* and *OBS1* for *CkFreq2* $\phi 1\_g$. This will cause the same behaviour, but with one less input on the OR-gates. This is illustrated in the unequal duty cycle version in figure 3.21.

Bubble ICG's behaviour is shown in figure 3.20. Tiles A and C illustrates when a bubble from either downstream neighbours of the slow domain, clocked by $\phi 1$, monitors inside the slow domain, clocked by $\phi 1$, or a bubble from Clock Control Logic reaches Bubble ICG. This will cause $\phi 2\_g$ to stall and then $\phi 1\_g$. Tile B and D shows the opposite, except that a bubble from Clock Control Logic always stalls $\phi 2\_g$ first. If a pulse on either $\phi 1\_g$ or $\phi 2\_g$ is stalled away, it will appear the next cycle. It is important to always keep the order intact, $\phi 1\_g$ always need to appear before $\phi 2\_g$.

Some modern circuits utilize a fine-grained clock gate scheme with a numerous clock gates. Bubble ICG will introduce a lot of area. This will also complicate the clustering. In some cases would it be more beneficial to redesign parts of a design with large gated domains.

3.4.2.2 Bubble ICG: Unequal Duty Cycle Version

It is possible to only gate $\phi 2\_g$ and use a inverted version of this as $\phi 1\_g$. This will reduce the timing constraint for *Enable*, and make Bubble ICG two cells smaller. The downside is that $\phi 1\_g$ now has inverted duty cycle with an active phase dependent on the enable signal. As a consequence, monitors cannot be clocked by $\phi 1\_g$. An error in a monitor clocked with a long active phase will be held until the next edge, causing one error to initiate a new bubble sequence every other fast clock period.

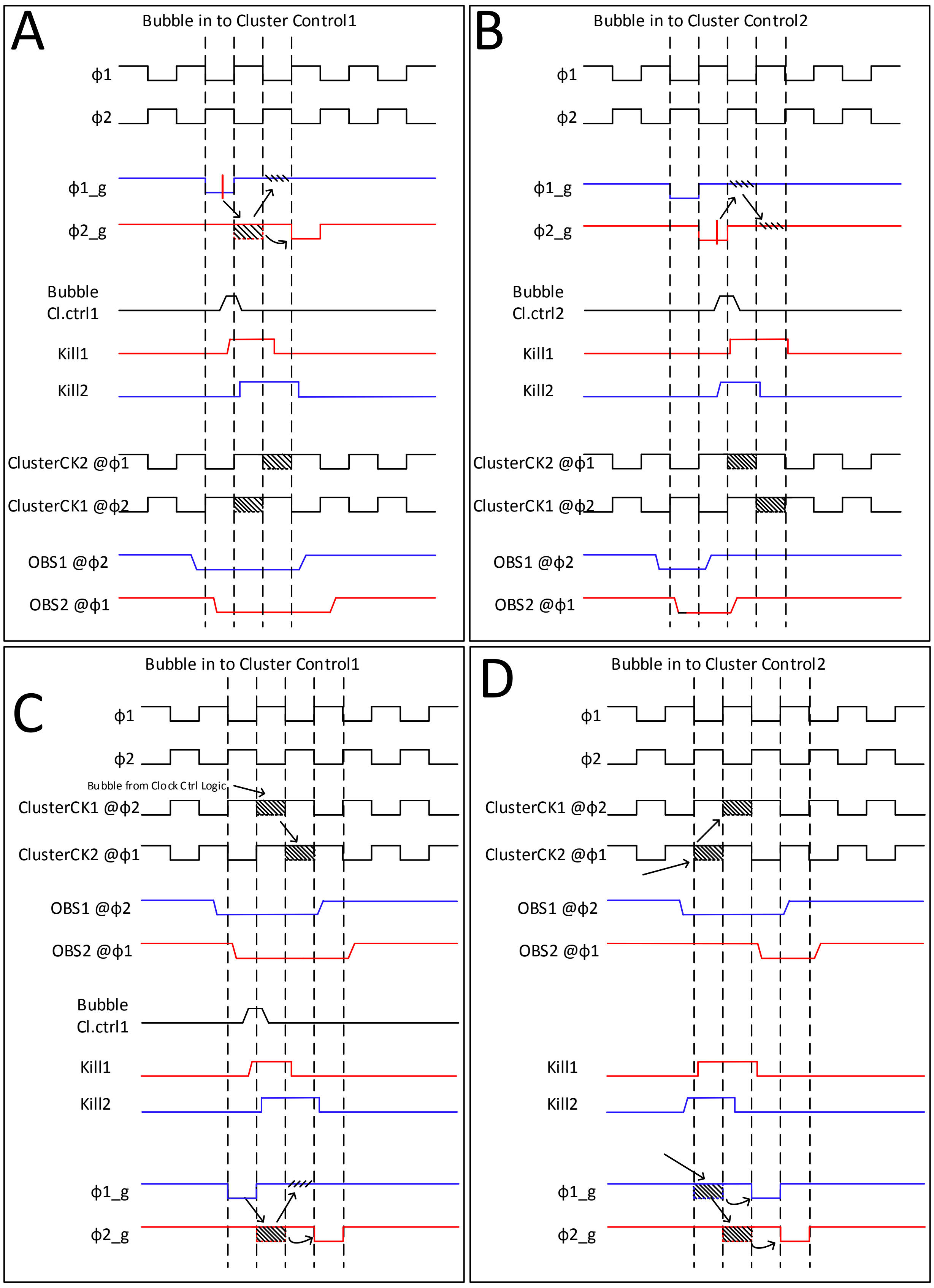


FIGURE 3.20: Wave form of *equal duty cycle* Bubble ICG's behaviour. Hatched areas are stalls, clock is forced high at stalls. $\phi1\_g$ and $\phi2\_g$ are the gated clocks clocking the slow clock domain.

A more complicated constraint is that feedback connections from the first stage in the slow clock domain, clocked by $\phi1\_g$, to the last stage in the neighboring upstream domain, clocked by $\phi2$, are not allowed. This is due to the violation of the non-overlap requirement done by the $\phi1\_g$ latches. Luckily, there are no such feedbacks prior to retiming of the latch circuit, since both nodes are inside a master-slave pair and can be easily avoided under retiming. It is important to ensure that no feedback exists. Oscillation loops will arise if so.

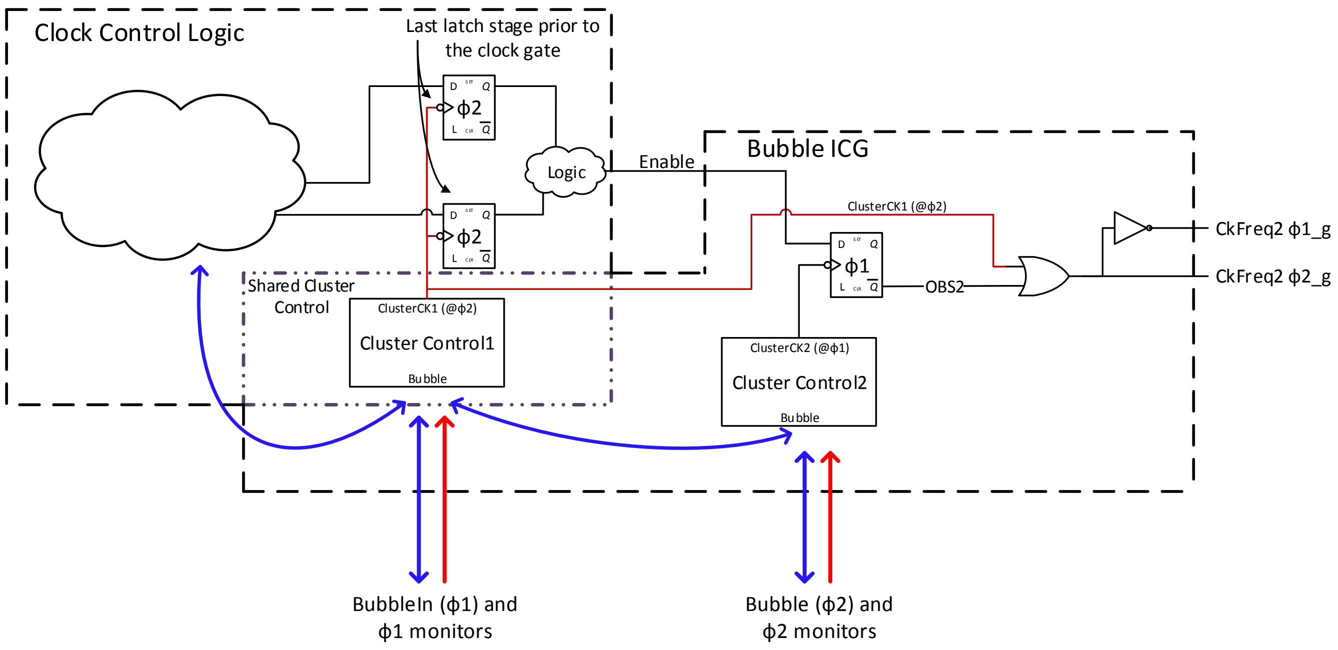


FIGURE 3.21: The alternative Bubble ICG. This removes the strict timing reqirement for *Enable*, but $\phi1\_g$ get an inverted duty cycle.

Figure 3.21 shows the alternative Bubble ICG. This also shows the alternative method of generating the stall on $\phi2\_g$ by gating the cluster clock, shown in red, instead of using the internal *Kill*-signal. *Cluster Control1* is still shared by the clock control circuit and the Bubble ICG. This version is preferred due to the less strict timing constraint, but cannot clock monitors on $\phi1\_g$, only on $\phi2\_g$, which is normally not a problem.

The behaviour is similar to the equal duty cycle version, shown in 3.20, but without a separate gate for $\phi1\_g$. The unequal duty cycle version also fulfils the requirement of getting the first pulse following a high *Enable*.

This versions connection is the same as the equal duty cycle version, shown in figure 3.18.

3.4.3 How to Handle the Error Signal

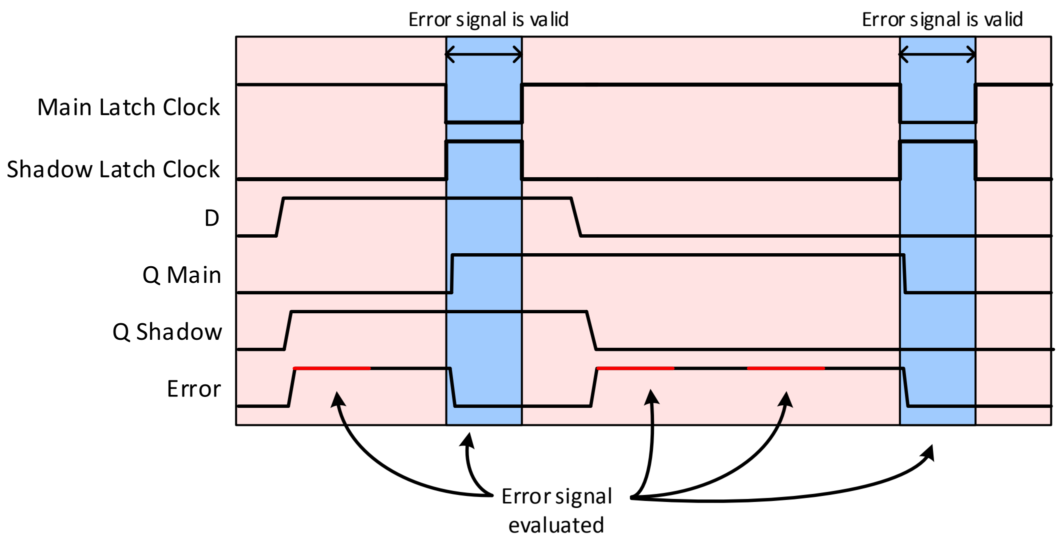


FIGURE 3.22: Illustration of problems regarding standard XOR-gate in an Razor latch, error signal is only a true error in blue areas. (Active low)

Another important element when Bubble Razor is applied to designs with multiple clock domains is the pull-down XOR-gate from figure 2.14. As mentioned, must all bubble and error signals have the resolution of the fastest clock in the system. Unfortunately, a normal XOR-gate cannot be used since this will let through error signals when the master latch is not active. As described in section 2.7.4, does the pull-down XOR-gate only generate an error signal in the timeslot when its valid. A Razor latch with a standard XOR will behave as in figure 3.22. Every error issued in the red areas is not true errors. Therefore, clusters and Bubble ICG evaluate these error signals on the fast clock and will then interpret these false errors as real and initiate the bubble process.

3.5 SPICE: Analogue Simulations

Every simulation up to this point has been ideal logic simulation with no timing. To test the actual behaviour of the Bubble Razor version of DigitalFilter and get accurate power estimates, it need to be simulated by an analogue simulator.

The SPICE simulator is Spectre. Both the original flip-flop DigitalFilter and the Bubble Razor DigitalFilter are simulated to compare power-results. Neither of these modules has gone through layout, so there will is no wire capacitance. For the same reason is there no clock tree buffers, the clock trees are ideal. However, as discussed later, is the clock tree in the flip-flop design much less complex since it is ideal all the way to most of the latche's clock pins, except the ones clocked by a clock gate. While all latches and monitors in the Bubble Razors are driven by the cluster controls.

The first step to simulate a Verilog-netlist in a SPICE simulator is obviously to convert the Verilog-netlist to a SPICE-netlist. This was done by using a small Verilog-SPICE converter. Power-pins are added to each cell, since these are not used in the Verilog-netlist.

Model files of both transistors and cells must be included in the newly generated SPICE-netlists. It is important that the Verilog-netlists only contain cells that are described in the model files, and no non-synthesizable syntax.

The input stimuli are generated from the stimuli vectors of the original Verilog-testbench. The clock setup made in section 3.2.2 gives us full control of duty-cycle, frequency and non-overlap. Since we chose to generate a clock tree for each phase and rout them out to two clock pins, instead of locally generating each phase from one three.

All simulations are ran at typical conditions: room temperature, typical-typical transistor models. The original flip-flop design is simulated for a reference, but it is only simulated at 1.2V and 1.4V, where 1.2 is the lower voltage limit for error-free operation. On the other hand, the Bubble Razor version of has no pre-set lower voltage boundary to prevent setup errors, only a boundary to prevent the paths exceeding the speculation window. The voltage in the Bubble Razor version is scaled down over a number of simulations to test it to its limit. The voltage could also be scaled in one long simulation, but more control were achieved by simulating each voltage step as one smaller simulation, in addition to reduce the turnover time in case of a bug.

The size of DemodFilter is relatively big for an analogue simulation, which results in long simulation time. More clock cycles will give a better representative estimation of the dynamic power. 42 clock cycles were chosen for simulation length. Simulation time took from about 30 minutes to above two hours for every voltage step, depending on how many CPU-cores the simulations utilize.

As explained in the earlier sections is the clock domain problem solution not implemented in DemodFilter due to the lack of some std.cells. Therefore do, like in the logic simulations of Bubble Razor DigitalFilter, all clock domains run at the fast clock. This is done by setting all clock gates controlling the domains to always open, merging all clock domains to one. This is done both in the flip-flop and the Bubble version.

Clock Frequency	52MHz
Voltage Range	0.9 - 1.4V
Simulation Length	42 ck cycles
Technology	180nm

Chapter 4

Power Results

The result from the power analysis was not as satisfying as we hoped for. An explanation for the result is discussed in the next chapter, Discussion.

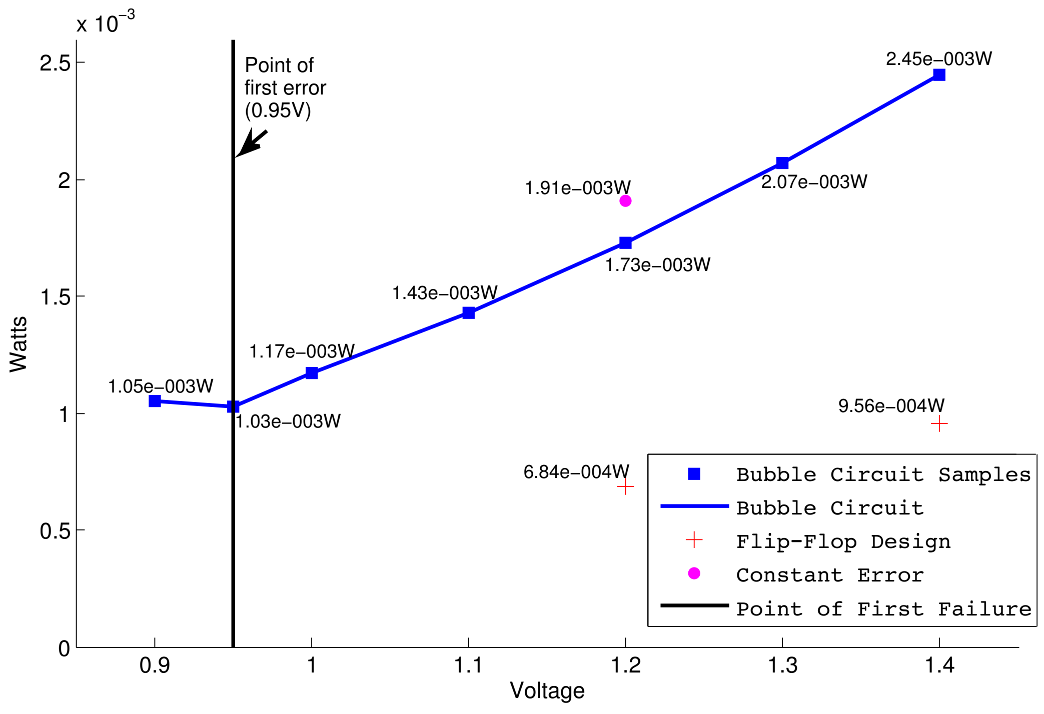


FIGURE 4.1: Plot of power consumption in flip-flop version and Bubble Razor version of DigitalFilter. Both design run at 52MHz.

Figure 4.1 shows how the power is scaled in DigitalFilter. The blue graph shows power consumption in the Bubble Razor version of DigitalFilter. The vertical line shows when the first error is detected by the Bubble logic, where one error was issued over the 42 clock cycles. At the next simulation at 0.9V the number of error increased to 7. The error detection worked perfectly, and recovered each instruction. The magenta circle is a simulation of Bubble Razor version with errors every cycle, and still operated perfectly. This were done by setting one bubble signal permanently high. By comparing the error-free and always error simulation at 1.2V we get an image of the power cost for error recovery, that is 10% at 1.2V.

Red cross are the original flip-flop design, this design is only guaranteed to operate error free down to 1.2V, therefore is the voltage not scaled further in this design. While the Bubble Razor does not have a lower limit, is the practical limit about 0.95V without affecting throughput. Indeed, at 1.2V the flip-flop design uses much less power than the Bubble Razor version due to the extra logic in the Bubble Razor version. However, the Bubble Razor version should have been more efficient at its optimal point, near PoFF, compared to the flip-flop design at 1.2V. There are explanations for why the Bubble Razor version does not perform as well in these simulations, discussed in the next chapter.

Chapter 5

Discussion

General Discussion and Reflection

The conversion from a flip-flop design to a Bubble Razor design is highly automatable. It is important that the current design-flow for designers is kept, as it is, where the insertion of Bubble Razor is a separate step. Module designers must be able to design modules without thinking about bubbles and latch design. From the experience of this study, the best place to implement the latch design and bubble design is after synthesis and before layout. This enables module designers to think of the circuit as a normal sequential flip-flop design. However, designers should be aware the clock domain problem, and try to avoid large clock domains and think through how the clock gates will affect the cluster logic.

The next step after Bubble Razor has been applied to a design, layout, is affected more than the steps prior to the insertion. Latch circuits have other timing requirements and the clock network is different, but the principle is the same. However, many of the backend steps are highly automated, and all the backend scripts and setups must be redone to fit latches and cluster circuits. Many of the steps in the current design-flow are automated by scripts and setups made by different engineers over many years, and multiple these must be redone or modified to work with Bubble Razor.

Retiming tools will affect the characteristics of the DVFS system. More balanced path, fewer near zero slack paths, will enable the system to scale further. The decision for the "dual error injection" version of the algorithm was based on the fact that every path in the system must be attracted as critical under low voltage condition. This means that the first downstream path of a failing path must be regarded as a possible critical path. In hindsight, this is not the case for DigitalFilter, although it is the safest option. The slack distribution in figure 3.12 says that only one of the phases is critical, meaning that DigitalFilter could have been implemented with the "optional first stall", which would have given a larger speculation window. On the other hand, the amount of voltage reduction is limited due to the uneven the slack distribution. A more even distribution between the two polarities would increase the amount of slack on the most critical path in exchange for some area overhead. Which means that "dual error injection" must be used. On a general basis, where the slack distribution is more even, as it should be, will "dual error injection" be the only option guaranteeing error free operation.

As mentioned is DFT, or design-for-test, a drawback with latch designs. Stuck-at testing is done very similar to normal sequential flip-flop design, but delay testing with scan chains is more difficult due to time-borrowing. However, the latch circuit in a Bubble Razor design is design with no time-borrowing at normal operation, which means that all paths should have positive slack at slow corner. Time-borrowing is considered as an error and will initialize data recovery. The whole point of Bubble Razor is to guard each path and make sure that the delay is small enough, which makes delay testing unnecessary and DFT not an issue for a Bubble Razor circuit.

The insertion of Razor latches and Cluster Controls are highly automated. The most challenging parts are the mapping of the design and the clustering. Mapping by using a STA-tool was rather effective in the way that all the information needed for signal routing and clustering is obtained. Although STA-tools is very useful for the task, a custom netlist parser may be used instead. A parser is able to get all the latch-to-latch routing information at a small amount of time, but getting setup slack will be more complicated.

There are different ways of handle the hierarchical structure of a design when inserting Bubble Razor. DigitalFilter, as a module, was initially a module with some sub-modules. Its hierarchy was flatten, which makes sense since it makes it easier to rout signals and it is a lot easier to cluster latches at the same hierarchical level. When it comes to the whole system, containing all modules at the same hierarchical level as DigitalFilter, it will give a good overview for the layout team if this structure is kept. By doing the process done in this project to all modules at a certain hierarchical level or a certain size and flatten each module's structure, for then connecting bubbles along the data dependencies between each flatten module. It will be easier to pinpoint which module is failing if bugs appear. Modules often come with testbenches that need the interface to be preserved, and will only work if the modules is kept intact.

Clustering where done by an algorithm which is not capable of escaping a local minima. This works fine for smaller modules like DigitalFilter, but a hill-climbing algorithm should be used on larger ones. There exist multiple of clustering tools with a selection of algorithms, which will be further explored in upcoming iterations. The clustering algorithm was not especially quick, and will increase exponentially with the number of latches.

The clock gate problem could introduce troubles, especially in existing design that are converted to a Bubble Razor design. Unlike a circuit designed from scratch where this problem should be taken into account from the start. Some circuits may be practical impossible to convert without a large commitment. Let's say that a chip is divided into two halves: one part is clocked by a gated clock and the other half is a complex state machine controlling the clock gate. If the gated area is too big for one Bubble ICG and need to be divided into two domains, each of these domains need a Bubble ICG leading to a large modification of the state machine. Consequently, increased area and violating the automatable insertion. Domains with a particularly slow clock may not need monitors due to the path capture time, which will enable one Bubble ICG to control a large domain. Not all design will fit Bubble Razor out of the box. Some circuits have strict real-time requirements that do not tolerate even a single stall. It is easy to stall within a chip, but the bubbles will only go so far. Some problems may occur in the interaction with other off chip components that do not support Bubble Razor.

When it comes to the two different versions of Bubble ICG, do we believe the unequal duty cycle version is the better option. By ensuring that the nodes between each master-slave latch are not used in a feedback, oscillation loops will not arise. This will not be a problem in most cases, since these nodes downstream neighbors are always clocked by the non-overlapping second phase and only latches outside the domain are overlapping relative to the masters inside the domain. Therefor is the master's overlapping clock are not believed to cause any problems. The gain of having a less constraint path for *Enable* ensures correct operation under the scaled voltage, which may be a problem

As a safety measure against setup violations, does the Bubble Razor system work perfectly. It alerts if an error occurs and recovers the datapaths on the fly. Error resilient design will be more and more necessary as technology is scaled. Another area where solutions like Bubble Razor probably will be essential in the future is sub-threshold circuits. Sub-threshold circuits are often sensitive to PVT-variations, and the yield would benefit a lot with the error protection of Bubble Razor.

Power results

The power results were not as good as we hoped, but we believe that the results is biased towards the flip-flop version. The main reasons for this are listed below.

Output drive in OR-trees

Due to a lack of large input dynamic OR-gates, static OR-gates trees of high output drive were synthesized to secure trees with low delay. However, these high drive gates will consume more power, especially in the combination of the next point. It must be said that dynamic gates are not known for power efficiently, especially when the pull-down network is activated. This will make the gate charge up, to then discharge every clock cycle. On the other hand, the dynamic OR-gates will stay charged all the time if using the XOR-gate proposed by [Fojtik et al., 2013], until a occasionally error is issued.

Unnecessary toggling in the error paths

The Bubble Razor version of DigitalFilter uses a standard static XOR-gate for the error signal generation, making the error trees toggle every time a Razor latch's datapin toggles. Glitches and toggles will propagate through the shadow latch when the error signal is not valid anyway, as showed in figure 3.22. This will cause dynamic power being used for unnecessary activity. The special XOR-gate from figure 2.14 will only evaluate its inputs when it should, and filter out all unnecessary glitches and toggles.

Clock tree

One of the most power consuming parts of a sequential flip-flop design is the clock tree. Experience show that, the power consumption in a clock tree is typically between $\frac{1}{3}$ to $\frac{1}{2}$ of the total power consumption, in 180nm. Both of the simulated designs have an ideal clock tree, but every latch in the Bubble Razor version are driven by a clock gate, unlike the flip-flop version, where most of the latches are driven by the ideal input port. This will unfairly favour the flip-flop design simulation. The latches would be driven by buffers and cells with a complete clock tree.

Technology

The test design provided is in 180nm, where PVT-variations are not a dominating factor when specifying the supply voltage. The voltage margins for PVT-variations in 180nm are small relative to the operation voltage. With newer technologies, these margins will contribute a lot more to the voltage. This is where Bubble Razor will be at its ace by letting any circuit to operate at brink of failure, where a non-error detecting design must operate at a pessimistic voltage to guarantee error-free operation.

Despite the power number, did Bubble Razor enable the circuit to operate at the lowest voltage possible, removing all margins. Hopefully will we be able to synthesize the design with a more modern technology, where the margins contribute more, and minimize the power overhead from the cluster logic.

The static XOR-gates used in the Razor latches have so many disadvantages compared to the pull-down version. First of all is it necessary for Bubble Razor to work with more clock domain, and secondly do it reduce the amount of toggling in the OR-trees. Glitches and toggles will leak through very time there is some activity in a monitored path, which could be close to every clock cycle. We noticed that unnecessarily toggling through the OR-trees happened almost 95% of the clock cycles, and is belived to be the main reason for the power overhead from Bubble Razor.

By no means do we discard Bubble Razor as a power saving method, [Fojtik et al., 2013] has proved the opposite. The next iterations will include both the pull-down XOR-gate and dynamic OR-trees. Dynamic OR-trees will be faster, which means the clusters can grow larger. This will save even more power.

Chapter 6

Conclusion

The methodology of inserting a Bubble Razor system to an existing flip-flop design was shown to be highly automatable. New circuits that are meant to get a Bubble Razor scheme can be design as normal flip-flop circuits, allowing module designers to continue designing circuits as they are used to and not bother about the Bubble Razor circuit. However, Bubble Razor needs to be taken into account when deciding clock domains and inserting clock gates.

As revealed does Bubble Razor suffer from a clock domain crossing problem. A presented solution based on clustering of whole clock domains is presented. A clock domain, including both latch polarities, will behave as one cluster by the use of the presented Bubble ICG, and connects bubbles from both upstream and downstream neighbours as well as the clock controlling circuit. Although, the maximum size of each clock domain is limited by the number of neighbouring clusters and monitors inside the domain. This constrain may cause a redesign of some circuits when converted to Bubble Razor. Clock gates and multiple synchronous clock domains are very often used in circuits, and Bubble ICG or a similar component will enable most of these designs to be converted to Bubble Razor.

Analogue simulations prove that a circuit with Bubble Razor is able to notify about and correct errors when propagation delay is too high in the most critical paths. This enables a voltage control to scale away all voltage margins and let the circuit operate at optimal voltage. The circuit itself behaves just like a sequential flip-flop design, with an occasional one cycle stall if voltage is scaled too far.

Power results in this report favour the normal flip-flop design. However, we strongly believe that with some modifications, the picture will be different. The switching activity in the bubble and error OR-trees will be drastically reduced by filtering out invalid errors. The ideal clock trees do favour the flip-flop design in the power analysis. A fully generated clock tree will lead to a more fair comparison.

If sequential designs are going to work in the future, with even more extreme demands for ultra-low supply voltage, extreme process technologies and high frequencies, the circuits need protection against errors and methods of reducing margins. Bubble Razor and future solutions like it will be a necessity.

6.1 Further Work

- Make the dynamic OR-gates and the pull-down XOR-gate. Insert them and re-run the power estimations.
- Test Bubble ICG in a larger system, and check if the non-overlap violation of the unequal duty cycle affect the retiming.
- Investigate the clock domains for the whole chip, what is the largest one? How many domains is there?
- Take DigitalFilter through layout.

Appendix A

Insertion of Master-Slave Latches

Script for extracting replacing flip-flops with master-slave latches.

```
#!/usr/bin/env perl

use warnings;
use strict;
#use feature qw{switch};;

#----------------------------------------------------
#-------------------- ABOUT -------------------------
#----------------------------------------------------
# This script replace all the flip-flop in a netlist
# with two latches in a master-slave configuration.
# The user must either modify @flip_array, or use the
# input argument -flop to state the cell name of the
# flip-flops. The latch cell is stored in the variable
# $latch_element, may be changed in script or by
# argument -latch.


#---------------KNOWN VARIABLE LIST-----------------
# These variables is changed to whatever your netlist
# requires.You may also use the input argument -flop.

my @flip_array = qw(retracted retracted retracted retracted retracted );
my $latch_element = "retracted"; #Latch with reset, added in 20110526 version

# Look-up strings in the FF declaration:
#INPUT
my $ff_input = "\.D";
#Scan in
```

```perl
my $ff_SI = "\.SI";
#Scan enable
my $ff_SE = "\.SE";
#Clock pin
my $ff_clock = "\.CK";
#Reset Negative
my $ff_RN = "\.RN";
#Data out
my $ff_Q = "\.Q";
#Data out inverted
my $ff_QN = "\.QN";
#----------------------------------------------------

print "Converting from Flip-Flops to Latches...\n";

my $num_args = $#ARGV + 1;
my $in;
my $in_name;
my $out;
my $out_name= "tmp_latch";
my $out2;
my $out_name2;
my $flop_string;
my @flop_array;

my $argv_string = join(" ", @ARGV);

#----------------------------------------------------
#----------------- INPUT ARGUMENTS ------------------
#----------------------------------------------------

if ($argv_string =~ m/.*(\-\-help|\-help)/) {
    print "\n\n\n";
    print "First argument: INPUT netlist\n";
    print "Second argument: OUTPUT netlist\n\n";
    print "OPTIONAL:\n";
    print "-latch\tSet the name of the latch to be used.\n";
    print "-flop\tSet the name of the flip-flops to be changed";
    exit;
}

die "Need in/out files-paths! $num_args\n" if $num_args<2;
if ($argv_string =~ m/^(.+?)\s+/) {
    $in_name = $1;
}
if ($argv_string =~ m/^\Q$in_name\E\s+?(.+)(\s|$)/) {
    $out_name2 = $1;
}
```

```
$in_name = $ENV{PWD}."/".$in_name;
print "INPUT: $in_name\n";
$out_name2 = $ENV{PWD}."/".$1;
print "OUTPUT: $out_name2\n";

if ($argv_string =~ m/\-latch\s(.+)(\s\-|$)/) {
    $latch_element = $1;
    print "Latch: $latch_element\n";
}
if ($argv_string =~ m/(\-flop|\-flip)\s(.+)(\s\-|$)/) {
    $flop_string = $2;
    print "Flip-Flops: $flop_string\n";
    @flop_array = split(/\s/, $flop_string);
    push (@flip_array, @flop_array);
}

#----------------------------------------------------
#-------------------- FILE BLOCK --------------------
#----------------------------------------------------

open($in,  "<",  "$in_name")  or die "Can't open $in_name: $!";
open($out, ">",  "$out_name") or die "Can't open $out_name: $!";
my $test_file;
open($test_file, ">", "out_test_debug") or die "";

#----------------------------------------------------
#-------------------- FUNCTIONS ---------------------
#----------------------------------------------------
# This is the functions that translate the flip-flop
# to two latches.
sub InsertLatch($){ #inputs: (Flip-Flop declaration) returns mid_wire name
    my $instance  = $_[0];
    my $instance_name;
    my $cell_name;
    my $input_node;
    my $SI_node;
    my $SE_node;
    my $clock_node;
    my $output_node;
    my $output_inv_node;
    my $reset_node;

    my $latch_1;
    my $latch_2;
#-------------- Extract Names and Nodes-------------
    if($instance =~ m/^\s*(\w+)\s/){
        $cell_name = $1;
    }else{
        print "WARNING: Could not fine FF's cell name! $instance\n";
    }

    if($instance =~ m/.*?\Q$cell_name\E\s+?(\w+)\s/){
        $instance_name = $1;
        #print "$instance_name\n";
    }else{
        print "WARNING: Could not fine FF's instance name! $instance\n";
    }

    if($instance =~ m/.*?\Q$instance_name\E.+?$ff_input\((.+?)\)/){
        $input_node = $1;
        #print "$input_node\n";
    }else{
        print "WARNING: Could not fine FF's input node! $instance\n";
    }

    if($instance =~ m/$ff_SI\((.*?)\)/){
        $SI_node = $1;
        #print "$SI_node\n";
    }else{
        print "WARNING: Could not fine FF's scan-in node! $instance\n";
    }

    if($instance =~ m/$ff_SE\((.*?)\)/){
        $SE_node = $1;
        #print "$SE_node\n";
    }else{
        print "WARNING: Could not fine FF's scan-enable node! $instance\n";
    }

    if($instance =~ m/$ff_clock\((.+?)\)/){
        $clock_node = $1;
        #print "$clock_node\n";
    }else{
        print "WARNING: Could not fine FF's clock node! $instance\n";
    }

    if($instance =~ m/$ff_Q\((.*?)\)/){
        $output_node = $1;
        #print "$output_node\n";
    }else{
        print "WARNING: Could not fine FF's output node! $instance\n";
    }

    if($instance =~ m/$ff_QN\((.*?)\)/){
        $output_inv_node = $1;
        #print "$output_inv_node\n";
```

```
    }else{
        print "WARNING: Could not fine FF's output inverted node! $instance\n";
    }

    if($instance =~ m/$ff_RN\((.*?)\)/){
        $reset_node = $1;
        #print "$output_inv_node\n";
    }else{
        print "WARNING: Could not fine FF's reset node! $instance\n";
    }
#------------ Extract Names and Nodes End-----------

    $latch_1 = "$latch_element $instance_name" . "_master ( .D($input_node)
  , .Q(" . "mid_" . "$input_node), .QN(), .GN($clock_node)
  , .RN($reset_node) );\n";
    $latch_2 = "$latch_element $instance_name" . "_slave ( .D(" . "mid_"
  . "$input_node), .Q($output_node), .QN($output_inv_node), .GN($clock_node"
  . "_phase2" . "), .RN($reset_node) );\n";

#--------------- Insert Reset-Latch ----------------
    print $test_file "$latch_1$latch_2";
    print $out "$latch_1$latch_2";

    return "mid_" . "$input_node";
#---------------- Insert Latch End -----------------

} #InsertLatch end

sub InsertWires{ #array of mid-nodes
    # declare alle wires needed for the nodes between
    # laches
    my %wire_index;
    foreach (@_){
        if($_ =~ /(.+?)\[(\d+)\]/){
            #Find wire arrays
            if(!$wire_index{$1}){
                $wire_index{$1} = $2;
            }elsif($2 > $wire_index{$1}){
                $wire_index{$1} = $2;
            }
        }else{
            #Single wire
            print $out "wire $_;\n";
        }
    }
    foreach my $i (keys (%wire_index)){
        print $out "wire \[$wire_index{$i}\:0\] $i\;\n";
    }
```

```perl
}

#----------------------------------------------------
#---------------- SWITCHING BLOCK -------------------
#----------------- STATE MACHINE --------------------
#----------------------------------------------------

my $search_string= (join "|", @flip_array);
print "\Q$search_string\E\n\n";
print "$search_string\n\n";

my $buffer = "";
my $state = 'Normal';
my @mid_wires; #Contains all the nodes that need wire declaration
while (my $line = <$in>){

    if($line =~ m/$search_string/){         #Look for Flip-Flop
        $state = 'FFstart';
    }

#------------------- STATE BLOCK -------------------
    #Non Flip-Flop lines
    if ($state eq 'Normal') {
        # declare alle wires needed for the nodes between
        # laches
        if($line =~ /\bendmodule\b/){
            InsertWires(@mid_wires);
            @mid_wires = qw{};
        }
        print $out "$line";

    #Decleration Start
    }elsif ($state eq 'FFstart') {
        $buffer = $line;
        chomp($buffer);

        if($line =~ m/\;/){
            $state = 'Insrt latch';
        }else{
            $state = 'FFrest';
        }

    #Declaration End
    }elsif ($state eq 'FFrest') {
        $buffer = "$buffer" . "$line";

        if($line =~ m/\;/){
            $state = 'Insrt latch';
        }
    }
    #Switch the FF decl. to latch
    if ($state eq 'Insrt latch') {
        chomp($buffer);
        $buffer =~ s/\s+/ /g;
        print $test_file "$buffer\n";
        push(@mid_wires, InsertLatch($buffer));
        $state = 'Normal';
    }
#----------------- END STATE BLOCK -----------------

}

close $in;
close $out;
open($out, "<", "$out_name") or die "Can't open $out_name: $!";
open($out2, ">", "$out_name2") or die "Can't open $out_name2: $!";
print "ack\n";
#Balance wire declarations
my @file_r = <$out>;
my @wire_buffer;
my $line_number = -1;
foreach (@file_r) {
    $line_number ++;
    #get all wire declarations from the module
    if($file_r[$line_number] =~/\b^\s*?module\b/){
        while($file_r[$line_number] !~/\;/){
            print $out2 "$file_r[$line_number]";
            $line_number ++;
        }
        my $tmp_line_number = $line_number;
        while($file_r[$tmp_line_number] !~ /\bendmodule\b/){
            $tmp_line_number ++;

            #wire found
            if($file_r[$tmp_line_number] =~ /^\s*?\bwire\b\s/){
                while($file_r[$tmp_line_number] !~ /\;/){
                    push(@wire_buffer, $file_r[$tmp_line_number]);
                    $tmp_line_number ++;
                }
                push(@wire_buffer, $file_r[$tmp_line_number]);

            #modules declared inside this module is filtered out
            }elsif($file_r[$tmp_line_number] =~/\b^\s*?module\b/){
                my $escape = 1;
                while($escape > 0){
                    $tmp_line_number ++;
```

```
                        if($file_r[$tmp_line_number] =~ /\bendmodule\b/){
                            $escape --;
                        }elsif($file_r[$tmp_line_number] =~ /\^\s*?module\b/){
                            $escape ++;
                        }
                    }
                }
            }
        }
        last unless defined($file_r[$line_number]);

        #Do not include wires twice
        if($file_r[$line_number] =~ /^\s*?\bwire\b\s/ ){
            while($file_r[$line_number] !~ /\;/){
                print "eeeeee: $file_r[$line_number]";
                $line_number ++;
            }
        }else{
            print "$file_r[$line_number]";
            print $out2 "$file_r[$line_number]";
        }

        print "@wire_buffer";
        print $out2 "@wire_buffer";
        splice(@wire_buffer);
        #write out to file..
    }

    close $out;
    close $out2;
    system ("rm $out_name");
```

Appendix B

Clock Routing

Script to setup two-phase clock.

```
#!/usr/bin/env /pri/mako/local/bin/python

import re
import sys
from sys import argv, exit

##-----------------------------------------
##------------ IN/OUT-FILES --------------
##-----------------------------------------
in_file_name = str(sys.argv[1])
tmp_file_name = 'netlist.tmp'
tmp_file2_name = 'netlist2.tmp'
out_file_name= str(sys.argv[2])
##-----------------------------------------

##-----------------------------------------
clock_names = ['ck', 'ck_g', 'retracted', 'ckFreq1', 'ckFreq2']
clock_gates = ['retracted'] #CKgate cell name

##-----------------------------------------
print clock_names

print "IN: ", sys.argv[1]
print "OUT:", sys.argv[2]

#STATES:

module_names = []

```

```
# Route through module declarations
# Route through module instansiations
# Handle clock-gates

# def
def module_dec_case(line):
    #module decleration
    match = re.search(r'^\s*?module\s*?(\w+?)[\s|\(]', line)
    if match:
        module_names.append(match.group(1))
        for clocks in clock_names:
            match = re.search(r'\s*?module.*?[\(|,]\s*?(%s)\s*?[\)|,]'
      % clocks, line)
            if(match):
                line = re.sub(r'([\(|,])\s*?%s\s*?([\)|,])' % clocks,
        r'\1 %s, %s_phase2 \2' % (clocks, clocks), line)
                state = 'wire'
                match = re.search(r'^\s*?module\s+?(.+?)\s', line)

    #input/output dec:
    match = re.search(r'^\s*?(input|output)', line)
    if match:
        for clocks in clock_names:
            line = re.sub(r'(\w\s|,)\s*?%s\s*?(,|\;)' % clocks,
      r'\1 %s, %s_phase2 \2' % (clocks, clocks), line)
    return {'out_line' : line}
# end def

# def
def module_inst_case(line):
    gate = 0
    for module in module_names :
        match = re.search(r'^\s*?%s' % module, line)
        if (match):
            for ck_gate in clock_gates:
                if re.search(r'^\s*?%s' % ck_gate, line):
                    gate = 1

            if gate:
                line = insert_dual_ck_gate(line)
            else:
                for clock in clock_names :
                    match = re.search(r'(\.\w*?)\(\s*?%s[\)|\s]'
          % clock,  line)
                    if match:
                        search_string = match.group(0)
                        search_string1= match.group(1)
                        line = re.sub(r'(\.\w*?)\(\s*?%s[\)|\s]'
               % clock, r'%s, %s_phase2(%s_phase2)'
               % (search_string, search_string1, clock), line)
    return {'out_line' : line}
# end def

def insert_dual_ck_gate(line):
    match = re.search(r'^\s*?\w+?\s+?(\w+?)\s', line)
    switch = match.group(1)
    line_tmp = re.sub(r'%s' % switch, r'%s_phase2' % switch, line)
    for clock in clock_names:
        if re.search(r'\(\s*?%s\s*?\)' % clock, line):
            line_tmp = re.sub(r'\(\s*?%s\s*?\)' % clock, r'(%s_phase2)'
      % clock, line_tmp)

    line = line + line_tmp
    return line


mycase = {
'module_dec' : module_dec_case,
#'oneline_module_inst' : oneline_module_inst_case,
'module_inst' : module_inst_case
}

f = open(in_file_name, 'r')
lines = f.readlines()
tmp = open(tmp_file_name, 'w')

# route two-phase clock through module declarations
current_state = 'module_dec'
for line in lines:
    myfunc = mycase[current_state]
    return_value = myfunc(line)
    if (return_value['out_line']):
        tmp.write(return_value['out_line'])

f.close()
tmp.close()
tmp = open(tmp_file_name, 'r')
tmp2= open(tmp_file2_name, 'w')
lines = tmp.readlines()

# Set all module instansiations on one line:
end = 1
for line in lines:
    for module in module_names :
        match = re.search(r'^\s*?%s' % module, line)
        if match:
                match = re.search(r';', line)
                if match:
                    end = 1
                else:
                    end = 0
    if (not end):
        match = re.search(r';', line)
        if match:
            end = 1
        else:
            end = 0
            line = re.sub(r'([^;])\s*\n', r'\1', line)
    tmp2.write(line)

tmp.close()
tmp2.close()
tmp2 = open(tmp_file2_name, 'r')
lines = tmp2.readlines()
out = open(out_file_name, 'w')

# route two-phase clock through module instansiations
current_state = 'module_inst'
print module_names
for line in lines:
    myfunc = mycase[current_state]
    return_value = myfunc(line)
    if (return_value['out_line']):
        out.write(return_value['out_line'])
```

Appendix C

Design Compiler Wrapper

Wrapper for DesignCompiler. Need two tcl scripts dependent on the module, one for clock reporting and one for time reporting. Both including the string FROM_LATCH, which is the string that is replaced by every latch. Outputs a short merged version of timing and clock report for the whole design.

```perl
#!/usr/bin/env perl

use warnings;
use strict;

#Variables
my $tmp_register_file = "dc_register_list.log";
my $tmp_report_file = "dc_tmp_report.log";
my $tmp_clock_file = "dc_tmp_clock.log";
my $main_report_file = "dc_report.log";
my $warning_log_file = "autoscript_warnings.log";
# Generate a compact list-file of all start-endpoint pairs

#Clock gate cell name
my @post_added_latches = ("retracted",
"retracted",
"retracted",
"retracted");


##FOR EACH LATCH:
## update dc_report: (must load the design)
## report_timing -nworst 2000 -max_paths 100000 -path_type short
## -start_end_pair from FROM_LATCH/Q -to [all_registers -data_pins]
##  put result in a tmp file (./dc_report)
##  append report in the main file...
## END FOR LOOP

#Get list of all registers:
system("dc_shell -f dc_scripts/dc_get_registers.tcl | tee $tmp_register_file");

my $register_file;
open($register_file, "<", "$tmp_register_file")
or die "Can't open $tmp_register_file";

my $warning_log;
open($warning_log, ">", "$warning_log_file")
or die "Can't open $warning_log_file";

my $start = 0;
my $reg_string = 0;
while (my $line = <$register_file>) {
    if($start == 0){
        if($line =~ /^\Qset data [all_registers -clock_pins]\E/){
            $start = 1;
        }
    }else{  # START OF LIST
        if($line =~ /^\{/){        # FIRST LINE
            $line =~ s/\{//;
            $line =~ s/\}//;
            $line =~ s/\n//;
            $reg_string = $line;
        }
    }
}

my @registers = split(" ", $reg_string);
$reg_string = 0;

@registers = (@registers, @post_added_latches);

#End cleaning
close($register_file);
system("rm $tmp_register_file");

sub extract(){
    my $report_file;
    open($report_file, '<', "$tmp_report_file") or die "Can't open $tmp_report
  _file\n";
    while (my $line = <$report_file>){
        if ($line =~ /^s*?warning/i){
            print $warning_log "$line";
        }

        if ($line =~ /(^\s*?Startpoint\:|^\s*?Endpoint\:|^\s*?Path\sGroup\:|
    clocked\sby\s|^\s*?slack|^\s*?time\sborrowed\sfrom)/){
            system("echo \"$line\" >> \Q$main_report_file\E");
        }
    }
    close($report_file);
}

sub extract_clock($){ #$register
    my $report_file;
    my $clock_line = "root";
    my $start = 0;
    open($report_file, '<', "$tmp_clock_file") or die "Can't open
  $tmp_clock_file\n";
    while (my $line = <$report_file>){
        if ($line =~ /^\s*?warning/i){
            print $warning_log "$line";
        }

        if ($line =~ /^The\sShortest\sPath/){
            $start = 1;
        }elsif($start == 1) {
            if ($line =~ /InvertedGate_OrGate/){
                $clock_line = $line;
            }
        }
    }
    system("echo \"clock\($_[0]\)\: $clock_line\" >> \Q$main_report_file\E");
}

# Report for each path

my $num_of_regs = 0;
my $array_size = $#registers + 1;
print $warning_log "Length of register array: $array_size\n";

system("echo \"COMPACT REPORT\" > \Q$main_report_file\E");
foreach my $register (@registers) {
    $num_of_regs ++;
# setup dc_report.tcl with right latch
    print "SETUP dc_report:\n";
    $register =~ /(.+?)\/GN/;
    system("cp dc_scripts/dc_report_template.template
  dc_scripts/dc_report.tcl");
    system("perl -p -i -e 's/\QFROM_LATCH\E/\Q$1\E/g'
  dc_scripts/dc_report.tcl");
```

```
    print "test";
    system("cat dc_scripts/dc_report.tcl");
# report timing to tempfile
    print "Report on latch: $register\n";
    system("dc_shell -f dc_scripts/dc_report.tcl | tee $tmp_report_file");
# extract needed data to main file
    print "Extract and append\n";
    extract();

# setup dc_get_clock_domains.tcl
    system("cp dc_scripts/dc_get_clock_domains_template.template
  dc_scripts/dc_get_clock_domains.tcl");
    system("perl -p -i -e 's/\QFROM_LATCH\E/\Q$register\E/g'
  dc_scripts/dc_get_clock_domains.tcl");
# report clock to tempfile
    system("dc_shell -f dc_scripts/dc_get_clock_domains.tcl | tee \Q$tmp_clock_file\E");
# extract clock to main file
    extract_clock($register);
}

print $warning_log "Reported registesrs: $num_of_regs\n";

## clean up
system("rm $tmp_report_file");
system("rm $tmp_clock_file");
system("rm dc_scripts/dc_report.tcl");
system("rm dc_scripts/dc_get_clock_domains.tcl");
```

Appendix D

Lookup generation and Clustering

Script for latch-lookup and cluster-lookup generation. Takes report from the DesignCompiler Wrapper as input.

```
#!/usr/bin/env /pri/mako/local/bin/python

import re
from sys import argv, exit

##----------------------------------------
##------------ IN/OUT-FILES --------------
##----------------------------------------
in_file_name = 'dc_report.log'
out_file_name = 'latches_clustered_final.loup'
cluster_file_name = 'cluster_clustered_final.loup'
##----------------------------------------

latches = {}
latch_list = []
number_of_latches = 0

cluster_list = []
cluster_hash = {}
number_of_clusters = 0

#----------- MAXIMUM OR SIZE -------------
max_monitor = 16
max_num_of_neighbours = 32
#----------------------------------------

##----------------------------------------
```

```
##---------- Monitor Condition -----------
##----------------------------------------
def IsLatchMonitor(latch):
        if(latch.GetType() == 'slave'):
                latch.monitor = 'True'
##----------------------------------------

class Latch:
  def __init__(self, name, UpStream, DownStream):
    self.latch_name = name
    if re.search("Master", name):
      self.latch_type = "master"
    if re.search("Slave", name):
      self.latch_type = "slave"

    self.latch_clock = ""        ## Relative phase
    self.latch_real_clock = ""   ## Actual clock source
    self.latch_reset = ""
    self.latch_path = ""
    self.latch_type = ""         ## Master or Slave

    self.monitor = False         ## Is this a Razor Latch?

    self.upStream = []
    self.upStream_hash = {}
    self.downStream = []
    self.downStream_hash = {}

    self.slack = -10000000
    self.time_borrowed_frome = ""
    self.time_borrowed = 0

    self.cluster = ""

  def AddUpStream(self, latch_in):
    if (not latch_in in self.upStream_hash):
      self.upStream_hash[latch_in] = 1
      self.upStream.append(latch_in)
  def AddDownStream(self, latch_in):
    if (not latch_in in self.downStream_hash):
      self.downStream_hash[latch_in] = 1
      self.downStream.append(latch_in)
  def AddClock(self, clock):
    self.latch_clock = clock
    if (re.search(r'\'', clock)):
      self.latch_type = "master"
    else:
      self.latch_type = "slave"
  def AddSlack(self, slack):
    self.slack = slack
  def GetNeighbours(self):
    return {'upStream' : self.upStream,
    'downStream' : self.downStream}
  def AddCluster(self, cluster):
    self.cluster = cluster
  def GetCluster(self):
    return self.cluster
  def AddRealClock(self, clock):
    self.latch_real_clock = clock
  def GetType(self) :
    return self.latch_type
  def GetClock(self):
    return {'clock': self.latch_clock,
    'real_clock': self.latch_real_clock}
  def PrintClass(self):
    print "-----------------------------------------------"
    print 'Latch:\t|\t', self.latch_name
    print ' Clock\t|\t', self.latch_clock
    print ' Type\t|\t', self.latch_type
    print ' Up\t|\t', self.upStream
    print ' Down\t|\t', self.downStream
    print "-----------------------------------------------"
  def PrintClass2File(self, file):
    file.write('-----------------------------------------------\n')
    file.write('Latch:\t\t|\t %s\n' % self.latch_name)
    file.write('Clock:\t\t|\t %s\n' % self.latch_clock)
    file.write('Real_Clock:\t|\t%s\n' % self.latch_real_clock)
    file.write('Type:\t\t|\t %s\n' % self.latch_type)
    file.write('Cluster:\t|\t %s\n' % self.cluster)
    file.write('Monitor:\t|\t %s\n' % str(self.monitor))
    file.write('Slack:\t\t|\t %s\n' % self.slack)
    file.write('Up:\t\t|\t %s\n' % self.upStream)
    file.write('Down:\t\t|\t %s\n' % self.downStream)
    file.write('-----------------------------------------------\n')

def print_all_latches():
  for i in latch_list:
    i.PrintClass()

def print_all_latches_to_file():
  out_file = open(out_file_name, 'w')
  out_file.write('--------------Latch-Connection--------------\n')
  out_file.write('TOTAL NUMBER OF LATCHES:\t %d\n' % number_of_latches )
  out_file.write('--------------------------------------------\n')

  out_file.write('-------------- Latch LookUp ----------------\n')
  out_file.write('Latch: \t\t Line\n')

  off_set = 5 + number_of_latches + 3
  line_number = off_set
  for i in latch_list:
    out_file.write('%s :\t\t %d\n' % (i.latch_name, line_number) )
    line_number += 11
  out_file.write('-------------------------------------------\n')

  ## Print all info
  for i in latch_list:
    i.PrintClass2File(out_file)

  out_file.close()

f = open(in_file_name)
lines = f.readlines()

## CASES
## def
def find_new_path_case(line, empty):
  global number_of_latches
  match = re.search(r'Startpoint\:\s+(.+?)[\n|$|]', line)
  if(match):
    #Look if the Startpoint is new
    if (not latches.has_key(match.group(1))):
      latch_list.append(Latch(match.group(1), '', ''))
      latches[match.group(1)] = latch_list[number_of_latches]
      number_of_latches += 1
    match2 = re.search(r'port\sclocked\sby\s(.+?)\)', line)
    if (match2):
      latches[match.group(1)].AddClock(match2.group(1))
      return {'return_string':match.group(1),
      'next_state': 'get_endpoint'}
    else:
      return {'return_string':match.group(1),
      'next_state':'get_start_clock'}

  return {'return_string':None, 'next_state':'find_new_path'}
## end def

## def
def get_endpoint_case(line, startpoint):
  global number_of_latches
  match = re.search(r'Endpoint\:\s+(.+?)[\n|$]', line)
  if(match):
    #Look if the Endpoint is new
    if (not latches.has_key(match.group(1))):
      latch_list.append(Latch(match.group(1), '', ''))
      latches[match.group(1)] = latch_list[number_of_latches]
      number_of_latches += 1

    ## ADD down and up stream on start and end latch
    latches[startpoint].AddDownStream(match.group(1))
    latches[match.group(1)].AddUpStream(startpoint)
    return {'return_string': match.group(1),
    'next_state':'get_slack'}

  return {'return_string':startpoint, 'next_state':'get_endpoint'}
## end def

## def
def get_start_clock_case(line, startpoint) :
  match = re.search(r'clocked\sby\s(.+?)\)', line)
  if(match):
    latches[startpoint].AddClock(match.group(1))
    return {'return_string': startpoint,
    'next_state':'get_endpoint'}
  return {'return_string': startpoint, 'next_state':'get_start_clock'}
## end def

## def
def get_end_clock_case(line, endpoint) :
  match = re.search(r'clocked\sby\s(.+?)\)', line)
  if(match):
    latches[endpoint].AddClock(match.group(1))
    return {'return_string': None, 'next_state':'find_new_path'}
  return {'return_string': endpoint, 'next_state':'get_end_clock'}
## end def

##def
def get_slack_case(line, endpoint) :
  match = re.search(r'slack.+?([\d|-]\d*?\.\d+?)', line)
  if(match):
    latches[endpoint].AddSlack(match.group(1))
    return {'return_string': None, 'next_state':'find_new_path'}
  return {'return_string': endpoint, 'next_state':'get_slack'}
##end def
## END CASES

## CASE lookup
mycase = {
'find_new_path': find_new_path_case,
'get_endpoint' : get_endpoint_case,
'get_start_clock' : get_start_clock_case,
'get_end_clock' : get_end_clock_case,
'get_slack' : get_slack_case
}
## END CASE lookup

def sniff_clock(line) :
  match = re.search(r'^\s*?clock\((.+?)\/GN\):\s*?(.+?)$', line)
  if(match):
    latches[match.group(1)].AddRealClock(match.group(2))
    #latches[match.group(1)].AddRealClock(' root')


## THE LOOP FOR MAKING LATCH CONNECTIVITY GRAPH
current_state = 'find_new_path'
next_state = ''
current_startpoint = ''
for line in lines:
  #print line
  sniff_clock(line)
  myfunc = mycase[current_state]
  return_value = myfunc(line, current_startpoint)
  if (return_value['next_state']):
    current_state = return_value['next_state']
    current_startpoint = return_value['return_string']
## END THE LOOP

## DECIDE IF LATCH SHOULD BE A RAZOR
for latch in latch_list:
  IsLatchMonitor(latch)


class Cluster:
  def __init__(self, type, name):
    self.type = type
    self.name = name

    #clock
    self.clock = ""
    self.real_clock = ""

    #member latches
    self.member_latches = []
    self.member_latches_hash = {}
    self.number_of_members = 0
    self.number_of_monitors = 0

    #oposite polarity neighbours(clusters),
    #this is the neighbours that need bubble from this vice versa
    self.neighbours = []
```

```
        self.neighbours_hash = {}
        self.number_of_neighbours = 0

        #same polarity clusters with common neighbours, no bubble connection,
        #but mergeable
        self.common_neighbours = [] #neighbour same polarity clusters
        self.common_neighbours_hash = {} #cluster as key, number of common
        #neighbours as value

    def AddLatch(self, latch):
        if(not self.member_latches_hash.has_key(latch)):
          self.member_latches.append(latch)
          self.member_latches_hash[latch] = 1
          self.number_of_members += 1

          clock = latch.GetClock()
          self.real_clock = clock['real_clock']
          self.clock = clock['clock']

          if (latch.monitor == 'True') :
              self.number_of_monitors += 1

          retval = latch.GetNeighbours()
          tmp_neighbours = self.neighbours + retval['upStream']
          + retval['downStream']
          for latch1 in tmp_neighbours:
              if(not self.neighbours_hash.has_key(latches[latch1].GetCluster())):
                  self.neighbours.append(latches[latch1].GetCluster())
                  self.neighbours_hash[latches[latch1].GetCluster()] = 1

    def MergeClusters(self, cluster):
        # tell all the neighbours that one cluster is deleted...
        #and remove this from the list...
        # tell all latch members about their new master...
        #Check that the merge do not violate OR-size, if this
        #is violated, we need to mark this cluster
        #so it does not get merged next time.
        if((self.number_of_monitors + cluster.number_of_monitors) > max_monitor):
          return 0

        temp_number_of_neighbours = len(self.neighbours)

        for new_neigh in cluster.neighbours:
          if (not self.neighbours_hash.has_key(new_neigh)):
            temp_number_of_neighbours = temp_number_of_neighbours + 1

        if(temp_number_of_neighbours > max_num_of_neighbours):
          return 0

        #Map neigbours again. Common neighbours must not be counted twice
        #and tell neigh cluster that cluster 2 is no longer.
        for new_neigh in cluster.neighbours:
          if (not self.neighbours_hash.has_key(new_neigh)):
            self.neighbours.append(new_neigh)
            self.neighbours_hash[new_neigh]=1
          #Remove old cluster as neighbour
          cluster_hash[new_neigh].RemoveNeighbour(cluster.name)
          #Add new cluster as neighbour
          cluster_hash[new_neigh].AddNeighbour(self.name)

        #Add the new members and tell about their new master
        for latch_for in cluster.member_latches:
          self.AddMember(latch_for)
          latch_for.AddCluster(self.name)

        #Update cluster_hash and list
        del cluster_hash[cluster.name]
        cluster_list.remove(cluster)

        #Delete old common neighbours, and calculate again for every cluster
        for cluster in cluster_list :
          cluster.MapCommonNeighbours()

        return 1

            def RemoveNeighbour(self, neighbour):
          if(self.neighbours_hash.has_key(neighbour)):
            del self.neighbours_hash[neighbour]
            self.neighbours.remove(neighbour)
            def AddMember(self, member):
          if(not self.member_latches_hash.has_key(member)):
            self.member_latches.append(member)
            self.member_latches_hash[member] = 1
            self.number_of_members += 1
            if(member.monitor == 'True'):
              self.number_of_monitors += 1
            def AddNeighbour(self, neighbour):
          if (not self.neighbours_hash.has_key(neighbour)):
            self.neighbours_hash[neighbour] = 1
            self.neighbours.append(neighbour)
            def MapCommonNeighbours(self) :
          self.common_neighbours = []
          self.common_neighbours_hash = {}
          for cluster in self.neighbours :
            for common in cluster_hash[cluster].neighbours :
              if (not cluster_hash[common].name is self.name) :
                if(not self.common_neighbours_hash.has_key(common)):
                  self.common_neighbours.append(common)
                  self.common_neighbours_hash[common] = 1
                else:
                  self.common_neighbours_hash[common] += 1
            def GetNumberOfNeighbours(self):
          return self.number_of_neighbours



    def printClusters(in_clust):
      if(in_clust):
        print cluster_hash[in_clust].name
        for latch in cluster_hash[in_clust].member_latches:
          print latch.latch_name
        print latch.upStream
        print latch.downStream
        print cluster_hash[in_clust].neighbours
      else:
        for cluster in cluster_list:
          print cluster.name, cluster.member_latches[0].latch_name
          print cluster.neighbours


    def printClusterFile():
      cluster_file = open(cluster_file_name, 'w')
      cluster_file.write('-------------Latch-Connection--------------\n')
      cluster_file.write('TOTAL NUMBER OF CLUSTERS:\t %d\n' % number_of_clusters )
      cluster_file.write('--------------------------------------------\n')

      cluster_file.write('-------------- Latch LookUp ----------------\n')
      cluster_file.write('Cluster: \t\t Line\n')

      off_set = 5 + number_of_clusters + 2
      line_number = off_set
      for i in cluster_list:
        cluster_file.write('%s :\t\t %d\n' % (i.name, line_number) )
        line_number += 6

      ## Print all info
      for i in cluster_list:
        cluster_file.write('--------------------------------------------\n')
        cluster_file.write('Cluster:\t%s\n' % i.name)
        cluster_file.write('Clock: \t\t%s\n' % i.clock)
        cluster_file.write('RealCK:\t\t%s\n' % i.real_clock)
        cluster_file.write('Members: \t')
        for latches in i.member_latches :
          cluster_file.write('%s ' % latches.latch_name)
        cluster_file.write('\n')
        cluster_file.write('Neighbours: \t%s\n' % i.neighbours )

      cluster_file.close()

    # --------------------------------------------------------
    # GENERAT CLUSTER GRAPHS
    # --------------------------------------------------------
    #GENERATE ONE CLUSTER PER LATCH
    #for all latches add cluster
    id = 0
    for latch in latch_list :
      cluster_name = "cluster_%d" % id
      latch.AddCluster(cluster_name)
      id += 1

    #for all latches, inst cluster and connect
    id = 0
    for latch in latch_list :
      cluster_name = "cluster_%d" % id
      cluster_list.append(Cluster(latch.GetType(), cluster_name))
      cluster_hash[cluster_name] = cluster_list[number_of_clusters]
      cluster_hash[cluster_name].AddLatch(latch)
      number_of_clusters += 1
      id += 1
      # inst. cluster
      # add latch as member
      # add polarity
      # add clock
      # add clustser as latch master
      ### add neighbour latches (up/down)

    #GET COMMON NEIGHBOURS
    # for each cluster
     # for each direct neighbour
      # add their direct neghbour/increment the hash (exept self)
    for cluster in cluster_list :
            cluster.MapCommonNeighbours()


    banned_clusters= {}
    def BannedClusters(from_clu, to_clu):
      if banned_clusters.has_key(from_clu):
        if(banned_clusters[from_clu].has_key(to_clu)):
          return 1
      if banned_clusters.has_key(to_clu):
        if(banned_clusters[to_clu].has_key(from_clu)):
          return 1
    return 0

def isCkGate(string):
    match = re.search(r'ClockGateSyncReset', string.latch_name)
    if match:
        return 1
    else:
        return 0

def FindNextCluster():
    max = 0
    from_clu = ''
    to_clu = ''
    for clu in cluster_list:
        for common in clu.common_neighbours_hash:
            if clu.common_neighbours_hash[common] > max and not
            BannedClusters(clu.name, common):
                if clu.real_clock == '' or cluster_hash[common].real_clock == '':
                    k = 1
                    #no nothing
                elif isCkGate(clu.member_latches[0]) or
                isCkGate(cluster_hash[common].member_latches[0]):
                    k = 1
                    #no nothing
                else :
                    max = clu.common_neighbours_hash[common]
                    from_clu = clu.name
                    to_clu = common

    if max is 0:
        return {'from' : 'False'  , 'to' : 'False'}
    return {'from' : from_clu, 'to' : to_clu}

# MERGING CLUSTERS
nop = 0
test = 0
while (nop < 10000000):
    retval = FindNextCluster()
    from_clu = retval['from']
    to_clu   = retval['to']
    if (from_clu is 'False' or to_clu is 'False'):
        break
    if from_clu is 'cluster_538' and to_clu is 'cluster_539':
        test = 1
    if (from_clu is '' or to_clu is '' or test is 1):
        #finnished!
        print "Looks like there nothing left to merge..."
        break
```

```
  else :
    if not cluster_hash[from_clu].MergeClusters(cluster_hash[to_clu]):
      if not banned_clusters.has_key(from_clu):
        banned_clusters[from_clu]={}
      if not banned_clusters.has_key(to_clu):
        banned_clusters[to_clu]={}
      banned_clusters[from_clu][to_clu] = 1
      banned_clusters[to_clu][from_clu] = 1
      nop += 1
    else:
      nop = 0
      print "Merging", from_clu , to_clu
      banned_clusters = {}

printClusterFile()

print_all_latches_to_file()

print "Name:\t\tMembers\tMonitors\tNeighbours\tFirstMemb"
for cluster in cluster_list:
  print '%s\t%d\t%d\t%d\t%s' % (cluster.name, len(cluster.member_latches)
  , cluster.number_of_monitors , len(cluster.neighbours)
  , cluster.member_latches[0].latch_name)
print "--------------------------------------------------------------"
```

Appendix E

Insertion of Bubble Razor Components

Script for insertion of the Bubble Razor components. Takes lookups and netlist as input.

```
#!/usr/bin/env /pri/mako/local/bin/python

import re
import sys
from subprocess import call
import os

#from sys import argv, exit

print "\n".join(sys.argv)
print sys.argv[1]

# VARIABLES
# Lines starting with these will be one-lined
oneline_string = "retracted retracted retracted"
# Must contain all latch cell-names
latch_string   = "retracted retracted"

input_ports_string    = "retracted"
output_ports_string   = "retracted"
# Ports that need bubbles up the hierarchy
#Input port : vector length
bubble_input_ports   = {'retracted' : 2, 'retracted' : 1, 'retracted': 1}
#Input port : vector length
```

```
bubble_output_ports = {'retracted' : 9, 'retracted' : 1}
module_reset_string   = "retracted" #Reset port string

bubble_verilog_models_string = "bubble.v" #file with bubble razor components

#CLOCK SETUP INVERTED
# clocks and subclocks and their phases
clock_nodes = {}
clock_nodes[r'root'] = {'slave' : 'ckFreq1', 'master' : 'ckFreq1_phase2'}

clock_nodes["retracted"] = {'slave' : 'ckFreq2', 'master': 'ckFreq2_phase2'}
clock_nodes[r'retracted'] = {'slave' : 'ckFreq2_phase2', 'master' : 'ckFreq2'}
clock_nodes[r'retracted'] = {'slave' : 'retracted', 'master': 'retracted'}
clock_nodes[r'retracted'] = {'slave' : 'retracted', 'master': 'retracted'}

for argument in sys.argv :
    match =  re.search(r'-help|-h', argument)
    if(match) :
        print "------------------------------------------------\n"
        print "Inserting Bubble-Razor system to a netlist\n"
        print "Need lookup tables for cluster and latches\n"
        print "Req. arguments:\n"
        print "1: cluster lookup\n"
        print "2: latch lookup\n"
        print "3: input netlist\n"
        print "4: output netlist\n"
        print ""
        print "Also need some config in script for cell_names"
        print "------------------------------------------------\n"
        sys.exit()

print sys.argv

if(len(sys.argv)-1 is 4) :
    cluster_lookup_name = sys.argv[1]
    latch_lookup_name = sys.argv[2]
    netlist_name = sys.argv[3]
    out_netlist_name = sys.argv[4]

else :
    print "------------------------------------------------\n";
    print "ERROR!\n";
    print "Need input arguments.\n";
    print "  -help for more info!\n";
    print "------------------------------------------------\n";
    sys.exit()

print "------------------------------------------------\n";
print "INPUTS\n";
print "Cluster lookup:\t%s\n" % cluster_lookup_name;
print "Latch lookup:\t%s\n" % latch_lookup_name;
print "Input netlist:\t%s\n" % netlist_name;
print "OUTPUT\n";
print "Output netlist:\t%s\n" % out_netlist_name;
print "------------------------------------------------\n";

#----------------------------------------------------
#INFO
# The actual connection of clusters and latches is
# done in the SelfWrite functions in both the Latch
# and Cluster classes.
#
#Cluster wires:
# Cluster CK: ck_"cluster_name"
# Bubble Out: bubbleOut_"cluster_name"
#
#
#Latch wires:
# Error: error_"latch_name" (optinal)
#
#Ports:
# Ports are a special case, they look like a cluster but is not
# defined in this module. Instead every data-port get a dummy cluster
#----------------------------------------------------



class Latch:
    def __init__(self, name):
        self.name = name
        self.cluster = ''
        self.monitor = 'False' #Should this latch have a monitor?
    def AddCluster(self, cluster):
        self.cluster = cluster
    def AddMonitor(self, bool):
        self.monitor = bool
    def GetCluster(self):
        return self.cluster
    def GetMonitor(self):
        return self.monitor
    def WriteWires(self, file):
        file.write("wire error_%s;" % self.name)
    def WriteSelf(self, line, out_file):
        if self.monitor == 'True':
           node_match = re.findall(r'\..+?\((.*?)\)', line)
           port_match = re.findall(r'\.(.+?)\(', line)

           if('RN' in port_match) :
                out_line = 'BubbleRazor ' + self.name + ' (.ck(), .rn(), .d()'\
           '.err(error_'\
                 '+ self.name +'), .q(), .qn());\n'
            elif('SN' in port_match):
                out_line = 'BubbleRazorNegSet ' + self.name + ' (.ck(), .sn()'\
        ', .d(), .err(error_'\
                '+ self.name +'), .q(), .qn());\n'
            else :
                print "ERROR: NO RESET PIN ON LATCH"

            out_file.write('wire error_' + self.name + ';\n')

            index=0
            for port in port_match:
                if  port == 'D' and node_match[index]:
                    out_line = re.sub(r'(.+?\.d\().*?(\).+)', r'\g<1>%s\g<2>'\
          % node_match[index],    out_line)
                elif port == 'GN' and node_match[index]:
                    cluster_clock = 'ck_' + self.cluster
                    out_line = re.sub(r'(.+?\.ck\().*?(\).+)', r'\g<1>%s\g<2>'\
          % ('ck_'+self.cluster), out_line)
                elif port == 'RN' and node_match[index]:
                    out_line = re.sub(r'(.+?\.rn\().*?(\).+)', r'\g<1>%s\g<2>'\
          % node_match[index],    out_line)
                elif port == 'SN' and node_match[index]:
                    out_line = re.sub(r'(.+?\.sn\().*?(\).+)', r'\g<1>%s\g<2>'\
          % node_match[index],    out_line)
                elif port == 'Q'  and node_match[index]:
                    out_line = re.sub(r'(.+?\.q\().*?(\).+)', r'\g<1>%s\g<2>'\
          % node_match[index],    out_line)
                elif port == 'QN' and node_match[index]:
                    out_line = re.sub(r'(.+?\.qn\().*?(\).+)', r'\g<1>%s\g<2>'\
          % node_match[index],    out_line)
                index +=1
            #print "L", line
            #print "O", out_line
            out_file.write(out_line)
        else :
            line = re.sub(r'(\.GN\().+?(\))', r'\g<1>ck_%s\g<2>'
      % self.cluster , line)
            out_file.write(line)

class Cluster:
    def __init__(self, name):
        self.name = name
        self.bubble_in = [] #neighbour clusters, defines all bubble in
        self.monitor_members = [] #member latches, defines all the cluster err in
        self.clock = ''
        self.real_clock = ''
        self.port = 'False'
    def AddClock(self, clock):
        self.clock = clock
    def AddRealClock(self, clock):
        self.real_clock = clock
    def AddMonitorMembers(self, monitor_members):
        monitor_members=re.sub(r'(\(.+?\))', '', monitor_members)
        monitor_members=re.sub(r'\s', '  ', monitor_members)
        match = re.findall(r'(?:\s|^)(.+?)(?:\s|$)', monitor_members)
        if self.name == 'cluster_60':
            print monitor_members
            print match
        if match:
            self.monitor_members = self.monitor_members + match
    def AddBubbleIn(self, bubbles):
        match = re.findall(r'\'(.+?)\'', bubbles)
        if match:
            self.bubble_in = self.bubble_in + match
    def GetClock(self):
        return self.clock
    def GetRealClock(self):
        return self.real_clock
    def GetMembers(self):
        return self.members
    def GetBubbleIn(self):
        return self.bubble_in
    def WriteWires(self, file):
        file.write("wire bubbleOut_%s, ck_%s;" % (self.name, self.name))
    def WriteSelf(self, out_file):
        if(not clock_nodes.has_key(r'%s' % self.real_clock) or
    re.search(r'^\s*?$', self.real_clock)):
            print "Ignoring cluster:" , self.monitor_members
            bubble_in_OR = 'assign bubbleIn_'\
     + GetCleanPort(self.monitor_members[0]) +' ='
            for neighbour in self.bubble_in :
                if clusters_dict[neighbour].port == 'False':
                    bubble_in_OR = bubble_in_OR + ' bubbleOut_'\
       + neighbour + " ||"
                else : #Ghost cluster, bubble for port
                  bubble_in_OR = bubble_in_OR + ' bubbleOut_' +\
                      GetCleanPort(clusters_dict[neighbour].monitor_members[0]) + " ||"
                  print "Add port bubble to cluster", ' bubbleOut_'\
                      + GetCleanPort(clusters_dict[neighbour].monitor_members[0])
            bubble_in_OR = re.sub(r'\|\|$', r';\n\n', bubble_in_OR)
            if not re.search(r'=\s*?$', bubble_in_OR):
                out_file.write(bubble_in_OR)
                print bubble_in_OR
            return 0

        #OUTPUT WIRES
        out_file.write("wire bubbleIn_"+ self.name + ', bubbleOut_'\
    + self.name + ", ck_" + self.name + ";\n")

        #BUBBLE IN
        bubble_in_OR = 'assign bubbleIn_' + self.name +' ='
        for neighbour in self.bubble_in :
            if clusters_dict[neighbour].port == 'False':
                bubble_in_OR = bubble_in_OR + ' bubbleOut_' + neighbour + " ||"
            else : #Ghost cluster, bubble for port
                bubble_in_OR = bubble_in_OR + ' bubbleOut_'\
        + GetCleanPort(clusters_dict[neighbour].monitor_members[0]) + " ||"
        bubble_in_OR = re.sub(r'\|\|$', r';\n', bubble_in_OR)
        out_file.write(bubble_in_OR)

        #CLUSTER ERR
        have_monitor = 'False'
        error_in_OR = 'assign error_'+ self.name +' ='
        for member in self.monitor_members:
            #Only if member is a monitor
            if latches_dict[member].monitor == 'True':
                have_monitor = 'True'
                error_in_OR = error_in_OR + ' error_'+ member  +' ||'
        error_in_OR = re.sub(r'\|\|$', r';\n', error_in_OR)

        if have_monitor == 'True':
            out_file.write(error_in_OR)
            out_file.write("wire error_"+ self.name +";\n")
            cluster_error_string = 'error_' + self.name
        else :
            cluster_error_string = '1\'b0'

        #CLOCK XXX NOTE: every cluster is clocked on fast clock
        match = re.search(r'\'', self.clock)
        clock_string_master = 'ERROR'
        clock_string_slave  = 'ERROR'
        if(self.real_clock == 'root'):
            if(match): #slave
                #phase two clock
                clock_string_master = clock_nodes['root']['slave']
                clock_string_slave  = clock_nodes['root']['master']

            else: #master
                #phase one clock
```

```
                    clock_string_master = clock_nodes['root']['master']
                    clock_string_slave  = clock_nodes['root']['slave']

            else :
                if(clock_nodes.has_key(self.real_clock)):
                    clock_string_master = clock_nodes[self.real_clock]['master']
                    clock_string_slave  = clock_nodes[self.real_clock]['slave']
                else :
                    print "Me no latch 2, but port:" , self.monitor_members
                    print self.real_clock

            out_line = 'BubbleClusterCtrl '+ self.name + \
                       ' ( .bubbleIn(bubbleIn_'+ self.name + \
                       '), .clusterErr('+ cluster_error_string + \
                       '), .ck('+ clock_string_master + \
                       '), .ck_b('+ clock_string_slave + \
                       '), .invEnable(' + module_reset_string + \
                       '), .clusterCk(ck_'+ self.name + \
                       '), .bubbleOut(bubbleOut_'+ self.name +'));\n\n'

            out_file.write(out_line)

    clusters_dict = {}
    latches_dict = {}

    def CheckIfPort(name):
        clean_name = re.sub(r'(\[.*?\])', r'', name)
        clean_name = re.sub(r'(\(.*?\))', r'', clean_name)
        clean_name = re.sub(r'(\s)', r'', clean_name)
        return (re.search(r'%s' % clean_name , input_ports_string) or
      re.search(r'%s' % clean_name, output_ports_string))

    def GetCleanPort(name):
        clean_name = re.sub(r'(\(.*?\))', r'', name)
        clean_name = re.sub(r'(\s)',      r'', clean_name)
        return clean_name

    def ReadToMem(file, offset) :
        line_num = 1
        start_line = 10000000
        cluster_or_latch = ''
        state = ''
        current_object = ''
        state = 'GET_OBJECT'
        for line in file :
            match = re.search(r'TOTAL\sNUMBER\sOF\s+(.+?)\:\s+?(\d+)$', line)
            if match :
                start_line = 5 #offset
                print start_line
                cluster_or_latch = match.group(1)
                print cluster_or_latch
            elif(line_num > start_line): # Fill classes
                # If Cluster
                if(cluster_or_latch == 'CLUSTERS') :
                    if state == 'GET_OBJECT':
                        match = re.search(r'Cluster:\s+(.+?)$', line)
                        if(match):
                            #print match.group(1)
                            state='GET_CLOCK'
                            current_object = match.group(1)
                            clusters_dict[current_object] = Cluster(current_object)

                    elif(state == 'GET_CLOCK'):
                        match = re.search(r'Clock:\s+(.+?)$', line)
                        if(match):
                            #print match.group(1)
                            state = 'GET_REALCK'
                            clusters_dict[current_object].AddClock(match.group(1))

                    elif(state == 'GET_REALCK'):
                        match = re.search(r'RealCK:\s+(.+?)$', line)
                        if(match):
                          #print match.group(1)
                          state = 'GET_MEMBERS'
                          clusters_dict[current_object].AddRealClock(match.group(1))

                    elif(state == 'GET_MEMBERS'):
                      match = re.search(r'Members:\s+(.+?)$', line)
                      if(match):
                        #print match.group(1)
                        state = 'GET_BUBBLES'
                        clusters_dict[current_object].AddMonitorMembers(match.group(1))
                        if CheckIfPort(match.group(1)) :
                          clusters_dict[current_object].port = 'True'

                    elif(state == 'GET_BUBBLES'):
                        match = re.search(r'Neighbours:\s+(.+?)$', line)
                        if(match):
                            #print match.group(1)
                            state = 'GET_OBJECT'
                            clusters_dict[current_object].AddBubbleIn(match.group(1))

                # If Latch
                elif(cluster_or_latch == 'LATCHES'):
                    if state == 'GET_OBJECT':
                        match = re.search(r'^Latch:\s+\|\s+(.+?)$', line)
                        if(match):
                            #print match.group(1)
                            state='GET_CLUSTER'
                            current_object = match.group(1)
                            latches_dict[current_object] = Latch(current_object)

                    if state == 'GET_CLUSTER':
                        match = re.search(r'^Cluster:\s+\|\s+(.+?)$', line)
                        if(match):
                            #print match.group(1)
                            state='GET_MONITOR'
                            latches_dict[current_object].AddCluster(match.group(1))

                    if state == 'GET_MONITOR':
                        match = re.search(r'^Monitor:\s+\|\s+(.+?)$', line)
                        if(match):
                            #print match.group(1)
                            state='GET_OBJECT'
                            latches_dict[current_object].AddMonitor(match.group(1))

                else :
                    print "ERROR! Error in lookup: %s" % cluster_or_latch
                    sys.exit()

            match = False
            line_num += 1

    cluster_lookup = open(cluster_lookup_name)
    latch_lookup = open(latch_lookup_name)

    out_netlist = open(out_netlist_name, 'w')

    # ----------------------------
    ReadToMem(cluster_lookup, 5)
    ReadToMem(latch_lookup, 5)
    # ----------------------------

    cluster_lookup.close()
    latch_lookup.close()

    def OneLineInst(file, out_file, match_string) :
        last = True
        out_line = ''
        for line in file :
            match = re.search(r'^\s*(.+?)\s+(?:.+?)\s*\(', line)
            if match :
                check_latch = match.group(1)
                match = re.search(r'(?:^|\s)(%s)(?:\s|$)'\
          % re.escape(check_latch), match_string) #check if comp is latch
                    if match :
                        last = False
                if not last:
                    match = re.search(r';', line)
                    if match:
                        out_line = line
                        last = True
                    else :
                        out_line = re.sub(r'\n', '', line)
                else :
                    out_line = line

                out_file.write(out_line)

    netlist = open(netlist_name)
    temp_file = open('tmp_netlist.v', 'w')
    OneLineInst(netlist, temp_file, oneline_string)
    temp_file.close()
    netlist.close()
    temp_file = open('tmp_netlist.v')

    for line in temp_file:
        module_match    = re.search(r'^\s*?module', line)
        wire_match      = re.search(r'^\s*?wire'  , line)
        object_match    = re.search(r'^\s+(.+?)\s+(.+?)\s*\(', line)
        endmodule_match = re.search(r'^\s*?endmodule\s', line)

        if   module_match:
            out_netlist.write(line)
        elif wire_match:
            out_netlist.write(line)
        elif object_match:
            match = re.search(r'(?:^|\s)(%s)(?:\s|$)'
        % re.escape(object_match.group(1)), latch_string) #check if comp is latch
            if match:
                latches_dict[object_match.group(2)].WriteSelf(line, out_netlist)
            else :
                out_netlist.write(line)
        elif endmodule_match:
            for cluster in clusters_dict:
                clusters_dict[cluster].WriteSelf(out_netlist)
            out_netlist.write(line)
        else :
            out_netlist.write(line)

    out_netlist.close()
    temp_file.close()
```

```

out_netlist = open(out_netlist_name, 'r')
temp_file   = open('tmp_netlist.v', 'w')

current_module = ''
wires = {}
wire_end = 'true'

#sniff all wires
file_content = out_netlist.readlines()
for line in file_content:

    match = re.search(r'^\s*?module\s+(.+?)(\s|\()', line )
    if match:
        current_modules = match.group(1)
        wires[current_module] = ''

    if re.search(r'^\s*wire\s', line):
        #append this line here
        wires[current_module] = wires[current_module] + line
        if re.search(r';\s*$', line):
            wire_end = 'true'
        else :
            wire_end = 'false'
    elif wire_end == 'false' :
        #append this line here
        wires[current_module] = wires[current_module] + line
        if re.search(r';\s*$', line):
            wire_end = 'true'
        else :
            wire_end = 'false'

wire_end = 'true'
first_wire = 'false'
inserted_bubble_ports = 'False'
for line in file_content:
    match = re.search(r'^\s*?module\s+(.+?)(\s|\()', line )
    if match:
        current_modules = match.group(1)
        for port in bubble_input_ports:
            line = re.sub(r'(\))',r', bubbleOut_%s, bubbleIn_%s)'
      % (port, port), line )
        for port in bubble_output_ports:
            line = re.sub(r'(\))',r', bubbleOut_%s, bubbleIn_%s)'
      % (port, port), line )

    if re.search(r'^\s*?(input|output)\s', line)
  and inserted_bubble_ports == 'False':
        inserted_bubble_ports = 'True'
        for port in bubble_input_ports:
            if bubble_input_ports[port] > 1:
                temp_file.write("input ["+ str(bubble_input_ports[port]-1)
        +":0] bubbleOut_"+ port +";\n")
                temp_file.write("output ["+ str(bubble_input_ports[port]-1)
        +":0] bubbleIn_"+ port +";\n")
            else :
                temp_file.write("input bubbleOut_"+ port +";\n")
                temp_file.write("output bubbleIn_"+ port +";\n")

        for port in bubble_output_ports:
            if bubble_output_ports[port] > 1:
                temp_file.write("input ["+ str(bubble_output_ports[port]-1)
        +":0] bubbleOut_"+ port +";\n")
                temp_file.write("output ["+ str(bubble_output_ports[port]-1)
        +":0]  bubbleIn_"+ port +";\n")
            else :
                temp_file.write("input bubbleOut_"+ port +";\n")
                temp_file.write("output bubbleIn_"+ port +";\n")

    if re.search(r'^\s*wire\s', line):
        if re.search(r';\s*$', line):
            wire_end = 'true'
        else :
            wire_end = 'false'

        if first_wire =='false':
            first_wire = 'true'
            #print all wires for module
            temp_file.write(wires[current_module])
            print "print wires"

    elif wire_end == 'false' :
        if re.search(r';\s*$', line):
            wire_end = 'true'
        else :
            wire_end = 'false'

    else :
        #write line...
        ff=0
        temp_file.write(line)

out_netlist.close()
temp_file.close()
print out_netlist_name
os.system("mv tmp_netlist.v %s" % out_netlist_name)
```

```

# append models
print "Appending models to end of file..."
out_netlist   = open(out_netlist_name, 'a')
bubble_models = open(bubble_verilog_models_string, 'r')
out_netlist.write(bubble_models.read())
```

Appendix F

Lookup Example

```
--------------------------------------------
Cluster:  cluster_8
Clock:    ckFreq1
RealCK:    root
Members:  ckFreq1_p2_REG455_S1 ckFreq1_p2_REG473_S1 ckFreq1_p2_REG471_S1
      ckFreq1_p2_REG469_S1 ckFreq1_p2_REG467_S1 ckFreq1_p2_REG465_S1
      ckFreq1_p2_REG463_S1 ckFreq1_p2_REG479_S1 ckFreq1_p2_REG481_S1
      ckFreq1_p2_REG483_S1 ckFreq1_p2_REG485_S1 ckFreq1_p2_REG461_S1
      ckFreq1_p2_REG475_S1 ckFreq1_p2_REG487_S1 ckFreq1_p2_REG489_S1
      ckFreq1_p2_REG477_S1
Neighbours:   ['cluster_14', 'cluster_33', 'cluster_0']
--------------------------------------------
```

```
-----------------------------------------------
Latch:    |  ckFreq1_p2_REG471_S1
Clock:    |  ckFreq1
Real_Clock: |  root
Type:   |  slave
Cluster:  |  cluster_8
Monitor:  |  True
Slack:    |  3.5
Up:     |  ['ckFreq1_p1_REG468_S2', 'ckFreq1_p1_REG470_S2',
          'ckFreq1_p1_REG496_S2', 'ckFreq1_p1_REG494_S2',
          'ckFreq1_p1_REG492_S2', 'ckFreq1_p1_REG490_S2',
          'ckFreq1_p1_REG488_S2', 'ckFreq1_p1_REG486_S2',
          'ckFreq1_p1_REG484_S2', 'ckFreq1_p1_REG476_S2',
          'ckFreq1_p1_REG462_S2', 'ckFreq1_p1_REG464_S2',
          'ckFreq1_p1_REG466_S2', 'ckFreq1_p1_REG472_S2',
          'rfDr[0] (input port clocked by ckFreq1)',
          'ckFreq1_p1_REG520_S1', 'ckFreq1_p1_REG478_S2',
          'ckFreq1_p1_REG516_S1', 'ckFreq1_p1_REG514_S1',
           'ckFreq1_p1_REG504_S1', 'ckFreq1_p1_REG502_S1']
Down:   |  ['ckFreq1_p1_REG472_S2']
-----------------------------------------------
```

Appendix G

Number of Bubble Razor Components

Tabular of the number of element in the different stages.

	Flip-flop version	Latch pre-retimeing and pre-clustering	Latch post-retiming and post-clustering
Flip-Flops	243	-	-
Latches*	-	486	524
-Monitors	-	257	257
Clusters	-	524	19

*Includes both latches and monitors

Number and size of OR-trees for bubble and error generation.

OR-three size	Count
3	2
4	2
7	2
16	16
26	1

Bibliography

D. Blaauw, A. Devgan, and F. Najm. Leakage power: trends, analysis and avoidance. In *Design Automation Conference, 2005. Proceedings of the ASP-DAC 2005. Asia and South Pacific*, volume 1, pages T – 2 Vol. 1, jan. 2005. doi: 10.1109/ASPDAC.2005.1466116.

A. Chandrakasan, S. Sheng, and R. Brodersen. Low-power cmos digital design. *Solid-State Circuits, IEEE Journal of*, 27(4):473 –484, apr 1992. ISSN 0018-9200. doi: 10.1109/4.126534.

Chinnery, Keutzer, Sanghavi, Killian, and Sheth. *Automatic Replacement of Flip-Flops by Latches in ASICs*. Springer US, march 2004. ISBN 978-1-4020-7113-3. doi: 10.1109/ISQED.2007.23.

S. Das, D. Roberts, S. Lee, S. Pant, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. A self-tuning dvs processor using delay-error detection and correction. *Solid-State Circuits, IEEE Journal of*, 41(4):792–804, 2006. ISSN 0018-9200. doi: 10.1109/JSSC.2006.870912.

S. Das, C. Tokunaga, S. Pant, W.-H. Ma, S. Kalaiselvan, K. Lai, D. Bull, and D. Blaauw. Razorii: In situ error detection and correction for pvt and ser tolerance. *Solid-State Circuits, IEEE Journal of*, 44(1):32–48, 2009. ISSN 0018-9200. doi: 10.1109/JSSC.2008.2007145.

M. Elgebaly and M. Sachdev. Variation-aware adaptive voltage scaling system. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 15(5):560 –571, may 2007. ISSN 1063-8210. doi: 10.1109/TVLSI.2007.896909.

D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. Razor: a low-power pipeline based on circuit-level timing speculation. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pages 7–18, 2003. doi: 10.1109/MICRO.2003.1253179.

M. Fojtik, D. Fick, Y. Kim, N. Pinckney, D. Harris, D. Blaauw, and D. Sylvester. Bubble razor: Eliminating timing margins in an arm cortex-m3 processor in 45 nm cmos using architecturally independent error detection and correction. *Solid-State Circuits, IEEE Journal of*, 48(1):66–81, 2013. ISSN 0018-9200. doi: 10.1109/JSSC.2012.2220912.

M. T. Kollerud. Propagation-delay monitoring techniques and path analysis for use in dvs. Specialization project, dec. 2012.

J. Park and J. Abraham. A fast, accurate and simple critical path monitor for improving energy-delay product in dvs systems. In *Low Power Electronics and Design (ISLPED) 2011 International Symposium on*, pages 391 –396, aug. 2011. doi: 10.1109/ISLPED.2011.5993672.

T. Sato and Y. Kunitake. A simple flip-flop circuit for typical-case designs for dfm. In *Quality Electronic Design, 2007. ISQED '07. 8th International Symposium on*, pages 539 –544, march 2007. doi: 10.1109/ISQED.2007.23.

Design Compiler Register Retiming Reference Manual. Synopsys, f-2011.09-sp2 edition, 2011.

A. Uht. Uniprocessor performance enhancement through adaptive clock frequency control. *Computers, IEEE Transactions on*, 54(2):132 – 140, feb. 2005. ISSN 0018-9340. doi: 10.1109/TC.2005.34.