Evaluation of Data-Path Topologies for Self-Timed Conditional Statements

by

Navaneeth Prasannakumar Jamadagni

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
in
Electrical and Computer Engineering

Dissertation Committee:
Robert W. Daasch, Chair
Ivan E. Sutherland
Xiaoyu Song
Jo C. Ebergen
Bryant W. York

Portland State University
2015

ABSTRACT

This research presents a methodology to evaluate data path topologies that implement a conditional statement for an average-case performance that is better than the worst-case performance. A conditional statement executes one of many alternatives depending on how Boolean conditions evaluate to true or false. Alternatives with simple computations take less time to execute. The self-timed designs can exploit the faster executing alternatives and provide an average-case behavior, where the average depends on the frequency of simple and complex computations, and the difference in the completion times of simple and complex computations. The frequency of simple and complex computations depends on a given workload. The difference in the completion times of a simple and complex computations depend on the choice of a data path topology.

Conventional wisdom suggests that a fully-speculative data path, independent of the design style, yields the best performance. A fully-speculative data path executes all the choices or alternatives in a conditional statement in parallel and then chooses the correct result. Using a division algorithm as an example of an instruction that embodies a conditional statement, the proposed methodology shows that a fully-speculative design is suitable for a synchronous design but a less-speculative design is suitable for a self-timed design. Consequently, the results from the SPICE simulation of the extracted netlists show that on average, the self-timed divider is approximately 10% faster, consumes 42% less energy per division and 20% less area than the synchronous divider.

In addition to the evaluation methodology, this research also presents the derivation of four new radix-2 division algorithms that offer a simpler quotient selection logic

compared to the existing radix-2 division algorithms. A circuit optimization technique called *Glissando* is presented in this research. *Glissando* exploits a simple idea that the non-critical bits can arrive late at the input of the registers to reduce the delay of the data paths. The effect of the variations in manufacturing on the functionality of the divider designs is also analyzed in this research.

To my mother and late father

Acknowledgments

I am indebted to Professors Ivan Sutherland and Robert Daasch for their helpful comments and suggestions throughout this research.

I owe a great debt to the Asynchronous Research Center for providing me with the financial, moral and technical support. Special thanks to my colleagues Marly Roncken, Chris Cowan, Swetha Mettalagilla and Hoon Park for engaging with me in fruitful technical discussions throughout this research. I also appreciate the helpful suggestions that Glenn Sherly and all the members of the Integrated Circuits Design and Test laboratory provided. I thank Professor Bryant York for helping me understand the RSA cryptosystems and showing a keen interest in this research.

I thank Oracle Labs for supporting this research through internships and proving me with the computing infrastructure necessary for this research. I also thank Jo Ebergen for mentoring me during the internships and for his helpful comments and suggestions throughout this research.

Finally, I owe a special thanks to my family and friends for keeping me sane during the tough times.

Table of Contents

List of Tables

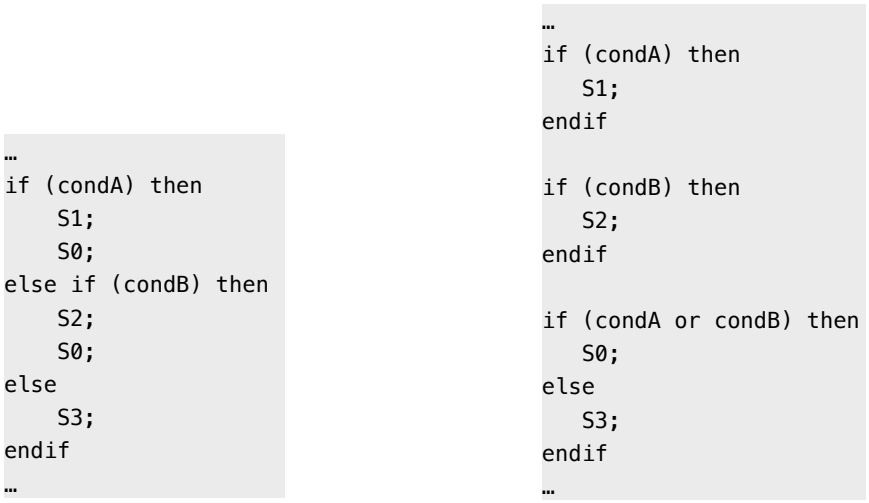
List of Figures

1

Introduction

Conditional statements execute one of many choices depending on how the Boolean conditions evaluate to true or false. Figure 1.0.1a shows an example of an `if-then-else` conditional statement for which we seek a hardware implementation. In the figure, `S0` to `S3` are some statements. Depending on how the conditions `condA` and `condB` evaluate, the code in the figure executes one of the following three sequence of statements: `S1&S0`, `S2&S0` and `S3`. There are different ways of writing the conditional statement in Figure 1.0.1a, and Figure 1.0.1b shows one such alternative.

Just as there are different ways of writing a conditional statement, there are different ways to implement a conditional statement in hardware. Figures 1.0.2a and 1.0.2b show two different ways of implementing the data path for the conditional statement in Figure 1.0.1a. In Figures 1.0.2a and 1.0.2b the modules with labels *S0, S1, S2* and *S3* implement the statements `S0`, `S1`, `S2` and `S3` respectively, and the module with label *cond* implements the logic for condition evaluation. The choices, `S1&S0`, `S2&S0` and `S3` may take different computation times. For example, `S3` may execute a simpler computation than either `S1&S0` or `S2&S0` resulting in faster execution of `S3`. In such a scenario an asynchronous or self-timed design may be preferred over a synchronous design to take advantage of the faster execution of the `S3` choice.



```
…
if (condA) then
    S1;
    S0;
else if (condB) then
    S2;
    S0;
else
    S3;
endif
…
```

(a)

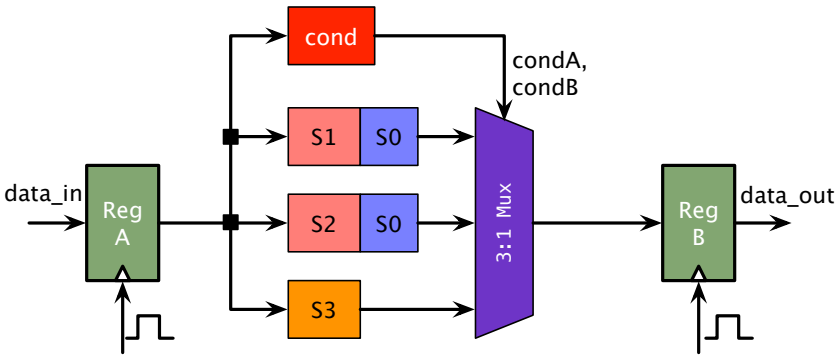
```
…
if (condA) then
   S1;
endif

if (condB) then
   S2;
endif

if (condA or condB) then
   S0;
else
   S3;
endif
…
```

(b)

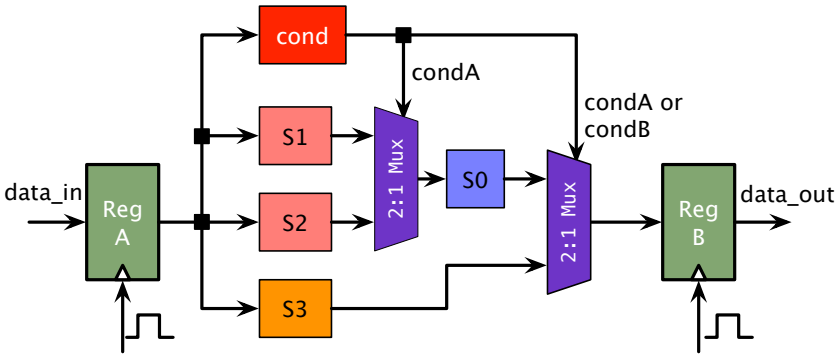
Fig. 1.0.1: Two different ways of writing a conditional statement.

Asynchronous or self-timed circuits can exploit a simple idea that some computations are faster than others. Self-timed circuits taking advantage of the faster computations have an average-case behavior, where the average depends on the difference in the completion times of a hard computation and an easy computation, and the frequency of hard and easy computations for a given workload. The difference in the completion times of a hard and an easy computation depends on the choice of a data path topology.

Conventional wisdom suggests that a fully-speculative execution technique, independent of the circuit design style, yields the best performance. A fully-speculative execution technique executes all the choices or alternatives in a conditional statement in parallel and then chooses the correct result. The data path in Figure 1.0.2a is an example of a fully-speculative execution technique which executes all three choices, `S1&S0`, `S2&S0` and `S3` in parallel and then selects a correct result depending on how the condition evaluates.



(a) Hardware design of the conditional statement in Fig. 1.0.1a



(b) Hardware design of the conditional statement in Fig. 1.0.1b

Fig. 1.0.2: Two different hardware designs for the conditional statements in Figure 1.0.1

In this research, I challenge the conventional notion that a fully-speculative data path yields an optimum performance for self-timed designs. In Figure 1.0.2a, if the select signals to the 3:1 multiplexer arrive last, then the select signals determine the delay of the design rather than the data signals, failing to take advantage of the faster execution of the S3 choice. There are many reasons why the select signals could arrive last, for example a high fan-out at the output of the *cond* module which is common in wide data paths. Therefore, a self-timed design using the data path topology in Figure 1.0.2a may fail to take advantage of the faster execution of the *S3* choice.

1.1 Research Objectives

The primary objective of this research is to present a systematic method to evaluate various data-path topologies that implement a conditional statement for an optimum average-case performance. To achieve an optimum average-case performance, the data computations such as S3 or S2&S0 must determine the delay of a data path rather than the select signals, condA or condB, of the multiplexers. This research uses a division algorithm as an example of a conditional statement to develop a methodology to evaluate data paths for average-case performance. Section 1.3 enumerates the reasons for considering a divider example.

The secondary objectives of this research are as follows:

1. Propose and evaluate modifications to the division algorithms presented in [10] and [34].

2. Present an analysis of a new circuit optimization technique.

3. Present a control-path circuit for the self-timed design that can take advantage of the faster execution of the S3 statement.

4. Analyze the response of synchronous and self-timed designs under the influence of process and environmental variations.

1.2 Research Summary

To achieve the primary objective, the methodology proposed in this research takes into account how frequently a division algorithm executes a shift-only and add operations. The methodology shows that while a fully-speculative design is suitable for a synchronous design, a less-speculative design is suitable for a self-timed design. Consequently, the results from the SPICE simulation of the extracted netlists show

that compared to the synchronous divider design, the self-timed counterpart offers an improvement of 10%, 42% and 20% in average latency per division, average energy per division and area, respectively. The results from SPICE simulations at different process and environment corners expose a new duty-cycle constraint. The duty-cycle violation results in functional failure. A regression analysis of the slacks from static timing analysis tool (STA) and SPICE simulation shows the possibility of predicting an yield-loss for a less-speculative design from STA's slack estimates. For a fully-speculative design, STA's slack estimates fail to predict yield-loss.

The other contributions of this research include three new division algorithms that offer a simpler quotient selection logic compared to the division algorithms in [10] and [34]. An optimization technique called *Glissando* that offers a way to increase the operand word-size without affecting the delay of the data path. A Design of a control path for the self-timed divider that can take advantage of the faster shift-only operation.

1.3 Why division?

Division is an iterative operation where each iteration involves either performing a shift-only or an addition operation. Shift-only operation is faster than an addition operation and the condition for performing a shift-only or an addition operation is based on the value of the partial remainder that makes a division algorithm a good example of the `if-then-else` conditional statement in Figure 1.0.1a.

Division is also the slowest of the basic arithmetic operations performed in a general-purpose microprocessors. Oberman and Flynn in [24] showed that although division occurs less frequently than other arithmetic operations, having an efficient divider is necessary for a good system performance. Even in case of modern microprocessors which contain multiple cores with multiple division modules, computer

architects give careful consideration to divide instructions to improve the system performance. Shah et al. in [32] describe how divide instructions in a SPARC T4 processor can potentially take over most or all of the core's shared resources resulting in performance degradation. Furthermore, from a design cycle perspective, a division module is a bottleneck in achieving the overall-system timing-closure. Failure to achieve timing-closure results in increasing the clock-period for the divide pipeline or the entire core, reducing the system performance.

1.4 Thesis Organization

The following is a general overview of the organization of this document. The organization is such that the chapters reflect various stages of a design cycle, from evaluating algorithms to analyzing the response of the designs to variations in manufacturing.

- Chapter 2 presents a background on the works related to division algorithms, self-timed designs and self-timed divider designs.

- Chapter 3 derives several radix-2 division algorithms that are modifications to the algorithms in [10] and [34]. The derivation method uses a graphical tool called Diamond diagram and invariants to prove the correctness of the algorithms. This chapter also evaluates the division algorithms for latency per division, average energy per division, area and the frequency of shift-only and add operations. The evaluation employs static methods such as STA to estimate latency, and switching activity and gates sizes to estimate energy. Dynamic simulations with a pair of input operands that represents a target workload gives the frequency of shift-only and add operations. This chapter addresses the secondary objective #1 in Section 1.1.

In the context of a design cycle, Chapter 3 reflects an initial stage in the design cycle where a designer must choose one of the several algorithms for a hardware implementation. At this stage of the design cycle, a designer can also make a decision about pursuing either synchronous or self-timed design style by examining the frequency of hard and easy computations.

- Chapter 4 presents a methodology to evaluate various data path topologies for an optimum average-case delay. The data path topologies implement a division algorithm selected in Chapter 3. To estimate an average-case delay, the proposed methodology takes into account how frequently a division algorithm executes shift-only and add operations. The method of logical effort is used to estimate the delay of the data path topologies. For comparison, ranking of the data path topologies using logical effort and static timing analysis tool is also presented in this chapter. Chapter 4 addresses the primary objective of this research. The outcome of this chapter is two data paths, one each for synchronous and self-timed divider designs.

  In the context of a design cycle, Chapter 4 reflects the stage in the design flow where a designer must evaluate the data path topologies that implement an algorithm from the previous stage for delay, energy and area, and choose a data path for potential optimization.

- Chapter 5 introduces and evaluates a new optimization technique called *Glissando* to further reduce the delay of the data paths selected in Chapter 4. The method of logical effort is used to evaluate the optimization technique. The *Glissando* technique is applicable for both synchronous and self-timed designs. This chapter addresses the secondary objective #2 in Section 1.1.

  Chapter 5 reflects the optimization stage in the design flow where a designer can make minor structural changes to the data path.

- Chapter 6 presents the physical design of synchronous and self-timed divider designs. This chapter also presents the design of a control path for self-timed divider that can take advantage of the faster shift-only operations in the data path. The proposed control path uses two bits from the data path to modulate the period of the synchronization pulses according to the operation performed in the data path. This chapter addresses the secondary objective #3 in section 1.1.

  Chapter 6 reflects the physical-design stage in the design flow where a design can undergo further optimizations such as gate sizing and Vt-swapping to satisfy timing, energy and area requirements.

- Chapter 7 compares the synchronous and self-timed divider designs developed in this research with other designs for delay, energy and area. The comparisons use results from the SPICE simulation of extracted netlists. This chapter also compares the response of synchronous and self-timed dividers to process variations. Furthermore, using a simple regression analysis this chapter examines if an earlier stage in the design flow can predict yield-loss. This chapter addresses the secondary objective #4 in Section 1.1.

  Chapter 7 reflects one of the final and important stages in the design flow where a designer performs a SPICE-level timing validation and estimates the functional-yield of the design, because a design with low functional-yield is often rejected from manufacturing.

The numbering of the figures, equations and tables follows the format `x.y.z`, where `x.y` is the chapter-section combination, and `z` is the figure, equation or table number.

2

Background

In this chapter, I discuss the previous works related to this research along with their limitations and describe how this research is different from the previous works.

2.1 Digit-Recurrence Division Algorithms

The digit-recurrence SRT division algorithm is the most frequently implemented division algorithm in general purpose processors [15]. The name SRT comes from the initials of the inventors of the algorithm, Sweeny, Robertson and Tocher [38, 30]. A standard radix-2 SRT algorithm retires a quotient digit from the set {-1, 0, 1}. Typically, the selection of a quotient digit relies on the four most-significant bits of the partial remainder in a redundant representation.

The latency of a division operation is a product of the number of iterations per division and the cycle time of an iteration. We can reduce the latency of a division operation by reducing the number of iterations per division, the cycle time of an iteration or both. A higher radix division algorithm reduces the number of iterations per division by retiring more quotient bits per iteration. For example, a radix-4 division algorithm retires two quotient bits per iteration and thus requires half as many

iterations per division as a radix-2 division algorithm that retires one quotient bit per iteration. The reduction in the number of iterations per division comes at the cost of increased cycle time of an iteration because of the increased complexity of the quotient selection function. Higher-radix division algorithms also require generating hard multiples of divisor such as $3D$, $5D$ etc, adding to the latency of the division operation. Harris, Oberman and Horowitz in [15] assert that the increased cycle time along with generating hard multiples of a divisor for high-radix division algorithms will limit the practical divider implementations to radix-2 and radix-4. This is evident from the work presented in [27, 18, 19] and [7], where the authors build a high-radix divider by cascading low-radix stages. In this research, comparisons with higher-radix division is limited to radix-4 implementations.

Alternatively, we can reduce the latency of a division operation by reducing the cycle time of an iteration. The logic that selects a quotient digit is called the quotient selection logic and it appears in the critical path of a divider design. Therefore, simplifying the quotient selection logic potentially leads to a low-latency divider design.

Burgess in [5] presented a radix-2 algorithm that simplified the quotient selection logic to inspect only the two most-significant bits of the partial remainder in a redundant representation to retire a quotient digit. This algorithm works only for the divisors in the range [1.5, 2) and according to the IEEE 754 standard a divisor can be in the range [1, 2). Therefore, the algorithm in [5] requires pre-scaling of both divisor and dividend for divisors in the range [1.5, 2). In [8], Cortadella and Lang proposed a technique of speculating the quotient digit. The speculated quotient digit has a high probability of being correct and when the speculation is incorrect, a rollback is performed. Because the rollback requires additional clock cycles, the number of cycles per division varies depending on the accuracy of the quotient speculation logic. Often, division and square-root operations share the same hardware which makes the algorithms in [5] and [8] difficult to implement.

Montuschi in [21], first presented the idea of using an over-redundant digit set to simplify the quotient selection logic for a radix-4 SRT algorithm. Srinivas, Parhi and Montalvo in [34] extended the work in [21] to develop a radix-2 SRT algorithm. The quotient selection logic for the algorithm in [34] inspects only the two most-significant bits of the partial remainder in a redundant representation. In [10], Ebergen, Sutherland and Chakraborty presented a new division algorithm that also simplifies the quotient selection logic to inspect only the two most-significant bits of the partial remainder in the a redundant representation. The difference between the algorithms in [34] and [10] are as follows:

- The algorithm in [34] uses a signed-digit representation for the partial remainder and carry-free additions. But, the algorithm in [10] uses a two's complement representation for the partial remainder and carry-save addition.

- The algorithm in [34] keeps the range of the partial remainder, $r \in (-2D, 2D)$. But, the algorithm in [10] keeps the range of the partial remainder, $r \in [-4, 4)$.

A detailed discussion on these differences along with several new division algorithms that enhance the performance characteristics of [33] and [10] appear in Chapter 3

In addition to simplifying the quotient selection logic of an algorithm, we can make optimizations at the circuit level to further reduce the latency of a division operation. When building a high-radix divider from low-radix stages, some of the computations in a stage can be overlapped with the computations in the next stage. Thus reducing the cycle time of an iteration. Harris et al., analyze various overlapped techniques for SRT dividers in [15]. Antelo et al., in [2] presented a technique of reducing the cycle time of an iteration by skewing the clock and applying the skewed clock to launch and capture the critical path. Burgess in [6] and Liu and Nannarelli in [19] presented radix-4 designs that use the technique presented in [2]. The overlapping techniques analyzed in [15] and the skewed-clock technique in [2] are a function of word size

and the cycle time increases as the word size increases. The optimization technique presented in this research called *Glissando* extends the technique in [2] to non-critical paths resulting in a cycle time that is independent of the word size. Chapter 5 presents a detailed analysis of the glisando optimization technique.

2.2 Self-Timed Design

There are two methods to design self-timed systems, completion-detection and bundled-data [3]. This research uses the bundled-data method.

The *Bundled-data* method uses a delay-matching technique. In this technique, the delay between the pulses latching the data at the two subsequent stages of a pipeline is matched to the worst-case delay in the data path. Figures 2.2.1a and 2.2.1b show a basic bundled-data stage and the timing-diagram, respectively. In Figure 2.2.1a, the module with the label `delay` is a delay module matching the delay in the data path such that $t_{fire} \geq t_{cq} + t_{dp} + t_{setup}$, where t_{fire} is the delay between the two fire pulses, $fire\_A$ and $fire\_B$, t_{cq} is the clock-to-q delay of the register `RegA`, t_{dp} is the critical-path delay in the data path and t_{setup} is the setup time of the register `RegB`. The matched delay is asymmetric; the delay is only for the request signal but not for the acknowledgment signal. The main advantage of the bundled-data method is that a standard synchronous single-rail implementation may be used, so implementations are easy to design, have low power and limited area. The key disadvantage, however, is that the completion is fixed to a worst-case computation, regardless of the actual data inputs.

In [23] Nowick et.al., presented a technique to allow bundled-data designs to operate at several different deterministic-speeds. The work presented in [23] uses multiple-delay models, one for each different case. The example in Figure 2.2.2a shows a bundled-data design using two delay models: one for the worst-case, `delay2`,

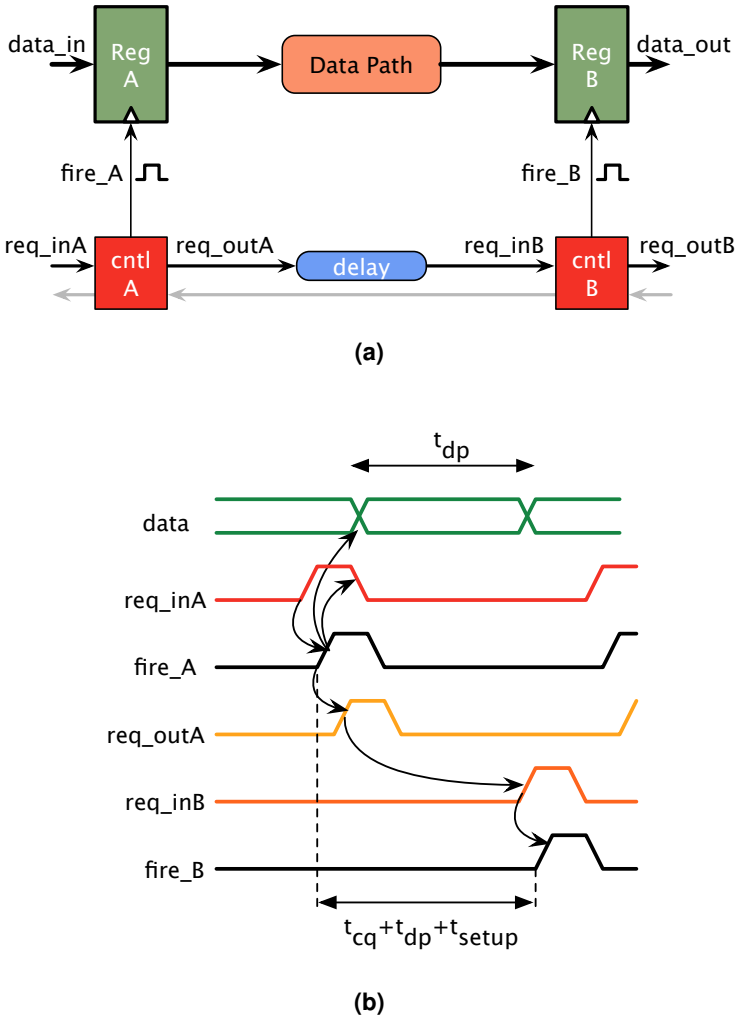


Fig. 2.2.1: Bundled-data method: Figure 2.2.1a shows a bundled-data stage with a unit-delay model and Figure 2.2.1b shows the singal transitions in a bundled-data method.

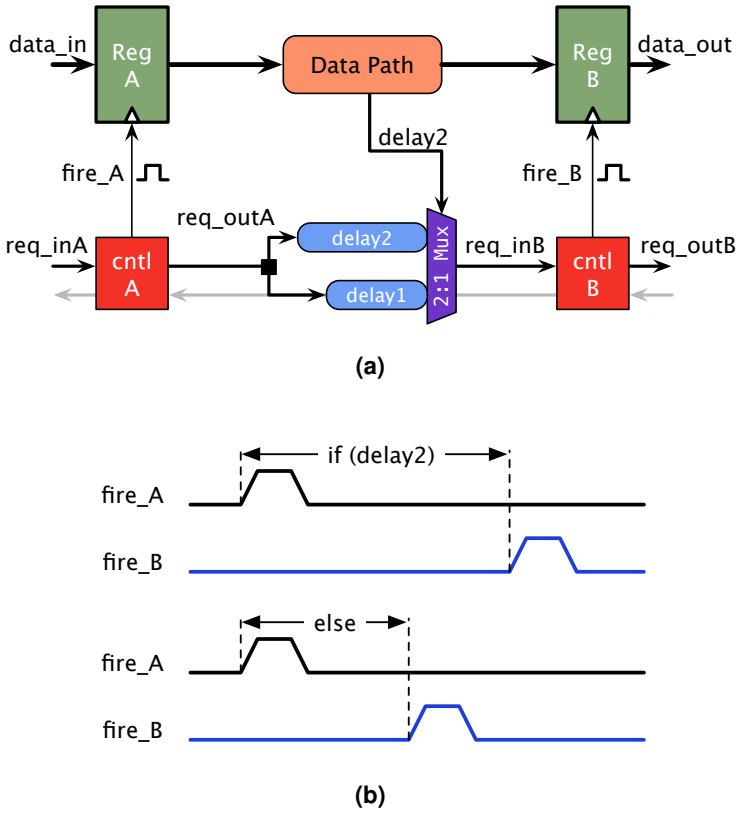


Fig. 2.2.2: Bundled-data method: Figure 2.2.2a shows a bundled-data stage with a two-delay model and Figure 2.2.2b shows the timing diagram for $fire\_A$ and $fire\_B$ signals in the two-delay model.

and the other one for the best-case, `delay1`.

There are several works [9, 4] and [14] that analyze the performance of self-timed pipelines, but after extensive literature search, I failed to find any work that quantitatively analyzes the effect of a design topology on the average-case performance. The methodology presented in Chapter 4 analyzes the effect of a design topology on the average-case performance of a division operation. The methodology can be easily extended to other conditional statements such as a booth-encoded iterative multiplier or at a macro-level for different conditional subroutine implementations.

Williams and Horowitz in [40] presented a self-timed divider implementing radix-2 SRT division algorithm. Matsubara et al., in [20] and Renaudin, Hassan and Guyot in

[29] also present self-timed divider designs that are extensions to the work in [40]. The self-timed dividers in [40, 29] and [20] use dynamic circuits with completion-detection design approach. All three self-timed designs fail to take advantage of the faster shift-only operation in the SRT algorithm. In this research, the self-timed divider uses static CMOS circuits and a single-rail bundle-data design method. Furthermore, the self-timed divider takes advantage of the faster shift-only operation using the method proposed in [23]. Chapter 6 presents the design of a control path for the self-timed divider.

3

Division Algorithms

This chapter presents the derivation of the division algorithms that offer simpler quotient selection logic compared to the division algorithms in [10] and [34]. The derivation method uses invariants and the *"Diamond diagram"* to prove the correctness of the algorithms. The diamond diagram is a graphical tool to visualize carry-save numbers and operations.

This chapter also evaluates the division algorithms for four figures of merit, latency per division, average energy consumption per division, area, and the fraction of shift-only operation and an addition operation per division. The fourth figure of merit, the fraction of shift-only operation and addition operation per division, is important when evaluating data paths for average-case performance, because the self-timed design can take advantage of the faster shift-only operation. By examining the fourth figure of merit, a designer can make a decision about pursuing a synchronous or a self-timed design style.

This research uses a division algorithm as an example of an `if-then-else` conditional statement and therefore the evaluation of the division algorithms presented in this chapter is applicable to other algorithms with `if-then-else` conditional statements, where different statements have different computation times.

3.1 Classification of Division Algorithms

Figure 3.1.1 shows the classification of the division algorithms considered. The classification is based on the type of addition that an algorithm performs. Division algorithms SRT, A1, A1b and A1c use a two's complement representation for the partial remainder and carry-save additions to perform addition and subtract operations. Algorithms B1, B1b and B1c use a binary signed-digit representation for the partial remainder and carry-free additions to perform addition and subtract operations. Algorithms A1, A1b, B1 and B1b are other contributions of this research.

The Section 3.4 explains the diamond diagram using a well-known standard radix-2 SRT algorithm [13, 25]. Section 3.5 presents the derivation of the algorithms A1, A1b and A1c. The derivation of the algorithms B1, B1b and B1c is in Appendix A.

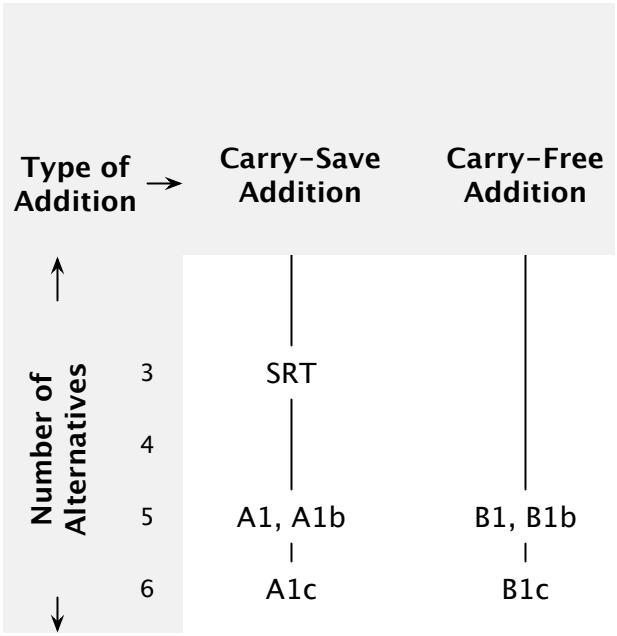


Fig. 3.1.1: Classification of the division algorithms based on the type of addition that an algorithm performs.

3.2 Division Preliminaries

Division Preliminaries

A division algorithm must compute an approximation to $Q = R/D$, where Q is the quotient, D is the divisor and R is the dividend. According to IEEE 754 standard,

$$R, D \quad \in \quad [1, 2). \tag{3.2.1}$$

For binary representations of R and D, performing the appropriate shift operations before the start of a division algorithm can satisfy these assumptions.

In general, digit-recurrence division algorithms can be described by a recurrence relation

$$r_{i+1} = 2 * r_i - q_i * D, \tag{3.2.2}$$

where i represents the iteration index and r_i is the remainder after the i-th iteration with initially $r_0 = R/2$, and q_i is the i-th quotient digit selected from the set $\{-1, 0, 1\}$. In each iteration, the algorithm doubles the remainder, then selects a quotient digit q_i, and subtracts $q_i * D$ from r_n. Alternatively, if we start with a different initialization $r_0 = R$, then we can use the recurrence relation

$$r_{i+1} = 2 * (r_i - q_i * D). \tag{3.2.3}$$

For an algorithm using the recurrence relation in (3.2.3), each repetition step starts with selecting a quotient digit q_i, then subtracting $q_i * D$, and finally doubling the result. This research assumes the latter recurrence relation (3.2.3) for describing all the algorithms.

Additionally, we require that the error interval of the computed quotient be less than one unit of least precision (ulp), where $ulp = 2^{-L}$ for some $L > 0$. In other words, if q is the computed quotient and the error, ϵ, is given by $\epsilon = q - R/D$,

then we require that $\epsilon \in (-ulp/2, ulp/2)$. Alternatively, the error interval may include one of the bounds, but not both bounds. For IEEE-754 single-precision format, $L = 23$, and for IEEE-754 double-precision format, $L = 52$.

3.3 Invariants and Termination

We can use recurrence relations and invariants to prove the correctness of the division algorithms and calculate the error in the computed quotient, similar to the one presented in [10]. The formula

$$Q * D = R \tag{3.3.1}$$

expresses the desired relation between Q, D, and R, where Q is the exact quotient. Lower-case variables q and r, represent the quotient calculated 'thus far,' and the remainder calculated 'thus far', respectively. The invariant for all the variables is

$$q * D + 2^{-i} * r = R, \tag{3.3.2}$$

where i is the iteration index and $q_i * 2^i$ is added to q in the ith iteration, where q_i is the quotient digit selected in ith iteration.

In addition, we also have a range invariant for the partial remainder which depends on the choice of a recurrence relationship and a division algorithm. The SRT algorithms that use the recurrence relation in (3.2.2) and a two's complement representation for the partial remainder, have a range invariant of

$$r = r_s + r_c \in [-D, D). \tag{3.3.3}$$

The SRT algorithms that use the recurrence relation in (3.2.3) and a two's complement representation for the partial remainder, have a range invariant of

$$r = r_s + r_c \in [-2D, 2D). \tag{3.3.4}$$

When we use a signed-digit representation for the partial remainder, the range invariant for the partial remainder excludes the lower bound, that is,

$$r = r_c - r_s \in (-D, D) \tag{3.3.5}$$

for recurrence relation in (3.2.2) and

$$r = r_c - r_s \in (-2D, 2D) \tag{3.3.6}$$

for recurrence relation in (3.2.3). Because the division algorithms in this document use the recurrence relation in (3.2.3), consider the invariants (3.3.4) and (3.3.6) for the SRT algorithms.

Some of the algorithms presented in this chapter have a different range invariant. The algorithms that use a two's complement representation for the partial remainder, $r = r_s + r_c$, have a range invariant of

$$r_s, r_c \in [-2, 2) \text{ and } r = r_s + r_c \in [-4, 4) \tag{3.3.7}$$

The algorithms that use a signed-digit representation for the partial remainder, $r = r_c - r_s$ have a range invariant of

$$r_s, r_c \in [0, 4) \text{ and } r = r_c - r_s \in (-4, 4) \tag{3.3.8}$$

We look for a number of program statements for the program variables q, r, and c that establish and maintain invariants (3.3.2), and (3.3.4) or (3.3.7) when two's complement representation is used for the partial remainder, or (3.3.6) or (3.3.8) when a signed-digit representation is used for the partial remainder. Once we have these program statements, we can then combine the statements in various ways to obtain a division algorithm.

The initialization `q=0; r=R; i=0` establishes invariant (3.3.2). Any quotient digit from an over-redundant digit-set $\{-2, -1, 0, 1, 2\}$ and the recurrence equation (3.2.3)

will maintain the invariant (3.3.2). The challenge is to choose a quotient digit that will maintain the range invariant (3.3.4) or (3.3.7) if the partial remainder is in a two's complement representation, or the range invariant (3.3.6) or (3.3.8) if the partial remainder is in a signed-digit representation.

Labels are used to denote the choice of a quotient digit and the statements executed to update the partial remainder according to (3.2.3), quotient, and the iteration index. Table 3.3.1 lists the labels corresponding to the choice of a quotient digit and the statements executed.

Table 3.3.1: Labels to denote the choice of a quotient digit and the statements executed.

Label	Quotient Digit, q_n	Statements executed
ADD2 & 2X	-2	r=2*(r+2D); q=q-2*2^{-n}; n=n+1
ADD1 & 2X	-1	r=2*(r+D); q=q-1*2^{-n}; n=n+1
2X	0	r=2*r; q=q-0*2^{-n}; n=n+1
SUB1 & 2X	+1	r=2*(r-D); q=q+1*2^{-n}; n=n+1
SUB2 & 2X	+2	r=2*(r-2D); q=q+2*2^{-n}; n=n+1

Now we need to make sure that the error in the computed quotient is small enough, that is $\epsilon \in [-ulp/2, ulp/2)$. Using invariant (3.3.2) we can express the error, ϵ, in the computed quotient as follows

$$\epsilon = \frac{R}{D} - q = 2^{-i} * \frac{r}{D} \in [\frac{-ulp}{2}, \frac{ulp}{2}). \tag{3.3.9}$$

For a given ulp and the range of the partial remainder, r, the expression in Equation. (3.3.9) translates into a condition that determines the value of i.

For example, if we consider $ulp = 2^{-L}$ and the range invariant (3.3.3), then from Equation 3.3.9, the termination condition becomes $i \geq L + 2$. The range invariant may also exclude the lower bound, that is, $(-2D, 2D)$ instead of $[-2D, 2D)$

If we consider the range invariants (3.3.7) or (3.3.8), then the termination condition is $i \geq L+3$. Consequently a division algorithm using the range invariants (3.3.7) or (3.3.8) requires one more iteration to obtain the same accuracy as a division algorithm using the range invariant (3.3.3) or (3.3.5).

3.4 SRT Algorithm

A standard radix-2 SRT algorithm uses a two's complement representation for the partial remainder, r, and carry-save additions (subtractions). The result of a carry-save addition is two numbers, r_s and r_c, whose sum is the actual value. Therefore, $r = r_s + r_c$, where r_s represents the sum or parity bits and r_c represents the carry or majority bits. The standard SRT algorithm has four non-fractional bits and carry-save addition is done modulo 2^4. More information on SRT algorithms can be found in [25, 13] and [17]. The selection of the quotient digit is based on the values of the four most-significant digits of the remainder in carry-save form r_s, r_c. Let $cpa(r_s, r_c)$ denote the result of a carry-propagate addition of only the four most significant digits of r_s and r_c. The algorithm selects a quotient digit q_i according to the following conditions

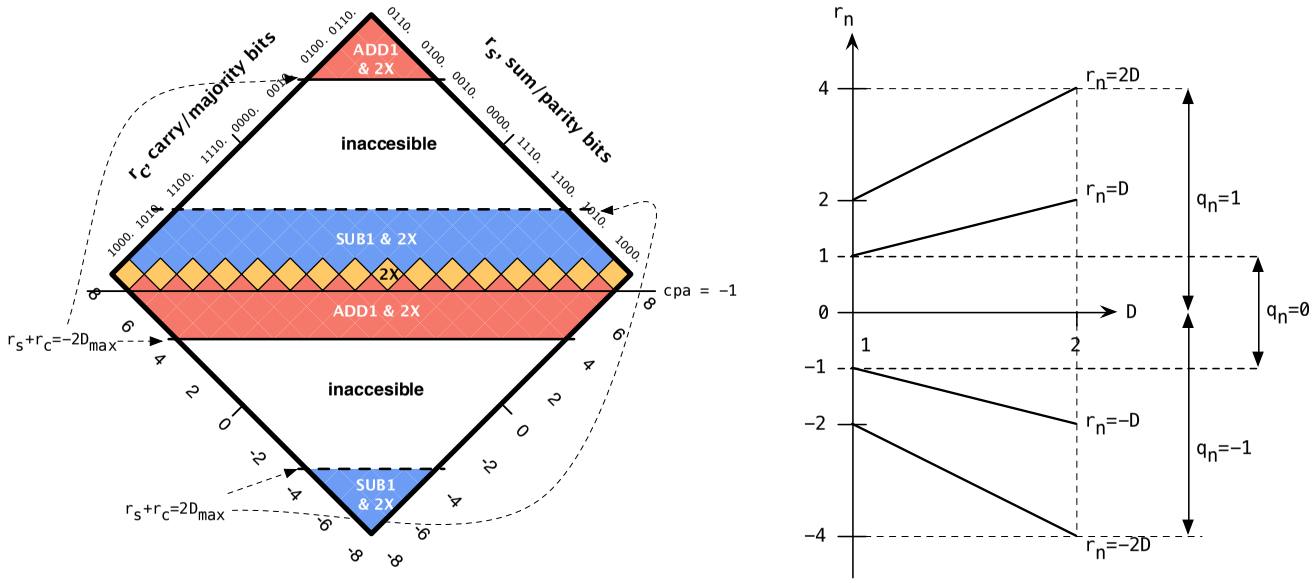
$$\begin{aligned} q_i &= 0 && \text{if } cpa(r_s, r_c) = \text{-1} \\ q_i &= \ + \ 1 && \text{if } cpa(r_s, r_c) > \text{-1} \\ q_i &= \text{-1} && \text{if } cpa(r_s, r_c) < \text{-1} \end{aligned}$$

Figures 3.4.1b and 3.4.1a show the quotient selection function in a conventional P-D diagram and in the diamond diagram respectively. In Figure 3.4.1b, the $x\text{-}axis$ represents the value of the divisor and the $y\text{-}axis$ represents the value of the partial remainder. Figure 3.4.1a show the quotient selection function in the diamond diagram. In the diamond diagram, the diagonal axes represent the value of r_s and r_c. The diagonal axes carry labels with both the four non-fractional bits of r_s and r_c in a two's complement representation (top) and the absolute value of r_s and r_c (bottom). The

vertical-axis represents the actual value of the partial remainder, $r = r_s + r_c$. A point in the diamond diagram has coordinates (r_s, r_c). Each horizontal line $r_s + r_c = r$ modulo 2^4 represents a set of points with the same remainder value.

The area labeled 2X is the area where $cpa(r_s, r_c) = -1$. For every remainder in this area, the SRT algorithm selects the quotient digit 0 and performs a doubling. The area labeled SUB1&2X is the area where $cpa(r_s, r_c) > -1$. For every remainder in this area, the SRT algorithm selects quotient digit 1 and performs a subtraction with D followed by a doubling. The area labeled ADD1&2X is the area where $cpa(r_s, r_c) < -1$. For every remainder in this area, the SRT algorithm selects quotient digit -1 and performs an addition with D followed by a doubling. Because addition is calculated modulo 2^4, the diagonal bands wrap around the square.

Because the SRT algorithm satisfies the invariant $r_s + r_c \in [-2D, 2D)$, only the colored areas are accessible. There are large inaccessible areas. In fact, at least half the area is inaccessible. These large inaccessible areas suggest that more efficient quotient selection functions can be derived.



(a) Standard radix-2 SRT algorithm in diamond diagram.

(b) P-D diagram for the standard radix-2 SRT algorithm.

Fig. 3.4.1: Standard radix-2 SRT algorithm. (a) the quotient selection areas (colored areas) for a standard radix-2 SRT division in the diamond diagram using only the 4 most-significant bits. In the diamond diagram, each point has coordinates (r_s, r_c). Each horizontal line $r_s + r_c = r$ modulo 2^4 represents a set of points with different r_s and r_c values but the same remainder value. Addition is modulo 2^4, so vertical-axis wrap around the diamond. For radix-2 SRT division, the remainder $r_s + r_c$ remains within the range $[-2D, 2D)$. (b) the P-D diagram for a standard radix-2 SRT algorithm. The $x\text{-}axis$ represents the value of the divisor and $y\text{-}axis$ represents the value of the partial remainder. In the figure $D_{max} = 2\text{-}ulp$.

3.5 Algorithms A1, A1b and A1c

The first set of algorithms assume that the partial remainder is in a two's complement representation and additions are carry-save additions. A two's complement representation of m non-fractional bits can represent numbers in the range $[-2^{m-1}, 2^{m-1})$. Note that the lower bound is inclusive while the upper bound is exclusive. Adding and subtracting numbers in two's complement arithmetic can be done by means of modulo 2^m carry-save addition . Therefore, to represent the initial value of $D \in [1, 2)$ and partial remainders r_s and r_c in the range $[-2, 2)$ we only need two non-fractional bits rather than four in the SRT algorithm of Figure 3.4.1.

Let us look at the effect of the addition and doubling operations on all the points that satisfy range invariant (3.3.7). The bold center-diamond, S0 to S15, in Figure 3.5.1 satisfies the range invariant (3.3.7). In the context of division algorithms, the diamonds with label are simply referred by their labels. For example, diamond S1 in Figure 3.5.1 is referred as S1. An addition or doubling a point in the center diamond can yield a point inside the center diamond or outside the center diamond. We can map the points that land outside the center diamond to the center diamond by means of a translation operation maintaining the range invariant (3.3.7).

Doublings and Translations

Figure 3.5.2 illustrates the effect of doubling any point in S6 and S9, and Figure 3.5.3 illustrates the effect of doubling a point in S0, S1, S4, S5, S10, S11, S14 and S15. Doubling a point (r_s, r_c) denotes multiplying a point (r_s, r_c) by 2. For example doubling a point $(-1, 1)$ yields a point $(-2, 2)$.

Doubling a point in S6 and S9 yields a point in S2∪S3∪S6∪S7 and S8∪S9∪S12∪S13 respectively, maintaining the range invariant (3.3.7). Doubling a point in S0∪S1∪S4∪S5 yields a point in bigger diamond Q0. In Q0, $r \in [-4, 4)$ but $r_s \in [-4, 0)$ and $r_c \in [0, 4)$

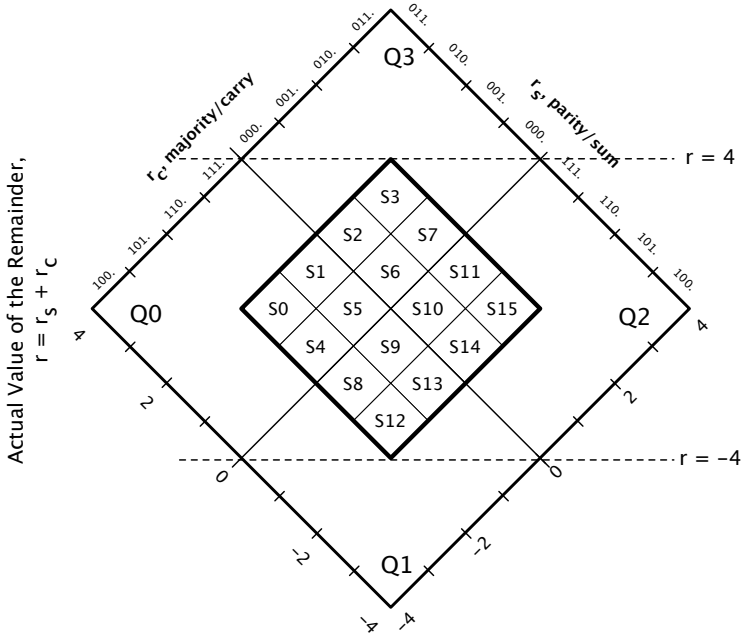


Fig. 3.5.1: The area of partial remainder (r_s, r_c) satisfying the range invariant (3.3.7). The value of the remainder, r, is $r = r_s + r_c$, where r_s and r_c are in a two's complement representation. The points (r_s, r_c) on a horizontal line, like $r_s + r_c = 4$, can have different values for r_s and r_c but have the same remainder value. The center diamond, S0 to S15, satisfies the range invariant (3.3.7) that is $r = r_s + r_c \in [-4, 4)$ and $r_s, r_c \in [-2, 2)$. The range of the partial remainder includes the lower bound and excludes the upper bound.

which violates invariant (3.3.7).

To maintain the range invariant (3.3.7), we need to map the points in the Q0 to the center diamond. A translation over $(2, -2)$ map the points in Q0 to the center diamond maintaining the the range invariant (3.3.7), as illustrated in Figure 3.5.4. Translation over $(2, -2)$ is adding 2 to r_s and subtracting 2 from r_c and keeps the value of $r_s + r_c$ unchanged. In fact, any translation of a point (r_s, r_c) over a distance $(t, -t)$ for any number t maintains the value of $r_s + r_c$. Because the translations involve addition and subtraction with a constant, a simple recoding of r_s and r_c can implement translations.

Any doubling of a point, (r_s, r_c) in S0∪S1∪S4∪S5 followed by a translation over $(2, -2)$ in effect yields a point in the center diamond. Similarly, doubling of a point

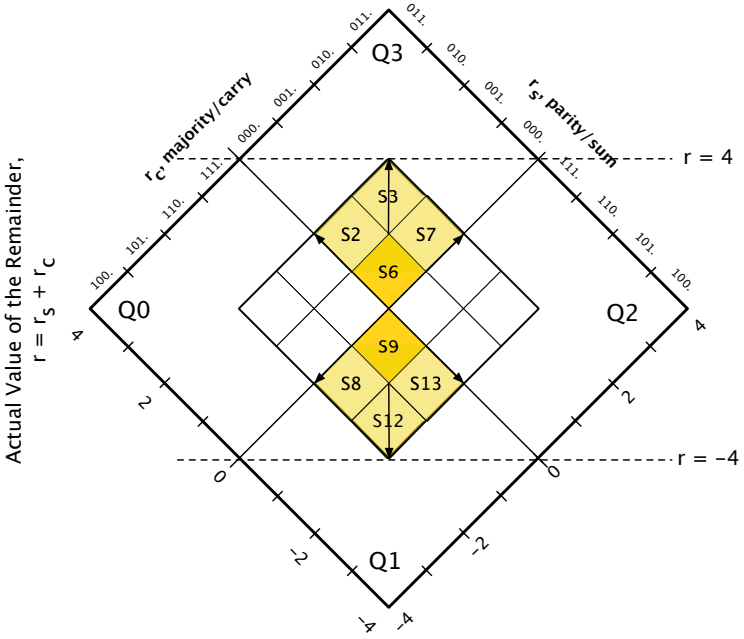


Fig. 3.5.2: The effect of doubling a point in diamonds S6 and S9. Doubling a point (r_s, r_c) denotes multiplying a point (r_s, r_c) by 2. For example doubling a point $(-1, 1)$ yields a point $(-2, 2)$.

in S10∪S11∪S14∪S15 followed by a translation over $(-2, 2)$ yields a point in the center diamond. In both the cases, a doubling followed by a translation will maintain invariant (3.3.2) and range invariant (3.3.7).

How do we implement these doublings and translations? Doublings can be implemented by left shifting the partial remainders r_s and r_c by one position. The translations over $(2, -2)$ and $(-2, 2)$ can be implemented by a simple recoding of the most-significant bits of r_s and r_c as follows.

$$\begin{aligned} 10 &\rightarrow 11 \\ 11 &\rightarrow 00 \\ 01 &\rightarrow 00 \\ 00 &\rightarrow 11 \end{aligned}$$

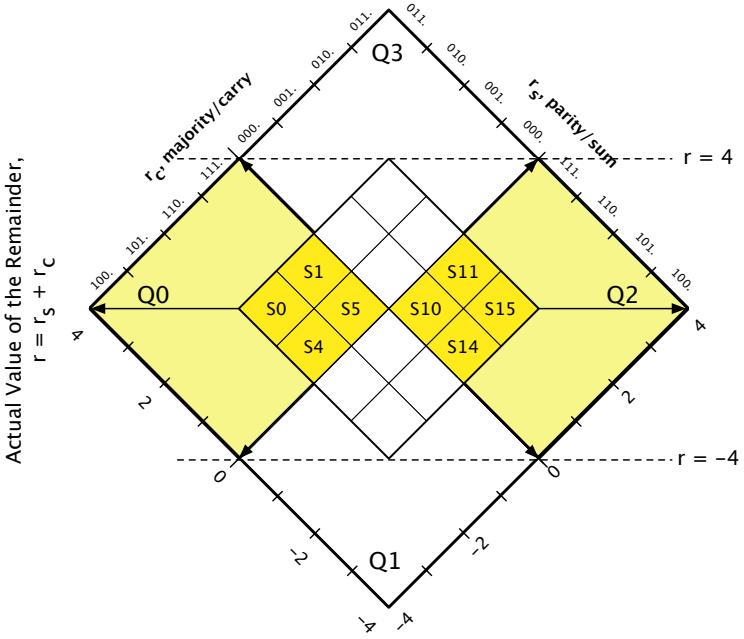


Fig. 3.5.3: The effect of doubling a point in diamonds S0, S1, S4, S5, S10, S11, S14 and S15

The second-most significant bit in each case changes and the most significant bit is a copy of the second-most significant bit.

If all operations start and end in the center diamond, we can apply some simplifications to the doubling and translation implementations. First, because the two most-significant bits of r_s and r_c are always the same in the center diamond, we can omit the most significant bit. Second, if we omit the most significant bit, a doubling followed by a translation of a point (r_s, r_c) in the center diamond simply becomes a left shift by one followed by an inversion of the most significant bit of both r_s and r_c. Because of the extra inversion of the most significant bit, the 2X* label denotes a doubling followed by a translation operation.

Here is an example of a doubling followed by a translation operation, 2X*. Consider a point (r_s, r_c) with three non-fractional bits $(000.u, 110.v)$, where u and v are some bit-sequences. Doubling the point $(000.u, 110.v)$ yields a point $(00?.u', 10?.v')$,

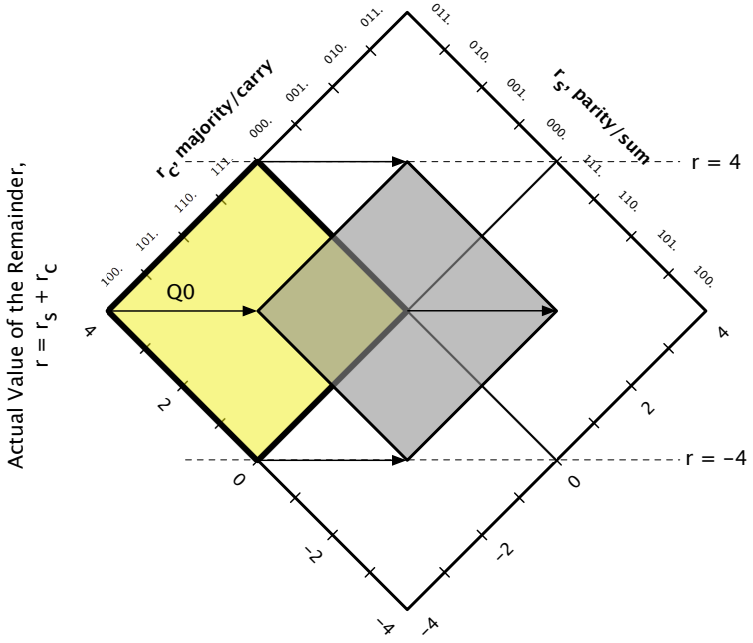


Fig. 3.5.4: The effect of translating Q0 over $(2, -2)$. Translation over $(2, -2)$ is adding 2 to r_s and subtracting 2 from r_c and keeps the value of $r_s + r_c$ unchanged.

where u' and v' are u and v left shifted by 1 position respectively, and ? represents a bit value of either 1 or 0 corresponding to the most-significant bit of u or v. Translation of the point $(00?.u', 10?.v')$ yields a point $(11?.u', 11?.v')$ in S8∪S9∪S12∪S13 of Figure 3.5.1.

Carry-Save Addition

Figure 3.5.5 shows the diamond diagram with center diamond partitioned into smaller diamonds. Consider subtracting D from a point (r_s, r_c) in the S1 diamond.

In a two's complement representation, $D = 001.x$, for some bit vector x, thus -D is represented by the bit-wise complement of D plus ulp, that is, -$D = 110.y + ulp$, where y is the bit-wise complement of x and ulp is the unit of least precision. The majority bits are left shifted by one position. Here is the calculation for a point in S1

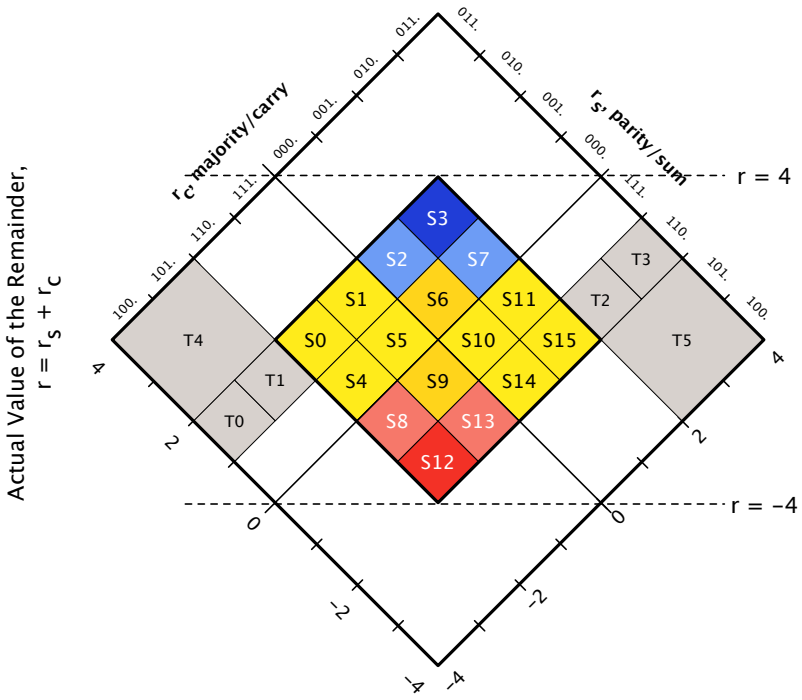


Fig. 3.5.5: The effects of carry-save additions and subtractions with D.

considering only the three most significant bits of each number.

$$
\begin{array}{rl}
r_s & \texttt{001.} \\
r_c & \texttt{111.} \\
\text{-}D & \texttt{110.y} \quad + \; ulp \\
 & \texttt{----------} \\
sum & \texttt{000.} \\
carry & \texttt{11?.}
\end{array}
$$

Following is the calculation for subtracting a point (r_s, r_c) in S2:

$$
\begin{array}{rl}
r_s & \texttt{001.} \\
r_c & \texttt{000.} \\
\text{-}D & \texttt{110.y} \quad + \; ulp \\
 & \texttt{----------} \\
sum & \texttt{111.} \\
carry & \texttt{00?.}
\end{array}
$$

Consequently, subtracting D from a point in S1 yields a point in S4∪S5. Subtracting D from a point in the S2 yields a point in S10∪S11. Because carry-save addition is symmetrical in r_s and r_c, subtracting D from a point in S11 also yields a point in S4∪S5.

The addition of D to a point in S4 or S14 yields a point in S10∪S11, and addition of D to any point in S8 or S13 yields a point in S4∪S5.

Table 3.5.1 gives a summary of adding and subtracting D from small diamonds. Subtraction is performed in the diamonds where the value of the partial remainder r is greater than 0. An addition is performed in the diamonds where the value of the partial remainder r is less than 0.

Table 3.5.1: The effect of subtracting or adding D

Origin	Destination after subtracting D	Origin	Destination after adding D
S1	S4∪S5	S4	S10∪S11
S2	S10∪S11	S8	S4∪S5
S3	T2∪T3	S9	S0∪S1
S6	S14∪S15	S12	T0∪T1
S7	S10∪S11	S13	S4∪S5
S11	S4∪S5	S14	S10∪S11

A subtraction of D from points in S1, S2, S6, S7, and S11 always ends in S0 ∪S1∪S4∪S5 or S10∪S11∪S14∪S15 of Figure 3.5.1. This means that any such point can subsequently undergo a doubling and a translation (ie. a 2X* operation) and land in the center square.

Diamond S3 is different. Subtraction of D from points in S3 yields a point in T2∪T3. Translating a point in T2∪T3 yields a point in S6∪S7, where the point must undergo another subtraction before a doubling. Instead, let us calculate what happens

when we subtract $2D$, instead of D, from any point in S3. First, recall that in a two's complement representation with 3 non-fractional bits $D = 001.bx$ for some bit b and bit vector x. Thus $2D = 01b.x0$, and -$2D$ is represented by the bit-wise complement of $2D$ plus ulp, that is, -$2D = 10d.y + ulp$, where d is the bit complement of b and y is the bit-wise complement of $x0$.

r_s	001.
r_c	001.
-$2D$	10d.y + ulp
	----------
sum	10?.
$carry$	01?.

Consequently, subtracting $2D$ from a point in S3 yields a point in T5 of Figure 3.5.5.

We can translate each point in T5 over $(2, -2)$ and the final result lands in S10∪S11∪S14∪S15 of Figure 3.5.1. Subsequently, for each point in S10∪S11∪S14∪S15 we can perform a doubling and translation (ie a 2X* operation) and obtain a point in the center diamond again.

We can make the same remarks for the additions of D or $2D$ to points in S4, S8, S9, S12, S13 and S14 of Figure 3.5.5. Addition of D a point in S4, S8, S9, S13, and S14 always yields a point inside S0∪S1∪S4∪S5 or S10∪S11∪S14∪S15 of Figure 3.5.1. This means that any such point can undergo a doubling and a translation and again land in the center diamond.

Addition of D to any point in S12 yields a point in T0∪T1. Translating a point in square T0∪T1 yields a point in S8∪S9 where a point must undergo another addition before a doubling. Adding $2D$, however, to any point in S12 yields a point in T4. Furthermore, T4 can be translated over $(-2, 2)$ to obtain a point in S0∪S1∪S4∪S5 of Figure 3.5.1. Doubling and translation of a point in S0∪S1∪S4∪S5 yields a point in the center diamond.

For each small diamond in the center square there is a sequence of operations that maintain invariants (3.3.2) and (3.3.7). Each sequence of operations ends with a doubling and a translation, which a subtraction or addition of D or $2D$ may be precede. For example, for a point in S12, the operations are an addition of 2D, followed by a translation, then a doubling and a translation. Some squares even have two possible sequences of operations that maintain invariant (3.3.2) and (3.3.7) (keeps a point in the center diamond of Figure 3.5.1). For example, for points in S4, the sequence of operations may be a doubling followed by a translation (2X*) or an addition of D followed by a doubling and a translation. The diamonds S1, S6, S9, S11, and S14 also have two possible sequences of operations.

By confining the points to the center diamond, we can omit the most-significant bit and consider the remaining two non-fractional bits. The operations are implemented as follows.

- Each addition is a carry-save addition in two's complement arithmetic.
- Each doubling is a left shift of both r_s and r_c by one position.
- Each translation is an inversion of the most-significant bit for r_s and r_c.

A translation followed by a doubling and then another translation is the same as a doubling followed by a translation, because each doubling throws away the most significant bit.

Putting Together the Division Algorithms

With the analysis of the previous section, we can put together various division algorithms. For each division algorithm we can specify what sequence of operations must be performed on the points (r_s, r_c) in each of the small diamonds, S0 to S15, in Figure 3.5.5. There are six sequences of operations to choose from which are described as follows:

- 2X: A doubling operation. The selected quotient digit is 0.
- 2X*: A doubling followed by a translation. The selected quotient digit is 0.
- SUB1&2X*: A subtraction of D followed by a doubling and then a translation. The selected quotient digit is +1.
- SUB2&2X*: A subtraction of $2D$ followed by a doubling and then a translation. The selected quotient digit is +2.
- ADD1&2X*: An addition of D followed by a doubling and then a translation. The selected quotient digit is -1.
- ADD2&2X*: An addition of $2D$ followed by a doubling and then a translation. The selected quotient digit is -2.

An inversion of the most-significant bit implements the translation over (2, -2) or (-2, 2).

Figures 3.5.6, 3.5.7 and 3.5.8 illustrate the three possible choices for a division algorithm. Other algorithms can be derived by making different choices for the diamonds S1, S4, S6, S9, S11 and S14. Algorithms A1, A1b and A1c are the symmetric choices. The selection of a quotient digit relies on only the two most-significant bits of r_s and r_c. Algorithm A1b has a simpler selection logic than Algorithms A1 and A1c, which can lead to a faster divider implementation.

Fig. 3.5.6: Algorithm A1. (a) the quotient selection areas using the diamond diagram and (b) P-D diagram.

Fig. 3.5.7: Algorithm A1b: (a) the quotient selection areas using the diamond diagram and (b) P-D diagram.

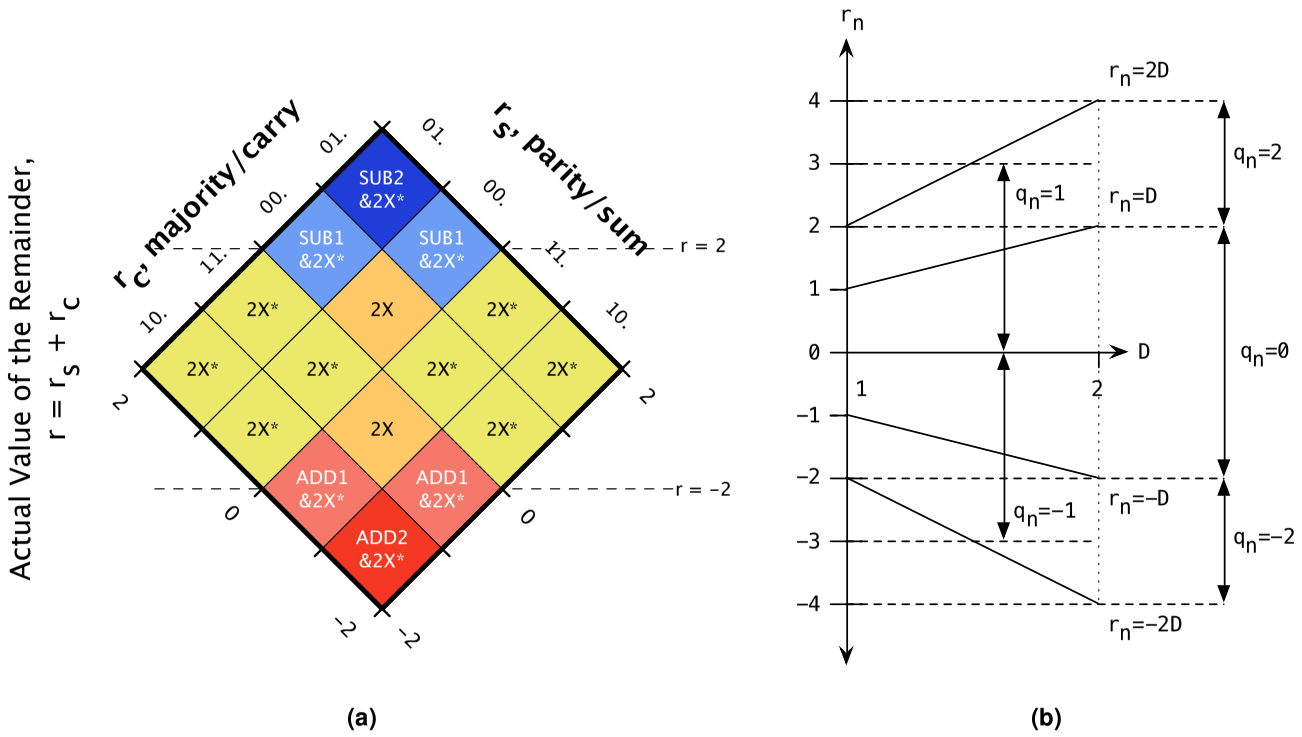


Fig. 3.5.8: Algorithm A1c: (a) the quotient selection areas using the diamond diagram and (b) P-D diagram. This is the same algorithm presented in [10]

A Different Range Invariant

Thus far, we have seen that algorithms A1, A1b and A1c maintain the range invariant (3.3.7) for the partial remainder, that is, $r = r_s + r_c \in [-4, 4)$. Following is the proof that algorithm A1 maintains a more conservative range invariant (3.3.4), that is, $r = r_s + r_c \in [-2D, 2D)$.

First, the range invariant (3.3.4) holds after initialization $r_s = R$; $r_c = 0$ for $R, D \in [1, 2)$.

Second, each of the operations ADD2&2X*, ADD1&2X*, 2X*, SUB1&2X*, and SUB2&2X* maintains the range invariant 3.3.4. For the proof, assume that the invariant (3.3.4) holds before each of those five sequences of operations.

In the regions of Figure 3.5.6 where algorithm A1 executes the SUB1&2X* operations, the range of the partial remainder is

$$r = r_s + r_c \in [0, 2D). \tag{3.5.1}$$

After subtracting D from $r_s + r_c$ and doubling r_s and r_c, the range of the partial remainder is

$$r = r_s + r_c \in [-2D, 2D), \tag{3.5.2}$$

which is the range invariant (3.3.4).

In the regions of Figure 3.5.6 where algorithm A1 executes the SUB2&2X* operations, the range of the partial remainder is

$$r = r_s + r_c \in [2, 2D). \tag{3.5.3}$$

After subtracting $2D$ from $r_s + r_c$ and doubling r_s and r_c, the range of the partial remainder is

$$r = r_s + r_c \in [2(2 - 2D), 0). \tag{3.5.4}$$

The lower bound $2(2 - 2D) = (4 - 2D) - 2D > -2D$, because $4 - 2D > 0$ for $D \in [1, 2)$. Consequently, after the operations SUB2&2X*, we have $r = r_s + r_c \in$

$[-2D, 2D)$. Replacing additions with subtractions, we can prove that the ADD1&2X* and ADD2&2X* operations also maintain invariant (3.3.4).

Finally, in the regions of Figure 3.5.6 where algorithm A1 executes the 2X* operation, the range of the partial remainder is, $r = r_s + r_c \in [-1, 1)$. For $D \in [1, 2)$, $r = r_s + r_c \in [-D, D)$. Consequently, after the 2X* operation, $r = r_s + r_c \in [-2D, 2D)$, satisfying the range invariant 3.3.4.

From the discussion in Section 3.3, because algorithm A1 maintains the range invariant (3.3.4), algorithm A1 must execute one fewer iteration than algorithms A1b and A1c. In other words, algorithm A1 must execute $L + 3$ iterations per division whereas algorithms A1b and A1c must execute $L + 4$ iterations per division, including an extra quotient digit that may be required for normalization.

3.6 Comparison of Division Algorithms

This section evaluates the division algorithms considered for this research for latency per division, average energy per division, area, and the fraction of shift-only and addition operations per division. A shift-only operation is one of the 2X or 2X* alternatives. An addition operation is one of the following four alternatives: ADD1& 2X*, ADD2& 2X*, SUB1& 2X* and SUB2& 2X*.

For a fair comparison, I synthesized the behavioral verilog code for all the algorithms using a TSMC 40nm standard cell library with the same compiler settings. Table 3.6.1 summarizes some of the important characteristics of the division algorithms considered for this research. The derivation of algorithms B1, B1b and B1c is in Appendix A.

Table 3.6.1: Summary of the characteristics of the division algorithms. For IEEE 754 double-precision format, $L = 52$.

Algorithm	Number of Alternatives per Iteration	Type of Addition	Number of Iterations per division, N
SRT	3	Carry-save	$L+3$
A1	5	Carry-save	$L+3$
A1b	5	Carry-save	$L+4$
A1c [10]	6	Carry-save	$L+4$
B1 [33]	5	Carry-free	$L+3$
B1b	5	Carry-free	$L+4$
B1c	6	Carry-free	$L+4$

Tables 3.6.2 lists the latency per division, average energy per division and the area consumption from a static timing-analysis tool for all the division algorithms. Algorithms A1b and B1b offer an improvement of about 8% in latency per division compared to the standard radix-2 SRT algorithm. This improvement comes at the cost of 28% and 47% more energy and area consumption. The standard radix-2 SRT

algorithm must choose from one of the three alternatives each iteration, but the rest of the algorithm must choose from one of the five or six alternative each iteration. Hence the radix-2 SRT algorithm consumes less energy and area. Compared to algorithms A1, A1c, B1 and B1c, algorithms A1b and B1b offer a modest improvement of 4% in latency per division.

Table 3.6.2: Comparison of the division algorithms for latency per division, energy per division and area.

Algorithm	**Latency per Iteration,** L_{iter} **in** ps	**Latency per Division,** L_{div} **in** ns	**Average Energy per Division,** E_{div} **in** pJ	**Area in** μm^2
SRT	250	13.75	5.7	7900
A1	240	13.20	7.3	11383
A1b	225	12.60	7.3	11676
A1c	235	13.16	7.5	11772
B1	240	13.20	7.3	11376
B1b	225	12.60	7.3	11673
B1c	235	13.16	7.5	11768

Table 3.6.3 lists the fraction of shift-only and addition operations per division obtained from two million simulations with a pair of uniform-random operands. From the point-of-view of developing a methodology for evaluating data path topologies for an asynchronous conditional statement, an algorithm that has approximately the same fraction of shift-only and addition operations per division is a good candidate. From Table 3.6.3, Algorithms A1b, A1c, B1b and B1c are equally good candidates and this research uses algorithm A1b.

Finally, we can convert a quotient digit from a redundant set {-1, 0, 1} or {-2, -1, 0, 1, 2} to a unique binary representation by means of on-the-fly conversion presented in [12, 13] and [25]. Appendix B shows how we can implement on-the-fly conversion using the method of invariants.

Table 3.6.3: Fraction of shift-only and addition operations per division.

Algorithm	Fraction of shift-only operations per division	Fraction of addition operations per division
SRT	0.35	0.65
A1	0.35	0.65
A1b	0.53	0.47
A1c	0.57	0.43
B1	0.35	0.65
B1b	0.53	0.47
B1c	0.57	0.43

4

Evaluation of Datapath Topologies

A circuit designer is often baffled by an array of choices for designing circuits to satisfy various constraints. What design style to choose? Which data path topology to implement? This chapter presents a methodology to evaluate the delay of the data path topologies for self-timed designs implementing an asynchronous conditional statement.

The following is the difference between evaluating the data path topologies for synchronous and self-timed designs. The synchronous designs always consider the worst-case delay but the self-timed designs consider both the worst and best case delays. Therefore, the self-timed designs have an average-case behavior where the average depends on the difference between the worst and best case delays, and the frequency of the worst and best cases. Frequency of the worst and best case delays depend on the workload, and the difference between the worst and the best delays depend on the data path topology.

This research uses the division algorithm A1b presented in Chapter 3 as an example of an asynchronous conditional statement. Hence this chapter uses the data path topologies that implement the `if-then-else` conditional statements in algorithm A1b. In a division algorithm, the best case is when the algorithm executes a

shift-only operation and the worst case is when the algorithm executes an addition operation. To calculate the frequency or the fraction of shift-only and addition operations per division, I assumed a pair of random input-operands.

The methodology proposed in this chapter evaluates the average-case delay of the data path topologies for the self-timed divider design considering the fraction of shift-only and addition operations per division. For comparison this chapter also evaluates the worst-case delay of the data path topologies for the synchronous divider design. After the evaluation of the data path topologies, I identify one data path each for the synchronous and self-timed designs that has an optimum delay for both the designs. The evaluation shows that the synchronous design prefers a fully-speculative design but the self-timed design prefers a less-speculative design.

4.1 Evaluation Methodology

The division algorithm A1b executes one of the following five alternatives every iteration: SUB2&2X*, SUB1&2X*, ADD1&2X*, ADD2&2X* and 2X*. The four alternatives, SUB2&2X*, SUB1&2X*, ADD1&2X* and ADD2&2X* require a carry-save addition followed by a shift. The 2X* alternative is a shift-only operation.

A data path topology for algorithm A1b has the following three paths: add, shift and select paths. An add path executes one of the four addition alternatives, the shift path executes the shift-only 2X* alternative and the select path selects the result from an add path or the shift path.

For the synchronous designs the delay of the data path is

$$D_{sync} = MAX(D_{sel},\ D_{add},\ D_{shift}), \tag{4.1.1}$$

where D_{sel}, D_{add} and D_{shift} are the delay of select, add and shift paths, respectively. The delay of the data path is also the cycle time of an iteration or the clock-period for the synchronous design.

A self-timed design can take advantage of the faster shift-only operation and we can estimate the average-case delay of the data path as follows

$$\begin{aligned} D_{async} = \; & N_{add} * MAX(D_{sel}, D_{add}) \\ & + N_{shift} * MAX(D_{sel}, D_{shift}), \end{aligned} \tag{4.1.2}$$

where N_{add} and N_{shift} are the fraction of the addition operations per division and the fraction of the shift-only operations per division. From Table 3.6.3 in Section 3.6, $N_{add} = 0.47$ and $N_{shift} = 0.53$ for algorithm A1b. For a self-timed design the term $MAX(D_{sel}, D_{add})$ in Equation (4.1.2) sets the period of the self-timed synchronization pulse for an addition operation, also referred to as add period. The term $MAX(D_{sel}, D_{shift})$ in Equation (4.1.2) sets the period of the self-timed synchronization pulse for a shift-only operation also referred to as shift period.

The evaluation of data path topologies for the synchronous and self-timed designs use equations (4.1.1) and (4.1.2), respectively. The delay estimated for synchronous designs is deterministic, but the delay estimated for self-timed designs is statistical. The method of logical effort is used to estimate the delay of the data paths.

4.2 Logical Effort Preliminaries

The *method of logical effort* estimates the delay of circuits and is effective for evaluating various design alternatives early in the design cycle [37, 39]. This section presents a brief introduction to the method of logical effort.

The method of logical effort statically estimates the delay of a logic gate by capturing the electrical environment of a gate, the drive capability of the logic gate, and the gate's topology.

The method of logical effort uses a linear delay model and describes the delay of a logic gate as the sum of two delays. The first delay is called the $effort\ delay$, and is proportional to the output load. The second delay is called the $parasitic\ delay$, p,

and is the intrinsic delay of the logic gates. The effort delay depends on the electrical environment and the drive capability of the logic gate. The $logical\ effort$, g, of a gate captures the ability of a gate to drive the output load. The $electrical\ effort$, h, captures the electrical environment of a gate. Thus the delay of a gate is expressed as

$$d = gh + p \tag{4.2.1}$$

and the electrical effort is

$$h = \frac{C_{out}}{C_{in}} \tag{4.2.2}$$

where C_{out} is the output capacitance of the gate and C_{in} is the input capacitance of the gate.

When analyzing multi-stage logic networks, $branching\ effort$ captures the effect of branches or fanout within a multi-stage network on delay. The branching effort b at the output of a logic gate is:

$$b = \frac{C_{on\text{-}path} + C_{off\text{-}path}}{C_{on\text{-}path}} \tag{4.2.3}$$

where $C_{on\text{-}path}$ is the input capacitance of the next logic gate along the path of analysis and $C_{off\text{-}path}$ is the sum of capacitances of the logic gates in the off path.

When analyzing multi-stage logic networks, $path\ effort$, F, captures the stage effort of the logic gates in the path as follows:

$$\begin{aligned} F &= GBH \\ \text{where, } G = \prod g_i,\ B &= \prod b_i \ \text{ and } \ H = \frac{C_{out}}{C_{in}} \end{aligned} \tag{4.2.4}$$

The subscript i indexes the logic stages along the path, and G, B and H are called path logical effort, path branching effort and path electrical effort, respectively.

The *path parasitic delay*, P, is the sum of all the parasitic delays of the gates in the path.

$$P = \sum p_i \tag{4.2.5}$$

For a path with N stages, the path delay is minimum when each stage bears the same stage effort, that is,

$$\hat{f} = F^{1/N} \tag{4.2.6}$$

The symbol $\hat{f}$ denotes an optimal value. Additionally, from [37], we know that any value from 2.4 to 6 for the stage effort gives optimal result and a value of 4 is a good choice. To get $\hat{f} = 4$, a path with path effort F, requires $\hat{N} = \log_4 F$ stages. In a path with N stages, where $N < \hat{N}$, inserting additional amplification stages reduces the delay of the path.

The delay of an $\hat{N}$-stage path is

$$D = \hat{N}F^{1/\hat{N}} + P \tag{4.2.7}$$

Expression in (4.2.7) is the key result of Logical Effort which allows the designer to estimate the delay of a given path without knowing the actual sizes or the drive strengths of the gates. The unit of delay estimated using equation (4.2.7) is τ. We can also express the delay in terms of fan-out-of-4 “FO4" inverter delays by dividing by 5 because one FO4 is approximately equal to 5τ. In this document, the delays estimated using the method of logical effort are expressed in terms of FO4 inverter delays.

Equation in (4.2.8) gives the capacitance transformation formula to calculate gate sizes. I have used the capacitance transformation formula to estimate the branching effort, b, when evaluating various data path alternatives.

$$C_{in_i} = \frac{g_i C_{out_i}}{\hat{f}} \tag{4.2.8}$$

Table 4.2.1 lists the logical effort and parasitic delay of the standard gates. The numbers in the table and the logical effort calculations in this chapter assume the following:

- The ratio of the sizes of the PMOS and NMOS transistors in a minimum sized inverter is two-to-one.

- The drain and gate capacitances of a transistor are the same.

Table 4.2.1: Logical effort and parasitic delay of standard gates [37]

Gate Type	Logical Effort Number of inputs				Parasitic Delay Number of inputs			
	1	2	3	4	1	2	3	4
INVERTER	1				1			
NAND		4/3	5/3	6/3		2	3	4
NOR		5/3	7/3	9/3		2	3	4
LATCH	2				2			
XOR, XNOR		4,4	6,6,12			4	6	
MAJORITY			2,4,4				6	

4.3 Data Path Modules

This section presents the description of some of the key components of the divider data paths in Section 4.4. The key components are carry-save adder, quotient-selection logic (QSLC) and multiplexers with different fan-in.

Carry-save Adder

Figure 4.3.1 shows a one-bit carry-save adder with three inputs and two outputs. The figure also carries the label denoting the logical effort for the corresponding input. There are different input configurations for the carry-save adder and the configuration in the figure distributes the logical effort more evenly. A carry-save adder consists of parity and majority circuits to produce parity and majority bits, par and maj, respectively. The majority bits are always left shifted by one position and therefore the inputs a, b and c at position j produce a majority bit at position $j+1$. The carry-save adder in Figure 4.3.1 requires inputs in both true and complement form. The amplification

stages that may be inserted to drive the carry-save adders can potentially use 0-1 or 1-2 forks to provide both true and complement signals. Therefore, we can defer considering the details of generating true and complement signals until later.

Table 4.3.1 lists the logical effort of parity and majority gates. The parasitic delays for both majority and parity gates, p_{maj} and p_{par} are 6. Because the input bundle $a*$ has higher logical effort than input bundles $b*$ and $c*$, connecting a signal that arrives early to the input bundle $a*$ would be prudent. In some data paths the divisor arrives first and in other data paths remainder arrives first. When evaluating data paths, I will make explicit what signals connect to which input of a carry-save adder.

When evaluating data paths, to eliminate the need to keep track of individual gates, it is useful to consider a carry-save adder as a single gate rather than two gates. We can use the path effort of the carry-save adder cell to treat the carry-save adder cell as a single gate. If both the parity and majority circuits drive the same output load, then the path effort for an input of the carry-save cell is the sum of the logical efforts of the parity and majority circuits for that input.

Considering the input bundle $c*$ and cases when both the parity and majority circuits drive the same output load, that is, $L_{par} = L_{maj} = L$, the branching effort b for a path through the parity circuit is

$$b = (6 * L + 4 * L)/(6 * L) = 5/3 \tag{4.3.1}$$

Assuming $H = 1$, the path effort is

$$F = GB = 6 * 5/3 = 10 \tag{4.3.2}$$

If we consider a path through the majority circuit, the path effort turns out to be 10. Following the calculation for the input bundle $c*$, for input bundles $a*$ and $b*$ the path effort is 14 and 10 respectively. Thus, when the parity and majority circuits drive the same load we can calculate the path effort of a carry-save adder by simply adding the

logical effort of the parity and majority gates. We can extend this reasoning to simple multi-stage modules such as the quotient selection logic, multiplexers etc.

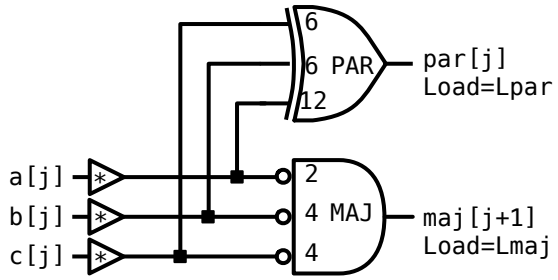


Fig. 4.3.1: One bit carry-save adder consisting of a parity and a majority circuit.

Table 4.3.1: Logical efforts of inputs for asymmetric parity and majority gates.

(a) Majority

Input	Logical effort, g_{maj}
a	2
b	4
c	4

(b) Parity

Input Bundle	Logical effort, g_{par}
$a*$	12
$b*$	6
$c*$	6

Quotient Selection Logic

The quotient selection logic or QSLC module is a key component in the divider design. The QSLC module accepts the two most significant bits of the partial remainder in carry-save form, r_s and r_c, and produces the select signals for the multiplexers in the data path. The quotient selection logic differs for data paths with different multiplexer organizations. For example, the QSLC modules for the two data paths T1D1 and T2D5 in Figures 4.4.3 and 4.4.4, respectively, are different because of the difference in the multiplexer organization. But the QSLC modules for the data paths T1D2

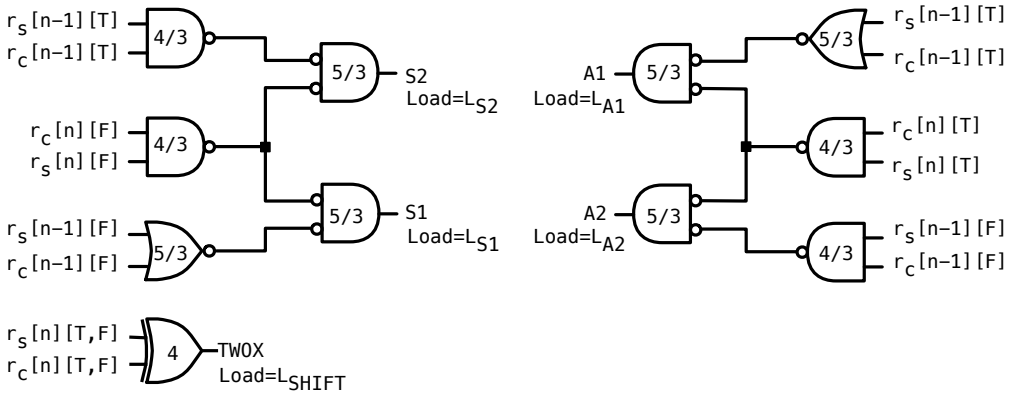
and T1D4, or T1D1 or T2D1 (see Appendix C) are the same because they have the same multiplexer organization. Therefore, we need three different QSLC modules and Figure 4.3.2 shows the gate-level implementation of all three QSLC modules. In the figures, indices [n] and [n-1] denote the most and second-most significant bits, respectively, and indices [T] and [F] denote true and false (complement) signals respectively. The data path T1D1 uses the QSLC in Figure 4.3.2a and data path T2D5 uses the QSLC in 4.3.2b. Following is the description of the QSLC signals:

1. $S2$: Selects the result from SUB2 & 2X* module or -2D for carry-save addition.
2. $S1$: Selects the result from SUB1 & 2X* module or -D for carry-save addition.
3. $A1$: Selects the result from ADD1 & 2X* module or D for carry-save addition.
4. $A2$: Selects the result from ADD2 & 2X* module or 2D for carry-save addition.
5. $TWOX$: Selects the result from the 2X* module.
6. ADD: Selects the result from the 4:1 multiplexer in data paths with D2 label or selects the result from CSA & 2X* module in data paths with D5 label.
7. A: Selects the result from ADD1 & 2X* or ADD2 & 2X* module in data paths with D3 label, or ADD & 2X* module in data paths D4 label.
8. S: Selects the result from SUB1 & 2X* or SUB2 & 2X* module in data paths with D3 label, or SUB & 2X* module in data paths with D4 label.

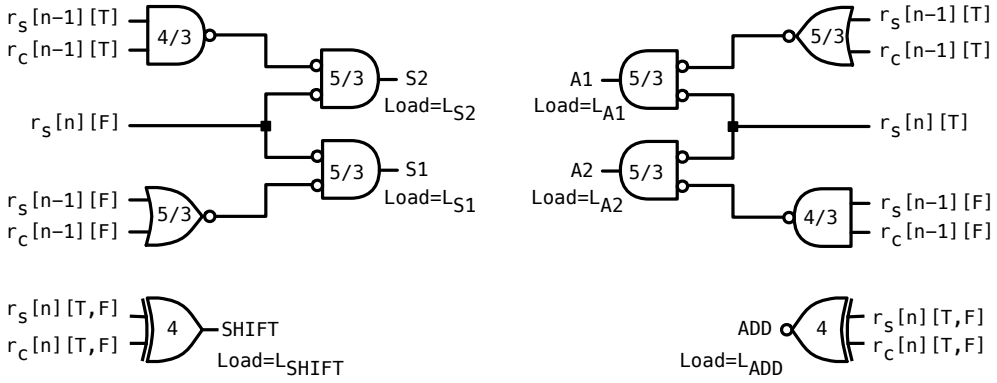
In Figures 4.3.2a, 4.3.2b and 4.3.2c, L_{S1}, L_{S2} etc., are the loads presented for the corresponding signals. Assuming that all the output signals of a QSLC design drive the same load, we can estimate the logical effort of the QSLC inputs. Table 4.3.2 lists the logical effort, g, and parasitic delay, p, of the input bundles for all three quotient selection logic modules.

Table 4.3.2: Summary of Logical Effort and Parasitic Delay of the three quotient selection logics.

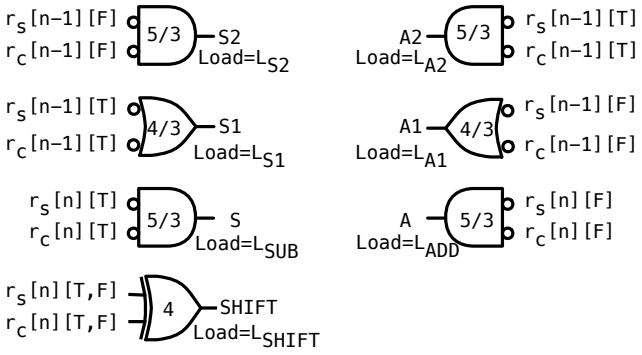
Design	Inputs	Logical Effort, g	Parasitic Delay, p
1	$r_s[n]*, r_c[n]*$	12.88	4
	$r_s[n\text{-}1]*, r_c[n\text{-}1]*$	10	4
2	$r_s[n]*, r_c[n]*$	11.33	4
	$r_s[n\text{-}1]*, r_c[n\text{-}1]*$	10	4
3	$r_s[n]*, r_c[n]*$	7.33	4
	$r_s[n\text{-}1]*, r_c[n\text{-}1]*$	6	4



(a) QSLC 1: This QSLC module used in data paths T1D1, T2D1 and T3D1



(b) QSLC 2: This QSLC module used data path T1D2, T1D5, T2D2, T2D5, T3D2 and T3D5

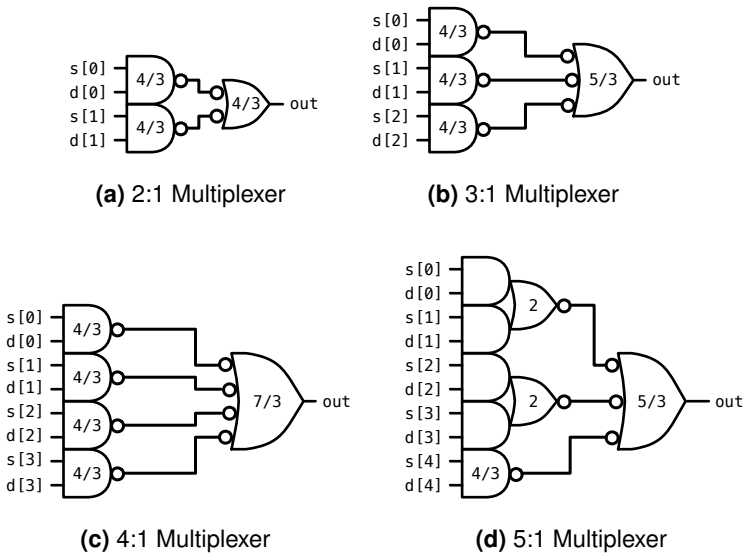


(c) QSLC 3: This QSLC module is used in data paths T1D3, T1D4, T2D3, T2D4, T3D3 and T3D4.

Fig. 4.3.2: Gate-level design of the quotient selection logic for various data paths.

Multiplexers and Latches

Figure 4.3.3 shows the gate-level structures of the multiplexers used for data path evaluation. In the figure, the select inputs carry labels $s[0]$ to $s[4]$ and the data inputs carry labels $d[0]$ to $d[4]$. Table 4.3.3 lists the logical effort and parasitic delay associated with each of the inputs.



(a) 2:1 Multiplexer **(b)** 3:1 Multiplexer

(c) 4:1 Multiplexer **(d)** 5:1 Multiplexer

Fig. 4.3.3: Multiplexers with different fan-in.

Table 4.3.3: Input logical effort and parasitic delay of multiplexers.

(a) Input logical effort and parasitic delay of 5:1 and 4:1 multiplexers.

Inputs	**5:1 Mux**		**4:1 Mux**	
	Logical effort, $g_{5:1Mux}$	**Parasitic delay**, $p_{5:1Mux}$	**Logical effort**, $g_{4:1Mux}$	**Parasitic delay**, $p_{4:1Mux}$
$s[0]$ to $s[3]$, $d[0]$ to $d[3]$	3.33	7	2.67	6
$s[4]$, $d[4]$	2.22	5		

(b) Input logical effort and parasitic delay of 3:1 and 2:1 multiplexers.

Inputs	**3:1 Mux**		**2:1 Mux**	
	Logical effort, $g_{3:1mux}$	**Parasitic delay**, $p_{3:1mux}$	**Logical effort**, $g_{2:1mux}$	**Parasitic delay**, $p_{2:1mux}$
$s[0]$, $s[1]$, $d[0]$, $d[1]$	2.22	5	1.78	4
$s[2]$, $d[2]$	2.22	5		

4.4 Data Path Topologies

Figure 4.4.1 shows the basic architecture of a two-stage pipeline that implements the division. In the figure, `RxReg`, `RecReg` and `TxReg` are the registers. The register `RxReg` receives the new data operands from FIFO-A and the register `TxReg` passes the output result for post-processing, for example, rounding, normalization etc to FIFO-B. The register `RecReg` is the recurrence register. The module with label `init` denotes the initialization of the different registers. The module with label

RecDP denotes a data path that implements the if-then-else statements in algorithm A1b. The 2-input multiplexer selects the data from either the init module or from the RecDP module. The path from the clock input of the register RecReg to the data-input of the register RecReg is the critical path and sets the clock-period for the pipeline in case of a synchronous design A self-timed synchronization pulse replaces the clock in a self-timed design. A self-timed design modulates the period of the synchronization pulse to the register RecReg according to the delay of an addition or shift-only operations in the RecDP module.

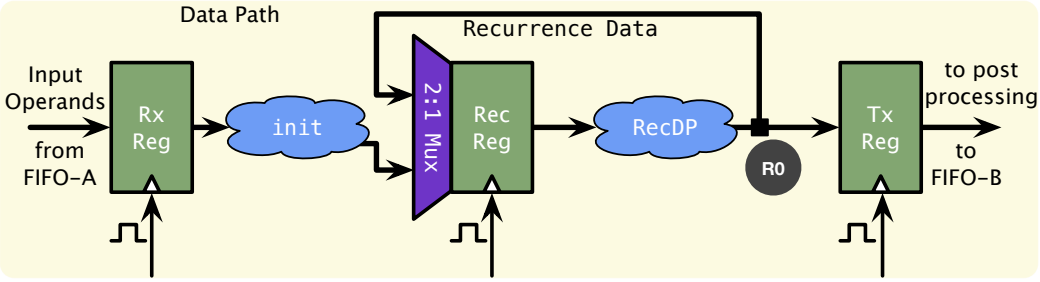


Fig. 4.4.1: A two-stage pipeline that implements the recurrence loop

There are fifteen different candidates for the data path. The fifteen data paths can be classified into three different topologies of five different data paths each. The five different data paths in a topology result from different multiplexer organizations. The classification of topologies is based on when during the clock period, the quotient selection logic (QSLC) is computed, as illustrated in Figure 4.4.2. Based on the two most-significant bits of the partial remainder, the quotient selection logic determines the quotient digit accumulated each iteration. In topologies 1, 2 and 3, the QSLC is computed at the beginning, middle and the end of the clock period respectively.

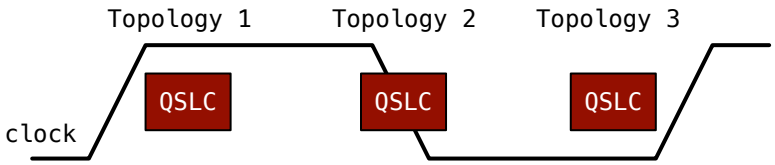


Fig. 4.4.2: Classification of the data path topologies based on when during the clock period, the quotient selection logic (QSLC) is computed. In topologies 1, 2 and 3, the QSLC is computed at the beginning, middle and the end of the clock period respectively.

The following four letter naming convention denotes different data-path topologies. The first two letters denote the topology and the last two letters denote one of the five data paths in that topology. For example, T1D1 denotes data path 1 in topology 1.

The Sections 4.4.1 and 4.4.2 show the logical effort calculations for the data paths T1D1 and T2D5 respectively. The logical effort calculations for the remaining data-path topologies appear in Appendix C.

4.4.1 Data Path T1D1

Figure 4.4.3 shows the data path T1D1. The data path T1D1 computes the QSLC at the beginning of the clock cycle, hence topology 1. In the figure, index [i] denotes the iteration index and all the signals originate at the output of registers and end at the input of the 2:1 multiplexer in Figure 4.4.1. This is a fully-speculative design because all five alternatives are executed speculatively and then the QSLC module chooses the correct result.

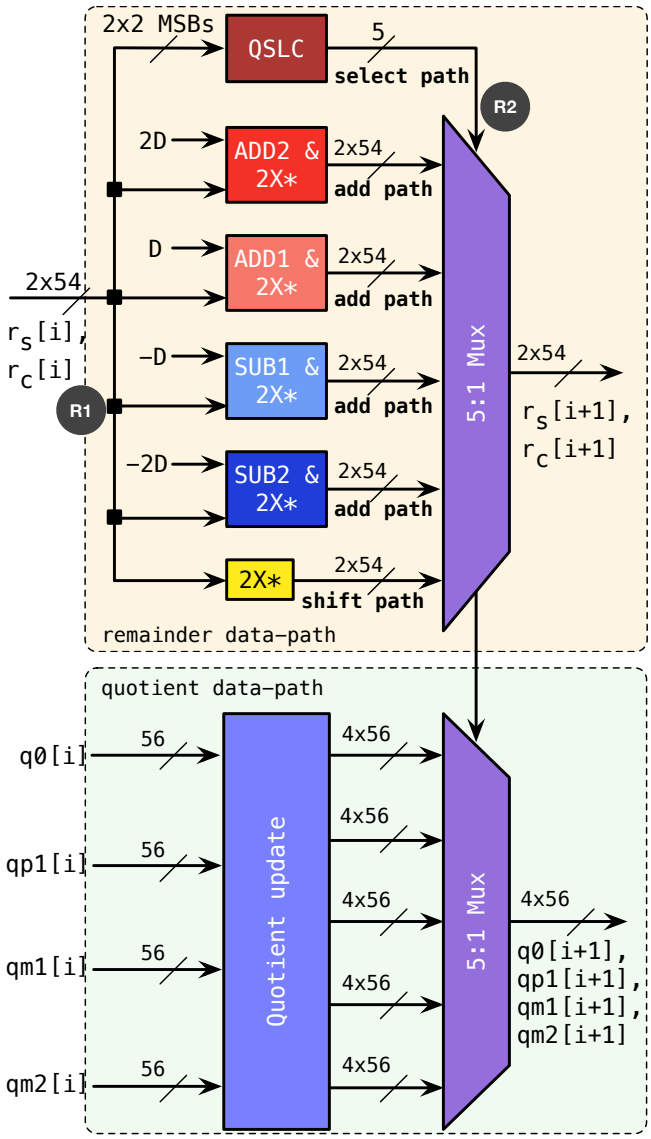


Fig. 4.4.3: Data path T1D1 with the quotient data path. This is a fully-speculative design because the design executes all five alternatives and then selects the correct alternative. This data path topology computes the quotient selection logic at the beginning of the clock period, hence topology 1.

Consider the following three paths to estimate the delay of the data path T1D1:

Add path: This is the path of the lesser-significant bits of the partial remainder through one of the four carry-save adders, that is, `register` $\rightarrow$ `ADD2&2X* or ADD1&2X* or SUB1&2X* or SUB2&2X*` $\rightarrow$ `5:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `register`. Add path executes one of the addition operations.

Shift path: This is the path of the lesser-significant bits of the partial remainder through the 2X* module, that is, `register` $\rightarrow$ `2X*` $\rightarrow$ `5:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `register`. Shift path executes the shift-only operation.

Select path: This is the path of the two most-significant bits of the partial remainder, r_s and r_c, through the QSLC module, that is, `register` $\rightarrow$ `qslc` $\rightarrow$ `5:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `register`. Select path selects the result from an add path or the shift path.

In the above paths, the last 2:1 multiplexer selects the data from the recurrence loop or the new operands in Figure 4.4.1.

Table 4.4.1 lists the logical effort and parasitic delay of the logic gates in the select, add and shift paths along with the number of stages in each gate. Table 4.4.2 lists the branching effort at various nodes in the select, add and shift paths.

Let D_{sel}, D_{add} and D_{shift} be the delays of the select, add and shift paths respectively. Using the values of G, P, and N from Table 4.4.1, and B from Table 4.4.2, the delay of the select, add and shift paths are as follows. The letters G, B, P and N denotes the path logical effort, path branching effort, total parasitic delay and total number of stages, respectively.

$$\begin{aligned} D_{sel} &= 10.3 \text{ FO4}, \\ D_{add} &= 7.6 \text{ FO4}, \\ D_{shift} &= 5.7 \text{ FO4}. \end{aligned} \tag{4.4.1}$$

Table 4.4.1: Data path T1D1: Logical effort and parasitic delay of the gates in the select, add and shift paths.

Gate	Select Path			Add Path			Shift Path		
	p	g	n	p	g	n	p	g	n
reg	2	2	1	2	2	1	2	2	1
QSLC	13	4	2						
CSA				10	6	1			
5:1 Mux	3.3	7	2	3.3	7	2	3.3	7	2
2:1 Mux	1.8	4	2	1.8	4	2	1.8	4	2
G	**153**			**119**			**8**		
P		**17**			**19**			**13**	
N			**7**			**6**			**5**

Table 4.4.2: Data path T1D1: Branching efforts in the select, add and shift paths.

Node	Select Path	Add Path	Shift Path
R0	2	2	2
R1		4	11
R2	332		
B	**664**	**8**	**22**

The delay in Equation (4.4.1) takes into account the additional amplification stages necessary to amplify the signals. Using the delay values in Equation (4.4.1), the worst-case delay of the data path for the synchronous design is

$$D_{sync} = MAX(D_{sel}, D_{add}, D_{shift}) = 10.3\ FO4. \tag{4.4.2}$$

For a self-timed design, the average-case delay of the data path is

$$\begin{aligned} D_{async} &= 0.47 * MAX(D_{select\text{-}path}, D_{add\text{-}path}) \\ &\quad + 0.53 * MAX(D_{select\text{-}path}, D_{shift\text{-}path}) = 10.3\ FO4. \end{aligned} \tag{4.4.3}$$

For the data path T1D1, the worst-case delay and the average-case delay are the same because the select path delay, D_{sel}, sets the delay of the data path for both

shift-only and add operations. A self-timed divider fails to take advantage of the faster shift-only operation using the data path T1D1. The data path T1D1, whilst suitable for synchronous design, is unsuitable for a self-timed design. The next section presents the data path T2D5 that is more suitable for a self-timed design.

4.4.2 Data Path T2D5

Figure 4.4.4 shows the data path T2D5. The data path T2D5 computes the QSLC in the middle of the clock cycle, hence topology 2. The data path T2D5 is a less-speculative data path because the data path selects an appropriate multiple of the divisor first and then performs a carry-save addition. In this data path the amplification of the select signals for the multiplexers in the lesser-significant bit position and in the quotient data path can be overlapped with the computation of the quotient selection logic, as illustrated in Figure 4.4.4.

Select Path

The select path is: `qslc-reg` $\rightarrow$ `amp` $\rightarrow$ `2:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `qslc-reg`. Table 4.4.3 shows the logical effort and parasitic delay of the logic gates in the select path along with the number of stages in each gate. Table 4.4.4 shows the the branching effort at various nodes in the select path. Using the values of G, P and N in Tables 4.4.3 and B in Table 4.4.4, the delay of the select path is

$$D_{sel} = 7 \text{ FO4}. \tag{4.4.4}$$

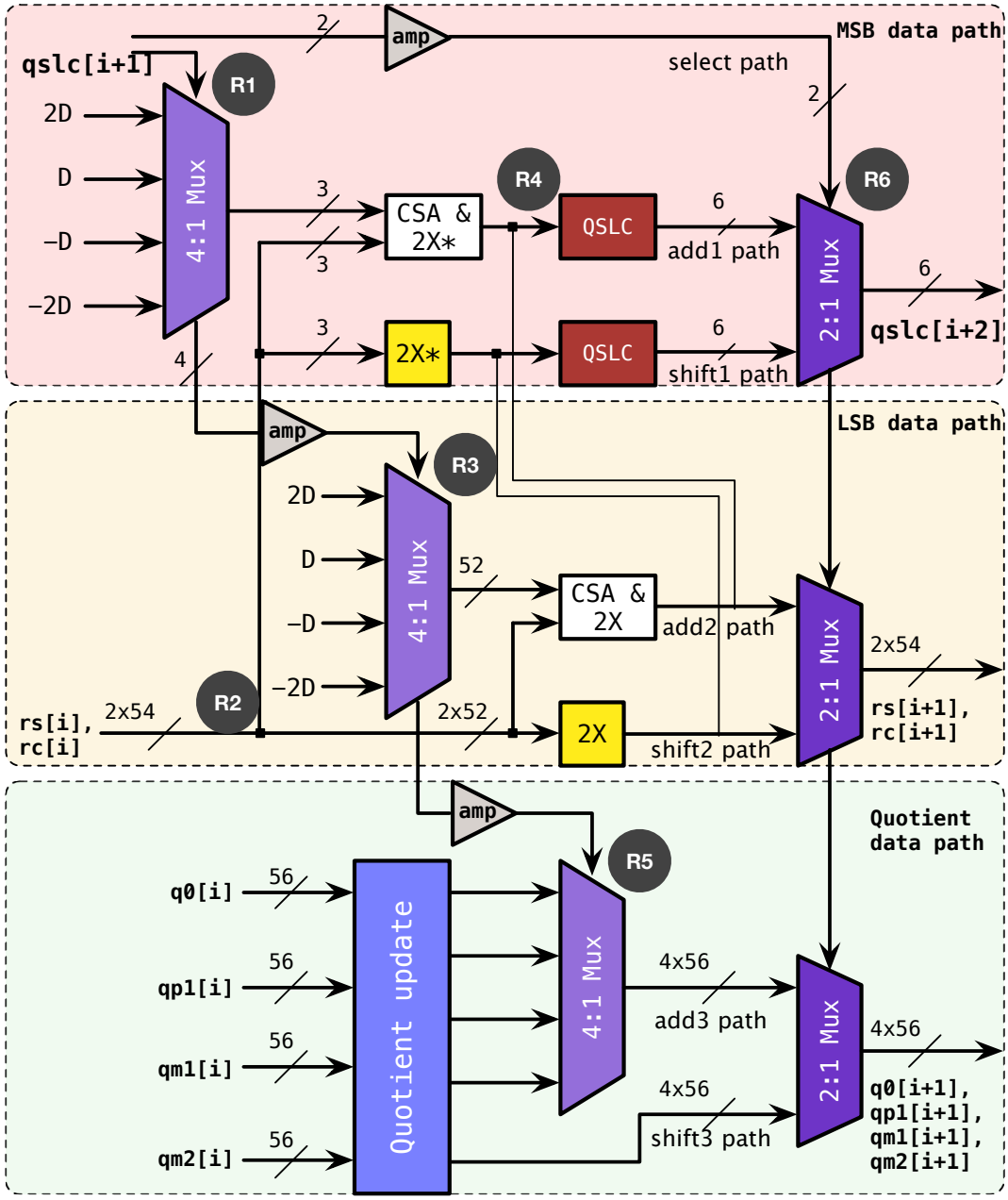


Fig. 4.4.4: Data path T2D5 showing the path of the two most-significant bits (MSB Path) and lesser-significant bits (LSB Path) of the partial remainder along with the quotient data-path. This is a less speculative data path because the data path selects an appropriate multiple of the divisor first and then performs the carry-save addition.

Table 4.4.3: Data path T2D5: Logical effort and parasitic delay of the gates in the select path along with number of stages in each gate.

Gate	Logical effort, g	Parasitic delay, p	Number of Stages, n
reg	2	2	1
amp	1	2	2
2:1 Mux	1.78	4	2
2:1 Mux	1.78	4	2
G	**6.4**		
P		**12**	
N			**7**

Table 4.4.4: Data path T2D5: Branching effort in the select path.

Node	Branching effort, b
R0	2
R6	338
B	**676**

Add Paths

The following are the four add paths that we have to consider for the data path T2D5:

Add1 path: `qslc-reg` $\rightarrow$ `4:1 Mux` $\rightarrow$ `CSA & 2X*` $\rightarrow$ `QSLC` $\rightarrow$ `2:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `qslc-reg`.

Add2 path: `qslc-reg` $\rightarrow$ `amp` $\rightarrow$ `4:1 Mux` $\rightarrow$ `CSA & 2X*` $\rightarrow$ `2:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `remainder-reg`.

Add3 path: `qslc-reg` $\rightarrow$ `amp` $\rightarrow$ `amp` $\rightarrow$ `4:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `quotient-reg`.

Add4 path: `remainder-reg` $\rightarrow$ `CSA & 2X*` $\rightarrow$ `QSLC` $\rightarrow$ `2:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `qslc-reg`.

Table 4.4.5 lists the logical effort and parasitic delay of the logic gates in the add paths along with the number of stages in each gate. Table 4.4.6 lists the branching effort at various nodes in the respective add paths.

Using the values of G, P and N in Tables 4.4.5 and B in Table 4.4.6, the delays of the add paths are as follows:

$$\begin{aligned} D_{add1} &= 11 \text{ FO4}, \\ D_{add2} &= 10.7 \text{ FO4}, \\ D_{add3} &= 10.6 \text{ FO4}, \\ D_{add4} &= 8.3 \text{ FO4}. \end{aligned} \tag{4.4.5}$$

The delay to consider for the add path is the maximum of the delays in the expression (4.4.5). Thus $D_{add} = 11$ FO4.

Table 4.4.5: Data path T2D5: Logical effort and parasitic delay of the gates in the add paths.

Gate	Add-path1			Add-path2			Add-path3			Add-path4		
	g	*p*	*n*	*g*	*p*	*n*	*g*	*p*	*n*	*g*	*p*	*n*
reg	2	2	1	2	2	1	2	2	1	2	2	1
amp				1	2	2	1	4	4			
4:1 Mux	2.7	6	2	2.7	6	2	2.7	6	2			
CSA & 2X*	10	6	1	10	6	1				14	6	1
QSLC	10	4	2							10	4	2
2:1 Mux	1.8	4	2	1.8	4	2	1.8	4	2	1.8	4	2
2:1 Mux	1.8	4	2	1.8	4	2	1.8	4	2	1.8	4	2
G	**1692**			**170**			**17**			**887**		
P		**26**			**24**			**20**			**20**	
N			**10**			**10**			**11**			**8**

Table 4.4.6: Data path T2D5: Branching effort in the add paths.

Node	Add-path1	Add-path2	Add-path3	Add-path4
R0	2	2	2	2
R1	5.2	2.6	23.9	
R2				1.25
R3		52		
R4	2.6			
R5			225	
B	**27.04**	**270.4**	**10707**	**2.5**

Shift Paths

To estimate the delay of the shift path, consider the following two paths.

Shift1 path: `rem-reg` $\rightarrow$ `2X*` $\rightarrow$ `QSLC` $\rightarrow$ `2:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `qslc-reg`

Shift2 path2: `rem-reg` $\rightarrow$ `2X` $\rightarrow$ `2:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `rem-reg`

Table 4.4.7 lists the logical effort and parasitic delay of the logic gates in the shift paths along with the number of stages in each gate. Table 4.4.8 lists the the branching effort at various nodes in the respective shift paths. The delay of the shift path, $D_{shift} = MAX(D_{shift1}, D_{shift2})$, where D_{shift1} and D_{shift2} are the delay of shift1 and shift2 paths.

Using the values of G, P and N in Tables 4.4.7 and B in Table 4.4.8, we get

$$\begin{aligned} D_{shift1} &= 6.6 \text{ FO4}, \\ D_{shift2} &= 4.3 \text{ FO4}. \end{aligned} \tag{4.4.6}$$

Therefore, $D_{shift} = D_{shift1} = 6.6$ FO4.

If data path T2D5 is used in a synchronous environment, then the delay of the data path is

$$D_{sync} = MAX(D_{sel}, D_{add}, D_{shift}) = 11 \text{ FO4}. \tag{4.4.7}$$

Table 4.4.7: Data path T2D5: Logical effort and parasitic delay of the gates in the shift paths.

Gate	Shift-path1			Shift-path2		
	g	*p*	*n*	*g*	*p*	*n*
reg	2	2	1	2	2	1
QSLC	10	4	2			
2:1 Mux	1.78	4	2	1.78	4	2
2:1 Mux	1.78	4	2	1.78	4	2
G	**64**			**6.4**		
P		**14**			**10**	
N			**7**			**5**

Table 4.4.8: Data path T2D5: Branching effort in the shift paths.

Node	Shift-path1	Shift-path2
R0	2	2
R2	9	5
B	**18**	**10**

On the other hand, if this data path is used in a self-timed design environment then the average-case delay of the data path is

$$\begin{aligned} D_{async} &= 0.47 * MAX(D_{sel}, D_{add}) \\ &\quad + 0.53 * MAX(D_{sel}, D_{shift}) = 8.9 \text{ FO4}. \end{aligned} \tag{4.4.8}$$

4.5 Evaluation of Data paths

The logical effort calculations for the remaining thirteen data paths are in Appendix C. In addition to the delay estimation using logical effort, I synthesized all the data paths using Synopsys Design Compiler and mapped to a TSMC 40nm standard cell library.

Table 4.5.1 shows the ranking of the data paths by delay using the logical effort method (Table 4.5.1a) and static-timing analysis (Table 4.5.1b) in a synchronous en-

vironment. Table 4.5.2 shows the ranking of the data paths by average delay using the logical effort method (Table 4.5.2a) and static-timing analysis (Table 4.5.2b) in a self-timed environment. The rank column in Tables 4.5.1b, 4.5.2 and 4.5.2b show the rank of the corresponding data path topology in Table 4.5.1.

Table 4.5.1: Ranking of data paths by speed for synchronous designs.

(a) Using Logical Effort

Data path	Delay in FO4	Rank
T2D1	10.3	1
T1D1	10.3	1
T3D1	10.3	1
T1D3	10.5	4
T2D3	10.6	5
T2D4	10.6	5
T3D3	10.6	5
T3D4	10.7	8
T1D2	10.8	9
T2D5	11.0	10
T2D2	11.2	11
T3D2	11.2	11
T3D5	11.2	11
T1D4	11.8	14
T1D5	12.2	15

(b) Using Design Compiler's STA

Data path	Delay in ps	Rank in Table 4.5.1a
T2D1	246	1
T1D1	247	1
T3D1	248	1
T1D3	252	4
T3D4	257	8
T2D3	258	5
T2D4	259	5
T1D2	261	9
T2D5	263	10
T3D3	264	5
T3D5	265	11
T2D2	271	11
T3D2	272	11
T1D4	280	14
T1D5	284	15

From the Tables 4.5.1 and 4.5.2 we can make the following observations:

1. For the synchronous designs, the three data paths T2D1, T1D1 and T3D1 appear at the top of the Tables 4.5.1a and 4.5.1b. This suggests that synchronous designs prefer a fully-speculative design.

Table 4.5.2: Ranking of data paths by speed for self-timed designs designs.

(a) Using Logical Effort

Data path	Avg. delay in FO4	Rank in Table 4.5.1a
T2D5	8.9	10
T2D2	9.0	11
T2D3	9.0	5
T2D4	9.0	5
T3D3	9.1	5
T2D1	9.2	1
T3D4	9.2	8
T3D5	9.3	11
T3D2	9.3	11
T3D1	9.5	1
T1D2	10.0	9
T1D3	10.0	4
T1D1	10.3	1
T1D5	10.6	15
T1D4	10.6	14

(b) Using Design Compiler's STA

Data path	Avg. delay in ps	Rank in Table 4.5.1a
T3D1	229	1
T3D4	232	8
T2D5	233	10
T3D3	235	5
T2D3	236	5
T3D5	236	11
T3D2	240	11
T2D1	242	1
T1D2	245	9
T1D3	246	4
T1D1	247	1
T2D4	248	5
T1D5	255	15
T2D2	257	11
T1D4	259	14

2. For self-timed designs, all the data paths in Topology 1 (T1) appears in the bottom-half of the table. Therefore, the data paths in Topology 1 are less suitable for self-timed designs.

3. There is a discordance between the ranking of the data paths using logical effort and static-timing analysis. The two primary reasons for the discordance are as follows:

    a) The logical effort calculations use the logical effort and parasitic delay values mentioned in the logical effort book [37]. The logical effort and

parasitic delay values of the gates in the standard cell library can be quite different from the values in the book, especially the parasitic delay values.

b) The delay calculation using the logical effort method ignores input transition times. The static-timing analysis engine, however, takes input transition times into account to estimate path delays. This is important because the difference in delay values for some of the datapaths is on the order of 1 to 2 picoseconds.

Figure 4.5.1 shows the scatter-plot of average energy consumption per iteration and delay per iteration for all the data paths. The data points in green are for self-timed designs and the data points in red are for synchronous designs. The figure also shows the possible data path choices for self-timed and synchronous design for further optimization. From the figure, we can notice that there are several data paths for self-timed designs that offer better performance while consuming less energy per iteration, on average, compared to synchronous design alternatives.

For a synchronous design, data paths T2D1, T3D1 and T1D1 are all good choices considering the delay per iteration. All three data paths are fully-speculative. I choose T1D1 data path for further optimization because this data path is presented in several references [15, 18, 19, 6] and it would be easier to make a fair comparison of the optimization techniques presented in the next chapter with the work presented in the references.

For a self-timed design, the data paths T3D1, T3D4, T2D5 and T3D3 are all good choices considering the average delay per iteration. I chose T2D5 data path for further optimization because this data path consumes the least energy of the four choices at the cost of 2% slower speed on average.

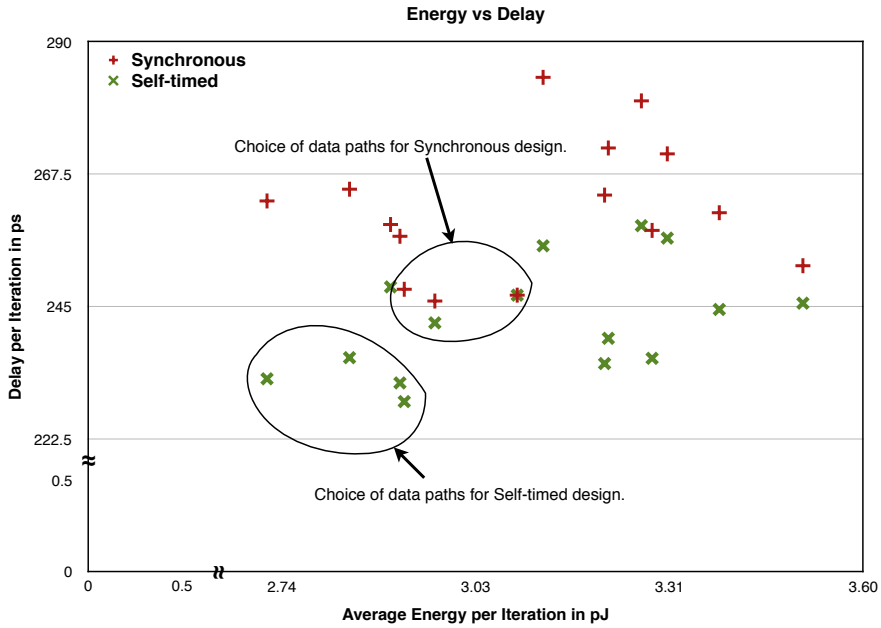


Fig. 4.5.1: Scatter plot of energy per iteration vs the average delay per iteration for various data path topologies. The green data points are for self-timed designs and the red data points are for synchronous designs.

4.6 Summary of Evaluation Methodology

In this chapter, using the divider example I presented a methodology to evaluate various data path topologies for a conditional statement. In the context of a division algorithm, a self-timed design can take advantage of the faster shift-only operation. Therefore, the evaluation methodology takes into account how frequently a shift-only operation and an addition followed by a shift operation are executed on average per division. Using this methodology I have shown that a fully-speculative data path, T1D1, is suitable for synchronous design but unsuitable for a self-timed design. For self-timed design, however, a less-speculative data path, T2D5, is suitable. For further data path optimization, I chose data path T1D1 for synchronous design and data path T2D5 for self-timed design.

5

Design Optimization Techniques

After evaluating the data paths for worst and average case delay in Chapter 4, I selected the data path T1D1 for the synchronous design and data path T2D5 for the self-timed design. In a divider data path, the high-capacitive load that the multiplexers present to their select signals limit the delay of the data paths. This chapter presents two optimization techniques that address the problem of high capacitance to further reduce the delay of the data paths. The first technique is the sizing optimization where the register bits that drive the critical path have bigger drive strengths than the rest of the register bits. The second technique is called *Glissando* which exploits a simple idea of delaying the computation of the non-critical bits.

Section 5.1 presents an analysis of the various cases of tapering of the register sizes using the method of logical effort. This analysis will give a designer an idea about the relative drive strengths of the gates required to achieve an optimum performance.

Section 5.2 presents an analysis of the *Glissando* optimization technique. The *Glissando* technique offers a way to increase the word-size of the operands without effecting the delay of the data path. This is very useful when we increase the accuracy requirements from double-precision to quadruple-precision or when consid-

ering a divider design to implement the “mod” function for a 1024-bit or 2048-bit RSA crypto-systems[28].

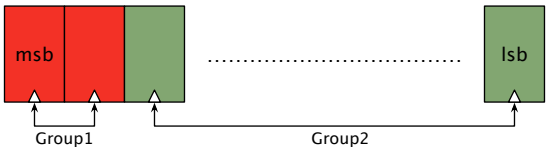
5.1 Sizing Optimization

The delay estimates in Sections 4.4.1 and 4.4.2 assume that all the register bits have the same drive strengths. Tapering the register sizes such that the registers driving the critical path have larger drive-strengths than the rest of the registers can reduce the delay of the data paths. Tapering of the register sizes can be sudden or gradual. Figure 5.1.1 illustrates that three cases I have considered in this chapter. *Case1* is an example of sudden tapering, and *case2* and *case3* are examples of gradual tapering.

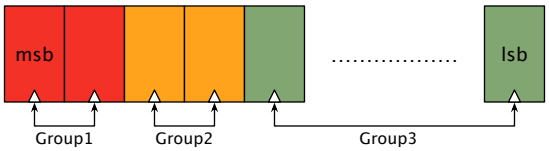
Case1 has two groups of registers: Group1 and Group2. Group1 registers consist of few most-significant bits and Group2 registers consist of the rest of the less-significant bits. Let Group1 registers be s times bigger than Group2 registers. *Case2* has three groups of registers: Group1, Group2 and Group3. Group2 registers consist of the next few most-significant bits and Group3 registers consist of the rest of the less-significant bits. Group1 registers are s^2 times bigger drive strengths than Group3 registers. *Case3* has four groups of registers: Group1, Group2, Group3 and Group4. Group3 registers consist of the next few most-significant bits and Group4 registers consist of the rest of the bits. Group1 registers are s^3 times bigger drive strengths than Group4 registers.

5.1.1 Data Path T1D1

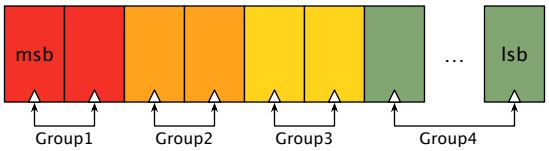
Consider case1 for T1D1 where there are two groups of registers: Group1 and Group2. Group1 registers drive the critical path and Group2 registers drive the non-critical path. In T1D1 data path, the critical path is the select path through the quotient selection logic. The non-critical path is an add path through one of the four carry-save



(a) Case1 has two groups of registers: Group1 and Group2. Group1 registers consist of few most-significant bits and Group2 registers consist of the rest of the less-significant bits. Let Group1 registers be s times bigger than Group2 registers.



(b) Case2 has three groups of registers: Group1, Group2 and Group3. Group2 registers consist of the next few most-significant bits and Group3 registers consist of the rest of the less-significant bits.



(c) Case3 has four groups of registers: Group1, Group2, Group3 and Group4. Group3 registers consist of the next few most-significant bits and Group4 registers consist of the rest of the bits. Group1 registers are s^3 times bigger drive strengths than Group4 registers.

Fig. 5.1.1: The three cases of tapering the register sizes.

adders. Therefore, Group1 registers consist of the two most-significant bits of r_s and r_c and Group2 registers consist of the rest of the less-significant bits of the partial remainder and the quotient bits.

Figure 5.1.2 shows the register-to-register path for the three most-significant bits of the partial remainder. A bit at position n is the most-significant bit. In the figure, the modules with label `MAJ` and `PAR` produce majority and parity bits from all five alternatives, respectively. Figure 5.1.2 shows Group1 bits in red color and Group2

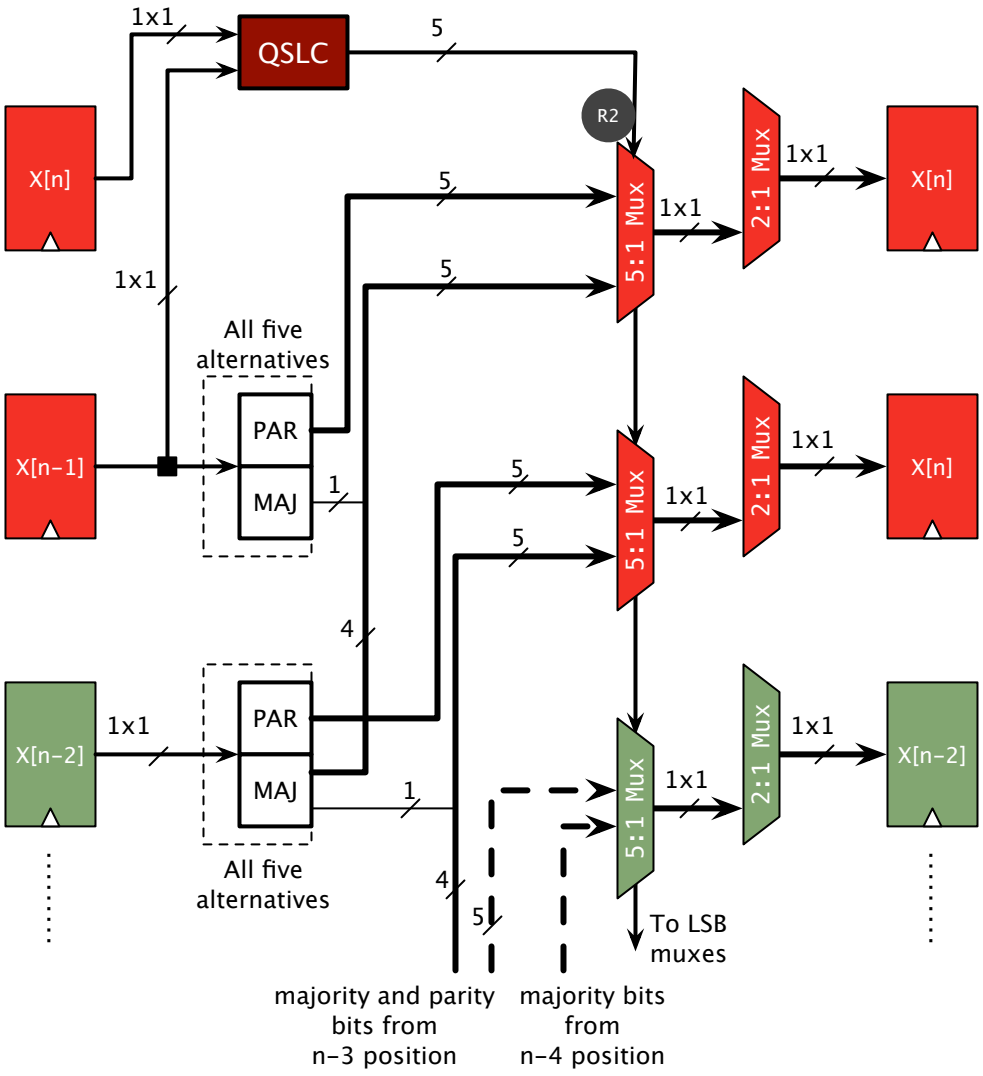


Fig. 5.1.2: Data path T1D1: Register-to-register path for the three most-significant bits.

bits in light-green color.

Let the Group1 registers be s times bigger than the Group2 registers. With this configuration, the branching effort at node R2 of Figure 5.1.2 is

$$b_{R2} = \frac{(4 * C_{5:1Mux,g1}) + 328 * C_{5:1Mux,g2}}{C_{5:1Mux,g1}}, \tag{5.1.1}$$

where $C_{5:1Mux,g1}$ and $C_{5:1Mux,g2}$ are the input capacitances of the 5:1 multiplexers

in Group1 and Group2 respectively. Because the Group1 registers are s times bigger than the Group2 registers, $C_{5:1Mux,g1} = s * C_{5:1Mux,g2}$ and yields

$$b_{R2} = 4 + \frac{328}{s}. \tag{5.1.2}$$

In equation (5.1.2), the first term is the number of multiplexers in Group1 and the second term is the number of multiplexers in Group2 scaled by s. The path effort of the select path is

$$F_{sel} = 306 * (4 + \frac{328}{s}). \tag{5.1.3}$$

From Equation (5.1.3), F_{sel} reduces as s increases. Increasing s comes at the cost of increasing the path electrical effort of an add path at bit positions n-2 and n-3 of the partial remainder for the following reasons. The register bits at position n-2 drive the majority and parity circuits that in turn drive 5:1 multiplexers at position n and n-1, respectively, in Group1. The register bits at position n-3 drive the majority circuit that in turn drive the 5:1 multiplexer at position n-1 in Group1. To estimate the delay of an add path, consider the bit at position n-2.

The path effort of an add path is

$$F_{add} = 1047s. \tag{5.1.4}$$

Assuming all the bits arrive at the input of the register at the same time, an optimum value for s is when the delay of the select path is the same as the delay of an add path. Omitting the effects of parasitic delay reduces the complexity of the algebra considerably and gives a good estimate for the value of s. Consequently,

$$F_{sel} = F_{add}, \tag{5.1.5}$$

$$1047s^2 - 1224s - 100368 = 0. \tag{5.1.6}$$

In Equation (5.1.5), the first term comes from the path effort of an add path, the second term comes from the path effort of the select path considering only the Group1

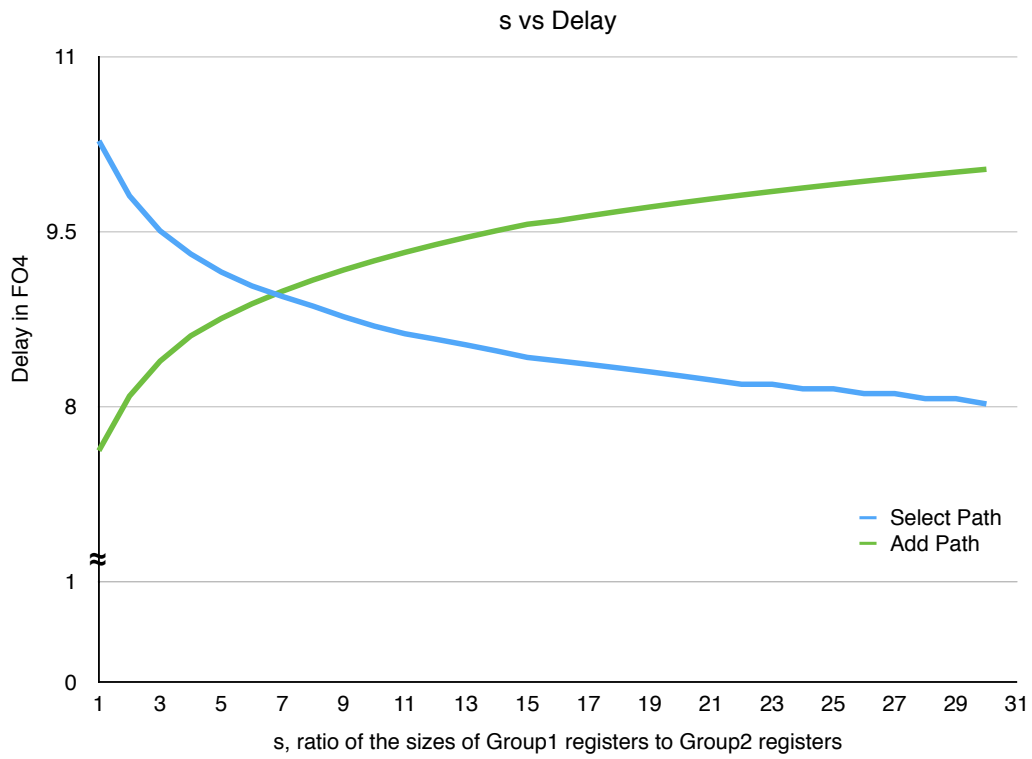


Fig. 5.1.3: Case1: The delay of select and add paths as a function of s. The delay of the select path decreases as the value of s increases. On the contrary, the delay of an add path increases as the value of s increases. An optimum value for s is when the delay of the select and add paths are equal. For case1, $s \approx 7$ and delay of the select path is approximately 9 FO4.

bits and the third constant-term comes from the path effort of the select path considering the Group2 bits. Solving the quadratic Equation in 5.1.5 gives $s \approx 10$. Using $s = 10$, we get 8.7 FO4 for the delay of the select path and 9.4 FO4 for the delay of the add path. Omitting the effects of parasitic delay results in an overestimated value for s.

A more accurate estimate for s can be obtained numerically by taking into account the effects of parasitic delay. Figure 5.1.3 shows the delay of the select and add paths as a function of s. An optimum value for s is the x-coordinate of the point of intersection of the two curves. Therefore, $s \approx 7$ and the delay of both select and add paths are approximately 9 FO4 each.

Case2 has three groups of registers: Group1, Group2 and Group3, where Group2 consists of the third and the fourth most-significant bit registers and Group3 consists of the rest of the less-significant remainder bits and the quotient bits. Let Group1 registers be s times bigger than Group2 registers and Group2 registers be $s1$ times bigger than Group3 registers. For all the bits to arrive at the input of the register bits at the same time requires that the path effort of the bits in Group2 and Group3 be the same. Consequently, $s = s1$ and Group1 registers are s^2 times bigger than the Group3 registers.

The polynomial that represents case2 configuration is

$$1047s^3 - 1224s^2 - 1224s - 99144 = 0 \tag{5.1.7}$$

In Equation 5.1.7, the first term comes from the path effort of an add path and the second, third and fourth terms come from the path efforts of the select path considering only the bits in Group1, Group2 and Group3 respectively. Finding a real-positive root for the polynomial in Equation (5.1.7) gives $s \approx 5$.

Taking the effect of parasitic delay into account and solving for s numerically gives $s \approx 4$. Figure 5.1.4 shows the plot of add and select paths as a function of s for the configuration in case2. For $s = 4$, the delay of the select and add paths are approximately 8.9 FO4.

Case3 has four groups of registers: Group1, Group2, Group3 and Group4, where Group3 consists of the fifth and sixth most-significant bit registers and Group4 consists of the rest of the lesser-significant remainder bits and the quotient bits. The polynomial that represents case3 configuration is

$$1047s^4 - 1224s^3 - 1224s^2 - 1224s - 97920 = 0. \tag{5.1.8}$$

Finding a real-positive root for the polynomial in Equation (5.1.8) gives $s \approx 4$.

Taking the effect of parasitic delay into account and solving for s numerically gives $s \approx 3$. Figure 5.1.5 shows the plot of select and add paths as a function of s for

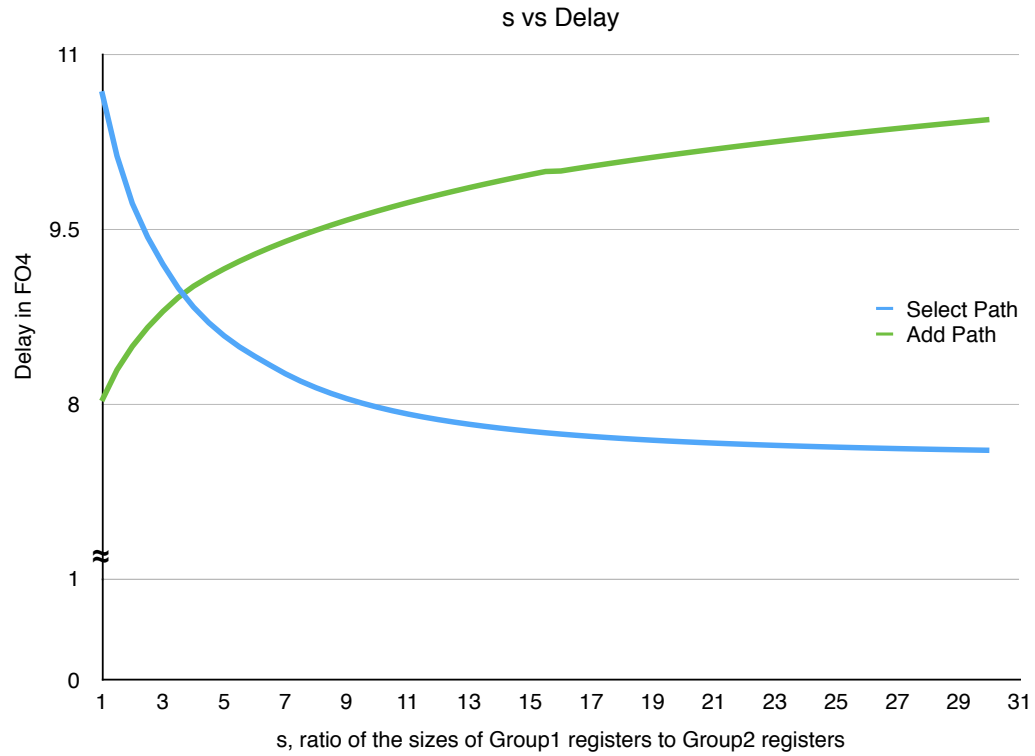


Fig. 5.1.4: Case2: Delay of the select and add paths as a function of s. An optimum value for $s \approx 4$ and the delay of the select path is 8.9 FO4.

the configuration in case3. For $s = 3$, the delay of the select and add paths are approximately 8.8 FO4.

From Equations (5.1.5), (5.1.7) and (5.1.8), we can see a pattern emerging that can be generalized for a case with n number of groups as follows:

$$F_{add} * s^n - F_{sel} * m * s^{n-1} \ldots - F_{sel} * m * s^{n-(n-1)} \\ -F_{sel} * (w - m * (n-1)) = 0, \tag{5.1.9}$$

where m is the number of multiplexers in each group, n is the total number of groups and w is the total number of multiplexers in the datapath.

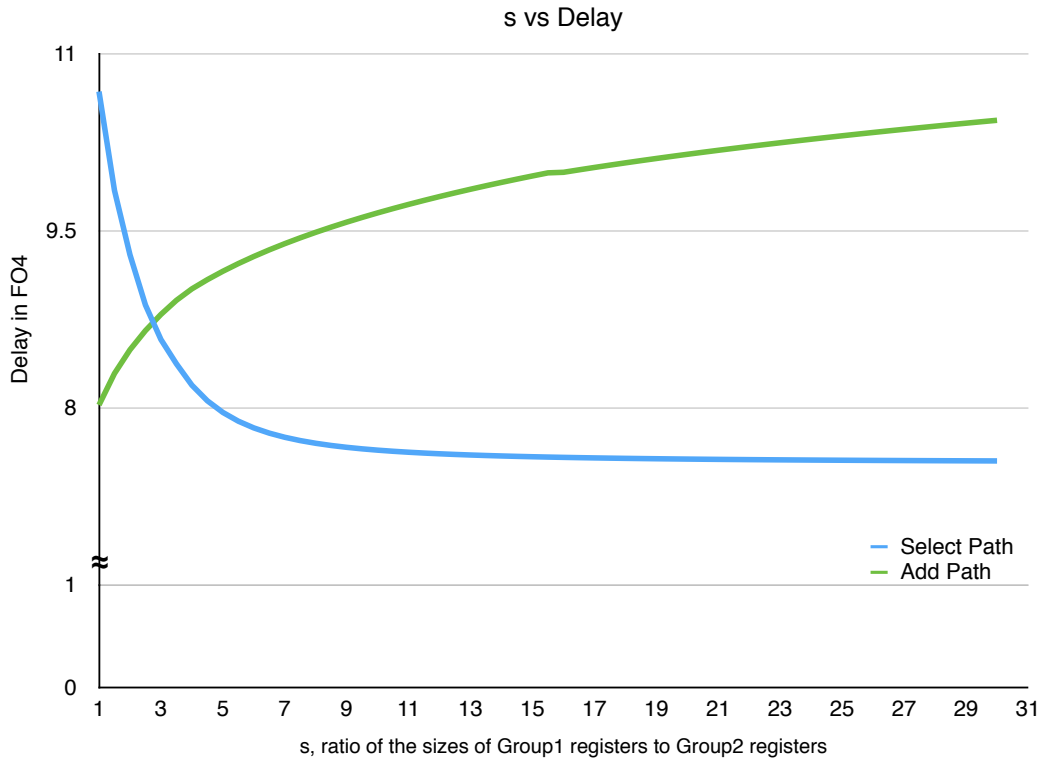


Fig. 5.1.5: Case3: Delay of the select and add paths as a function of s. An optimum value for $s \approx 3$ and the delay of the select path is 8.8 FO4.

Table 5.1.1 summarizes the estimated delay for the T1D1 data path considering each case. From the table, a case with sudden tapering of the register sizes (case1) yields an improvement of 13% in the delay compared to a case with no tapering at all (case0). A gradual tapering of the register sizes, case2 and case3, yields a minimal improvement to the delay. It is clear that sudden tapering offers the maximum incremental improvement. Furthermore, gradual tapering requires big registers which consume more power and area. For example, if 4X is the drive strength of the registers in less-significant position, then Group1 registers need to be 64X in case2 and 108X in case3. Keeping this in mind, I will now analyze the effect of sudden tapering on the average delay of the T2D5 data path.

Table 5.1.1: Delay of the data path T1D1 considering four different cases of tapering.

Case	Summary	Comments
case0	$s = 1$ and Delay = 10.3 FO4	All the registers are of the same size
case1	$s = 7$ and Delay = 9.0 FO4	Group1 = 7 x Group2
case2	$s = 4$ and Delay = 8.9 FO4	Group1 = 16 x Group3
case3	$s = 3$ and Delay = 8.8 FO4	Group1 = 27 x Group4

5.1.2 Data Path T2D5

This research uses the data path T2D5 in a self-timed divider design. Hence the objective is to reduce the average-case delay rather than the worst-case delay for the data path T2D5. The average-case delay of the data path is

$$\begin{aligned} D_{avg} &= 0.47 * MAX(D_{sel}, D_{add}) \\ &\quad + 0.53 * MAX(D_{sel}, D_{shift}), \end{aligned} \tag{5.1.10}$$

where, D_{sel}, D_{add} and D_{shift} are the delays of the select, add and shift paths respectively. From the analysis presented in Section 4.4.2, we know that $D_{sel} = 7FO4$, $D_{add} = 11FO4$ and $D_{shift} = 6.6FO4$. Therefore,

$$\begin{aligned} MAX(D_{sel}, D_{add}) &= D_{add} = 11 \text{ FO4}, \\ MAX(D_{sel}, D_{shift}) &= D_{sel} = 7 \text{ FO4}. \end{aligned} \tag{5.1.11}$$

In the data path T2D5, D_{add} and D_{sel} sets the add and shift periods, respectively. The objective is to reduce the average-case delay of the data path by reducing the delay of the select path or add path or both.

First, consider the register configuration in *case1* to reduce the delay of the select path. *Case1* has two groups of registers: Group1 and Group2. In the T2D5 data path, Group1 has six QSLC register bits and two most-significant bits of r_s and r_c.

Group2 has the rest of the register bits. Let Group1 registers be s times bigger than the Group2 registers. With this configuration, the path effort of the select path is

$$F_{sel} = 12.6 * (10 + \frac{328}{s}). \tag{5.1.12}$$

An optimum value of s is when the delay of the select and shift paths are the same. Omitting the effects of parasitic delay, an optimum value for s is approximately 4. Using $s = 4$, $D_{sel} = 6.2$ FO4 and $D_{shift} = 6.6$ FO4. The value of s has no effect on the shift path delay, D_{shift}, because the bits driving the shift path are in Group1.

With $s = 4$, the branching effort at node R1 of Figure 4.4.4 is

$$b_{R1} = 3 + \frac{2.2}{s}. \tag{5.1.13}$$

For $s = 4$, $b_{R1} \approx 3.5$ which in turn gives an add path delay, $D_{add} = 10.8$ FO4. Increasing the value of s in Equation (5.1.13) reduces the branching effort at node R1. The minimum value of b_{R1} can be 3 which then gives an add path delay of 10.7 FO4. Therefore, there is not much benefit in attempting to find an optimum value of s considering the add path. Consequently with sudden-tapering configuration, the average-case delay is $D_{avg} \approx 8.6FO4$. Approximately a 3% reduction in average-case delay with sudden tapering can be obtained compared to a configuration where all the register bits are of the same size.

5.2 Glissando

Thus far, we have considered inserting the amplification stages in the select path to drive all the multiplexers at the same time so that all the bits arrive at the input of the registers at the same time. What if we relax this constraint by allowing the critical bits to arrive sooner and the non-critical bits to the arrive later? This relaxed constraint is the main idea behind the optimization technique called *Glissando*. *Glissando* is a

musical term that refers to a glide from one pitch to another. In the optimization presented in this section, the critical higher-order bits arrive at the registers first and the non-critical lower-order bits arrive at the registers gradually late. This action resembles sliding a finger over the keys of a piano creating a *Glissando*.

The glissando optimization technique minimizes the high capacitance in the select path by delaying the computation of the lesser-significant bits. Let me explain the glissando technique with help of T1D1 data path.

5.2.1 Data Path T1D1

Figure 5.2.1 shows the main idea of glissando using the T1D1 data path. In the figure, there is an amplification stage between two successive groups. Amplification stages carry label `amp` in the figure. Typically, an amplification stage is an inverter, but I will use a buffer as an amplification stage to avoid signal inversions. Let d_{amp} be the delay of an amplification stage. An amplification stage between two successive groups delays the selection of the less-significant bits by d_{amp}. For example, in Figure 5.2.1, an amplification stage between groups $\phi 1$ and $\phi 2$ delays the selection of the less-significant bits in group $\phi 2$ by d_{amp}. A design with n number of groups has n-1 amplification stages between groups $\phi 1$ and ϕn, delaying the selection of the bits in group ϕn by $(n\text{-}1) * d_{amp}$.

Inserting amplification stages between the groups reduces the capacitance at the output of the QSLC module but skews the arrival of the bits at the input of the registers in the following manner. The bits from groups $\phi 1$, $\phi 2$ and ϕn arrive first, second and last respectively. To compensate for the skew in the arrival times of the bits, the arrival times of the clock are also skewed appropriately. The amplifiers carrying the label ϕ in Figure 5.2.1 serves the purpose of skewing the clock. The clocks, `clk1`, `clk2` ... `clkn` have the same clock period, t_{clk}, but with a phase difference equal to the delay of the ϕ module, d_{ϕ}. I will use the term *phase group* to denote a group of

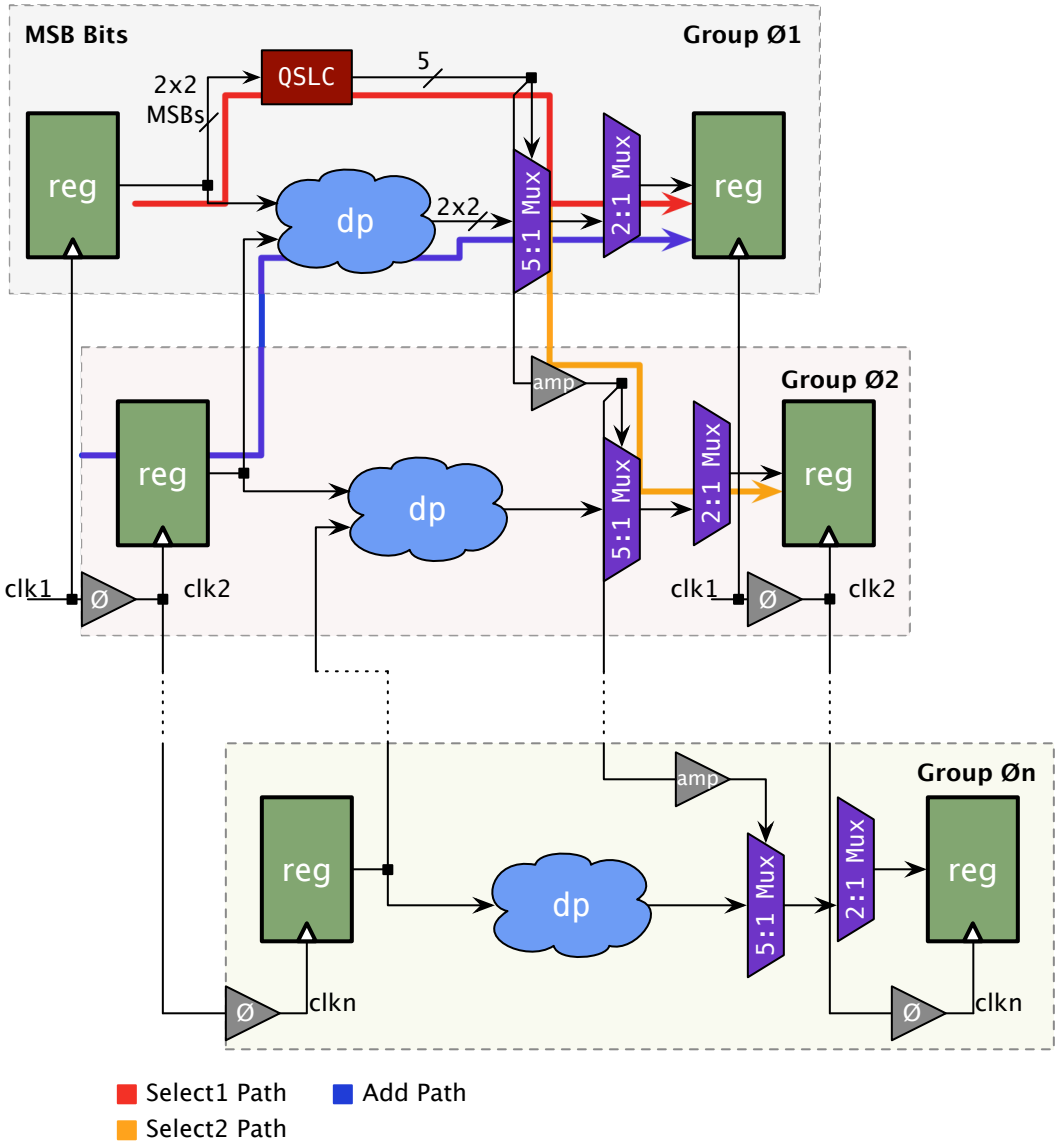


Fig. 5.2.1: Glissando: Main idea. The figure illustrates the glissando technique for the T1D1 data path. The main idea behind glissando is to delay the selection of the lesser-significant bits by inserting additional amplification stages for lesser-significant bits.

registers that receive the same clock period and phase. Registers in different phase groups receive the same clock period but different phase.

In Figure 5.2.1, the module carrying the label *dp* implements all five alternatives to update the partial remainder and the quotient. Because of the 2X* operation at the end of every iteration, there are bits that move from phase group ϕn to $\phi(n\text{-}1)$. In Figure 5.2.1, a line from phase group ϕn to $\phi(n\text{-}1)$ shows the movement of bits from phase group ϕn to $\phi(n\text{-}1)$. Because the launching of the bits in phase group ϕn is delayed by d_ϕ, we need to consider the paths originating in phase group ϕn and terminating in phase group $\phi(n\text{-}1)$ to estimating the clock period.

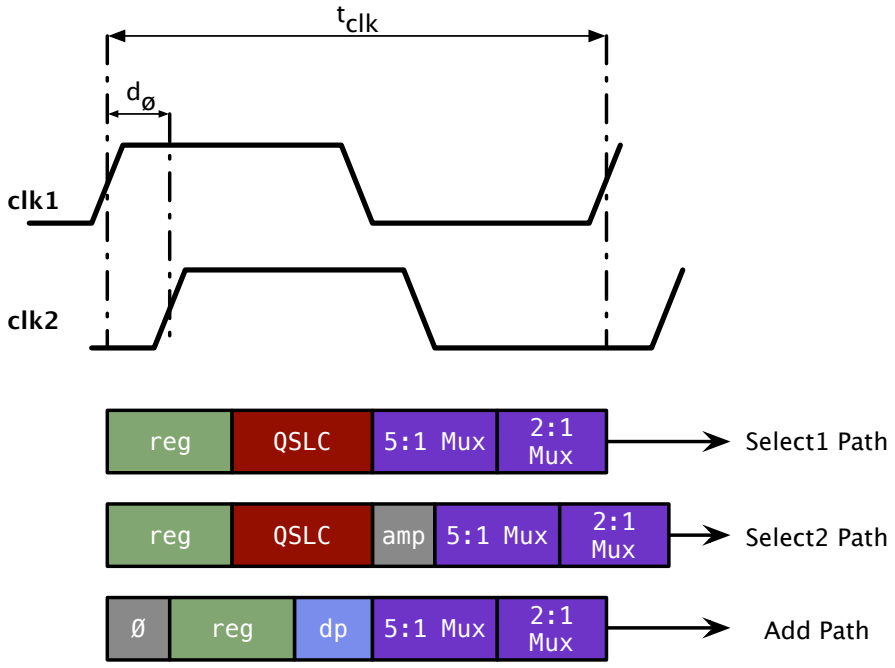


Fig. 5.2.2: Timing requirement for the bits in select1, select2 and add paths in Figure 5.2.1

To estimate the clock period, consider the following three paths: select1, select2 and add paths. In Figure 5.2.1, red, orange and blue lines show select1, select2 and add paths. Figure 5.2.2 shows the launching and capturing clocks for each path. Clk1 launches the bits in select1 and select2 paths, and clk2 launches the bits in the add

path. Clk1 captures the bits from select1 and add paths, and clk2 captures the bits from select2 path. The delays of the select1, select2 and add paths are

$$\begin{aligned} d_{sel1} &= d_{reg} + d_{qslc} + d_{5:1Mux} + d_{2:1Mux}, \\ d_{sel2} &= d_{reg} + d_{qslc} + d_{amp} + d_{5:1Mux} + d_{2:1Mux}, \\ d_{add} &= d_{\phi} + d_{reg} + d_{dp} + d_{5:1Mux} + d_{2:1Mux}, \end{aligned} \tag{5.2.1}$$

where d_{req}, d_{ϕ}, d_{qslc}, $d_{5:1Mux}$, $d_{2:1Mux}$, and d_{dp} are the delays of register, ϕ, QSLC, 5:1 Multiplexer, 2:1 multiplexer and dp modules respectively.

The clock period, t_{clk}, is

$$t_{clk} \geq MAX(d_{select1\text{-}path}, d_{add\text{-}path}) \tag{5.2.2}$$

and From Section 4.4.1, we know that $d_{reg} + d_{dp} + d_{5:1Mux} + d_{2:1Mux} = 7.6$ FO4. Therefore,

$$d_{add} = (d_{\phi} + 7.6) \text{ FO4} \tag{5.2.3}$$

Setting $d_{\phi} = 0$ in Equation (5.2.3) takes us back to the assumption that all the bits are launched and captured at the same time resulting in $d_{qslc} >> d_{dp}$.

The expressions in (5.2.1) present the following challenges. On the one hand, if each phase group has as few as four multiplexers, then the total number of phase groups required for a divider with 332 multiplexers is 81. Managing 81 phase groups is cumbersome. On the other hand, adding more multiplexers in each group increases the delay of the amplifier, d_{amp}, resulting in an increased clock period. So the objective is to find an optimum number of phase groups and the number of bits per phase group that will minimize the clock period.

To estimate the clock period, we need to find a value for d_{ϕ} and d_{sel1}. Assuming $d_{\phi} = d_{amp}$, we can estimate d_{ϕ} by subtracting d_{sel1} from d_{sel2}. The delays, d_{sel1} and d_{sel2}, depend on how the bits are distributed across different phase groups.

Lets us consider Figure 5.2.3 to estimate d_{sel1} and d_{sel2}. The figure shows a simpler model of the select paths with n number of phase groups. Let phase-group

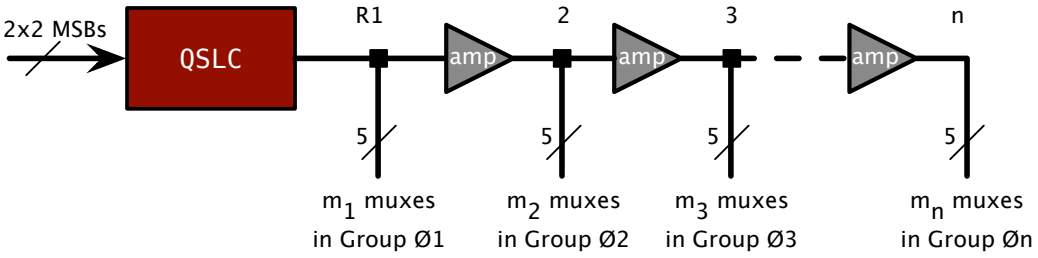


Fig. 5.2.3: A Simple model for the select path

$\phi 1$ contain m_1 bits, phase-group $\phi 2$ contain m_2 bits and so on. The total capacitance at node R1 is

$$C_{R1} = m_1 * C_{5:1Mux} + \frac{m_2 * C_{5:1Mux}}{f^2} + \frac{m_3 * C_{5:1Mux}}{(f^2)^2} + \ldots + \frac{m_n * C_{5:1Mux}}{(f^2)^{n-1}}, \tag{5.2.4}$$

where f is the stage effort and $f = 4$ is a good choice. To estimate d_{sel1} and d_{sel2}, I make the following two assumptions:

1. The phase-group $\phi 1$ contains the minimum number of bits required to compute the the select signals for the next cycle, that is, $m_1 = 4$.

2. The number of bits in each phase group increase in a geometric progression, that is,
$$\frac{m_2}{m_1} = \frac{m_3}{m_2} = \ldots \frac{m_n}{m_{n-1}} = r, \tag{5.2.5}$$
where r is the common ratio.

Figure 5.2.4 shows the delay d_{sel2} as a function of the number of phase groups and each data point represents delay for different values of r for $r > 1$. The figure also shows the two choices for distributing the bits across different phase groups. Choosing series1 results in three phase groups with $m_1 = 4$, $m_2 = 36$, and $m_3 = 290$. I chose series2 that results in four phase groups with $m_1 = 4$, $m_2 = 16$, $m_3 = 64$, and $m_4 = 248$ (remaining bits). For this configuration of bit distribution,

$d_{sel2} = 9.3$ FO4 and $d_{sel1} = 7.4$ FO4. Consequently, $d_\phi = d_{amp} = 1.8$ FO4. Using $d_\phi = 1.8$ FO4 in Equation 5.2.3 and substituting the values of d_{sel1} and d_{add} in Equation 5.2.1 results in $t_{clk} \geq 9.4$ FO4.

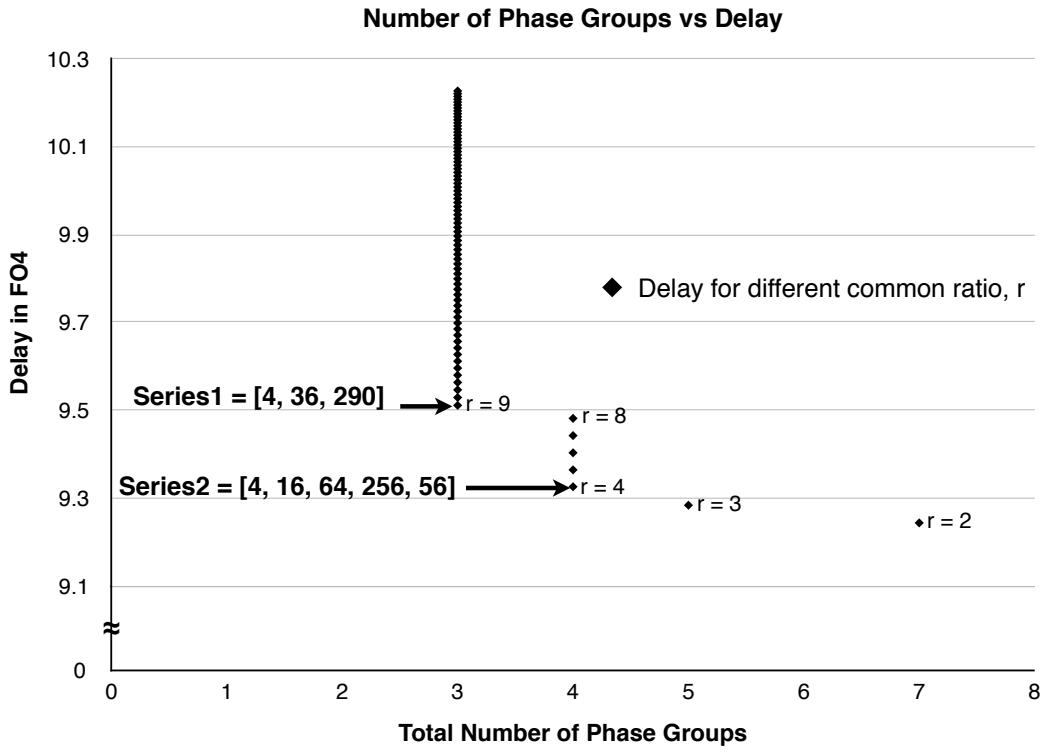


Fig. 5.2.4: Delay of the select2 path as a function of the number of phase groups for the T1D1 data path. Different data point represents different value for the common ratio, r.

5.2.2 Data Path T2D5

For the T2D5 data path, we need to estimate two different clock-periods: shift and add periods. Shift period is for selecting the result from the shift-only operation and add period is for selecting the result from an add operation. Let t_{shift} and t_{add} denote the shift and add periods respectively.

Figure 5.2.5 shows the paths that we need to consider to estimate t_{shift}. We need to consider the following four paths: select1, select2, shift1 and shift2 paths. The four

paths are color coded. Figure 5.2.5 shows select1, select2, shift1 and shift2 paths with red, orange, green and blue lines. In the figure, clk1 and clk2 have the same shift-period but clk2 is delayed by d_ϕ.

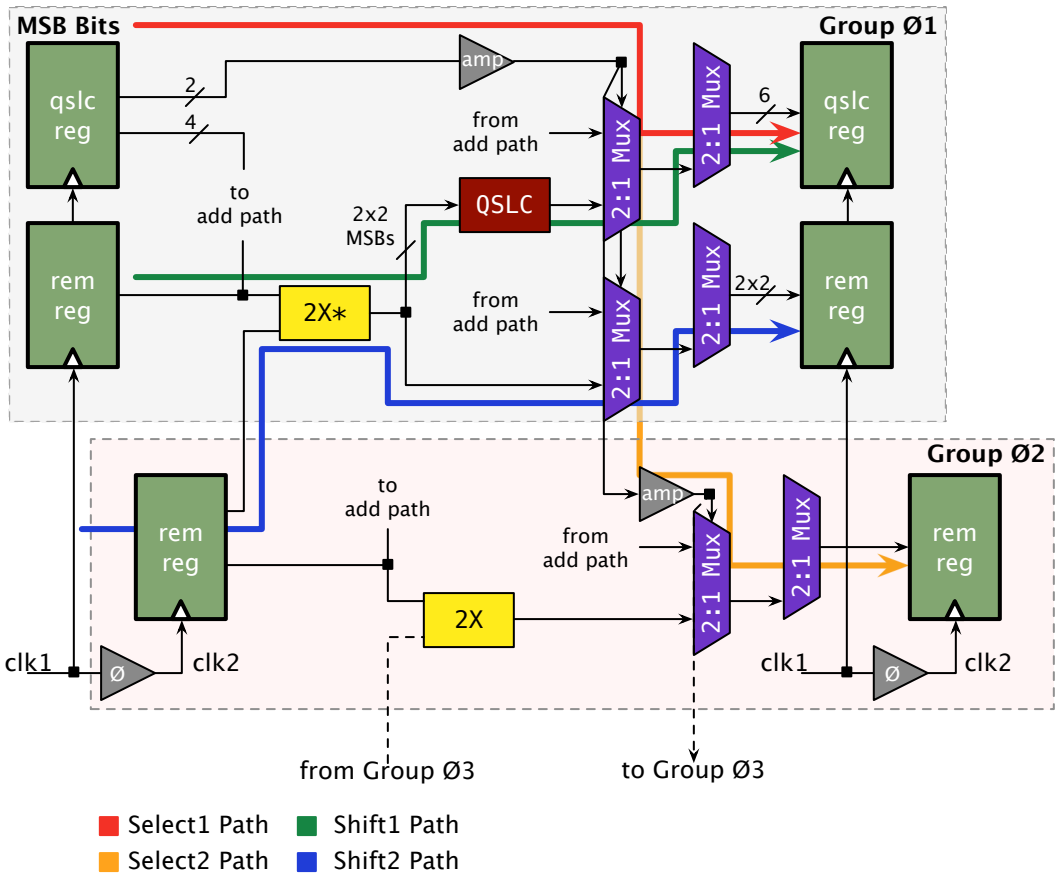


Fig. 5.2.5: The select and shift paths for the T2D5 data path with the glissando technique.

Timing diagram in Figure 5.2.6 shows the launching and capturing clocks associated with each path. Clk1 launches the bits in select1, select2 and shift1 paths, and clk2 launches the bits for the shift2 path. Clk1 captures the bits from the select1, shift1 and shift2 paths, and clk2 captures the bits from the select2 path. The delays

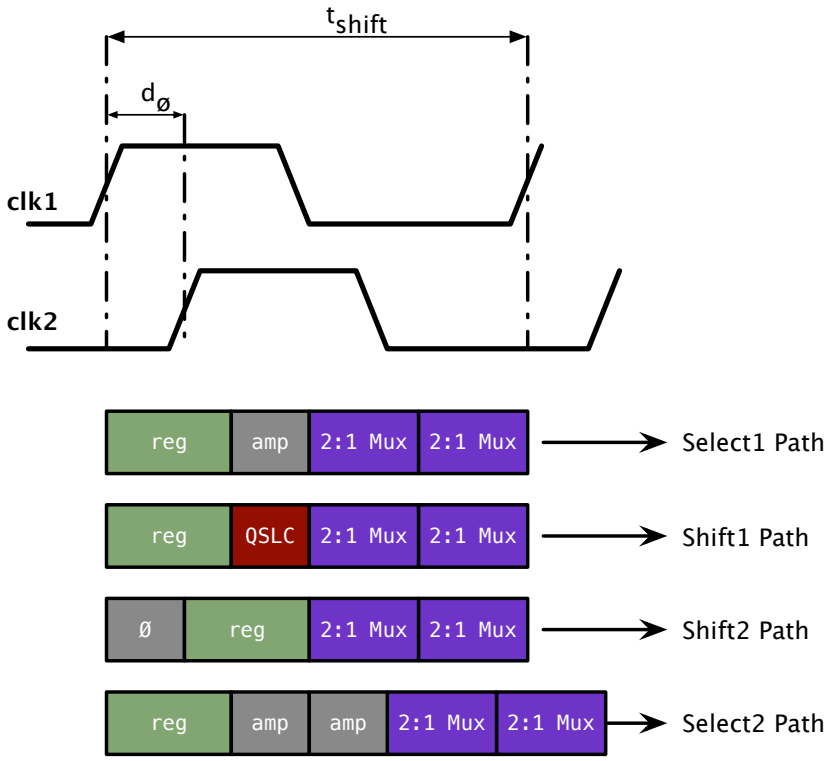


Fig. 5.2.6: Timing requirement for the bits in select and shift paths in Figure 5.2.5.

of select1, select2, shift1 and shift2 paths are calculated as follows:

$$\begin{aligned} d_{select1\text{-}path} &= d_{reg} + d_{amp} + d_{2:1Mux} + d_{2:1Mux}, \\ d_{shift1\text{-}path} &= d_{reg} + d_{qslc} + d_{2:1Mux} + d_{2:1Mux}, \\ d_{shift2\text{-}path} &= d_{\phi} + d_{reg} + d_{2:1Mux} + d_{2:1Mux}, \\ d_{select2\text{-}path} &= d_{reg} + d_{amp} + d_{amp} + d_{2:1Mux} + d_{2:1Mux}. \end{aligned} \tag{5.2.6}$$

The shift-period, t_{shift}, is

$$t_{shift} \geq MAX(d_{sel1}, d_{shift1}, d_{shift2}) \tag{5.2.7}$$

From Section 4.4.2, we know that $d_{shift1} = 6.6FO4$ and $d_{shift2} = d_{\phi} + 4.9\ FO4$. Assuming $\phi = d_{amp}$, we can estimate d_{amp} by subtracting d_{sel1} from d_{sel2}. The

delays, d_{sel1} and d_{sel2}, depend on how the bits are distributed across different phase groups.

Figure 5.2.7 shows a plot of delay d_{sel2} as a function of number of phase groups. Choosing a configuration with 10, 40, 160 and 128 bits in groups $\phi 1$, $\phi 2$, $\phi 3$, and $\phi 4$ gives $d_{sel2} = 6.7$ FO4 and $d_{sel1} = 4.8$ FO4 resulting in $d_{amp} = d_{\phi} = 1.9$ FO4. The value of d_{amp} is the same for different configurations of bit distribution. Substituting the values of d_{sel1}, d_{shift1} and d_{shift2} in Equation (5.2.7) yields $t_{shift} \geq 6.8FO4$.

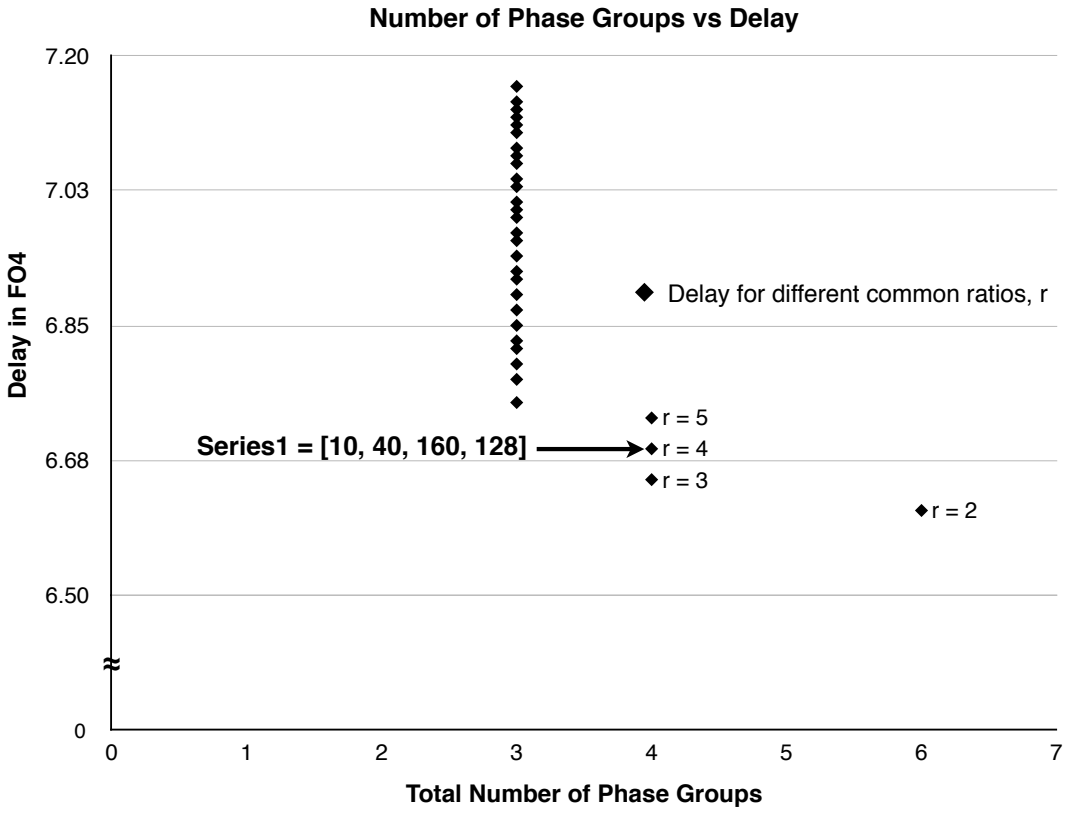


Fig. 5.2.7: Delay of select2 path as function of total number of phase groups for the T2D5 data path. Different data point represents different value for common ratio, r.

To estimate the add period, consider Figure 5.2.8 and the following two add paths: add1 and add2 paths. Figure 5.2.8 shows add1 and add2 paths with red and blue

lines. The delays of add1 and add2 paths are calculated as follows:

$$d_{add1} = d_{reg} + d_{4:1Mux} + d_{csa} + d_{qslc} + d_{2:1Mux} + d_{2:1Mux},$$
$$d_{add2} = d_{\phi} + d_{reg} + d_{csa} + d_{qslc} + d_{2:1Mux} + d_{2:1Mux}. \quad (5.2.8)$$

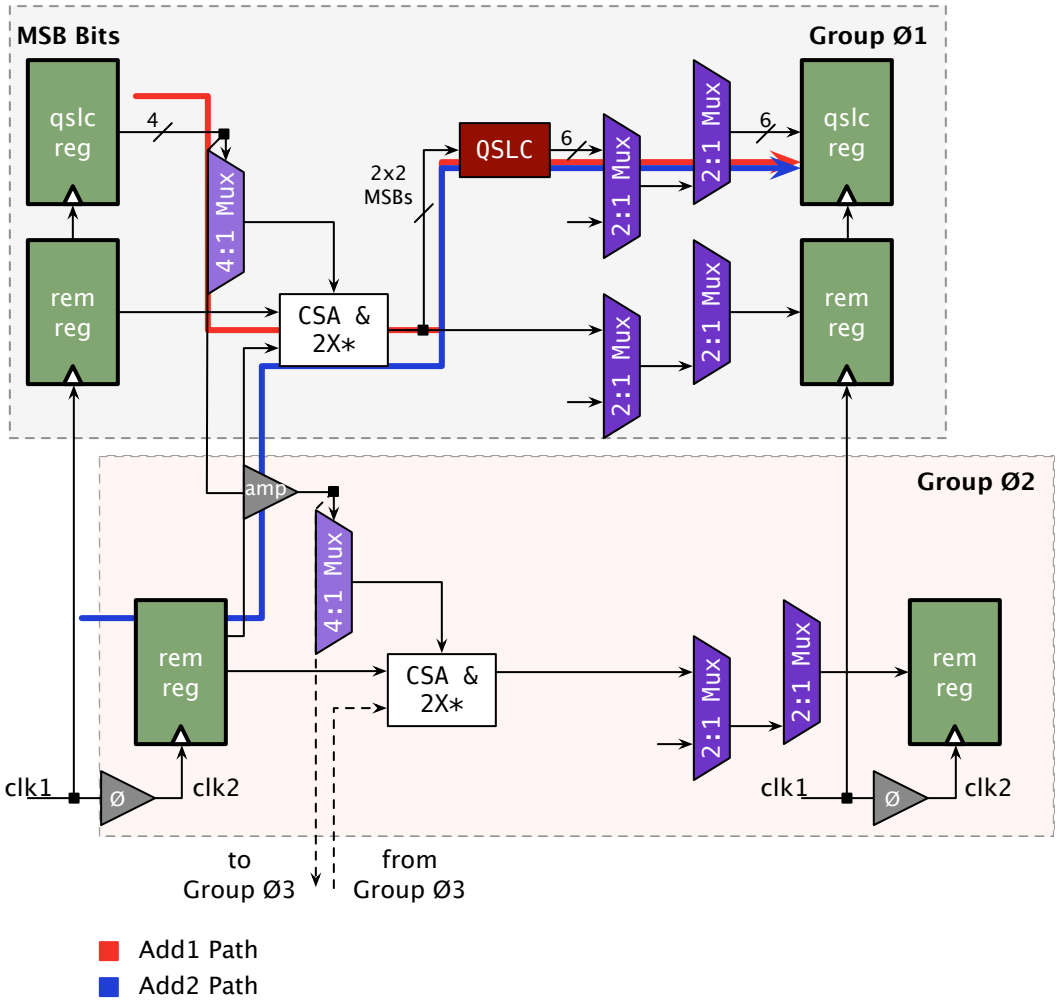


Fig. 5.2.8: The add paths for the T2D5 data path with the glissando technique.

The add period, t_{add}, is

$$t_{add} \geq MAX(d_{add1}, d_{add2}). \quad (5.2.9)$$

Using the configuration [10, 40, 160, 128], $d_{add1} = 10.7$ FO4 and $d_{\phi} = 1.9$ FO4. From Section 4.4.2, we know that $d_{add2} = d_{\phi} + 8.6$ FO4 $= 10.5$ FO4. Consequently, $t_{add} \geq 10.7FO4$.

5.3 Determining A Race Condition in Glissando

In glissando, we need to examine for race conditions, that is, can the select signals for the multiplexers generated in i^{th} clock cycle can override the select signals generated in $(i\text{-}1)^{th}$ cycle?. For example, in T1D1 data path, race occurs if the following condition holds true

$$d_{qslc-wc} \geq t_{clk} + d_{qslc-bc}, \tag{5.3.1}$$

where $d_{qslc-wc}$ and $d_{qslc-bc}$ are the worst and best case delays of the QSLC module associated with certain input patterns and transitions. Static timing analysis tools take input transitions into account when estimating the delay, but fail to consider input patterns. A dynamic simulation of the QSLC with different input patterns may be necessary.

Alternatively, we can guarantee that no race occurs if we can guarantee that the QSLC signals produced during $i\text{-}1$ cycle arrive at a multiplexer in the least-significant bit position before the i^{th} cycle begins. This condition translates to a limit on the number of phase groups as follows:

$$\text{number of phase groups} \leq \frac{t_{clk}}{\phi}. \tag{5.3.2}$$

For the T1D1 data path, the number of phase groups should be less than five, we have four phase groups. For the T2D5 data path, we have to consider shift and add periods separately. For $t_{clk} = t_{add}$, the number of phase groups should less than six. For $t_{clk} = t_{shift}$, the number of phase groups should less than four. Currently there are four phase groups in the T2D5 data path.

5.4 Comparison of Optimization Techniques

Tables 5.4.1 and 5.4.2 compare the two optimization techniques presented in this chapter for the delay per iteration for the data paths T1D1 and T2D5, respectively. The delay estimates are from a static-timing analysis tool after synthesizing the two data paths using a TSMC 40nm standard cell library. In the tables, the comparison of optimization techniques is with respect to a design without any optimization.

For the data path T1D1, optimizing the register sizes reduces the delay per iteration by 11%. A design with glissando optimization alone reduces the delay per iteration by 9%. Combining the sizing and glissando optimizations reduces the delay per iteration of the data path by 15%.

Table 5.4.1: Data path T1D1: Comparison of the optimization techniques presented in this chapter. The delay estimates are from a static-timing analysis tool.

Optimization Technique	Delay per Iteration in ps	Improvement in %
No Optimization	247	
Bigger-sized MSBs	220	11
Glissando	225	9
Glissando with bigger-sized MSBs	210	15

For the data path T2D5, optimizing the register sizes reduces the average-case delay by 3%. Combining glissando with sizing optimizations reduces the average-case delay by 13%. In the data path T2D5, the two 2:1 multiplexers in sequence can be merged together to form one 3:1 multiplexer. Merging the two 2:1 multiplexers in addition to glissando and sizing optimizations reduces the average-case delay by 23%.

Table 5.4.2: Data path T2D5: Comparison of the optimization techniques presented in this chapter. The delay estimates are from a static-timing analysis tool.

Optimization Technique	Add Delay in ps	Shift Delay in ps	Weighted Average in ps	Improvement in %
No Optimization	262	209	237	
Bigger-sized MSBs	249	208	230	3
Glissando with bigger-sized MSBs	245	165	207	13
Glissando with bigger-sized MSBs and Mux optimization	220	140	182	23

5.5 Summary of Optimization Techniques

In this chapter, I presented two optimization techniques to reduce the delay of the data paths. Commercial synthesis tools such as Design Compiler, RTL Compiler etc implement register sizing optimization. The tools will choose an appropriate size of a register from a given standard cell library to meet a given timing, power and area constraints. A designer can use the sizing analysis presented in this chapter to either add gates with bigger drive strengths to the standard-cell library to achieve a certain amount of improvement or estimate how much improvement is possible with the given standard-cell library. Typically, adding more gates to a standard-cell library is a feedback to the standard-cell library developer and can potentially increase the design cycle- time.

The glissando optimization technique offers a different way to think about the high fanout problem. Compute the bits that are necessary for the very next iteration im-

mediately and the bits that are less important can come later. Without glissando, the select path sets the clock period for a synchronous design and the shift period for a self-timed design. With the glissando technique an add or shift path sets the clock period for a synchronous design, and the add and shift periods for a self-timed design. What this means is that increasing the operand word-size will have no or little effect on the clock, add or shift periods. Increasing the word-size simply means adding more phase groups. For example, a 1024-bit divider for a 1024-bit RSA crypto-system has approximately 6k bits. A divider implementing the glissando optimization will have about 8 phase groups. I suspect that at some point wire-capacitance will limit the number of bits per group.

The next chapter presents a physical design for synchronous and self-timed dividers.

6

Design Implementations

In this chapter, I present the design flows used to implement the synchronous and self-timed divider designs, the control path for the self-timed design and timing constraints for the control path. The control path design presented in this chapter can be extended to any data path implementing an `if-then-else` conditional statement.

Figure 6.0.1 shows the divider pipeline implemented in this research for both the synchronous and self-timed designs. There are two major differences between the synchronous and self-timed designs. First, the `recDP` module implements the data path T1D1 for the synchronous design and data path T2D5 for the self-timed design. Second, the control path in Figure 6.0.1 has a single clock that drives all the registers in different stages of the pipeline for the synchronous design. A clock-tree network amplifies the clock signal. Typically, a phase-lock loop (PLL) circuit generates a clock. Design of PLL is out-of-the-scope of this research. The control path for the self-timed design has a network of self-timed modules that generate synchronization pulses for each stage in a pipeline. To avoid confusion with the synchronous clock, $fire$ is used to denote the synchronization pulses in the self-timed design. In the self-timed design, the control path consists of GasP circuits that generate $fire$ signals only when required, unlike a clock that is always ON. The self-timed designs also require a buffer-

tree to amplify the $fire$ signal, but the tree distribution is on a per stage basis rather than per pipeline basis as in the case of synchronous designs.

Both the synchronous and self-timed designs implement the glissando optimization technique along with appropriately sizing the register bits.

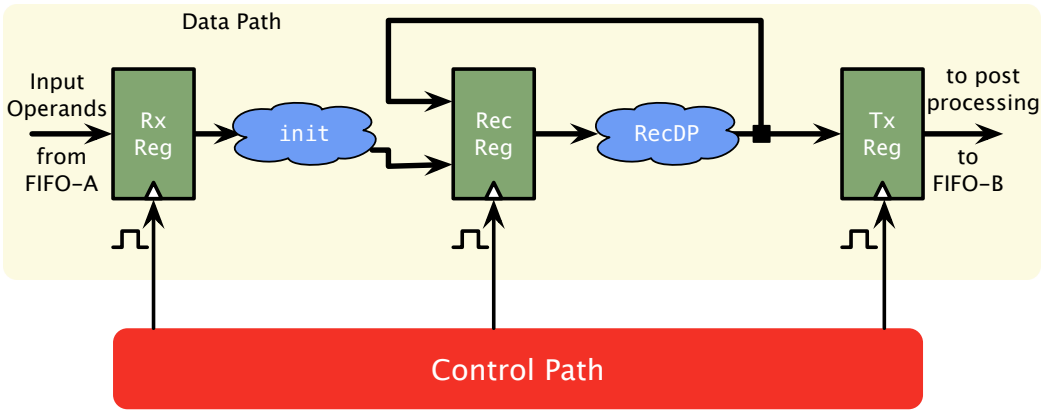


Fig. 6.0.1: Divider Pipeline: For the synchronous design, the `recDP` module implements the data path T1D1 and for the self-timed design, the `recDP` module implements the data path T2D5. The control path for the synchronous design is a clock-tree network that a single clock source drives. The control path for the self-timed design consists of GasP modules that produce pulses called $fire$ pulses only when required. The self-timed control path produces three $fire$ signals, one each for the three registers in the data path.

6.1 Design Flow

Figure 6.1.1 shows the design flow used in this research to implement the synchronous and self-timed divider designs. This research uses two design flows, one for data path and the other for control path. The synchronous divider uses only the datapath design flow but the self-timed divider uses both the design flows. A standard-cell based design flow implements the data paths for synchronous and self-timed dividers and the custom design flow implements the control path for the self-timed design flow.

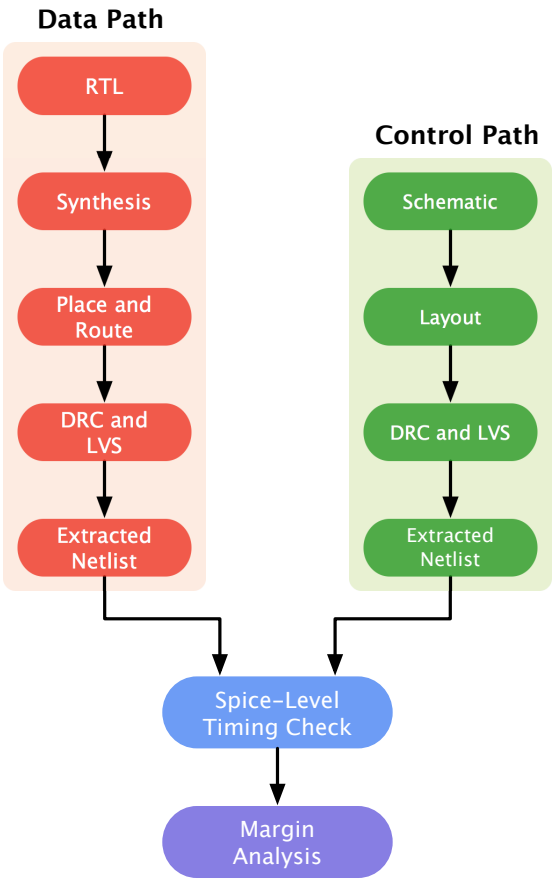


Fig. 6.1.1: Design flow to implement synchronous and self-timed dividers. The data path implementation uses a standard-cell based design flow and the control path implementation uses a custom design flow.

The standard-cell based flow begins with an RTL description of the divider. The RTL description is a structured verilog code. The synthesis tool maps the verilog description to the gates in a standard size library and performs gate sizing. The synthesized netlist is then given to the place-and-route tool. The place-and-route tool performs floor planning, cell placement, clock-tree synthesis and routing. To implement the glissando optimization technique, the clock-tree synthesis tool can be guided to insert amplifiers in the clock-tree or buffer-tree to appropriately delay the clock for different groups of bits.

The custom design flow begins with a schematic of the control path for the self-timed divider. Gate sizing and the layout of the control path is done manually to match the delays in the control path with the appropriate delays in the data path. The next section presents the design of the control path for the self-timed divider and a discussion on various timing constraints for the control path appears in Section 6.3.

6.2 Control Path

Figure 6.2.1 shows an overview of the control path for the self-timed divider. The control path consists of five modules: `Rx`, `Capture`, `kc`, `Timing` and `Tx`. These GasP modules generate pulses to enable the proper registers at appropriate times. These pulses are called $fire$ pulses. In the Figure 6.2.1, the signals $fireRx$, $fireRec$ and $fireTx$ are the $fire$ pulses and drive the registers `RxReg`, `RecReg` and `TxReg` in the data path (see Fig. 6.0.1), respectively. To take advantage of the faster-shift only operation in the data path, the period of the $fireRec$ signal must modulate according to the shift-only and add operations in the data path. For this purpose the period of the $fireRec$ signal is normally set for the shift-only operation and the period is increased for an add operation.

The following is a brief description on the behavior of the control path. Circuit details of the modules are discussed in Section 6.2.2. The control path has two loops: an inner and outer loops. The `Rx` and `Tx` modules form the outer loop. The `Rx` module receives a request for a division operation from FIFO-A and initiates a new division operation when the previous division operation completes. Upon completion of a division operation, the `Tx` module sends the result to FIFO-B for further post processing and informs the completion of the division to the `Rx` module.

The `Capture` module along with `Timing` and `kc` modules form the inner loop. The inner loop is active for i number of iterations, where i is the number of quotient bits

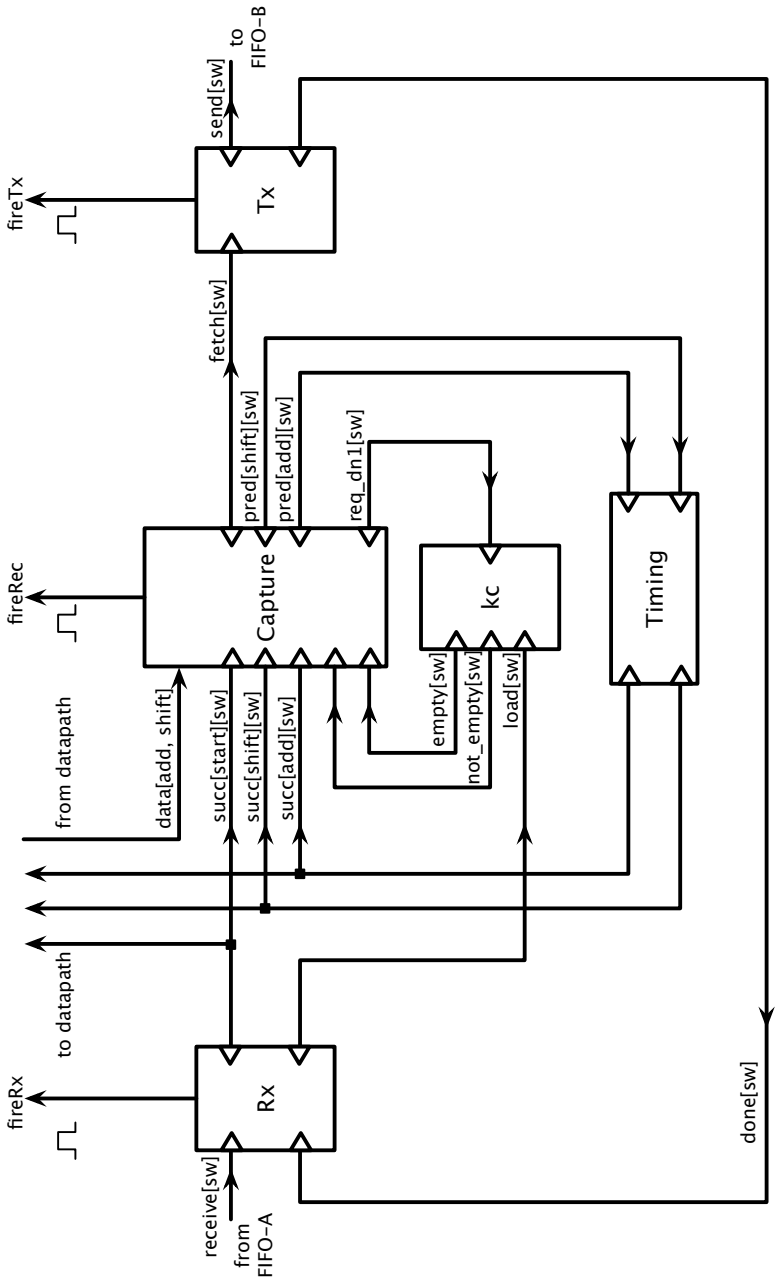


Fig. 6.2.1: Control path for the self-timed divider. The control path has two loops: an inner and outer loops. The `Rx` and `Tx` modules form the outer loop. The `Capture` module along with `Timing` and `kc` modules form the inner loop. The outer loop initiates a new division operation and the inner loop performs $L + 4$ number of iterations, where $L = 52$ and $L = 23$ for IEEE 754 double and single precision formats.

to accumulate. The `Capture` module receives two data bits, `data[add, shift]`, from the data path informing if the next iteration is an add operation or a shift-only operation. The `Capture` module encodes this information in `pred[add][sw]` and `pred[shift][sw]` wires. The `Timing` module then produces an appropriate delay for an add operation. The `kc` module is a down counter and keeps track of the number of iterations performed. A count value of zero terminates the inner loop. The wires `succ[start, shift, add][sw]` drive the select inputs of the three-input multiplexers in the data path.

The pseudo-code in Algorithm 1 describes the behavior of the control path.

```
1  if request[sw] & done[sw] then
2      i := L+4;
3      do
4          if add then
5              fireRec period = add period;
6          else
7              fireRec period = shift period;
8          end
9          i = i-1;
10     while i>0;
11 else
12     wait for request[sw] & done[sw];
13 end
```

Algorithm 1: Statements that the self-timed control path executes.

6.2.1 GasP

The control path for the self-timed divider uses a network of GasP modules. Sutherland and Fairbanks first introduced the GasP circuit family in [35]. Figure 6.2.2 shows the circuit of a simple GasP module. The `pred[sw]` and `succ[sw]` are implemented as a tri-state wire with half keepers. A tri-state wire is a wire that can be either “driven HI”, “driven LO”, or “undriven”. In the undriven state, the keeper keeps the previous

state of the wire. For this reason, wires `pred[sw]` and `succ[sw]` are also called state wires. When the GasP modules are connected in a pipeline, the `pred[sw]` state wire connects to a predecessor stage and the `succ[sw]` connects to a successor stage. In Figure 6.2.1 all the state wires have names that end with `[sw]`.

The divider's control path follows the *HI means FULL* and *LO means EMPTY* convention for the state wires. FULL and EMPTY also means request and acknowledgment respectively. During the operation, when the state wire `pred[sw]` becomes HI and the state wire `succ[sw]` becomes LO, the GasP circuit produces a brief $fire$ pulse. The $fire$ pulse performs the following three actions:

- Copies the data from input to output of the registers.
- Turns on the NMOS transistor driving the `pred[sw]` state wire LO, sending an acknowledge to the predecessor stage.
- Turns on the PMOS transistor driving the `succ[sw]` state wire HI, sending a request to the successor stage.

For the GasP circuits to work properly, all transistors must be sized such that each gate has about the same delay [36]. When properly sized, we can express the delay in terms of "gate delays". A gate delay is about 1 FO4 delay.

Figure 6.2.3 shows a timing diagram for the GasP module in Figure 6.2.2. *Forward latency* is the minimum time that a GasP module takes to pass the request to the next stage. *Reverse latency* is the minimum time that a GasP module takes to pass the acknowledgment to the previous stage. *Cycle time* is the minimum time that a GasP module takes to process the next request. The GasP module described in Figure 6.2.2 has forward and reverse latencies of six and four gate delays, respectively. Hence, the GasP module in the figure is also called 6-4 GasP module. A 6-4 GasP module has a cycle time of ten gate delays. Self-timed designs often use forward latency, reverse latency and cycle time to characterize the performance of pipelines.

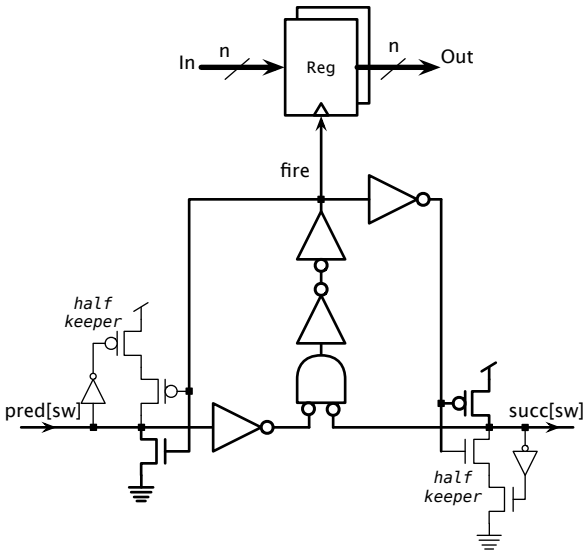


Fig. 6.2.2: A 6-4 GasP Module. The signals `pred[sw]` and `succ[sw]` are the state wires. The GasP circuit produces a brief pulse on the $fire$ signal. The $fire$ signal is usually connected to the registers in the data path. A pulse on the $fire$ signal does three things: copies the data from the input of the register to the output, drives the `pred[sw]` LO and `succ[sw]` HI. A HI on the state wire is a request to process the data along with an indication of the validity of the data in the data path. A 6-4 GasP module has forward and reverse latencies of six and four gate delays, respectively.

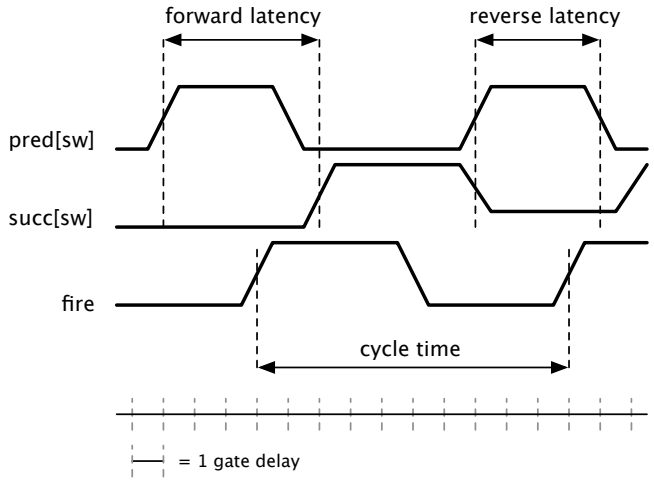


Fig. 6.2.3: Timing diagram of the signals in the 6-4 GasP module in Figure 6.2.2. A 6-4 GasP module has forward and reverse latencies of six and four gate delays respectively, and a cycle time of ten gate delays.

6.2.2 Control Path Modules

Figure 6.2.4 shows the circuit for the Rx module. The figure omits the state wire keepers. The Rx module produces a pulse on the $fireRx$ signal when both receive[sw] and done[sw] state wires are HI and both load[sw] and start[sw] state wires are LO. The start[sw] state wire connects to the Capture module and the load[sw] state wire connects to the counter module. A HI on the load[sw] state wire initializes the count value to L+4, where L=52 for double-precision and 23 for single-precision.

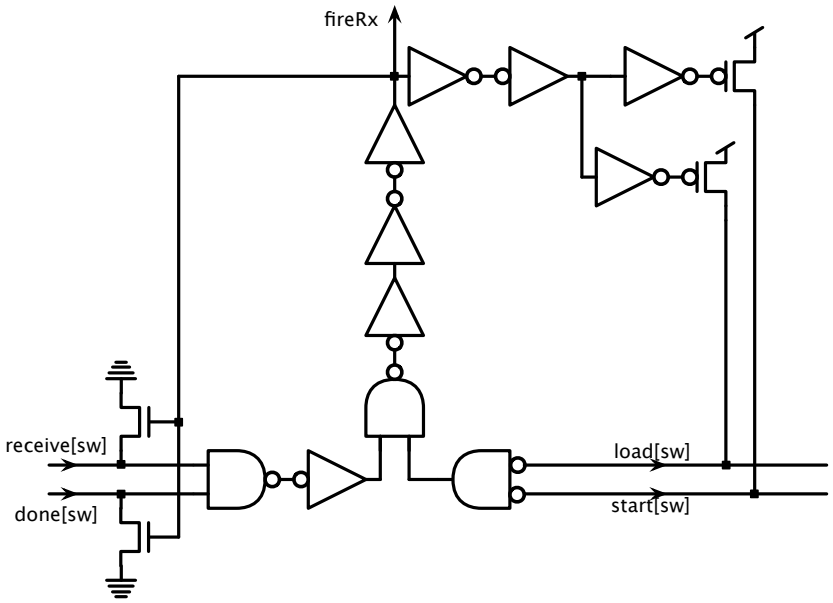


Fig. 6.2.4: The Rx module produces a pulse on the $fireRx$ signal when both receive[sw] and done[sw] state wires are HI, and both start[sw] and load[sw] state wires are LO.

Figure 6.2.5 shows the circuit for the Tx module. The Tx module produces a pulse on the $fireTx$ signal when fetch[sw] state wire is HI and both send[sw] and done[sw] state wires are LO. The input state wire, fetch[sw], comes from the Capture module. The output state wires, send[sw] and done[sw], connect to a stage in FIFO-B and the Rx module respectively.

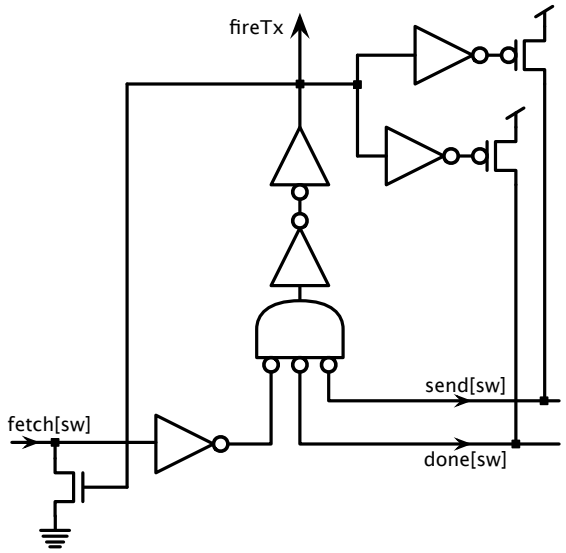


Fig. 6.2.5: The Tx module produces a pulse on the $fireTx$ signal when `fetch[sw]` is HI and `send[sw]` and `done[sw]` are LO.

Figure 6.2.6 shows the circuit for the `Capture` module. The `Capture` module has five input state wires, four output state wires and two bits of data input. The five input state wires are: `pred[add][sw]`, `pred[shift][sw]`, `pred[start][sw]`, `not_empty[sw]` and `empty[sw]`. The `pred[start][sw]` input comes from the `Rx` module. A HI on `pred[start][sw]` indicates the beginning of the new division operation. The `pred[add][sw]` and `pred[shift][sw]` indicate if the current iteration is an add or a shift-only operation. The three state wires, `pred[start, shift, add][sw]`, are mutually exclusive.

The state wires, `not_empty[sw]` and `empty[sw]`, come from the counter module and are also mutually exclusive. The counter design is based on the design described in [11]. The design details of the counter are in Appendix D. A HI on `not_empty[sw]` denotes a non-zero value in the counter, indicating that the control path has yet to complete L+4 iterations. A HI on `empty[sw]` denotes that the counter's value is zero, indicating the completion of L+4 iterations. Thus, the gates in the upper part

of the schematic (Fig.6.2.6) are active for L+4 iterations performing the following four actions:

- Generates a $fire$ pulse to copy the data from the input of the RecReg register to the output of the register in the data path.
- Captures the two data bits, data[add] and data[shift] from the data path to appropriately set the state of the output state wires succ[add][sw] and succ[shift][sw].
- Sets the output state wire req_dn1[sw] HI to request the counter to decrement by 1.
- Drives the four input state wires connected to the upper part LO. The output state wires succ[add][sw] and succ[shift][sw] connect to the Timing module.

For the one iteration the empty[sw] state-wire is HI, the Capture module simply requests the Tx stage to fetch the result by setting the fetch[sw] state-wire HI.

Figure 6.2.7 shows the circuit for the Timing module. The input state wires, pred[add][sw] and pred[shift][sw], come from the output of the Capture module. The output state wires, succ[add][sw] and succ[shift][sw], connect to the input of the Capture module. The cell with label delay implements the additional delay required to capture the result from an add operation in the data path. A string of buffers can implement this delay cell.

It is important to note that for correct operation, there must be at least two latches or a flip-flop in a loop. The recurrence loop in the data path has a flipflop. In the control path, we can think of a state wire connection between two GasP modules as a simple latch. The Capture and Timing modules together form two latches, and Capture and kc modules together also form two latches.

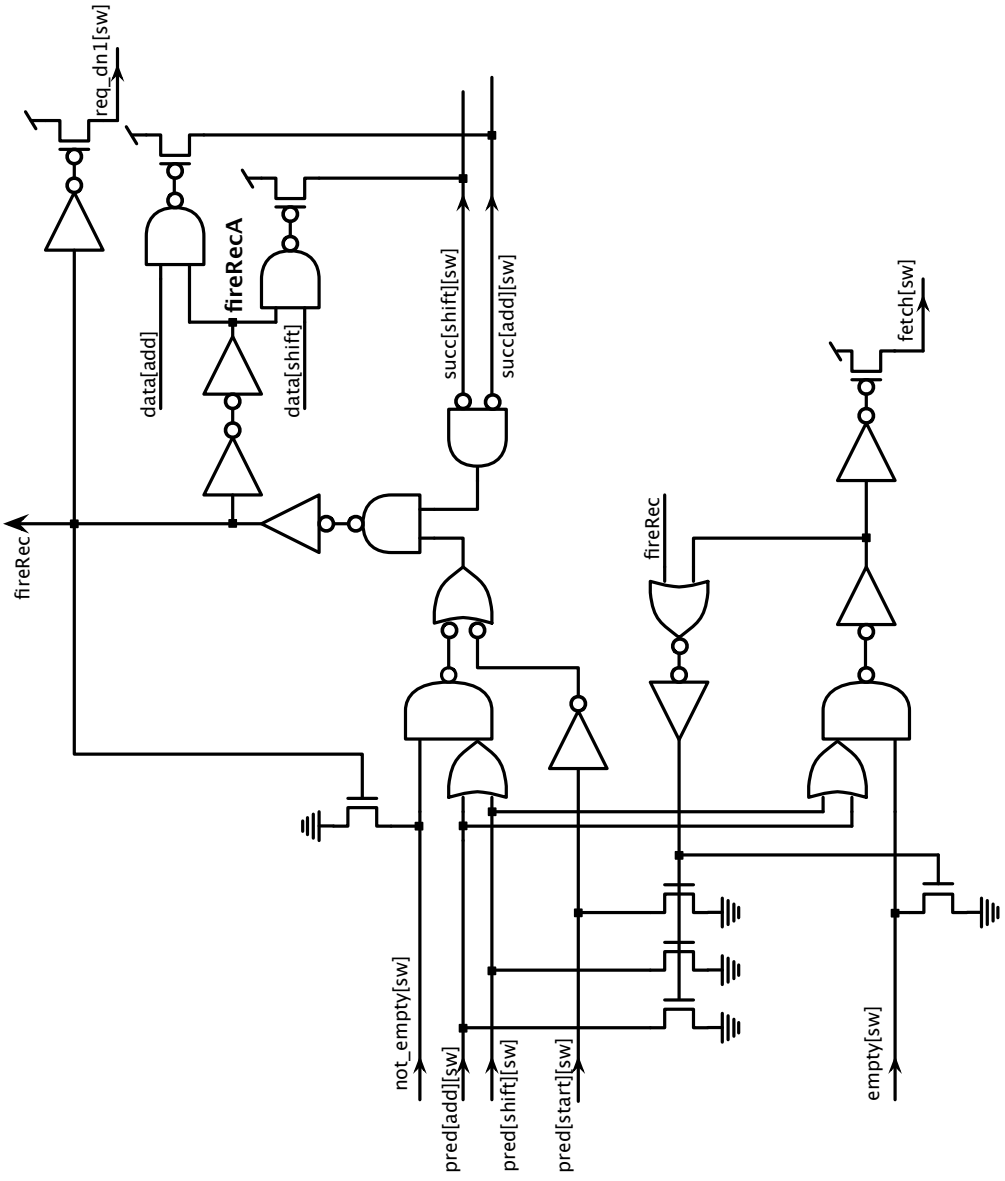


Fig. 6.2.6: The Capture module captures the `data[add]` and `data[shift]` signals from the datapath accordingly sets the state wires `succ[add][sw]` or `succ[shift][sw]` HI. State wires `succ[add][sw]` and `succ[shift][sw]` are mutually exclusive. Similarly, `pred-[add][sw]`, `pred[succ][sw]` and `pred[start][sw]` are also mutually exclusive.

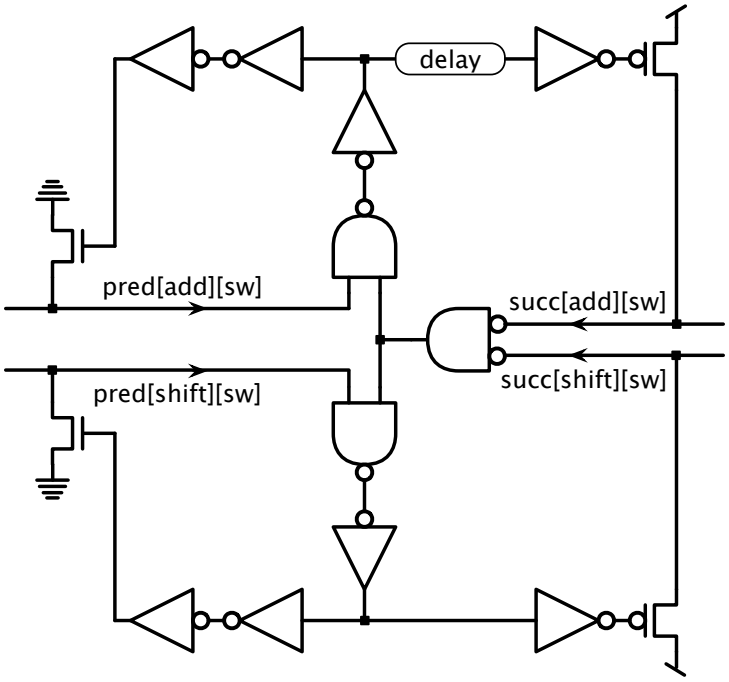


Fig. 6.2.7: The Timing module generates the necessary timing delay for the add period. The add period is the shift period plus some delay. A chain of buffers implements the `delay` module.

6.3 Timing Constraints

The control path must satisfy a set of timing constraints to ensure correct operation of the circuit. There are two sets of timing constraints. The first set of timing constraints ensure that there are no drive conflicts at the state wires and the second set of constraints satisfy setup and hold times in the data path. Drive conflicts lasting for longer duration of time at the state wire can potentially result in the creation of an invalid request or erroneously acknowledging an existing token causing incorrect behavior. Setup and hold time violations cause functional failure.

First consider the set of timing constraints to avoid drive conflicts on the state wires. In the divider control path, the important state wires to consider are `succ-`

[start,shift,add][sw], pred[add,shift][sw] and fetch[sw]. Consider Figure 6.3.1 to derive timing constraints to avoid drive conflict on succ[start][sw]. The succ[start][sw] connects to Rx and Capture modules. The figure omits the state-wire keepers and the gates in the Capture module that drive other state wires. In general, the GasP circuits have two loops, predecessor and successor loops. A loop with minimum delay sets the pulse width of the $fire$ signals. In Figure 6.3.1, the green and red lines show the predecessor and successor loops of the Rx module, and the blue line shows the predecessor loop of the Capture module. The drive conflict on succ[start][sw] occurs when $fireRx$ signal drives succ[start][sw] HI and $fireRec$ signal drives succ[start][sw] LO. We have to consider the following two cases to derive constraints to avoid the drive conflicts: source and sink limited cases. In a source limited case the Capture module waits for succ[start][sw] to go HI and in a sink limited case the Rx module waits for succ[start][sw] to go LO. Following are the constraints:

- In the source limited case, the pulse width of the $fireRx$ signal driving a state wire HI should be less than or equal to the delay of the predecessor loop that drives the state wires LO. Consequently,

$$MIN(d_{predRx},\ d_{succRx}) \leq d_{predB}, \tag{6.3.1}$$

  where d_{predRx} and d_{succRx} are the delays of the predRx and succRx loops of the Rx module, and d_{predB} is the delay of the predB loop of the Capture module.

- In the sink limited case, the pulse width of the $fireRec$ signal driving a state wire LO should be less than or equal to the delay of the successor loop that drives the state wire HI. Consequently,

$$MIN(d_{predB},\ d_{succ}) \leq d_{succRx}, \tag{6.3.2}$$

  where d_{succ} is the delay of the successor loop of the Capture module.

In the divider implementation, the sink limited case never occurs because the AND gate at the input of the `Rx` module blocks the new request to progress until the current division operation is completed. We can guarantee that by the time the division operation is complete, the `Capture` module will have drained the `succ[start][sw]` state wire. Nevertheless, the condition in Equation (6.3.3) satisfies the constraints in Equations (6.3.1) and (6.3.2).

$$MIN(d_{predRx},\ d_{succRx}) = MIN(d_{predB},\ d_{succ}). \tag{6.3.3}$$

If all the gates in the `Rx` and `Capture` modules are sized to have equal delays, then $MIN(d_{predRx},\ d_{succRx}) = MIN(d_{predCapture},\ d_{succCapture}) = 7$ gate delays.

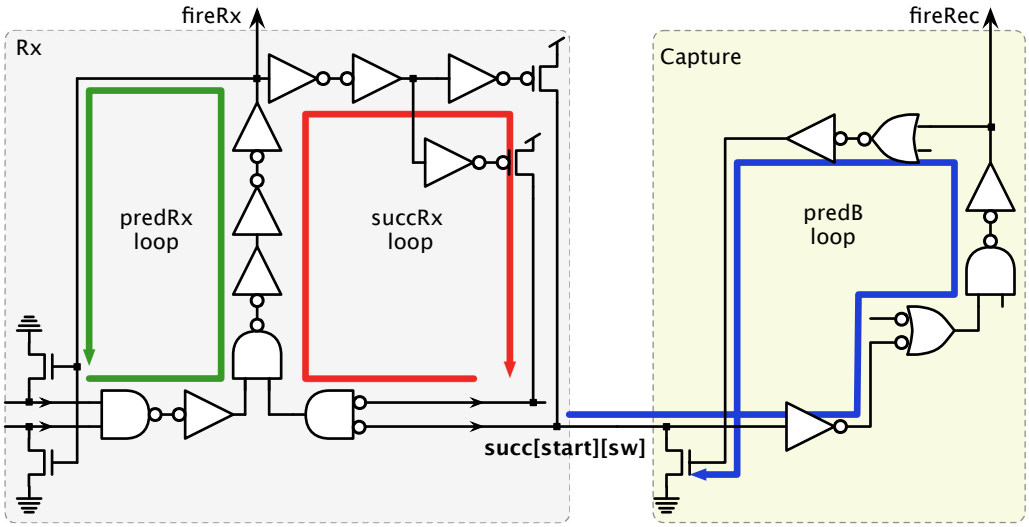


Fig. 6.3.1: The drive conflict on `succ[start][sw]` wire occurs when both $fireRx$ and $fireRec$ signal drive the state wire to opposite states. To avoid the drive conflict, the minimum delays of the predRx and succRx loops of the `Rx` module must be less than or equal to the delay of the predB loop of the `Capture` module.

Similarly, considering Figures 6.3.2 and 6.3.3 we can derive constraints to avoid drive fights on state wires `succ[add][sw]`, `succ[shift][sw]`, `pred[add][sw]`, `pred[shift][sw]` and `fetch[sw]`. Equations (6.3.4), (6.3.5) and (6.3.6) give the

constraints. In Equations (6.3.4) and (6.3.5), d_{predA}, d_{predB}, d_{succ} are the delays of the predA, predB and succ loops of the `Capture` module respectively, and d_{pred_add}, d_{pred_shift}, d_{succ_add}, and d_{succ_shift} are the delays of pred_add, pred_shift, succ_add and succ_shift loops in the `Timing` module. In Equation (6.3.6), d_{pred_empty} and d_{predTx} are the delays of the pred_empty and predTx loop in `Capture` and `Tx` modules respectively.

- For `pred[add][sw]` and `succ[add][sw]` state wires,

$$MIN(d_{predA},\ d_{predB},\ d_{succ}) = MIN(d_{pred\_add},\ d_{succ\_add}). \tag{6.3.4}$$

- For `pred[shift][sw]` and `succ[shift][sw]` state wires,

$$MIN(d_{predA},\ d_{predB},\ d_{succ}) = MIN(d_{pred\_shift},\ d_{succ\_shift}). \tag{6.3.5}$$

- For the `fetch[sw]` state wire,

$$d_{pred\_empty} \leq d_{predTx}. \tag{6.3.6}$$

If the gates in the `Capture`, `Timing` and `Tx` modules are sized to have equal delays then the control path satisfies the constraints to avoid drive conflicts. In the actual design there may be drive conflicts that may last for a few picoseconds. This is acceptable as long as the correct driver wins the conflict.

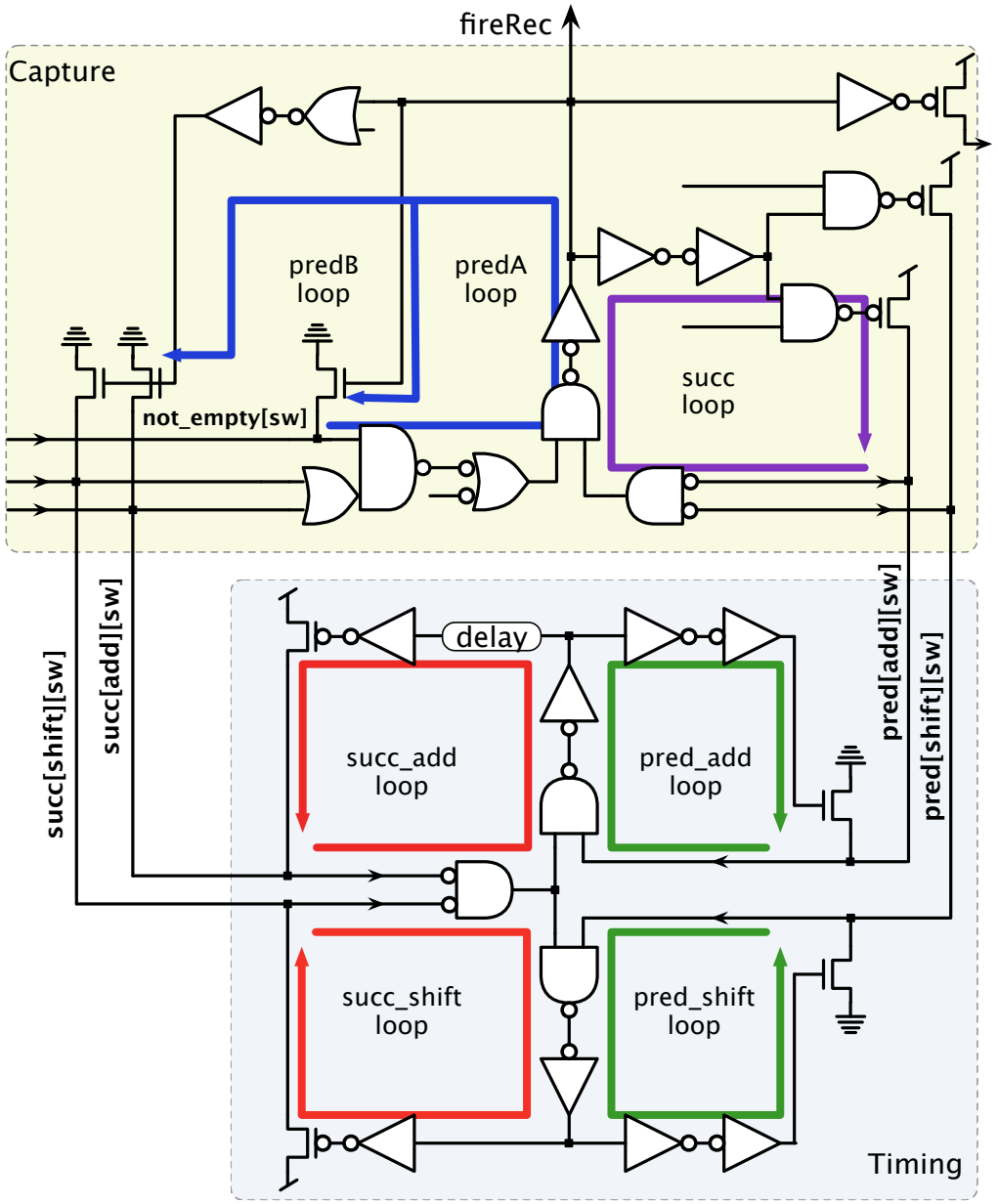


Fig. 6.3.2: To avoid the drive conflicts on `succ[add][sw]`, `succ[shift][sw]`, `pred[add][sw]` and `pred[shift][sw]` state wires, the minimum delay of the three loops in the `Capture` module should be equal to the minimum delay of the two loops in the `Timing` module for the corresponding state wire as described in Equations (6.3.4) and (6.3.5).

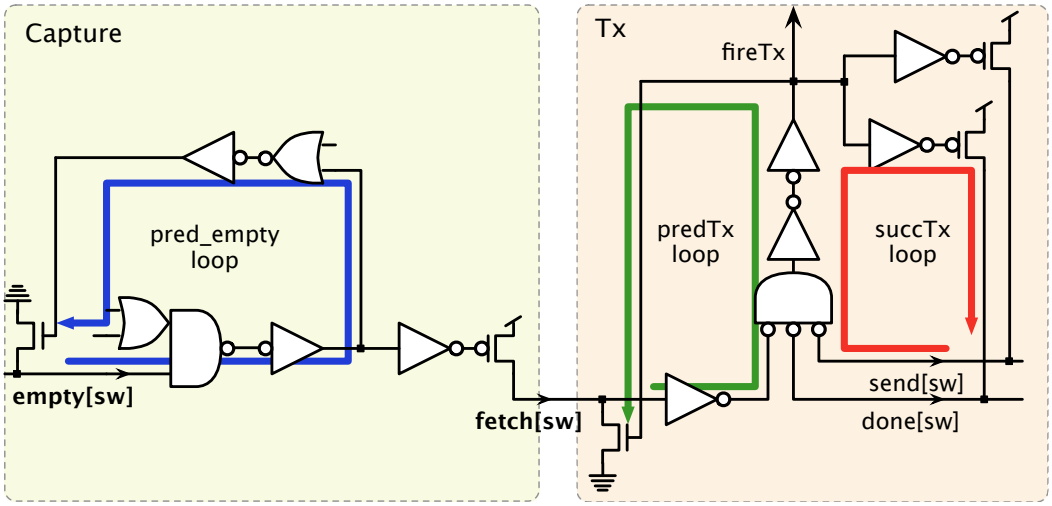


Fig. 6.3.3: To avoid drive conflicts on `fetch[sw]` state wire, the delay of the `pred_empty` loop in the `Capture` module should be less than or equal to the delay of the predTx loop in the `Tx` module.

Now let us consider the timing constraints to match the delays in the data path, which in turn translate to constraints to satisfy the setup and hold times for the registers. It is important to note that the registers in the data path are flip-flop based rather than latches. The setup and hold time violations result in functional failure. The Equations (6.3.7) to (6.3.14) give the constraints to satisfy the setup and hold times. In the equations, $\uparrow$ and $\downarrow$ denote a rising and falling transitions respectively, the function $\delta()$ denotes the time separation between two signal transitions, i is the iteration index, d with subscript denotes the delay of the module in the subscript and d_{setup} is the setup time of the flip-flop.

The Equations (6.3.7) to (6.3.10) give the constraints for the paths that are launched and captured by $fire$ pulses. The delay of a register is the delay from the clock-to-q delay.

$$\delta(fireRx\uparrow, fireRec\uparrow) \geq d_{RxReg} + d_{init} + d_{3:1Mux} + d_{setup}. \tag{6.3.7}$$

$$\delta(fireRec_i \uparrow, fireRec_{i+1} \uparrow) \geq d_{RecReg} + d_{shift-path} + d_{3:1Mux} + d_{setup}. \quad (6.3.8)$$

$$\delta(fireRec_i \uparrow, fireRec_{i+1} \uparrow) \geq d_{RecReg} + d_{add-path} + d_{3:1Mux} + d_{setup}. \quad (6.3.9)$$

$$\delta(fireRec \uparrow, fireTx \uparrow) \geq d_{RecReg} + d_{amp} + d_{setup}. \quad (6.3.10)$$

The three state wires, `succ[start,add,shift][sw]`, drive the select input of the 3:1 multiplexers. If the select inputs arrive last, then we have to consider the paths that are launched by the state wires and captured by the $fireRec$ pulse and the equations in (6.3.11) give the constraints for such paths.

$$\begin{aligned} \delta(succ[start][sw] \uparrow, fireRec \uparrow) &\geq d_{3:1Mux} + d_{setup}. \\ \delta(succ[add][sw] \uparrow, fireRec \uparrow) &\geq d_{3:1Mux} + d_{setup}. \\ \delta(succ[shift][sw] \uparrow, fireRec \uparrow) &\geq d_{3:1Mux} + d_{setup}. \end{aligned} \quad (6.3.11)$$

Every path in the data path must go through at least a 3:1 multiplexer or an amplification stage. As long as the sum of the delays of a flip-flop and 3:1 multiplexer or sum of the delays of a flip-flop and amplification stage is greater than the hold time of a flip-flop, we can guarantee the absence of hold time violations.

The Equations (6.3.12), (6.3.13), and (6.3.14) ensures that the `Capture` module in the control path captures the valid data bits, `data[add,shift]`, from the data path. We can think of the constraints in Equations (6.3.12) and (6.3.13) as a setup time constraint to capture the data bits. The constraints in Equations (6.3.12) and (6.3.13) are conservative because for the `Capture` module to capture the data bits, the valid data bits must arrive before the falling edge of the $fireRecA$ pulse rather than the rising edge. The constraint in Equation (6.3.14) is a hold time constraint to ensure that the data bits hold their valid values for the entire duration of the $fireRec$ pulse.

$$\delta(fireRec_i \uparrow, fireRecA_{i+1} \uparrow) \geq d_{RecReg} + (d_{add-path} \text{ or } d_{shift-path}) + d_{3:1Mux} + d_{amp}. \tag{6.3.12}$$

$$\begin{aligned} \delta(succ[start][sw] \uparrow, fireRecA \uparrow) &\geq d_{3:1Mux} + d_{amp}. \\ \delta(succ[add][sw] \uparrow, fireRecA \uparrow) &\geq d_{3:1Mux} + d_{amp}. \\ \delta(succ[shift][sw] \uparrow, fireRecA \uparrow) &\geq d_{3:1Mux} + d_{amp}. \end{aligned} \tag{6.3.13}$$

$$\begin{aligned} \delta(succ[start][sw] \downarrow, fireRecA \downarrow) &\leq d_{3:1Mux} + d_{amp}. \\ \delta(succ[add][sw] \downarrow, fireRecA \downarrow) &\leq d_{3:1Mux} + d_{amp}. \\ \delta(succ[shift][sw] \downarrow, fireRecA \downarrow) &\leq d_{3:1Mux} + d_{amp}. \end{aligned} \tag{6.3.14}$$

In addition to the timing constraints discussed thus far, the states of the internal state-wires before the start of the division operation and after completion of the division operation must be the same for the correct operation of successive requests. Table 6.3.1 lists the states of the state wires before the beginning of a division operation and after the completion of the division operation. The state wires `receive[sw]` and `send[sw]` connect to other FIFOs and can be considered as external state wires.

Table 6.3.1: States of the state wires before the start of the division and after completion of the division.

State wire	Before Initialization	After Completion
`receive[sw]`	HI	HI or LO
`done[sw]`	HI	HI
`load[sw]`	LO	LO
`succ[start][sw]`	LO	LO
`succ[shift][sw]`	LO	LO
`succ[add][sw]`	LO	LO
`not_empty[sw]`	LO	LO
`empty[sw]`	LO	LO
`pred[add][sw]`	LO	LO
`pred[shift][sw]`	LO	LO
`fetch[sw][sw]`	LO	LO
`send[sw]`	HI or LO	HI

7

Design Comparisons

This chapter presents a comparison of the synchronous and self-timed divider designs for latency per division, average energy per division, and area. Furthermore, the results of this research are compared with the results of other works.

Variations in process parameters result in variations in device length, threshold voltage, gate oxide thickness etc. These variations change device behavior causing yield loss. Process variation is prominent in technology nodes less than 65nm. This chapter also analyzes the response of synchronous and self-timed dividers to process variation.

7.1 Physical Design

Figures 7.1.1 and 7.1.2 show the physical design of the synchronous divider and the data path for the self-timed divider respectively. Both the dividers implement the glissando optimization technique and the figures show the cells in different phase groups in different colors. Figure 7.1.3 shows the physical design of the self-timed divider with the control path. The synchronous design and the data path for the self-timed design use cells from Oracle's 40nm standard-cell library. The control path for

the self-timed design uses custom cells to satisfy the timing constraints described in Section 6.3. I verified that all the timing constraints were satisfied using SPICE simulations.

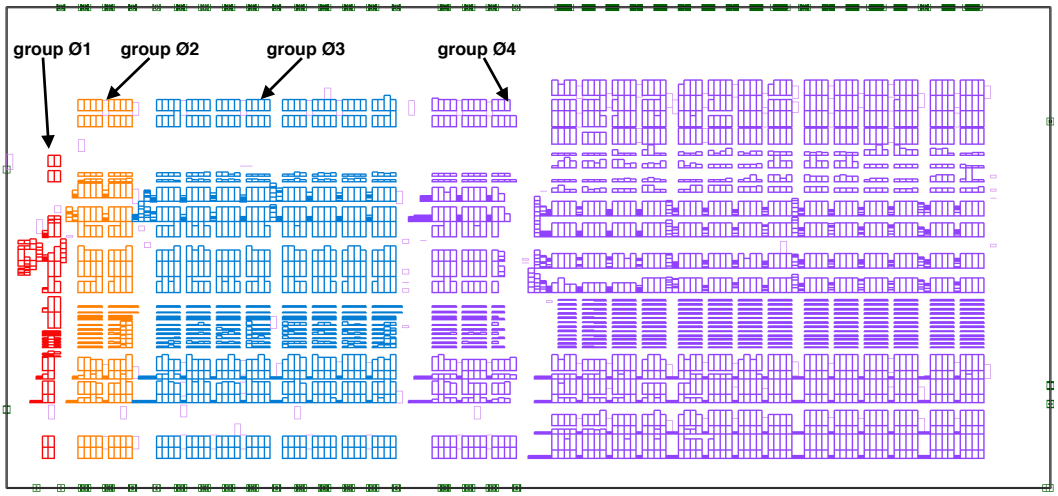


Fig. 7.1.1: Physical design of the synchronous divider. Different colors in the figure denote different phase groups.

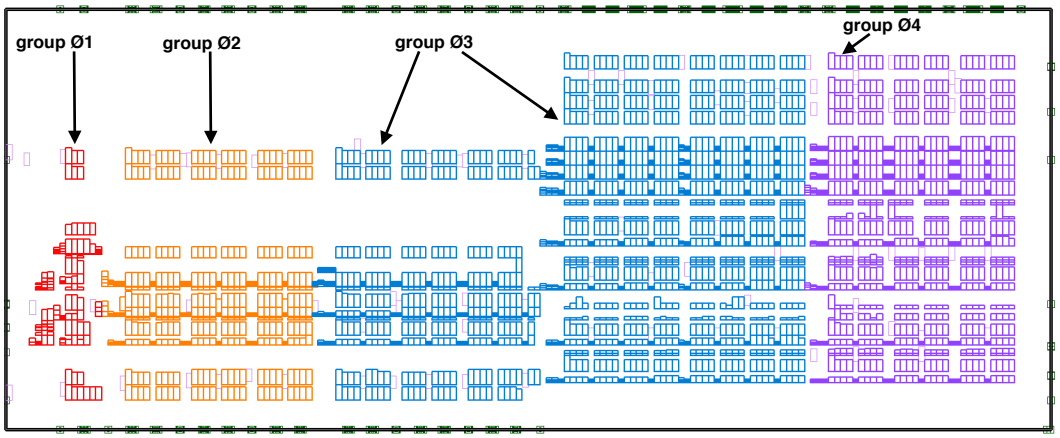


Fig. 7.1.2: Physical design of the data path for the self-timed divider. Different colors in the figure denote different phase groups.

Figures 7.1.4a and 7.1.4b show the phase-difference in the clocks and the $fireRec$ signals, respectively, in different phase groups. Figure 7.1.4b also shows the self-

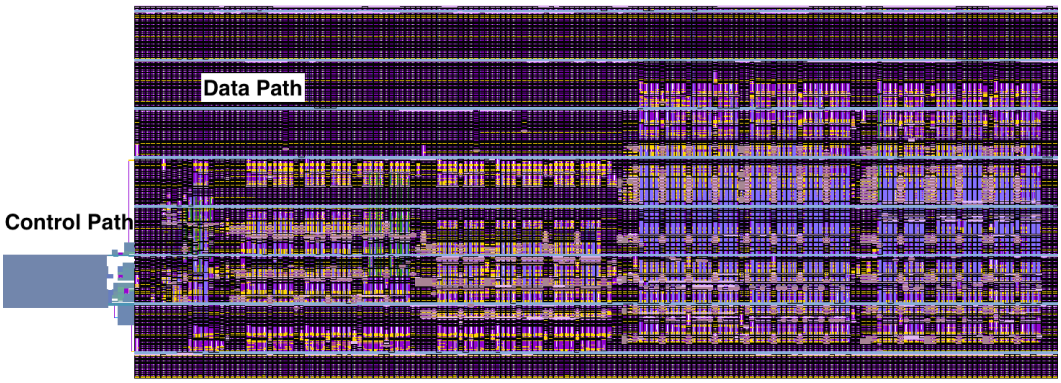
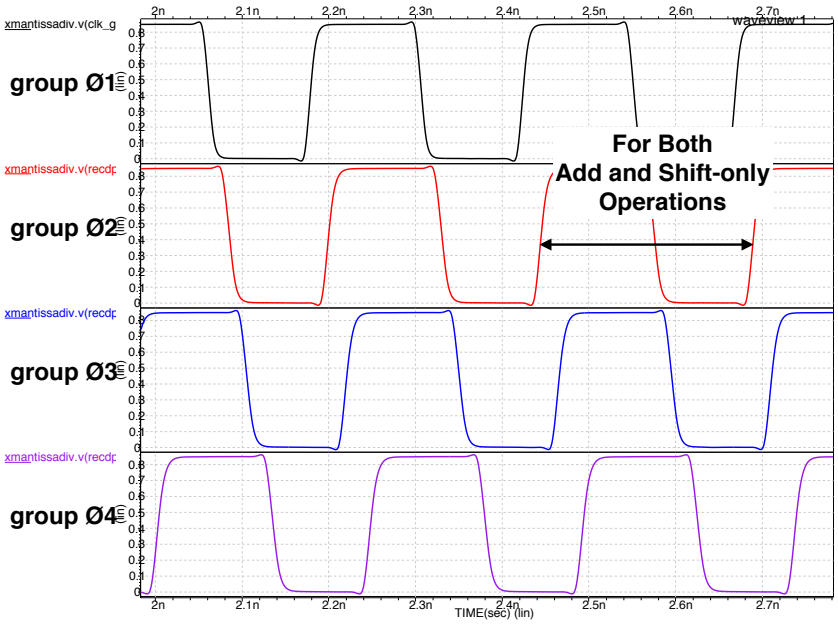


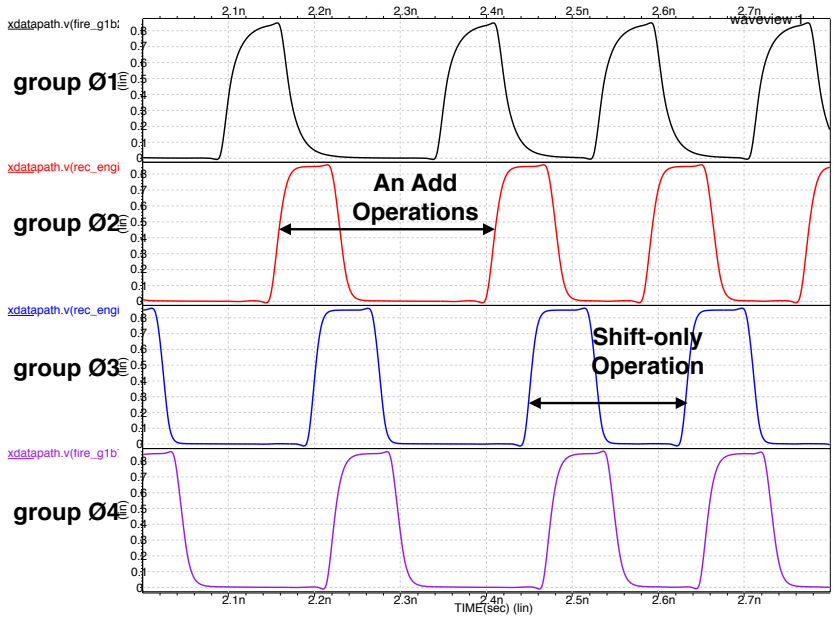
Fig. 7.1.3: Physical design of the self-timed divider along with the control path.

timed design modulating the period of the $fireRec$ signal according to an addition or a shift-only operation. Figures 7.1.5a and 7.1.5b show the arrival of select signals for multiplexers in different phase groups for synchronous and self-timed dividers.

I verified the functionality of the circuit by comparing the result of the SPICE simulation with the result of the verilog simulation for both synchronous and self-timed designs.

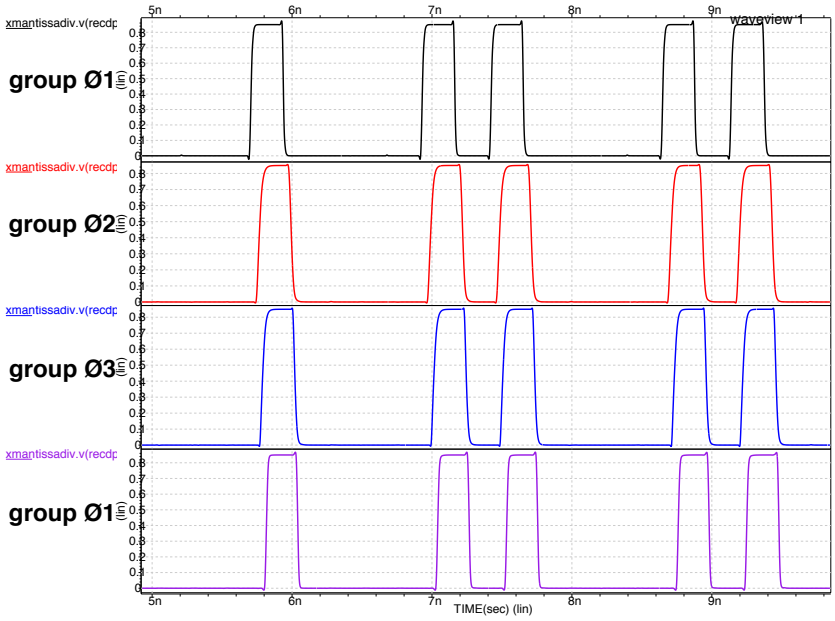


(a) Waveform showing the phase-difference of the clock in different phase groups.

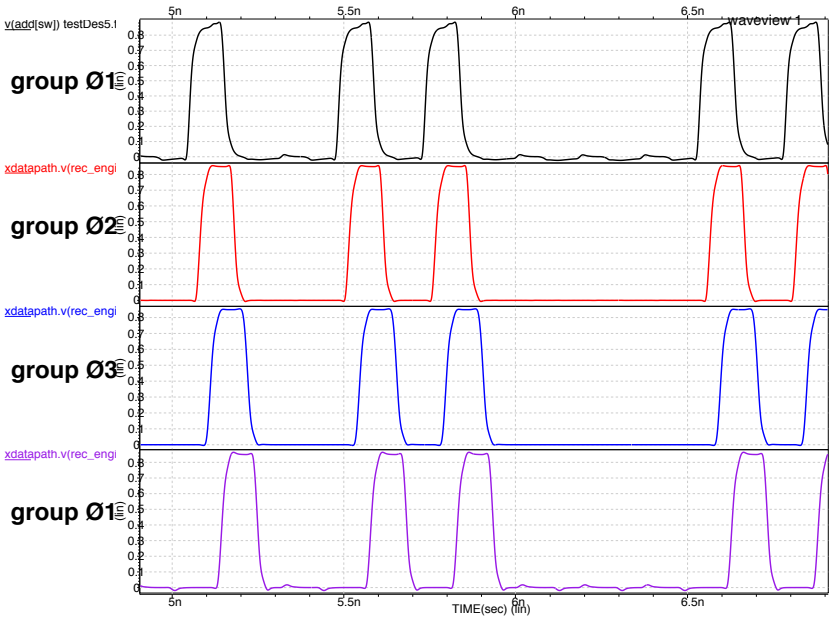


(b) Waveform showing the phase-difference of the $fireRec$ signal in different phase groups.

Fig. 7.1.4: Waveforms showing: (a) clocks in different phase groups and (b) $fireRec$ signals in different phase groups. The self-timed design modulates the period of the $fireRec$ signal according to an addition or a shift-only operation.



(a) For synchronous design.



(b) For self-timed design.

Fig. 7.1.5: Waveforms showing the arrival of the select-signals for the multiplexers in different phase groups.

7.2 Comparison of Divider Designs

This section compares the synchronous and self-timed divider designs for the following three figures of merit: latency per division, average energy per division and area. For the self-timed design we have to consider the average-case latency per division.

For the synchronous design, the latency per division is

$$L_{div} = ((L+4)+2) * D_{iter}, \tag{7.2.1}$$

Where, $L+4$ is the number of quotient digits to accumulate, D_{iter} is the delay per iteration or the clock-period, and plus 2 is the two clock cycles, one each to receive the new operands and send to result for further post processing. The clock period of the synchronous design is $D_{iter} = 245ps$.

For the self-timed design, the average latency per division is the delay from the rising transition of the $fireRx$ signal to the rising transition of the $fireTx$ signal. Therefore,

$$\begin{aligned} L_{div\text{-}avg} =& \delta(fireRx\uparrow, fireRec_1\uparrow) + \sum_{i=2}^{L+5} \delta_{avg}(fireRec_i\uparrow, fireRec_{i+1}\uparrow) \\ &+ \delta(fireRec_{L+6}\uparrow, fireTx\uparrow) \end{aligned} \tag{7.2.2}$$

In the above equation, the first and last terms on the right hand side of the equation are the delays associated with receiving the new operands and sending the result of division for post processing. The delay $\delta_{avg}(fireRec_i\uparrow, fireRec_{i+1})$ is the weighted average of the add and shift periods.

The average energy per division is the product of latency per division and average power consumption per division. Table 7.2.1 compares the three figures of merit for synchronous and self-timed designs. From the table it is clear that the self-timed design offers improvement in all three figures of merit. On average, the self-timed divider is approximately 10% faster, consumes 42% less energy and 20% less area

compared to the synchronous design. The clock period and delays of the $fire$ signals in the table include a 20ps margin to account for the uncertainty in the arrival of the clock and $fire$ pulses. It is interesting to note that the 10% difference in speed between synchronous and self-timed designs persisted throughout various stages of design implementation, that is right from the logical effort calculations to the SPICE simulation of the extracted netlists.

Table 7.2.1: Comparison of synchronous and self-timed designs.

Design	**Latency per Division,** L_{div}**, in** ns	**Average Energy per Division,** E_{div} **in** pJ	**Area in** mm^2
Synchronous	13.9	26	0.10
Self-Timed	12.5	15	0.08

To make a fair comparison of the divider designs developed in this research with other published designs, I do the following:

- Delay per quotient bit is used as a figure of merit for speed, so that a radix-2 design can be compared with a radix-4 design. Comparisons with higher-radix designs is limited to radix-4. To calculate the delay per quotient bit, I omit the delays associated with receiving the new operands and sending the result for post-processing, and the 20ps of margin. Furthermore, the delays are normalized to a technology-independent metric of fanout-of-4 or FO4 delays.

- For energy comparisons, the power or energy numbers are normalized to a 1V supply process.

- Additionally, some works report data from a fabricated chip and post-layout simulations but others report data from pre-layout simulations and logical effort calculations. To make fair comparison, I segregate the results from a chip and

post-layout simulations from pre-layout and logical effort calculations. The designs that report delays in arbitrary units of gate delays are omitted from the comparison.

For the 40nm process technology used in the simulations, 1FO4 $\approx$ 22ps at typical-typical, low-voltage and high temperature corner. Tables 7.2.2 and 7.2.3 compare the two divider designs developed in this research with other works. In the tables, a cell with label N.R denotes "Not Reported".

Table 7.2.2 compares the results of the two divider designs developed in this research with other designs. From Table 7.2.2 we can make the following observations. Williams and Horowitz's divider design in [40] remains one of the faster designs. There are two reasons for this: first, the divider design uses dual-rail domino circuits and self-timed control path. Domino circuits are approximately 1.5-1.7x faster than the static CMOS circuits. Second, the divider design in [40] uses a ring of five stages to hide the sequencing overhead. Sequencing overhead is the delay in the flip-flops or latches. The works presented [29] and [20] are extension of Williams and Horowitz's divider design. Renaudin et al., in [29] used Low-power Differential Cascode Voltage Switch Logic (LDCVSL) circuits and a ring of three stages. Matsubara et al., in [20] used a ring of four stages and used DCVSL circuits only in the critical path to reduce both cycle time and power consumption. However, usage of dynamic circuits in data path is highly discouraged in process nodes less than 90nm because of the high power and leakage issues associated with the dynamic circuits.

The divider designs developed in this research compares favorably with other divider designs that use static CMOS circuits in terms of delay per quotient digit and energy per division. Prabhu and Zyner's divider in [27] uses three radix-2 stages to build a radix-8 divider. Prabhu and Zyner overlapped some of the computation in one stage with other stages to reduce the cycle time of an iteration. Moreover, cascading three stages averages the sequencing overhead over the three stages, effectively

reducing the delay per quotient digit. The synchronous divider presented in this research offers an 8% improvement in delay per quotient digit over the divider design in [27]. The self-timed divider offers an improvement of 13% in delay per quotient digit compared to the design in [27].

The delay per quotient digit for the dividers in [26] and [31] is suspiciously high. Here is how I obtained the numbers for the delay per quotient digit. The authors in [26] and [31] reported a cycle time of 1.89ns and 1.85ns for a radix-4 divider in a 65nm process technology. Both the authors fail to report FO4 for the technology. According to [1], for 65nm process technology 1FO4 $\approx$ 13ps in typical-typical process corner at 1.2V and room temperature (27°C). Additionally, Rust and Noll in [31] do report that they used a standard cell library characterized at worst case, that is, slow-slow process corner at low voltage and high temperature. Therefore, for the designs in [26] and [31], I assumed a conservative estimate of 1FO4 $\approx$ 30ps. Using 1FO4 $\approx$ 30ps for the designs in [26] and [31], we get the numbers reported in Table 7.2.2. I advise the reader to take these numbers *"with a grain of salt"*.

In terms of improvements in energy per division, the synchronous divider of this research offers an improvement of approximately 87% and 27% over the designs in [29] and [20] respectively. The self-timed design of this research offers an improvement of 92% and 60% in energy per division compared to the designs in [29] and [20] respectively.

In Table 7.2.2, the area numbers reported are the die areas for the designs in this research and the designs in [40] and [29]. For the designs in [26] and [31] the area reported is only the standard-cell area. Moreover, the area for the designs in this research include the area of the input and output registers.

Table 7.2.3 compares the pre-layout divider designs of this research with the pre-layout divider designs in [15, 18] and [19]. In [15], Harris, Oberman and Horowitz compared various SRT implementation schemes for delay per quotient digit and area

Table 7.2.2: Post-Layout comparison of various divider implementations for delay, energy and area.

Design	Delay per quotient digit in FO4	Energy per Division in pJ	Area in mm^2	Circuit Style	Radix
Williams and Horowitz [40]	5.6 - 6.7	N.R	7	Dynamic	2
Prabhu and Zyner [27]	11.1	N.R	N.R	Static CMOS	8
Renaudin et al., [29]	16	279	0.7	DCVSL	2
Matsubara and Ide [20]	7.5	49	N.R	Static CMOS + Dynamic	2
Pham and Swartzlander [26]	31.5	201	0.02	Static CMOS with low-vt cells	4
Rust and Noll [31]	30.8	112	0.01	Static CMOS, low power cells	4
This work, synchronous	**10.2**	**36**	**0.1**	**Static CMOS**	**2**
This work, self-timed	**8.9**	**20**	**0.08**	**Static CMOS**	**2**

per bit per cycle. The non-overlapped implementation in [15] is similar in architecture and circuit family to the designs presented in this research. The delay per quotient digit in Table 7.2.3 for [15] includes the delay of sequencing overhead of 4.4 FO4 delays. Liu and Nannarelli in [18] presented two radix-4 implementations, one using only the high-speed cells and the other using a combination of high-speed and low-power cells. In [19], Liu and Nannarelli presented another radix-4 implementation that takes advantage of the clock-skew in the critical path to reduce the cycle time of an iteration.

The synchronous divider of this research offers an improvement of 38%, 46% and 10% in delay per quotient digit compared to the divider implementations in [15, 18] and [19]. In terms of energy comparisons, the synchronous divider of this research consumes approximately 87% and 44% less energy consumption per division compared to the dividers in [18] (low-power) and [19].

The self-timed divider of this research offers an improvement of 47%, 54% and 22% in delay per quotient digit compared to the designs in [15, 18] and [19], and 94% and 73% in energy per division compared to the designs in [18] (low-power) and [19].

The standard cell area of the divider designs presented in this research is in the same range as the designs in [18] and [19], and about fifteen times smaller than the design in [15]. The designs in [18] and [19] used a 90nm technology and the design in [15] used a $1\mu m$ technology.

Table 7.2.3: Pre-Layout comparison of various divider implementations for delay, energy and area.

Design	Delay per quotient digit in FO4	Energy per Division in pJ	Area in mm^2	Radix
Harris et al., [15]	13.9	N.R	0.46	4
Liu and Nannarelli [18]	16	402 and 261 (Low Power)	0.02	4
Liu and Nannarelli [19]	9.5	57	0.04	4
This work, synchronous (pre-layout)	8.6	32	0.03	2
This work, self-timed (pre-layout)	7.4	15	0.02	2

7.3 Process Variation

In this section, I will discuss how the synchronous and self-timed dividers implemented in this research respond to variations in process and environment. The two sources of environmental variations are supply voltage and temperature. The sup-

ply voltage variations result in variations in transistor current causing delay variations. The temperature variations affect the threshold voltage of a transistor also resulting in delay variations.

At die-level the process variations can be classified into two: inter-die and intra-die variations. In inter-die variations a transistor in one die behaves differently in another die. Inter-die variations are also called Die-to-Die $(D2D)$, process-shift, and global variations. In intra-die variations every transistor within a die can behave differently. Intra-die variations are also called Within-Die (WID), mismatch, and local variations.

Designers typically use SPICE-level monte carlo simulations to estimate the effects of process variation on circuit performance and functionality. The number of monte carlo runs required depends on the type of variation to analyze and number of process parameters for a given technology. For example, if we want to estimate the effect of global variation on the circuit's behavior then

$$\text{\# of runs } = \frac{(\text{\# of global parameters + 2}) * (\text{\# of global parameters + 1})}{2} \tag{7.3.1}$$

and for local variation it is,

$$\text{\# of runs } = (\text{\# of transistors}) * (\text{\# of local parameters}) \tag{7.3.2}$$

The 40nm process technology used in this research has about twenty-four global parameters and three local parameters. Estimating the local variation for a circuit with approximately fifty-thousand transistors is very time consuming because each simulation takes about five to six hours. Therefore, only global or inter-die variation is considered. To further reduce the number of simulation runs required, I developed a 2-factorial design-of-experiments (DOE).

Figure 7.3.1 shows the scatter plot of NMOS and PMOS currents for different monte carlo simulation runs. The $x\text{-}axis$ is the NMOS current and the $y\text{-}axis$ is the PMOS current. The blue, green and red ellipses are the 1σ, 2σ and 3σ density ellipses, respectively. The gray data-points are from 1024 monte carlo runs and denote

the total variation space for an NMOS and PMOS transistor. The red data-points denote the NMOS and PMOS transistor configurations that I considered for the 2-factorial DOE simulations. The 2-factorial DOE simulations allow us apply the same parameter-value configurations to both synchronous and self-timed designs. In Figure 7.3.1, the two-letter label denotes a process corner. For example, the label FF denotes fast-NMOS and fast-PMOS corner.

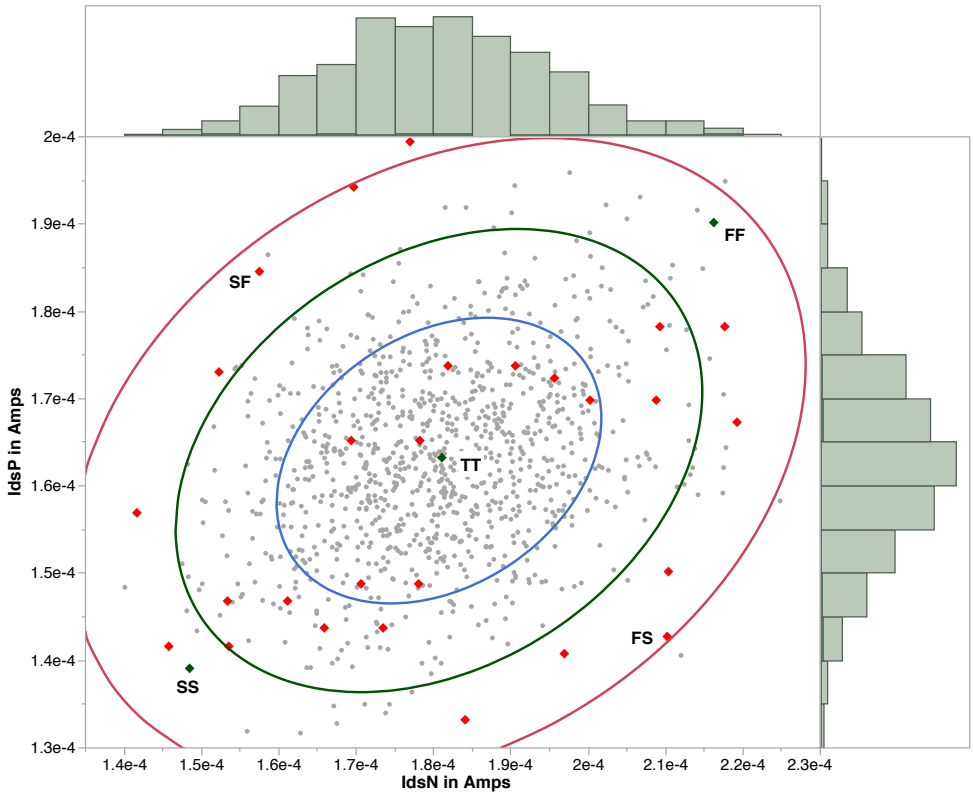


Fig. 7.3.1: Scatter plot of NMOS and PMOS currents under process variation. The $x\text{-}axis$ is the NMOS current and the $y\text{-}axis$ is the PMOS current. The gray data-points are from 1024 monte carlo runs and denote the total variation space for an NMOS and PMOS transistor. The red data-points denote the NMOS and PMOS transistor configurations that I considered for the 2-factorial DOE simulations.

Table 7.3.1 lists the number of samples collected at different process corners. In addition to process corner, I also considered the following two environmental corners:

low-voltage with high-temperature, and high-voltage with low-temperature. Labels LH and HL denote low-voltage with high-temperature corner and high-voltage with low-temperature corner, respectively. Table 7.3.2 lists the values for supply voltage and temperature for the two environmental corners. In summary, with 2-factorial DOE simulations we have 26 samples of process-variation at two different environment corners for synchronous and self-timed divider designs.

Table 7.3.1: Number of samples collected at different process corners.

Process Corner	Number of Samples
FF	8
FS	5
SF	5
SS	8
Total	**26**

Table 7.3.2: The two environmental corners considered in this research.

Environmental Corner	Voltage in volts	Temperature in ∘C
LH	0.85	105
HL	1.20	0

Table 7.3.3 summarizes the functional yield for the synchronous and self-timed divider designs. In the table, “pass” means that the result of the division matched the expected value and “fail” means that the result of the division differed from the expected value. The functional yield is the ratio of the number of “pass” to the sum of the number of “pass” and “fail”. The synchronous design has a functional yield of 77%, including both the environmental corners. The self-timed design has a functional yield of 60%, including both the environmental corners. It is interesting that the synchronous and self-timed designs fail in different environmental corners.

Table 7.3.3: Functional Yield for self-timed and synchronous divider designs at all corners

Sample Number	LH		HL	
	Self-Timed	**Synchronous**	**Self-Timed**	**Synchronous**
FF,1	pass	pass	fail	pass
FF,2	pass	pass	fail	pass
FF,3	pass	pass	fail	pass
FF,4	pass	pass	fail	pass
FF,5	pass	pass	fail	pass
FF,6	pass	pass	fail	pass
FF,7	pass	pass	fail	pass
FF,8	pass	pass	fail	pass
FS,1	pass	pass	pass	pass
FS,2	pass	pass	pass	pass
FS,3	pass	fail	pass	pass
FS,4	pass	pass	pass	pass
FS,5	pass	pass	pass	pass
SF,1	pass	fail	fail	pass
SF,2	pass	pass	fail	pass
SF,3	pass	fail	fail	pass
SF,4	pass	pass	fail	pass
SF,5	pass	fail	fail	pass
SS,1	pass	fail	fail	pass
SS,2	pass	fail	fail	pass
SS,3	pass	fail	fail	pass
SS,4	pass	fail	fail	pass
SS,5	pass	fail	fail	pass
SS,6	pass	fail	fail	pass
SS,7	pass	fail	fail	pass
SS,8	pass	fail	fail	pass
Functional Yield	**100%**	**46%**	**19%**	**100%**

The fails are interesting in terms of understanding the behavior of two different design styles. First let us consider consider the synchronous design. For the synchronous design simulations, the clock signal came from an ideal source and the clock-period remained fixed even in presence of variations. Therefore, in the slow corners at low-voltage and high-temperature, the data path is slower than the clock period resulting in setup-time violations.

For the self-timed design, the $fire$ signals came from the control path and the period of the $fire$ signals changed according to the process variation. In the FFHL and SSHL corners, the fails were because of the setup-time violations. This suggests that the control path ran at a slightly higher-speed than the data path.

Figures 7.3.2 and 7.3.3 show the cumulative distribution function of slacks for the synchronous and self-timed designs respectively. Slack is the difference between the arrival times of the data and the clock or $fire$ signals. A positive slack denotes that the data arrived before the clock and a negative clock denotes that the clock arrived before the data. A setup time of a flip-flop is the minimum slack required to correctly capture the data.

Figures 7.3.2a and 7.3.2b show the cumulative distribution function (CDF) plots of the slacks for a passing and failing instances of a synchronous divider, respectively. The blue line indicates the setup time or the minimum slack required for that instance. The failing instance in Figure 7.3.2b has a heavier tail compared to the passing instance in Figure 7.3.2a. Figures 7.3.3a and 7.3.3b show the CDF plots of the slacks for a passing and failing instances of a self-timed divider, respectively.

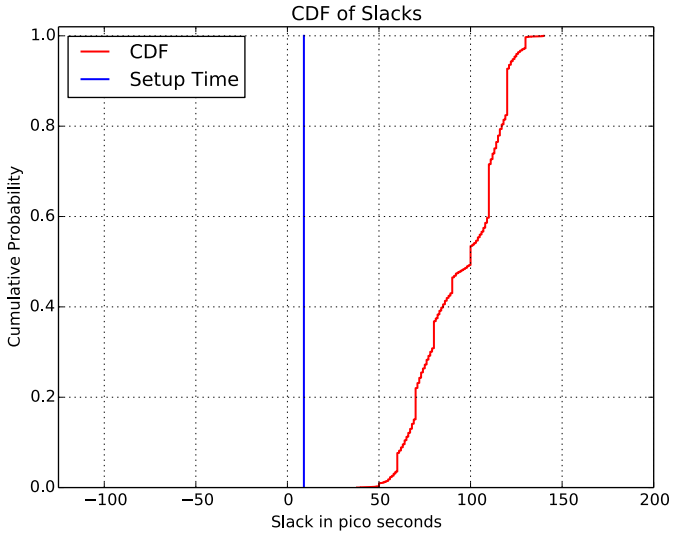
Tables 7.3.4 and 7.3.5 lists the minimum slack required and the measured minimum slack from SPICE simulations for both the divider designs at LH and HL environmental corners respectively.

Table 7.3.4: Minimum slack required and the measured slack from simulations in various process corners at low voltage and high temperature (LH).

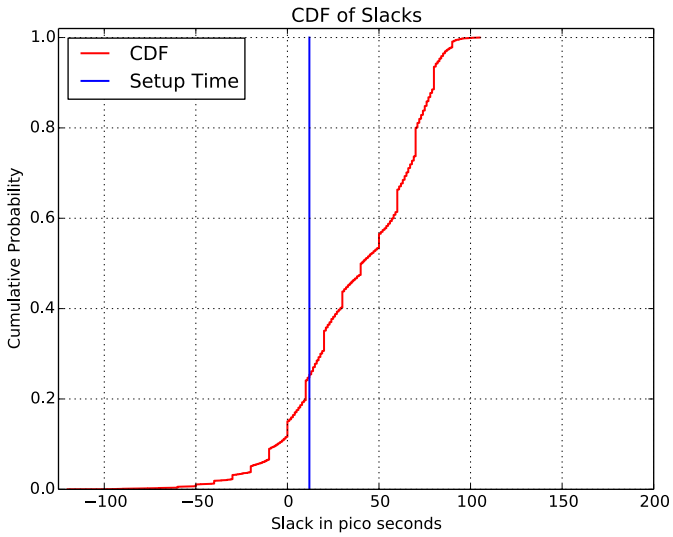
Sample Number	Minimum Slack Required in ps	Measured Slack from Simulations in ps	
		Self-Timed	Synchronous
FF,1	9	20	38
FF,2	10	20	38
FF,3	8	19	40
FF,4	9	18	40
FF,5	8	20	28
FF,6	9	27	28
FF,7	7	20	34
FF,8	7	26	35
FS,1	5	20	17
FS,2	4	20	30
FS,3	6	30	-68
FS,4	6	27	16
FS,5	5	20	27
SF,1	8	14	-63
SF,2	7	19	10
SF,3	8	25	-100
SF,4	8	18	30
SF,5	8	18	-26
SS,1	12	30	-120
SS,2	14	25	-97
SS,3	10	30	-80
SS,4	10	34	-90
SS,5	12	40	-120
SS,6	12	40	-120
SS,7	8	38	-110
SS,8	8	40	-120

Table 7.3.5: Minimum slack required and the measured slack from simulations in various process corners at high voltage and low temperature (HL).

Sample Number	Minimum Slack Required in ps	Measured Slack from Simulations in ps	
		Self-Timed	Synchronous
FF,1	10	7	116
FF,2	10	2	116
FF,3	12	5	120
FF,4	9	6	119
FF,5	9	8	110
FF,6	9	9	110
FF,7	8	7	110
FF,8	8	8	110
FS,1	7	8	108
FS,2	7	8	110
FS,3	9	11	100
FS,4	8	9	109
FS,5	6	7	110
SF,1	8	10	109
SF,2	8	9	109
SF,3	9	1	100
SF,4	8	-2	110
SF,5	8	9	110
SS,1	14	10	100
SS,2	14	10	100
SS,3	12	9	100
SS,4	12	10	100
SS,5	12	11	100
SS,6	12	12	99
SS,7	12	11	100
SS,8	12	11	100

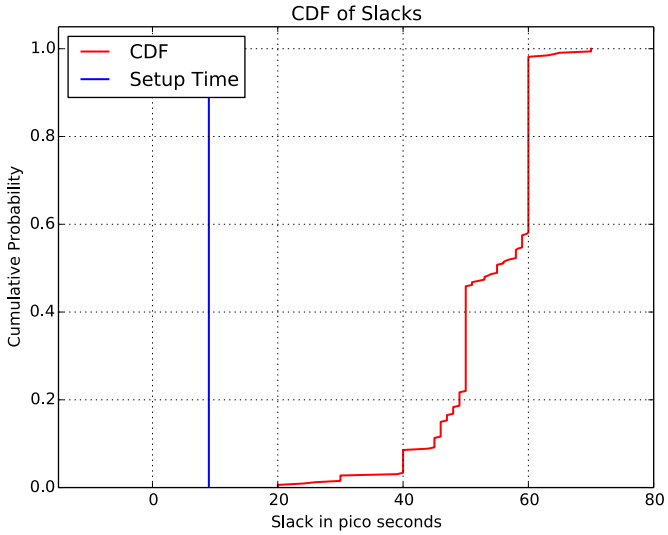


(a)

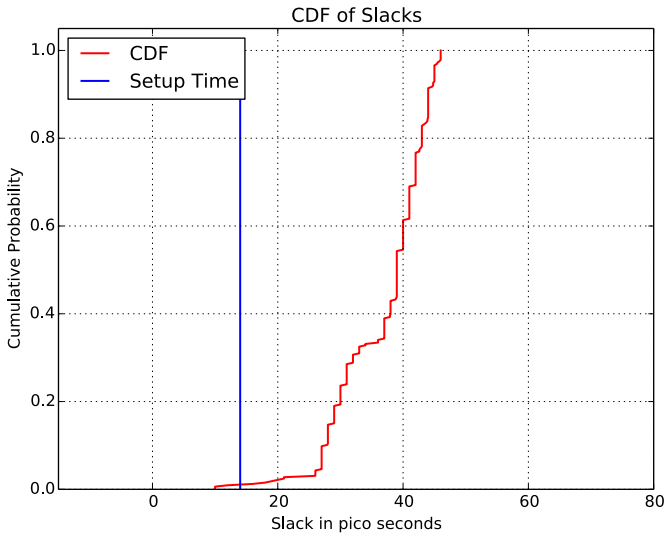


(b)

Fig. 7.3.2: Cumulative distribution function (CDF) of slacks for the synchronous divider design: (a) for a passing sample in an FFLH corner and (b) for a failing sample in a SSLH corner. The blue line indicates the setup time or the minimum slack required for that instance.



(a)



(b)

Fig. 7.3.3: CDF of slacks for the self-timed divider design: (a) for a passing sample in a FFLH corner and (b) for a failing sample in a SSHL corner. The blue line indicates the setup time or the minimum slack required for that instance.

The fails in the FSHL corner for the self-timed systems is because of two reasons. First, there were some bits with setup time violations. Second, on some occasions, the flip-flops failed to capture the data bits even when the valid data-bit arrived at the input of the flip-flops, long before the $fire$ signal arrived.

Consider Figure 7.3.4 to understand the cause of the second problem. The figure shows a differential flip-flop circuit similar to the one used in the divider design. The sense amplifier is the key component of the differential flip-flop in 7.3.4. The sense amplifier responds to small differential input voltages. When the $fire$ input is LO, the nodes x and y pre-charge to some threshold value. When the $fire$ input is HI, the data input pulls either x or y node to LO and the cross-coupled PMOS transistors act as a keeper for the other node. In the FS process corner, the input PMOS transistors, P1 and P2, are slow to pre-charge the nodes x and y. In the HL configuration, the period of the $fire$ signal is shorter than in the LH configuration giving less time for P1 and P2 transistors to act. Therefore, in the FSHL corner, the flip-flops fail to pre-charge nodes x and y to the correct value and thus failing to capture the correct data. This problem occurs only during the shift cycles, because the shift-period is less than the add-period. Figure 7.3.5 shows the waveform illustrating the failure of a flip-flop to capture the data even when the data is setup long before the $fire$ signal arrived. The second time, however, the flip-flop captures the input because the nodes x and y are pre-charged to their appropriate values. We can see in the waveform that after the first fail the data input failed to pull the nodes x and y all the way to 0 which makes it easier for the P1 and P2 PMOS transistors to pre-charge the nodes x and y to appropriate values. We can think of the problem of pre-charge as a duty-cycle constraint where the time to pre-charge limits the minimum OFF duration in the clock or $fire$ signals.

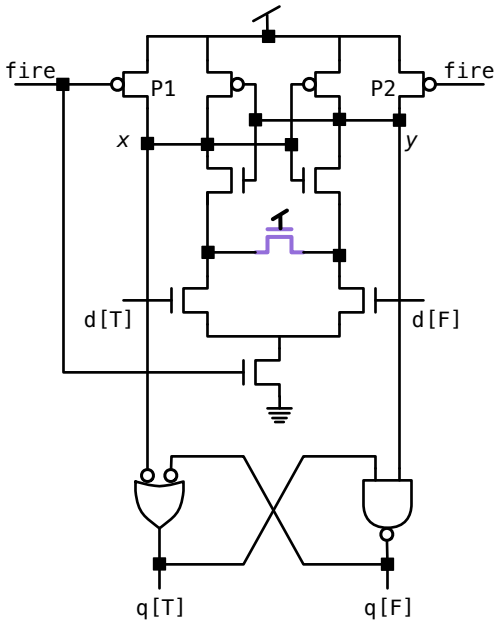


Fig. 7.3.4: A typical Differential Flop-Flop circuit [39].

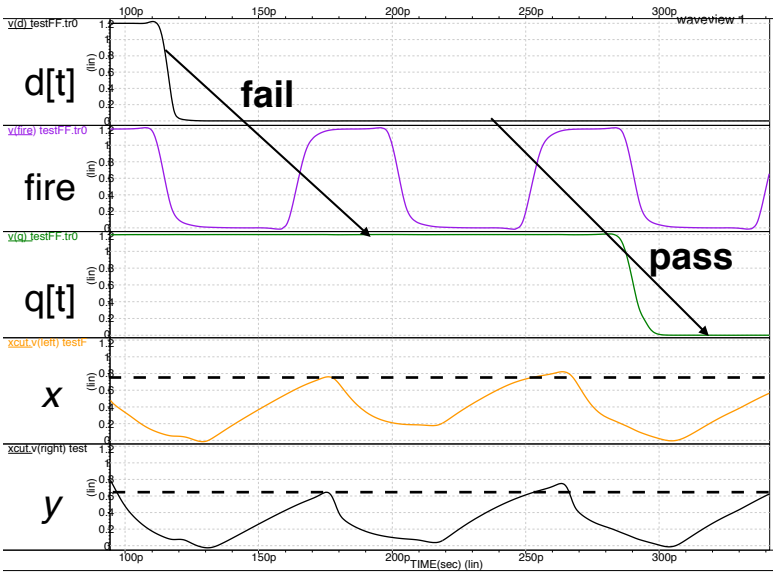


Fig. 7.3.5: Waveform illustrating the failure of the flip-flop circuit in Fig.7.3.4 to capture the data in FSHL corner.

7.3.1 Predicting Yield-Loss

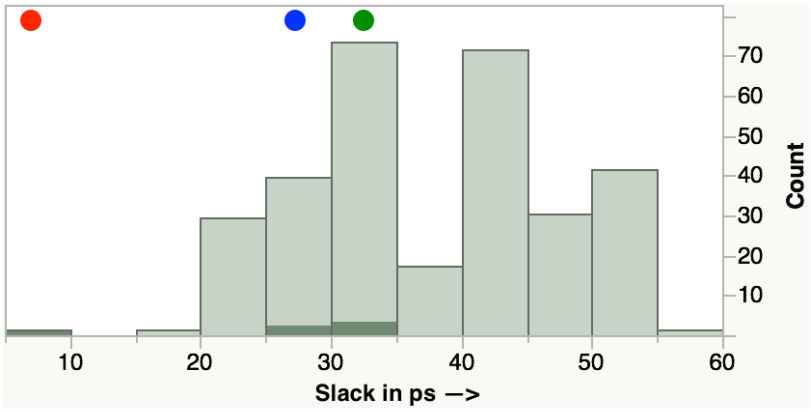
Running SPICE-level Monte-Carlo simulations to estimate the yield of a large design is time-consuming. It would be beneficial if a designer can predict the yield-loss at an earlier stage in the design flow and thereby decrease the overall design-cycle time. An obvious stage in the design flow to predict yield-loss is after the physical-design stage and by looking at the slack estimates from the STA tool.

Figures 7.3.6a and 7.3.6b show the slack distributions obtained from the SPICE simulation and a static-timing analysis tool for the synchronous divider design in the TTLH corner. The colored dots denote the slack of the bits in the TTLH corner that violated the setup-time constraint in other corners.

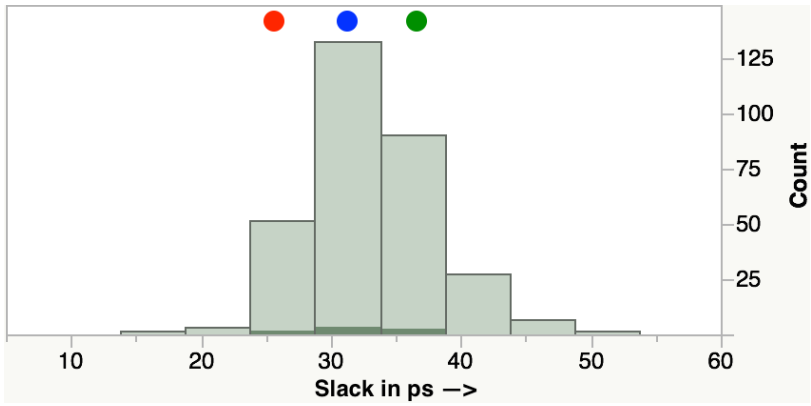
In Figures 7.3.6a and 7.3.6b, we can observe that the bit indicated by the red-dot appears in the tail of the slack distribution measured from the SPICE simulation, but appears in the middle of the slack distribution estimated from an STA tool which indicates a weak correlation between the measured and estimated slack for the failing bits.. For a bit denoted by the red-dot, the measured slack from the SPICE simulation is approximately 9ps but the estimated slack from an STA tool is approximately 22ps. This shows that an STA tool overestimates the slack for a failing data-bit (red-dot). Fails or yield-loss for the bits that an STA tool overestimates the slack are harder to predict by just looking at the slack estimates from an STA tool.

Figures 7.3.7a and 7.3.7b show the slack distributions obtained from the SPICE simulation and a static-timing analysis tool for the self-timed divider design in the TTLH corner. The colored dots denote the slack of the bits in the TTLH corner that violated the setup-time constraint in other corners.

For the self-timed divider we can observe that the failing bits that appear in the tail of the slack distribution measured from the SPICE simulation also appear in the tail of the slack distribution estimated from an STA tool which indicates a strong correlation



(a) Synchronous divider: Slack distribution from SPICE simulation.

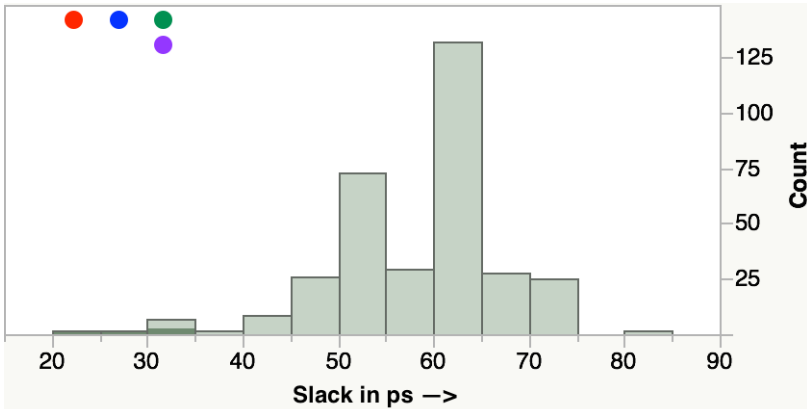


(b) Synchronous divider: Slack distribution from STA tool.

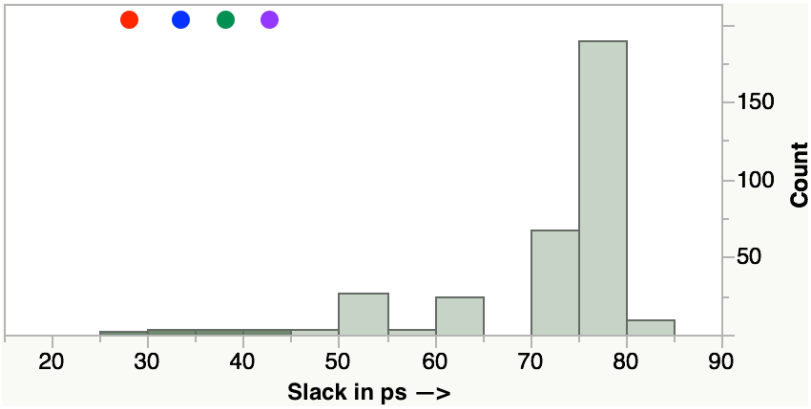
Fig. 7.3.6: Slack distribution of the synchronous divider: (a) from a SPICE simulation and (b) from an STA tool.

between the measured and estimated slack for the failing bits. The strong correlation of the failing bits suggests that we can potentially predict the failing bits and therefore yield-loss by looking at the slack estimates from an static-timing analysis tool.

One possible reason for the strong correlation of the failing bits in the self-timed design is that the self-timed design uses the data-path topology that has fewer logic gates than the data-path topology that the synchronous divider uses. Fewer logic gates typically implies fewer paths for the STA tool to get confused with. Hence more accurate slack estimates for a data path with fewer gates.



(a) Self-timed divider: Slack distribution from SPICE simulation.



(b) Self-timed divider: Slack distribution from STA tool.

Fig. 7.3.7: Slack distribution of a self-timed divider: (a) from a SPICE simulation and (b) from an STA tool.

8

Conclusion and Future Opportunities

In this research, I presented a methodology to evaluate data-path topologies that implement a conditional statement for an average-case performance that is better than the worst-case performance. I used a division algorithm as an example of a conditional statement. Contrary to conventional wisdom, the proposed methodology shows that a less-speculative data path yields a better average-case performance compared to a fully-speculative data path. This research explored the various stages of a design cycle, from algorithms to manufacturing.

The four new radix-2 division algorithms developed during this research offer a simpler quotient selection logic compared to the radix-2 division algorithms in [34, 25, 13] and [10]. Evaluating the algorithms for the frequency of hard and easy computations allows a designer to make a decision about pursuing a self-timed design or synchronous design early in the design cycle.

The *glissando* optimization technique developed in this research exploits a simple idea that the non-critical bits can arrive at the input of the registers later than the critical bits to reduce the delay of the data paths. The glissando technique enables the delay of the data paths to be independent of the word size which is very useful when designing a divider circuit for 1024-bit or 2048-bit RSA crypto-systems.

The results from the SPICE simulation of the extracted netlists show that compared to the synchronous divider, the self-timed divider is 10% faster on average, consumes 42% less energy per division on average and 20% less area. The improvement in all three figures of merit for self-timed divider is a consequence of choosing a less-speculative data path and designing a control path to take advantage of the faster shift-only operation.

Analyzing the response of the divider designs to variations in process and environment shows that the synchronous design offers a parametric yield of 77% and the self-timed divider design offers a parametric yield of 60%. In the synchronous design, the the setup-time violations caused yield-loss. By the increasing the clock period or the supply voltage we can potentially decrease the yield-loss. In the self-timed design, the the setup-time and duty-cycle violations caused yield-loss. The duty-cycle constraint was previously unknown and shows that a designer must consider the internal structure of a flip-flop or latch when designing a control path for the self-timed designs. To reduce the yield-loss in the self-timed design requires redesigning the control path which increases the design cycle time. Alternatively, we can gradually reduce the supply voltage to decrease the yield-loss because of the 100% yield at the low-voltage and high-temperature corner.

A simple regression analysis of the slack estimates from a static-timing analysis tool and the simulated slacks from SPICE simulation show that we can predict an yield-loss at an earlier stage in the design cycle for a less-speculative data path. For a fully-speculative data path yield-loss prediction is unlikely.

8.1 Future Opportunities

The following are the opportunities to further extend this research.

Modern microprocessors implement a radix-4 or radix-16 divider. A radix-16 di-

viders use two radix-4 stages [22] and therefore extending the algorithms A1b and B1b to radix-4 will be useful. Alternatively, an analysis of radix-2 variable-iteration division algorithms will also be useful. A radix-2 variable-iteration division algorithm retires at least one quotient digit per iteration and occasionally can retire two or three quotient digits per iteration. Because the variable-iteration algorithm can retire multiple quotient-digits per iteration, the number of iterations per division varies depending on the values of the input operand. Self-timed designs can take advantage of the occasional fewer-iterations per division in addition to faster shift-only operations.

From the point-of-view of manufacturing, the self-timed designs need a knob to control the speed of the control path after manufacturing to reduce yield loss because of setup-time and duty-cycle violations. Depending on the level of granularity required, a knob could be as simple as providing a different supplies to control and data paths or as complex as inserting programmable delay modules between the stages of a pipeline.

This research analyzed the response of the two designs considering only the global variation. Analyzing the response of synchronous and self-timed divider designs for local variation may reveal additional constraints that can affect the yield. The control path in this research use a string of inverters to increase the period of the synchronization pulse for an addition operation. Replacing the string of inverters with the same kind of gates that appear in the add path of the data path may produced better parametric yield. The hypothesis is that the gates with the same transistor topology behave the same way in presence of manufacturing variations.

Lack of EDA tools for self-timed design continue to hinder the design of complex circuits. Using two different flows to design complex self-timed circuits is cumbersome and error-prone. A tool that integrates the design of both data path and self-timed control path will be highly invaluable.

References

[1] http://cpudb.stanford.edu/. [cited at p. 126]

[2] E. Antelo, T. Lang, P. Montuschi, and A. Nannarelli. Digit-Recurrence Dividers with Reduced Logical Depth. *Computers, IEEE Transactions on*, 54(7):837 –851, july 2005. [cited at p. 11, 12]

[3] Peter A Beerel, Recep O Ozdag, and Marcos Ferretti. *A Designer's Guide to Asynchronous VLSI*. Cambridge University Press, 2010. [cited at p. 12]

[4] Peter A Beerel and Aiguo Xie. Performance Analysis of Asynchronous Circuits using Markov Chains. In *Concurrency and Hardware Design*, pages 313–343. Springer, 2002. [cited at p. 14]

[5] N. Burgess. A Fast Division Algorithm for VLSI. In *Computer Design: VLSI in Computers and Processors, 1991. ICCD '91. Proceedings, 1991 IEEE International Conference on*, pages 560 –563, oct 1991. [cited at p. 10]

[6] N. Burgess. Retiming the ARM VFP-11 Divide and Square Root Macrocell. In *Signals, Systems and Computers, 2007. ACSSC 2007. Conference Record of the Forty-First Asilomar Conference on*, pages 363–366, 2007. [cited at p. 11, 69]

[7] James Coke, Harikrishna Baliga, Niranjan Cooray, Edward Gamsaragan, Peter Smith, Ki Yoon, James Abel, and Antonio Valles. Improvements in the Intel Core

2 Penryn Processor Family Architecture and Microarchitecture. *Intel Technology Journal*, 12(3):179–193, 2008. [cited at p. 10]

[8] J. Cortadella and T. Lang. High-Radix Division and Square-Root with Speculation. *Computers, IEEE Transactions on*, 43(8):919–931, 1994. [cited at p. 10]

[9] J. Ebergen and R. Berks. Response-Time Properties of Linear Asynchronous Pipelines. *Proceedings of the IEEE*, 87(2):308–318, 1999. [cited at p. 14]

[10] J. Ebergen, I. Sutherland, and A. Chakraborty. New Division Algorithms by Digit Recurrence. In *Signals, Systems and Computers, 2004. Conference Record of the Thirty-Eighth Asilomar Conference on*, volume 2, pages 1849 – 1855 Vol.2, nov. 2004. [cited at p. xiv, 4, 5, 6, 11, 16, 19, 37, 40, 143]

[11] J.C. Ebergen and A. Megacz. Asynchronous loadable down counter, September 27 2011. US Patent 8,027,425. [cited at p. 106]

[12] M.D. Ercegovac and T. Lang. On-the-fly Rounding for Division and Square Root. In *Computer Arithmetic, 1989., Proceedings of 9th Symposium on*, pages 169–173, 1989. [cited at p. 41]

[13] M.D. Ercegovac and T. Lang. *Digital Arithmetic*. Morgan Kaufmann, 2003. [cited at p. 17, 22, 41, 143, 152, 155, 156]

[14] G. Gill, V. Gupta, and M. Singh. Performance Estimation and Slack Matching for Pipelined Asynchronous Architectures with Choice. In *Computer-Aided Design, 2008. ICCAD 2008. IEEE/ACM International Conference on*, pages 449–456, 2008. [cited at p. 14]

[15] D.L. Harris, S.F. Oberman, and M.A. Horowitz. SRT Division Architectures and Implementations. In *Computer Arithmetic, 1997. Proceedings., 13th IEEE Symposium on*, pages 18 –25, jul 1997. [cited at p. 9, 10, 11, 69, 126, 127, 128]

[16] N. Jamadagni and J. Ebergen. An Asynchronous Divider Implementation. In *Asynchronous Circuits and Systems (ASYNC), 2012 18th IEEE International Symposium on*, pages 97 –104, may 2012. [cited at p. 211]

[17] P. Kornerup. Digit Selection for SRT Division and Square Root. *Computers, IEEE Transactions on*, 54(3):294 – 303, march 2005. [cited at p. 22]

[18] Wei Liu and A. Nannarelli. Power Dissipation in Division. In *Signals, Systems and Computers, 2008 42nd Asilomar Conference on*, pages 1790 –1794, oct. 2008. [cited at p. 10, 69, 126, 127, 128]

[19] Wei Liu and A. Nannarelli. Power Efficient Division and Square Root Unit. *Computers, IEEE Transactions on*, 61(8):1059–1070, August 2012. [cited at p. 10, 11, 69, 126, 127, 128]

[20] G. Matsubara and N. Ide. A Low Power Zero-Overhead Self-Timed Division and Square Root Unit combining a Single-Rail Static Circuit with a Dual-Rail Dynamic Circuit. In *Advanced Research in Asynchronous Circuits and Systems, 1997. Proceedings., Third International Symposium on*, pages 198 –209, apr 1997. [cited at p. 14, 15, 125, 126, 127]

[21] P. Montuschi and L. Ciminiera. Over-Redundant Digit Sets and the Design of Digit-by-Digit Division Units. *Computers, IEEE Transactions on*, 43(3):269 –277, mar 1994. [cited at p. 11]

[22] A. Nannarelli. Radix-16 Combined Division and Square Root Unit. In *Computer Arithmetic (ARITH), 2011 20th IEEE Symposium on*, pages 169–176, 2011. [cited at p. 145]

[23] S.M. Nowick, K.Y. Yun, P.A. Beerel, and A.E. Dooply. Speculative Completion for the Design of High-Performance Asynchronous Dynamic Adders. In *Advanced*

Research in Asynchronous Circuits and Systems, 1997. Proceedings., Third International Symposium on, pages 210–223, 1997. [cited at p. 12, 15]

[24] S.F. Oberman and M.J. Flynn. Design Issues in Division and Other Floating-Point Operations. *Computers, IEEE Transactions on*, 46(2):154 –161, feb 1997. [cited at p. 5]

[25] B. Parhami. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford university press, 2000. [cited at p. 17, 22, 41, 143]

[26] Tung N. Pham and Earl E. Jr. Swartzlander. Design of Radix-4 SRT Dividers in 65 Nanometer CMOS Technology. In *Application-specific Systems, Architectures and Processors, 2006. ASAP '06. International Conference on*, pages 105 –108, sept. 2006. [cited at p. 126, 127]

[27] J.A. Prabhu and G.B. Zyner. 167 MHz Radix-8 Divide and Square Root using Overlapped Radix-2 Stages. In *Computer Arithmetic, 1995., Proceedings of the 12th Symposium on*, pages 155 –162, jul 1995. [cited at p. 10, 125, 126, 127]

[28] Md Mijanur Rahman, Tushar Kanti Saha, and Md Al-Amin Bhuiyan. Implementation of RSA Algorithm for Speech Data Encryption and Decryption. *International Journal of Computer Science and Network Security*, 12(3):74–82, 2012. [cited at p. 72]

[29] M. Renaudin, B.E. Hassan, and A. Guyot. A New Asynchronous Pipeline Scheme: Application to the Design of a Self-timed Ring Divider. *Solid-State Circuits, IEEE Journal of*, 31(7):1001 –1013, jul 1996. [cited at p. 15, 125, 126, 127]

[30] James E. Robertson. A New Class of Digital Division Methods. *Electronic Computers, IRE Transactions on*, EC-7(3):218 –222, sept. 1958. [cited at p. 9]

[31] I. Rust and T.G. Noll. A Radix-4 Single-Precision Floating Point Divider based on Digit Set Interleaving. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pages 709 –712, 30 2010-june 2 2010. [cited at p. 126, 127]

[32] M. Shah, R. Golla, G. Grohoski, P. Jordan, J. Barreh, J. Brooks, M. Greenberg, G. Levinsky, M. Luttrell, C. Olson, Z. Samoail, M. Smittle, and T. Ziaja. Sparc T4: A Dynamically Threaded Server-on-a-Chip. *Micro, IEEE*, 32(2):8–19, 2012. [cited at p. 6]

[33] Hosahalli Rajarao Srinivas. *High Speed Computer Arithmetic Architectures*. PhD thesis, University of Minnesota, Minneapolis, MN, USA, 1994. UMI Order No. GAX95-08963. [cited at p. 11, 40]

[34] H.R. Srinivas, K.K. Parhi, and L.A. Montalvo. Radix 2 Division with Over-Redundant Quotient Selection. *Computers, IEEE Transactions on*, 46(1):85 –92, jan 1997. [cited at p. 4, 5, 6, 11, 16, 143, 161, 162]

[35] I. Sutherland and S. Fairbanks. GasP: A Minimal FIFO Control. In *Asynchronus Circuits and Systems, 2001. ASYNC 2001. Seventh International Symposium on*, pages 46–53, 2001. [cited at p. 101]

[36] I.E. Sutherland and J.K. Lexau. Designing Fast Asynchronous Circuits. In *Asynchronus Circuits and Systems, 2001. ASYNC 2001. Seventh International Symposium on*, pages 184 –193, 2001. [cited at p. 102]

[37] I.E. Sutherland, R.F. Sproull, and D.F. Harris. *Logical Effort: Designing Fast CMOS Circuits*. Morgan Kaufmann, 1999. [cited at p. ix, 45, 47, 48, 68]

[38] K. D. Tocher. Techniques of Multiplication and Division for Automatic Binary Computers. *The Quarterly Journal of Mechanics and Applied Mathematics*, 11(3):364–384, 1958. [cited at p. 9]

[39] Neil Weste and David Harris. *CMOS VLSI Design: A Circuits and Systems Perspective*. Addison-Wesley Publishing Company, USA, 4th edition, 2010. [cited at p. xix, 45, 139]

[40] T.E. Williams and M.A. Horowitz. A Zero-Overhead Self-Timed 160-ns 54-b CMOS Divider. *Solid-State Circuits, IEEE Journal of*, 26(11):1651 –1661, nov 1991. [cited at p. 14, 15, 125, 126, 127]

Appendix A

Algorithms B1, B1b and B1c

This appendix presents the derivation of the algorithms B1, B1b and B1c.

Instead of using a two's complement representation for the partial remainder, we can use a binary signed digit (BSD) representation. In a BSD representation, a vector of signed bits from the set $\{-1, 0, 1\}$ represents the partial remainder r. Instead of using a single vector of signed bits, we use two vectors of unsigned bits r_s and r_c, such that $r = r_c - r_s$ where, r_s is the partial sum and r_c is the partial carry. In BSD representation partial sum, r_s, carries a negative weight. Please see [13] for more details on BSD representation and carry-free addition.

As mentioned in Equation (3.3.8), the range invariant for the partial remainder, $r = r_c - r_s$, is $(-4, 4)$ when we use BSD representation for the partial remainder. Note that for the BSD representation, both bounds are excluded. The left bold-diamond, S0 to S15, in Figure A.1 consists of all numbers (r_s, r_c) satisfying range invariant (3.3.8).

To analyze what happens when we perform doublings and additions on points satisfying range invariant (3.3.8), consider a larger diamond in Figure A.1, where each binary vector is represented with one extra digit at the most-significant position. We are interested in a series of operations that takes a point in the bottom-left bold

diamond and returns a point in the bottom-left bold diamond. As with the two's complement representations, the operations must end with a doubling and an addition or subtraction may preceded the doubling. Each sequence of operations must maintain invariant (3.3.2).

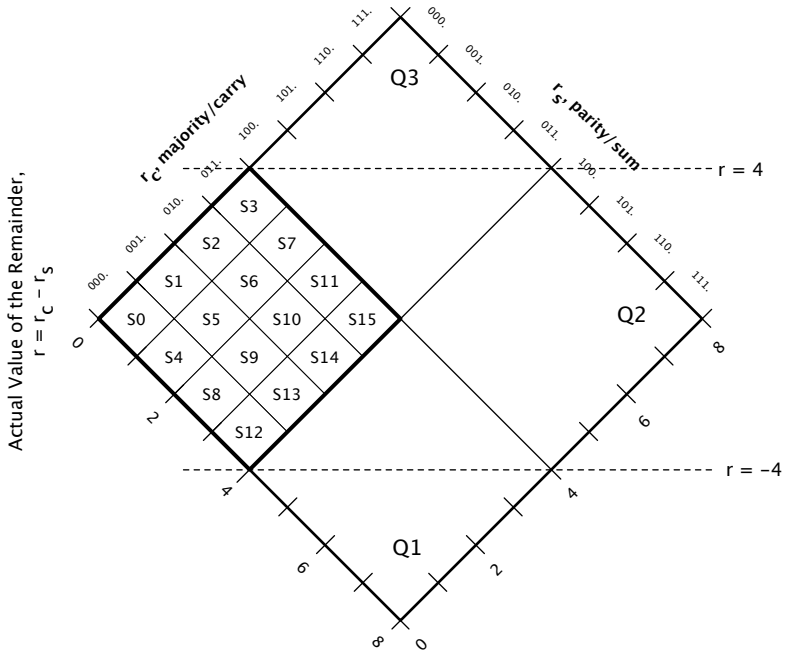


Fig. A.1: The area for the partial remainders (r_s, r_c), where the value of the partial remainder r is given by $r = r_c - r_s$. The left bold-diamond consisting of S0 to S15 diamonds satisfies the range invariant (3.3.8)

Doublings and Translations

Let us consider Figure A.2. Doubling a point in S6 and S9 yields a point in R3 and R1 respectively. As illustrated in Figure A.5, we can bring each point, (r_s, r_c), in R3 and R1 back to left bold-square by a translation over $(-2, -2)$.

Let us now consider Figure A.4. Doubling a point in S0∪S1∪S4∪S5 yields a point in the left bold-diamond, S0 to S15. Doubling a point in S10∪S11∪S14∪S15

yields a point in Q2 of Figure A.4. We can bring a point in Q2 back to left bold-diamond by a translation over $(-4, -4)$, as illustrated in Figure A.5. Any translation over $(-t, -t)$ leaves the value of $r = r_c - r_s$ invariant and thus each translation maintains invariant (3.3.2)).

Translation

A simple recoding implements a translation over $(-2, -2)$ for a point in R1 and R3 as follows:

$$\begin{array}{lcl} 01 & \rightarrow & 00 \\ 00 & \rightarrow & 11 \\ 11 & \rightarrow & 10 \\ 10 & \rightarrow & 01 \end{array}$$

Notice that the second most-significant bit changes. The most-significant bit is a copy of the second most-significant bit if the original value of the most-significant bit is 0 otherwise the most-significant bit is an inverse of the second most-significant bit.

An inversion of the most-significant bit (bit position with weight 2^2) implements a translation over $(-4, -4)$ for any point in Q2.

Note that if we omit the most-significant bit, then the translation over $(-2, -2)$ can be implemented by inverting the bit at position 2^1 and the translation over $(-4, -4)$ requires no operation. I use * to denote the extra inversion required for the translation over $(-2, -2)$. Therefore, the 2X* operation denotes doubling followed by translation over $(-2, -2)$.

Additions

Let us analyze what happens if we subtract D from a point in left bold-diamond in Figure A.6.

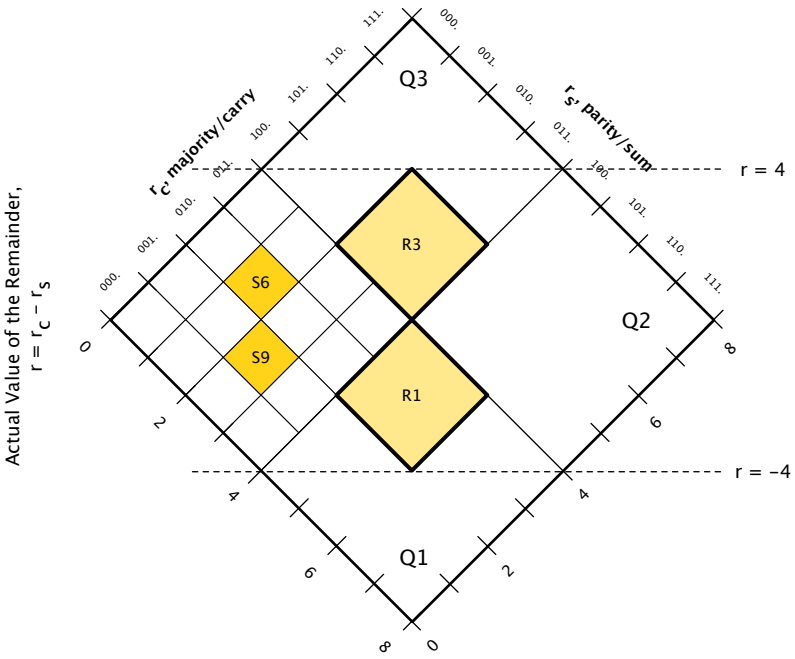


Fig. A.2: Effect of doubling a point in S6 and S9. Doubling a point (r_s, r_c) in S6 and S9 yields a point in R3 and R1 respectively.

When the partial remainder is in a binary signed-digit representation, we implement the addition $r + z$ of a remainder $r = r_c - r_s$ and a choice $z \in \{-2D, -D, D, 2D\}$ by means of a carry-free addition [13]. As shown in [13], we have

$$BSDcarry - BSDsum = r_c - r_s + z, \tag{A.0.1}$$

where $BSDcarry$ and $BSDsum$ are the result of a carry-save addition of $\overline{r_s}, r_c$, and z, with an inversion of the parity result:

$$BSDcarry = 2 * majority(\overline{r_s}, r_c, z), \tag{A.0.2}$$

$$BSDsum = \overline{parity(\overline{r_s}, r_c, z)}. \tag{A.0.3}$$

With m non-fractional bits, the majority and parity functions forming the carry-free addition can be done modulo 2^m. For the radix-2 division algorithms, $m = 3$.

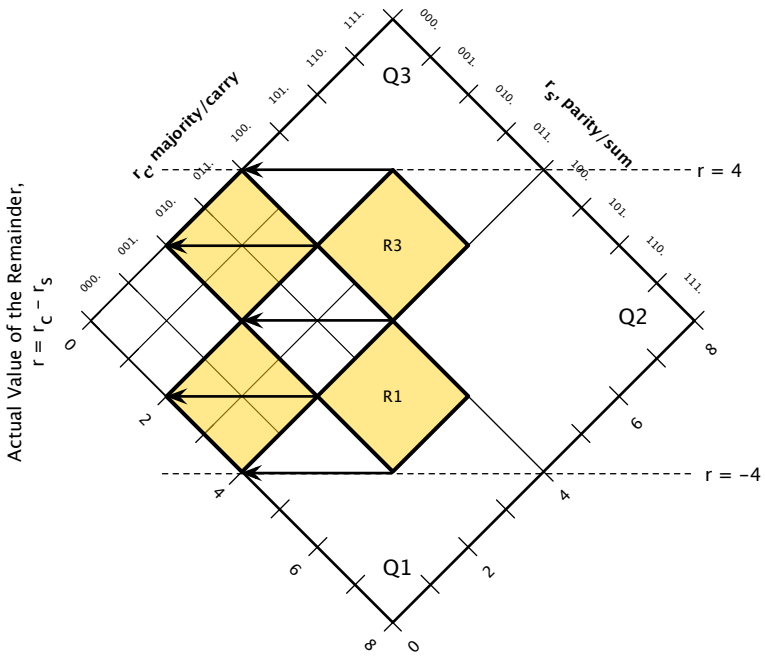


Fig. A.3: Translating a point in R1 and R3 back to the left bold-diamond. Translation is implemented by subtracting 2 from r_s and r_c. Note that translation keeps the value of the remainder, $r = r_c - r_s$, unchanged.

This implementation of carry-free addition applies only if we take a two's complement representation for z [13]. Thus $-D$ and $-2D$ can be represented by $-D = 110.y + ulp$ and $-2D = 10b.y + ulp$ respectively, for some bit vector y and bit b.

Following is the calculation for subtracting D from a point in S1

$$\begin{array}{rl} r_s & 000. \\ r_c & 001. \\ \text{-}D & 110.\text{y} \quad + \; ulp \\ & \text{----------} \\ BSDsum & 111. \\ BSDcarry & 11?. \end{array}$$

These values for *BSDsum* and *BSDcarry* correspond to a point (r_s, r_c) in squares T14 or T15.

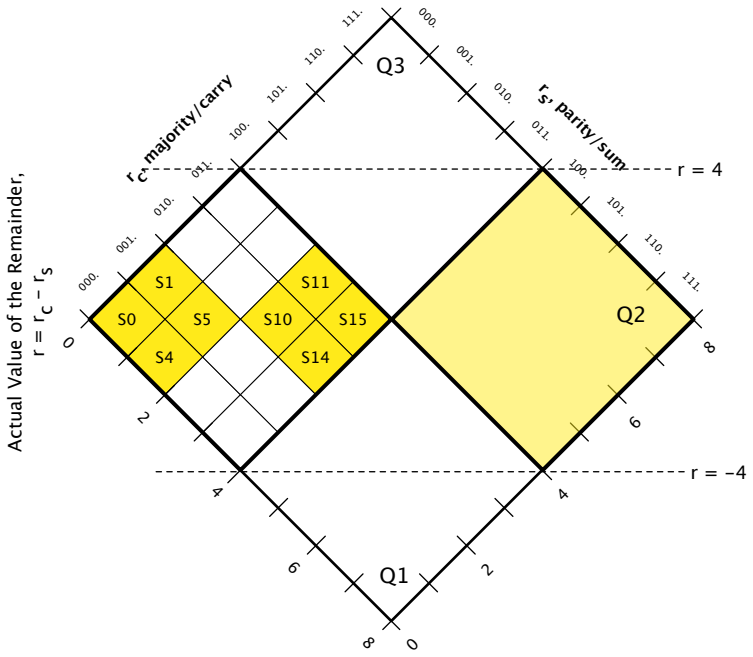


Fig. A.4: Effect of doubling a point in S10∪S11∪S14∪S15.

Following is the calculation for adding D to a point in square S4

r_s	001.
r_c	000.
D	001.x

$BSDsum$	000.
$BSDcarry$	00?.

These values for $BSDsum$ and $BSDcarry$ correspond to a point in S0 or S1.

Table A.1 shows the results of carry-free addition in small diamonds. For subtraction, consider the diamonds where the value of the partial remainder r is greater than 0. For addition, consider the diamonds where the value of the partial remainder r is less than 0.

All subtractions of D from diamonds S1, S2, S4, S5, and S8 yield points in small diamonds T3, T6, T7, T8, T10, or T11 of Figure A.6 which can be translated, doubled,

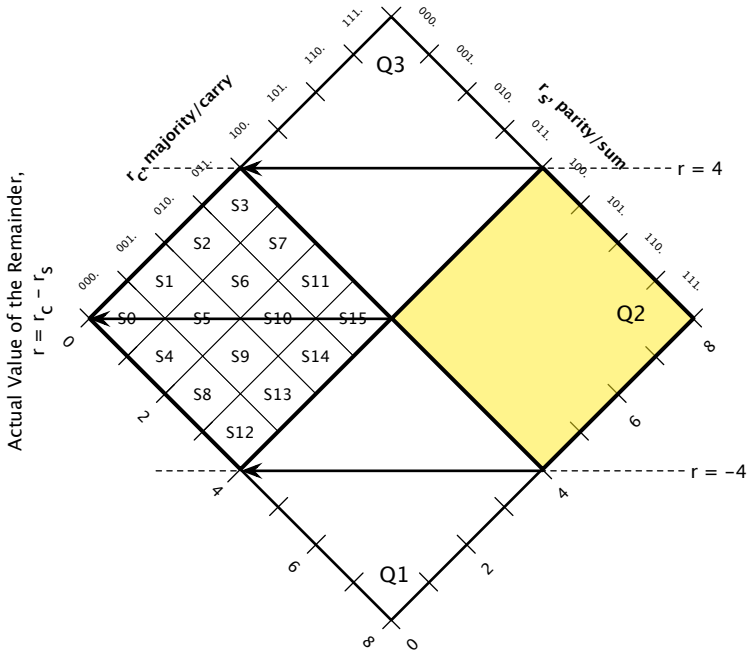


Fig. A.5: Translating a point in Q2 back to the left bold-diamond. Translation is implemented by subtracting 4 from r_s and r_c. Note that translation keeps the value of the remainder, $r = r_c - r_s$, unchanged.

Table A.1: The effect of adding or subtracting D

Origin	Destination after subtracting D	Origin	Destination after adding D
S1	T14∪T15	S4	S0∪S1
S2	T0∪T1	S8	S14∪S15
S3	T6∪T7	S9	S10∪S11
S6	T4∪T5	S12	S8∪S9
S7	T0∪T1	S13	S14∪S15
S11	T14∪T15	S14	S0∪S1

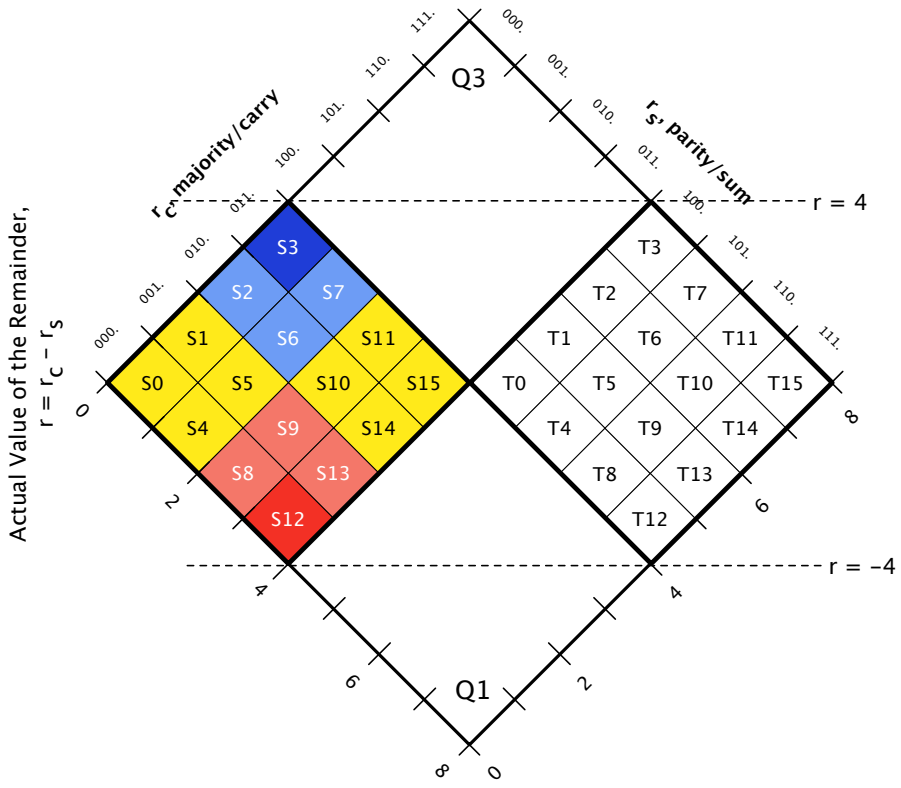


Fig. A.6: The effect of carry-free additions and subtractions with D or $2D$. The division algorithms can perform carry-free additions or subtraction in the shaded squares.

and translated again to return a point in one of the S0 to S15 diamonds.

Subtracting D from any point in S0 yields a point in T0 or T4. Translating a point in the T0 or T4 yields a point in S1 or S5 where the point must undergo another subtraction. Therefore, instead of subtracting D, we subtract $2D$ from a point in S0 using carry-free addition

r_s	000.
r_c	011.
-2D	10?.y + ulp

$BSDsum$	11?.
$BSDcarry$	11?.

The resulting values for $BSDsum$ and $BSDcarry$ correspond a point in T1, T3, T5, or T6. Points in these diamonds can be translated, doubled and translated again to return a point in one of the smaller diamonds in the left bold-diamond.

All additions of D to points in S7, S10, S11, S13, and S14 yield points in S2, S3, S6, S7, S8, or S12. Points in these diamonds can all be doubled and translated to return a point in one of the S0 to S15 diamonds.

Addition of D to a point in S15 yields a point in S10 or S14 where the point must undergo another addition rather than a doubling. Adding $2D$, however, to any point in S15 returns a point in S8, S9, S12, or S13, which can be doubled and translated to remain in one of the S0 to S15 diamonds.

With the analysis of the doublings, translations, and additions, we can put together a number of division algorithms based on the BSD representation for the partial remainder and carry-free additions.

Before giving the five possible choices for sequences of operations, we can make a few simplifications. Because each set of operations in our division algorithm takes a point in one of the S0 to S15 diamonds and returns a point in one of the S0 to S15 diamonds, we can omit the most-significant bit, which is always 0, and use only the two non-fractional bits. The omission of the most-significant bit simplifies the implementation of the translation over (-4, -4) to an operation that is implemented automatically.

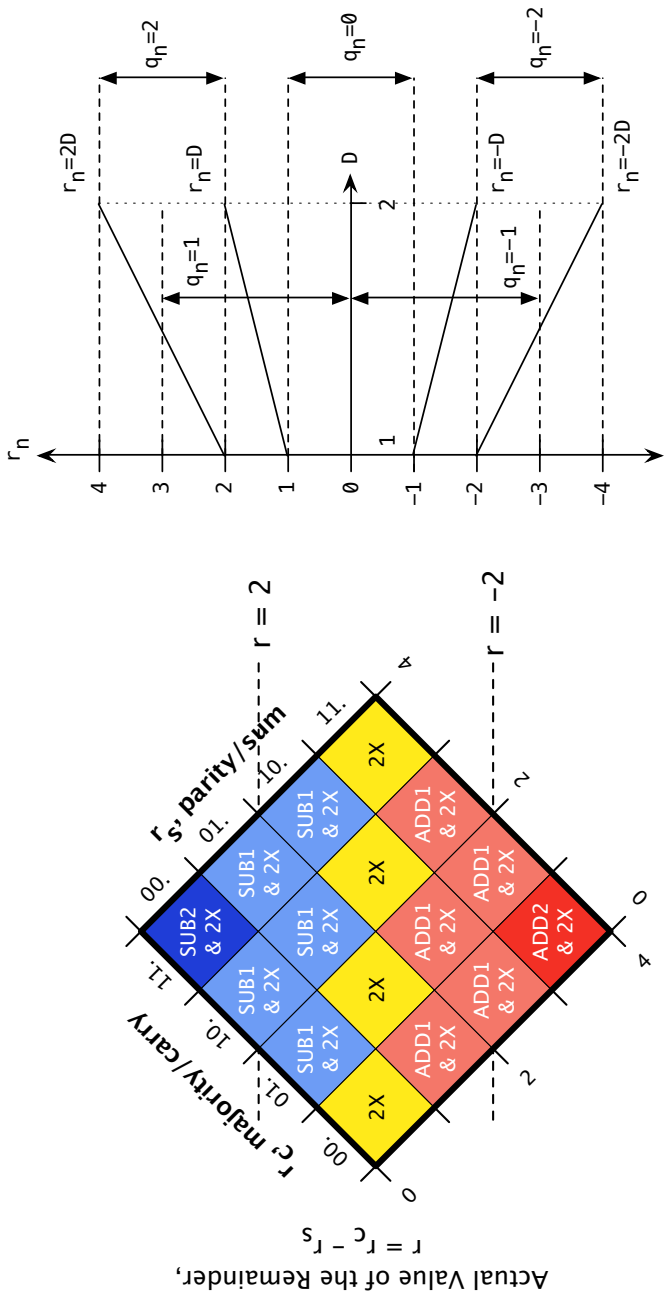
The five choices for the sequences of operations are: 2X, 2X*, SUB1&2X, SUB2&2X, ADD1&2X and ADD2&2X. The quotient digit selected and the statements executed for

each of these operation is listed in Table 3.3.1. Furthermore, each of these operations maintain invariant (3.3.2) and range invariant (3.3.8).

We can compose a division algorithm by choosing one sequence of operations for each small diamond S0 through S15. For S2, S7, S8, and S13 there are two choices for selecting a quotient digit. For S2 and S8 the two choices are as follows: selecting a quotient digit 0 and performing a 2X operation or selecting a quotient digit 1 and performing a SUB1&2X operation on the partial remainder. For S7 and S13 the two choices are as follows: selecting a quotient digit 0 and performing a 2X operation or selecting a quotient digit -1 and performing a ADD1&2X operation on the partial remainder. For all other squares there is only one choice for selecting a quotient digit. The three most symmetric algorithms appear in Figures A.7, A.8 and A.9.

Algorithms B1, B1b and B1c in Figures A.7 , A.8 and A.9 satisfy the range invariant (3.3.8). Algorithm B1, also satisfies the range invariant $r = r_c - r_s \in (-2D, 2D)$. The proof that Algorithm B1 satisfies the range invariant $r \in (-2D, 2D)$ is essentially the same as the proof for algorithm A1. Therefore, algorithms B1 requires at least $L + 3$ iteration and algorithms B1b and B1c require at least $L + 4$ iterations to terminate with an error $\epsilon \in (-ulp/2, ulp/2)$.

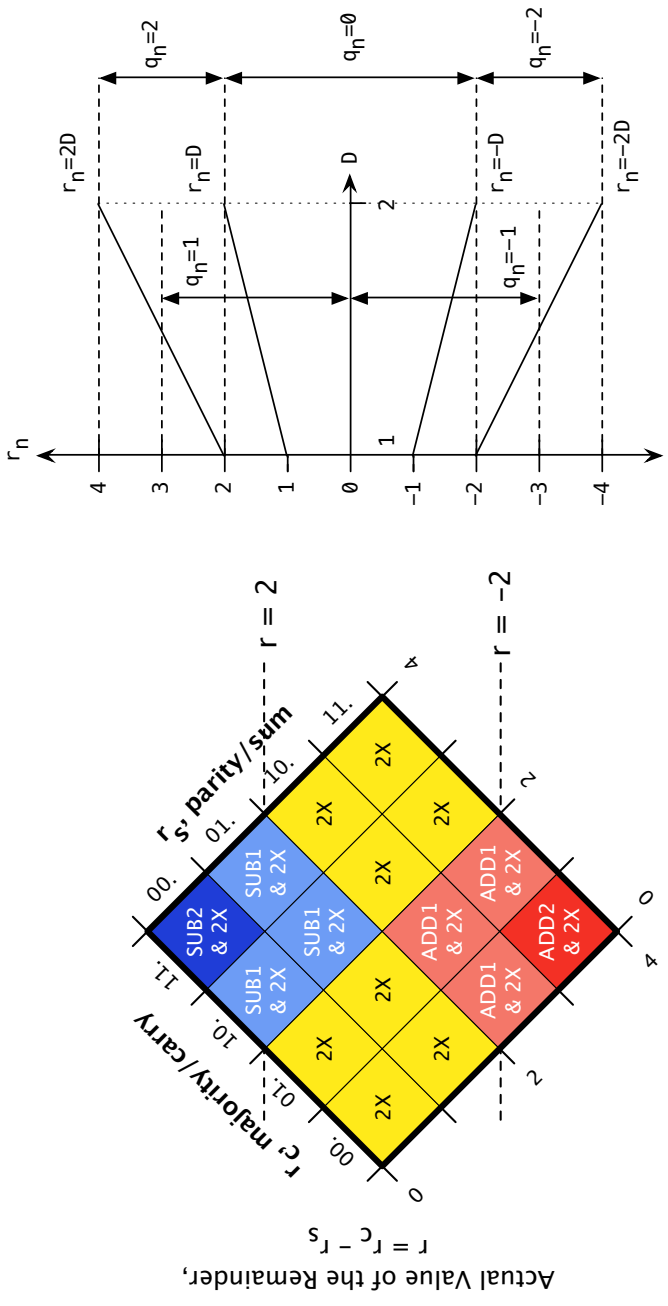
Algorithm B1 in Figure A.7 is the same algorithm as presented in [34]. Note that in [34], the authors use the recurrence relation in (3.2.2) and hence the authors use the range invariant $r \in (-D, D)$ for the partial remainder. Because we have considered the recurrence relation in (3.2.3) throughout this paper, the range invariant for the partial remainder in [34] translates to $r \in (-2D, 2D)$.



(a) Diamond Diagram for Algorithm B1.

(b) P-D Plot for Algorithm B1.

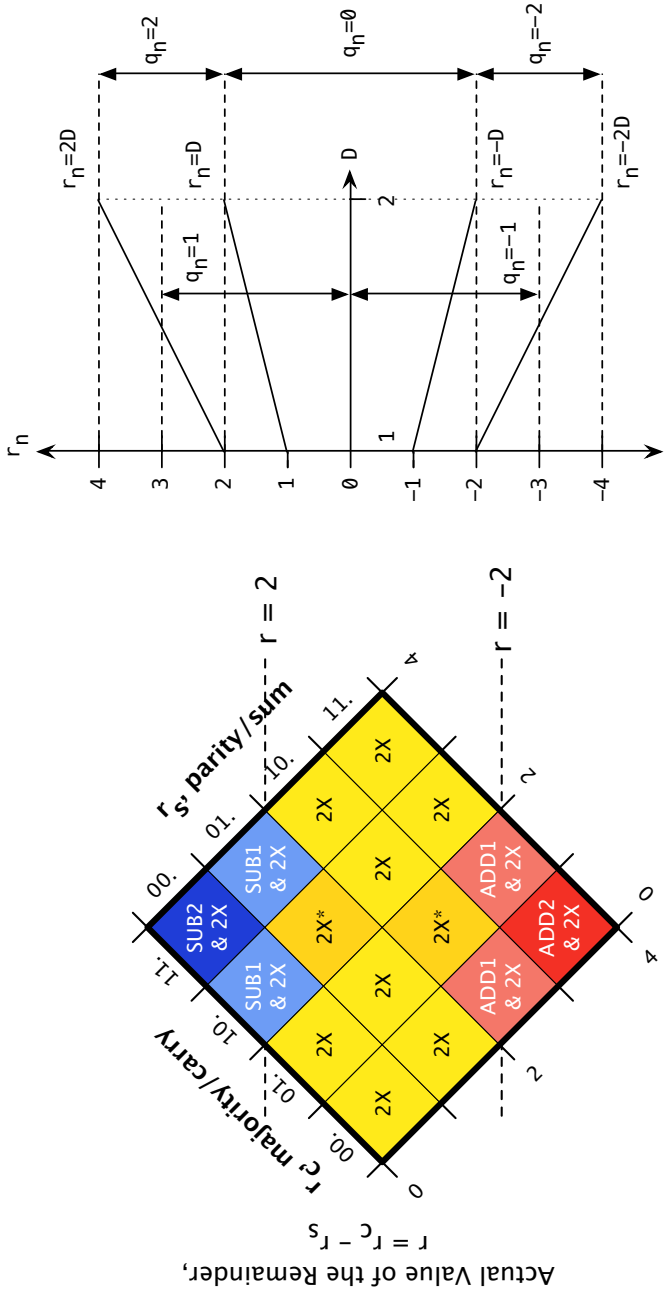
Fig. A.7: Algorithm B1: Figures (A.7a) and (A.7b) illustrate the quotient selection function of algorithms B1 in Diamond Diagram and P-D diagram. Algorithm B1 is the same algorithm as presented in [34].



(a) Diamond Diagram for Algorithm B1b.

(b) P-D Plot for Algorithm B1b.

Fig. A.8: Algorithm B1b: Figures (A.8a) and (A.8b) illustrate the quotient selection function of algorithms B1b in Diamond Diagram and P-D diagram.



(a) Diamond Diagram for Algorithm B1c.

(b) P-D Plot for Algorithm B1c.

Fig. A.9: Algorithm B1c: Figures (A.9a) and (A.9b) illustrate the quotient selection function of algorithms B1b in Diamond Diagram and P-D diagram.

Appendix B

On-the-Fly Conversion

The division algorithms in Chapter 3 select a quotient digit from the redundant set {-1, 0, 1} or {-2, -1, 0, 1, 2}. With slight modification, a division algorithm can be used to compute square-root of a number. Hence, the square-root and division operations often share the same hardware. Square-root algorithms operation execute operations such as $r = r - 2 * q + c$. Here, q is the quotient "thus far" and $c = 2^{-i-1}$ denotes the unit of least-significant position of q and i is the iteration index. Note that in square-root algorithms, quotient denotes the root. If both r and q are in a redundant representation then the carry-save additions and subtractions require a 4:2 (four-input) rather than 3:2 (three-input) carry-save adders. A 4:2 carry-save adder takes more time than 3:2 carry-save adder. A 3:2 carry-save adder suffices if r is in a redundant representation and q is in a unique representation. Because the digit-recurrence algorithms compute one-digit per iteration, we can calculate a unique representation of the quotient on-the-fly without requiring expensive carry-propagate additions.

B.1 On-the-Fly Conversion

Division and square-root algorithms update the quotient according to the expression in Equation (B.1.1), where q_j is the quotient digit from the set {-1, 0, 1} or {-2, -1, 0, 1, 2}.

$$q = q - q_i * c;\ c = \frac{c}{2} \tag{B.1.1}$$

Consider on-the-fly conversion from the set {-1, 0, 1} to {0, 1}. Let Q denote the unique binary representation of q. The least-significant bit position of Q is 2^{-i}. With $c = 2^{-i-1}$, $2 * c$ denotes the least-significant bit position of Q. Consequently, we can compute the unique representation of q by postfixing Q with 1 when the algorithm reties a quotient digit 1 and postfixing Q with -1 when the algorithm retires a quotient digit -1, and so on. To avoid a carry-propagate addition when postfixing Q with -1, consider the invariant in Equation (B.1.2)

$$\begin{aligned} &Q_0 \text{ is the unique representation of } q \\ &Q_{-1} \text{ is the unique representation of } q\text{-}2 * c \end{aligned} \tag{B.1.2}$$

If the invariant in Equation (B.1.2) holds initially, then the following statements maintain the invariant in Equation (B.1.2).

For $q_i = -1$:

$$q = q\text{-}c;\ c = \frac{c}{2};\ Q_0 = Q_{-1}1;\ Q_{-1} = Q_{-1}0$$

For $q_i = 0$:

$$q = q;\ c = \frac{c}{2};\ Q_0 = Q_00;\ Q_{-1} = Q_{-1}1$$

For $q_i = 1$:

$$q = q\ +\ c;\ c = \frac{c}{2};\ Q_0 = Q_01;\ Q_{-1} = Q_00$$

Here are two examples to show that the above statements maintain the invariants in Equation (B.1.2): First consider the case for $q_i = -1$.

$$\begin{aligned} q &= q - c;\ \ c = \frac{c}{2} \\ Q_0 &= q = q - c = q - 2c + c = Q_{-1} + c = Q_{-1}1 \\ Q_{-1} &= q - 2c = (q - c) - c = q - 2c = Q_{-1}0 \end{aligned} \tag{B.1.3}$$

Now consider the case for $q_i = 1$.

$$\begin{aligned} q &= q + c;\ \ c = \frac{c}{2} \\ Q_0 &= q = q + c = Q_0 + c = Q_0 1 \\ Q_{-1} &= q - 2c = (q + c) - c = q = Q_0 0 \end{aligned} \tag{B.1.4}$$

For on-the-fly conversion from the set {-2, -1, 0, 1, 2} to {0, 1}, consider the invariant in Equation (B.1.5)

$$\begin{aligned} Q_1 &\text{ is the unique representation of } q + 2 * c \\ Q_0 &\text{ is the unique representation of } q \\ Q_{-1} &\text{ is the unique representation of } q - 2 * c \\ Q_{-2} &\text{ is the unique representation of } q - 4 * c \end{aligned} \tag{B.1.5}$$

The following statements maintain the invariant in Equation (B.1.5).

For $q_i = -2$:

$$q = q - 2 * c;\ \ c = \frac{c}{2};\ \ Q_{+1} = Q_{-1}1;\ \ Q_0 = Q_{-1}0;\ \ Q_{-1} = Q_{-2}1;\ \ Q_{-2} = Q_{-2}0;$$

For $q_i = -1$:

$$q = q - c;\ \ c = \frac{c}{2};\ \ Q_{+1} = Q_0 0;\ \ Q_0 = Q_{-1}1;\ \ Q_{-1} = Q_{-1}0;\ \ Q_{-2} = Q_{-2}1;$$

For $q_i = 0$:

$$q = q;\ \ c = \frac{c}{2};\ \ Q_{+1} = Q_0 1;\ \ Q_0 = Q_0 0;\ \ Q_{-1} = Q_{-1}1;\ \ Q_{-1} = Q_{-1}0;$$

For $q_i = 1$:

$$q = q \ + \ c; \ c = \frac{c}{2}; \ Q_{+1} = Q_{+1}0; \ Q_0 = Q_0 1; \ Q_{-1} = Q_0 0; \ Q_{-2} = Q_{-1}1;$$

For $q_i = 2$:

$$q = q \ + \ 2 * c; \ c = \frac{c}{2}; \ Q_{+1} = Q_{+1}1; \ Q_0 = Q_{+1}0; \ Q_{-1} = Q_0 1; \ Q_{-2} = Q_0 0;$$

Appendix C

Delay Estimates for the Data Path Topologies

This appendix gives the delay estimates for the remaining thirteen data path topologies using the method of logical effort.

C.1 Topology 1

C.1.1 Data Path T1D2

Figure C.1 shows the T1D2 data path. For the data path T1D2 consider the following paths to estimate the delay:

- Select Path: `remainder-reg` $\rightarrow$ `QSLC` $\rightarrow$ `2:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `remainder-reg`
- Add Path: `remainder-reg` $\rightarrow$ `QSLC` $\rightarrow$ `4:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `remainder-reg`

- Shift Path: `remainder-reg` $\rightarrow$ `2X*` $\rightarrow$ `2:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `remainder-reg`

Table C.1 lists the logical effort and parasitic delay of the logic gates in the select, add and shift paths along with the number of stages in each gate. Table C.2 lists the branching effort at various nodes in the respective paths.

Using the values of G, P and N in Table C.1 and B in Table C.2, the delays of the data path when used in a synchronous and self-timed environment are

$$\begin{aligned} D_{sync} &= 10.8 \text{ FO4}, \\ D_{async} &= 10.0 \text{ FO4}. \end{aligned} \tag{C.1.1}$$

Table C.1: Data Path T1D2: Logical Effort and parasitic delay of the gates in the select, add and shift paths.

Gate	Select Path			Add Path			Shift Path		
	g	p	n	g	p	n	g	p	n
reg	2	2	1	2	2	1	2	2	1
4:1 Mux				2.7	6	2			
2:1 Mux	1.8	4	2	1.8	4	2	1.8	4	2
2:1 Mux	1.8	4	2	1.8	4	2	1.8	4	2
QSLC	11.3	4	2	10	4	2			
G	**72**			**170**			**7**		
P		**14**			**20**			**10**	
N			**7**			**9**			**5**

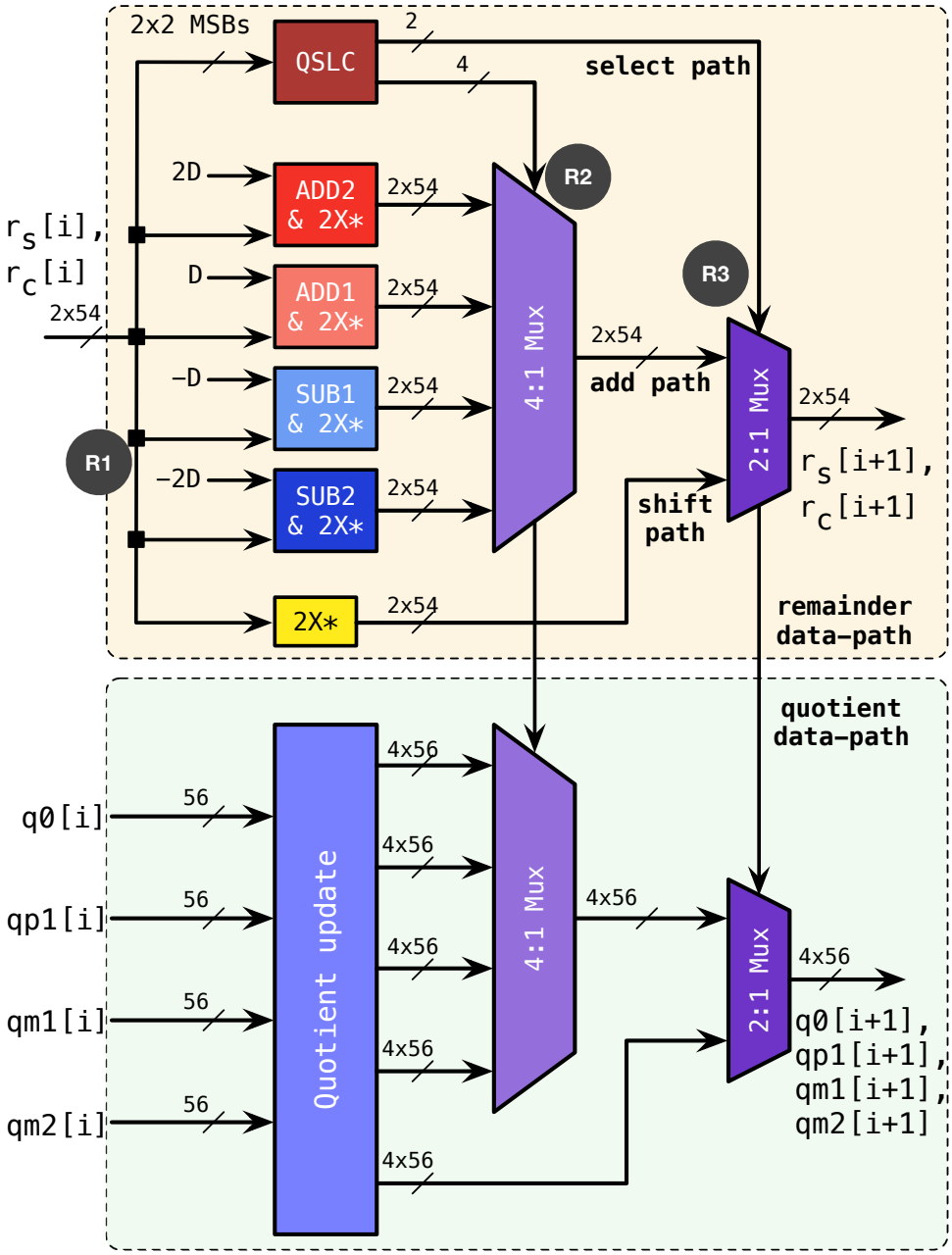


Fig. C.1: Data Path T1D2

Table C.2: Data Path T1D2: Branching effort.

Node	Select Path	Add Path	Shift Path
R0	2	2	2
R1		1.3	2.7
R2		332	
R3	332		
B	**664**	**863**	**6**

C.1.2 Data Path T1D3

Figure C.2 shows the T1D3 data path. For the data path T1D3 consider the following paths to estimate the delay:

- Select Path: `remainder-reg` $\rightarrow$ `QSLC` $\rightarrow$ `3:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `remainder-reg`
- Add Path: `remainder-reg` $\rightarrow$ `2:1 Mux` $\rightarrow$ `3:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `remainder-reg`
- Shift Path: `remainder-reg` $\rightarrow$ `2X*` $\rightarrow$ `3:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `remainder-reg`

Table C.3 lists the logical effort and parasitic delay of the logic gates in the select, add and shift paths along with the number of stages in each gate. Table C.4 lists the branching effort at various nodes in the respective paths.

Using the values of G, P and N in Table C.3 and B in Table C.4, the delays of the data path when used in a synchronous and self-timed environment are

$$\begin{aligned} D_{sync} &= 10.5 \text{ FO4}, \\ D_{async} &= 10.0 \text{ FO4}. \end{aligned} \tag{C.1.2}$$

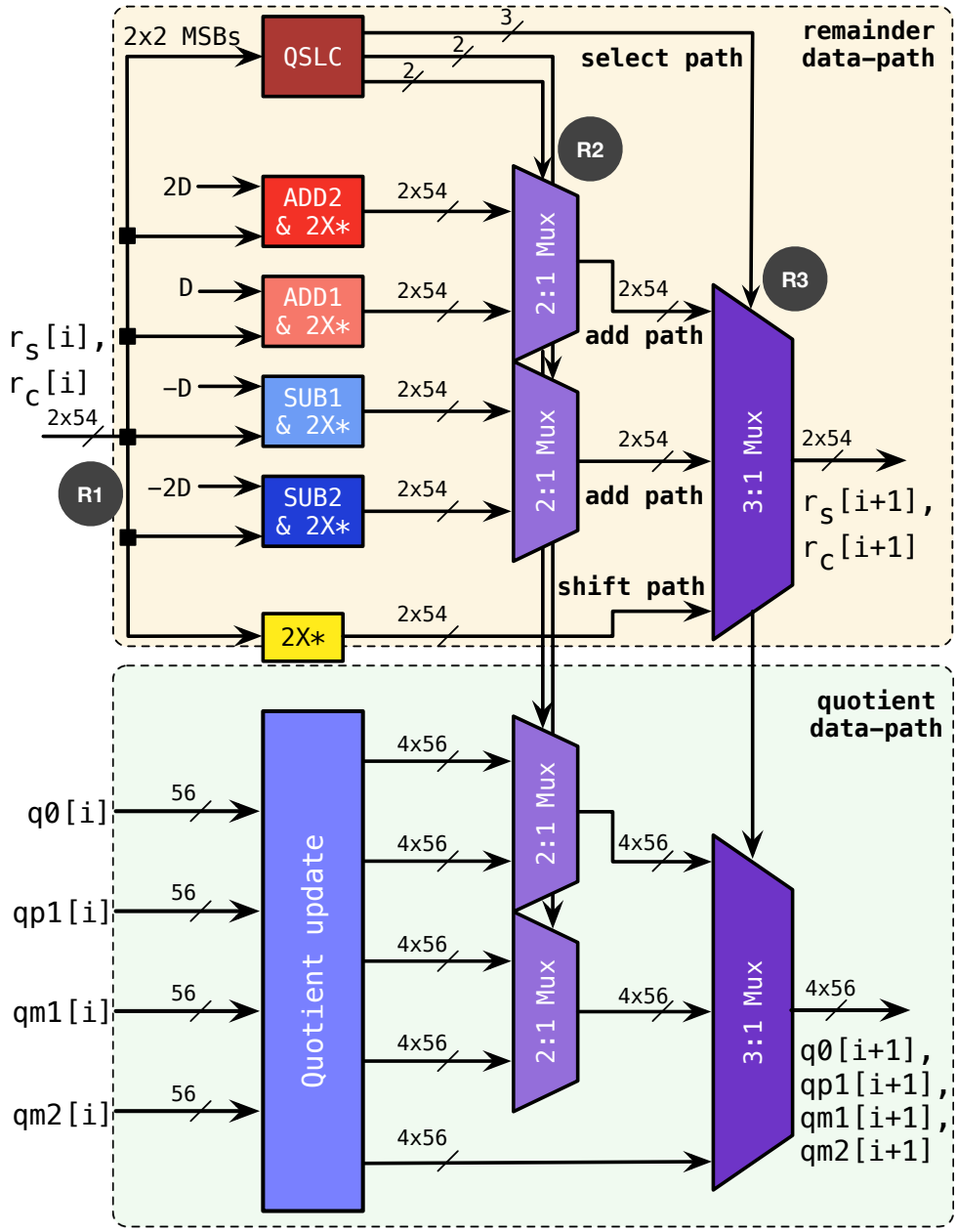


Fig. C.2: Data Path T1D3

Table C.3: Data Path T1D3: Logical Effort and parasitic delay of the gates in the select, add and shift paths.

Gate	Select Path			Add Path			Shift Path		
	g	p	n	g	p	n	g	p	n
reg	2	2	1	2	2	1	2	2	1
2:1 Mux				1.8	4	2			
3:1 Mux	2.7	5	2	2.7	5	2	2.7	5	2
2:1 Mux	1.8	4	2	1.8	4	2	1.8	4	2
QSLC	7.3	4	1	6	4	1			
G	**70**			**102**			**10**		
P		**15**			**19**			**11**	
N			**6**			**8**			**5**

Table C.4: Data Path T1D3: Branching effort.

Node	Select Path	Add Path	Shift Path
R0	2	2	2
R1		1.5	5.5
R2		332	
R3	332		
B	**664**	**996**	**11**

C.1.3 Data Path T1D4

Figure C.3 shows the T1D4 data path. For the data path T1D4 consider the following paths to estimate the delay:

- Select Path: `remainder-reg` → `QSLC` → `3:1 Mux` → `2:1 Mux` → `remainder-reg`
- Add1 Path: `remainder-reg` → `QSLC` → `2:1 Mux` → `ADD&2X*` or `SUB&2X*` → `3:1 Mux` → `2:1 Mux` → `remainder-reg`

- Add2 Path: `remainder-reg` $\rightarrow$ `amp` $\rightarrow$ `2:1 Mux` $\rightarrow$ `3:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `quotient-reg`
- Shift Path: `remainder-reg` $\rightarrow$ `2X*` $\rightarrow$ `QSLC` $\rightarrow$ `3:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `remainder-reg`

Tables C.5 lists the logical effort and parasitic delay of the logic gates in the select, add and shift paths along with the number of stages in each gate. Table C.9 lists the branching effort at various nodes in the respective paths.

Using the values of G, P and N in Table C.5 and B in Table C.6, the delays of the data path when used in a synchronous and self-timed environment are

$$\begin{aligned} D_{sync} &= 11.8 \text{ FO4}, \\ D_{async} &= 10.6 \text{ FO4}. \end{aligned} \tag{C.1.3}$$

Table C.5: Data Path T1D4: Logical Effort and parasitic delay of the gates in the select, add and shift paths.

Gate	Select Path			Add1 Path			Add2 Path			Shift Path		
	g	p	n	g	p	n	g	p	n	g	p	n
reg	2	2	1	2	2	1	2	2	1	2	2	1
amp							1	2	2			
CSA				10	6	1						
2:1 Mux				1.8	4	2	1.8	4	2			
3:1 Mux	2.7	5	2	2.7	5	2	2.7	5	2	2.7	5	2
2:1 Mux	1.8	4	2	1.8	4	2	1.8	4	2	1.8	4	2
QSLC	7.3	4	1	6	4	1	6	4	1			
G	**70**			**1016**			**102**			**10**		
P		**15**			**25**			**21**			**11**	
N			**6**			**9**			**10**			**5**

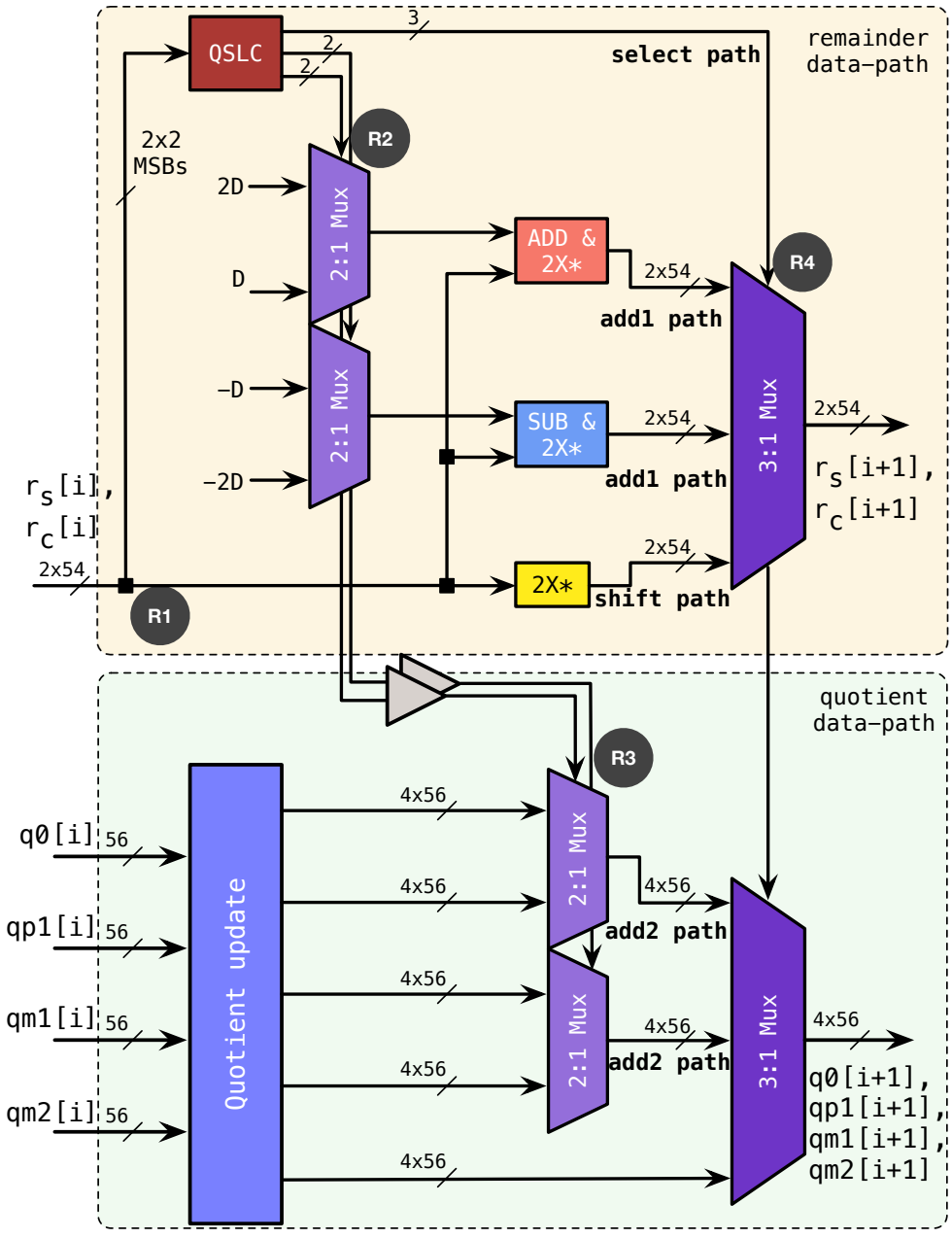


Fig. C.3: Data Path T1D4

Table C.6: Data Path T1D4: Branching effort.

Node	Select Path	Add1 Path	Add2 Path	Shift Path
R0	2	2	2	2
R1		1.3	1.3	8
R2		60	11	
R3			224	
R4	332			
B	**664**	**156**	**6290**	**16**

C.1.4 Data Path T1D5

Figure C.4 shows the T1D5 data path. For the data path T1D5 consider the following paths to estimate the delay:

- Select1 Path: `remainder-reg` $\rightarrow$ `QSLC` $\rightarrow$ `2:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `remainder-reg`
- Add1 Path: `remainder-reg` $\rightarrow$ `QSLC` $\rightarrow$ `4:1 Mux` $\rightarrow$ `CSA&2X*` $\rightarrow$ `2:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `remainder-reg`
- Add2 Path: `remainder-reg` $\rightarrow$ `amp` $\rightarrow$ `4:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `quotient-reg`
- Shift Path: `remainder-reg` $\rightarrow$ `2X*` $\rightarrow$ `2:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `remainder-reg`

Tables C.7 lists the logical effort and parasitic delay of the logic gates in the select, add and shift paths along with the number of stages in each gate. Table C.8 lists the branching effort at various nodes in the respective paths.

Using the values of G, P and N in Table C.7 and B in Table C.8, the delays of the data path when used in a synchronous and self-timed environment are

$$D_{sync} = 12.2 \text{ FO4}, \qquad D_{async} = 10.6 \text{ FO4}. \tag{C.1.4}$$

Table C.7: Data Path T1D5: Logical Effort and parasitic delay of the gates in the select, add and shift paths.

Gate	Select Path			Add1 Path			Add2 Path			Shift Path		
	g	p	n	g	p	n	g	p	n	g	p	n
reg	2	2	1	2	2	1	2	2	1	2	2	1
amp							1	2	2			
CSA				10	6	1						
4:1 Mux				2.7	6	2	2.7	6	2			
2:1 Mux	1.8	4	2	1.8	4	2	1.8	4	2	1.8	4	2
2:1 Mux	1.8	4	2	1.8	4	2	1.8	4	2	1.8	4	2
QSLC	11.3	4	1	10	4	2	10	4	2			
G	**72**			**1692**			**170**			**7**		
P		**14**			**24**			**22**			**10**	
N			**6**			**10**			**11**			**5**

Table C.8: Data Path T1D5: Branching effort.

Node	Select Path	Add1 Path	Add2 Path	Shift Path
R0	2	2	2	2
R1		1.2	1.2	81
R2		60	12	
R3			224	
R4	332			
B	**664**	**144**	**6451**	**162**

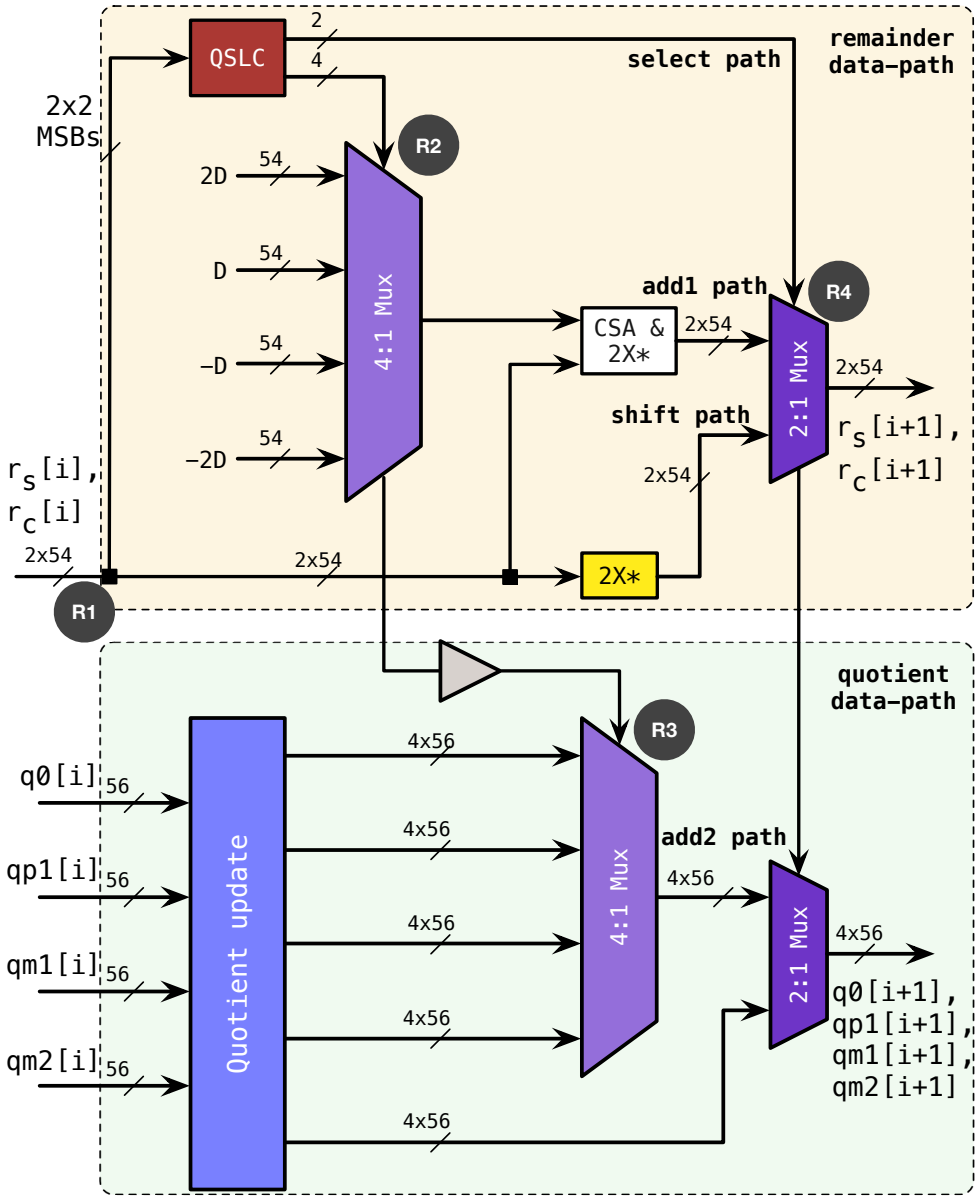


Fig. C.4: Data Path T1D5

C.2 Topology 2

C.2.1 Data Path T2D1

Figure C.1 shows the T2D1 data path. For the data path T2D1 consider the following paths to estimate the delay:

- Select Path: `qslc-reg` $\rightarrow$ `amp` $\rightarrow$ `5:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `qslc-reg`
- Add Path: `remainder-reg` $\rightarrow$ `ADD2&2X* or ADD1&2X* or SUB1&2X* or SUB2&2X*` $\rightarrow$ `QSLC` $\rightarrow$ `5:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `qslc-reg`
- Shift Path: `remainder-reg` $\rightarrow$ `2X*` $\rightarrow$ `QSLC` $\rightarrow$ `5:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `qslc-reg`

Table C.1 lists the logical effort and parasitic delay of the logic gates in the select, add and shift paths along with the number of stages in each gate. Table C.2 lists the branching effort at various nodes in the respective paths.

Using the values of G, P and N in Table C.1 and B in Table C.2, the delays of the data path when used in a synchronous and self-timed environment are

$$\begin{aligned} D_{sync} &= 10.3 \text{ FO4}, \\ D_{async} &= 9.2 \text{ FO4}. \end{aligned} \tag{C.2.1}$$

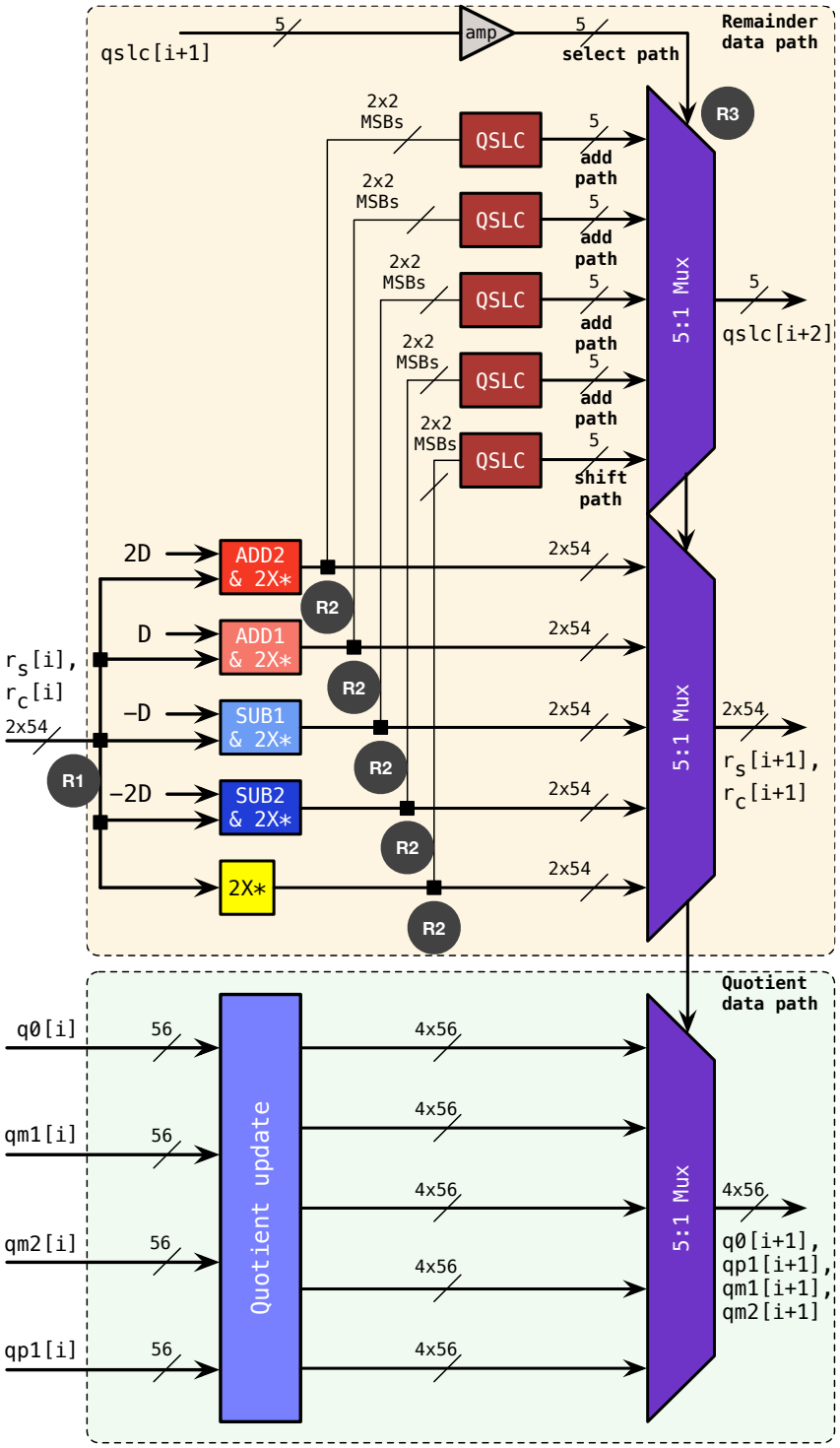


Fig. C.1: Data Path T2D1

Table C.1: Data Path T2D1: Logical Effort and parasitic delay of the gates in the select, add and shift paths.

Gate	Select Path			Add Path			Shift Path		
	g	p	n	g	p	n	g	p	n
reg	2	2	1	2	2	1	2	2	1
amp	1	2	2						
CSA				10	6	1			
5:1 Mux	3.3	7	2	3.3	7	2	3.3	7	2
2:1 Mux	1.8	4	2	1.8	4	2	1.8	4	2
QSLC				10	4	2	10	4	2
G	**12**			**1186**			**119**		
P		**15**			**23**			**17**	
N			**7**			**8**			**7**

Table C.2: Data Path t2D1: Branching effort.

Node	Select Path	Add Path	Shift Path
R0	2	2	2
R1		4.4	28.4
R2		2.6	
R3	338		
B	**676**	**23**	**57**

C.2.2 Data Path T2D2

Figure C.2 shows the T2D2 data path. For the data path T2D2 consider the following paths to estimate the delay:

- Select Path: `qslc-reg` $\rightarrow$ `2:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `qslc-reg`
- Add1 Path: `qslc-reg` $\rightarrow$ `amp` $\rightarrow$ `amp` $\rightarrow$ `4:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `qslc-reg`

- Add2 Path: `remainder-reg` $\rightarrow$ `ADD2&2X* or ADD1&2X* or SUB1&2X* or SUB2&2X*` $\rightarrow$ `QSLC` $\rightarrow$ `4:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `qslc-reg`
- Shift Path: `remainder-reg` $\rightarrow$ `2X*` $\rightarrow$ `QSLC` $\rightarrow$ `2:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `qslc-reg`

Table C.3 lists the logical effort and parasitic delay of the logic gates in the select, add and shift paths along with the number of stages in each gate. Table C.4 lists the branching effort at various nodes in the respective paths.

Using the values of G, P and N in Table C.3 and B in Table C.4, the delays of the data path when used in a synchronous and self-timed environment are

$$\begin{aligned} D_{sync} &= 11.2 \text{ FO4}, \\ D_{async} &= 9.0 \text{ FO4}. \end{aligned} \tag{C.2.2}$$

Table C.3: Data Path T2D2: Logical Effort and parasitic delay of the gates in the select, add and shift paths.

Gate	Select Path			Add1 Path			Add2 Path			Shift Path		
	g	p	n	g	p	n	g	p	n	g	p	n
reg	2	2	1	2	2	1	2	2	1	2	2	1
amp	1	2	2	1	2	2						
amp				1	2	2						
CSA							10	6	1			
4:1 Mux				2.7	6	2	2.7	6	2			
2:1 Mux	1.8	4	2	1.8	4	2	1.8	4	2	1.8	4	2
2:1 Mux	1.8	4	2	1.8	4	2	1.8	4	2	1.8	4	2
QSLC							10	4	2	10	4	2
G	**7**			**17**			**1692**			**64**		
P		**12**			**20**			**26**			**14**	
N			**7**			**11**			**10**			**7**

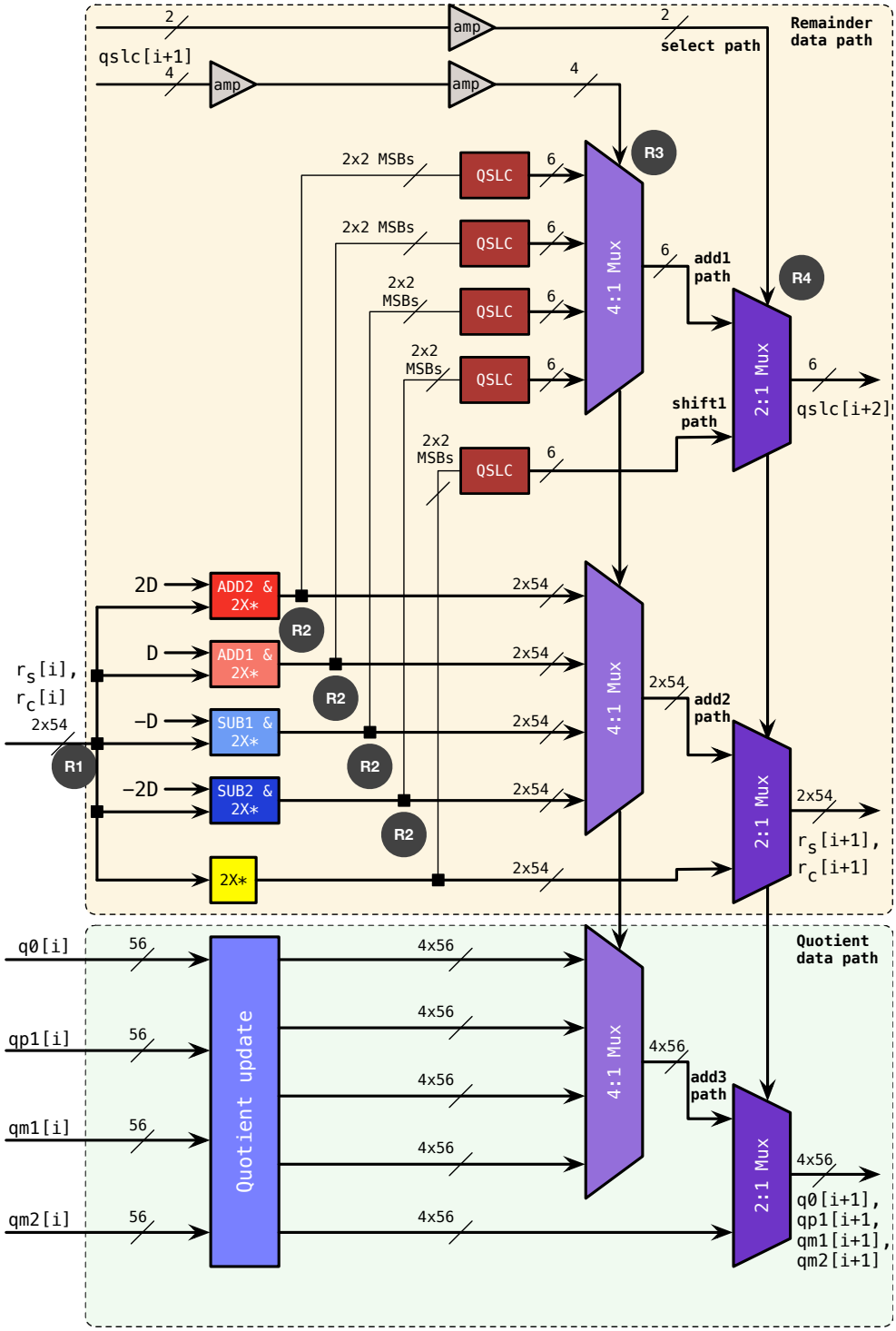


Fig. C.2: Data Path T2D2

Table C.4: Data Path T2D2: Branching effort.

Node	Select Path	Add1 Path	Add2 Path	Shift Path
R0	2	2	2	2
R1			6.4	7
R2			2.6	
R3		338		
R4	338			
B	**676**	**676**	**33**	**14**

C.2.3 Data Path T2D3

Figure C.3 shows the T2D3 data path. For the data path T2D3 consider the following paths to estimate the delay:

- Select Path: `qslc-reg` $\rightarrow$ `amp` $\rightarrow$ `3:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `qslc-reg`
- Add1 Path: `qslc-reg` $\rightarrow$ `amp` $\rightarrow$ `amp` $\rightarrow$ `2:1 Mux` $\rightarrow$ `3:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `qslc-reg`
- Add2 Path: `remainder-reg` $\rightarrow$ `ADD2&2X* or ADD1&2X* or SUB1&2X* or SUB2&2X*` $\rightarrow$ `QSLC` $\rightarrow$ `2:1 Mux` $\rightarrow$ `3:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `qslc-reg`
- Shift Path: `remainder-reg` $\rightarrow$ `2X*` $\rightarrow$ `QSLC` $\rightarrow$ `3:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `qslc-reg`

Table C.5 lists the logical effort and parasitic delay of the logic gates in the select, add and shift paths along with the number of stages in each gate. Table C.6 lists the branching effort at various nodes in the respective paths.

Using the values of G, P and N in Table C.5 and B in Table C.6, the delays of the data path when used in a synchronous and self-timed environment are

$$D_{sync} = 10.6 \text{ FO4},$$
$$D_{async} = 9.1 \text{ FO4}. \tag{C.2.3}$$

Table C.5: Data Path T2D3: Logical Effort and parasitic delay of the gates in the select, add and shift paths.

Gate	Select Path			Add1 Path			Add2 Path			Shift Path		
	g	p	n	g	p	n	g	p	n	g	p	n
reg	2	2	1	2	2	1	2	2	1	2	2	1
amp	1	2	2	1	2	2						
amp				1	2	2						
CSA							10	6	1			
2:1 Mux				1.8	4	2	1.8	4	2			
3:1 Mux	2.7	5	2	2.7	5	2	2.7	5	2	2.7	5	2
2:1 Mux	1.8	4	2	1.8	4	2	1.8	4	2	1.8	4	2
QSLC							6	4	1	6	4	1
G	**10**			**17**			**1016**			**58**		
P		**13**			**19**			**25**			**15**	
N			**7**			**11**			**9**			**6**

Table C.6: Data Path T2D3: Branching effort.

Node	Select Path	Add1 Path	Add2 Path	Shift Path
R0	2	2	2	2
R1			7.3	3.7
R2			1.7	
R3		338		
R4	338			
B	**676**	**676**	**25**	**7**

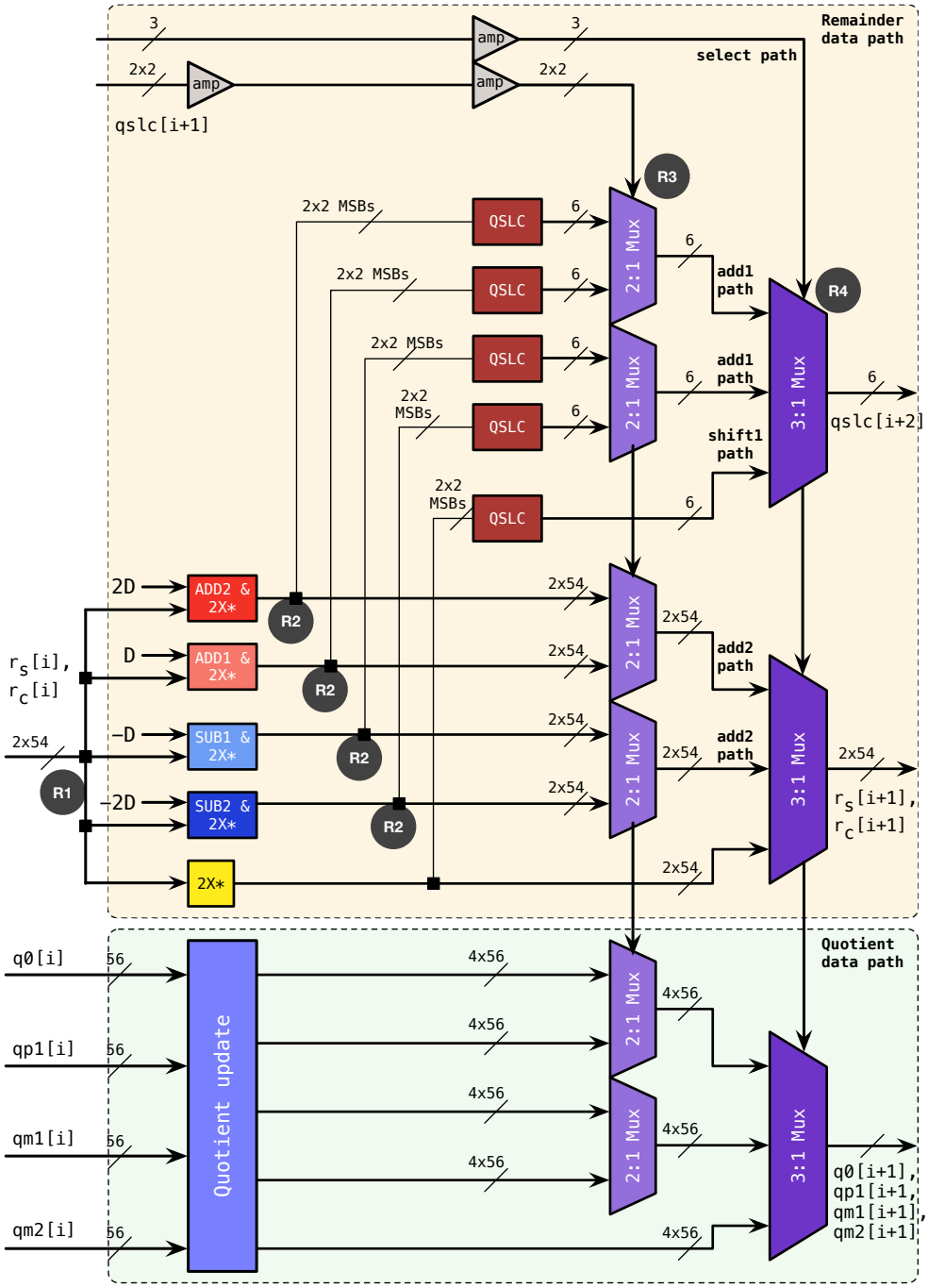


Fig. C.3: Data Path T2D3

C.2.4 Data Path T2D4

Figure C.4 shows the T2D4 data path. For the data path T2D4 consider the following paths to estimate the delay:

- Select Path: `qslc-reg` $\rightarrow$ `amp` $\rightarrow$ `3:1 Mux` $\rightarrow$ `QSLC` $\rightarrow$ `2:1 Mux` $\rightarrow$ `qslc-reg`
- Add1 Path: `qslc-reg` $\rightarrow$ `2:1 Mux` $\rightarrow$ `ADD&2X* or SUB&2X*` $\rightarrow$ `QSLC` $\rightarrow$ `3:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `qslc-reg`
- Add2 Path: `qslc-reg` $\rightarrow$ `amp` $\rightarrow$ `2:1 Mux` $\rightarrow$ `ADD&2X* or SUB&2X*` $\rightarrow$ `3:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `remainder-reg`
- Add3 Path: `qslc-reg` $\rightarrow$ `amp` $\rightarrow$ `amp` $\rightarrow$ `2:1 Mux` $\rightarrow$ `3:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `quotient-reg`
- Shift Path: `remainder-reg` $\rightarrow$ `2X*` $\rightarrow$ `QSLC` $\rightarrow$ `3:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `qslc-reg`

Tables C.7 and C.8 list the logical effort and parasitic delay of the logic gates in the select and shift paths, and add paths, respectively, along with the number of stages in each gate. Table C.9 lists the branching effort at various nodes in the respective paths.

Using the values of G, P and N in Tables C.7 and C.8, and B in Table C.9, the delays of the data path when used in a synchronous and self-timed environment are

$$\begin{aligned} D_{sync} &= 10.6 \text{ FO4}, \\ D_{async} &= 9.0 \text{ FO4}. \end{aligned} \tag{C.2.4}$$

Table C.7: Data Path T2D4: Logical Effort and parasitic delay of the gates in the select and shift paths.

Gate	Select Path			Shift Path		
	g	p	n	g	p	n
reg	2	2	1	2	2	1
amp	1	2	2			
3:1 Mux	2.7	5	2	2.7	5	2
2:1 Mux	1.8	4	2	1.8	4	2
QSLC				6	4	1
G	**10**			**58**		
P		**13**			**15**	
N			**7**			**6**

Table C.8: Data Path T2D4: Logical Effort and parasitic delay of the gates in the add paths.

Gate	Add1 Path			Add2 Path			Add3 Path		
	g	p	n	g	p	n	g	p	n
reg	2	2	1	2	2	1	2	2	1
amp				1	2	2	1	2	2
amp							1	2	2
CSA	10	6	1	10	6	1			
3:1 Mux	2.7	5	2	2.7	5	2	2.7	5	2
2:1 Mux	1.8	4	2	1.8	4	2	1.8	4	2
2:1 Mux	1.8	4	2	1.8	4	2	1.8	4	2
QSLC	6	4	1						
G	**1016**			**170**			**17**		
P		**25**			**23**			**19**	
N			**9**			**10**			**11**

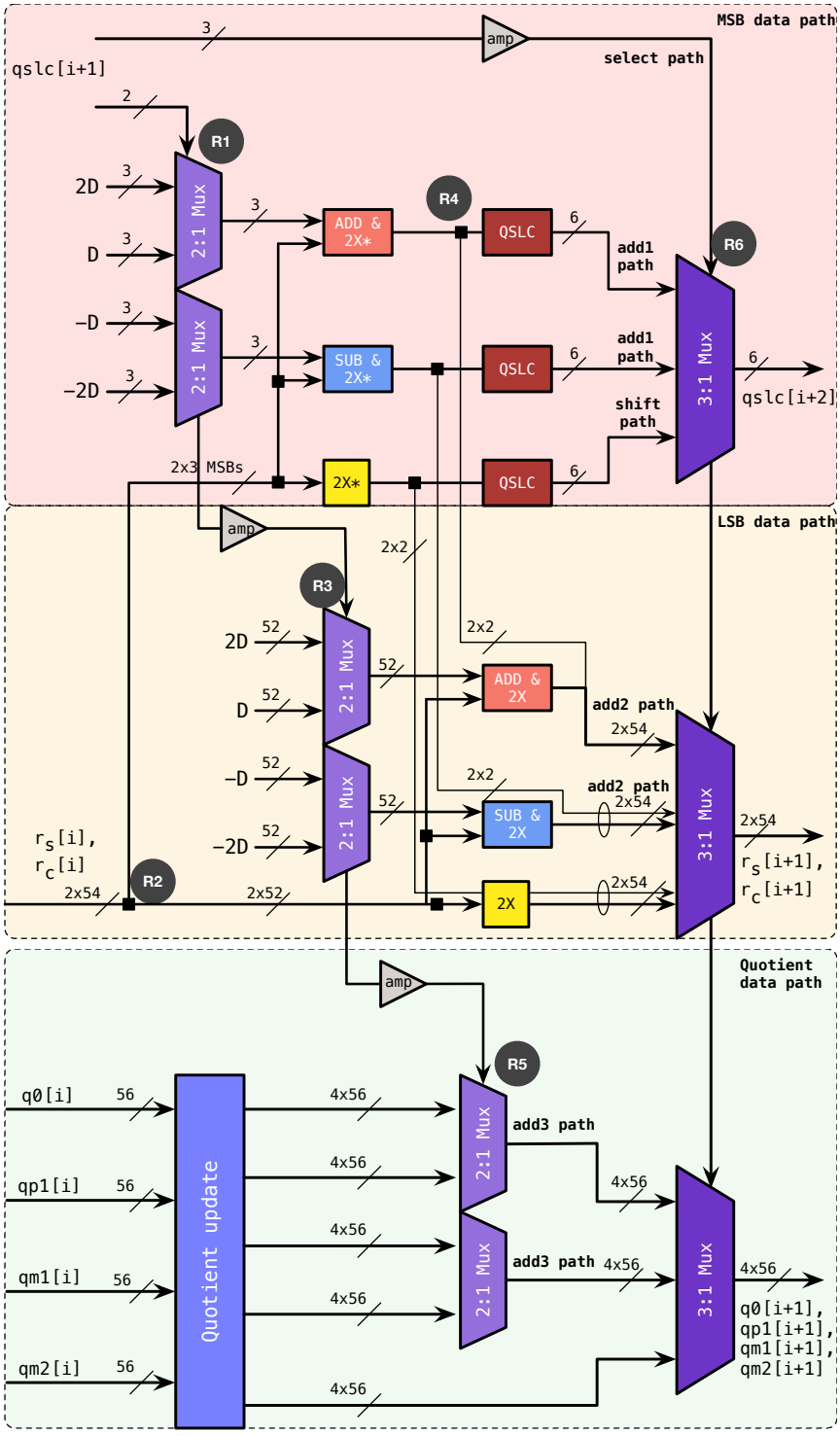


Fig. C.4: Data Path T2D4

Table C.9: Data Path T2D4: Branching effort.

Node	Select Path	Add1 Path	Add2 Path	Add3 Path	Shift Path
R0	2	2	2	2	2
R1		4.4	3.4	32.3	
R2					10
R3			52		
R4		1.7			
R5				224	
R6	338				
B	**676**	**15**	**354**	**14470**	**20**

C.3 Topology 3

C.3.1 Data Path T3D1

Figure C.1 shows the T3D1 data path. For the data path T3D1 consider the following paths to estimate the delay:

- Select1 Path: `qslc-reg` → `amp` → `5:1 Mux` → `QSLC` → `2:1 Mux` → `qslc-reg`
- Select2 Path: `qslc-reg` → `amp` → `5:1 Mux` → `QSLC` → `2:1 Mux` → `qslc-reg`
- Add Path: `remainder-reg` → `ADD2&2X* or ADD1&2X* or SUB1&2X* or SUB2&2X*` → `5:1 Mux` → `QSLC` → `2:1 Mux` → `qslc-reg`
- Shift Path: `remainder-reg` → `2X*` → `5:1 Mux` → `QSLC` → `2:1 Mux` → `qslc-reg`

Table C.1 lists the logical effort and parasitic delay of the logic gates in the select, add and shift paths along with the number of stages in each gate. Table C.2 lists the branching effort at various nodes in the respective paths.

Using the values of G, P and N in Table C.1 and B in Table C.2, the delays of the data path when used in a synchronous and self-timed environment are

$$\begin{aligned} D_{sync} &= 10.3 \text{ FO4}, \\ D_{async} &= 9.5 \text{ FO4}. \end{aligned} \tag{C.3.1}$$

Table C.1: Data Path T3D1: Logical Effort and parasitic delay of the gates in the select, add and shift paths.

Gate	Select1 Path			Select2 Path			Add Path			Shift Path		
	g	p	n	g	p	n	g	p	n	g	p	n
reg	2	2	1	2	2	1	2	2	1	2	2	1
amp	1	2	2	1	2	2						
amp				1	2	2						
CSA							10	6	1			
5:1 Mux	3.3	7	2	3.3	7	2	3.3	7	2	3.3	7	2
2:1 Mux	1.8	4	2	1.8	4	2	1.8	4	2	1.8	2	2
QSLC	10	4	2				10	4	2	10	4	2
G	**119**			**12**			**1186**			**119**		
P		**19**			**17**			**23**			**17**	
N			**9**			**9**			**8**			**7**

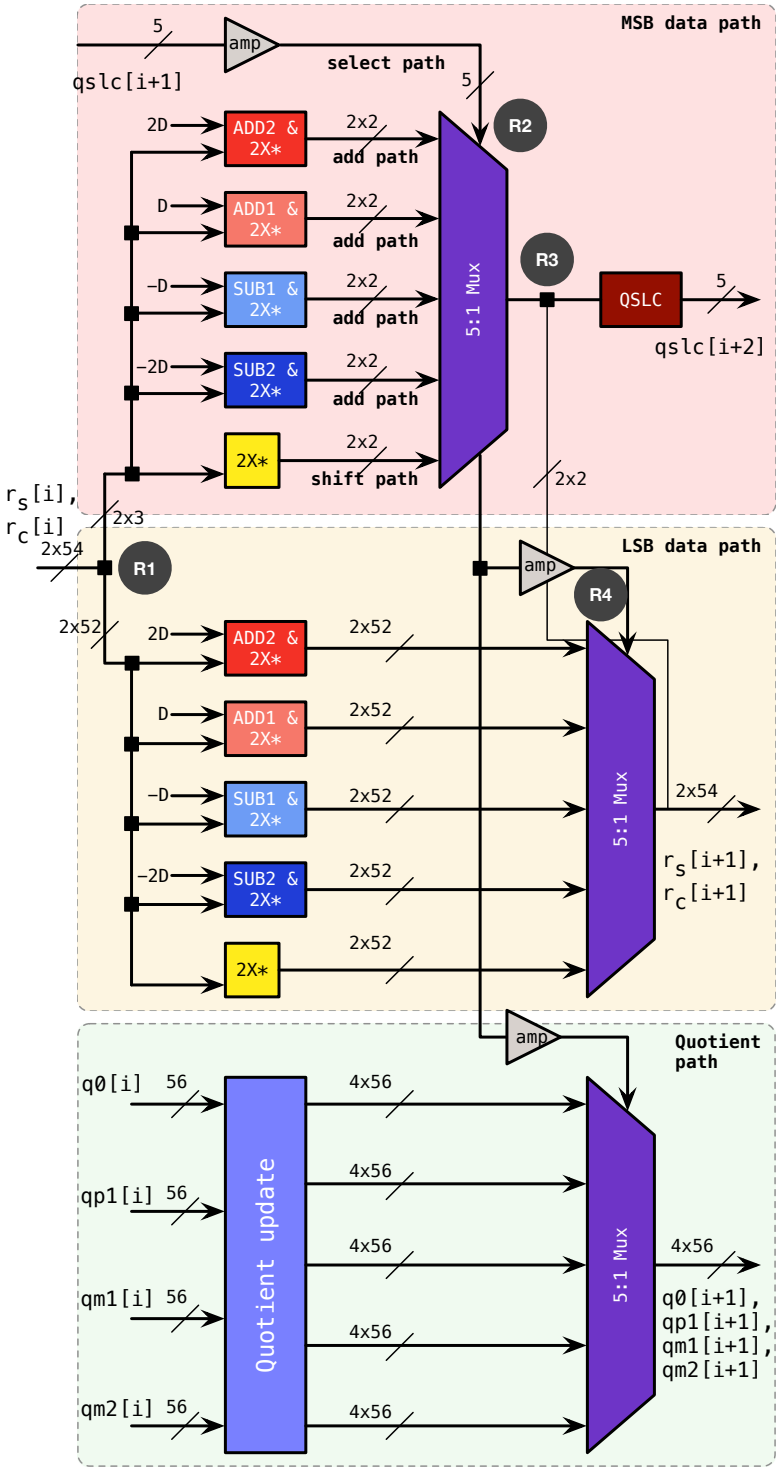


Fig. C.1: Data Path T3D1

Table C.2: Data Path T3D1: Branching effort.

Node	Select1 Path	Select2 Path	Add Path	Shift Path
R0	2	2	2	2
R1			4.4	11
R2	16.7	4.2		
R3	2.6		2.6	2.6
R4		104		
B	**87**	**874**	**23**	**57**

C.3.2 Data Path T3D2

Figure C.2 shows the T3D2 data path. For the data path T3D2 consider the following paths to estimate the delay:

- Select1 Path: `qslc-reg` $\rightarrow$ `2:1 Mux` $\rightarrow$ `QSLC` $\rightarrow$ `2:1 Mux` $\rightarrow$ `qslc-reg`
- Select2 Path: `qslc-reg` $\rightarrow$ `amp` $\rightarrow$ `2:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `remainder-reg`
- Add1 Path: `qslc-reg` $\rightarrow$ `amp` $\rightarrow$ `4:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `QSLC` $\rightarrow$ `2:1 Mux` $\rightarrow$ `qslc-reg`
- Add2 Path: `qslc-reg` $\rightarrow$ `amp` $\rightarrow$ `amp` $\rightarrow$ `4:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `remainder-reg`
- Add3 Path: `remainder-reg` $\rightarrow$ `ADD2&2X* or ADD1&2X* or SUB1&2X* or SUB2&2X*` $\rightarrow$ `4:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `QSLC` $\rightarrow$ `2:1 Mux` $\rightarrow$ `qslc-reg`
- Shift Path: `remainder-reg` $\rightarrow$ `2X*` $\rightarrow$ `2:1 Mux` $\rightarrow$ `QSLC` $\rightarrow$ `2:1 Mux` $\rightarrow$ `qslc-reg`

Tables C.3 and C.4 list the logical effort and parasitic delay of the logic gates in the select and shift paths, and add paths, respectively, along with the number of stages in each gate. Table C.5 lists the branching effort at various nodes in the respective paths.

Using the values of G, P and N in Tables C.3 and C.4, and B in Table C.5, the delays of the data path when used in a synchronous and self-timed environment are

$$\begin{aligned} D_{sync} &= 11.2 \text{ FO4}, \\ D_{async} &= 9.3 \text{ FO4}. \end{aligned} \tag{C.3.2}$$

Table C.3: Data Path T3D2: Logical Effort and parasitic delay of the gates in the select and shift paths.

Gate	Select1 Path			Select2 Path			Shift Path		
	g	p	n	g	p	n	g	p	n
reg	2	2	1	2	2	1	2	2	1
amp				1	2	2			
2:1 Mux	1.8	4	2	1.8	4	2	1.8	4	2
2:1 Mux	1.8	4	2	1.8	4	2	1.8	4	2
QSLC	10	4	2				10	4	2
G	**64**			**7**			**64**		
P		**14**			**12**			**14**	
N			**7**			**7**			**8**

Table C.4: Data Path T3D2: Logical Effort and parasitic delay of the gates in the add paths.

Gate	Add1 Path			Add2 Path			Add3 Path		
	g	p	n	g	p	n	g	p	n
reg	2	2	1	2	2	1	2	2	1
amp	1	2	2	1	2	2			
amp				1	2	2			
CSA							10	6	1
4:1 Mux	2.67	6	2	2.7	6	2	2.7	6	2
2:1 Mux	1.8	4	2	1.8	4	2	1.8	4	2
2:1 Mux	1.8	4	2	1.8	4	2	1.8	4	2
QSLC	10	4	2				10	4	2
G	**170**			**17**			**1692**		
P		**22**			**20**			**26**	
N			**11**			**11**			**10**

Table C.5: Data Path T3D2: Branching effort.

Node	Select1 Path	Select2 Path	Add1 Path	Add2 Path	Add3 Path	Shift Path
R0	2	2	2	2	2	2
R1					6.4	2.7
R2	16.5	4.2	17.7	4.1		
R3	2.6		2.6		2.6	2.6
R4		104		104		
B	**86**	**874**	**92**	**853**	**33**	**14**

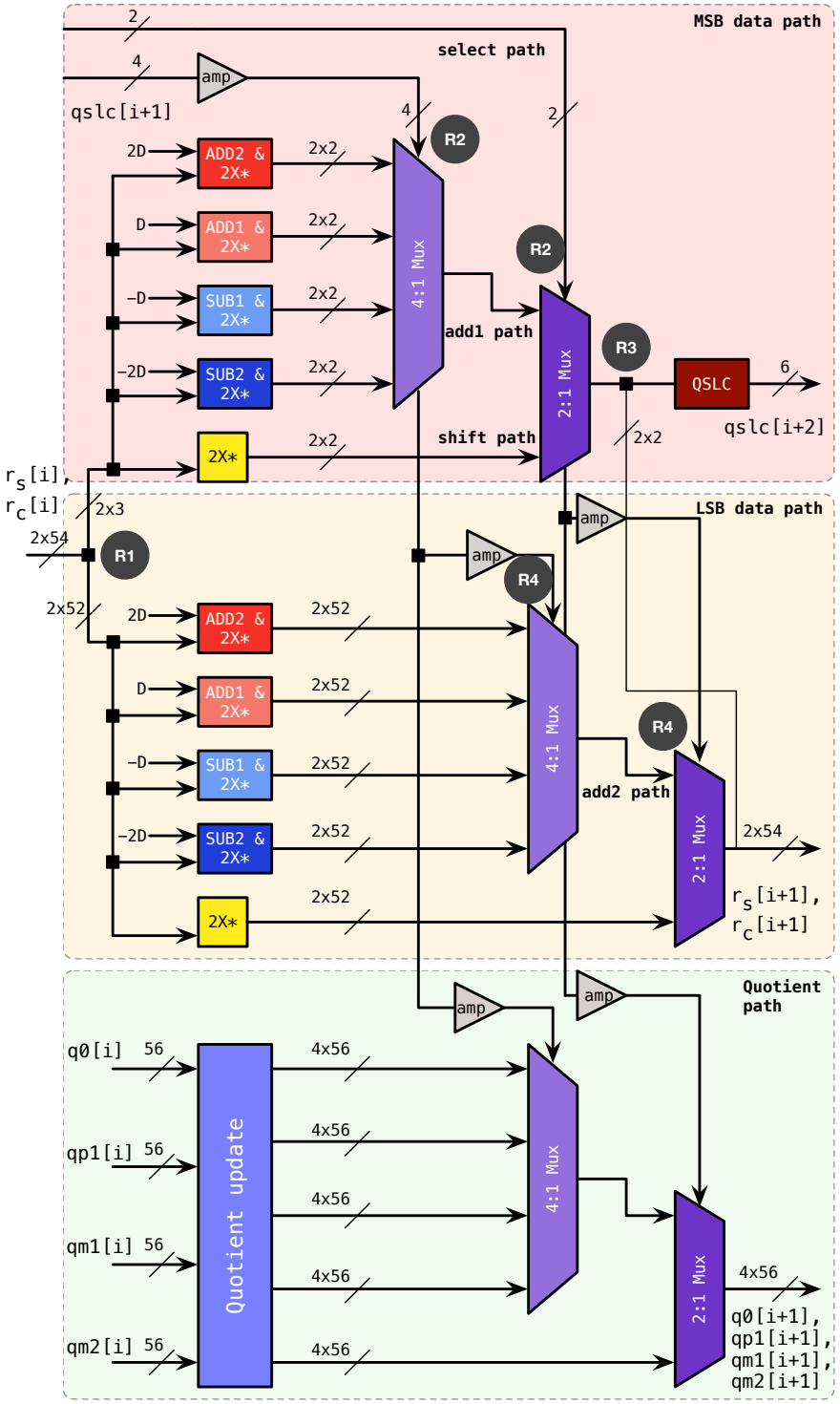


Fig. C.2: Data Path T3D2

C.3.3 Data Path T3D3

Figure C.3 shows the T3D3 data path. For the data path T3D3 consider the following paths to estimate the delay:

- Select1 Path: `qslc-reg` $\rightarrow$ `3:1 Mux` $\rightarrow$ `QSLC` $\rightarrow$ `2:1 Mux` $\rightarrow$ `qslc-reg`
- Select2 Path: `qslc-reg` $\rightarrow$ `amp` $\rightarrow$ `3:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `remainder-reg`
- Add1 Path: `qslc-reg` $\rightarrow$ `amp` $\rightarrow$ `2:1 Mux` $\rightarrow$ `3:1 Mux` $\rightarrow$ `QSLC` $\rightarrow$ `2:1 Mux` $\rightarrow$ `qslc-reg`
- Add2 Path: `qslc-reg` $\rightarrow$ `amp` $\rightarrow$ `amp` $\rightarrow$ `2:1 Mux` $\rightarrow$ `3:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `remainder-reg`
- Add3 Path: `remainder-reg` $\rightarrow$ `ADD2&2X* or ADD1&2X* or SUB1&2X* or SUB2&2X*` $\rightarrow$ `2:1 Mux` $\rightarrow$ `3:1 Mux` $\rightarrow$ `QSLC` $\rightarrow$ `2:1 Mux` $\rightarrow$ `qslc-reg`
- Shift Path: `remainder-reg` $\rightarrow$ `2X*` $\rightarrow$ `3:1 Mux` $\rightarrow$ `QSLC` $\rightarrow$ `2:1 Mux` $\rightarrow$ `qslc-reg`

Tables C.6 and C.7 list the logical effort and parasitic delay of the logic gates in the select and shift paths, and add paths, respectively, along with the number of stages in each gate. Table C.8 lists the branching effort at various nodes in the respective paths.

Using the values of G, P and N in Tables C.6 and C.7, and B in Table C.8, the delays of the data path when used in a synchronous and self-timed environment are

$$\begin{aligned} D_{sync} &= 10.6 \text{ FO4}, \\ D_{async} &= 9.1 \text{ FO4}. \end{aligned} \tag{C.3.3}$$

Table C.6: Data Path T3D3: Logical Effort and parasitic delay of the gates in the select and shift paths.

Gate	Select1 Path			Select2 Path			Shift Path		
	g	p	n	g	p	n	g	p	n
reg	2	2	1	2	2	1	2	2	1
amp				1	2	2			
3:1 Mux	2.7	5	2	2.7	5	2	2.7	5	2
2:1 Mux	1.8	4	2	1.8	4	2	1.8	4	2
QSLC	6	4	1				6	4	1
G	**58**			**10**			**58**		
P		**15**			**13**			**15**	
N			**6**			**7**			**9**

Table C.7: Data Path T3D3: Logical Effort and parasitic delay of the gates in the add paths.

Gate	Add1 Path			Add2 Path			Add3 Path		
	g	p	n	g	p	n	g	p	n
reg	2	2	1	2	2	1	2	2	1
amp	1	2	2	1	2	2			
amp				1	2	2			
CSA							10	6	1
3:1 Mux	2.7	5	2	2.7	5	2	2.7	5	2
2:1 Mux	1.8	4	2	1.8	4	2	1.8	4	2
2:1 Mux	1.8	4	2	1.8	4	2	1.8	4	2
QSLC	6	4	1				6	4	1
G	**102**			**17**			**1016**		
P		**21**			**19**			**25**	
N			**10**			**11**			**10**

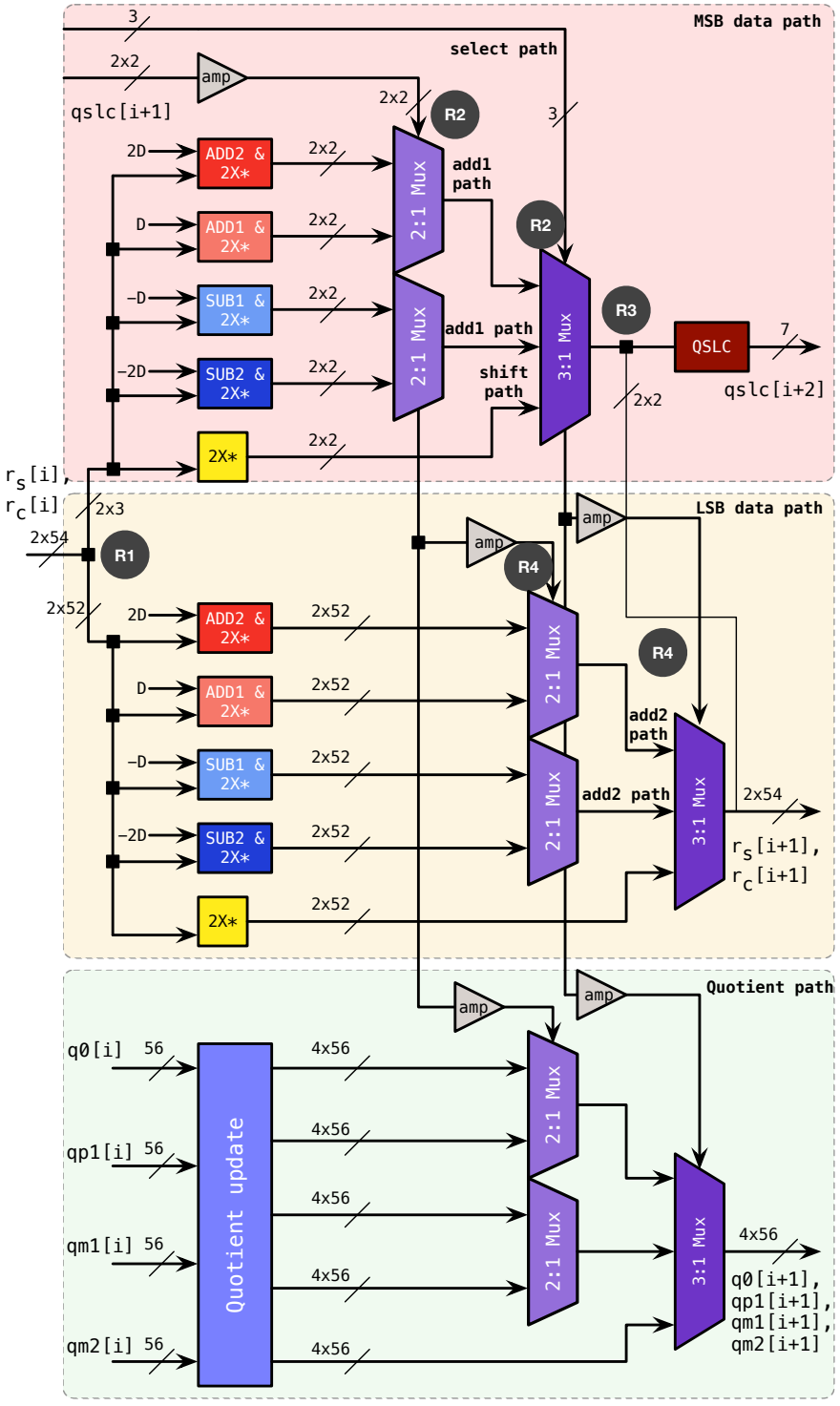


Fig. C.3: Data Path T3D3

Table C.8: Data Path T3D3: Branching effort.

Node	Select1 Path	Select2 Path	Add1 Path	Add2 Path	Add3 Path	Shift Path
R0	2	2	2	2	2	2
R1					7.6	2.1
R2	12.3	4.7	12.2	4.7		
R3	1.7		1.7		1.7	1.7
R4		104		104		
B	**42**	**978**	**42**	**978**	**26**	**7**

C.3.4 Data Path T3D4

Figure C.4 shows the T3D4 data path. For the data path T3D4 consider the following paths to estimate the delay:

- Select1 Path: `qslc-reg` $\rightarrow$ `3:1 Mux` $\rightarrow$ `QSLC` $\rightarrow$ `2:1 Mux` $\rightarrow$ `qslc-reg`
- Select2 Path: `qslc-reg` $\rightarrow$ `amp` $\rightarrow$ `3:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `remainder-reg`
- Add1 Path: `qslc-reg` $\rightarrow$ `2:1 Mux` $\rightarrow$ `ADD&2X* or SUB&2X*` $\rightarrow$ `3:1 Mux` $\rightarrow$ `QSLC` $\rightarrow$ `2:1 Mux` $\rightarrow$ `qslc-reg`
- Add2 Path: `qslc-reg` $\rightarrow$ `amp` $\rightarrow$ `2:1 Mux` $\rightarrow$ `ADD&2X* or SUB&2X*` $\rightarrow$ `3:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `remainder-reg`
- Add3 Path: `qslc-reg` $\rightarrow$ `amp` $\rightarrow$ `amp` $\rightarrow$ `2:1 Mux` $\rightarrow$ `3:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `quotient-reg`
- Shift Path: `remainder-reg` $\rightarrow$ `2X*` $\rightarrow$ `3:1 Mux` $\rightarrow$ `QSLC` $\rightarrow$ `2:1 Mux` $\rightarrow$ `qslc-reg`

Tables C.9 and C.10 list the logical effort and parasitic delay of the logic gates in the select and shift paths, and add paths, respectively, along with the number of

stages in each gate. Table C.11 lists the branching effort at various nodes in the respective paths.

Using the values of G, P and N in Tables C.9 and C.10, and B in Table C.11, the delays of the data path when used in a synchronous and self-timed environment are

$$\begin{aligned} D_{sync} &= 10.7 \text{ FO4}, \\ D_{async} &= 9.2 \text{ FO4}. \end{aligned} \tag{C.3.4}$$

Table C.9: Data Path T3D4: Logical Effort and parasitic delay of the gates in the select and shift paths.

Gate	Select1 Path			Select2 Path			Shift Path		
	g	p	n	g	p	n	g	p	n
reg	2	2	1	2	2	1	2	2	1
amp				1	2	2			
3:1 Mux	2.7	5	2	2.7	5	2	2.7	5	2
2:1 Mux	1.8	4	2	1.8	4	2	1.8	4	2
QSLC	6	4	1				6	4	1
G	**58**			**10**			**58**		
P		**15**			**13**			**15**	
N			**6**			**7**			**9**

Table C.10: Data Path T3D4: Logical Effort and parasitic delay of the gates in the add paths.

Gate	Add1 Path			Add2 Path			Add3 Path		
	g	p	n	g	p	n	g	p	n
reg	2	2	1	2	2	1	2	2	1
amp				1	2	2	1	2	2
amp							1	2	2
CSA	10	6	1	10	6	1			
3:1 Mux	2.7	5	2	2.7	5	2	2.7	5	2
2:1 Mux	1.8	4	2	1.8	4	2	1.8	4	2
2:1 Mux	1.8	4	2	1.8	4	2	1.8	4	2
QSLC	6	4	1						
G	**1016**			**170**			**17**		
P		**25**			**23**			**19**	
N			**9**			**10**			**6**

Table C.11: Data Path T3D4: Branching effort.

Node	Select1 Path	Select2 Path	Add1 Path	Add2 Path	Add3 Path	Shift Path
R0	2	2	2	2	2	2
R1						6
R2			5.5	4.1	37.7	
R3	12.3	4.7				
R4		104		52		
R5					224	
R6	1.7		1.7			1.7
B	**42**	**978**	**19**	**426**	**16890**	**20**

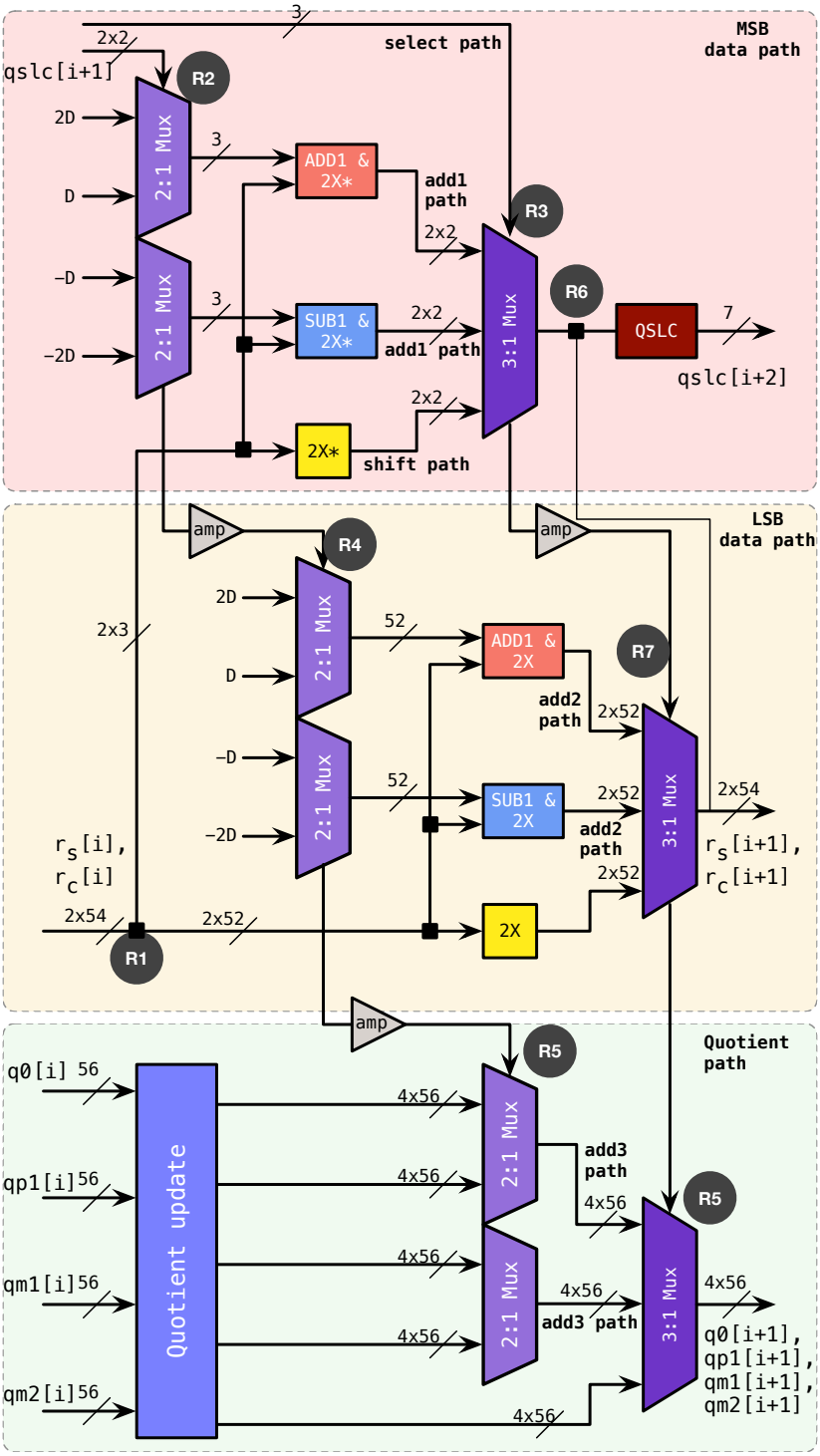


Fig. C.4: Data Path T3D4

C.3.5 Data Path T3D5

Figure C.5 shows the T3D5 data path. For the data path T3D5 consider the following paths to estimate the delay:

- Select1 Path: `qslc-reg` $\rightarrow$ `2:1 Mux` $\rightarrow$ `QSLC` $\rightarrow$ `2:1 Mux` $\rightarrow$ `qslc-reg`
- Select2 Path: `qslc-reg` $\rightarrow$ `amp` $\rightarrow$ `2:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `remainder-reg`
- Add1 Path: `qslc-reg` $\rightarrow$ `4:1 Mux` $\rightarrow$ `CSA&2X*` $\rightarrow$ `2:1 Mux` $\rightarrow$ `QSLC` $\rightarrow$ `2:1 Mux` $\rightarrow$ `qslc-reg`
- Add2 Path: `qslc-reg` $\rightarrow$ `amp` $\rightarrow$ `4:1 Mux` $\rightarrow$ `CSA&2X*` $\rightarrow$ `2:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `remainder-reg`
- Add3 Path: `qslc-reg` $\rightarrow$ `amp` $\rightarrow$ `amp` $\rightarrow$ `4:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `2:1 Mux` $\rightarrow$ `quotient-reg`
- Shift Path: `remainder-reg` $\rightarrow$ `2X*` $\rightarrow$ `2:1 Mux` $\rightarrow$ `QSLC` $\rightarrow$ `2:1 Mux` $\rightarrow$ `qslc-reg`

Tables C.12 and C.13 list the logical effort and parasitic delay of the logic gates in the select and shift paths, and add paths, respectively, along with the number of stages in each gate. Table C.14 lists the branching effort at various nodes in the respective paths.

Using the values of G, P and N in Tables C.12 and C.13, and B in Table C.14, the delays of the data path when used in a synchronous and self-timed environment are

$$\begin{aligned} D_{sync} &= 11.2 \text{ FO4}, \\ D_{async} &= 9.3 \text{ FO4}. \end{aligned} \tag{C.3.5}$$

Table C.12: Data Path T3D5: Logical Effort and parasitic delay of the gates in the select and shift paths.

Gate	Select1 Path			Select2 Path			Shift Path		
	g	p	n	g	p	n	g	p	n
reg	2	2	1	2	2	1	2	2	1
amp				1	2	2			
2:1 Mux	1.8	4	2	1.8	4	2	1.8	4	2
2:1 Mux	1.8	4	2	1.8	4	2	1.8	4	2
QSLC	10	4	2				10	4	2
G	**64**			**7**			**64**		
P		**14**			**12**			**14**	
N			**10**			**7**			**7**

Table C.13: Data Path T3D5: Logical Effort and parasitic delay of the gates in the add paths.

Gate	Add1 Path			Add2 Path			Add3 Path		
	g	p	n	g	p	n	g	p	n
reg	2	2	1	2	2	1	2	2	1
amp				1	2	2	1	2	2
amp							1	2	2
CSA	10	6	1	10	6	1			
4:1 Mux	2.7	6	2	2.7	6	2	2.7	6	2
2:1 Mux	1.8	4	2	1.8	4	2	1.8	4	2
2:1 Mux	1.8	4	2	1.8	4	2	1.8	4	2
QSLC	10	4	2						
G	**1692**			**170**			**17**		
P		**26**			**24**			**20**	
N			**10**			**10**			**11**

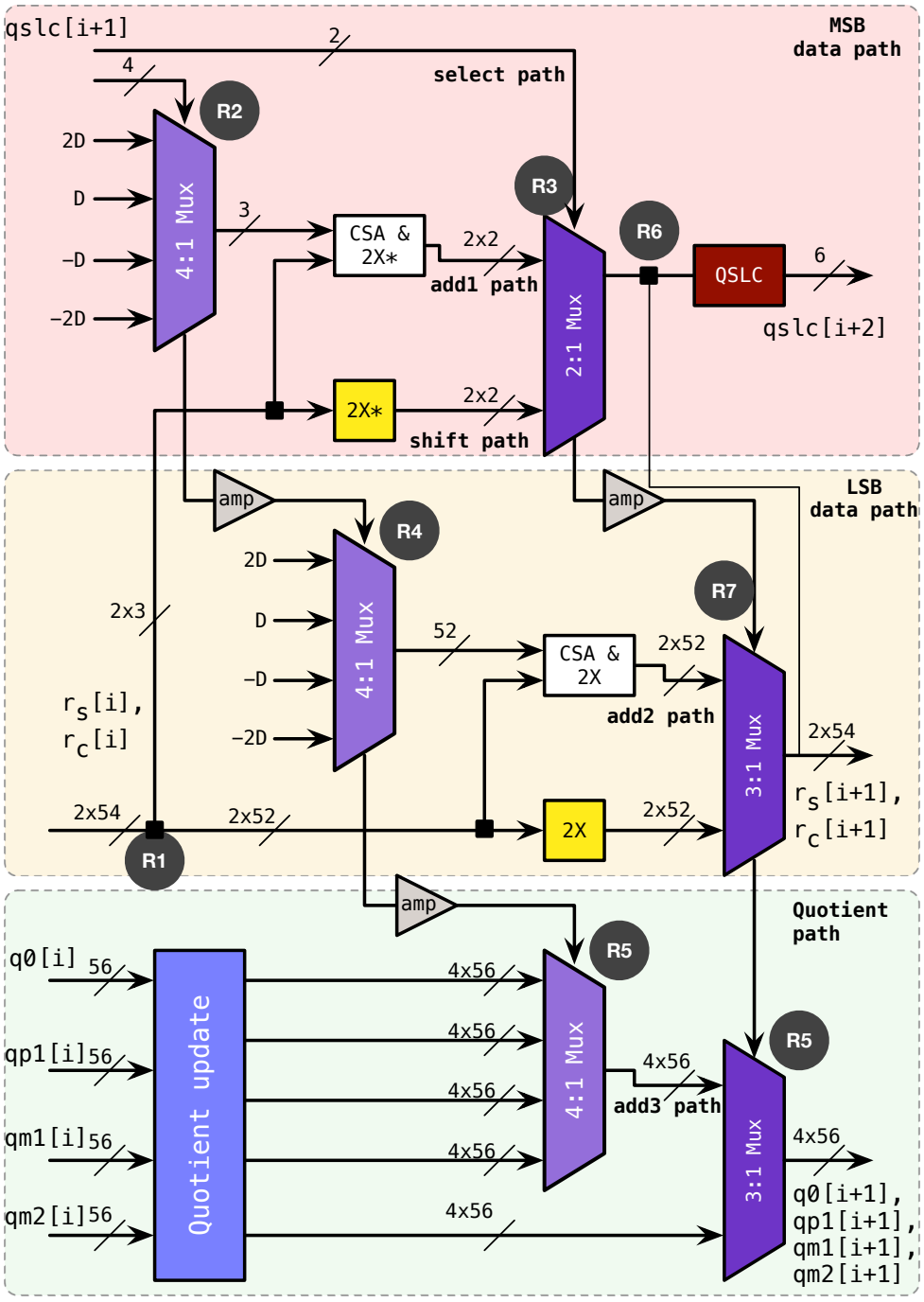


Fig. C.5: Data Path T3D5

Table C.14: Data Path T3D5: Branching effort.

Node	Select1 Path	Select2 Path	Add1 Path	Add2 Path	Add3 Path	Shift Path
R0	2	2	2	2	2	2
R1						3.5
R2			6.3	3.1	28.6	
R3	16.5	4.2				
R4				52		
R5					224	
R6	2.6		2.6			2.6
R7		104				
B	**86**	**874**	**33**	**322**	**12813**	**18**

Appendix D

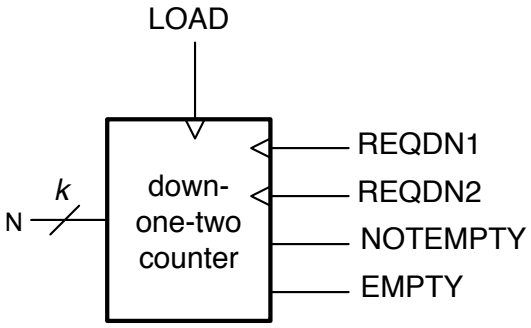
Design of a Down Counter that can Decrement by one or two

D.1 Specification of Down Counters

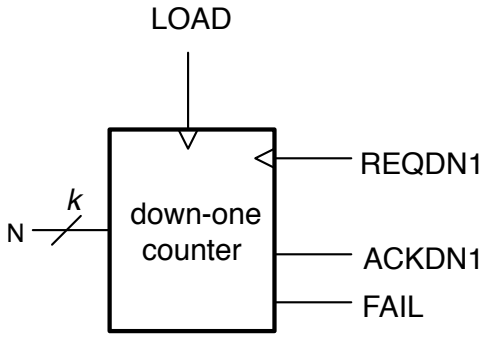
Down counters can be classified into two types: “test-after-decrement" and “test-before-decrement" counters. A “test-after-decrement" counter first decrements the counter and then tests if the value of the counter is zero, whereas a “test-before-decrement" counter first tests if the value of the counter is zero and, if not, then decrements the counter. This paper presents the design of a “test-after-decrement" down-one-two counter using a “test-before-decrement" down-one counter, because we have found that “test-before-decrement" counters are often easier to design.

The specification of the down-one and down-one-two counters are as follows. A down counter, upon initialization, loads a user-determined k-digit count value, N. We denote the initialization request of the counter by `LOAD`. After initialization, the “test-after-decrement" down-one-two counter can repeatedly be decremented by one or

two. The decrement requests are denoted by REQDN1 and REQDN2 respectively. As long as the count value is non-zero, the counter acknowledges each request with a "not empty " response, denoted by NOTEMPTY. When the count value reaches zero, the counter acknowledges a request with an "empty" response, denoted by EMPTY. After an "empty" response, the counter can accept the same or a different count value using a LOAD request. Figure D.1a illustrates the circuit symbol for the down-one-two counter.



(a) down-one-two counter



(b) down-one counter

Fig. D.1: Symbol of: a) down-one-two counter b) down-one counter

The circuit symbol for a down-one counter that tests the count value *before* a decrement appears in Figure D.1b. If the counter value before a decrement is larger than zero, then the counter acknowledges a successful decrement, denoted by ACKDN1.

If the counter value before a decrement is zero, then the counter acknowledges an unsuccessful decrement, denoted by `FAIL`.

The response `FAIL` means that the down request failed, but can also be interpreted as the counter value is zero. Note that a response `ACKDN1` simply means that a decrement has occurred and that the value of the counter can be anything, including zero. After a `FAIL` response, the counter can be loaded with a new count value using a `LOAD` request.

In Figure D.1 we denote the "test-after-decrement" counter and the "test-before-decrement" counter with similar circuit symbols, but the outputs have different names to indicate the difference between the counters.

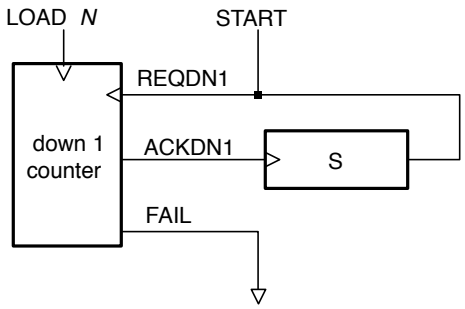
The "test-before-decrement" counter comes in handy when implementing a `while` or `for` loop. For example, the `while` repetition below can be implemented as shown in Figure D.2a.

```
n:=N;
while n>0 do
    S;
    n:=n-1;
endwhile
```

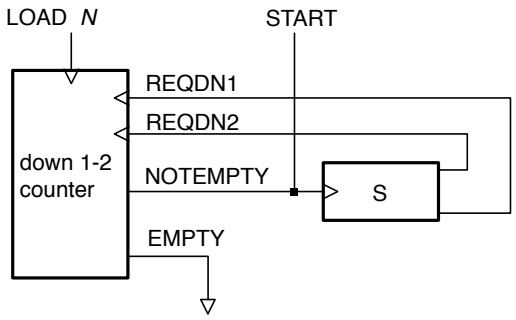
A "test-after-decrement" counter, which can decrement by 1 or 2, comes in handy when implementing the following `do..while..` loop.

```
n:=N;
do
   S;
   if B
     then begin ...; n:=n-1; ... end
     else begin ...; n:=n-2; ... end
   endif
while n>0;
```

An implementation of this repetition appears in Figure D.2b. The division algorithm presented in [16] has a `do..while..` loop as above.



(a) Repetition with a down-one counter



(b) Use of a down-one-two counter

Fig. D.2: Use of down-one counter (a) and down-one-two counter (b).

D.2 Down-one counter

This section presents the design of a “test-before-decrement" down-one counter using GasP modules.

D.2.1 The Idea for an implementation

To describe our down-one counter implementation, we illustrate the behavior of the counter by means of an example first. Assume that we load a six-bit counter with the binary value

```
100101
```

The left-most bit is the most significant bit. Thus, this binary representation denotes the value $1 * 2^5 + 0 * 2^4 + 0 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 37$. Although the initial count is in a unique binary representation, during operation the counter uses a redundant representation of the count, by allowing each digit to assume one of three values `0`, `1`, and `2`. Using this notation, the following are valid representations for 37.

```
020021
012021
011221
```

To test if the value of the counter is zero, we must test if all digit values are zero. To avoid testing all the digit values for 0, we need one more value which indicates that all more significant digits are 0. We call this value `E` for "Empty."

During operation, any digit with a value 0 will attempt to borrow a 1 from its more significant neighbor. A successful borrow from a more significant neighbor results in adding 2 to the digit's own value of 0, which results in a 2. An unsuccessful borrow results in changing the digit's own value from 0 to E. For example, the sequence `..10..` changes to `..02..`, the sequence `..20..` changes to `..12..` and the sequence `..E0..` changes to `..EE...` By default the most-significant digit is always `E`. With these definitions, it follows that whenever the least-significant digit's value is `E`, then the counter value is 0. Taking into account the above rules for transition, the following sequence of representations can occur during operation

```
E011221
EE11221
          -1, successful decrement
EE11220
EE11212
          -1, successful decrement
EE11211
          -1, successful decrement
EE11210
```

```
EE11202
          -1, successful decrement
EE11121
 .            after several
 .            successful decrements
EEEEEE1
          -1, successful decrement
EEEEEE0
EEEEEEE
          -1, unsuccessful decrement
EEEEEEE
```

We can summarize the transitions of two neighboring digit values, called MSN for "more-significant neighbor" and LSN for "less-significant neighbor" in table D.1. The table gives the neighboring digit values before and after the transitions in both digits. The transitions together are called the "firing."

D.2.2 Specification of the cells

For a loadable down counter, Kessels uses one cell per digit plus a special end cell. The communication behavior of each cell with its neighbors can be described with a finite state machine. The complete counter is then the parallel composition of all finite state machines.

The end cell communicates with just one neighbor, whereas all other cells communicate with two neighbors. We specify the behavior of one such cell as a sequence of communication actions between the user and the cell on the one hand and between the cell and the sub-counter on the other hand, as illustrated in Figure D.1. The communication actions between two neighboring cells appear in Table D.1. We use these actions as the communication actions between user and cell. We prefix the actions between the cell and the sub-counter with "`s.`" to obtain `s.dnU`, `s.dnS1`, and `s.dnS2`. Finally we introduce one more communication action representing the loading of the digit value into the cell: `load` and `s.load`.

Table D.1: Table of transitions of neighboring digit values and their associated state transition names

Before firing		*After firing*		Action
MSN	LSN	MSN	LSN	Name
E	0	E	E	dnU
1	0	0	2	dnS1
2	0	1	2	dnS2

Each cell can be in one of five states: `L`, `S0`, `S1`, `S2` and `FAIL`. State `L` is the initial state and load state, where the cell performs a load action with the user and then performs a load on the sub-counter in parallel with going to state `S0`, if the digit loaded is 0, or to state `S1`, if the digit loaded is 1. Note that the load action can initialize the sub-counter to state `S2`, if the digit loaded is 2. States `S0`, `S1`, and `S2` are the states of the cell where the value of the digit stored in the cell is 0, 1, or 2 respectively. In state `FAIL` the value of the digit stored in the cell is `E`.

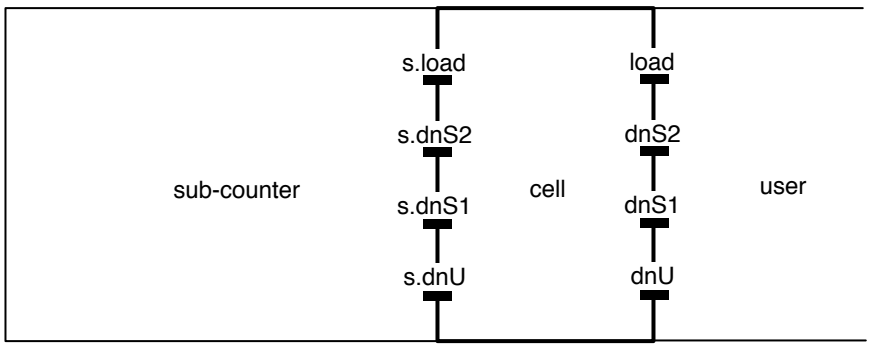


Fig. D.1: The counter consists of a cell communicating with a sub-counter and a user. Communication actions of the cell are placed at the boundary between the two communicators.

The specification of a cell appears in Figure D.2(a), where `bn` represents the digit value for that cell. In state `S2`, the cell performs a successful down action and goes to state `S1`. In state `S1`, the cell performs a successful down action and goes to state `S0`.

```
state L where
   L = ( load -> s.load,
                 if bn then S1
                       else S0 )
   S1 = ( dnS1 -> S0 )
   S2 = ( dnS2 -> S1 )
   S0 = ( s.dnS2 -> S2
        | s.dnS1 -> S2
        | s.dnU -> FAIL )
   FAIL = ( dnU -> L )
end
```

(a)

```
state L where
   L = ( load -> FAIL )
   S1 = ( dnS1 )
   S2 = ( dnS2 )
   FAIL = ( dnU -> L )
end
```

(b)

Fig. D.2: Specifications of the cells in Kessels's loadable down counter: (a) Specification of a cell; (b) Specification of end cell

In state `S0` the cell tries to borrow a bit from its sub-counter by performing a down action on the sub-counter. If the cell and sub-counter perform a successful down action, the cell goes to state `S2`. If the cell and sub-counter perform an unsuccessful down action, the cell goes to state `FAIL`. In state `FAIL`, the cell performs an unsuccessful down action with the user and then goes to state `L`, waiting for the next load action.

A specification for the end cell appears in Figure D.2(b). The specification includes all actions `load`, `dnS1`, `dnS2`, and `dnU`. The actions `dnS1` and `dnS2`, however, never occur, because the end cell always alternates between state `load` and `FAIL`.

D.2.3 Mapping a finite state machine to a GasP implementation

Mapping a finite-state-machine to a GasP implementation is straightforward: map every event to a GasP module and map every state to a connection between GasP modules. Figure D.3(a) shows the implementation of the simple state transition

```
P = ( a -> Q )
```

Let us call state P an *input state* for event a and state Q an *output state* for event a. Notice that input state P maps to a connection that is a self-resetting input to GasP module a, and output state Q maps to a connection that is an output of GasP module a. Module a implements an occurrence of event a by setting output state Q and resetting input state P.

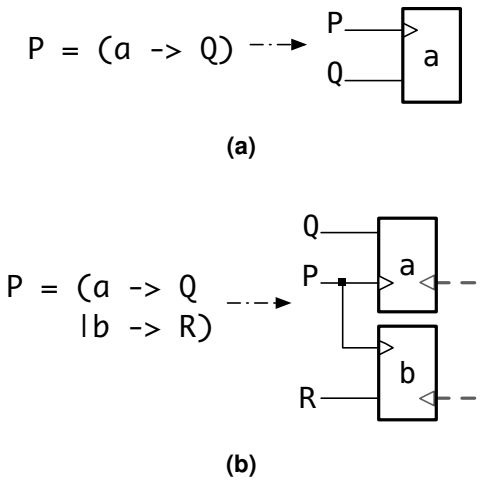


Fig. D.3: Mapping state transitions to GasP modules: (a) a simple state transition; (b) a state transition with choice

The implementation of the choice

```
P = ( a -> Q
    | b -> R )
```

appears in Figure D.3(b). Here state P is an input state of two state transitions. When input state P is set, both GasP modules a and b may fire. Because this is a deterministic choice, however, either module a or module b fires, and the environment chooses which one fires. The environment determines which module fires by setting either the grey input port of module a or the grey input port of module b. The input ports are in grey, because they are part of a neighbor finite state machine. When module a or b fires, it resets state P and sets either state Q or R respectively. Notice that when module a or b resets state P, it prevents the other module from firing.

In the translation we also must decide for each state what the time separation is between the enter and exit events for that state. The assignment of labels 2 or 4 to a state determines whether the time separation between any entry and exit event for that state is two or four gate delays. The easiest delay assignment would be to assign a delay of 2 units to each state. This would yield the fastest implementation. Unfortunately, such an assignment is not always possible. A delay assignment must satisfy the condition that each cycle lasts at least six gate delays for 2-4 GasP and ten gate delays for 4-6 GasP.

Besides being simple, this translation has another attractive property. While one state can be an input state of multiple events and an output state of multiple events, the state itself can be implemented by a single wire connection, possibly long and with multiple forks.

D.2.4 One-hot implementation of the counter

Figure D.4 shows the GasP implementation of the counter cells, using the one-hot encoding of the previous section. Note that each GasP module is part of two neighboring finite state machines. Thus, a GasP module fires only when both finite state machines can engage in the state transition implemented by the GasP module.

The complete implementation of the loadable down counter is the parallel composition of the cells or simply the superposition, or "AND," of all finite-state-machine implementations of the cells. Figure D.5 gives a complete implementation of a 2-bit loadable down counter using the one-hot state encoding.

D.3 Down-one-two counter

This section describes the design of a "test-after-decrement" down-one-two counter. The down-one-two counter can be loaded with an initial value of *N*, where $N > 0$. We

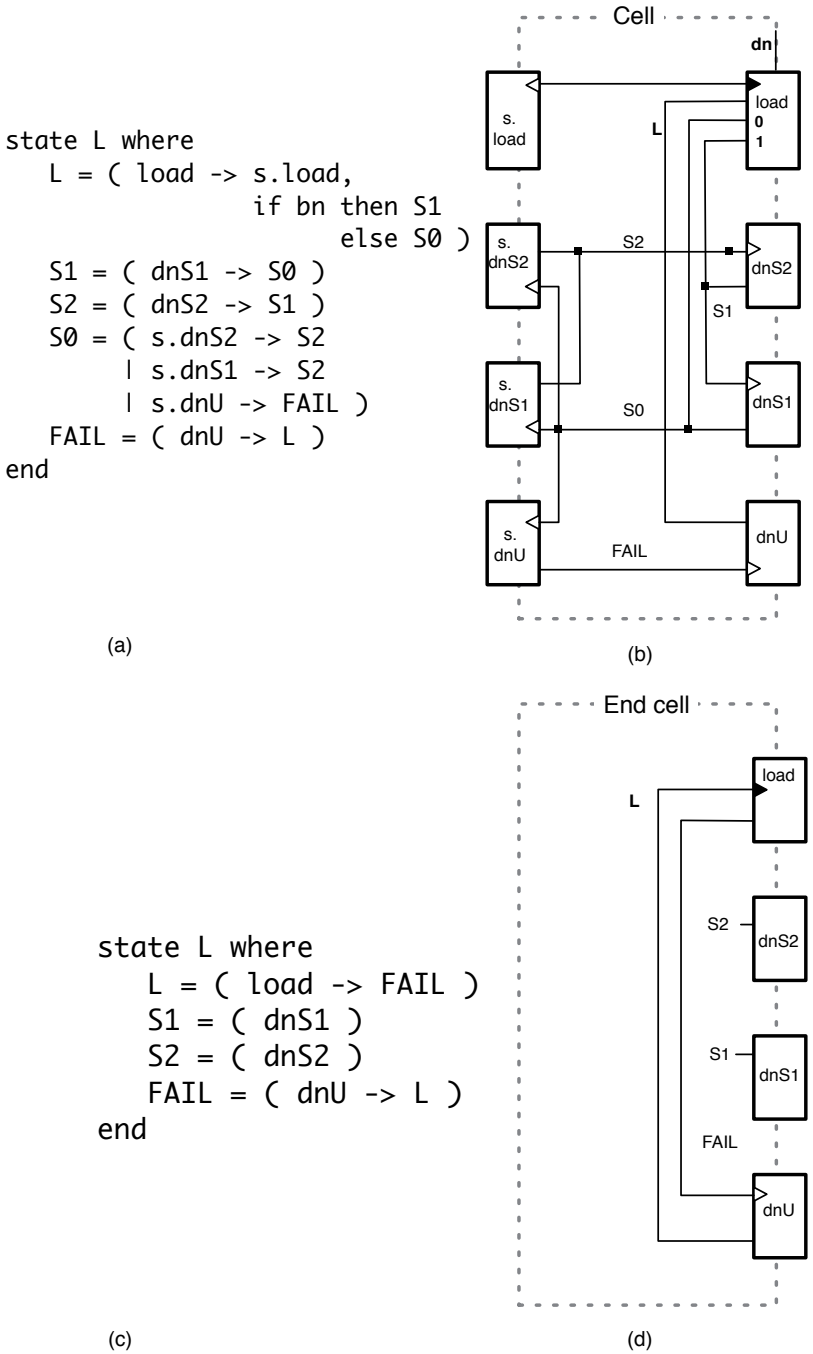


Fig. D.4: Implementations for Kessels's loadable down counter: (a) Specification of a cell; (b) implementation of a cell; (c) specification of end cell; and (d) implementation of end cell

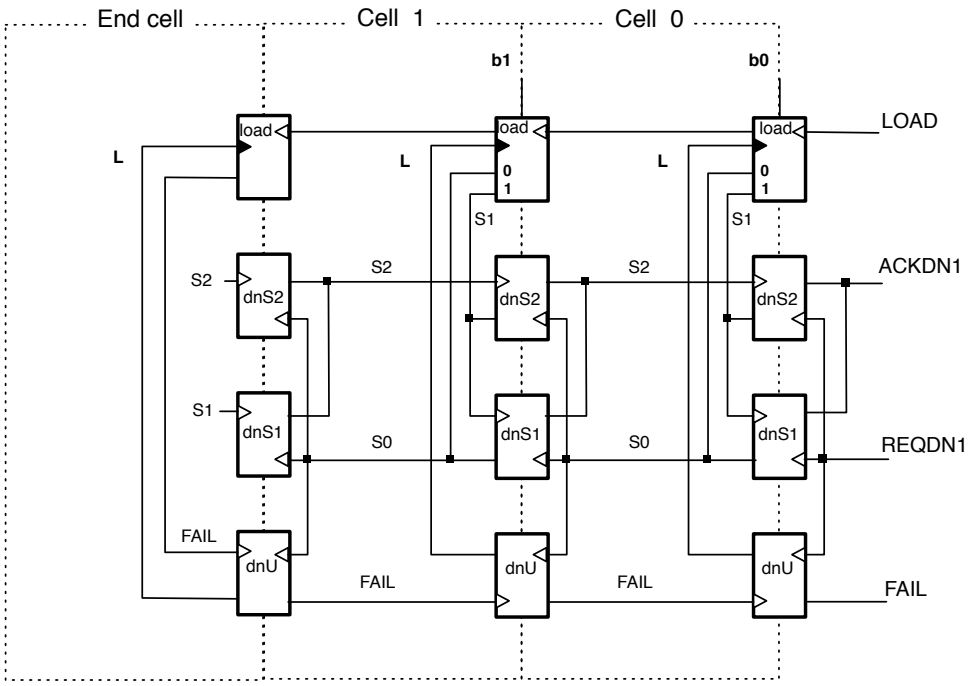


Fig. D.5: An implementation of a "test-before-decrement" 2-bit loadable down-one counter

first specify the user of the down-one-two counter as a finite state machine, exposing only the states that are important for the communication behavior: `LOAD`, `REQDN1`, `REQDN2`, `EMPTY`, and `NOTEMPTY`. We represent the value of the down counter by the variable n.

```
USER = LOAD where
  LOAD  = ( load(n:=N) -> ( REQDN1 | REQDN2 ))
  REQDN1 = ( n:=n-1;
             if n>0 -> NOTEMPTY
             |  n<=0 -> EMPTY
             endif )
  REQDN2 = ( n:=n-2;
             if n>0 -> NOTEMPTY
             |  n<=0 -> EMPTY
             endif )
  EMPTY  = ( <user actions> -> LOAD )
  NOTEMPTY = ( <user actions> -> (REQDN1 | REQDN2) )
end
```

Initially the finite state machine USER starts in state LOAD in which the counter is being loaded with its initial value N. Then the user goes to state REQDN1 or REQDN2 depending on whether a decrement by one or two is requested respectively. The counter responds with transitioning the user to state NOTEMPTY or EMPTY, depending on the value of the counter. In state EMPTY, the user performs some actions and then may transition to state LOAD again. In state NOTEMPTY, the user performs some actions and then may transition to state REQDN1 or REQDN2 again.

D.3.1 Implementation of down-one-two counter for $N > 0$

We can make a “test-after-decrement" down-one-two counter for $N > 0$ from a sub-counter and some other cells. The sub-counter is a simple “test-before-decrement" down-one counter, described in Section D.1. The other cells implement finite state machines that ensure that the composite behaves as a “test-after-decrement" down-one-two counter. We use two extra cells, called HEAD0 and HEAD1. Our implementation is in principle similar to Figure D.5, where the end cell is replaced by the down-one counter, Cell 1 is replaced by HEAD1, and Cell 0 is replaced by HEAD0. Because the “test-after-decrement" down-one-two counter is more complicated than a simple “test-before-decrement" down-one counter, the cell HEAD0 has more states and more communication actions than Cell 0 from Figure D.5.

In order to implement a down-one-two counter with constant response time, the HEAD0 cell must always have a count at least 2, unless the sub-counter and HEAD1 are empty. In our implementation HEAD0 cell has a count value of $n0$, with $0 \leq n0 \leq 4$, and HEAD1 cell has a count value of $n1$, with $0 \leq n1 \leq 2$. The value of the complete counter at any moment is $n = 4 * s + 2 * n1 + n0$, where s is the count of the sub counter, $n1$ is the count of HEAD1, and $n0$ is the count of HEAD0. Thus upon initialization we have $N = 4 * s + 2 * n1 + n0$.

Cell HEAD0 has 7 states: N4, N3, N2, N1, E2, E1, and E0. The states la-

beled with E indicate that the sub-counter and HEAD1 are empty. The states labeled with N indicate that the sub-counter or HEAD1 may have a nonzero count value. Furthermore, the digit value in each state's name indicates the count value of the HEAD0 cell. Thus, in state N2 the HEAD0 cell stores a count of 2.

We introduce two communication actions between HEAD0 and HEAD1 to represent a successful decrement and a failed decrement: s.dn1 and s.fail respectively. The specification of HEAD0 is as follows.

```
HEAD0  = E0 where
    E0 = ( load -> (N4 | N3 | E2 | E1) )
    N4 = ( dn1 -> N3
         | dn2 -> N2 )
    N3 = ( dn1 -> N2
         | dn2 -> N1 )
    N2 = ( s.dn1  -> N4
         | s.fail -> E2 )
    N1 = ( s.dn1  -> N3
         | s.fail -> E1 )
    E2 = ( dn1 -> E1
         | dn2 -> E0 )
    E1 = ( dn1 -> E0
         | dn2 -> E0 )
end
```

Cell HEAD1 communicates between cell HEAD0 and a down-one sub-counter. Specification of cell HEAD1 is as follows. The communication actions s.dn1 and s.fail represent communication actions between HEAD0 and HEAD1. The communication actions t.dn1 and t.dnU represent communication actions between HEAD1 and the down-one counter, where t.dn1 is a successful decrement and t.dnU is an unsuccessful decrement.

```
HEAD1 = LOAD where
    LOAD = ( s.load -> (S1 | S0 )
    S2   = ( s.dn1 -> S1 )
    S1   = ( s.dn1 -> S0
```

```
    S0   = ( t.dn1 -> S2
           | t.dnU -> FAIL )
    FAIL = ( s.fail -> LOAD )
end
```

Each state name indicates the current count value of cell HEAD1: S2 represents count value 2, S1 represents count value 1, S0 and FAIL represent count value 0. State FAIL also indicates that the sub-counter is empty. Note that state S0 corresponds to state REQDN1 for the down-one counter, which either responds with a transition to state FAIL after an unsuccessful decrement t.dnU or a transition to state ACKDN1 after a successful decrement t.dn1. State ACKDN1 for the down-1 counter is state S2 for HEAD1.

Figure D.1 shows the complete implementation using cells HEAD0 and HEAD1. Each communication action s.dn1 and s.fail between HEAD0 and HEAD1 occurs if and only if both HEAD0 and HEAD1 agree on the next action. For example, when HEAD0 is in state N2 and HEAD1 is in state S1, the only action that can occur is s.dn1S1N2, where s.dn1S1N2 denotes the action s.dn1 in state S1 for cell HEAD1 and state N2 for cell HEAD0.

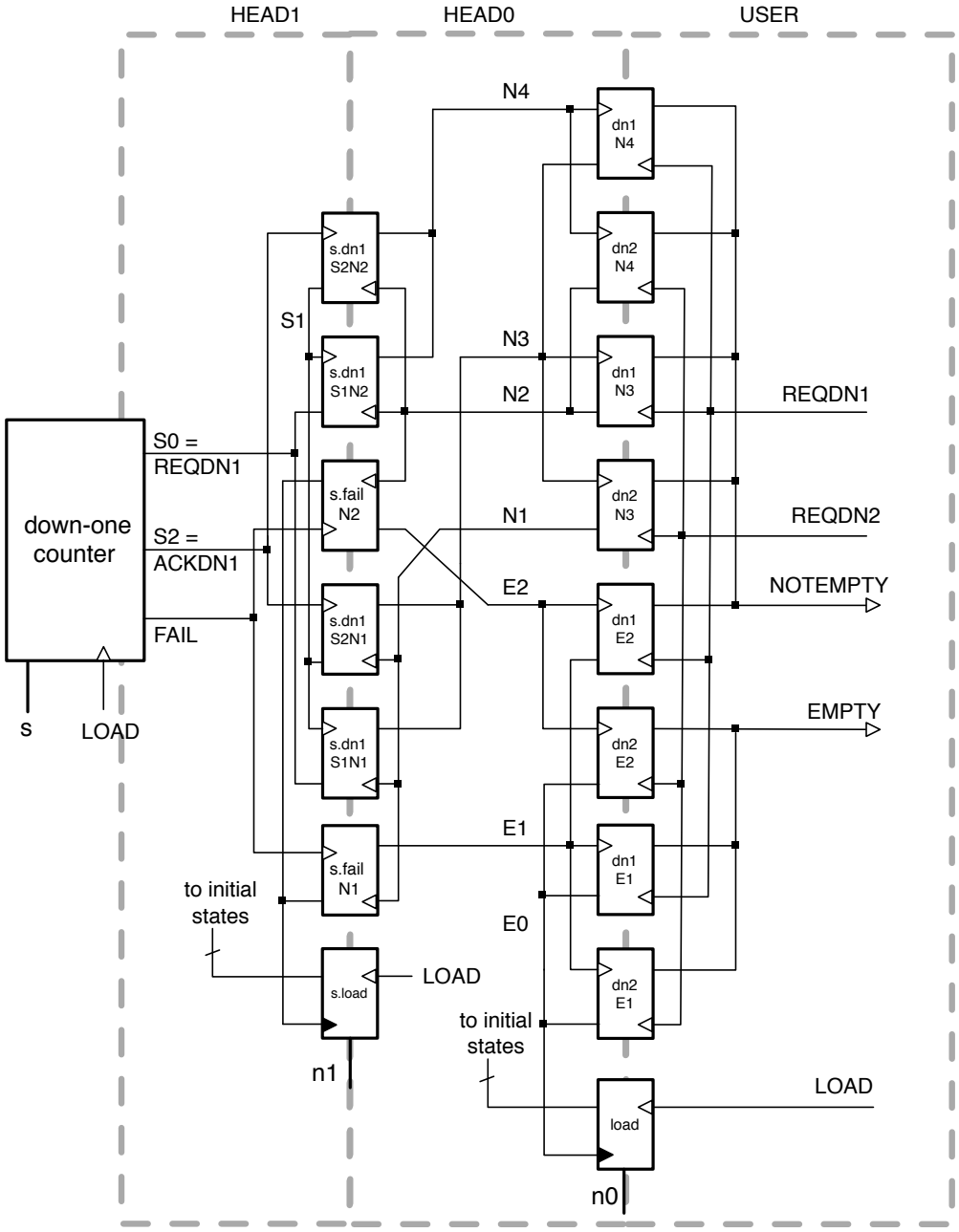


Fig. D.1: Implementation of a down-one-two counter using cells `HEAD0` and `HEAD1`