Anurag Pallaprolu
256733/T002967
Practice School Intern
STMicroelectronics, Greater Noida

An Excursion in

Digital Logic

via a detour to Asynchronous Design

: deviation from a direct, definite, or proper course

ex/cur/sion, definition from Merriam-Webster.

: Controlling the timing of operations by the use of pulses sent when the previous operation is completed rather than at regular intervals.

a/synch/ro/nous, definition from The Oxford Online Dictionary.

Chapter 1

Prologue

The computer has come a long way since its humble roots nearly half a century ago. Sizes have been reducing consistently with respect to Gordon Moore's observation and things are becoming denser. Power consumption has also scaled similarly and the machines which once took whole rooms to solve a differential equation now need only a handful of space to plot all possible solutions of the same. However, the single most important facet of electronic design that has led to such an explosion of development in both architecture and programmability is the availability of a central 'clock', an independently pulsing signal generated by an on-chip crystal, whose rising edges signify timestamps at which processes scheduled for execution are fired. Central synchronization greatly simplifies programming as one can now say with some confidence that some instruction A is executed and *then* some other instruction B is executed. It also gives us the illusion that (at low frequencies) one can concurrently execute instructions (although this is prohibited by relativistic physics).

We are currently running these clocks at frequencies in the GHz range, and this implies that the computer has about a few nanoseconds of time to react to a given instruction before proceeding to the next edge. When clocks were slow the delays that were added while propagating the signal to different subsystems of the computer were small enough to not show any impact on the performance but the delays now lead to significantly different 'clock' signals to be transmitted to spatially separated electronic blocks. This causes a major issue in data transmission across the computer and sometimes may even cause fatal system failures such as metastable oscillations which even-

tually increase the internal temperature to a point of combustion. The only solution to such a pressing concern involves removing the central clock altogether, and to allow the blocks to communicate with each other via the protocol of handshaking. This forces the architecture to be pipelined and thus also shows an innate improvement in the throughput. Removing the clock also saves a major chunk of the power which was being used to drive the clock lines all over the computer before. However, it turns out that asynchronous designs occupy more area on an average but the trade-off is justified when compared to the amount of stability that is assured. The reader might be wondering as to why he/she hasn't seen an asynchronous processor floating around, and the answer is quite obvious. Shifting from the synchronous paradigm involves giving up all the benefits of the synchronous domain, and sequential execution of instructions is something that was done quite well by clocked state machines. In an asynchronous processor even an erroneous signal generated during the execution of an expected signal is executed concurrently due to the lack of sequencing, and thus it becomes extremely hard to write programs that protect themselves from these external glitches. The other reason as to why this workflow hasn't picked up is due to the poor documentation and support for such a design process, and this is not surprising for the theoretical barrier set for fully understanding the asynchronous fabric is far higher than that required for clocked designs. In addendum to this, there are very few open source simulators and almost no commercial libraries available for constructing a circuit via some standardized procedure.

This report contains my study of the delay insensitive asynchronous fabric which I carried out at STMicroelectronics, Greater Noida. It deals with the theory of asynchronous logic presented as chronological notes, followed by three realistic case studies of the concepts developed. The motivation was to learn something new without resorting to formalization of the learning process, that is, to learn by trial. Many notebooks were done away with scribbles and sometimes extremely silly circuit ideas, some of which are included in this report. I hope this 'excursion' will be fun and worthwhile for the reader, it surely was for me.

A. Pallaprolu
12th June, 2017
Noida, India.

Contents

Acknowledgements

I would like to thank my mentors Alok Tripathi, Swati Narang and Namerita Khanna, STMicroelectronics, for seamlessly guiding me through the project by taking time out of their work schedules. They had created an atmosphere which motivated self-discovery and open minded design, and I would never have come across the wonderful branch of asynchronous digital logic if it wasn't for them.

I would like to thank the Practice School Division, BITS Pilani, for providing me with the opportunity to explore such hidden concepts in the theory of electronic engineering from an industry perspective. It has been an interesting exercise to work at STMicroelectronics, and I can confidently say that I have benefitted positively from my limited tenure here. I would like to thank Prof. R. K. Tiwary, PS Faculty, BITS Pilani, for handling our transition from an academia based perspective to that of the company.

A special thanks to Pavan Kulkarni, STMicroelectronics, for designing the layout for the Wake Up Flop and to Srinivas R., BITS Pilani, Hyderabad Campus, for reviewing this work at such short notice.

I would finally like to thank my parents for supporting every professional endeavor of mine so far.

Chapter 2

The Basics

'You can invent things like automatic popcorn poppers.
You can invent things like steam-powered window washers.
But you can't invent more time.'

Lemony Snicket, A Series Of Unfortunate Events,
Book 1 - The Bad Beginning.

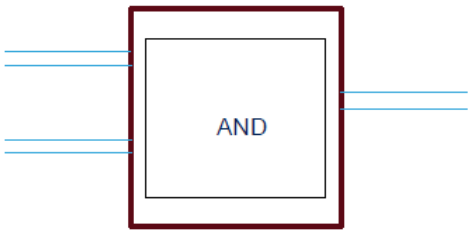
2.1 Synthesis of Asynchronous Circuits

Dated : 18th January, 2017

The synthesis of asynchronous circuit blocks depends upon the communication protocol chosen for the fabric. For the four phase dual rail protocol one has two lines (labeled as true line and false line) dedicated to each input and the logic level for each input is decided in four phases.

False	True	Output
0	0	"EMPTY" or "SPACER"
0	1	1
1	0	0
1	1	ERROR/RACE HAZARD

Since the output is determined as a fixed logic level, the synthesis of asynchronous four phase dual rail circuits is essentially solving two Karnaugh map reductions for the true and the false output rails. For instance, the four phase dual rail AND gate block shown below



If I call the two inputs as A (`A.t, A.f`) and B (`B.t, B.f`) and the two outputs as C (`C.t, C.f`) then the simple behavioral model of the AND gate can be translated into the following truth table

A.t	A.f	B.t	B.f	C.t	C.f
0	1	0	1	0	1
0	1	1	0	0	1
1	0	0	1	0	1
1	0	1	0	1	0

The circuit can be then made using traditional logic blocks, in this case,

$$C.t = A.t \& B.t$$

$$C.f = A.f|B.f$$

The two phase dual rail protocol involves two rails per port and the logic levels are decided based on the occurrence of an edge or a transition on the corresponding rail than an absolute level held by either the true or the false line. That is, if there is a transition on the true line of a port and the false line continues to hold a level then the pin is said to be logic level high and so on. As in the previous case, a simultaneous edge/transition on both the true and false lines of a port might lead to errors or race conditions. After going through the TIEMPO ASC 28FDSOI Reference Guide (Version 2.0, October 2014), it was noted that the devices that were constructed in the library used the two phase dual rail protocol and the functional description shows that the output ports have almost always been represented as a combination of a set and a reset signal associated with the port. That is, if we are considering a two input asynchronous AND gate being implemented in the two phase protocol and if the outputs are labeled as Z0 and Z1 for the false and true lines respectively (this convention is followed henceforth for other ports also), then,

$$Z0 = S0 + R0.Z0$$

$$Z1 = S1 + R1.Z1$$

Where S0, S1 are the set signals for Z0, Z1 and R0, R1 are the respective reset signals. The structure of the output port is inspired from the fundamental block of the asynchronous circuit toolchain, the **Muller C element** and the names set and reset would make more sense if the Z0, Z1 on the RHS are considered to be the values of Z0 and Z1 in the past (it would be better to call them Z0- and Z1-). In this document three devices would be synthesized under the two phase protocol restriction: the AND gate, the XOR gate and the one bit Latch. The idea is to assume that whenever there is a transition on either the true or the false lines the variable takes a Boolean high. Thus, the behavioral truth table for the AND device is given by the same table on the previous page. It can be seen from the two equations above that whenever Z0 = 0, S0 has to be 0 and whenever Z1 = 0, S1 has to be 0. Drawing the extended truth table

A0	A1	B0	B1	Z0	Z1	S0	S1
1	0	1	0	1	0	?	0
1	0	0	1	1	0	?	0
0	1	1	0	1	0	?	0
0	1	0	1	0	1	0	?

If one notices the last column i.e. S1, one can see that if the last entry is 0 the set pin for Z1 output is a trivial ground and this would leave us with

$$Z1 = R1.Z1$$

This would mean that R1 had to be identical to Z1 and this defeats the purpose of the transformation. The only alternative left is to pick the last entry as 1, and using the same logic one can also see that the first three entries in the S0 column turn out to be 1. (It might look like Z0 = S0, and Z1 = S1 are trivial solutions, but then again the transformation is pointless if you only end up changing labels). Using the sum-of-minterms idea, one can easily see that

$$S0 = A0.B0 + A0.B1 + A1.B0$$

$$S1 = A1.B1$$

Plugging this into the fundamental equations, we have

$$Z0 = (A0.B0 + A0.B1 + A1.B0) + R0.Z0$$

$$Z1 = (A1.B1) + R1.Z1$$

To figure out how R0 and R1 look, we plot another truth table, this time around it will strictly involve the Boolean variables which occur in the corresponding set pins. For R0, we have to consider all four literals available (since S0 includes three out of the four)

A0	B0	A1	B1	Z0	R0
1	1	0	0	1	X
1	0	0	1	1	Y
0	1	1	0	1	Z
0	0	1	1	0	W

Using the updated fundamental equation, we have the following equations for X, Y, Z, W

$$1 = 1 + X.1$$

$$1 = 1 + Y.1$$

$$1 = 1 + Z.1$$

$$0 = 0 + W.0$$

Taking the sum-of-minterms again, we have

$$R0 = A0.B0 + A0.B1 + A1.B0 + A1.B1 = A0 + A1 + B0 + B1$$

The second equality can be proven with simple algebra. Similarly, noticing that S1 contains only A1 and B1 as Boolean variables in its functional description it would be logical in assuming that R1 too depends only on A1 and B1. Drawing the truth table, we have

A1	B1	Z1	R1
0	0	0	X′
0	1	0	Y′
1	0	0	Z′
1	1	1	W′

Using the updated fundamental equation, we have the following equations for X, Y, Z, W

$$0 = 0 + X.0$$

$$0 = 0 + Y.0$$

$$0 = 0 + Z.0$$

$$1 = 1 + W.1$$

Taking the sum-of-minterms, we have R1 = A1 + B1 + A1.B1 = A1 + B1 The same analysis can be extended to synthesize the 2 input two phase dual rail XOR gate in asynchronous fabric. Again, assuming that the lines with an occurrence of a transition to be taken as a Boolean high, one can do the set and reset construction again to obtain the following extended table for the set pins exclusively.

A0	A1	B0	B1	Z0	Z1	S0	S1
1	0	1	0	1	0	1	0
0	1	1	0	0	1	0	1
1	0	0	1	0	1	0	1
0	1	0	1	1	0	1	0

$$S0 = A0.B0 + A1.B1$$

$$S1 = A1.B0 + A0.B1$$

The reset pins are then figured out by plugging these expressions into the fundamental equations, thereby giving us

$$Z0 = (A0.B0 + A1.B1) + R0.Z0$$

$$Z1 = (A1.B0 + A0.B1) + R1.Z1$$

It can be seen that for both the reset pins, all four Boolean variables have to be involved in the truth table. Doing so for R0, we have

A0	A1	B0	B1	Z0	R0
1	0	1	0	1	X
0	1	1	0	0	Y
1	0	0	1	0	Z
0	1	0	1	1	W

Running the same through the fundamental expression, we get the following, 1 = 1 + X.1 0 = 0 + Y.0 0 = 0 + Z.0 1 = 1 + W.1 This implies that the sum-of-minterms expression for R0 is R0 = A0.B0 + A1.B0 + A0.B1 + A1.B1 = A0 + B0 + A1 + B1 It turns out that R1 = R0 and the same procedure can be used to verify this claim. The final example is that of a single input dual rail two phase protocol asynchronous latch with external control. When the control is low, the latch is inactive and does not latch onto the input presented. The behavioral truth table of the asynchronous latch is shown below

A0	A1	C	Z0	Z1
0	1	0	0	0
1	0	0	0	0
0	1	1	0	1
1	0	1	1	0

It can be seen that the following expressions hold for the set pins (as they are supposed to mimic the Z behavior)

$$S0 = C.A0$$

$$S1 = C.A1$$

The reset pin functional description can then be obtained by following the usual route. First, substitute S0 in the fundamental expression

$$Z0 = C.A0 + R0.Z0$$

Noticing that C and A0 are the only literals used in S0, one can restrict the domain to search for a functional description of R0 in the same domain, thereby giving the following table.

A0	C	Z0	R0
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	1

This gives the following expression for R0 and R1:

$$R0 = A0 + C + C.A0 = C + A0$$

$$R1 = A1 + C + C.A1 = C + A1$$

This concludes the demonstration on synthesizing asynchronous circuits on both the transfer protocols.

2.2 A Workflow for Synthesizing Asynchronous Circuits

Dated 19th January, 2017

This document deals with an attempt at a procedure to synthesize standard blocks using the two phase dual rail asynchronous fabric introduced earlier. Due to logical consistency issues, the previous approach was viable only for small combinational blocks and did not provide much correlation to the calculated results and the results provided in the Tiempo ASC28FDS0I manual. It is hoped that this procedure fills in the gap left by the previous one and clarifies the actual intent behind the set and reset pins used for every output. Their meaning follows from their actual colloquial definitions, in the sense, they do set the output to 1 and reset the output to 0 when asserted. This fact will be undercurrent in the rest of the discussion. Let us work with the humble AND gate once again and work out the equations for the output ports. Drawing the truth table from the behavioral model we have

A0	A1	B0	B1	Z0	Z1
0	0	0	0	0	0 (A)
1	0	1	0	1	0 (B)
1	0	0	1	1	0 (C)
0	1	1	0	1	0 (D)
0	1	0	1	0	1 (E)

Notice that there is a spacer row added at the beginning to signify the initialization state and this will be taken as a reference state (A). Now the idea for the port Z0 to have set and reset components is to change its state from its previous state, if it was holding a 0, the set action changes it to 1 and the reset action guards it at 0. Thus, S0, the set pin for the Z0 port will be asserted whenever Z0 changes to logic 1 from the reference state (A). There are three such transitions which come under this class, namely, (A→ B), (A→C) and (A→D). These are mutually exclusive and can happen in any order. This leads us to conclude that whenever these transitions are independently asserted the Z0 pin goes high. That is,

$$S0 = (A \rightarrow B) + (A \rightarrow C) + (A \rightarrow D) = A0.B0 + A0.B1 + A1.B0$$

Similarly, it can be seen that S1 = A1.B1.

We shall return to the reset pins in a moment but it is to be understood that this procedure for computing the set pins Boolean expression would always work because we are considering valid transitions in the so-called asynchronous finite state machine of the AND gate. That is, one could in general construct an automaton like structure out of the truth table by considering the changes in the output ports to be transitions between states. This idea will be central to figuring out the structure of the reset pins. For the reset pin structure of Z0, we go back to the truth table to note down all the transitions that lead to a reset i.e. a 1→0 transition of the Z0 output. These transitions can also include 0→0 types as they can be considered to be a variant of a reset (you could call it a strong reset). Accounting for all the possibilities it can be seen that

$$R0 = (B \rightarrow E) + (C \rightarrow E) + (D \rightarrow E) + (E \rightarrow A)$$

$$\implies R0 = (\tilde{A}0.A1) + (\tilde{B}0.B1) + (\tilde{A}0.A1.\tilde{B}0.B1) + (\tilde{A}1.\tilde{B}1)$$

$$\implies R0 = (\tilde{A}0.A1) + (\tilde{B}0.B1) + (\tilde{A}1.\tilde{B}1)$$

This can be verified to be isomorphic to the expression provided in the manual by plugging in the values of the Boolean variables for the given domain. One can see, either by manually constructing the truth table by hand or by using a computer tool like CleanLogic to see that they are the same for all the input variations in the present domain. (They might not be Boolean identical but for the restricted choice of inputs they are equivalent, hence isomorphic.) A similar analysis can be done to construct R1. A query could be raised as to why (B→A) and so on were not considered when (E→A) was considered and the answer to this involves the idea of the asynchronous block being a finite state automaton, in which one has to go through the latter state to arrive at the former transition thereby counting it into the summation anyway. As a final example, the synthesis of a full adder block is demonstrated along with the steps taken to derive the set and the reset pins of the S0 and S1 ports of the sum output.

A0	A1	B0	B1	CI0	CI1	S0	S1	CO0	CO1
0	0	0	0	0	0	0	0	0	0 (A)
1	0	1	0	1	0	1	0	1	0 (B)
1	0	1	0	0	1	0	1	1	0 (C)
1	0	0	1	1	0	0	1	1	0 (D)
1	0	0	1	0	1	1	0	0	1 (E)
0	1	1	0	1	0	0	1	1	0 (F)
0	1	1	0	0	1	1	0	0	1 (G)
0	1	0	1	1	0	1	0	0	1 (H)
0	1	0	1	0	1	0	1	0	1 (I)

For the set pin of S0, say SetS0, we need to note all the transitions from the initialization state (A) which lead to setting the S0 pin high, namely

$$SetS0 = (A \to B) + (A \to E) + (A \to G) + (A \to H)$$

$$\implies SetS0 = A0.B0.CI0 + A0.B1.CI1 + A1.B0.CI1 + A1.B1.CI0$$

The reset pin for the S0 output needs a bit more work. One needs to see all possible soft and hard reset transitions and sum them all together. Scanning through the truth table, with sufficient focus, it can be noted that (the redundant transitions are written anyway for they do not contribute any new terms to the expression, for instance, (G→A) is redundant for it is counted for by (G→I) and (I→A).)

$$ResetS0 = (B \to C) + (B \to D) + (C \to D) + (E \to F) + (H \to I) +$$

$$(H \to A) + (G \to I) + (G \to A) + (I \to A)$$

Combining these terms together and simplifying the residual Boolean expression, one ends up with the following function

$$ResetS0 = \tilde{CI}0.CI1 + \tilde{B}0.B1 + \tilde{A}0.A1 + \tilde{A}1[\tilde{B}1.\tilde{CI}0 + \tilde{B}0.\tilde{CI}1 + \tilde{B}1.\tilde{CI}1]$$

Using either CleanLogic or manually verifying, it can be seen that this expression too is isomorphic to the one given in the manual for the domain of variables chosen here. The reason as to why this procedure works can almost always be related to the restricted domain of the literals involved as the above expression does not give the same outputs for all possible combinations of the six tuple (A0, A1, B0, B1, CI0, CI1) but does give the same outputs for the tuples presented in the truth table. Similarly, for the reset pin for S1 one can see that

$$ResetS1 = (C \to E) + (D \to E) + (F \to G) + (F \to H) + (G \to H) +$$

$$(I \to A)+(I \to B)+(A \to B) = \tilde{B}0.B1+C\tilde{I}0.CI1+\tilde{A}1.\tilde{B}1.C\tilde{I}1+A0.B0.C0$$

Once again, this turns out to be isomorphic to the actual expression in the restricted domain of the truth table. This concludes the demonstration.

2.3 Muller Circuit Synthesis from Truth Tables

Dated 20th January, 2017

In the previous two documents basic circuit block synthesis was discussed in the asynchronous fabric using the two phase dual rail communication protocol. The first document had a few loopholes which were sealed by the second one, but still there was an air of dissatisfaction with the author regarding the workflow for constructing asynchronous circuitry. It is with the motivation to clarify things that the present document is written. The previous documents dealt with the synthesis keeping in mind the set-reset paradigm of design. That is, every output port is assumed to be driven by a C gate instead of being directly driven to the output (which happens in single phase synchronous circuits) and the two components of the C gate output were constructed from the truth table. The second document concluded the discussion in a flavor similar to a discussion on finite state automata.

However, David Muller's seminal work [1] holds most of the basic information to construct delay-insensitive speed-independent circuits (which are called Muller circuits in todays colloquia) which use bundled multi-phase communication protocols to construct asynchronous logic. The current document is almost an exposition of [1]. Unlike traditional synchronous designs (combinational or sequential), the basic logic blocks in Muller circuits are centered on the Muller C element/gate. The n-input generalized Muller C gate can be defined functionally as follows:

$$F(A_1, A_2, A_3, , A_n) = 1 \; if all A_i = 1$$

$$F(A_1, A_2, A_3, , A_n) = 0 \; if all A_i = 0$$

$$F(A_1, A_2, A_3, , A_n) = F(A_1, A_2, A_3, , A_n)(t-) \; otherwise$$

Where $F(A_1, A_2, A_3, , A_n)(t-)$ indicates the value latched by the Muller gate in the previous iteration of input data entering from the other end of the pipeline. Mullers thesis is that even the combinational blocks such as AND, XOR et cetera involve the usage of the C element as a basic element and hence involves the concept of feedback so that the block can retain information instead of being passive. Let us discuss the synthesis of an AND gate in the Muller style of design. This involves recalling the same old truth table

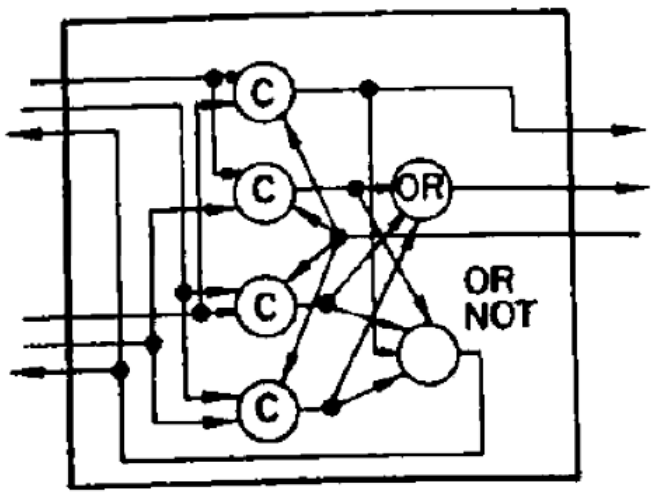
A0	A1	B0	B1	Z0	Z1
0	0	0	0	0	0 (A)
1	0	1	0	1	0 (B)
1	0	0	1	1	0 (C)
0	1	1	0	1	0 (D)
0	1	0	1	0	1 (E)

The crucial point in designing Muller circuits is that we need to replace the AND operation (.) with the Muller gate operation (*). On two inputs this implies that instead of the output of $Z = X.Y$ being $X.Y$ it is now $Z = X.Y+(X+Y)Z$. This minor modification is sufficient to construct a fully asynchronous design in Muller style. Writing the modified SOP canonical form for Z0 and Z1, we have:

$$Z0 = A0 * B0 + A0 * B1 + A1 * B0$$

$$Z1 = A1 * B1 = A1.B1 + (A1 + B1)Z1$$

Referring to [1], the full block diagram for an AND gate is shown with the acknowledge line bundled with the request and data lines (the lines which we call A0, A1 and so on).



The procedure to synthesize the acknowledge section for the pipeline is also explained and is rather very simple. If the output lines are transmitting bits

from left to right, they are summed using an OR gate just like constructing the modified canonical SOP form. On the other hand, if the bits are being transmitted opposite to the generic flow in the pipeline then they are summed using a NOR gate instead. As another conclusive example repeated in the previous document, the full adder design becomes even more simplified in the Muller workflow. Repeating the truth table,

A0	A1	B0	B1	CI0	CI1	S0	S1	CO0	CO1
0	0	0	0	0	0	0	0	0	0 (A)
1	0	1	0	1	0	1	0	1	0 (B)
1	0	1	0	0	1	0	1	1	0 (C)
1	0	0	1	1	0	0	1	1	0 (D)
1	0	0	1	0	1	1	0	0	1 (E)
0	1	1	0	1	0	0	1	1	0 (F)
0	1	1	0	0	1	1	0	0	1 (G)
0	1	0	1	1	0	1	0	0	1 (H)
0	1	0	1	0	1	0	1	0	1 (I)

(The alphabets in the parenthesis are for state labeling and can be ignored for the present discussion). Writing the modified SOP canonical Boolean expressions for each of the outputs (it is stressed that * is Muller gate operation and not the AND operation),

$$S0 = M0 * N0 * CI0 + M1 * N1 * CI0 + M1 * N0 * CI1 + M0 * N1 * CI1$$

$$= (M0 * N0 + M1 * N1) * CI0 + (M1 * N0 + M0 * N1) * CI1$$

$$S1 = (M0 * N0 + M1 * N1) * CI1 + (M1 * N0 + M0 * N1) * CI0$$

$$CO0 = M0 * N0 * (CI0 + CI1) + (M0 * N1 + M1 * N0) * CI0 =$$

$$M0*N0+(M0*N1+M1*N0)*CI0CO1 = M1*N1+(M0*N1+M1*N0)*CI1$$

These expressions can be further optimized with Boolean evaluation to create compact designs with lesser transistor count but we have finally hit a definitive workflow to synthesize circuitry in the asynchronous fabric (the above expressions are verified by [4]). These circuits are called Muller circuits as opposed to Huffman circuits which are in turn synthesized using an altogether different data structure known as the Huffman flow table [2]. The Muller circuit philosophy is closely tied to the idea of representing these circuits using event driven automata and it has its mathematical basis in the theory of communicating sequential processes [3].

2.3.1 References

1 Muller, D.E., Asynchronous Logics and Application to Information Processing, Univ. of Illinois.

2 Myers, C.J., Asynchronous Design, Wiley Interscience, 2001.

3 Hoare, C.A.R., Communicating Sequential Processes, Prentice Hall International, 1985

4 Christmann, J.F., Prog A Asynchronous Design.

2.4 A Note On The Set-Reset Representation

Dated 23rd January, 2017

The present document is written to clarify the long standing issue with trying to coherently put together the previous documents titled A Workflow for Synthesizing Asynchronous Circuits [4] and Muller Circuit Synthesis from Truth Tables [5] both of which were written by the author. The former document introduces an ad-hoc remedy to fix the erstwhile broken method of trying to represent each output of an asynchronous logic block in the so-called Set-Reset representation where each output is assumed to be driven by a Muller C-element somewhere inside the architecture.

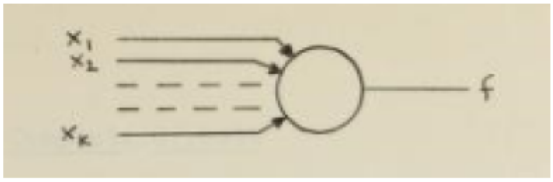
The latter document introduces the idea of a completely new way of synthesizing asynchronous blocks from their behavioral truth tables by abruptly replacing the AND operation in the canonical procedure with the * operation which signified the Muller-C operation. To recapitulate,

$$Z = A * B = A.B + (A + B)Z$$

Where Z is the value held by the output pin before any stimulus was applied to the inputs. The contention/conundrum that we were presented with was to convince ourselves that the procedure charted out in the second document was necessary and sufficient to build asynchronous circuits in general, and that the workflow which was figured out to work for some circuits in the first document formed a subset of the more general procedure. This involved a study into the history of the Muller logic [1, 2] and it was realized that the design methodology actually derived from an older process due to Huffmann [3].

It is advised that the reader go through the references, most importantly [1, 2] for a much more thorough treatment of the matter at hand as they include a complete axiomatic treatment towards asynchronous circuitry in general. To quote Muller directly, it is possible to design circuits in such a way that the relative speeds of the elements (in the pipeline) do not affect the overall behavior of the circuit. This is precisely how a delay insensitive or a speed independent circuit is defined. Thus, before even getting to the idea of a special set and reset representation, it is to be understood that the reason behind such a representation even existing is because it is driven in such a delay insensitive method to comply with Mullers thesis.

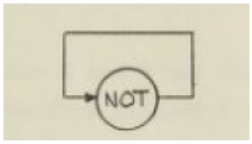
On the contrary, think of a design which did not have this restriction. Although there is a lot of flexibility and freedom in the choice of the devices in the pipeline now, it should be apparent that this process would rather become cumbersome as one has to take care of the delay of each new element being included in the pipeline to match with the required delay specification for the overall pipeline delay to match some other given specification. In addendum to this, the workflow cannot be parallelized and thus increases the number of design iterations.



The Muller style of logic design then claims to use a fancy decision element which will soon turn out to be the Muller-C gate, but is actually introduced in [1] entirely independent of the C element. An asynchronous circuit is now defined as (from now onwards whenever we refer to an asynchronous circuit it is assumed to comply with speed independence, and note that Huffmann circuits [3] need not always comply with this condition) a circuit which is built out of combinations of wires and these decision elements. The wires are obviously assumed to be perfect and thence delay free. The only specification given about the behavior of the decision element is that if the element is at equilibrium at some time t_0, and it is excited, then the output will show response to the stimulus at a time $t_0 + T$ where T is upper bounded by some $M > 0$.

As abstract as the definition of the decision element sounds, the idea should get clearer once some more terminology is introduced. A complete circuit is defined as one in which each node is driven by some decision element in the network. An immediate state (I-state) of the circuit is given by the signal values at all nodes of the circuit at a given instant of time, and an equilibrium state (E-state) is said to be attained by an I-state if all the decision elements in that instant of time are said to be in equilibrium (and thus will continue to be in equilibrium, unless external stimulus is applied). It is to be stressed here that all the combinational blocks which were built using the Muller style

asynchronous fabric all depend upon an E-state to exist in the sequence of allowed states to reproduce the behavior of the static synchronous (?) combinational logic. The cumulative state (C-state) of the circuit with respect to a given I-state S is defined as the Hamming distance of the present I-state of the circuit to S.



To demonstrate the ideas presented above, consider the simple looped inverter oscillator which would oscillate at a frequency given by the inverse of the propagation delay of the inverter. I-states for the system are given by 0, 1, 0, 1, 0 and so on. The C-states are correspondingly given by 0, 1, 2, 3 and so on. The circuit is not strictly speed independent as it does not have any well-defined equilibrium state. The definitions were introduced as a formality but we will be using only the concepts of I and E states for the present discussion.

Consider the oft-discussed example of the asynchronous AND gate. The truth table is given below for the sake of review and the discussion follows henceforth, in which the Set-Reset paradigm will be clarified as a subset of the more general Muller workflow.

A0	A1	B0	B1	Z0	Z1
0	0	0	0	0	0 (A)
1	0	1	0	1	0 (B)
1	0	0	1	1	0 (C)
0	1	1	0	1	0 (D)
0	1	0	1	0	1 (E)

The second document discusses the relatively straightforward way in which this behavioral table can be brought down to a working design. This implies that

$$Z0 = A0 * B0 + A0 * B1 + A1 * B0$$

$$Z1 = A1 * B1$$

The reason behind such a deliberate assignment will be clear if one notices the table at an atomic scale. Think of the state (B) and assume that the circuit has attained (B) as an E-state. That is, it went through a sequence of I-states and stopped (or rather looped) at (B). In technical literature these sequences of I-states which lead to an E-state are known as allowed sequences and an I-state S2 is said to directly follow an I-state S1 when they occur in that order in an allowed sequence of asynchronous operation. The definition of an E-state renders us to expect that with transitions on solitary input ports the decision elements inside the block latch onto their states to guard their previous states. This behavior is precisely what makes these circuits asynchronous. Although clock-less-ness can be seen as an identifier for such circuitry, it is their self-timed-ness which makes them an attractive alternative to synchronous design. Thus, we would want Z0 to latch onto its 1 if there are any glitches on the input pins. The idea is to now work backwards and discover how the decision element should look like for the block to have state (B) as a valid E-state. Say B0 transitioned from 1 to 0. It can be seen that there is a true behavioral change on Z0 (and not any glitch) only when A0 also transitions from 1 to 0. Until then the circuit will hold onto the logical high it attained when it was at the perfect E-state (B). The truth table for such a configuration can be written down and it can be seen that

$$Z0(+) = A0.B0 + A0.Z0 + B0.Z0$$

Where Z0(+) represents the next state taken by the output, Z0 represents the current state which is to be guarded. Readers should note that the . represents the logical AND operation. Rearranging the terms we have

$$Z0(+) = A0.B0 + (A0 + B0).Z0$$

This is precisely the behavior of the Muller C-element and thus one can confirm that the decision element which forms the basis for the asynchronous fabric is nothing but the Muller C-gate.

With this point out of the way, it only is a matter of logical consistency to see that the outputs of all the Muller C-elements are to be ORed to drive the output which they are being summed over. Consistency is stressed upon because this is what explains why the reset pin behaves the way it does. And

this is where the asynchronous FSM picture which was used in [4] fits into the compilation. Imagine that the AND gate is instead in the E-state (C). We can compactly record the state of the gate using the five tuple notation (Z0, A0, A1, B0, B1). Thus, we are assuming that we are in the E-state (1, 1, 0, 0, 1). If there is a transition on B1 and the system glitches, then we are in the transient I-state (1, 1, 0, 0, 0). Note that the output has been safely latched. Now, if there is a transition where B0 glitches now, it is up to the system to know beforehand that this is a proper stable E-state rather than an intermediate I-state. If one took only the latching at state (C) then the behavioral model would never match the functional description and the circuit would bypass the state as an I-state. Thus, all the states (B, C, and D) are considered for the summation to describe the proper latching operation of the output pin Z0.

This should clarify the (slightly semantic) argument regarding the structure and functional description of the reset pin description. Although it is explicitly defined in [6], it need not be separately calculated as the Muller workflow accounts for this property by design, and the coherence in both the ideas is hopefully made clearer with this document.

2.4.1 References

1 Muller, D. E., The Theory of Asynchronous Circuits I, 1955.

2 Muller, D. E., Bartky, W. S., The Theory Of Asynchronous Circuits II.

3 Huffmann, D. A., The Synthesis of Sequential Switching Circuits.

4 Pallaprolu, A., A Workflow for Synthesizing Asynchronous Circuits.

5 Pallaprolu, A., Muller Circuit Synthesis from Truth Tables.

6 Christmann, J.F., Prog A Asynchronous Design.

2.5 Implementation of the Asynchronous Fabric in Verilog

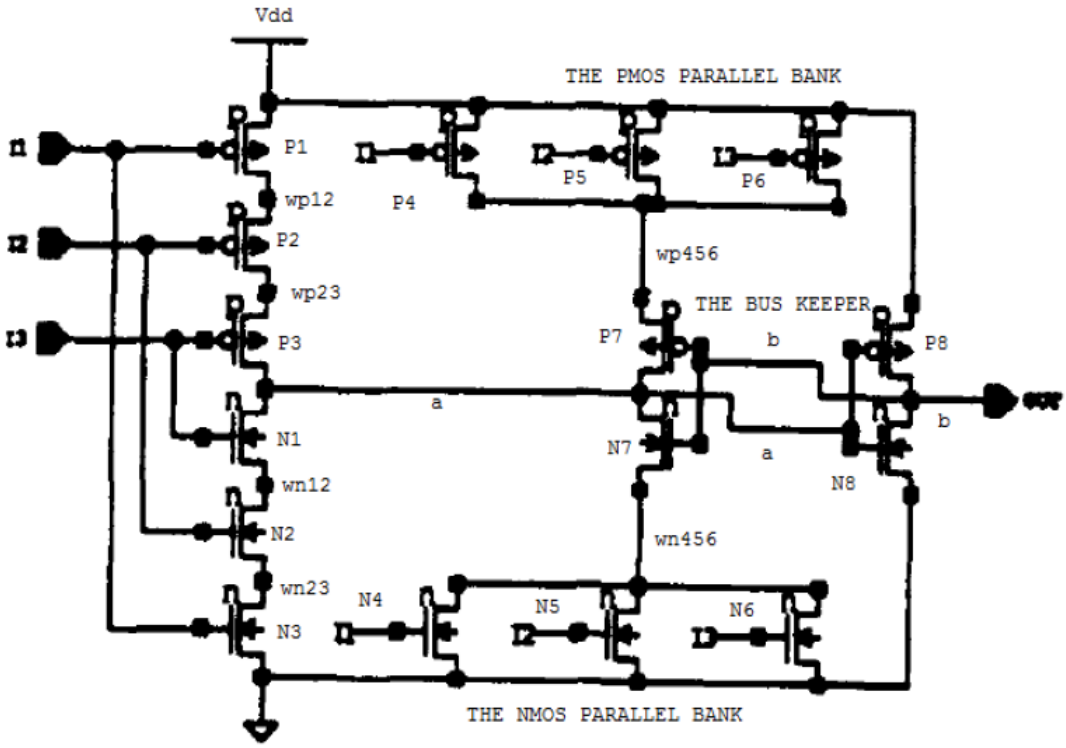
Dated 27th January, 2017

The previous document charted out an approximately universal procedure for the synthesis of combinational logic in the asynchronous logic style and it was realized that the Muller C element played a central role in this process. Hence, it is of paramount importance to understand how the C element is designed and how it can be optimized to give an overall performance boost from the root level. This study is still in progress when this document is being written, so the details might be a bit hazy but clarifications will be made in later documentation.

The first resource from which we will fish out our switch level (or) gate level implementation of the multi-input Muller C logic is the technical report [1] which also does the excellent job of recapitulating older designs and is focused on generating efficient multiple input C element modules. Charles Seitzs [2] design of the C gate is probably a small improvement over the most basic design idea possible for a functional version, the n PMOS and n NMOS in series structure for an n-input gate. However, the gate has the additional feature in which it latches onto the previous output whenever there is a transition on exclusive lines and not on all lines in unison. For this, cascaded inverter logic is used as a basic latch, this module is also known as bus keeper and bus holder in technical literature.

Seitz's design is further improved upon for quicker pull-up and pull-down by not letting the bus keeper connect to the power rails directly but instead via a parallel MOS bank, PMOSs on the pull-up side and NMOSs on the pull-down side. The final schematic is shown on the next page. The labels for the wires are given to make it easy for the reader to correlate the design with the Verilog model. Instead of going for the (obviously) easier route of defining a behavioral function for the C gate and using this to make the base Verilog module I have chosen to go with the switch level implementation for ease of implementing optimizations in the base design. To this end, it would be prudent to go over the basics of switch level design in Verilog.

Verilog provides the facility of using primitives such as `pmos, nmos, cmos`, for negative threshold, positive threshold and bidirectional switches. The instantiation of these modules does not necessarily translate to a physical placement of the MOS device but rather as a digital switch.



The usage of these switch level primitives is shown below for simple instantiations:

```
switch [instance name] (output, input, control)
```

A simple CMOS inverter can be designed henceforth in the following way, as shown below.

All the scripting was done on GVim running on Cygwin/X 1.19.1.0 using iverilog on a Windows 7 Enterprise machine.

```
module inverter(input x, output f);
supply1 vdd;
supply0 gnd;
pmos p1 (f, vdd, x);
nmos n1 (f, gnd, x);
endmodule
```

The test-bench for this module is given below, and it is compiled using the sequence of shell commands given thereafter. The output is finally displayed.

```
module inverter-tb;
reg x;
wire f;
initial begin
$ dumpfile("inverter-tb.vcd");
$ dumpvars(1, inverter-tb);
$ monitor("x = %b, f = %b",x,f);
x = 0;
# 20 $ finish;
end
always
begin #5 x = !x;
end
inverter i0(.x(x),.f(f));
endmodule
```

```
$ iverilog o inverter inverter-tb.v inverter-test.v
$ vvp inverter
```

```
pallaprolua@DLHCWD0137 ~/muller
$ vvp inverter
VCD info: dumpfile inverter_tb.vcd opened for output.
x = 0, f = 1
x = 1, f = 0
x = 0, f = 1
x = 1, f = 0
x = 0, f = 1
```

With this, the tutorial introduction for the switch level Verilog modelling is complete. The next step is to see how one can implement the most basic asynchronous circuit in HDL, that is, the implementation of the bus holder/-cascaded inverter topology. As simple as this sounds, it is slightly non trivial and interesting.

It is non-trivial because Verilog was not essentially designed for asynchronous operation. It too runs on some synchronous fundamental where it can execute sequential processes only on repeated verification of certain registers specified in a list (this is most widely used as the `always @(list)` block). A nave implementation of the cascaded inverter latch would lead one to the following module design:

```
module ilatch(d, q);
input d;
output wire q;
not u1 (q, d);
not u2 (d, q);
endmodule
```

If one is more careful or one tries to get to the complete basic switch level implementation of this, one might end up with:

```
module ilatch(d, q);
input d;
output wire q;
pmos p1 (q, vdd, d);
nmos n1 (q, gnd, d);
pmos p2 (d, vdd, q);
nmos n2 (d, gnd, q);
endmodule
```

It turns out that both these implementations would lead to faulty implementations as Verilog would end up assigning q = x for almost all test cases. There are two ways to get around this apparent roadblock. One is to use the fact that the simulator is forced to take the dont care output because it is not getting enough time to decide which logic level the output settles at. A deliberate gate delay which is introduced by a modified incantation of the

basic switch instances can lead to results. The syntax for this is given below:

```
mos-sw #pd [instance name] (output, input, control)
mos-sw #(trise, tfall) [instance name] (output, input, control)
mos-sw #(trise, tfall, toff) [instance name] (output, input, control)
```

where `#pd` refers to the propagation delay amount in the scale specified at the header. A better workaround for the issue is to specify the device signal strength capacities while instantiating the inverter `not` block. A functional Verilog module for the inverter latch is given below along with the test-bench and the output.

```
module ilatch(d, q);
input d;
output wire q;
//BUSK01 d;
not u1 (q, d);
not (weak0, weak1) u2 (d, q);
endmodule


module ilatch-tb;
reg d;
wire q;
initial begin
$dumpfile("ilatch-tb.vcd");
$dumpvars(1,ilatch-tb);
$monitor("d =%b, q = %b", d, q);
d = 0;
#10 d = 0;
#30 d = 0;
#50 d = 1;
#70 $finish;
end
ilatch i0(.d(d),.q(q));
endmodule
```

```
pallaprolua@DLHCWD0137 ~/muller
$ vvp ilatch
VCD info: dumpfile ilatch_tb.vcd opened for output.
d = 0, q = 1
d = 1, q = 0
```

With this model checking out functionally, it is only a matter of putting together the netlist for the Seitz design shown on the top of the second page with the bus keeper module being defined as above. There is a slight logistic problem in the inclusion of this module directly into the main design as the `not` gates defined in the above module are all powered from `Vdd` to `Gnd` directly whereas the inverter latch in the Seitz design has the parallel MOS banks offsetting the power rail voltages. This issue is handled by applying the signal strength restriction only to the second inverter which does indeed have its power rails connected from `Vdd` to `Gnd` directly.

With these technicalities out of the way, the module for the two input version of the Seitz implementation for the Muller C gate is given below. This is the same as the circuit on the second page, except that the number of inputs has been reduced to 2 from 3. The test-bench and the output for the simulation are given henceforth.

```
module muller-s(i2, i3, out);
input i2, i3;
output out;
wire a, wp12, wp23, wn12, wn23, wp456, wn456, b;
supply1 vdd;
supply0 gnd;
pmos p2 (wp23, vdd, i2);
pmos p3 (a, wp23, i3);
nmos n1 (a, wn12, i3);
nmos n2 (wn12, gnd, i2);
pmos p5 (wp456, vdd, i2);
pmos p6 (wp456, vdd, i3);
pmos p7 (a, wp456, out);
nmos n5 (wn456, gnd, i2);
nmos n6 (wn456, gnd, i3);
nmos n7 (a, wn456, out);
```

```
not (weak0, weak1) u0 (out, a);
endmodule

module muller2-tb;
reg i2, i3;
wire out;
initial begin
$dumpfile("muller-c-1-tb.vcd");
$dumpvars(1,muller2-tb);
$monitor("i2 = %b, i3 = %b, out = %b", i2, i3, out);
i2 = 0; i3 = 0;
#10 i2 = 1; i3 = 0;
#20 i3 = 1; i2 = 1;
#30 i3 = 1; i2 = 0;
#40 i3 = 0; i2 = 0;
#70 $finish;
end
muller-s m0(.i2(i2),.i3(i3),.out(out));
endmodule
```

```
pallaprolua@DLHCWD0137 ~/muller
$ vvp muller2
VCD info: dumpfile muller-c-1_tb.vcd opened for output.
i2 = 0, i3 = 0, out = 0
i2 = 1, i3 = 0, out = 0
i2 = 1, i3 = 1, out = 1
i2 = 0, i3 = 1, out = 1
i2 = 0, i3 = 0, out = 0
```

This module is now ready for full deployment into the workflow of Muller circuit synthesis which we discussed in the last document. As a conclusive example, I will demonstrate the Verilog synthesis of the full adder in the asynchronous fabric. The gate level circuit diagram is taken from the presentation [3] and for more information regarding the mathematics behind the workflow of the last document; the reader is urged to refer to [4]. The main module is given on the next page followed by the test-bench and the output tested for three sets of inputs. In the next document this module will be used as the atom in the implementation of the asynchronous multiplier block which will then be tested for power and timing constraints in a future report.

Main Module:

```
//One bit full adder using Seitz's Series-Parallel implementation
of Muller
//C Logic.
//
//Dual rail, two phase asynchronous communication.
//
//A. Pallaprolu
//STMicroelectronics, Noida.
//25/1/17
‘include "muller-c-2.v"
module fa-async(m0,m1,n0,n1,cin0,cin1,s0,s1,cout0,cout1);
input m0, m1, n0, n1, cin0, cin1;
output wire s0, s1, cout0, cout1;
wire w0, w1, w2, w3, w4, w5, w6, w7, w8, w9, a0, a1;
muller-s d0(.i2(m0),.i3(n1),.out(w0));
muller-s d1(.i2(m1),.i3(n0),.out(w1));
muller-s d2(.i2(m0),.i3(n0),.out(a0));
muller-s d3(.i2(m1),.i3(n1),.out(a1));
assign w2 = w0 | w1;
assign w3 = a0 | a1;
muller-s d4(.i2(w2),.i3(cin1),.out(w8));
muller-s d5(.i2(w2),.i3(cin0),.out(w9));
muller-s d6(.i2(w3),.i3(cin1),.out(w4));
muller-s d7(.i2(w2),.i3(cin0),.out(w5));
muller-s d8(.i2(w3),.i3(cin0),.out(w6));
muller-s d9(.i2(w2),.i3(cin1),.out(w7));
assign cout1 = a1 | w8;
assign cout0 = a0 | w9;
assign s1 = w4 | w5;
assign s0 = w6 | w7;
endmodule
```

When writing the test-bench, many concepts regarding the asynchronous fabric are clarified. For instance, if we would want to sequentially process three sums through our full adder and we feed them sequentially into the

adder pipeline using the following nave test-bench, then we would see the following output, which might set off a few alarms.

```
module fa-async-1-tb;
reg m0, m1, n0, n1, cin0, cin1;
wire s0, s1, cout0, cout1;
initial begin
$dumpfile("fa-async-1-tb.vcd");
$dumpvars(1,fa-async-1-tb);
$monitor("m0 = %b, m1 = %b, n0 = %b, n1 = %b, cin0 = %b, cin1 = %b,
s0 = %b, s1 = %b, c0 = %b, c1 = %b", m0, m1, n0, n1, cin0, cin1,
s0, s1, cout0, cout1);
m0 = 0; m1 = 0; n0 = 0; n1 = 0; cin0 = 0; cin1 = 0;
#10 m0 = 0; m1 = 1; n0 = 1; n1 = 0; cin0 = 1; cin1 = 0;
#20 m0 = 0; m1 = 1; n0 = 1; n1 = 0; cin0 = 0; cin1 = 1;
#30 m0 = 1; m1 = 0; n0 = 0; n1 = 1; cin0 = 1; cin1 = 0;
#70 $finish;
end

fa-async f0(.m0(m0),.m1(m1),.n0(n0),.n1(n1),.cin0(cin0),.cin1(cin1),
.s0(s0),.s1(s1),.cout0(cout0),.cout1(cout1));
endmodule
```

The output (which is obviously wrong) for the above test-bench is given below:

```
pallaprolua@DLHCWD0137 ~/muller
$ vvp fa_async
VCD info: dumpfile fa_async_1_tb.vcd opened for output.
m0 = 0, m1 = 0, n0 = 0, n1 = 0, cin0 = 0, cin1 = 0, s0 = 0, s1 = 0, c0 = 0, c1 = 0
m0 = 0, m1 = 1, n0 = 1, n1 = 0, cin0 = 1, cin1 = 0, s0 = 0, s1 = 1, c0 = 1, c1 = 0
m0 = 0, m1 = 1, n0 = 1, n1 = 0, cin0 = 0, cin1 = 1, s0 = 1, s1 = 1, c0 = 1, c1 = 1
m0 = 1, m1 = 0, n0 = 0, n1 = 1, cin0 = 1, cin1 = 0, s0 = 1, s1 = 1, c0 = 1, c1 = 1
```

This is fundamentally because the asynchronous adder does not have any value for its functionality if it is not used in a pipeline as such, and in this case, clearly, it is being used as an independent device being made to sum independent inputs separated by 10 time unit intervals. Referring to [5], it is

to be understood that all the digital logic blocks discussed so far are almost always used in a pipeline and when a block is placed in a pipeline, it will follow the protocol established for process communication and there would always be a spacer code sent after every successful operation. This is what was forgotten in the test-bench previously, that is, we need to make the test-bench simulate the adder block as if it was in an asynchronous micropipeline. Therefore, after every add request, a clear spacer is sent to refresh the block for the next operation. The improved test-bench is shown below.

```
module fa-async-1-tb;
reg m0, m1, n0, n1, cin0, cin1;
wire s0, s1, cout0, cout1;
initial begin
$dumpfile("fa-async-1-tb.vcd");
$dumpvars(1,fa-async-1-tb);
$monitor("m0 = %b, m1 = %b, n0 = %b, n1 = %b, cin0 = %b, cin1 = %b,
s0 = %b, s1 = %b, c0 = %b, c1 = %b", m0, m1, n0, n1, cin0, cin1,
s0, s1, cout0, cout1);
m0 = 0; m1 = 0; n0 = 0; n1 = 0; cin0 = 0; cin1 = 0;
#10 m0 = 0; m1 = 1; n0 = 1; n1 = 0; cin0 = 1; cin1 = 0;
#15 m0 = 0; m1 = 0; n0 = 0; n1 = 0; cin0 = 0; cin1 = 0;
#20 m0 = 0; m1 = 1; n0 = 1; n1 = 0; cin0 = 0; cin1 = 1;
#25 m0 = 0; m1 = 0; n0 = 0; n1 = 0; cin0 = 0; cin1 = 0;
#30 m0 = 1; m1 = 0; n0 = 0; n1 = 1; cin0 = 1; cin1 = 0;
#70 $finish; end
fa-async f0(.m0(m0),.m1(m1),.n0(n0),.n1(n1),.cin0(cin0),.cin1(cin1),.s0(s0),.s
endmodule
```

2.5.1 References

1 Wuu T. Y., Sarma B. K. Vrudhula Sastry, A Design of a Fast and Area Efficient Multi-Input Muller C Gate, Information Sciences Institute Technical Report ISI/RR-93-302, January 1993.

2 Seitz C. L., System Timing.

3 Christmann, J.F., Prog A Asynchronous Design.

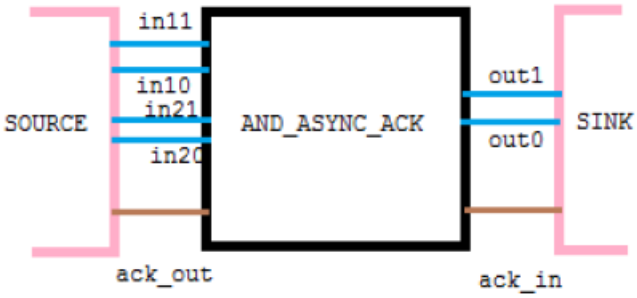
4 Muller D.E., Bartky J., The Theory Of Asynchronous Circuits: Part I and II., The University of Illinois, Urbana-Champaign.

5 Sutherland I. E., Micropipelines, The Turing Award Lecture, 1988.

2.6 Pipelined Asynchronous Logic

Dated 5th February, 2017

In this note, firstly, the claim that Verilog can successfully simulate the two bit multiplier will be proven wrong. It turns out that it is not possible for an inherently synchronous test-bench system to simulate an actual pipeline. The proof is done using the following modified modules for the asynchronous AND logic designed with the acknowledge lines inclusive. It was expected that the block was pipeline ready until it was actually tested on a pipeline which gave the same erroneous results as the two bit asynchronous multiplier. Before we speed away to the analysis of the conclusion, the main module and the test-bench for the modified AND logic is given below, along with the output for the test on a single AND block in a pipeline. (This means that there exist hypothetical data source and sink in the model around the AND block which inject the `AckIn` signal and absorb the `AckOut` signal. Refer to the module description for clarity.)

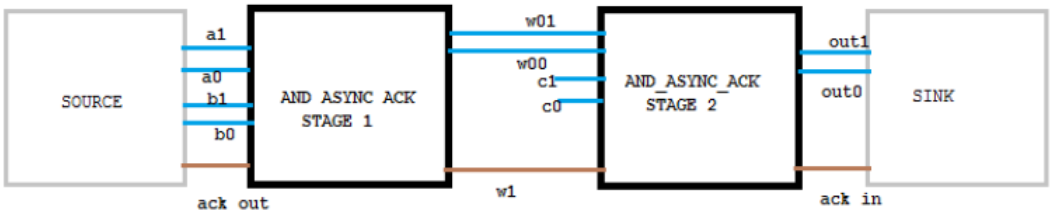



```
 //Asynchronous AND Logic
//Includes the acknowledge line, hence making it viable for pipelining
//Also has bit vector inputs.  Much enhanced from the sloppy previous
//implementation.
//A. Pallaprolu
//6-2-2017
//STMicroelectronics, Greater Noida.
`include "muller-c-1.v"
module and-async-ack(in1, in2, ack-in, out, ack-out);
input [1:0] in1, in2;
```

```
input ack-in;
output [1:0] out;
output ack-out;
wire w0, w1, w2, w3;
muller-s d0(in1[0], in2[0], ack-in, w0);
muller-s d1(in1[0], in2[1], ack-in, w1);
muller-s d2(in1[1], in2[0], ack-in, w2);
muller-s d3(in1[1], in2[1], ack-in, w3);
assign out[0] = w0 | w1 | w2;
assign out[1] = w3;
assign ack-out = ~(w0 | w1 | w2 | w3);
endmodule


module and-async-ack-tb;
reg [1:0] in1, in2;
reg ack-in;
wire [1:0] out;
wire ack-out;
initial begin
$dumpfile("and-async-ack-tb.vcd");
$dumpvars(1, and-async-ack-tb);
$monitor("in1 = %b, in2 = %b, ack-in = %b, out = %b, ack-out = %b",
in1, in2, ack-in, out, ack-out);
in1 = 2'b00; in2 = 2'b00; ack-in = 0;
#10 in1 = 2'b10; in2 = 2'b10; ack-in = 0;
#20 in1 = 2'b10; in2 = 2'b10; ack-in = 1;
#30 $finish;
end
and-async-ack a0(in1, in2, ack-in, out, ack-out);
endmodule
```

```
pallaprolua@DLHCWD0137 ~/async_toolchain_ack
$ vvp and_async_ack
VCD info: dumpfile and_async_ack_tb.vcd opened for output.
in1 = 00, in2 = 00, ack_in = 0, out = 00, ack_out = 1
in1 = 10, in2 = 10, ack_in = 0, out = 00, ack_out = 1
in1 = 10, in2 = 10, ack_in = 1, out = 10, ack_out = 0
```

So far, so good. It looks like the block is responding really well in the presence of the imaginary source and sinks providing (imaginary?) stimuli. This is actually the furthest the block gets in terms of its functionality. The following module along with its test-bench and the output seal the dispute once and for all. The block diagram for the module is shown below. It is a simple two stage asynchronous AND pipeline which consists of two modified asynchronous AND logic blocks in sequence.




```
 //Asynchronous AND Pipeline
//Uses the and-async-ack module as individual pipe element
//The pipeline is simple.  IN A, B -> (A.B) -> ((A.B).C) -> OUT
//
//The entire pipeline still needs an acknowledge in and out from
//imaginary sources and sinks.
//
//A. Pallaprolu
//6-2-2017
//STMicroelectronics, Greater Noida.
`include "and-async-ack.v"
module and-async-pipe(a, b, c, ack-in, out, ack-out);
input [1:0] a, b, c;
input ack-in;
output [1:0] out;
output ack-out;
wire [1:0] w0;
wire w1;
and-async-ack st1(a, b, w1, w0, ack-out);
and-async-ack st2(w0, c, ack-in, out, w1);
```

```
endmodule

module and-pipeline-tb;
reg [1:0] a, b, c;
reg ack-in;
wire [1:0] out;
wire ack-out;
initial begin
$dumpfile("and-pipeline-tb.vcd");
$dumpvars(1, and-pipeline-tb);
$monitor("Pipe Monitor v0.1");
$monitor("a = %b, b = %b, c = %b, ack-in = %b, out =%b, ack-out =
%b", a, b, c, ack-in, out, ack-out);
a = 2'b00; b = 2'b00; c = 2'b00; ack-in = 0;
#10 a = 2'b10; b = 2'b10; c = 2'b10; ack-in = 1;
#20 c = 2'b01; ack-in = 1;
#30 $finish;
end
and-async-pipe a0(a, b, c, ack-in, out, ack-out);
endmodule
```

```
pallaprolua@DLHCWD0137 ~/async_toolchain_ack
$ vvp and_pipe
VCD info: dumpfile and_pipeline_tb.vcd opened for output.
a = 00, b = 00, c = 00, ack_in = 0, out = xx, ack_out = x
a = 10, b = 10, c = 10, ack_in = 1, out = xx, ack_out = x
a = 10, b = 10, c = 01, ack_in = 1, out = xx, ack_out = x
```

Not a single bit line settled down to a stable value, the output is useless even for comparison with other inputs as it does not show any variation in the response to any actual stimulus. After a little thought, the reason for this behavior and for the behavior of the two bit asynchronous multiplier in [1], it can be seen that while the test-bench assigns the values for the inputs (in this case a, b, c and ack-in), the signals are not symmetric with respect to time. The ack-in signal occurs in the second stage of the pipeline which, if assuming that time moves forward to the right of the block diagram, occurs much later than the signals a and b. Thus the simulator continuously pumps

an ack-out value of x for it does not have any valid w1 values and this is because it does not have any valid w0 values which are not generated solely by ack-in. This finishes the proof of the incapability of Verilog as a simulator for strict asynchronous logic. This does not mean that Verilog is useless, for it did make it this far, and it did partly verify the functionality of the multiplier to its best. The issue comes with the way the test-bench is written, in which we inadvertently resort to thinking in a synchronous fashion when we write #10, #20 et cetera. We need to give up even this basic thought process to get anywhere with the synthesis of asynchronous circuitry.

2.7 Communicating Hardware Processes

Dated 7th February, 2017

A refreshing take on the rather mundanely confusing subject of asynchronous VLSI is given in [2]. It offers a concurrent programming approach to digital VLSI design. The circuit to be designed is first implemented as an independent concurrent programming problem in any higher level specification language, such as [3]. This program should achieve the logical specification and the appropriate behavior of the circuit. Semantic preserving program transformations are then used to compile the program to a functional circuit implementation [4]. It is claimed that the circuit will have to be correct by mathematical fundamentals of concurrency and CSP. If one has to pinpoint as to what could be the major reason behind the lag in the development of commercial EDA tools for asynchronous circuit design, it would be the fact that it is hard to find an interface which provides a good separation of the physical design concerns and the algorithm design issues.

If we were just developing an algorithm agnostic of the fabrication, timing and the electronic design, our task would be a standard problem in a CSP type theory. Speed independence/delay insensitivity/Muller style circuitries reduce the emphasis on the physical concerns with their robust design, as demonstrated in previous notes. A clear advantage in building circuits this way is that they are developed from mathematical ideas than design ideas, thereby; algebraic manipulations to the process calculus might give never-before-known optimizations and significant improvements in speed and area metrics.

Communicating Hardware Processes (CHP) [2], is a concurrent programming code inspired from CSP type languages discovered by the computer scientist Tony Hoare (which also includes Edsger Dijkstra's Guarded Command Language (GCL)). The core idea is to use CHP to describe the behavioral model in a semantically appropriate manner to aid the transformation of the truth table to a data structure known as a Petri Net. Here onwards, various techniques (described in detail in [2]) such as Process Decomposition, Handshaking Expansion, Production Rule Expansion and Operator Reduction are used to optimize and obtain the final delay-insensitive circuit. These are examples of semantic preserving transformations spoken about earlier. It is

to be emphasized that CHP is a program notation and not an actual HDL (this is a very common misconception).

The only fundamental data types in CHP are Boolean (`bool`) types (this makes sense because in physical design they all actually are just electrical lines holding zero or `Vdd` volts). Every other data type can be constructed using this type in syntax very much similar to PASCALs record data type. The bool type has only one property, and that is of assignment; in which it can be either `TRUE` or `FALSE`.

```
bool : b := TRUE;
bool : b := FALSE;
```

An integer of length n can be described using n such Boolean fields in a record data type. The definition of an integer along with the assignment of the record entries is shown below.

```
x : integer(8);
x.2 := FALSE;
x.7 := TRUE;
```

One can make any type of records. Shown below is the example of an ALU process record.

```
alu = record
      op: alu.15……alu.12
      x: alu.11…alu.8
      y: alu.7……alu.4
      z: alu.3……alu.0
 end
```

The next elements in the syntax are the control flow structures. These consist of two types and are wholly inspired from GCL. They are the selection commands and the repetition commands, analogous to the if-else and the do-while loops in standard procedural languages like C.

```
select = [G1 -> S1|G2 -> S2|.....|Gn->Sn]
```

Here the Gs are the guards for the Ss and whenever any G is Boolean true then the corresponding S is executed. If more than one G is true, the S is selected arbitrarily. Similarly, the repetition command is stated below

```
repeat = *[G1 -> S1|G2 -> S2|.....|Gn->Sn]
```

Whenever any G is true, the corresponding S is executed until G is false. Again, ties are broken with arbitration. These procedures have simpler cousins such as the following:

```
[G] => [G -> skip]
*[S] => *[true -> S]
```

The first one literally translates to skip the instruction the moment G is true. The second one similarly translates to while true do S which translates to repeat S forever. The next procedure is the reactive process given by

```
rp = *[[G1->S1 | .....|Gn -> Sn]]
```

The process description shown above can be translated to colloquial English using the definitions of the previous two constructors. The translation is Repeat forever: if G is true then execute S else skip. Note that the inner quote is repeated forever and a skip occurs only if all the guards are false.

We need to have controlled repetition type commands, that is, repetitions whose count is in the control of the user. These constructs are similar to the for loop in procedural languages like C. This is made explicit with the following construct, known as the replication construct.

```
<L i: n..m : S(i)> = S(n) if n = m
                   = S(n)L <L i: n+1..m : S(i)> if n < m
```

This is a recursive definition with the base case given by the first line and the recurrence given in the next line. To clarify the notation, for example,

```
[<|i: 0..3: Gi -> Si>] = [G0 -> S0 | G1 -> S1 | G2 -> S2 | G3 -> S3]

<;i: 0..2: x.i = y.((i+1)mod 3)> = x.0 = y.1; x.1 = y.2; x.2 = y.0;
```

Note that semicolon delimited statements are to be executed sequentially.

Concurrent execution of processes is represented by the —— symbol. Thereby, if p1, p2, p3..., pn are processes waiting for parallel execution, then the whole process is represented as

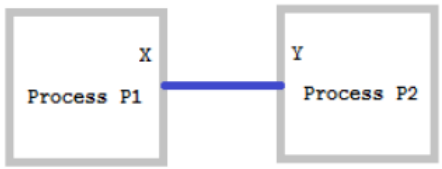
```
P = p1 || p2 || ..... || pn
```

For CHP to be more useful, we need to use the above defined building blocks to make processes which represent higher levels of design hierarchy. These are the process and function constructs. The procedure construct involves the definition of the input and output ports, much like Verilog, followed by the description of the process using the basic constructors of the notation. Functions are reduced procedures where the procedure call itself carries the output information.

```
procedure p(x:in; y:out); S
p(x:a; y:b) -> x:=a; S(x,y); y:=b;

function y(x:in); S
y(x:=a); S(x);
```

The first line demonstrates the syntax and the second line is a banal example of the same. We would also like some other elements in the upper hierarchy which connect procedures and functions of processes. These are the port and channel constructors, which come under the communication constructor class in general. Let us say we have two processes p1 and p2 which communicate with each other asynchronously and also execute concurrently. How could one represent this situation in CHP? We introduce the chan construct which solves this issue. The framework is shown below for arbitrary p1 and p2.

```
P1 := process(x) ..... end
P2 := process(y) ..... end

P1 || P2

chan(p1.x, p2.y)
```



This is not all that there is to describing the channel, but there are a few axioms which the ports should inherently follow for sane synchronization. We denote the number of completed send actions from port X shown in the block diagram above by cS and the number of completed receive actions from port Y by cR. For a channel with delays, the difference, cS cR would represent the number of actions still in the channel buffer. For ideal channels, cS = cR but in general, cR ¡ cS or else the channel would be carrying negative information. But in most cases the processes would be operating concurrently and the ports would not be responding so nicely in a sequential manner. This requires us to introduce the probe construct, invented by Alain Martin (refer [5]). The idea behind the probe is to facilitate the latching and waiting of port X if port Y is reflecting a pending/busy status. This idea actually forms the basis for the Muller element while translating from a higher level description to fabrication.

The predicate for process P is pending is denoted by appending a q in front of the process indicator, such as qP. For those comfortable with RF communication terminology, in the previous example, qX = TRUE would refer to the Tx pin of P1 being busy and qY = TRUE would mean that Rx pin of P2 is busy. If an ideal channel is at hand, then another axiom which can be stated in general is that ˜qX + ˜qY = `TRUE`, where + refers to `OR` logic. For the sake of avoiding obfuscation, \x is also used sometimes in the place of qY and \y is used in the place of qX. To conclude, \x implies that y is pending and \y implies that x is pending.

Example 1 : The Multiplexor.
The CHP description for a two to one line multiplexor is given below. The mux has two inputs X and Y and the output is X if Y is pending and vice versa.

```
sel := *[[\x -> x | \y -> y]]
```

This translates to repeatedly process either X or Y if Y or X is pending respectively. The process hits a block if both X and Y are pending, and then ties would be broken arbitrarily.

Example 2: Fair Multiplexor.
As an improvement on the previous example, here is a mux which doesnt break ties arbitrarily.

```
fsel := *[[\x -> x; [\y -> y | ~\y -> skip]
          |\y -> y; [\x -> x | ~\x -> skip]
          ]]
```

A literal translation would look like the following: (if Y is pending do X then (if X is pending Y else skip) else do Y then (if Y is pending do X else skip)).

Example 3: Stream Merge.
The module has two streams of possibly infinite sequences of inputs, denoted by X and Y. For the sake of discussion, these inputs are assumed to be integers of size 8. The process arbitrarily merges X and Y into one single output stream Z. There is a deadlock when both the input streams are empty, when the process is unable to decide whether the transmission has ended or it is a legitimate empty stream.

```
STREAM_MERGE := process(x?int(8), y?int(8), z!int(8))
                u := int(8)
                *[[\x -> x?u; z!u]
                 |[\y -? Y?u; z!u]
                 ]
                end
```

A literal translation would look like the following: (if Y is pending sample X to Z) or (if X is pending then sample Y to Z).

Example 4: One Bit Buffer.
It is a very simple process to describe. Given an input, pick it up, assign it to a dummy variable and push it to the output. The interesting variant is the higher order pipelined buffer explained in Example 5.

```
BUF1 := process(L?int(8), R!int(8))
        X: int(8)
        *[L?x; R!x]
        end
```

The literal translation of this would look like: repeat forever (pick up x from L, place it in R).

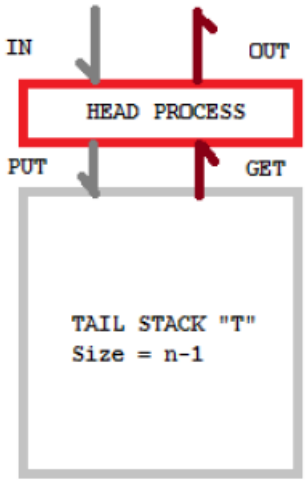
Example 5: N Bit Buffer.
The N bit buffer uses the replicator construct along with the channel constructor for linking N one bit buffer processes (BUF1s) to construct the pipeline. This is a very basic example of an asynchronous pipeline described in CHP and one can notice the compactness in the description.

```
BUFn := process(L?int(8), R!int(8))
        p(i=0..n-1): BUF1
        <||i:0..n-1:p(i)>
        chan<i:0..n-2:(p(i).R, p(i+1).L>
        p(0).L = BUF(n).L
        p(n-1).R = BUF(n).R
        end
```

The first line gives the process port description. Both the input and output are integers of size 8. The second line instantiates n free floating BUF1 modules. These modules are then executed concurrently in the third line using the replicator construct. The fourth line uses the channel constructor to link the consecutive BUF1 processes, and this might be a bit confusing at the first go but it really is the replicator used in tandem with the chan constructor. The fifth and the sixth lines finally describe the boundary conditions for the BUFn process.

Example 6: The Lazy Stack.
It is assumed that the reader has familiarity with the stack data structure. It is a LIFO data structure with two operations: PUSH for adding a new incoming element onto the top of the stack and POP for accessing the topmost element of the stack. One can clearly see that this definition is amenable to a recursive implementation in CHP. This can be seen with the following piecewise definition:
Let S be the stack process of size n (n ¿ 0) and let h be the so-called head process of the stack. As one might have already guessed by now, h is what you would get if you pop S at any instant. The distinction between the data structure and the process is that the head element is still a process and popping it does not truly mean the process has exited but it rather implies that

the process has fired. The rest of the stack is represented by T which is a stack process of size n-1. Thus, if n = 1, then S = h.
The following block diagram should clarify the input and output communication ports of the head process.



The PUT and GET communication ports are analogous to the PUSH and POP actions on the stack abstract data structure. One could go about writing a single process for the head process and one could in principle use the replicator to tag along many such processes to create a functional stack (thereby making the stack truly lazy), but a more pragmatic approach would involve identifying empty and full slots in the stack process. Each sub-process in the stack process could either be an empty process or could be a process which is holding some data along with staying in the stack process. These two processes are defined as E and F. This style of construction bases its genesis on the fact that at some point or the other, every sub-process in the stack process gets to be the head process at some point in its lifetime. Added to this, every new process entering the stack will first be the head process and would later on join T as a sub-process.

```
E := procedure
     [qout -> in?x; F |
      qin -> get?x; out!x; E]
     end

F := procedure
     [qin -> out!x; E |
      qout -> put!x; in?x; F]
     end
```

The stack could start with a basic empty element E and the elementary process description of E would take care of the rest of the functionality. The literal translation of the process E can be given as: if out is pending then (take input from in and pass to dummy x then go to procedure F) else if in is pending then (take input from get and pass it to dummy x then pass x to out like a buffer and finally go to procedure E). The literal translation of the process F can be given as: if in is pending then (output x and go to procedure E where the token is now present) else if out is pending then (output x via put and pass input via in using the same dummy sequentially and then go to procedure F).

The stack operation can be explained now. If it is empty, then it checks for the in/out lines. If out is pending/busy, then it takes input via in and calls F, thereby making the block a full one. Once F is called, it checks if out is still pending, if yes, it pushes x onto the T stack via the put line and waits for new input on in. However, if in was busy now, then it simply pushes x to out and empties the head process again. A nice trick due to Peter Hofstee of Caltech [1] is to replace the recursive calls with the infinite repeator construct to make the implementation block independent. Thus, E would now look like the following.

```
E := *[[ qout -> in?x |
         qin -> get?x ];
       [ qin -> out!x |
         qout -> put!x]
      ]
```

This can be literally translated to: Repeat forever (if out is pending then (pass in to dummy x thereby being ready to put) else if in is pending then (pass get to dummy x thereby being ready to out); if in is still pending then (pass x dummy to out thereby completing the POP) else if out is still pending then (pass x dummy to put thereby completing the PUSH)).

The following examples are slightly advanced in nature, and a review of earlier examples is suggested before proceeding any further (if you have come here directly).

Example 7: The Palindrome Recognizer.

Problem: You are provided with a sequence of bits entering the module as a stream depicted by S(i,0..n-1)and the module has to output a Boolean value. It returns true if the sequence entered so far qualifies to be a palindrome. A palindrome is defined as a bit string which has equal bits in diametrically opposite positions about the center.

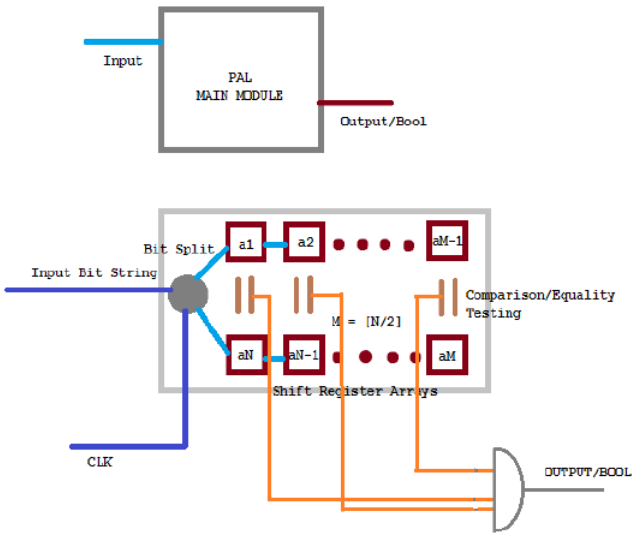
```
In: 0 -> Out: 1
In: 10 -> Out: 0
In: 010 -> Out : 1
In: 0010 -> Out : 0
and so on.
```

Our task is to design a process description for the module in CHP. In theoretical computer science there are many algorithms available to perform operations of such type but we will resort to the most simple (and also the fastest due to the usage of the asynchronous fabric) procedure of parallel comparison of bits diametrically opposite with respect to the center of the input bit vector so far. That is,

```
For all i: 0..[n/2]-1,

S[i] = S[n-1-i]

If even one of the parallel equalities is not satisfied, return 0.

[] stands for the least integer function.
```

Solution: The base case involves a single bit input which is taken as a valid palindrome without loss of generality. Thus the default output for the first bit entering the process is 1. The next bit which enters is then compared with the first bit for equality. If the two bits are equal then the result is 1 or else it is 0. Then the next bit enters, but this time the third bit has to be compared only to the first bit and the second bit has no significance for it is the odd one out. If both match then one outputs 1 or else 0. Then the fourth bit enters and now we need to do two comparisons in parallel. One is that of the second and third bits, and the other is that of the first and the last bits. If either of the equality tests fails, one has to report a 0 or else it is a palindrome and the process exits with a 1.

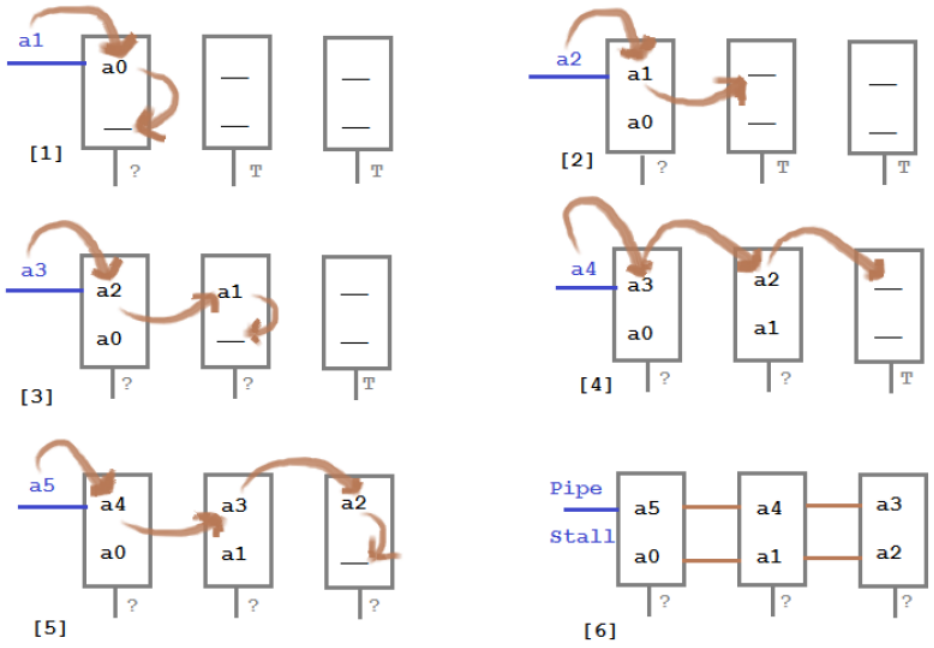
This style of operation inherently calls for a modular design method where the equality testing can be conveniently fit into a sub-module. That is, the flow of data should be handled in such a way that consecutive bits that enter must be placed into the right sub-modules so that the right equality tests can happen. The output from each sub-module is then ANDed to get the output of the process as a whole (as each equality is supposed to fire successfully for the whole bit stream to be a palindrome). The block diagram for the main procedure is shown below, along with an approximate picture of how the synchronous design for the module would look like. This is done just to compare with the pipelined design we will get to next.



The point that is emphasized in this example is that the fabric of design is independent from the algorithm implementing the process. The algorithm is described below in a style similar to the input specification. Notice that there is no mention of the clocking or non-clocking of the procedure. They are just instructions executed when the input matches the specification. This description can be thought of as a way a human would approach the issue of recognizing palindromes.

```
In: a0 -> Out: True
In: a0a1 -> Out: (a0=a1).True
In: a0a1a2 -> Out: (a0=a2).True
In: a0a1a2a3 -> Out: (a0=a3).(a1=a2).True
```

And so on. One can write an output prescription for any given input specification. Implementing this directly into hardware would need the system to come up with the prescription consciously. This is fortunately tackled with asynchronous pipelining. The following flow diagram shows how the data is expected to settle down in the pipeline for the process to implement the output specification just the way we want it to.



It can be seen that there is a pattern in the way the pipeline is filled as we move from the state [1] to state [6]. This is not a special pattern but

this removes the necessity of having a synchronous bit-splitter module which pushes alternate bits into two parallel shift registers controlled by a common clock. Each pipe element can be codified in the following way in CHP:

```
cell:= process(in?char, out!bool, put!char, get?bool)
       var x, y: char
       var z : bool
       in?x; out!true; --these lines are executed sequentially.
                       --and the output is set to default 1.
       z := true
       *[in?y; out!((x=y).z);put!y;get?z]
       end
```

The repeator construct line might cause some confusion but a second look at the flow diagram shows how the individual cells synchronize via this assignment policy. The first input is taken and the default output is set to 1 while the system waits for the next input to arrive. If no input arrives the palindrome is confirmed as the base case, and if there is a next input it is compared with the first input and it is ANDed with the previous output stored in z which holds 1 by default for an empty cell. After the comparison is done and the output is generated, the new input is pushed to the next cell in anticipation of more bits entering the present cell. Notice that the confirmation of this behavior is given by the fact that the lower bits remain there once they get there and the higher bits propagate deeper into the pipe until they become lower bits and stabilize.

The final blow to the synchronization procedure is dealt by the actual palindrome recognizer module which essentially uses the replicator construct to connect these cells in parallel.

```
pal:= process(in?char, out!bool)
      p(i:0..M-1) : cell
      <||i:0..M-1:p(i)>
      chan<i:0..M-2:(p(i).put, p(i+1).in)>
      chan<i:0..M-2:(p(i)|.get, p(i+1).out)>
      p(0).in = pal.in
      p(0).out = pal.out
      end
```

The module description must be self-explanatory by now. The new inputs are pushed deeper into the pipeline as explained before, this happens in the fourth line. The output status of the previous pipe stage is fed into the next pipe stage in the fifth line. Finally the boundary conditions are applied making the processor ready to use.

The next example deals with the famous dining philosophers problem due to E. W. Dijkstra and Leslie Lamport [7]. The asynchronous implementation is given in [6] and the discussion is inspired from the same.

Example 8: The Distributed Mutex and The Ring Of Processes.
Problem: An arbitrary number k ¿ 1 of master processes make independent requests for exclusive access to a shared resource. The process design should handle the requests so that the following two conditions are always satisfied:

1 A request processed by the master process is eventually granted. That is, there is no scenario where a master process is denied the access to the shared resource.
2 At any given point of time, at most one master process can use the resource.

It is to be kept in mind that all the master processes are independent of each other at all times and do not influence the decision making of the ring as a whole. Every master process M is provided with a server process m. A server process is said to be privileged when it allows mutex access to the resource (mutex is a contraction of the phrase mutually exclusive). The server access is private only to the master but independent servers can interact with each other via dedicated request-acknowledge lines.

The distributed mutual exclusion principle is quite famous in theoretical computer science and comes in various incarnations such as the dining philosophers problem, the cigarette smoker problem, the sleeping barber problem and so on. These problems have the common symptom that they lack either one or all three of the following properties: concurrency, synchronization and stability. Many solutions have been given in the past by pioneers in computer science, the most famous ones include Dijkstra and Lamport, Mani Chandy and J Misra, and Suhas S. Patil and they mostly involve granting a privilege to the server which is accessing the resource. The way in which the privilege is allotted varies from implementation to implementation. Some advocate

the usage of a drifting privilege in which the privilege is continuously circulated and is picked up whenever it reaches the master process which needs the access to the resource. Some advocate the usage of a distributed privilege, which is what we will be using to solve the issue in this note. The other paradigm which is rather nave is the Perpetual Continuum approach where it is assumed that there is always a master process which is requesting access. All the approaches are discussed and implemented in CHP in [6].

The distributed privilege approach involves the propagation of two signals around the ring of processing servers. Let us assume, for the sake of discussion, that there exists a server m* which has the privilege at the current instant of time and is giving access to the resource to its master M*. Let us now assume that some other (non-adjacent, without loss of generality) master process M requests access via its server m. The request from the server m to m* is sent in a clockwise fashion around the ring. When the server m* receives the request, it hands over the privilege token to m by passing it around the ring of processes in a counterclockwise fashion.

The process description for the seeking master process M is easy to write. It handles one of the two events, either it is in need of the critical resource (identified by the unknown process CS) or it does not (identified by the unknown process NCS). These processes are okay to be unknown to us because they depend upon the construction of M to begin with and that detail is actually irrelevant to the discussion.

```
master := *[NCS -> out!D | CS -> out!D]
```

It can be seen that the process construct was not explicitly used although the output port specification was given. The reader is asked to get comfortable with such omissions for simplified representations. The server process description is slightly tricky and the literal translation is given after the code for the sake of clarity.

```
server := *[[qD -> [b.R -> skip | ~b -> R := true; end]; b := true
           |qR -> [b.L -> skip | ~b -> R := true; end]; L := true;
            b := false]]
```

The program becomes simplified when broken down to the atomic level. Firstly, it is a characteristic reactive process. That is, the inner selector

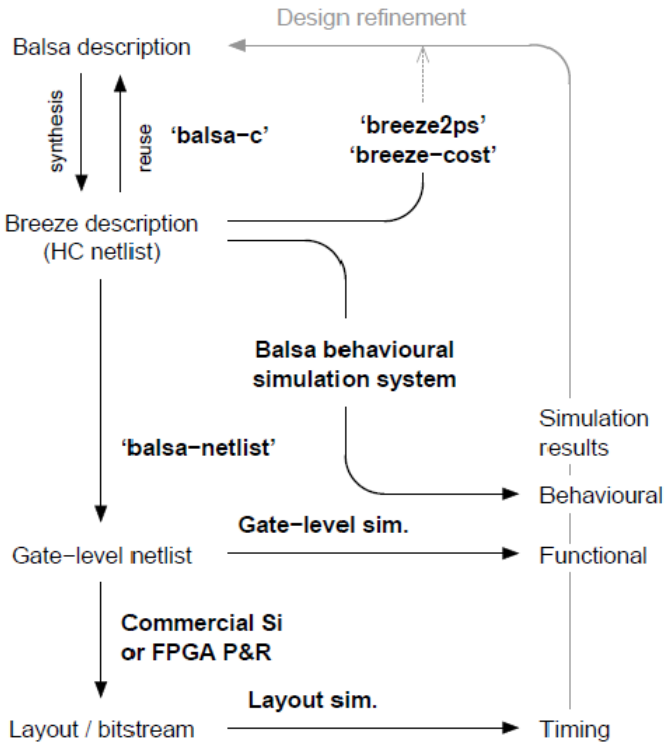
is repeated infinitely. If the server to master line is busy then the process checks whether the bit line b.R is high or low, thereby verifying whether the server holds the privilege at the moment. If b.R is high then this means that the server holds the privilege and is transmitting the resource to M and therefore, there is nothing much left to be done, so it skips the control flow structure and sets b to true again to confirm the usage of the resource. If, however, b is low, then the R line is set to high thereby starting the clockwise request operation. The propagation of this request is handled by the second guarded command in the repeator construct. The R line being busy checks whether the current server has the privilege bit b as high or not. If b.L is high, then this means the request has reached the destination and thus the inner selector can be skipped to the outer commands, where the L is set to true again to confirm the propagation of the token counterclockwise as expected. Then the privilege bit for the present server process is set to false thereby concluding the local sequence of events. This description assures two things: it only caters when the master process is requesting the server process by keeping the U-D line busy and the access request and access grant tokens viz. the R line and the b bit respectively are propagated in opposing directions thereby preventing any deadlocks.

The final step that remains is to combine the above descriptions to construct a larger process for the whole ring. This happens in two steps. One sub-process is constructed for each master-server pair and this is used as the atom to build the ring. The descriptions are given below and the literal explanations are left to the reader as an exercise (!).

```
master-slave := process(L, R)
                m: server
                M: master
               (m || M)
                chan(m.U, M.D)
                end

ring := process
        p(i:0..n-1): master-slave
        <||i:0..n-1:p(i)>
        chan<i:0..n-1:(p(i).R, p((i+1)mod n).L)>
        end
```

It is to be noted that the process for the ring has no inputs or outputs because it is inherently a self-sustaining system and not because of any wanton suppression for compactifying the representation.

The BALSA system developed by the VLSI team at University of Manchester provides a ready to use platform for writing the process descriptions in high level BALSA script which gets analyzed and optimized to a gate level netlist compatible with a CAD system of your choice. The BALSA script is firstly converted to a Breeze script which acts as an intermediate RTL compiled file ready for either netlisting, or for speed-cost metric analysis, or even for generating the Petri net used in optimizing the fabric. The workflow is shown below and it is taken from the manual [3].



2.7.1 References

1 A. Pallaprolu, Synthesis of Higher Order Asynchronous Circuits: Pipelining, February 2017.

2 A.J. Martin, Synthesis of Asynchronous VLSI Circuits, Caltech-CS-TR-93-28.

3 D. Edwards, et. al., BALSA: A Tutorial Guide, Univ. Of Manchester.

4 D. Borrione, et. al., Introducing Formal Validation in an Asynchronous Circuit Design Flow, TIMA Laboratory, Grenoble, France.

5 A.J. Martin, The Probe: An Addition To Communication Primitives, Information Processing Lett. 20, 1985.

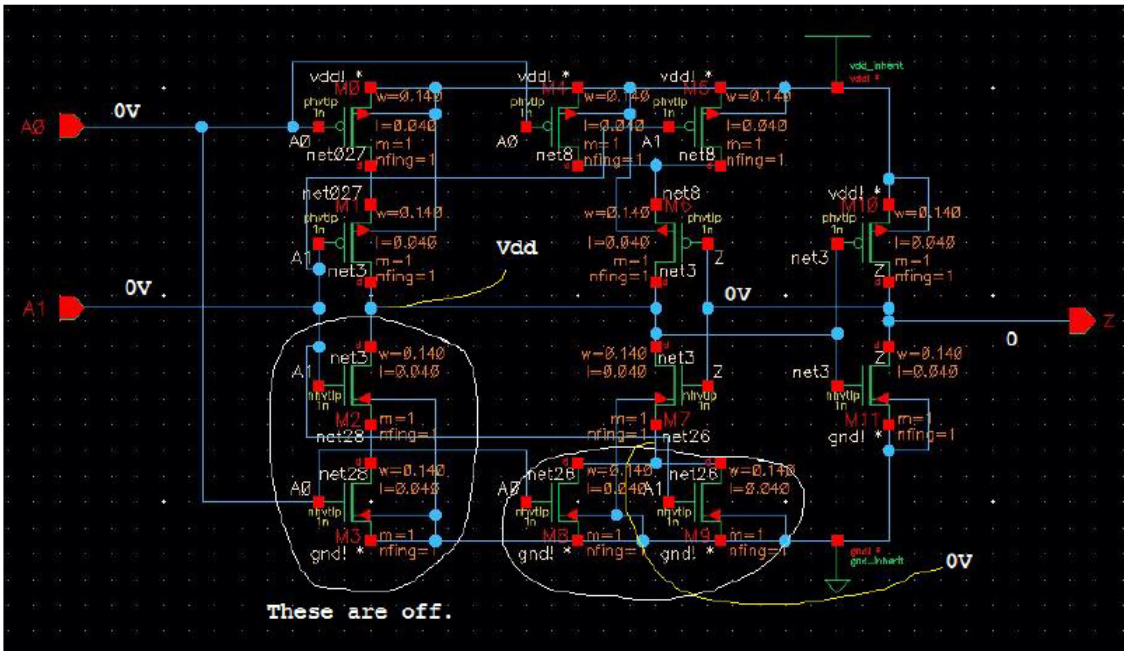
6 A. J. Martin, Distributed Mutual Exclusion On A Ring Of Processes, Science Of Computer Programming, Vol 5, North-Holland, 1985.

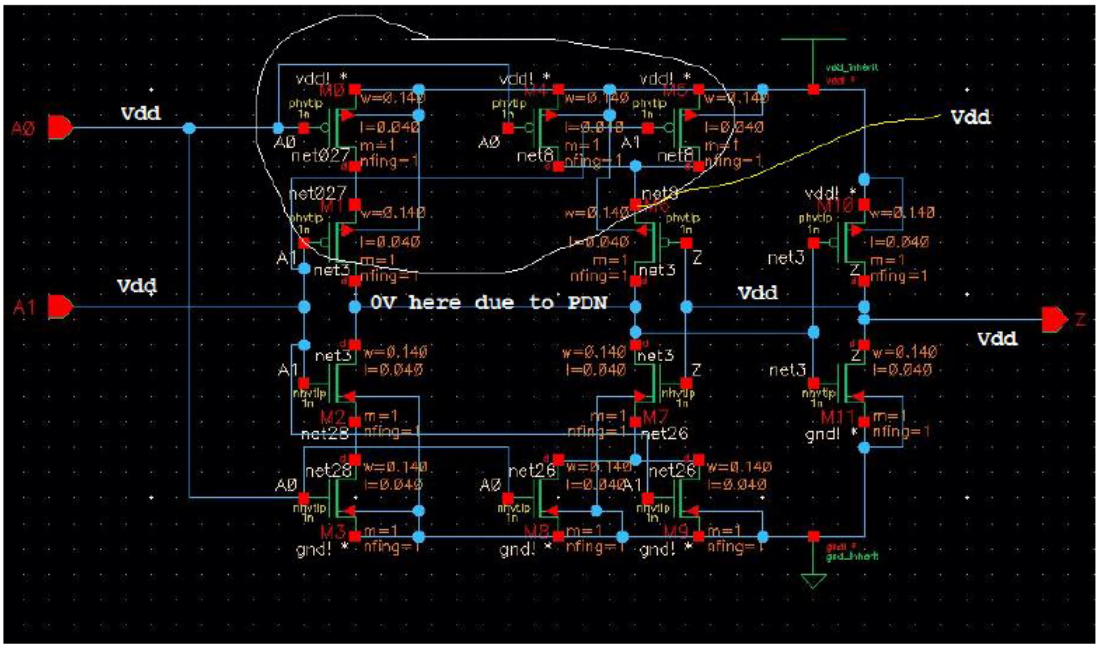
7 E. W. Dijkstra, A Discipline of Programming, Prentice-Hall, 1976.

2.8 Asymmetry in the Muller C Element

Insofar as whatever discussions took place with regards to the Muller C element or the asynchronous logic synthesized using the element the order in which the inputs to the C element was provided was considered to be immaterial to the operation of the device. This is strictly not true and we are rather fortunate that we have avoided such peculiarities so far. In most of the simulations done so far, the initial input that was supplied to the C element segments of the device was the spacer symbol 00. This acted as a flushing signal and set the outputs to Boolean low as soon as this configuration was sensed. These outputs are considered as strong Boolean low for they are actually at a strict 0V in reality.

The explanation to this behavior is also pretty simple. If the inputs are both low, then the NMOS arm is disconnected and the PMOS arm is conducting along with the bus keeper section where the NMOS has a base voltage of 0V due to the parallel NMOSs being switched off. This implies that +1V is propagated via the coupled inverters to give the output as 0V. Refer to the schematic below for clarity.



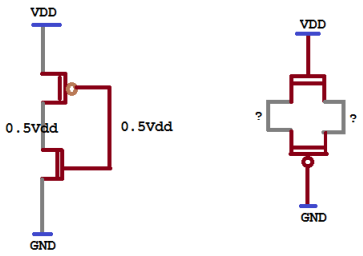


The same reasoning can be used to understand the output of the C element when it is fed with a 11 type input. While in practice this never occurs for 11 is branded as the test signal for an error/race hazard, it is not a forbidden input, and it only leads to the PMOS series arm to be switched off and the NMOS series arm to be activated fully, thereby pulling the common line voltage to 0V. The output thence stands at +1V and this is a strong Boolean high.

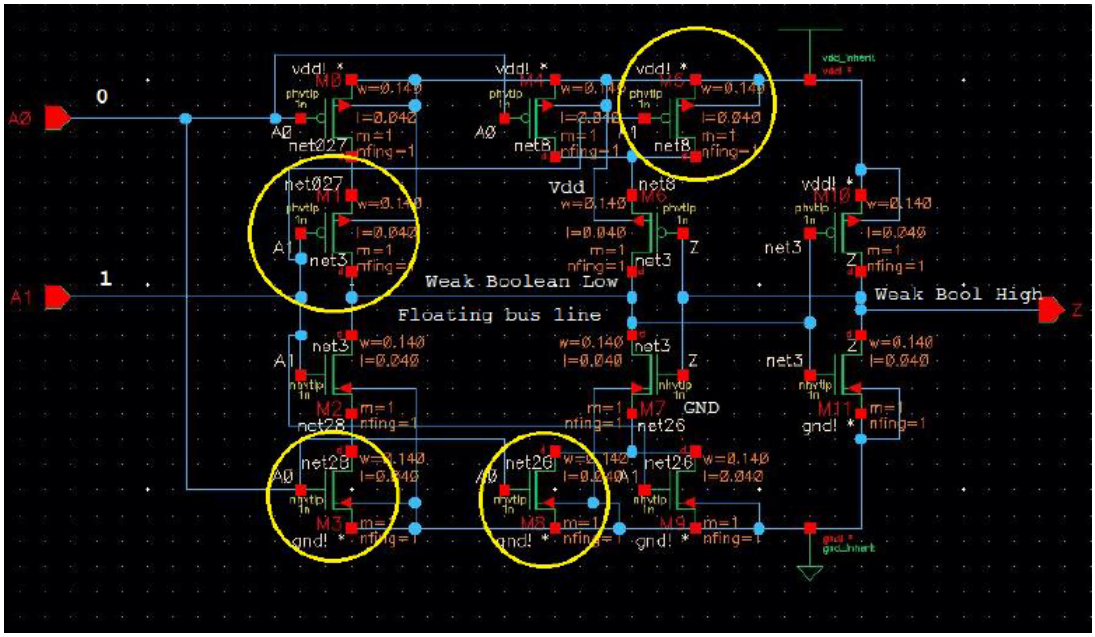
What would happen if the initial input that is supplied is not a set type input? That is, what if the initialization looked like 01 or 10? How would the circuit behave? It can be noted that this is not the first time when one comes across a situation in electronics where one starts with the state of the sequential machine which depends upon the (seemingly) non-existent past of the device. Two stark examples are shown on the next page for demonstration. The first example is a fully powered inverter with the PMOS and NMOS gates shorted but not supplied with any real voltage value to invert. The interesting question that can be asked is whether the output would oscillate or settle down at a value, and after some static analysis one can see that the only solution for the given system is if both the input and the output

settle down at the switching threshold of the inverter. This is typically taken as Vdd/2 for calculation purposes (assuming equal pull-up and pull-down strengths).

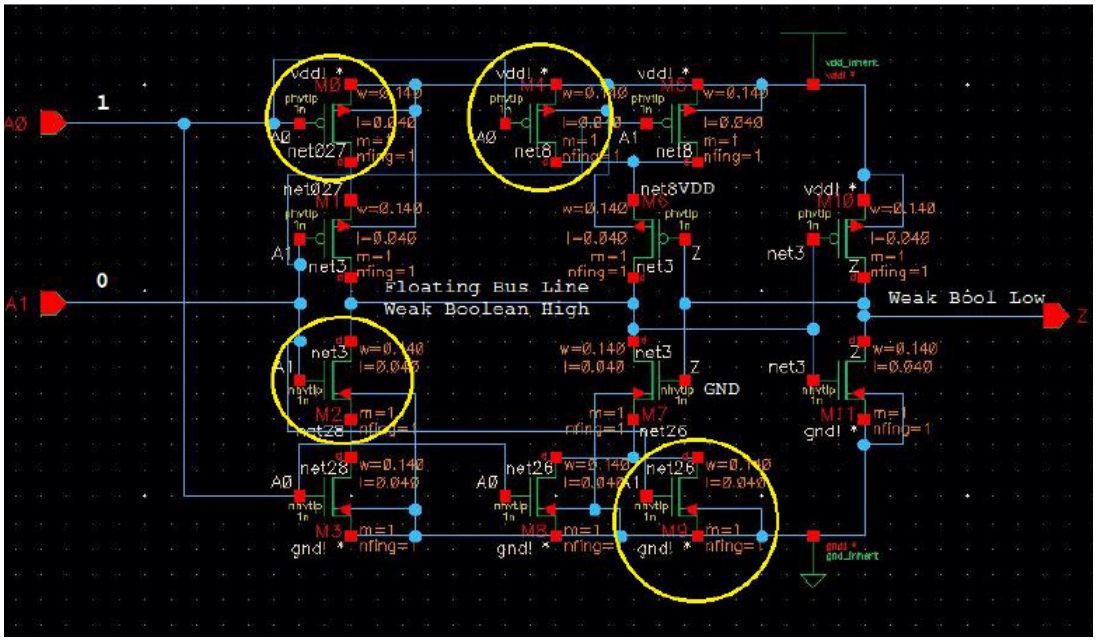
The second example is that of a transmission gate with its control pins powered but there is no actual input fed into the gate nor is there any output taken from it. The solution to this is left to the reader as an exercise.



Onwards with the C element, we see that if we apply a 01 input, the following schematic would result with the switched off sections of the circuitry circled in yellow.



The series MOS bank is completely disconnected with just the active NMOS driving the bus line which connects to the bus keeper stage. This pulls the line to a weak Boolean low voltage where the line is held around 80 microvolts due to leakage effects. This state is then inverted to a weak Boolean high output which is around 0.9995 V. The asymmetry in the operation of the C element can be observed if the 10 input is fed instead as the initial input. Refer to the schematic shown below.



In this case, the floating bus line is held only by the active PMOS in the series MOS bank and this drives the line to a weak Boolean high thereby leading the output to be driven to a weak Boolean low. Although the C element looks symmetric in operation, it actually behaves symmetrically only when it is initialized correctly. The inherent asymmetry in the design has been hopefully demonstrated in this document to a certain extent.

Chapter 3

The Wake Up Flop

'Machines take me by surprise with great frequency.'

Alan Mathison Turing

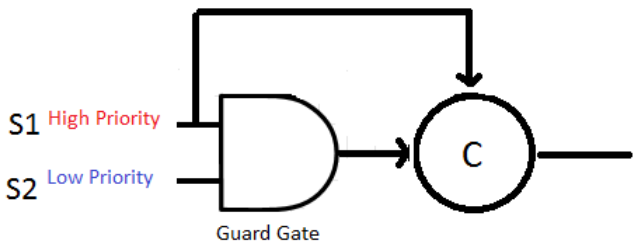
'I am very seldom interested in applications. I am more interested in the elegance of a problem. Is it a good problem, an interesting problem?'

Claude Shannon

3.1 Prelude

The last section of the previous chapter ended with a demonstration of how the static C element showed an input asymmetry when initialized with different bits, and this means that when the C element has to poll two inputs which have different signal priorities, one has to carefully wire these inputs to the C element so that the expected behavior is obtained. This would become extremely cumbersome if one had to do this every time a C element is used.

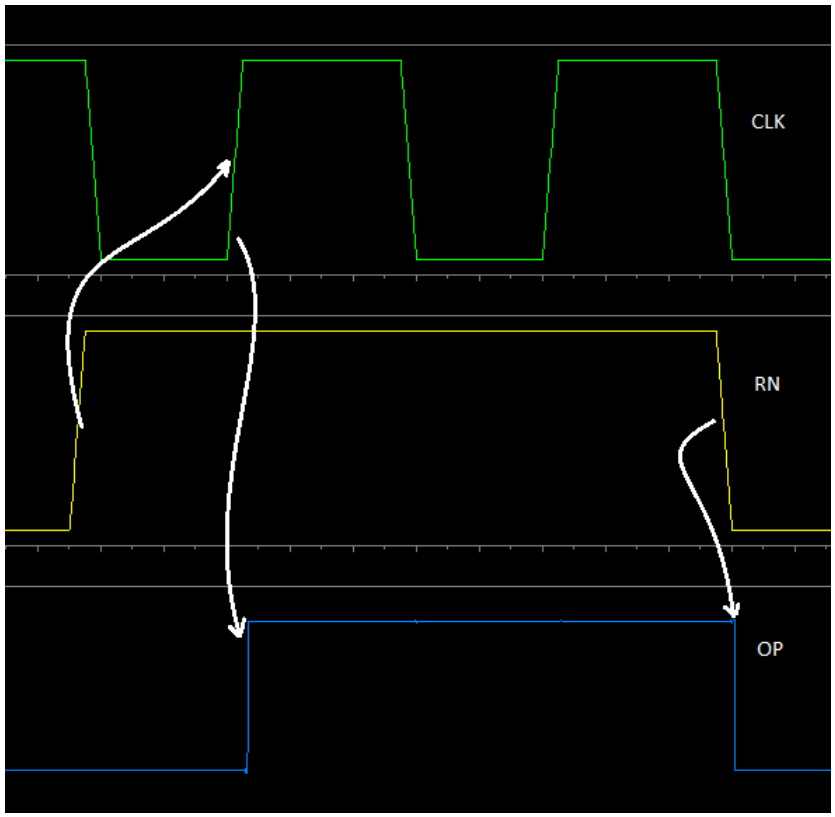
It can be proved that it is impossible to create a device whose Boolean nature matches the truth table of the C element *and* has symmetric inputs for any two independent signals, and the proof relies on the fact that any attempt to synthesize this device leads to quantization of time, essentially getting us back to synchronous design. However, if the signals at the inputs are ordered and not mutually independent, then there is a way out of this issue and it is done by 'guarding' the C element with an `AND` gate to ensure priority. That is, the signal with the highest priority has the ability to single handedly control the output of the C element independent of the other signal but the lower priority signal has the power to ensure latching of the C element until it decides to set an output.



This idea is inspired from *Event-driven Asynchronous Voltage Monitoring in Energy Harvesting Platforms* by Christmann, Beigne et. al. (New Circuits and Systems Conference, 2012), and I would like to thank my mentor, Mr. Alok Tripathi, for providing me with this reference.

3.2 Problem Statement

We are provided with the waveform shown below, and we are to construct a device in the Muller (delay insensitive) fabric which emulates the output behaviour given the two input signals.



Inputs A periodic signal (CLK) and an asynchronous trigger (RN). The device is called a 'Wake Up Flop' due to it's characteristic absence of a 'Data' input, thus in reality it does not store any data to form a synchronous signal but just 'wakes up' whenever the trigger is pulled.

Output An asynchronous signal OP.

Rule 1 If RN shows a rising edge, look for a rising edge on the CLK input and then rise the output concurrently.

Rule 2 If RN shows a falling edge, pull the output to ground.

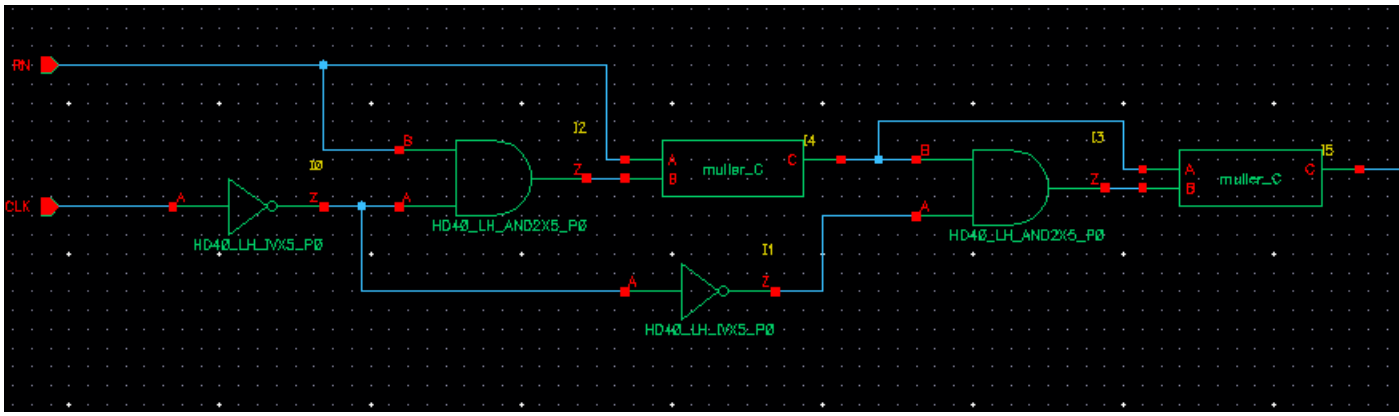
What makes the device challenging to make is that it is neither fully asynchronous nor fully clocked. It's rising edge characteristic is more like a flip flop, but it's falling edge nature shows an outright combinational structure. It turns out that the guarded C element design we discussed in the previous section will come extremely handy when combating such specifications. We have discussed the static flavor of the C element in the topics presented so far, but there are many versions for the same behavior and we use this moment as an opportunity to introduce three popular models, and their performance when used to build the above circuit.

It should be kept in mind as a disclaimer that the current circuit is designed while being agnostic to the operational frequency, and thus it is not mandatory that the device will replicate the behavior with the exact timing properties at high clock rates. It was specified during design that the flop will be used to power subsystems from a shut-down phase, and thus would be operating at relatively slow clock speeds.

3.3 The Static Wake Up Flop

The static version of the flop uses the Seitz static design of the C element, which, to recapitulate, uses a weak feedback inverter for latching at unequal inputs. The design is done by placing the guarded C element in a **master-slave** architecture, and this is done to satisfy Rule 1. It can be noted that the guarded gate is actually a level sensitive latch of-sorts and would need a two stage processing to make it edge sensitive (this is actually how a D flip flop is designed from two D latches/two inverter loops).

The confirmation of Rule 2 in the behavior comes as a side-effect of organizing the guarded C element in the architecture. When the RN line is pulled down, it is to be noted that it overrides all CLK inputs and gets the output to zero. This implies that the RN signal represents the high priority input in this case, and from the block diagram shown below it can be seen how this property conveniently fits with the architecture.



We replace the block labelled `muller-C` with the corresponding style of the C element and then simulate the resulting circuitry. The following tools were used during the simulation process

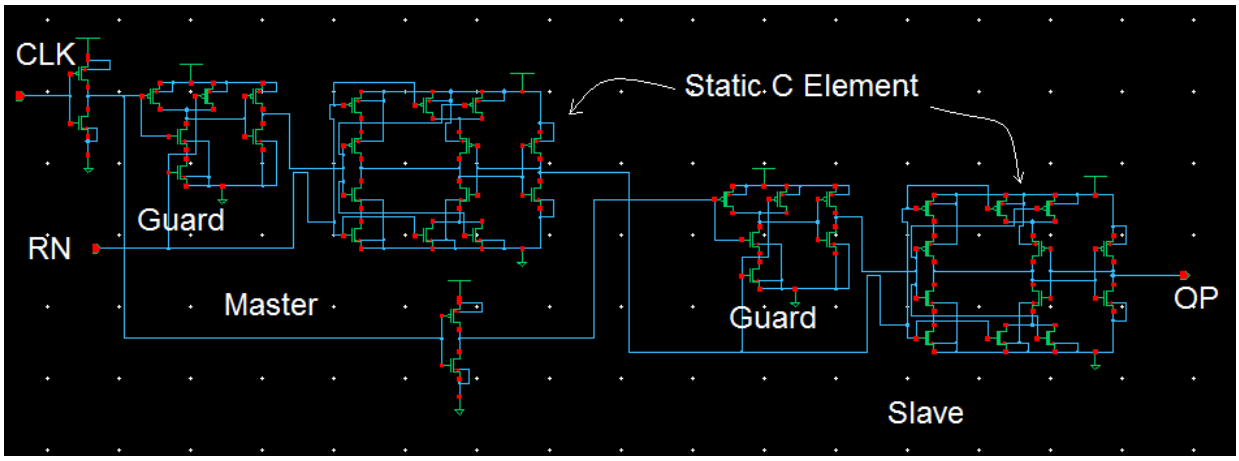
Design Suite Cadence Virtuoso Custom IC Design Environment - Version IC6.1.6-64b.500.13.2

Schematic Editor Virtuoso Schematic Editor L - Using ST's CMOSM40 40nm CMOS Process and HVT Transistors.

Simulator Eldo SPICE running on a RHEL 5.9 Server.

Waveform Viewer EZwave 16.4 Patch 1

The schematic looks as follows:



The device is simulated with the following parameters:

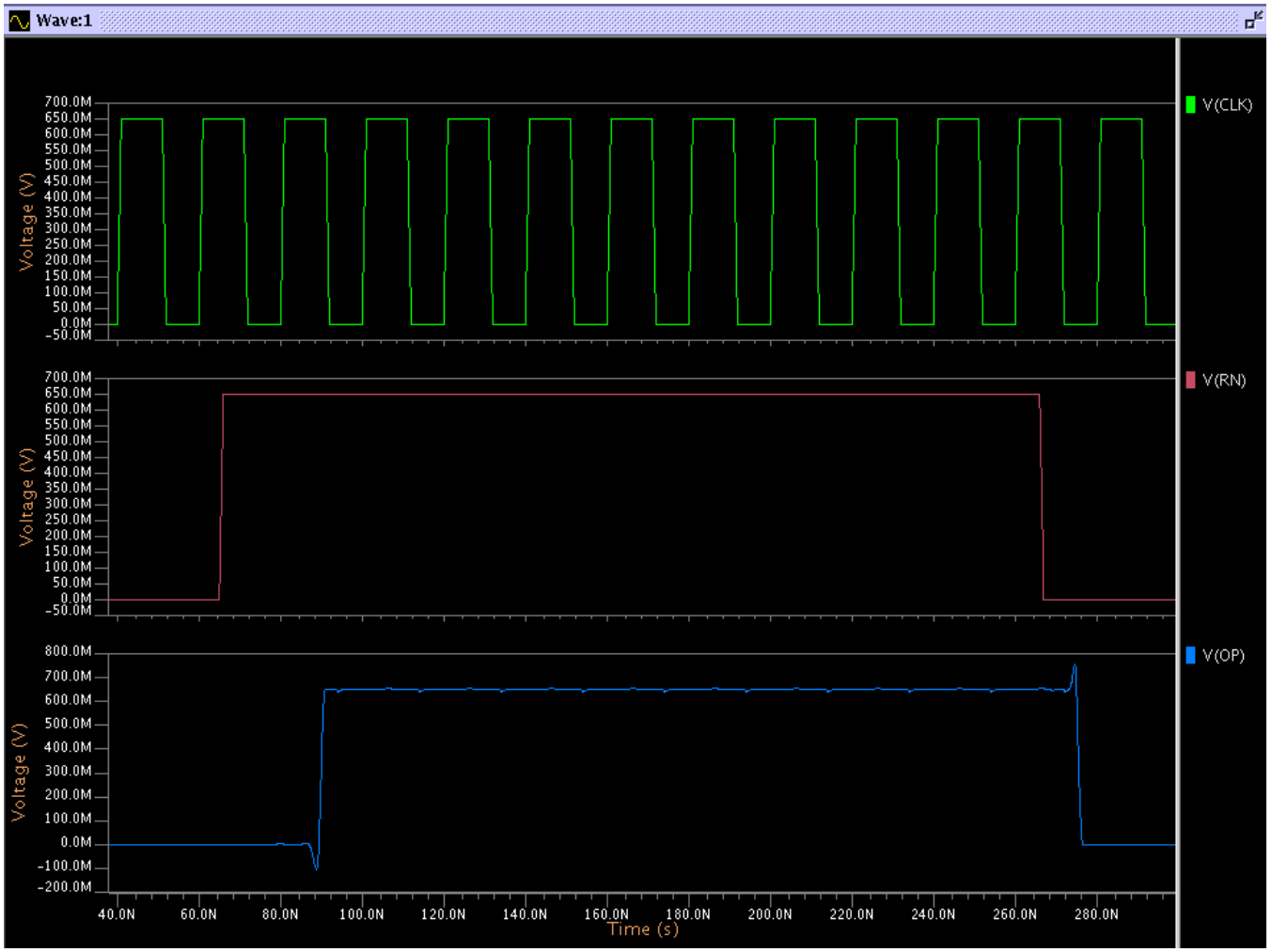
CLK The time period for the perioidic CLK input is taken as 20ns. This is much larger than the nominal frequency at which the actual device will be running (which is around a MHz or so, more on this after a couple of sections), but this value was chosen for the sake of discussion. The rise and fall times are taken to be 1ns for functional simulation. The amplitude is 0.65V.

RN The RN pulse will be much slower than CLK, and it is taken as a periodic square pulse of 400ns as the time period. The signal is initialized with a delay of 65ns to simulate the asynchronous nature of the trigger. This signal also has the rise and fall times fixed at 1ns. The amplitude is 0.65V.

Simulation A transient simulation is launched for 1000ns, with a 1ns timestep at the typical corner of 25 degrees Celsius.

It should be noted that the fall time for the device will be faster when compared to the rise time, and this is simply due to the propagation delays in the paths that lead to the rise and fall. For the output to be pulled to ground, the RN pin is reduced to zero, and after two AND gates and two C elements (both of them are latching a "0" and technically speaking, the latching time for a "0" is actually faster than latching a "1") the output is brought down to zero. When the RN pin goes high, the master-slave property of the device makes it wait for the positive edge and then adds the zero latching delay to this to push the output to V_{DD}.

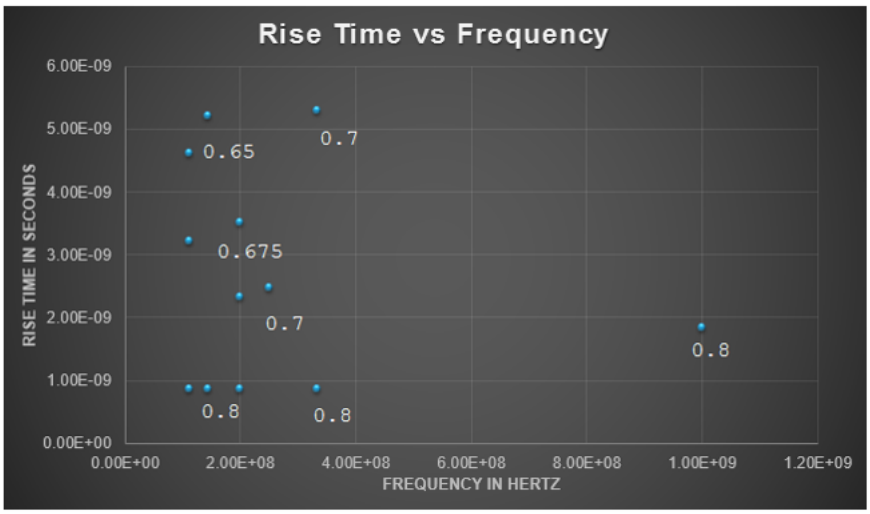
The specimen waveform for the static C element is shown below (for a restricted window):

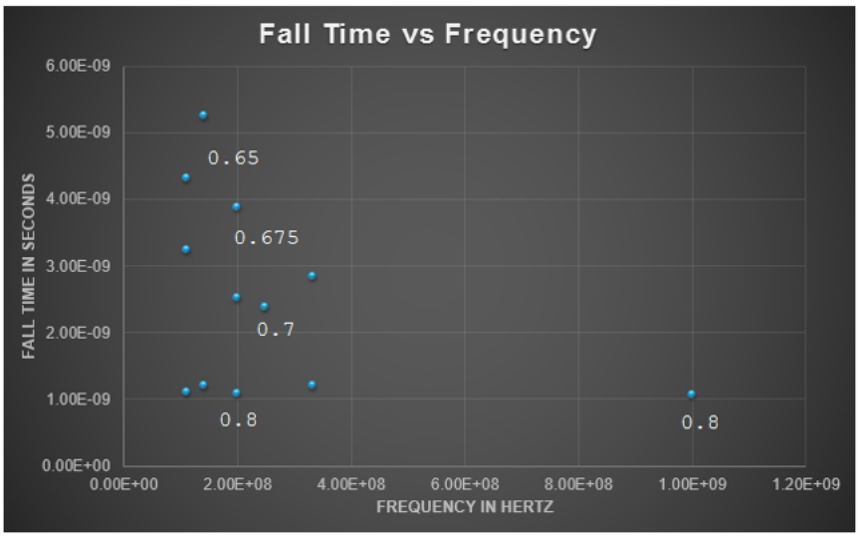


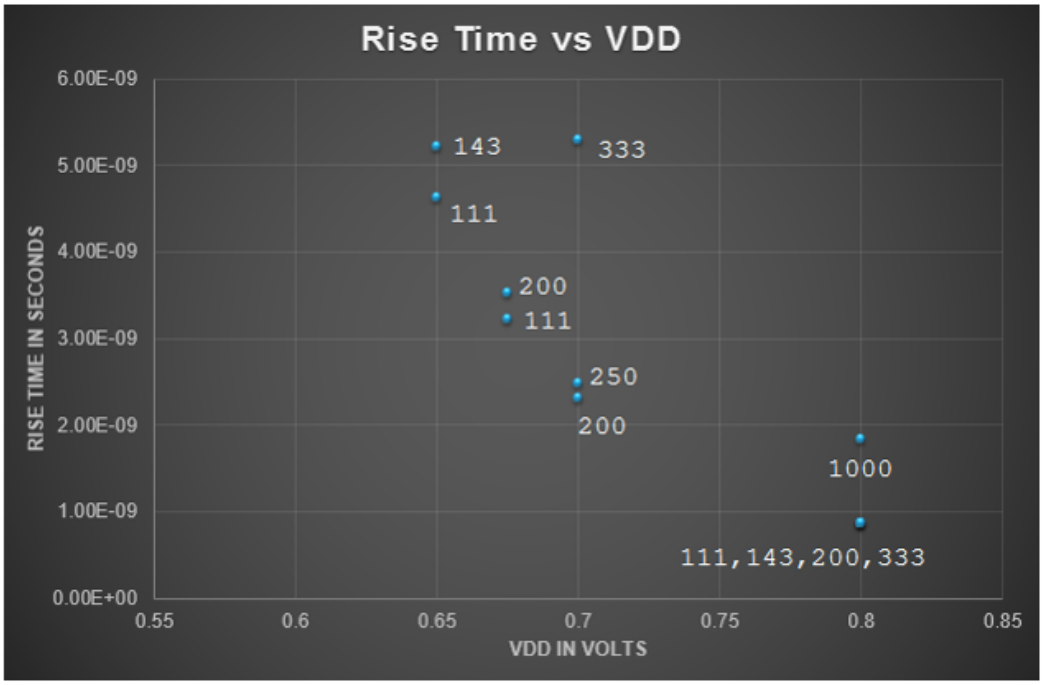
It can be seen here that the latching of logic 1 actually takes around half the clock cycle to happen. That is, after the positive edge, the device takes nearly 10ns to pull the output to high. This is not so bad if the reader recollects that the simulation is being done on 40nm FETs whose V_T is around 0.5V, and the peak voltage we are operating at is just 150mV above the same. This fact can be confirmed from the fall time, which roughly equals the rise time, contrary to our expectation of it being a noticeable factor lower than the latter.

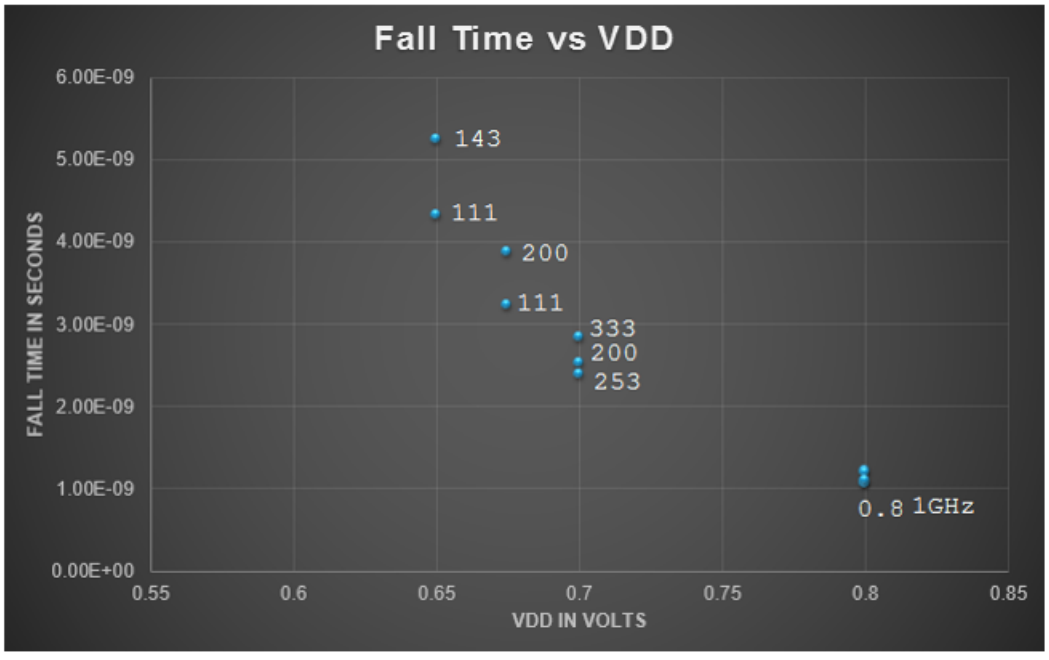
3.4 Important Metrics - Static Flavor

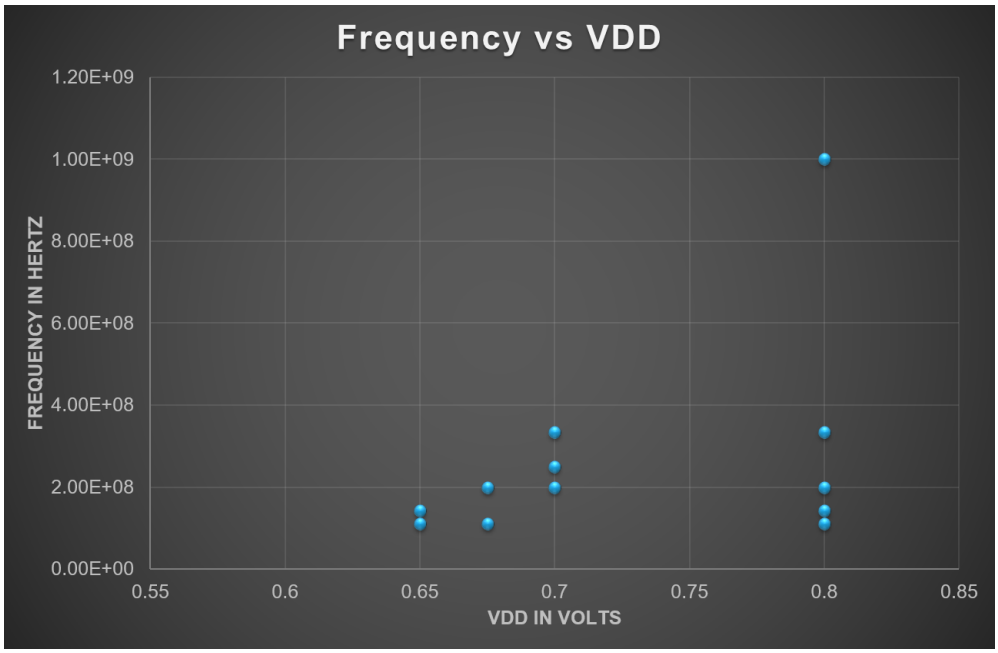
All numbers on graphs indicate the conjugate metric being held constant. For instance, on a frequency dependent graph, the number represents the core voltage and vice versa. All core voltages are in Volts and the frequencies are assumed to be in MHz unless specifically mentioned.





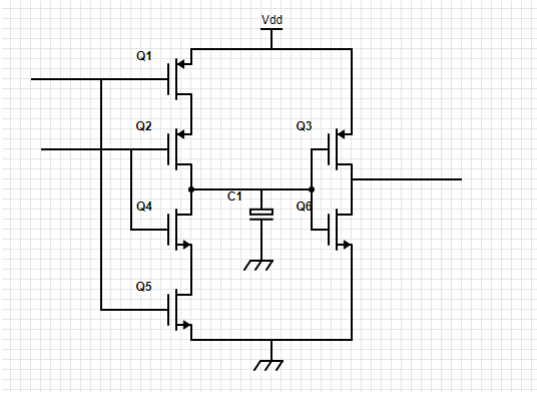

Rise Time vs VDD
RISE TIME IN SECONDS
6.00E-09
5.00E-09
4.00E-09
3.00E-09
2.00E-09
1.00E-09
0.00E+00
0.55
0.6
0.65
0.7
0.75
0.8
0.85
VDD IN VOLTS
143
333
111
200
111
250
200
1000
111,143,200,333


Fall Time vs VDD
FALL TIME IN SECONDS
6.00E-09
5.00E-09
4.00E-09
3.00E-09
2.00E-09
1.00E-09
0.00E+00
0.55
0.6
0.65
0.7
0.75
0.8
0.85
VDD IN VOLTS
143
111
200
111
333
200
253
0.8 1GHz



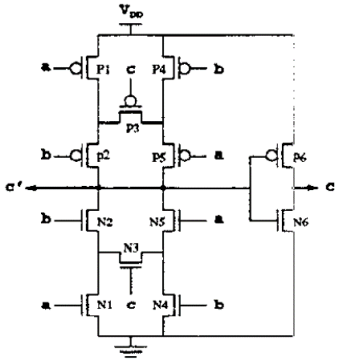
3.5 Other Flavors

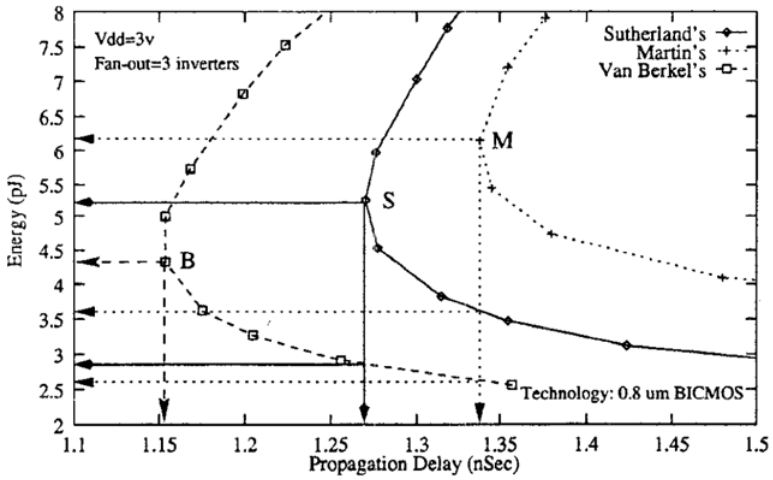
One can replace the static C element (Seitz's C element) with the dynamic or the van Berkel flavor and one can obtain two different designs with important trade-offs. Firstly, the dynamic C element does away with the weak feedback scheme of the static version, but instead bases it's functionality on the capacitance of the net shown in the diagram.



The dynamic C element turns out to be the fastest of the three models,

but it trades off speed for stability due to the dependence on the internal capacitance. The van Berkel C element, discovered by the Dutch electrical engineer C. H. van Berkel, involves a clever rearrangement of the inverter loop latch in the static C element to make the gate symmetric with respect to the inputs of the C element (a problem with both the static and the dynamic versions), but does not offer any significant trade off otherwise, both area and speed wise. It is however found that the van Berkel flavor offers the minimum energy delay product.

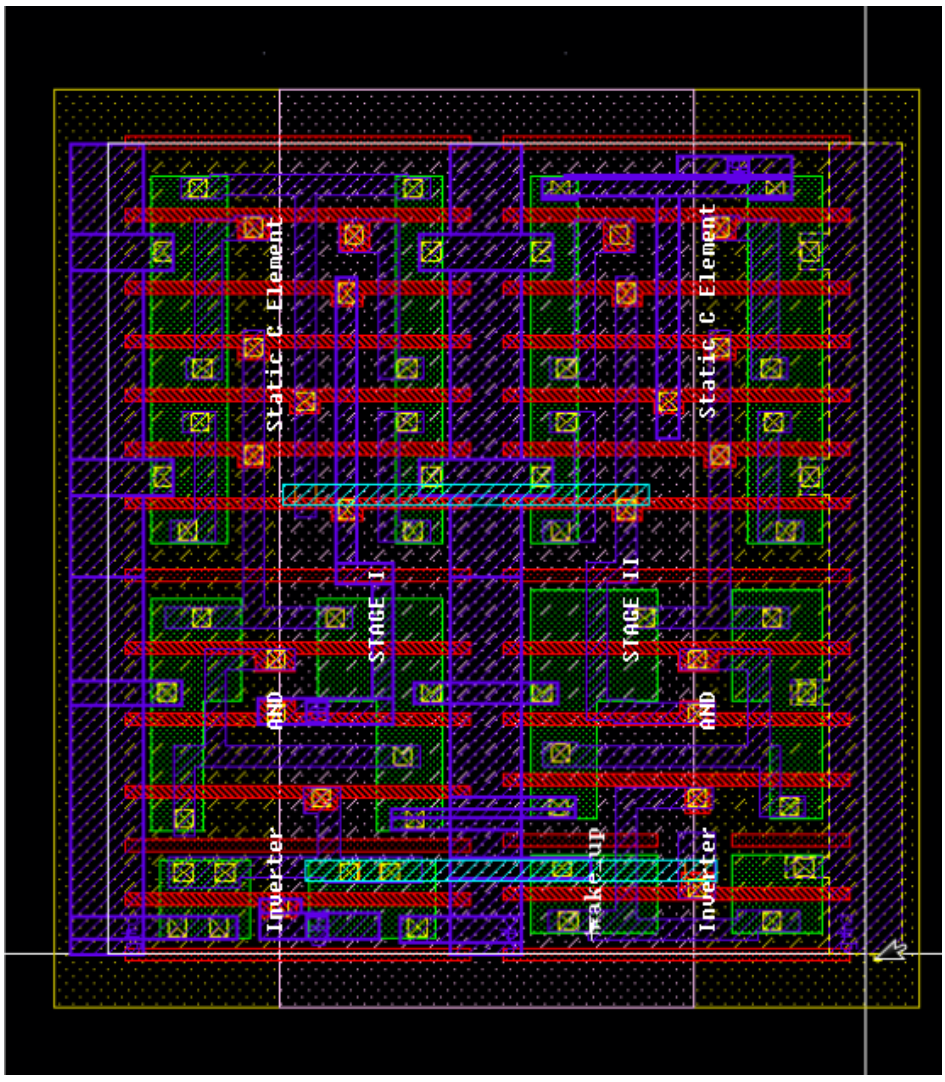




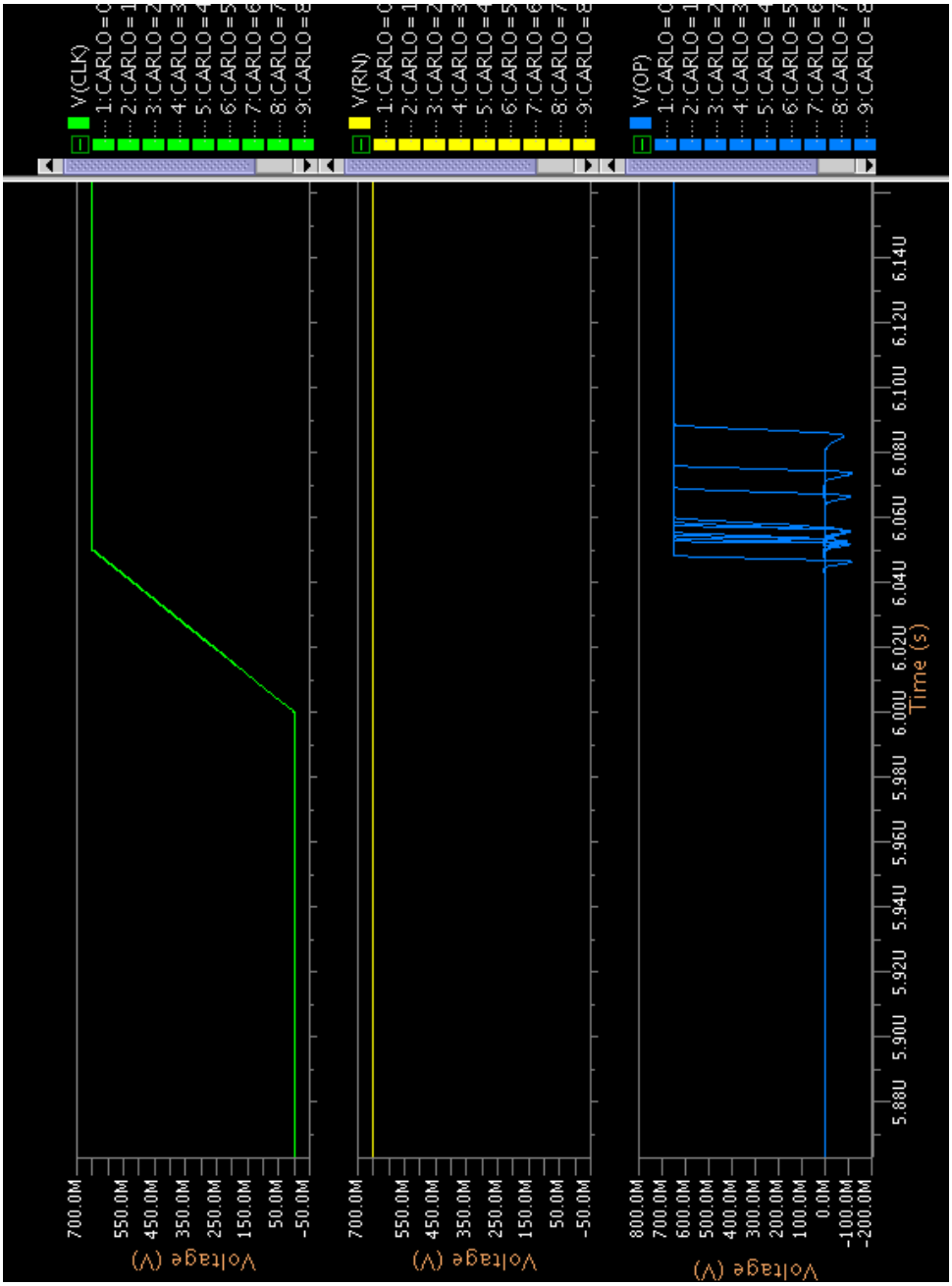
[1] A Comparison of CMOS Implementations of the C Element – Shams, Ebergen et al – ISLPED 1996

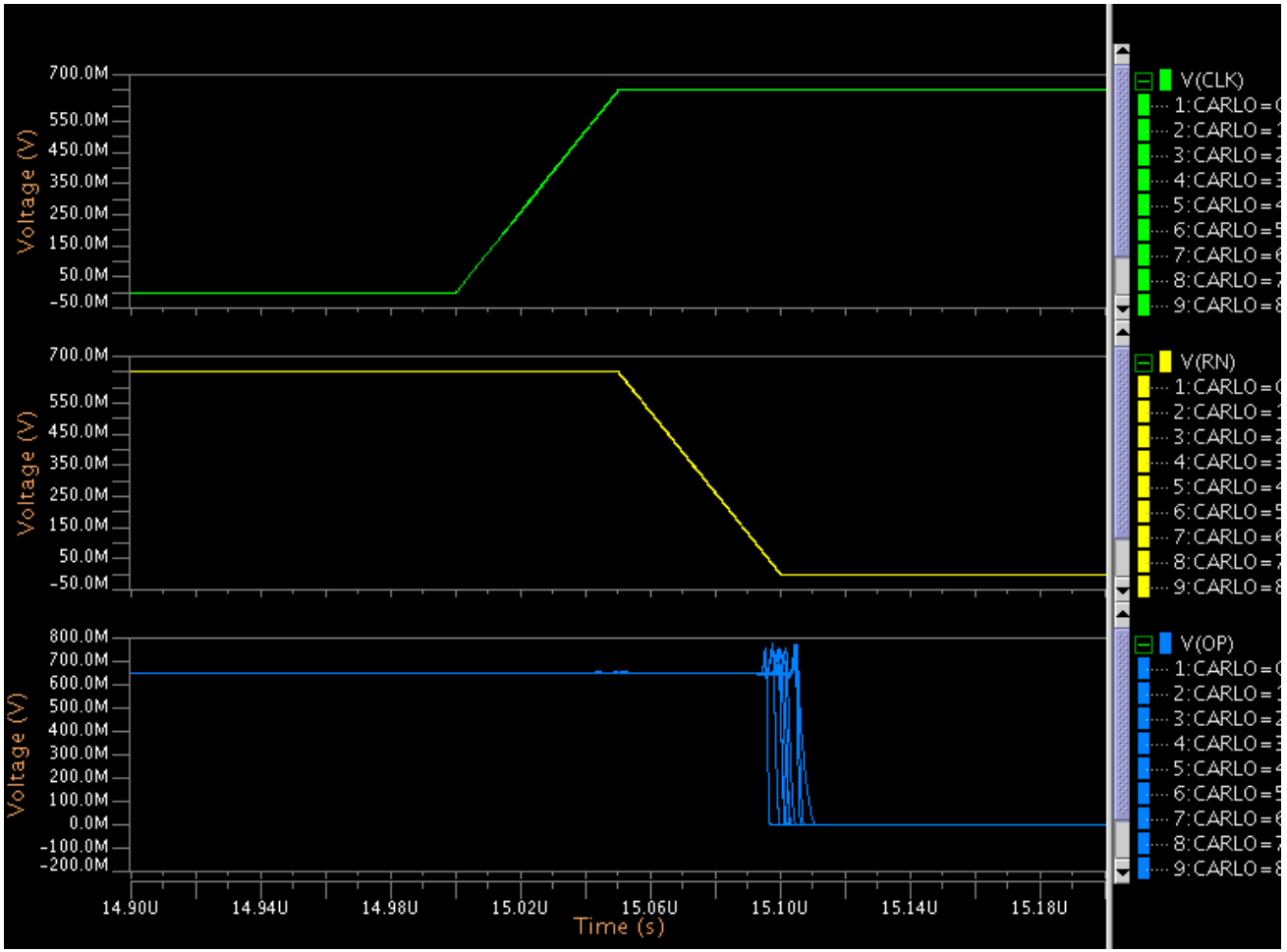
Layout for The Static Flavor

June 12, 2017

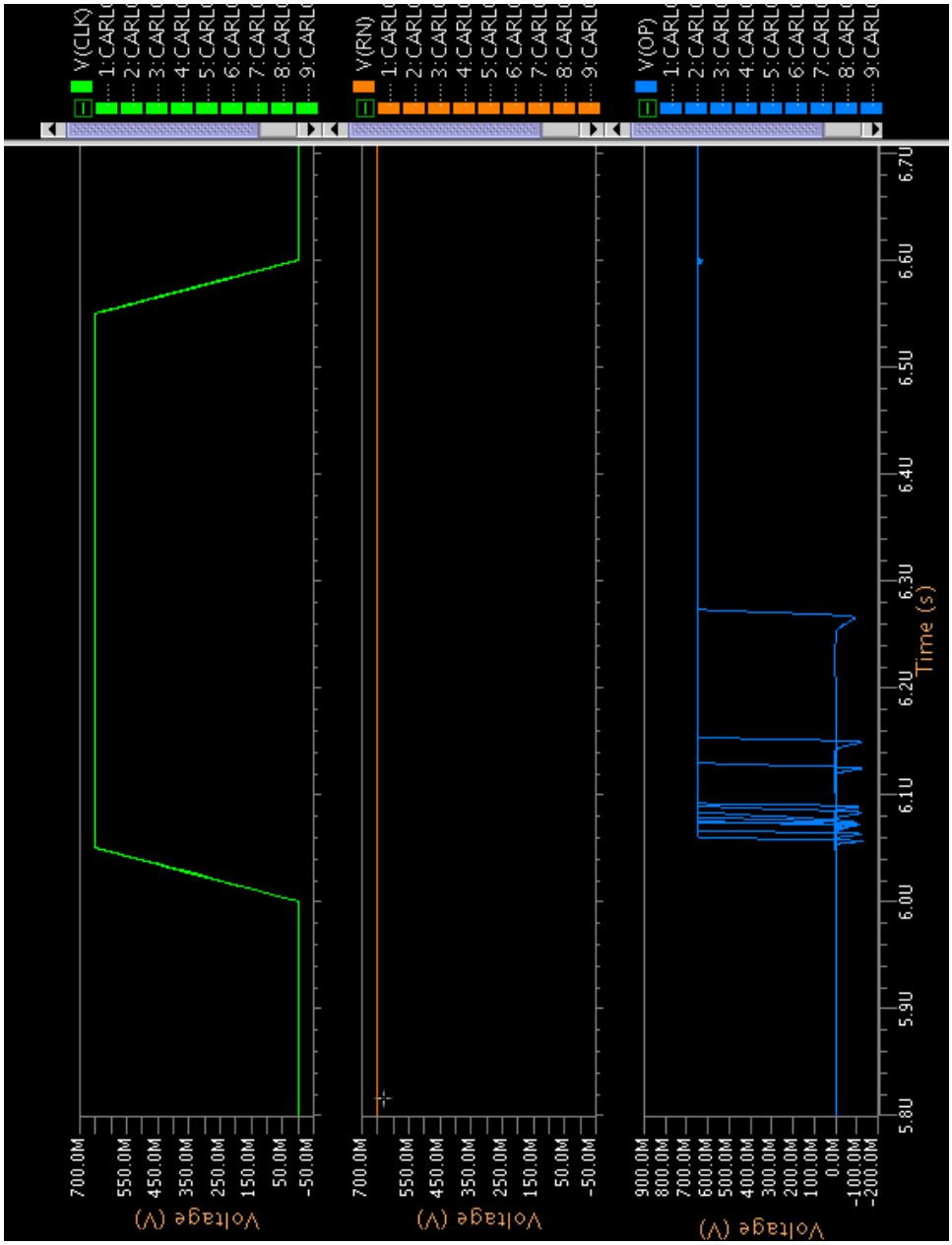


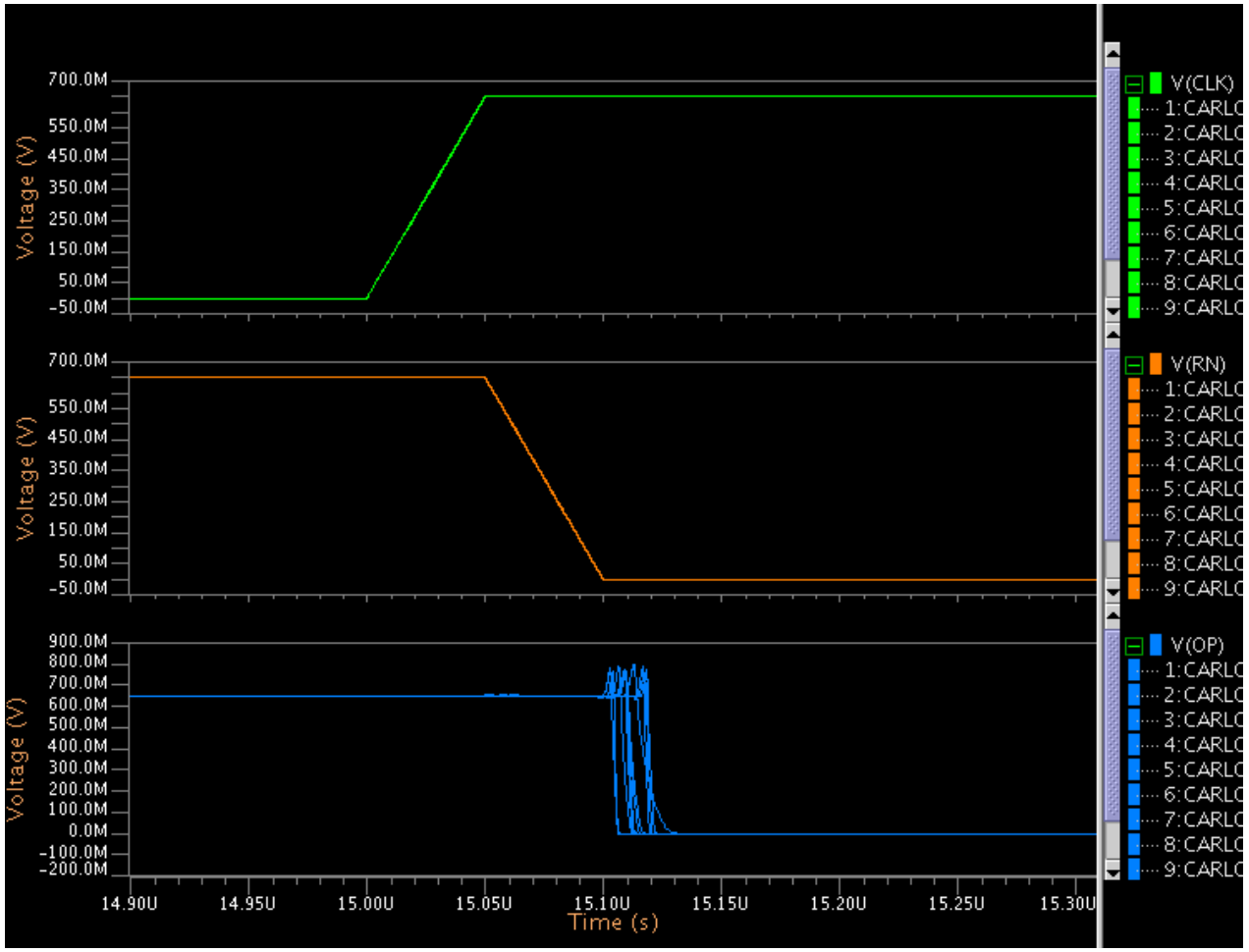
Post Layout Simulation at 25 Degrees Celsius



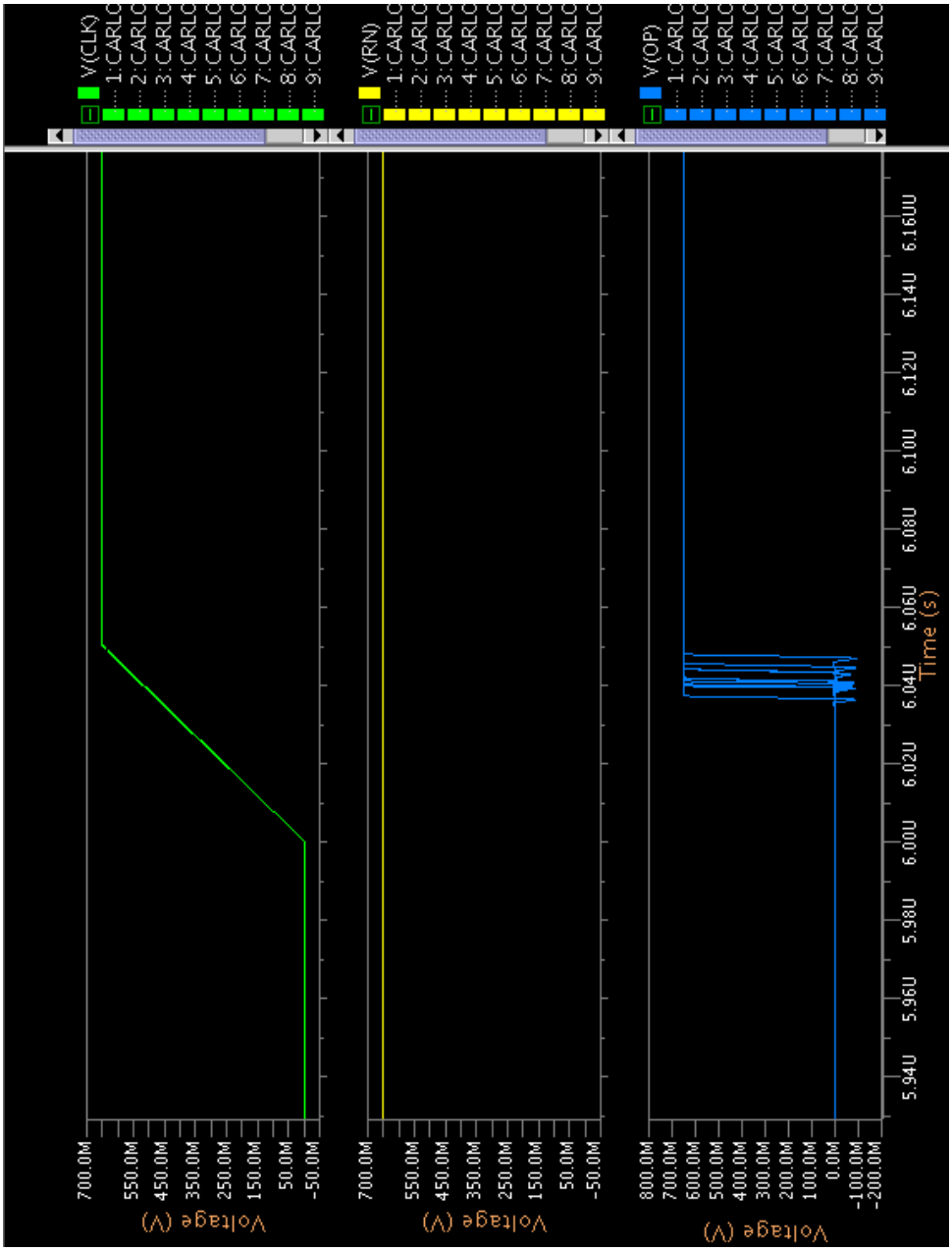

700.0M
550.0M
450.0M
350.0M
250.0M
150.0M
50.0M
-50.0M
Voltage (V)
V(CLK)
1:CARLO=0
2:CARLO=1
3:CARLO=2
4:CARLO=3
5:CARLO=4
6:CARLO=5
7:CARLO=6
8:CARLO=7
9:CARLO=8
700.0M
550.0M
450.0M
350.0M
250.0M
150.0M
50.0M
-50.0M
Voltage (V)
V(RN)
1:CARLO=0
2:CARLO=1
3:CARLO=2
4:CARLO=3
5:CARLO=4
6:CARLO=5
7:CARLO=6
8:CARLO=7
9:CARLO=8
800.0M
700.0M
600.0M
500.0M
400.0M
300.0M
200.0M
100.0M
0.0M
-100.0M
-200.0M
Voltage (V)
V(OP)
1:CARLO=0
2:CARLO=1
3:CARLO=2
4:CARLO=3
5:CARLO=4
6:CARLO=5
7:CARLO=6
8:CARLO=7
9:CARLO=8
14.90U
14.94U
14.98U
15.02U
15.06U
15.10U
15.14U
15.18U
Time (s)

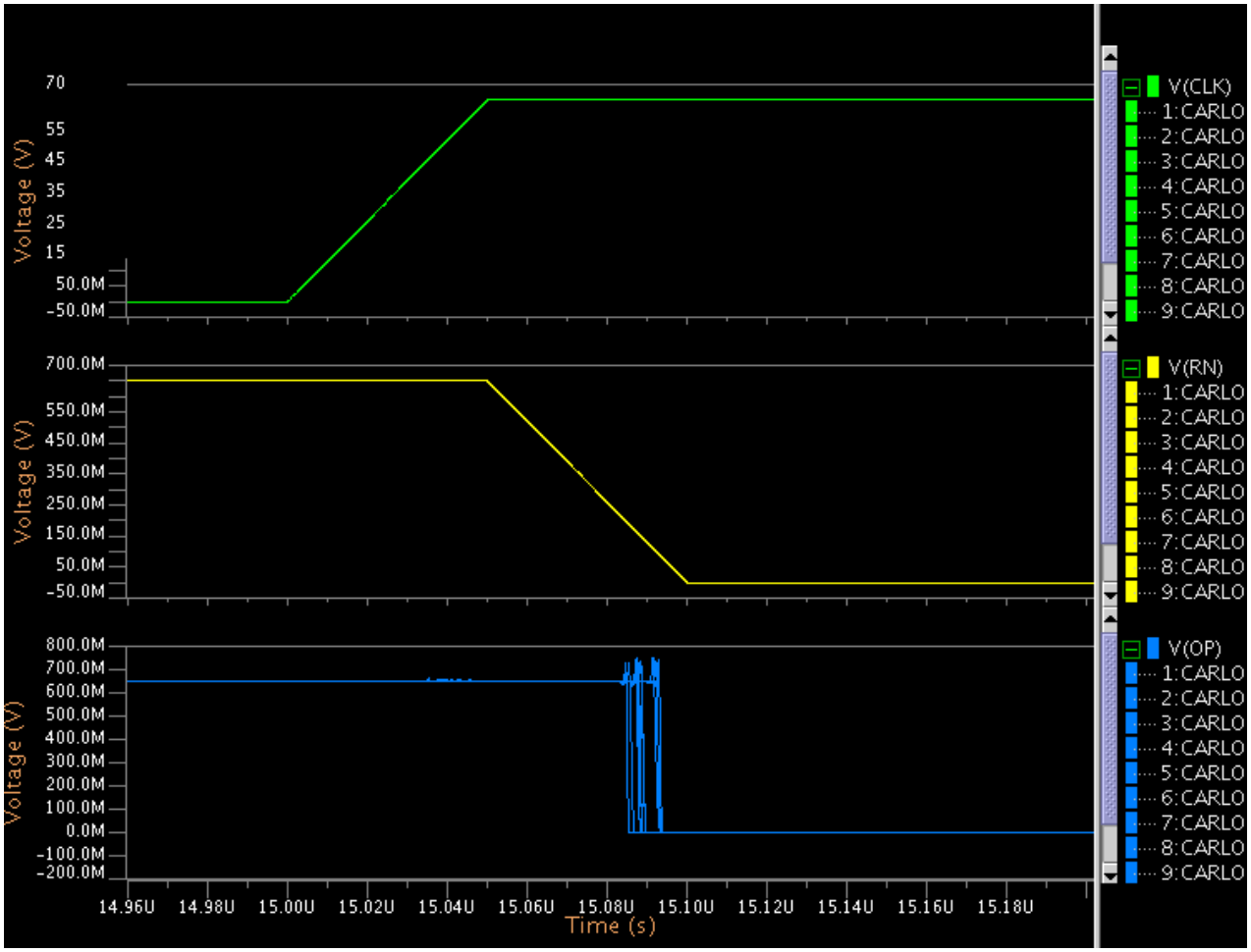
Post Layout Simulation at - 40 Degrees Celsius




700.0M
550.0M
450.0M
350.0M
250.0M
150.0M
50.0M
-50.0M
Voltage (V)
V(CLK)
1:CARLO
2:CARLO
3:CARLO
4:CARLO
5:CARLO
6:CARLO
7:CARLO
8:CARLO
9:CARLO
700.0M
550.0M
450.0M
350.0M
250.0M
150.0M
50.0M
-50.0M
Voltage (V)
V(RN)
1:CARLO
2:CARLO
3:CARLO
4:CARLO
5:CARLO
6:CARLO
7:CARLO
8:CARLO
9:CARLO
900.0M
800.0M
700.0M
600.0M
500.0M
400.0M
300.0M
200.0M
100.0M
0.0M
-100.0M
-200.0M
Voltage (V)
V(OP)
1:CARLO
2:CARLO
3:CARLO
4:CARLO
5:CARLO
6:CARLO
7:CARLO
8:CARLO
9:CARLO
14.90U
14.95U
15.00U
15.05U
15.10U
15.15U
15.20U
15.25U
15.30U
Time (s)

Post Layout Simulation at 125 Degrees Celsius




70
55
45
35
25
15
Voltage (V)
50.0M
-50.0M
V(CLK)
1:CARLO
2:CARLO
3:CARLO
4:CARLO
5:CARLO
6:CARLO
7:CARLO
8:CARLO
9:CARLO
700.0M
550.0M
450.0M
350.0M
250.0M
150.0M
50.0M
-50.0M
Voltage (V)
V(RN)
1:CARLO
2:CARLO
3:CARLO
4:CARLO
5:CARLO
6:CARLO
7:CARLO
8:CARLO
9:CARLO
800.0M
700.0M
600.0M
500.0M
400.0M
300.0M
200.0M
100.0M
0.0M
-100.0M
-200.0M
Voltage (V)
V(OP)
1:CARLO
2:CARLO
3:CARLO
4:CARLO
5:CARLO
6:CARLO
7:CARLO
8:CARLO
9:CARLO
14.96U
14.98U
15.00U
15.02U
15.04U
15.06U
15.08U
15.10U
15.12U
15.14U
15.16U
15.18U
Time (s)

Chapter 4

The Synchronizer

'Right time, right place, right people equals success. Wrong time, wrong place, wrong people equals most of the real human history.'

Idries Shah, Reflections, 1978.

4.1 Prelude

This article deals with the age old synchronizer problem and offers a new approach to handling the passage of data from one clock domain to the other. The problem has been analyzed many times and there exists enough literature outside (see [1] for fourteen different solutions) to get the reader up to speed and it is assumed that he/she has a basic knowledge about timing constraints and static timing analysis (an introduction can be found in [2]). The inspiration for the current flavor of the synchronizer comes from the seminal work on the technology of GALS interconnect (see [3]) where the synchronization process is mediated via an asynchronous medium, and thus facilitates the handling of multiple clock domains simultaneously. It is hoped that the scheme discussed in this article leads to a refreshing take on the problem of clock domain crossing via an intermediate clock synthesis.

4.2 Problem Statement

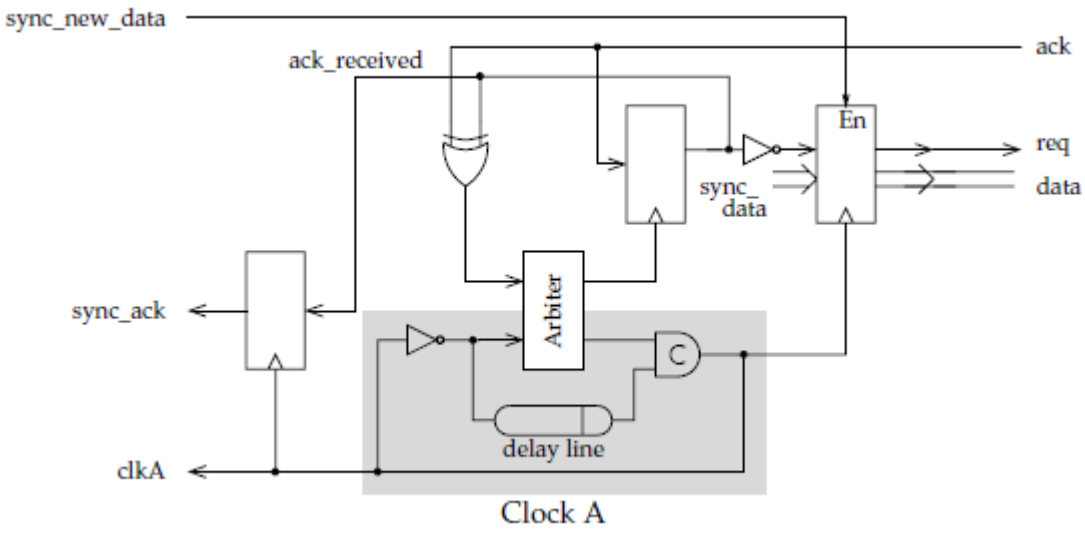
We are provided two clock domains A and B, and we would like to transfer data from A to B (push synchronization). It is to be emphasized beforehand that we are strictly dealing with a unidirectional bus. By clock domain, we simply refer to any subsystem of an electronic device which has its own clocking scheme/local clock. Due to the exponential increase in the on chip clock rates, it has become cumbersome to handle clock skews for every such instance on the circuit. Instead, the whole device is now split up into multiple clock islands and these are interfaced via a GALS (Globally Asynchronous Locally Synchronous) Interconnect (refer [4], [5]).

Our task is to design a module/procedure to safely allow latching of data from A to B. We do not impose the strict rule of positive edge synchronization, wherein one has to ensure that the data latched onto the posedges of the flop A match the corresponding posedges of B (and it is known that for the slower clock having a frequency lesser than half the faster clock there would inevitably be data losses), but we would obviously want to minimize the data losses during crossing. We represent the clocking in domain A as CLKA and in domain B as CLKB. It is not necessary that both A and B have the same frequencies, and the consequences of clock stretching are discussed later.

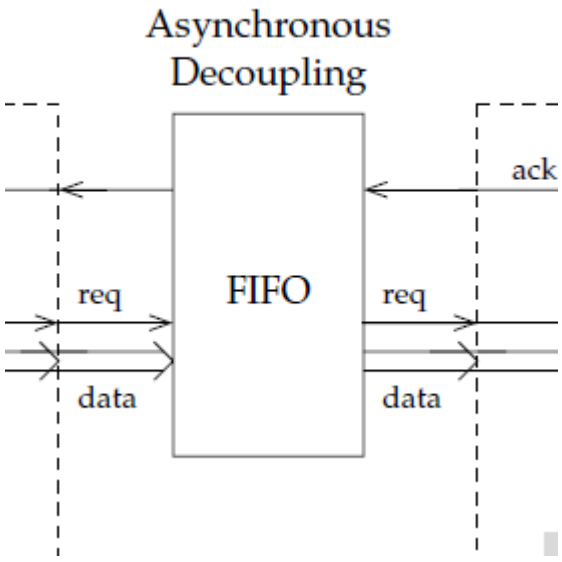
4.3 Clock Pausing

The paper [3] deals with the problem of designing an efficient interconnect strategy for a GALS system by constructing a Point-to-Point mechanism, in which the domain crossing is handled in three stages operating in a pipeline. For the sake of discussion, we shall call the domain A as the producer domain and B as the consumer domain. Most of the traditional synchronizer strategies deal with the producer and the consumer having a common data bus and handle the issues with latching by manipulating the control signals of either domain. However, in the Point-to-Point mechanism data is always transferred to an intermittent asynchronous medium and then latched safely at the consumer end. The three stages of transmission involve:

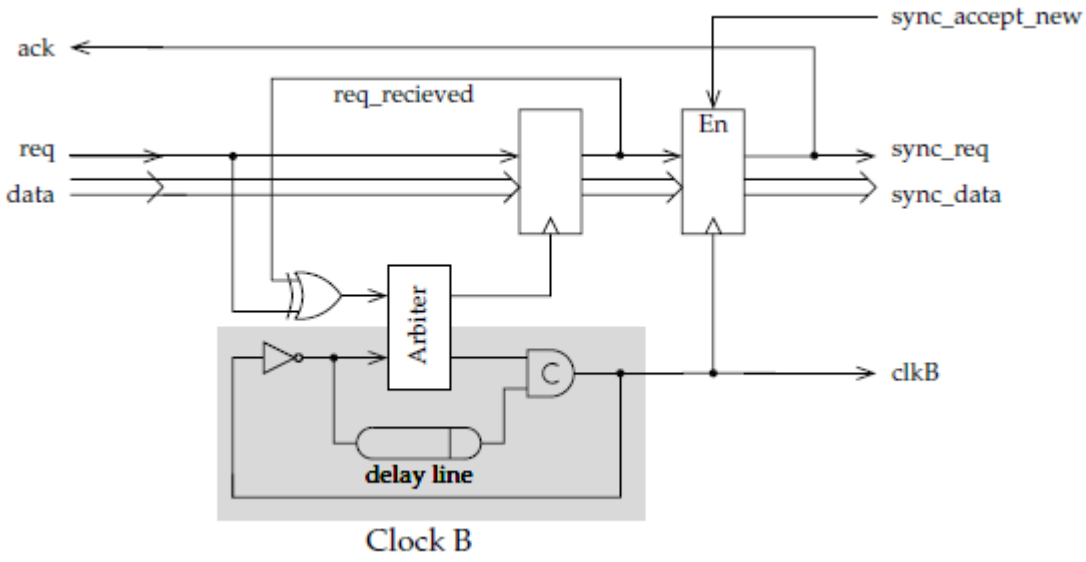
a. The synchronous producer handing data over to the asynchronous medium. This is done by the module shown below (taken from [3])



b. The asynchronous medium transmitting data consistently



c. The synchronous consumer latching the data safely obeying the timing constraints of the corresponding clock domain



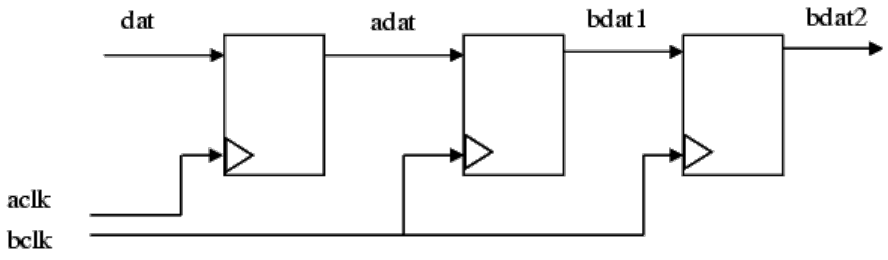
The intermediate stage of asynchronous transmission is not something that we will be dealing with in detail and one can program standard SRAM units to behave as a FIFO element for operational purposes. The interesting point

to be noted lies in the transmission of the data from a synchronous domain out into the asynchronous space, and it can be seen that both the stages somewhat operate by creating their own local clocks by using an inverter driven oscillator (as shown by the gray shading).

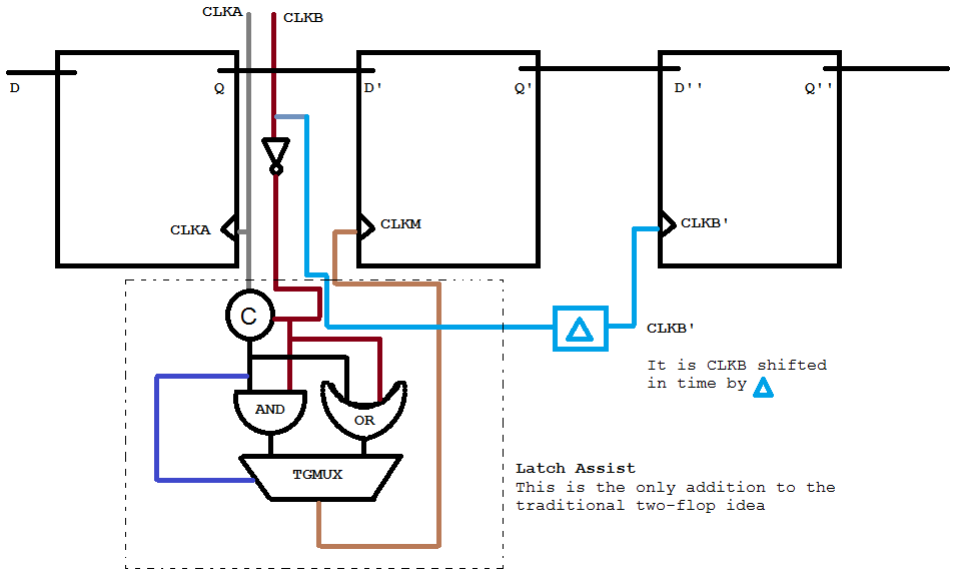
The obvious drawback with the Point-to-Point scheme is the amount of space consumed by the whole mechanism, and this not only involves the modules presented above, but also the control circuitry for the delay lines present in these blocks. The delay lines used in [3] are precise tunable delays which are explained in [6] written by the same authors. Thus, we seek a solution which is compact enough to be made dense, but flexible enough to handle difficult situations such as fast-to-slow clock synchronization and so on.

We similarly construct a clock signal CLKM from the independent clocks CLKA and CLKB (CLKM is local to CLKB domain, as is the convention in synchronizer design). The principle behind the construction of CLKM is **to cause a premature latching to happen so that by the time CLKB gets to latch the data, it is already stable**. This can be best understood by recalling that for every clocked system, there exists a metastability window given by the sum of the setup and hold times around the rising edge of the clock. We are making sure that there is stable data available by the time the window arrives by causing a premature transition via CLKM.

To put it simply, our design is a hybrid of the more popular two flop synchronizer and the GALS interconnect idea (from which the clock synthesis/pausing/recovery idea is borrowed from). We improve the latching of the first flop in the commercial two flop synchronizer by providing it with a clock of its own (CLKM) instead of CLKB and making sure that the assisted latching has enough time to give stable (non-oscillatory) bus lines. For reference, a two flop synchronizer is shown below:



The modified architecture is shown below



4.4 Latch Assist

The latch assist circuitry combines the detection of the correct latching on the producers side with the timing of the approaching positive edge on the consumers side. When the producer and the consumer have the same frequency, the CLKM generation can be handled with the C element and the AND logic alone but in a general setting where (bounded) clock stretching might be possible, due to the inherent frequency separation, beats are produced which sometimes lead to concurrent transitions on the input lines of the C element. Due to the inertia property of the C element, these concurrent transitions (such as $01 \rightarrow 10$) lead to the same data being latched repeatedly a few cycles before showing a transition. The addition of the OR equivalent and the multiplexor resolve this issue by checking for the appropriate latching before the concurrent transition (this is done by the select line of the multiplexor, which transitions just as the output of the C element appears) and then selecting the corresponding channel.

It is rightfully claimed in [7] that in principle there cannot exist a bounded-time metastability detector, and the proof relies on the truth of the fact that there cannot exist a failsafe synchronizer. The correct way to look at the

situation involves the study of what is called as the Choice Uncertainty Principle (see [9]) or more classically known as Buridans Principle (see [10]) after the 12th century French philosopher Jean Buridan. It is more likely that this principle leads to the non-existence of a metastability detector, and it is interesting to note that the concept not only applies to microscopic systems, but equally well to macroscopic beings ([10] gives ample examples to support this).

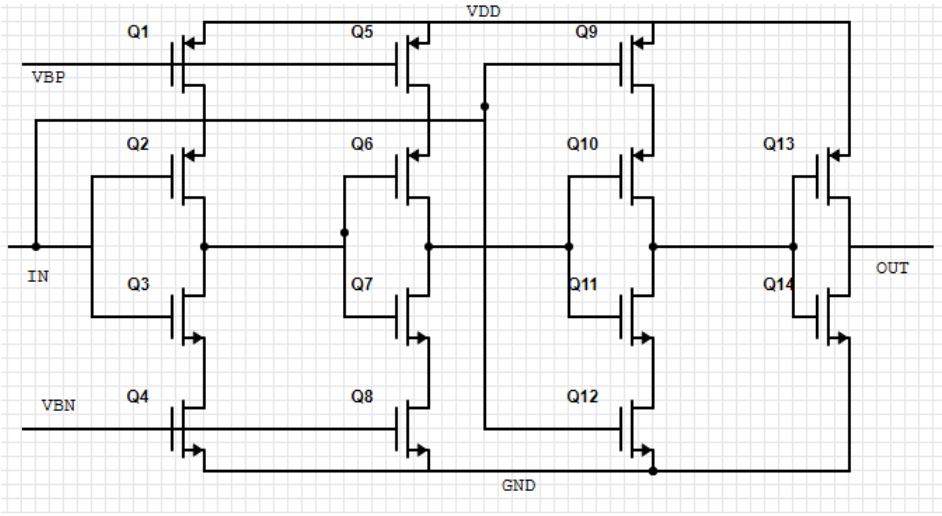
Buridans Ass is a phrase that refers to a donkey placed equidistant between two piles of hay. It was claimed by Buridan that the donkey, due to its inability to decide which hay stack to eat, will die of starvation eventually. Translating this to our scenario, the donkey is the flip flop and the direction in which the donkey will move depends on the probability of a transition on the data line. The mathematical formulation of the same is stated in [9], and thus rests the case of the existence of a perfect time-bound synchronizer. The snippet from the paper is given below for reference.

> *Buridan's Principle.* A discrete decision based upon an input having a continuous range of values cannot be made within a bounded length of time.

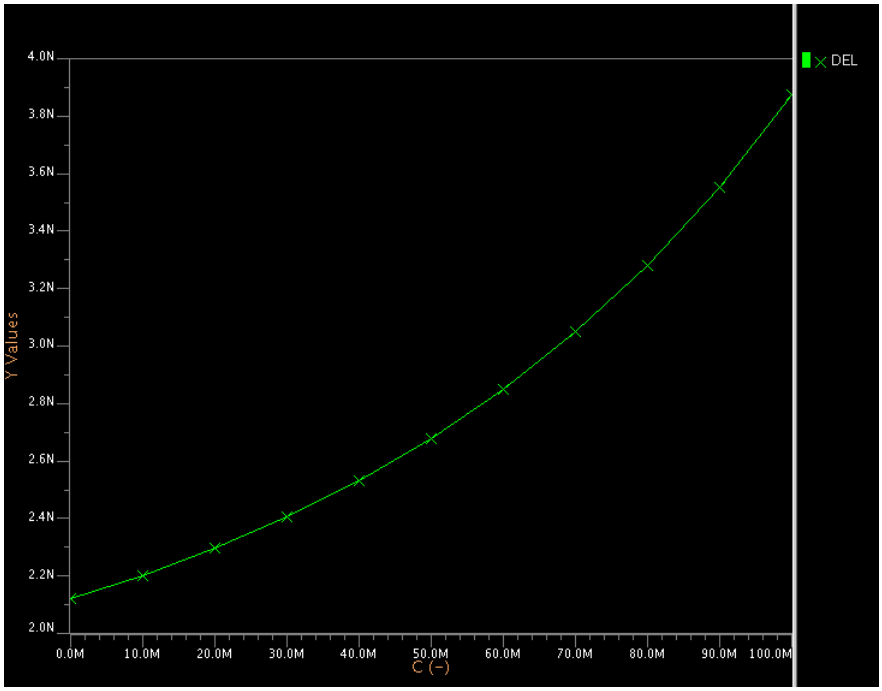
4.5 Delay Lines

The design we have presented requires the usage of a delay line and for the sake of functional testing we used a very simple inverter chain, but it was soon noted that the delay we must place there depends upon the frequencies of operation and the initial phase offset between the two clocks. The latter issue can be put to rest by phase-locking the signals before attempting to synchronize, but the former issue leads to a particularly notable side-effect, that of frequency drift.

Before we head on to that, we would like to present an alternative voltage controlled delay line which can be programmed to give a variable delay during run-time. This is based on the design given in [10], but there is a subtle change made to the structure to prevent the trimming of the output duty cycle. The design is presented in the next page along with its observably linear characteristic. The procedure is referred to as current-starving in [10]

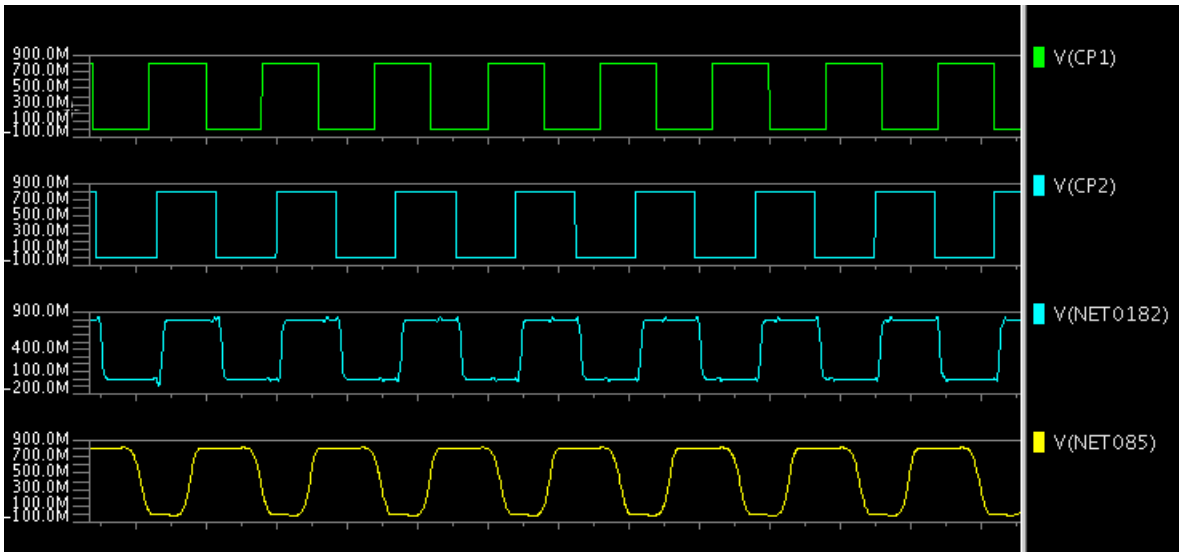


The voltages VBP and VBN are externally controlled and always sum up to VDD. The amount of delay depends upon how much you starve the upper stack transistors Q1, Q5 and the lower stack transistors Q4, Q8. We plot the delay measured for a sample square wave train versus VBP. Here VDD is taken as 0.8V and all transistors are of the 40nm CMOS process.



The linear characteristic can be noted from the graph and thus, the delay line allows calibrated increment of delay. This feature should aid a future design of the latch assist circuit in which the delayed CLK2 signal can be generated internally without any initial calibration.

4.6 Results



CP1 CLKA signal arriving at CLKB domain.

CP2 CLKB signal local to CLKB domain.

Net085 CLKB signal.

Net0182 CLKM signal.

One can see that it is essential to include the delay to avoid premature latching at CLKB positive edges (this defeats the purpose of the latch assist otherwise). In the above simulation CLKA is clocked at 125MHz and CLKB is clocked at 117MHz, a little lesser than the first domain, to demonstrate the drift phenomenon. At lower clock rates the delay element can be done away with, but when the propagation delays become comparable to the clock period, it is mandatory.

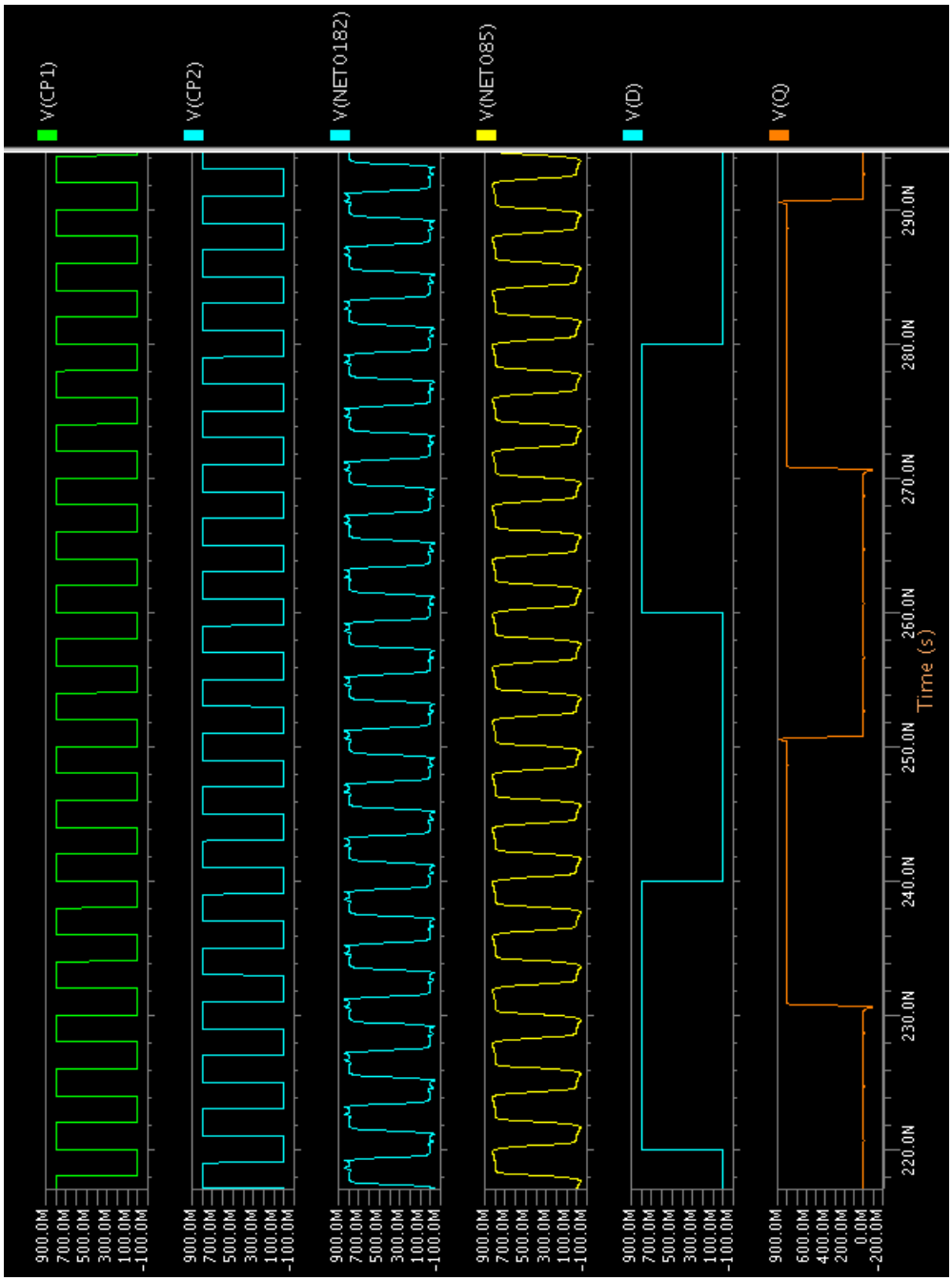
When both the clock domains have unequal frequencies, even if one does

start with a zero or a locked phase shift, the phase shift becomes a sinusoidal function of time due to the beats phenomenon. In the cases in which the frequencies are the same, but phase skews (even for extreme cases) tend to play the dominant role, it is reiterated that one can do away with the multiplexor and the OR assembly, and the device is still able to perform safe latching at frequencies around 500MHz also.

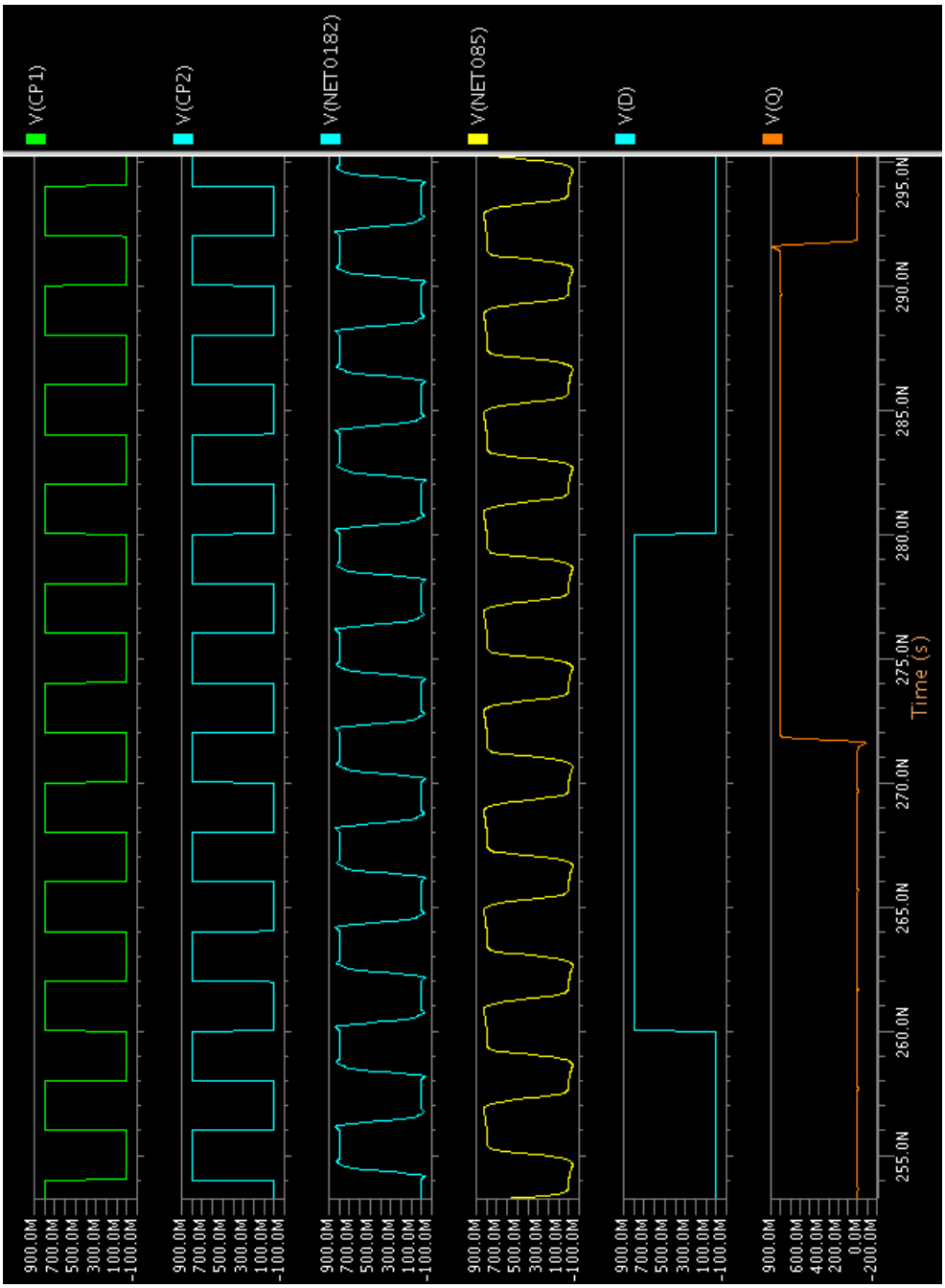
4.7 Caveat

One might argue that the clock that the data is being synchronized to ultimately is a phase shifted version of the original clock, but this is necessary to counter the effect of the frequency stretching. In essence, the device is a perfect phase skew corrector, and this fact can be used to place **another stage** of the same device to fix the added phase skew. So this is an apparent caveat.

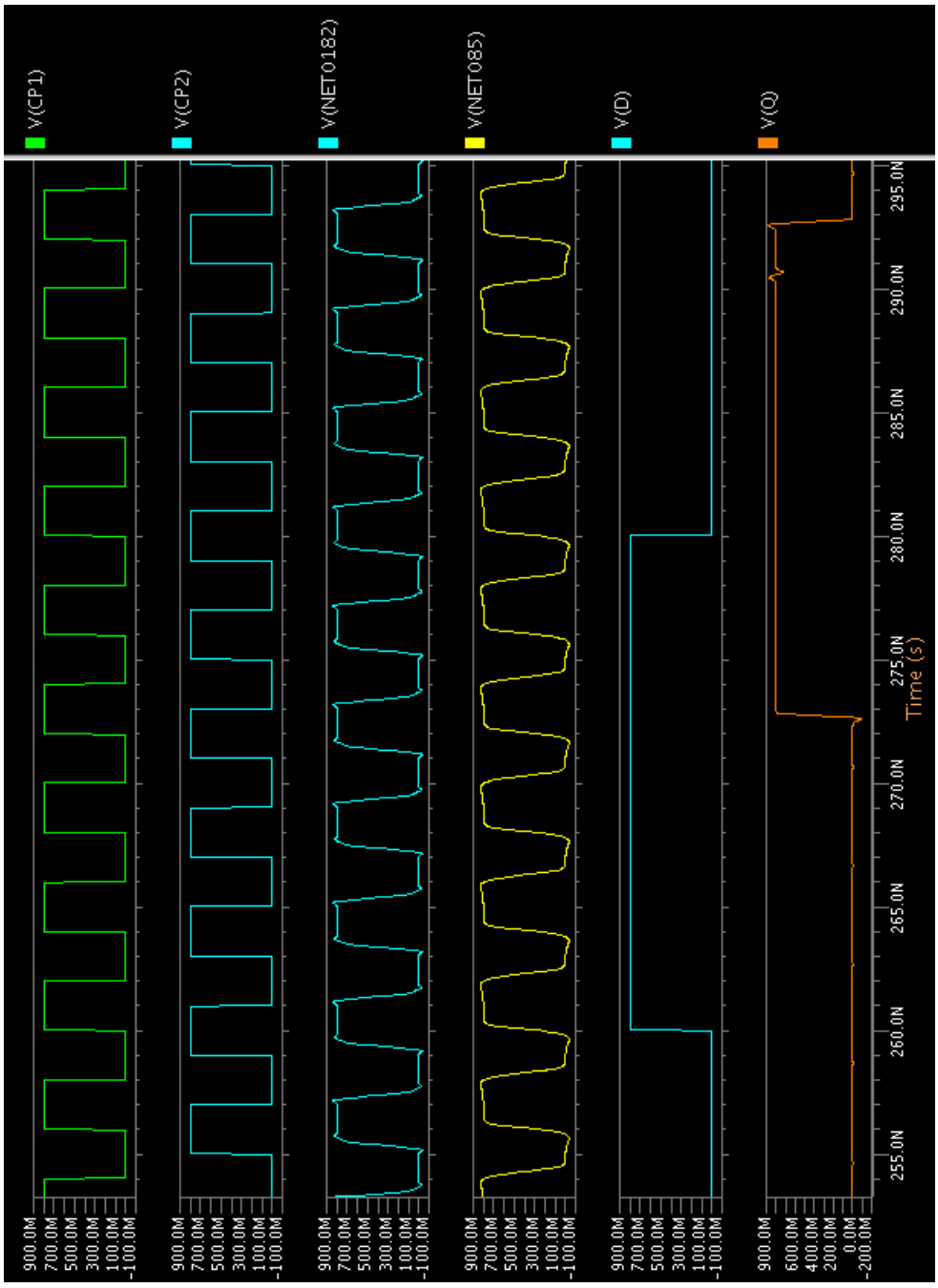
CP1 and CP2 at 250MHz. Phase shifted by a quarter of the cycle.



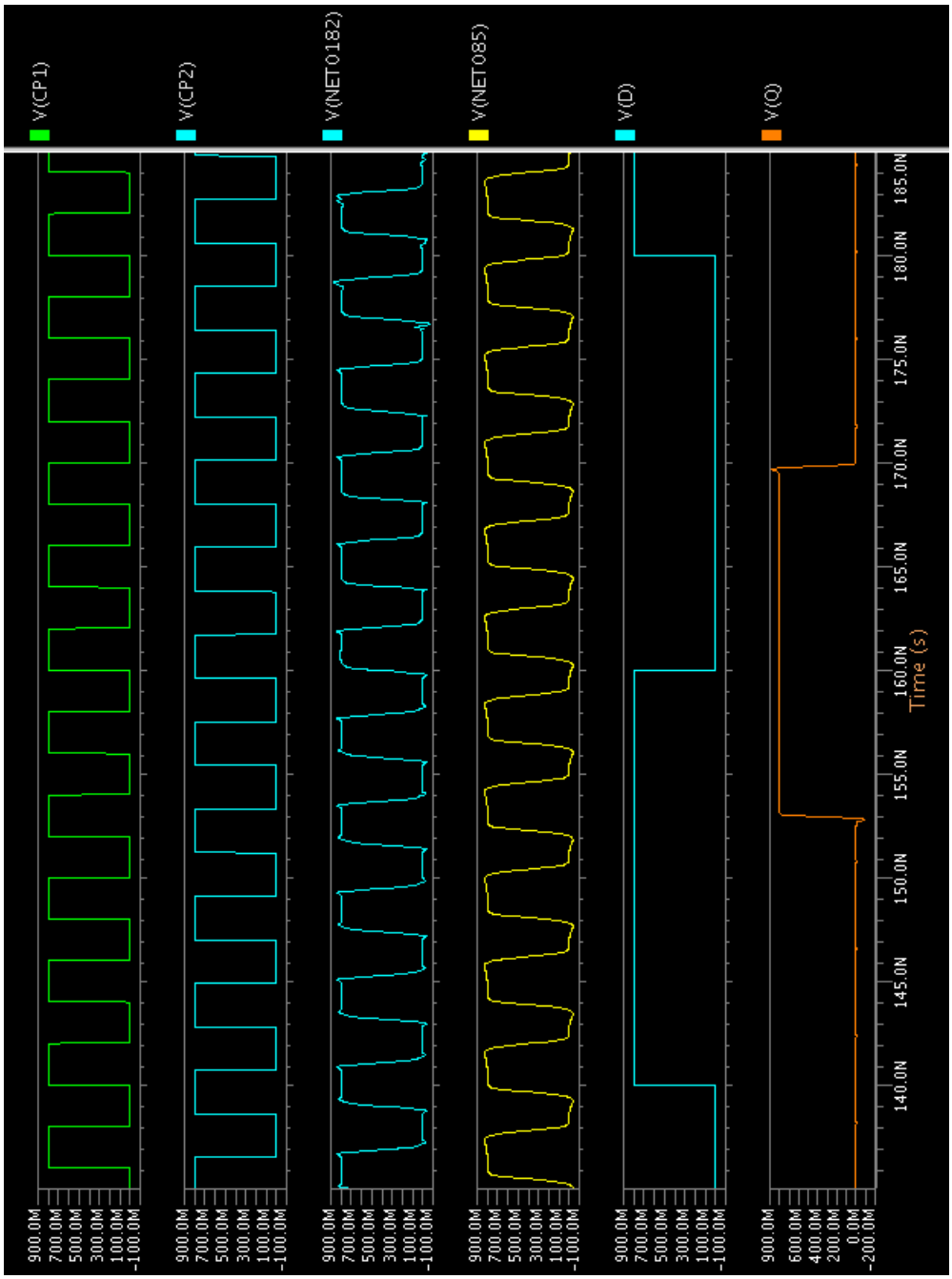
CP1 and CP2 at 250MHz. Phase shifted by a half cycle.



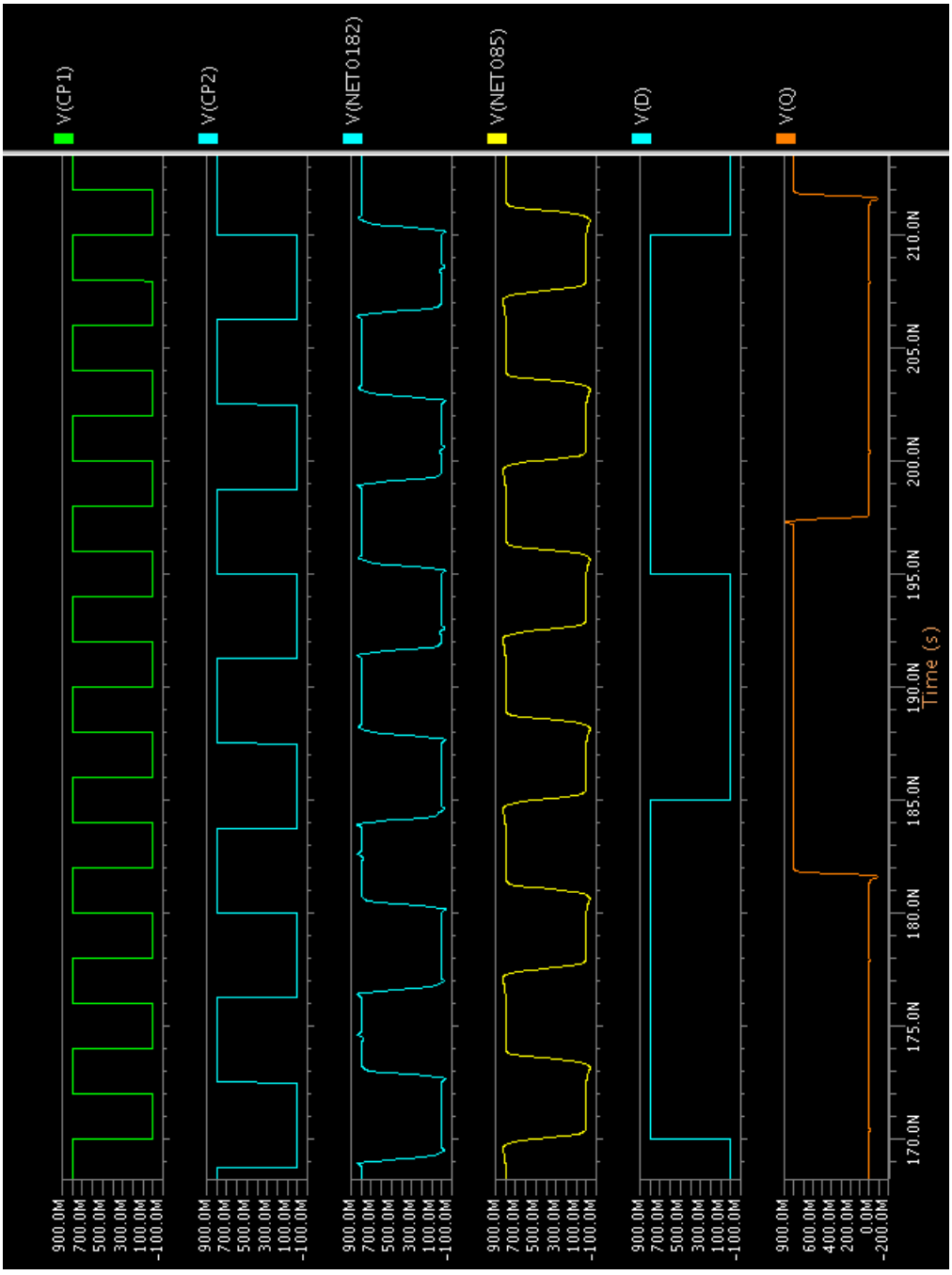
CP1 and CP2 at 250MHz. Phase shifted by three quarters of cycle.



CP1 at 250MHz and CP2 at 238MHz. Phase locked to demonstrate drifting.



CP1 at 250MHz and CP2 at 133MHz. Phase locked. This is near $f/2$ fast-to-slow data transfer.



4.8 References

1 Ginosar, Ran, Fourteen Ways to Fool Your Synchronizer, Proceedings of the 9th International Symposium on Asynchronous Circuits and Systems, Washington, DC, USA, 2003.

2 Ginosar, Ran, Metastability and Synchronizers: A Tutorial, IEEE Des. Test, IEEE Computer Society Press, Los Alamitos, CA, USA, 2011.

3 Mullins, Robert and Taylor, George and Robinson, Peter and Moore, Simon, Point to Point GALS Interconnect, Proceedings of the 8th International Symposium on Asynchronous Circuits and Systems, Washington, DC, USA, 2002.

4 Chapiro, Daniel Marcos, Globally-asynchronous Locally-synchronous Systems (Performance, Reliability, Digital), Stanford University, Stanford, CA, USA, 1985.

5 Krstic, Milovs and Grass, Eckhard and Gurkaynak, Frank K. and Vivet, Pascal, Globally Asynchronous, Locally Synchronous Circuits: Overview and Outlook, IEEE Des. Test, Los Alamitos, CA, USA, 2007.

6 Taylor, George and Moore, Simon and Wilcox, Steve and Robinson, Peter, An On-Chip Dynamically Recalibrated Delay Line for Embedded Self-Timed Systems, Proceedings of the 6th International Symposium on Advanced Research in Asynchronous Circuits and Systems, Washington, DC, USA, 2000.

7 Manohar, Rajit, On the (non) existence of bounded time metastability detectors, Technical note: AVLSI-TN-2014-1, Cornell NYC Tech, New York, NY.

8 Jovanovic G. J., Stojcev M. K., Current starved delay element with symmetric load, International Journal of Electronics, 93, 3, March 2006.

9 Denning, Peter J., The Choice Uncertainty Principle, Commun. ACM, November 2007.

10 Lamport, Leslie, Buridans Principle, Foundations of Physics, Springer USA, 2012.

Chapter 5

The Bayesian Multiplier

'If its stupid but works, it isnt stupid.'

Amphibious Warfare Review, Spring 1989.

This section is not available in the actual report submitted to the Practice School Division due to its tentative publication progress in the IEEE Transactions on Circuits and Systems II : Express Briefs.

5.1 Prelude

This document deals with the theory and modelling of a high accuracy precision bound Bayesian Multiplier using the Muller C gate as the processing unit. By a Bayesian Multiplier we refer to a circuit which can perform inference using Thomas Bayes' famous theorem, stated below

$$P(B_k|A) = \frac{P(A|B_k) \times P(B_k)}{\sum_i P(A|B_i) \times P(B_i)}$$

The digital blocks designed by the procedure explored in this chapter can represent any general multi-valued function on R, but this generalization comes at an added cost of a complicated representation and a larger resultant circuit. The problem is analyzed from a formal systems/metamathematical perspective where parallels are drawn to simplify the process of obtaining an appropriate representation for the inputs. Essentially, this involves looking at the bits in the bit vectors representing the FP numbers as symbols belonging to a logic system with certain production rules which can be used to extend a given representation to one which has more number of bits (i.e., $> M$). Our task then simplifies to finding the right rules and this translates to finding proofs for the higher order symbol strings in our equivalent formal system.

However, we **do not** use the paradigm of SC to perform the Bayesian multiplication, and this is where our work diverges from that of Dr. Friedman's. We try to reduce the amount of information loss due to the randomness that sets in while stochastically generating a bit vector representation for a floating point number by looking for a suitable **code scheme** with which the device performs the Bayesian product with as much accuracy as possible for a given bit vector length (or precision, alternately). We present two such coding schemes: the search tree algorithm and the brute force search (backtracking based search) idea.

We report an estimated 70 per cent accuracy with the search tree procedure while the brute force search gives around 92 per cent accuracy, both the schemes using only 16 bits to represent floating point numbers. Of course, one cannot expect 16 bits to be sufficient for a precision of five digits after the decimal, but one only has to perform the same procedure **only once** to obtain a suitable **code book** for higher order bit lengths.

5.2 A Metamathematical Inspiration

The problem tackled in this chapter deals with the construction of fast and area efficient floating point arithmetic units. The preliminary hurdle that one encounters in the pursuit of a solution is that of representing floating point numbers in an way that is efficient for digital logic. One such method is encountered in the esoteric computing paradigm of "Stochastic Computing" where pure fractions i.e., $\frac{p}{q}$ where $p < q$ and p, q belong to $\mathbb{Z}$, are represented using bit vectors of size q with p ones. This definition is made arbitrarily although it can be proven easily that it's variations are neccessary for most floating point operations. Infact, it is obvious that since we are dealing with outputs that are also going to be represented similarly, our functions have the range of $[0, 1]$. That is,

$$f :: [0, 1] \times [0, 1] \rightarrow [0, 1]$$

If we make the simple modification to the definition that we are now going to be representing $\frac{p}{q}$ with a bit vector which has $[\frac{p}{k}]$ ones and q bits, we would have function whose definition would look like

$$f :: [0, k] \times [0, k] \rightarrow [0, k]$$

Obviously for larger and larger k's you would need longer bit vectors to have sufficient number of ones representing the pure rational. In this paper however, we will discuss the simplest case of $k = 1$. Now that the issue of representation has been settled, we need to design the logic for mapping the inputs to the correct outputs. Here is where the idea of looking at the system as a system of formal logic happens. Let us say we take the function

$$f(a, b) = a * b$$

Let us discuss this case with a 4 bit representation. We know how the function behaves mathematically, that is, it multiplies two FP numbers, and we can do the multiplication manually. In four bits, one can represent four possible pure fractions and one can assign an arbitrary representation for each of them as shown in Table 1.

Table 5.1: Four bit arbitrary representation

Fraction	b_4	b_3	b_2	b_1
0/4	0	0	0	0
1/4	0	1	0	0
2/4	0	0	1	1
3/4	1	1	0	1
4/4	1	1	1	1

It is to be emphasized that the representation is arbitrary because they make *no sense* when there are no rules provided for operations amongst them, rules which will conserve the multiplicative property of the function. We are looking for a scheme, labeled by $\mathcal{R}$, such that,

$$\mathcal{R}(2/4, 3/4) = \mathcal{R}([0,0,1,1],[1,1,0,1]) = [0,1,0,0] = 1/4$$

Thus, we have to play a game where we know the initial configuration and we know the final configuration, but we need to figure out the common set of rules so that we can extend the precision to higher order bit vectors without much computational effort. Solutions to such problems lie in the mathematical field of **proof theory** and formal systems, which deal with looking at the process behind proving theorems as such, and thus also fall under the broad umbrella of metamathematics.

A nice introduction to the subject of formal logic can be found in the very first chapter of [1], where the author invents a pedagogical system known as the M, U, I system to pose the unsolvable MU puzzle. In this system, the only allowed symbols are M, U, I and the string MI is said to be "provided" to us from the start. This can be equivalently stated by calling MI an "axiom" of our system. Three rules, called $\mathcal{R}_1, \mathcal{R}_2, \mathcal{R}_3$, are provided for extending the "axiom" to arrive at a different string of the symbols. They take a certain string as an input, and produce a string with either lesser or greater number of symbols, and by repeated application of these rules, the string MU is required to be "proved" in this system. Rules are described by providing a certain schema for their operation, for instance,

$$\mathcal{R} :: Mx \rightarrow Mxxx$$

can be a valid rule in our system, and its application to the MI axiom gives us

$$\mathcal{R}[MI] = MIII$$

It is to be noted that the symbol x does not belong to our system, but is used as a dummy variable for the schema. Obtaining $MIII$ this way can be seen as a "proof" of the "theorem" $MIII$ by using the axiom MI first and then casting the rule $\mathcal{R}$ on it to obtain the required result. If this is the only rule in our system it can be seen that the strings

$$MIII, MIIIIIIIII, ...$$

are **provable**, whereas the strings

$$MII, MU, MUI, ...$$

are clearly **unprovable** in the current setting. Such systems were investigated extensively by the Norwegian mathematician Axel Thue, and later on elaborated further by the American logician Emil L. Post. They are colloquially known as "String Rewriting Systems" and mathematically known as semi-Thue systems (see [12]). Although Thue laid the foundations for the field, it is Post's work on generalized symbol-manipulation systems that forms the heart of our inspiration, and thus, we would be more correct in calling our system a Post Canonical System instead. The rules $\mathcal{R}_i$ are called Post Production Rules, the string that the rule operates on is called the "antecedent" and the resultant string is called the "consequent". It was stated in the previous section that we need to find out these rules so that we can save time in constructing higher order representations, and we shall show how things map from the FP domain to the canonical domain so that we can proceed with the discussion. To sum things up, we have (the discussion henceforth is borrowed mostly from [2])

Logic System. It is a set of axioms and a set of production rules. Represented by $L = [A, R]$. In our case we have a specific set of predefined bit vectors representing $0/M, 1/M, ..., M/M$ in the axiom set, and an unknown production rule set.

Alphabet. It is a *finite* set of symbols. Whenever not stated it is to be assumed that a string of symbols always comes from a finite alphabet. Represented by $\mathcal{A} = [\alpha_1, \alpha_2, ..., \alpha_n]$. In our case, the alphabet is the set containing 0 and 1 only.

Axiom It is a finite string which cannot be "proved" in L but its truth value is assumed to be known by default. The set of axioms A can be infinite.

Rules A rule is an **effectively computable function** $\mathcal{R}$ which takes $n+1$ inputs, $s_1, s_2, s_3, ..., s_n$ and $\tilde{s}$, and gives a Boolean output. If the output is 1, we can say that we have derived $\tilde{s}$ from $s_1, s_2, s_3, ..., s_n$ and thus, "proved" $\tilde{s}$ in our system. These rules are unknown in our case, and finding them will solve our problem.

To prove anything or to *check* a symbol string, we would need effective computability as a prerequisite. However, finding a proof for a symbol string and checking the legitimacy of the string are two completely different things in our system. For the former one would need access to **all** the production rules, for the latter one would only need to check whether the symbol string represents the product of the two floating point numbers given as inputs (we are still discussing with respect to the function $f(a, b) = a * b$) and this sheds no light on the actual dynamics for higher order multiplication. Although in [2] systems with finite axiom sets are discussed, our system inherently has an infinite axiom structure when considering the representations of all FP numbers in $\mathbb{R}$. This will however not be a problem as we can instead provide an *axiom schema* to collate similar axioms into countable infinite sets. A trivial axiom schema is that of the representation of $0/M$ in which we have M 0's for all M. A better example would be the schema for the representation of $M - 1/M$, where we could enforce the fact that the first bit be always a 0, thereby the representation becomes a 0 followed by $M - 1$ 1's for all M.

Our problem has been reduced to the task of hunting proofs in a Post canonical system which is isomorphic to our scenario. A straightforward way out would be to list out all possible symbol strings of a given length k and then *checking* each of them individually. In the case of the M, I, U system, one can construct a much more concise "theorem tree" due to the ready availability of production rules. This is shown in Figure 1. To progress from a given node to its children, the rules are succesively applied, and for any given node the proof of the symbol string represented by the node can be derived by simply traversing the tree upwards from the node to the axiom root. Due to the lack of the presence of a production rule set, one instead ends up with the tree that looks like Figure 2. To uncover the production rules, we need to consider the process of generating a symbol string of size

k. It can be proved that there would be 2^{k-1} different substring sets that can be used to generate the parent string. Equivalently, we have generated all possible proofs of a theorem of size k, and proof checking can be done with exponential complexity. If the string turns out to be a theorem after all, you would end up with a process which terminates at some node, but if it isn't a theorem (like the string MU in the M, U, I system), one would have a never ending tree. It turns out that (see [2], pp. 224-226) there cannot be any finite time algorithm for arriving at **correct** theorems of a general canonical system. It also turns out that this claim is not strict, and that people have discussed about elementary systems of algebra and Euclidean geometry which have procedures to arrive at correct theorems (see [3], the reader is referred to the extensive bibliography of [2] for further information). One obtuse deviation from the rule can be found in the idea of **truth tables**. Truth tables are a direct evidence of symbol string theorems being true due to their very existence in the table (the concept of a production rule here is abstract and will not be discussed for the sake of clarity), and they can be used to construct combinational CMOS circuits which represent the same. Thus, for the case of the four bit FP multiplication function, the device would have the truth table given in Table 3. To re-emphasize, a truth table can be considered as the perfect theorem checking procedure/the perfect four bit multiplier due to the fact that it will always end up giving the right answer for it *always existed* in the table, even before the user started operating it. That is, we have a symbol string which is a theorem to us but an axiom to the truth table.

Table 5.2: Perfect 4 Bit Multiplier Truth Table

Operation	a_0	a_1	a_2	a_3	b_0	b_1	b_2	b_3	z_0	z_1	z_2	z_3
0/4 * 0/4	0	0	0	0	0	0	0	0	0	0	0	0
0/4 * 1/4	0	0	0	0	0	1	0	0	0	0	0	0
.	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	.
2/4 * 2/4	0	0	1	1	0	0	1	1	0	1	0	0
2/4 * 3/4	0	0	1	1	1	1	0	1	0	1	0	0
2/4 * 4/4	0	0	1	1	1	1	1	1	0	0	1	1
.	.	.	.	.	.	.	.	.	.	.	.	.
4/4 * 4/4	1	1	1	1	1	1	1	1	1	1	1	1

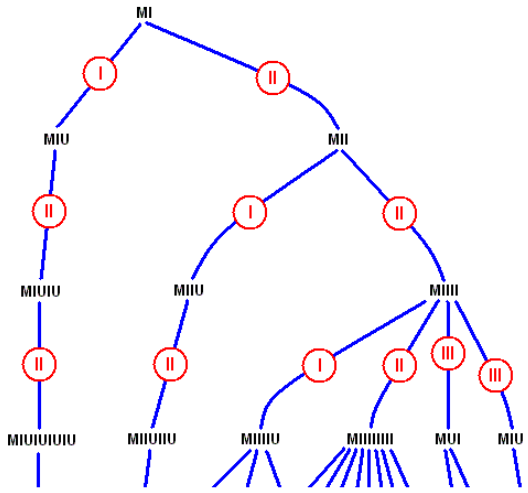


Figure 5.1: The theorem tree for Hofstadter's MIU system.

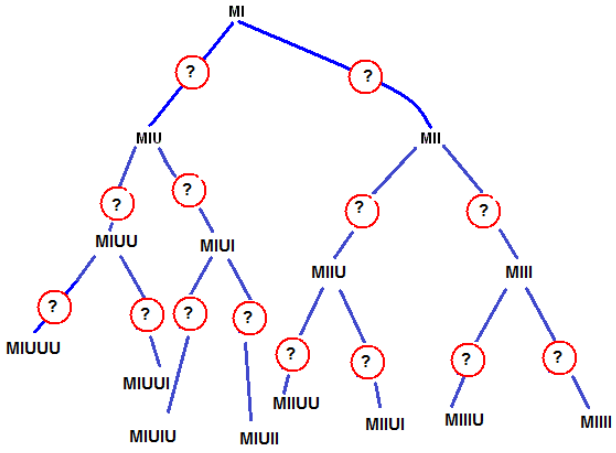


Figure 5.2: The theorem tree for Hofstadter's MIU system without the production rules.

We can now construct Boolean functions for the outputs z_i and thence use logic minimization to get combinational circuitry with minimal area. It can also be seen that due to the combinational nature of the design, it processes all input bits in parallel, thereby improving its thoroughput. The complexity of the process however still remains exponential, as the logic minimization step becomes the bottleneck in the algorithm (and it is known that the number of prime implicants for an n input truth table grows as $O(3^n/n)$, see [9], [10]). Also, the truth tables, and consequently, the circuits grow in size quite quickly. Instead of the standard floating point multiplication function, we have taken the modified Bayesian multiplication operation as the function for our case study to show how quickly the method becomes ineffective. The function is defined as follows:

$$\mathcal{B} :: [0,1] \times [0,1] \rightarrow [0,1]$$

$$\mathcal{B}(a,b) = \frac{a \times b}{a \times b + (1-a) \times (1-b)}$$

To account for experimental variation, we have taken two sets of input encoding schemes to see the changes in the topology of the resulting combinational block. We have repeated the experiment for both 8 and 16 bit flavors, the truth tables being omitted from the paper due to their sheer size. The first encoding scheme uses the naive way of representing a pure rational by filling the ones from the LSB to the MSB. That is,

$$k/M = [0,0,0,..,1,1,1,1..(\times k)..,1]$$

The second encoding uses a representation for k/M which is picked randomly from all possible $C(M,k)$ combinations. The truth tables are sent through a two level logic minimization scheme (see [11] for the algorithmic details), namely that of the heuristic minimzer ESPRESSO. The resulting Boolean functions are further optimized and then translated into a technology independent CMOS mapping. The statistics for the gate distribution are finally obtained and are presented in the following tables. For the sake of demonstration, the actual CMOS gate level schematic for the first table is also provided. From the statistics presented above, it can be seen that the area and power consumption of the combinational Bayes multiplier increases with increasing M, and thus it would become impractical for operating at higher precision levels (that is, at lower 1=M). This is not unexpected as the device performs the operation on all input bits in parallel in one pass. From the

Table 5.3: Eight Bit Naive Combinational Bayes Multiplier

Gate	Fanin	Quantity
AND-OR	4	2
AND-OR-INV	4	8
BUFFER	3	12
INVERTER	1	7
NAND	2	13
NAND	3	5
NAND	4	9
NOR	2	16
NOR	3	2
NOR	4	1
OR-AND-INV	4	5
Total		80

Table 5.4: 16 Bit Naive Combinational Bayes Multiplier

Gate	Fanin	Quantity
AND-OR-INV	4	30
BUFFER	3	370
INVERTER	1	204
NAND	2	117
NAND	3	77
NAND	4	124
NOR	2	269
NOR	3	97
NOR	4	78
OR-AND-INV	4	23
OR	4	8
Total		1397

discussion in Sec. 2, there are an exponential number of logic circuits which can perform the FP operation and these circuits include both sequential and combinational circuits. The flavor of the circuit you end up with depends on how you construct the truth table (the decision procedure) and in the previous section we considered the bit vectors as a whole for each cycle of execution. Alternatively, we can consider other constructions of the truth

Table 5.5: Eight Bit Random Combinational Bayes Multiplier

Gate	Fanin	Quantity
AND-OR	4	7
AND-OR-INV	4	61
BUFFER	3	89
INVERTER	1	96
MUX	3	2
NAND	2	48
NAND	3	30
NAND	4	33
NOR	2	95
NOR	3	33
NOR	4	32
OR-AND-INV	4	16
Total		542

Table 5.6: 16 Bit Random Combinational Bayes Multiplier

Gate	Fanin	Quantity
AND-OR	4	10
AND-OR-INV	4	220
BUFFER	3	491
INVERTER	1	505
MUX	3	2
NAND	2	391
NAND	3	259
NAND	4	296
NOR	2	653
NOR	3	279
NOR	4	302
OR-AND-INV	4	213
OR	4	14
XOR	2	2
Total		3637

table to get a different set of logic circuits belonging to the same exponential space. One such construction would be to consider input vectors to be

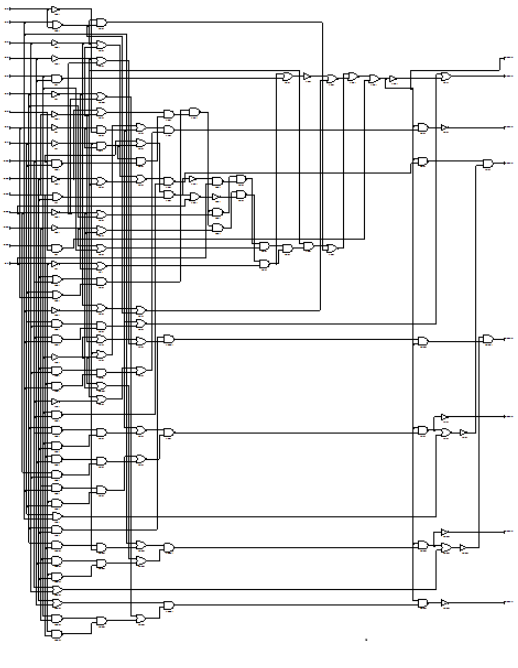


Figure 5.3: Eight Bit Naive Combinational Bayes Multiplier. Synthesis was repeated with just two input NAND and NOR logic (inverter and buffer are default gates).

processed bit by bit concurrently, and for the case of the Bayes multiplier, it turns out that there does exist a sequential block, namely the Muller C Element (see [4]) which may be the circuit living in the exponential space satisfying the functional requirement with the modified decision procedure we stated. We will discuss its construction and operation henceforth.

5.3 Problem Statement

The Muller C element is the atom of the delay insensitive asynchronous fabric. It is widely used in handshake transactions between electronic stages and as completion detection circuitry during control signal transfer from one clock domain to the other. It was introduced by David E. Muller while designing the fully asynchronous ILLIAC system. The two input flavour of the C element gives the output as 1 when both the inputs are 1 and as 0 when both the inputs are zero. In any other case, it gives the output as the previously latched data. The static and dynamic styles of the two input C element are shown below. To understand how this device is the core of the

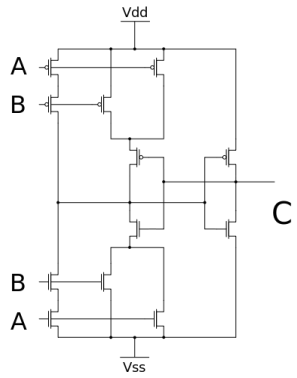


Figure 5.4: Static

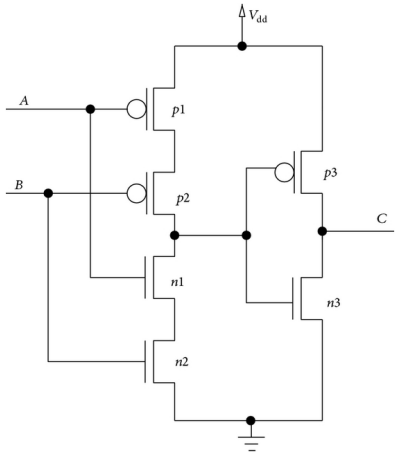


Figure 5.5: Dynamic

current design, one needs to see how it actually operates on **concurrent bit vector inputs**. In reality no two signals can be exactly concurrent, but we assume for the sake of discussion that we can quantize time finely enough to call two signals occurring simultaneously. Let us say that we are sending two bit vectors, labelled as $[\alpha_k]$ and $[\beta_k]$. We also choose that the first bit be always zero, and this choice is not made due to any reason (and can be assumed to be made from the flip of a coin and so on) and this implies that the device initializes with a zero output. It is to be noted that the C element has an inherent metastability condition in which initializations such as $[\alpha_0, \beta_0] = [1, 0]$ or $[0, 1]$ lead to the device "looking" (?) for its non-existent past, so to say. But the initialization $[\alpha_0, \beta_0] = [1, 1]$ is perfectly valid, and

not choosing this does not cause any loss in generality. If the number of bits we send for the α input is taken as M, and if k of these bits are 1, then we say that the probability represented by α as

$$P(\alpha) = \frac{k}{M}$$

If we were assuming the bit vectors to be coming in at random, then the probability that a certain bit at a given instant of time will be 1 is numerically equivalent to the stochastic representation. In VLSI design, when one deals with power and related metrics, one usually considers the switching to play a major role in deciding the average power consumed. The switching factor that is taken into consideration in the expression for the dynamic power consumption is known as the **activity factor**. The activity factor is directly given by the probability that a certain bit of the input is 1 because it is only when the line is at a logic high voltage that it consumes a dynamic power of CV^2f (and when it is at a low voltage, the device is essentially off, thereby giving a power dissipation of zero).

To get the story back to where we began, let us look at the truth table of the C element. There are only two outputs which are known exactly, and

α	β	Output
0	0	0
0	1	$Z(-)$
1	0	$Z(-)$
1	1	1

we know that the other two outputs, labelled by $Z(-)$ are both going to be either 0 or 1. That is, for a $Z(-)$ to take 0, a $(0, 0)$ **must** have happened in the past. This implies that if we account for the exact output cases we indirectly include these too since they follow causally from the former. Thus, the activity factor, and from the above connection, the probability represented at the output as a floating point labelled by a bit vector is given by

$$P(Output) = P(\gamma) = \frac{P(\alpha)P(\beta)}{P(\alpha)P(\beta) + P(\bar{\alpha})P(\bar{\beta})}$$

$$= \frac{P(\alpha)P(\beta)}{P(\alpha)P(\beta) + (1 - P(\alpha))(1 - P(\beta))}$$

The brilliance in Dr. Friedman's observation lies in the fact that if you substitute

$$P(\alpha) = P^*(E_k) = \frac{P(E_k|F)}{P(E_k|F) + P(E_k|\bar{F})}$$
$$P(\beta) = P(F)$$

then you end up with the following

$$P(\gamma) = \frac{P(E_k|F)P(F)}{P(E_k|F)P(F) + P(E_k|\bar{F})P(\bar{F})}$$

This is nothing but Bayes' theorem! We would suggest the reader to go through the paper in full, as the authors explain the usage of such ideas to build inference trees, naive spam filters and so on, with extremely low area and power consumption when compared to current day FPUs. However, the paper is completely based upon the random generation of the bit vectors to work out for them, and it indeed does at large (> 1000) bit rates. This seriously affects the speed of the device as it takes more than a thousand bits to represent one floating point number.

Thus we reach the problem statement. Is it possible to introduce some determinism in the way the bit vectors are mapped to floating point numbers so that for a *low* bit vector size (thus high symbol rate) one can get a decent level of accuracy? We call a bit vector pair **accurate** if the following criterion holds (the equation is written in Haskell code, more on this later, but it is pretty much self explanatory).

$$\texttt{abs(muller(}\ \alpha,\beta\texttt{)}\ \texttt{-}\ \texttt{bayes(}\ \alpha,\beta\texttt{))} < \tfrac{1}{M} \text{ and } M < 50?$$

5.4 The Search Tree (with Arbitrary Tie-breaking)

The problem statement has been presented with an accuracy constraint. In addition to this there are two other implicit constraints. The first one is the obvious point that the number of ones divided by the bit vector size must always represent the pure fraction representation of the floating point number. The other one is based on what is known as the **switching rate** of the bit vector. It is defined as the number of times the bit changes from 0 to 1 or the other way around. Dr Friedman specifically mentions this as an important constraint due to the inertial behaviour of the C element leading to bit errors. The idea is simple - for every i bit substring $S_{(\alpha,i)}$ in the bit vector α representing a floating point number f,

$$P(S_{\alpha,i}) \rightarrow f$$

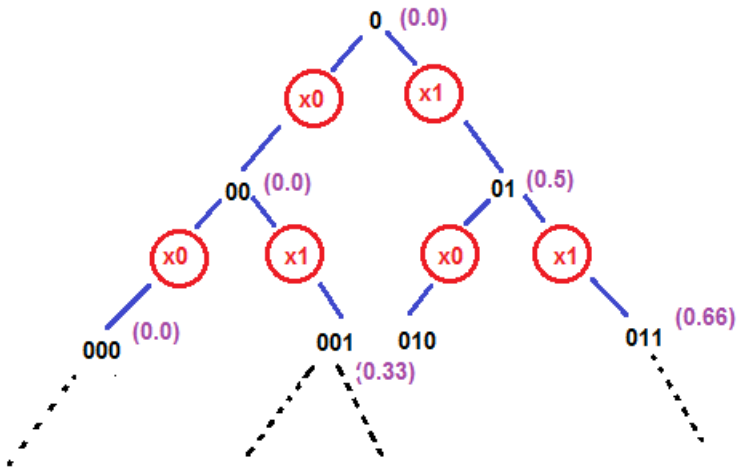
This criterion is the essence of the search tree method. The method is sensible as the tree that is generated in theory contains all possible representations of all possible floating point numbers ever present, so it is only a matter of traversing/searching in the tree to find the right encoding for f. The construction of the tree is given by the following Haskell code, followed by it's explanation.

```
data Tree a = Empty | Node a (Tree a) (Tree a) deriving
   (Show, Eq)
--Generate the G-STAT Parse Tree
generateTree :: [Int] -> Int -> Int -> Tree ([Int],
   Float)
generateTree v max count
  | count == max =
    Node (v, b2f v) Empty Empty
  | otherwise =
    Node (v, b2f v)
        (generateTree (v ++ [0]) (max - 1) (count))
    (generateTree (v ++ [1]) (max - 1) (count))
```

The first line generates a tree data structure with a recursive definition. It allows comparisons and being able to "show", basically an IO operation. The next definition is that of a function which generates the tree. The type

definition system in Haskell is pretty convenient here to explain what the function actually does. It takes a vector of Int's, basically a Bool array but True and False being represented by integers 1 and 0 respectively, and this vector represents the code of the root of the tree (and it is taken as "0" by default, for the most general tree).

Then it takes the maximum depth you would want to extend the tree upto, and this is equal to M, your bit vector size. The third input is a running counter to enable a recursive definition. The explanation as to how the function works can be sufficed with by noting the actual way the tree would look like (on the next page).



The actual tree structure generated holds an ordered pair at each node, whose first entry is the bit vector representation of the second entry (which is given by the type **Float**) with the precision decided by the depth of the tree. Thus, for finding the representation of a given $\frac{p}{q}$ in M bits such that at every $i^{th}(< M)$ level the representation is closest to $\frac{p}{q}$, one starts to traverse the tree from the root, and does the following operations at each level of the tree. We define a function to access a given node and extract the ordered pair it holds, as shown below.

```
--Read a given node in the tree.

readNode :: Tree ([Int], Float) -> ([Int], Float)
```

```
readNode (Node a left right)  = (fst a, snd a)
```

The actual traversal is quite simple and is done in two steps:

1 At a given node $N(l, r, (\alpha_i, f_i))$ and depth i, compute the difference between f and the floating point numbers represented by the bit vectors stored in l and r. If the differences are not equal then select the child which minimizes the difference.

2 Tied differences are broken arbitrarily.

```
compare :: Float -> Tree ([Int], Float) -> Float -> Char
compare f (Node a left right) rand
| abs (snd (readNode left) - f) < abs (snd (readNode
   right) - f) = 'L'
| abs (snd (readNode left) - f) > abs (snd (readNode
   right) - f) = 'R'
| otherwise = if (rand < 0.5) then 'L' else 'R'
```

The **rand** input is the Bernoulli trial which is done at each node which presents a tied difference during the traversal. Generalizing this arbitration to the whole tree is done by recursively extending the **compare** function, and generating a new random number at each level, if not for each node (slightly higher determinism, hence). The random number generation is done with the first function, while the traversal is executed by the second function, given as follows.

```
--Generate a random list of floats from a given random
   postiive integer

randomList :: Int -> [Float]
randomList seed = randoms (mkStdGen seed) :: [Float]

--Map compare to the whole tree given an f.
compare_m :: Float -> Tree ([Int], Float) -> [Char] ->
   Float -> Int -> [Char]
compare_m f (Node a l r) soFar rand count
--Return the soFar iterator if the end of the tree is
   found.
| l == Empty || r == Empty = soFar
--If the compare at the current node returned 'L' add a
   0 to the code.
```

```
--Update the random number
| stat == 'L' =
  compare_m f l (soFar ++ "0")
  (head $ randomList (count + round (rand * 10000))) (
     count + 1)
--If the compare at the current node returned 'R' add a
   1 to the code.
--Update the random number
| stat == 'R' =
  compare_m f r (soFar ++ "1")
  (head $ randomList (count + round (rand * 10000))) (
     count + 3)

  where stat = compare f (Node a l r) rand
```

That is all there is to the algorithm. To test this out, we need to first build a behavioural framework for the Muller C gate, and a test framework for measuring the approximate deviation from the expected Bayesian result. These are shown in the code snippets that follow.

```
--Behavioral C Element Boolean Function
muller :: [Int] -> [Int] -> [Int] -> Int -> Int -> [Int]
muller (a:as) (b:bs) sofar count max
| count == max  = sofar ++
  [if a == 1 && b == 1 then 1
   else if a == 0 && b == 0 then 0
   else last sofar]
| otherwise     =
  muller as bs (sofar ++
  [if a == 1 && b == 1 then 1
   else if a == 0 && b == 0 then 0
   else last sofar]) (count + 1) max


--Behavioral C Element Bayesian Inference Generator
--This is the experimental model which
--needs to be tested against actual Bayesian inference.
muller_float f s = b2f $ muller f s [] 1 (length (f))
```

The test framework is given in the code snippet below, and this is a simple random number generator which is seeded at runtime and provides ordered pairs of floats for testing purposes.

```
--Convert a vector of strings to a vector of integers.
funct :: [String] -> [Int]
funct = map read

--Experimental Inference.
einf :: Float -> Float -> Int -> Float -> Float
einf alpha beta prec rand =
  muller_float
  (funct
  $ tail
  $ splitOn ""
  (compare_m alpha (generateTree [0] prec 1) ['0'] rand
     1))
  (funct
  $ tail
  $ splitOn ""
  (compare_m beta (generateTree [0] prec 1) ['0'] rand
     1))


--Actual Bayesian product.
bayes :: Float -> Float -> Float
bayes alpha beta =
  (alpha * beta)/(alpha * beta + ((1-alpha)*(1-beta)))

--Random Float Pair generator.
randomTuples :: Int -> Int -> [(Float, Float)]
randomTuples k seed =
  [(x, y)
   | x <- take k $ randomList seed, y <- take k $
      randomList seed,
     x < y]  --double-counting fixed.



theoretical :: Int -> [Float]
theoretical rand =
  [bayes (fst k) (snd k) | k <- randomTuples 10 (rand)]
```

```
experimental :: Int -> Float -> [Float]
experimental rand rand2 =
  [einf (fst k) (snd k) 512 rand2 | k <- randomTuples 10
      (rand)]

sol :: Num t => Int -> Float -> [t]
sol rand rand2 =
  [if
   (abs ((theoretical rand)!!i - (experimental rand
      rand2)!!i)) < 0.1
   then 1
   else 0
   | i <- [0..(length (theoretical rand)) - 1]]
```

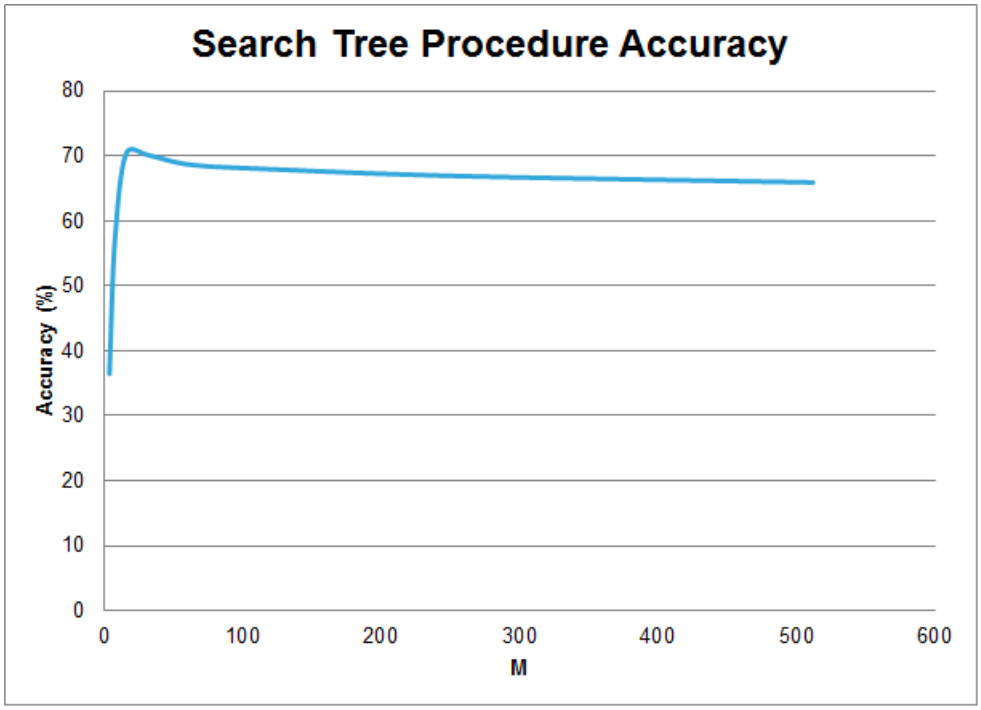
The function **theoretical** takes a random (large) integer as an input and uses it as a seed to generate 10 random pairs of floating point numbers, and then generates an array holding the actual Bayesian product of the two floats. Using the same seed, the function **experimental** uses the search tree procedure **einf** to get the Bayesian product from the C element's behavioural model stored in the function **muller**. The function **sol** finally computes the absolute difference between the two Bayesian products to give a 1 if the difference falls below the threshold (which is 0.1 in the snippet, and is supposed to be $\frac{1}{M}$ according to our claim). The actual testing is done by seeing how many test cases actually give a "1" and this is done by simply summing over the output of the function **sol**. This is included in the **main** function shown below.

```
--Pick an element from a given list at random.
pick :: [a] -> IO a
pick xs = fmap (xs !!) $ randomRIO (0, length xs - 1)

submain :: (Eq t, Num t, Num a) => t -> [a] -> IO [a]
submain 0 = return ()
submain n = do
  numb <- randomIO
  numb2 <- randomIO
  picked <- pick $ (take 400 $ randomList (numb2))
  print $ sum (sol numb picked)
  submain (n-1)

main :: IO ()
```

```
main = do
  submain 100
```

Around 2×10^6 tests were performed over eight different bit vector sizes. The results are shown in the table and the corresponding graph plotted for the same.

Bit Vector Size (M)	Accuracy (%)
4	36.4549
8	57.2191
16	70.3755
32	70.14747
64	68.629
128	67.9119
256	66.9242
512	65.9332



It can be seen that the accuracy for the scheme reaches a peak at around 16 bits/symbol rate and then reduces gradually but very slowly as M increases

further. A possible cause for the trend might be the fact that the randomness in the whole code generation scheme takes over after the number of bits that are controlled by a deterministic execution are considerably lesser than the new bits entering the code.

5.5 The Brute Force Search (with Backtracking)

The reason why we were not able to get more than 75 per cent accuracy in the previous scheme was due to the increase in the stochasticness of the system at higher M values. The errors that lead to the reduction in the accuracy are primarily due to a bit position mismatch which then propagates due to the inherent inertia of the C element. Given an M, what if we were able to design codes for $\frac{1}{M}, \frac{2}{M}, ..., \frac{M-1}{M}$ so that for any Bayesian product between $\frac{k}{M}, \frac{k'}{M}$ done using the C gate, we have the minimum deviation from the actual mathematical value?

This is the motivation for the brute force search idea, and this involves noting the following points before proceeding into the actual functional implementation. Firstly, one must understand that for a given M, the representation of $M-1$ is fixed. This is obvious, as the first bit must always be "0" according to our convention (stated in Sec 2) and the rest of the $M-1$ bits need to be 1's so that the vector represents $\frac{M-1}{M}$. That is,

$$\beta_{M-1} = [0, 1, 1, 1, ..., 1]$$

In addition to this, we know the trivial element in the set, that of $\frac{0}{M}$,

$$\beta_0 = [0, 0, 0, 0.....0]$$

Apart from the above, for any other k, β_k would be represented by a bit vector which is indexed by it's permutation index in the set of all $C(M, k)$ combinations of k ones in M bits. By a permutation index, we mean the index at which the required bit vector β_k is obtained in the list of all arrangements of k ones in M bits. In order to proceed any further, we would need the machinery to generate these combinations and allow access to them via their indices. This is given in the code snippet below.

```
choose :: [b] -> Int -> [[b]]
_      `choose` 0       = [[]]
[]     `choose` _       =  []
(x:xs) `choose` k       =  (x:) `fmap` (xs `choose` (k
   -1)) ++ xs `choose` k

changePos :: [Int] -> Int -> [Int]
```

```
changePos list size =
  [if elem i list && i /= 1 then 1 else 0 | i <- [1..(
     size)]]

genVector :: Int -> Int -> [[Int]]
genVector size ones =
  [changePos u size | u <- posList] where posList =
     choose [1..size] ones

showPos :: Int -> Int -> [[Int]]
showPos size ones =
  (filter (\x -> sum x == ones) (genVector size ones))
```

The function **choose** takes an input list, and the number of combinations it has to create out of all the elements of the input list and gives out a list of such combination lists. This function is used in the function **genVector** which generates all possible bit vectors with a specified number of ones, and this employs the auxiliary function **changePos**. The functions also take care of the fact that in no situation do the combinations start with a "1", which is in accordance with our convention from the start. The function **showPos** allows accessing the indexed vector by giving an ordered list of bit vectors as the output.

To reduce the naivety in the whole search procedure, we note the interesting fact that the vector β_{M-1} always gives the minimum Bayesian product divergence with β_1 only if the permutation index of β_1 is given by $\frac{M}{2} - 1$ or the least integer to the same. The exact reason for the behavior is related to the fact that $C(M, k)$ is maximized at $k = \frac{M}{2}$ and that the output vector will switch only if there is a 1 encountered in β_1. Thus, so far, we have been able to fix $\beta_0, \beta_1 and \beta_{M-1}$. To ease out the computation for the other bit vector codes, we use the following auxiliary functions:

```
fpDiv :: Int -> Int -> Float
fpDiv a b =
  (fromIntegral a)/(fromIntegral b)

minCodeIndex :: [Int] -> Int -> Int -> [Int]
minCodeIndex prev curr prec =
  findIndices ( ==
    (minimum $ map (\x -> abs $ (muller_float prev x) -
       (bayes (fpDiv (sum prev) prec) (fpDiv curr prec))
```

```
        )
          (showPos prec curr)))
         (map (\x -> abs $ (muller_float prev x) - (bayes
             (fpDiv (sum prev) prec) (fpDiv curr prec)))
          (showPos prec curr))

diffGen :: [Int] -> [Int] -> Float
diffGen a b =
  muller_float a b - bayes (fpDiv (sum a) (length a)) (
     fpDiv (sum b) (length b))

intersectM :: [[Int]] -> [Int] -> [Int]
intersectM (x:xs) soFar
  | xs == []    = x `intersect` soFar
  | otherwise   = intersectM xs (x `intersect` soFar)

extendIntersect :: Int -> [(Int, Int)] -> Int -> [Int]
extendIntersect prec soFar_tuples pos =
  intersectM
    [minCodeIndex (showPos prec (fst a) !! (snd a)) pos
       prec| a <- soFar_tuples]
        (minCodeIndex (showPos prec (fst b) !! (snd b))
           pos prec) where b = soFar_tuples !! 0
```

The function **fpDiv** is used to convert a pure fraction/rational number into a floating point number. Haskell inherently does not process patterns such as "3/4" automatically as "0.75" but instead assumes it to be of the type Rational. We do not want that, and hence we use the conversion. Then, **minCodeIndex** returns all possible permutation indices at which the Bayesian product is minimized for β_{curr} when being multiplied with β_{prev} at a precision "prec" which will numerically be the same as M. **diffGen** takes two codes as inputs and shows the difference between the Bayesian products computed for the same via the C element and the actual value, and this function will be used for verifying how close we actually get with a given encoding.

The next two functions essentially enforce the concept that if one finds a code representing a certain β_k then this code should return the minimum when multiplied with $\beta_0, \beta_1, \beta_2, ..., \beta_{k-1}$. Since the Bayesian product is symmetric with respect to the inputs, it would suffice if we compared the divergences between β_i and β_j for all $i \leq j$. The function **intersectM** extends the **inter-**

sect function of Haskell, which gives the intersection between two given input lists, to include finding intersections between more than two sets. The intersections we are looking for are the common permutation indices of β_k which keep the Bayesian product at a minimum when taken with $\beta_0, \beta_1, \beta_2, ..., \beta_{k-1}$. **extendIntersect** is a wrapper around the previous function, and is written to make the whole intersection taking business more streamlined by maintaining a running list of tuples $[(k, I(k))]$ where $I(k)$ is the permutation index for the correct code word mapped to $\frac{k}{M}$.

This is all there is to the machine's involvement in constructing these exact codes. It turns out that one does not always end up with non empty intersections at all times (although we believe that the chances that there will never be a non empty intersection increase as M increases, and thus the chance that there will be an alternative coding scheme increases too), and at such times one must resort to the additional process of **locally fixing** the codes, and this cannot be exactly done by a machine. This involves observing changes in the whole β_M system by dynamically varying the error threshold above $\frac{1}{M}$ until all the products settle to the new threshold.

Table 5.7: Eight Bit Exact Bayesian Product Code - Accurate To $\Delta = 0.1$

β_1	0	0	0	0	1	0	0	0
β_2	0	0	1	0	0	0	0	1
β_3	0	0	1	0	1	0	1	0
β_4	0	1	1	1	1	0	0	0
β_5	0	1	1	0	1	1	0	1
β_6	0	1	1	1	1	1	0	1
β_7	0	1	1	1	1	1	1	1

Table 5.8: Ten Bit Exact Bayesian Product Code - Accurate to $\Delta = 0.1$

β_1	0	0	0	0	0	1	0	0	0	0
β_2	0	0	0	1	0	0	0	1	0	0
β_3	0	0	1	0	1	0	0	1	0	0
β_4	0	1	1	0	1	0	0	1	0	0
β_5	0	1	1	0	0	1	0	1	1	0
β_6	0	1	1	0	0	1	0	1	1	1
β_7	0	1	1	1	1	1	0	1	0	1
β_8	0	1	1	1	1	1	1	1	0	1
β_9	0	1	1	1	1	1	1	1	1	1

Table 5.9: 16-Bit Exact Bayesian Product Code - Accurate to $\Delta = 0.07$

β_1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
β_2	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0
β_3	0	0	0	1	0	0	0	0	1	0	0	0	1	0	0
β_4	0	0	1	1	1	0	0	0	0	0	0	1	0	0	0
β_5	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0
β_6	0	1	0	1	1	0	0	0	0	0	1	1	0	0	0
β_7	0	0	1	1	0	1	0	1	1	0	1	0	0	1	0
β_8	0	1	0	1	0	0	1	0	1	0	1	1	1	0	0
β_9	0	1	1	1	0	1	1	0	1	1	0	0	1	0	1
β_{10}	0	1	0	1	1	0	0	1	1	0	1	1	1	1	0
β_{11}	0	1	1	1	1	0	1	1	1	0	0	1	1	1	1
β_{12}	0	1	1	1	0	1	0	1	1	1	0	1	1	1	1
β_{13}	0	1	1	1	0	1	1	1	1	1	1	1	0	1	1
β_{14}	0	1	1	1	1	1	1	1	1	1	1	1	1	0	1
β_{15}	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Table 5.10: Statistics For Brute Force Search

Bit Length (M)	Threshold	Accuracy
8	0.125	95.032
8	0.1	49.572
10	0.1	83.705
16	0.0625	87.292
16	0.07	89.548

5.6 References

1 Douglas R. Hofstadter, Godel, Escher, Bach: An Eternal Golden Braid, Basic Books, New York, 1999.

2 Marvin L. Minsky, Computation : Finite And Infinite Machines, Prentice Hall, New Jersey, 1967.

3 Alfred Tarski, A Decision Method For Elementary Algebra and Geometry, University of California Press, Berkeley, 1951.

4 Joseph S. Friedman, Laurie E. Calvet, et al., Bayesian Inference With Muller C-Elements, IEEE Transactions On Circuits and Systems - I : Regular Papers, February 2016, DOI : 10.1109/TCSI.2016.2546064.

5 Armin Alaghi, John Hayes, Survey of Stochastic Computing, ACM Transactions on Embedded Computing Systems, Vol. 12, No. 2s, Article 92, May 2013, DOI : 10.1145/2465787.2465794.

6 B. R. Gaines, Stochastic Computing, Spring Joint Computer Conf., 1967.

7 Emil L. Post, Formal Reductions of the General Combinatorial Decision Problem, American Journal of Mathematics, Vol. 65, No. 2 (Apr., 1943), pp. 197-215.

8 Emil L. Post, Recursive Unsolvability of a Problem of Thue, The Journal of Symbolic Logic, Vol. 12, No. 1. (Mar., 1947), pp. 1-11.

9 W. V. O. Quine, A Way To Simplify Truth Functions, The Amer. Math. Monthly, 62 (9), pp. 627-631, November 1955.

10 Edward J. McCluskey Jr., Minimization of Boolean Functions, Bell Sys. Tech. Jour., 35 (6), pp. 1417-1444, November 1956.

11 Robert K. Brayton, Gary D. Hatchel, et al., Logic Minimization Algorithms for VLSI Synthesis, Kluwer Academic Publishers, Dordrecht, 1984.

12 Ronald V. Book and Friedrich Otto, String-rewriting Systems, Springer Pub., 1993, ISBN 0-387-97965-4.

Birla Institute of Technology and Science
Pilani, Rajasthan
Practice School Division

Station
STMicroelectronics N.V.
Center
Greater Noida, U. P.
ID No. and Names
Anurag Pallaprolu
2012B5A3405P
Title of the Project
Asynchronous Digital Logic

Code No.	Response Option	Course No.(s) & Name
1.	A new course can be designed out of this project.	
2.	The project can help modification of the course content of some of the existing Courses	
3.	The project can be used directly in some of the existing Compulsory Discipline Courses (CDC)/ Discipline Courses Other than Compulsory (DCOC)/ Emerging Area (EA), etc. Courses	
4.	The project can be used in preparatory courses like Analysis and Application Oriented Courses (AAOC)/ Engineering Science (ES)/ Technical Art (TA) and Core Courses.	
5.	This project cannot come under any of the above mentioned options as it relates to the professional work of the host organization.	

Signature of Student
Date:

Signature of Faculty
Date: