

Asynchronous Self Timed Processing: Improving Performance and Design Practicality

D. Browne and L. Kleeman

October 11, 2005

Department of Electrical and Computer Systems Engineering.
Monash University.
Australia

Abstract

The area of self timed circuits has been investigated with the intention of using the method to increase the performance of a processing circuit. Two main implementation systems have been investigated, VLSI layouts and FPGAs. The resulting circuits have been shown to have advantages over similar synchronous circuits. The research has also resulted in self timed, clock-less, processing circuits on a standard FPGA. In addition CAD tools are being developed to decrease the design to implementation time of a circuit. Finally a simulation program has been written to decrease the processing time required to simulate the performance of a transistor based circuit.

Contents

1	Introduction	3
2	List of Acronyms	4
3	What is a Self Timed Circuit	5
3.1	Production Rules	5
3.2	Dual-Rail Encoding	6
3.3	Ordering	7
3.3.1	Weak Conditions	7
3.3.2	Four Phase Handshaking	8
3.4	Isochronic Forks	8
4	Implementation Systems	11
4.1	VLSI	11
4.1.1	Precharge Half Buffer	11
4.1.2	Comparison with Equivalent Synchronous Circuits	21
4.1.3	Large Scale Simulator	23
4.1.4	Automated Circuit Creation	26
4.2	FPGA	27
4.2.1	Implementation of Self-Timed Circuits Using an RS latch with precedence	27
4.2.2	Current Work	30
5	Appendix A: Limitations to CMOS DI Circuits	32
6	Appendix B: Full Adder Circuit Definition	34

1 Introduction

Most commercially available processors have a clock and digital sequential designs; we refer to these circuits as synchronous in this report. The period of the clock is based on the worst case delay through its various combinational logic blocks. The maximum throughput of a synchronous processor is governed by the worst case delay. Asynchronous self-timed processing, on the surface, appears to have a number of performance advantages that warrant further investigation [1]. A number of large scale projects have shown that these advantages can be implemented to the benefit of the performance of the final design [4, 5, 7, 14, 17]. The dominance of synchronous systems has meant that most design tools and hardware description languages are designed for creation of synchronous systems. Reconfigurable logic hardware is also designed for synchronous systems, with each programmable logic block containing some configurable combinational logic (or lookup table), usually followed by an edge triggered flip flop and/or latch. Hence the design of asynchronous self-timed systems can be a very laborious task. The introduction of design tools that can produce circuits with similar performance to synchronous design tools would increase the practicality of implementing very large scale processor designs. Currently there are a large number of proposed methods for implementing self-timed circuits each with their own advantages and disadvantages [11]. This research intends to create design methodologies that are complemented with a set of design tools. The aim is that these methodologies will be able to quickly implement large scale designs that are not bounded by synchronous design rules.

2 List of Acronyms

DI: Delay Insensitive. A circuit design which operates regardless of any lengths of delays of signals within the circuit.

CMOS: Complimentary Metal Oxide Semiconductor. Logic Circuits that use complementing PMOS and NMOS transistors to actively drive the output of a function.

FPGA: Field Programmable Gate Array. An array of configurable logic elements. Not limited to either synchronous or asynchronous elements, though nearly all commercial FPGAs are intended for synchronous use.

NMOS: N-channel Metal Oxide Semiconductor. Either a metal oxide semiconductor transistor with N (electron) majority carriers, or Logic circuits that only use NMOS transistors which actively drive circuits to ground but are passively driven high through an NMOS transistor configured as a pull up resistor.

PCHB: Pre-charge Half Buffer. A type of basic self timed processing element that uses NMOS transistors to preform the calculation and resets the circuit using PMOS transistors. The calculation transistors and reset transistors do not form fully complementary circuits but are mutually exclusive.

PMOS: P-channel Metal Oxide Semiconductor. A metal oxide semiconductor transistor with P (hole) majority carriers.

PR: Production Rules. A method for describing the operation of logic gates of all types.

QDI: Quasi Delay Insensitive. A circuit that operates correctly regardless of the delays of signals within the circuit, except for some assumptions about certain delays, usually the isochronous fork assumption.

SPICE:Simulation Program with Integrated Circuits Emphasis. The generally accepted standard for simulation of electrical circuits.

VLSI: Very Large Scale Integration. In general the system of describing a logic circuit at the transistor layout level.

3 What is a Self Timed Circuit

Self timed circuits rely on enforcing the order of events for their correct operation. We can determine whether or not a computation is completed if we know the order in which events must occur to complete the computation. This is how self timed circuits work. The order of events is enforced by the circuits and the control logic. The following is an explanation and description of these orders.

3.1 Production Rules

Production rules, introduced by Martin [2], are a convenient way of describing the operation, and ordering of self timed circuits. In the remainder of the report, the operation and ordering of signal and gates will be described using production rules.

The simple assignments $x := true$ and $x := false$ are denoted by $x \uparrow$ and $x \downarrow$ respectively. A gate with an output Z has the production rules:

$$\begin{aligned} B_u &\mapsto Z \uparrow \\ B_d &\mapsto Z \downarrow \end{aligned}$$

B_u is a Boolean function that, when true, causes $Z \uparrow$. Likewise B_d is a Boolean function that, when true, causes $Z \downarrow$. B_u and B_d must be mutually exclusive by definition. For example for an AND gate, with inputs A and B, is described as follows:

$$\begin{aligned} A \wedge B &\mapsto Z \uparrow \\ \neg A \vee \neg B &\mapsto Z \downarrow \end{aligned}$$

The symbols \wedge , \vee , \neg assume their normal predicate logic functions. That is AND, OR and NOT respectively.

State holding can be implied by these production rules. An ideal RS latch (with no forbidden state) can be described by the following production rules:

$$\begin{aligned} S \wedge \neg R &\mapsto Q \uparrow \\ R \wedge \neg S &\mapsto Q \downarrow \end{aligned}$$

A Muller C element can be described as follows:

$$\begin{aligned} A \wedge B &\mapsto C \uparrow \\ \neg A \wedge \neg B &\mapsto C \downarrow \end{aligned}$$

In self timed circuits, the validity of data is important as well. The functions, $v(X)$ and $n(X)$, are true when X is valid and X is null respectively. When looking at a multiple bit signal, null is quite different to not valid, this is explained later. Where a self timed Boolean gate, with input X and output Y , assigns an output according to a Boolean function $f(X)$ the output assignment $Y \uparrow$ is such that $Y = f(X)$ holds and $v(Y)$ holds as a post condition of $Y \uparrow$. Likewise the reset of such a gate, is such that $n(Y)$ holds as a post condition of $Y \downarrow$.

We can wait for a condition to hold by using the production rule $B \mapsto skip$ which means "wait until B holds" and is written as $[B]$. The ordering of successive events can be described using this and the semicolon. The following means "wait for A to hold and then set B to a valid value".

$$[A]; B \uparrow$$

This is only performed once. To repeat $[B]$ indefinitely we write $*[B]$.

To perform two events in any order or concurrently we use the comma. For example, if a single gate has multiple outputs, Y and Z , then the production rules may be written as follows.

$$\begin{aligned} B_u &\mapsto Y \uparrow, Z \uparrow \\ B_d &\mapsto Y \downarrow, Z \downarrow \end{aligned}$$

For self timed circuits the operation may only be correct given a particular ordering of inputs. We call this the environment of the circuit. For example a self timed circuit may only operate correctly if the input is set to a valid state only once the output is null, and also will only reset to a null state correctly once the output is in a valid state. The environment of such a circuit, with input X and output Y , would have to be described similar to the following equation.

$$[[n(Y)]; X \uparrow; [v(Y)]; X \downarrow]$$

For processing systems we may want to select the output according to some function. To do this we use the guarded selection notation:

$$[B_1 \mapsto S_1 \mid \dots \mid B_n \mapsto S_n]$$

This requires that at most one of B_1 - B_n can be true. If none is true then we wait until one is true.

For concurrent operations we write:

$$S_1 \parallel S_2$$

Where S_1 and S_2 are completed concurrently.

3.2 Dual-Rail Encoding

As there is no clock in self timed systems to indicate the validity of data, that validity must be encoded with the data itself. The most common way of achieving this is via dual-rail encoding. Each data bit is encoded on two signals called rails. For a data bit X , the two rails are X^0 and X^1 , meaning X is zero and X is one respectively. If X^0 is asserted, then the data is valid and has a value of zero. If X^1 is asserted, then the data is valid and has a value of one. If neither is asserted then the data bit is null. The case of both being asserted is not allowed and can, in some circumstances, represent an error. For a single data bit X , $n(X)$ is true when neither line is asserted. Likewise $v(X)$ is true when one of the two lines is asserted. This can be summarized by the following definition. For a single data bit:

$$\begin{aligned} v(X) &\stackrel{def}{=} (X^0 \neq X^1) \\ n(X) &\stackrel{def}{=} (\neg X^0 \wedge \neg X^1) \end{aligned}$$

For an N bit signal Y made up of the data bits Y_1, \dots, Y_n

$$\begin{aligned} v(X) &\stackrel{def}{=} (\forall k : 1 \dots N - 1 : v(Y_k)) \\ n(X) &\stackrel{def}{=} (\forall k : 1 \dots N - 1 : n(Y_k)) \end{aligned}$$

3.3 Ordering

As mentioned earlier, self timed circuits operate correctly by enforcing the correct order of operations. This report will look at two possible ways of conceptualizing this order. These two concepts are the weak conditions [8] and four phase handshaking [3]. It will become apparent that these two concepts in fact describe the same ordering of events.

3.3.1 Weak Conditions

The weak conditions are a set of conditions that a self timed circuit must obey in order to operate correctly. The concept is quite simple; don't set the output to a valid state until all inputs are valid. The concept of the weak conditions is generally simpler to understand than four phase handshaking because the weak conditions are a set of linguistic rules. The weak conditions are summarized in Table 1.

Table 1: Weak Conditions

1.	Some input becomes defined	Precedes	Some output becomes defined
2.	All inputs become defined	Precedes	All outputs become defined
3.	All outputs become defined	Precedes	Some inputs become undefined
4.	Some input becomes undefined	Precedes	Some output becomes undefined
5.	All inputs become undefined	Precedes	All outputs become undefined
6.	All outputs become undefined	Precedes	Some input becomes defined

Though sufficient as described above, it can be useful to visualize the weak conditions. Figure 1 is a graph showing the dependencies of the weak conditions.

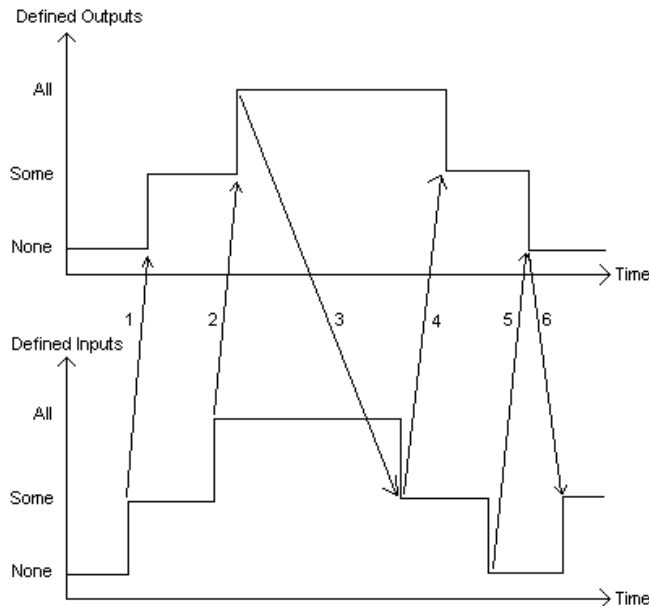


Figure 1: Weak Conditions Graph, derived from [6]

Conditions 1,2,4 and 5 govern how the self time circuit must be designed, while conditions 3 and 6 govern the control logic of the system overall. Only through careful application of these conditions can a system with average case performance be achieved. Later we will see two implementations of an asynchronous ripple carry adder that, through careful application of the weak conditions, achieves logarithmic performance on average. In the case of a full adder, the carry signal can be produced before all the inputs to the gate are defined. When arranged in the form of a ripple carry the performance varies with the length of the longest carry chain propagation for that calculation, which is generally considered logarithmic on average.

3.3.2 Four Phase Handshaking

Four phase handshaking is a description of the communication process between asynchronous elements. When communication between cells is implemented using four phase handshaking, the weak conditions are met. It should be noted that four phase handshaking is not the only way to communicate between basic cells, but rather four phase handshaking is a delay insensitive (DI) communication method. By delay insensitive we mean that the correct operation is guaranteed regardless of delays within the implementation. In regard to communication this means the communication will operate correctly regardless of the delays along wires.

Four phase handshaking essentially makes sure no cell receives any non null input until it is ready to receive it and produces no non null output until the cell following it is ready to receive it. In addition, no cell receives a null input until it is ready to receive it, and produces no null output until the cell following it is ready to receive it. These requirements can be simply written using the production rule notation explained earlier. Splitting the system up into a producer and consumer, four phase handshaking can be written as follows:

$$\begin{aligned} \text{producer} &\equiv *[[ci]; \text{produce } X; X \uparrow; [-ci]; X \downarrow] \\ \text{consumer} &\equiv *[[n(X)]; ci \uparrow; [v(X)]; \text{consume } X; ci \downarrow] \end{aligned}$$

Obviously a system with the above cells does not do anything useful, i.e. there is no processed output derived from an input. Since any useful system must have an output, each cell must be both a producer and a consumer. Combining the two equations gives a function evaluation cell, F with input X and output Y, and the surrounding environment, E [3].

$$\begin{aligned} F &\equiv *[[v(X)]; Y \uparrow; [n(X)]; Y \downarrow] \\ E &\equiv *[\text{produce } X; [n(Y)]; X \uparrow; [v(Y)]; X \downarrow; \text{consume } Y] \end{aligned}$$

It can be easily seen from these equations that the behavior is identical to that described by the weak conditions. The function equation refers to conditions 1, 2, 4 and 5 while the environment equation refers to conditions 3 and 6.

3.4 Isochronic Forks

Though communication between cells can be DI, the operation of the cells themselves cannot be entirely DI. Instead these cells can only be Quasi Delay Insensitive (QDI) circuits. By this we mean that the circuit is delay insensitive except for a few assumptions about certain delays. This assumption is the Isochronous

Fork assumption. This assumption says that when a signal breaks off into two forks, the delays along each of the two lines are approximately the same. The allowable differential delay between each line of the fork is usually one gate delay, if the delay is greater than that, the circuit may fail.

It has been shown that the only gate that has a delay insensitive implementation is the Muller C element [2]. However, it is simple to show that the standard CMOS implementation of the C element is not delay insensitive because of the complementary nature of CMOS. See appendix for details.

Figure 2 is a circuit of a fully functional QDI gate. The circuit is delay insensitive except for forks. This gate is a dual rail implementation of a XOR function and is implemented in a style known as the precharge half buffer (PCHB). The circuit is a dynamic implementation, meaning it will only hold its state using the wire capacitances and relies on low leakage currents.

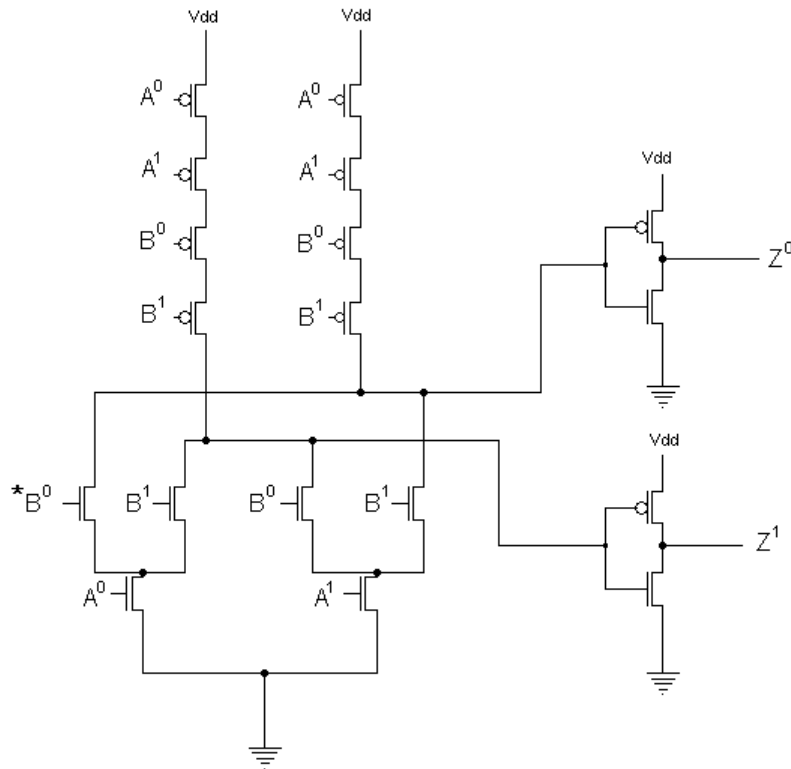


Figure 2: Self Timed XOR Gate

Assume all forks are isochronic, except the B^0 input marked with a *. Assume that this fork is much slower than the others. The weak conditions, 3 and 6, will be met by the surrounding environment. Start with inputs $A = 0$ and $B = 0$, that is A^0 and B^0 are asserted. The gate stays in this state for a long time, and the output switches to 0, that is Z^0 becomes asserted. According to condition 3 the input may now change to a null state. All of the inputs shown in the circuit below go low, except for $*B^0$ due to the large delay. The circuit resets the output to a null state, that is Z^0 and Z^1 are not asserted. According

to condition 6 the input can now be set to a valid state. This time the inputs become $A = 0$, $B = 1$. However, $*B^0$ is still asserted and so, according to the circuit, Z^0 and Z^1 will both become asserted. Since the correct operation of the circuit is dependent on some delays the circuit is not DI.

In the above example not all forks were isochronic. If all forks are isochronic then the circuit obeys all the weak conditions and operates correctly, regardless of the other delays in the circuit.

4 Implementation Systems

In order to verify the operation of any self timed circuit the design should be implemented. Simulations alone are limited. There are many different ways of implementing this sort of design. Ultimately the design is expected to be of a very large scale; hence some methods are impractical, such as PLA's and the interconnection of several ICs. The two most obvious large scale schemes are FPGAs and a VLSI design. Both have there advantages and disadvantages. Currently this research is exploring both options.

4.1 VLSI

This method of design is the fairest method for comparison with synchronous systems. It allows for the optimization of both synchronous and asynchronous circuits at the transistor level and is ultimately the desired implementation method for both systems. This method can be used to simulate the effect of the complexity of the design. The more complicated a layout is, the slower the circuit may perform. It may be that the effect of the complexity would make a design too slow to be useful even though simulations that do not take the layout into account would indicate that the circuit performs adequately.

Though the design time for VLSI circuits is generally longer than that of an FPGA this is not necessarily the case for self timed circuits. Because of the problems with synthesis and layout of FPGA implementations associated with self timed logic, VLSI design is not that much slower than the FPGA equivalent. In addition, the FPGA implementation of self timed circuits turns out to use a large amount of resources. Therefore the main disadvantage of a VLSI design is implementation cost that can be many tens of times larger than the cost of an FPGA.

4.1.1 Precharge Half Buffer

The Precharge Half Buffer (PCHB) style circuit is generally the circuit most commonly used in the fastest asynchronous microprocessors [16]. This style of circuit has three main parts as shown in Figure 3. These parts are the calculating circuitry, the reset circuitry and the state holding circuitry.

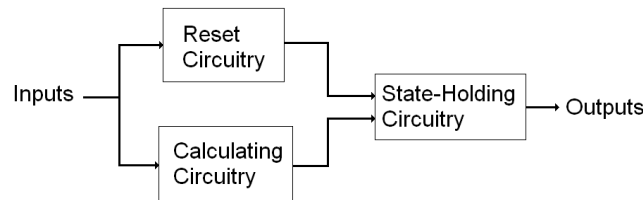


Figure 3: General Layout of a Precharge Half Buffer

The operation of the PCHB, with input vector X and output Y , can be described by the following equation:

$$\text{PCHB} \equiv *[[v(X)]; Y \uparrow; [n(X)]; Y \downarrow]$$

The ordering of the intermediate states of Y between $n(X)$ being true and $n(Y)$ begin true, is not important, as long as no intermediate state is a valid number.

The reset circuitry produces a signal dependent on the validity of the input ($n(X)$). The calculating circuitry produces a signal dependent on both the validity ($v(X)$) and the value on the input data ($Y = f(X)$). The state-holding circuitry holds the state of the circuit while inputs are in an intermediate state.

For a VLSI design the reset circuitry is usually a chain of PMOS transistors (followed by an inverter) that pulls down the output when $n(X)$ is met. The state holding circuitry can be either dynamic or static. The dynamic version is just one inverter and relies on the wire capacitances and low leakage currents to hold the state. The static version requires some sort of feedback to hold the state indefinitely. The calculating circuitry can be a tree of transistors whose interconnections determine the operation of the circuit.

Optimization of Reset Circuitry: The reset circuitry can be optimized by features of the function being implemented. The operation of the reset circuitry must obey the weak conditions 4 and 5. Generally the circuit will operate faster if we can reduce the fan in of the pull up chain, and so this will be our aim in this section.

There are two features of the function that can be used to reduce the fan in of the reset chains. The first uses clever application of weak condition 4, that is ‘some inputs become undefined precedes some outputs become undefined’. For multiple output functions the pull up chain can be split up between the circuits that calculate each output. There is no restriction on how the chain is split up; this means data bits can be split between two circuits, i.e. the X^0 PMOS transistor can be part of one pull up chain while the X^1 PMOS transistors is part of another.

The second feature is that only rails used to calculate the result for that rail are required to be used in the pull up chain. The simplest example of this is a self timed ‘and’ gate. Assume the gate has inputs A and B , and output Z . The Z^1 value is calculated using only A^1 and B^1 and hence only A^1 and B^1 is required in the Z^1 pull up chain. The reason is simple, if the circuit requires A^0 or B^0 to return to zero in order to reset the circuit then Z^1 was never asserted and hence they can be removed from the pull up chain.

Optimization of Calculating Circuitry: The aim of the optimization of calculating circuitry is to increase the operating speed of the gate and to reduce the number of transistors required to implement the function. In some cases these two requirements can be at odds with each other, however we will present a method that achieves both optimal transistor numbers and has been shown to be both faster and more energy efficient than comparable optimization methods demonstrated by Martin [3].

It should be noted that this method, along with all others, bounds the transistor count by an exponential function of the number of inputs. The method shown in this report does result in a lower transistor count than other proposed

methods and for a small number of inputs will result in adequate transistor counts. The problem is inherent in dual rail encoding and the requirements of the weak conditions. Luckily most gates used in data paths in synchronous processors have only a small number of inputs and hence this method should compare well with large scale synchronous systems.

Similar to the reset circuitry, there are two features of the function that can be used to reduce the transistor count of the gate. The method starts with a general template for implementing logic functions and reduces the transistor count according to the features of the function.

Starting with the basic case of a function that has just one output, the calculating circuitry must pull up the output only when all inputs are valid. What this means is that the calculating circuitry must pull down one of the two output rails for every possible valid input. Since each valid input is unique, for an N input function there must be 2^N paths from ground through the calculating circuitry. Figure 4 shows the simplest, and most naive, way of achieving this for a three input function. Each of the paths are tied to either the Z^0 or Z^1 line according to the truth table of the function being implemented. It is fairly simple to see that the circuit can be interchanged with the circuit shown in Figure 5 without any concern for the function being implemented. This is because each rail of each data bit is asserted mutually exclusively with respect to the other rail, hence at any time there is at most one path to ground. The circuit in Figure 5 will be our template circuit. It should also be noted that later an inverter will follow these paths and hence this circuit pulls up the output.

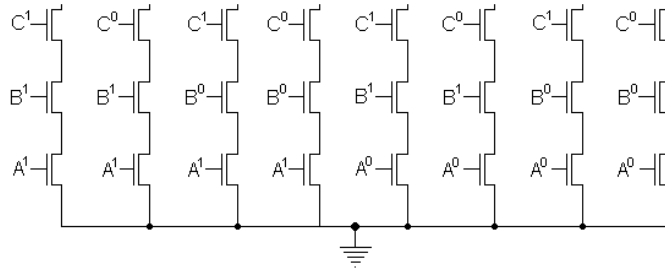


Figure 4: Basic Non-Optimized Transistor Array

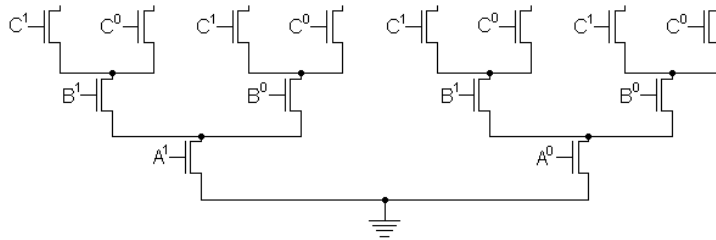


Figure 5: Basic Three Input Transistor Tree

This transistor tree reduces the number of transistors required by sharing paths through some transistors. The truth table of the function also reveals other paths that can be shared. For example, in the above circuit, paths through the C transistors can be shared if C^1 and C^0 both map to the same output lines for two or more pairs of C transistors. Table 2 is the truth table for the function generated by the reduced transistor tree in Figure 6. This optimization method means that at most, there should be eight transistors in the last level in any calculation circuit for a precharge half buffer, since there are only four possible output combinations for any pair of C transistors, regardless of the number of inputs.

Although this function is independent of the value of B , this is not an altogether useless circuit. Requirements of self timed circuits [6] mean that some functions may have to wait for the validity of some variables before the output becomes valid.

Table 2: Truth Table of Reduced Transistor Tree

A	B	C	F(Out)
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

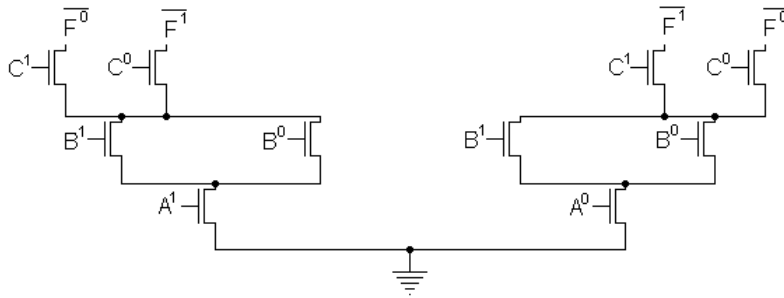


Figure 6: Example of Reduced Transistor Tree

Although more complicated, it is possible to share paths, in the same way, through transistors in other levels. However, as you go further down the transistor levels the maximum number of transistors increases exponentially and hence it becomes far less likely (for functions selected at random) for shared paths to be found. Furthermore, in general, the more shared paths there are, the less useful the circuit is. Using just this reduction method on the highest level of transistors means the maximum number of transistors, for the calculating circuitry, is bounded by the equation

$$\text{No. Transistors} \leq \begin{cases} (2^N + 6)M & \text{if } N \geq 3 \\ (2^{N+1} - 2)M & \text{if } N < 3 \end{cases} \quad (1)$$

where N is the number of inputs and M is the number of outputs. For small N this figure is quite acceptable. The basic template tree (Figure 5) accounts for $2^{N+1} - 2$ transistors per function output. When the number of inputs exceeds three, the basic template (Figure 5) results in more than eight transistors at the highest level. Since we know that at most only eight transistors are needed at the highest level, we use the template of an $N - 1$ function and add eight transistors. This results in the above equation.

The second type of optimization that can be applied to the calculating circuitry is removing transistor levels through careful application of the weak conditions 1 and 2. These two conditions specify that all outputs may not become valid until all inputs are valid, but that some outputs can become valid after some inputs become valid. For multiple output gates, one of the outputs can become valid once enough inputs to calculate that value become valid. To illustrate how this improves the performance of the system, we will look at the example of the full adder.

The full adder has three inputs, A , B and ‘Carry In’ (C), and two outputs, Sum and ‘Carry Out’ ($Cout$). Looking at the behavior of a full adder, the sum can only be calculated once all three inputs are known. The carry can be calculated once any two inputs are known provided those two inputs have the same value. If all of these inputs are defined at the same time, this fact does not help the performance of the system much. However, if one of the inputs is known to arrive later than the other two then this does help the performance. When full adders are laid out in a ripple carry adder, it is known that the carry input to each adder will, generally, arrive later than the other two inputs A and B . If the carry out signal can be calculated after only A and B are defined then the average performance of the ripple carry adder will be much faster.

The truth table reveals whether or not a calculation relies on an input. In the case of a transistor tree shown in Figure 5 the output is not dependent on the input C for any pair of C transistors if the output of each of those two transistors are the same nodes. In the case of the ripple carry adder, the C transistors following the path A^0 and B^0 are both connected to $Cout^0$. Similarly the C transistors following the path A^1 and B^1 are both connected to $Cout^1$. Therefore, for a multiple output full adder, these two transistors can be removed, giving the following two circuits for the full adder, Figure 7 and Figure 8. Because the sum output is always dependent on C , all of the outputs will not become defined until all of the inputs are defined, so these two circuits together meet the weak conditions.

Though the pull down transistors create a longer path than the circuit produced by Martin in [3], this circuit uses less transistors and actually has a smaller propagation delay and lower energy consumption than Martin’s circuit. Figures 9 and 10 show the differences in rise time between the two carry circuits for identical input signals. Subsequent to designing this circuit the author found a later design by Martin [5], which is almost identical to these circuits.

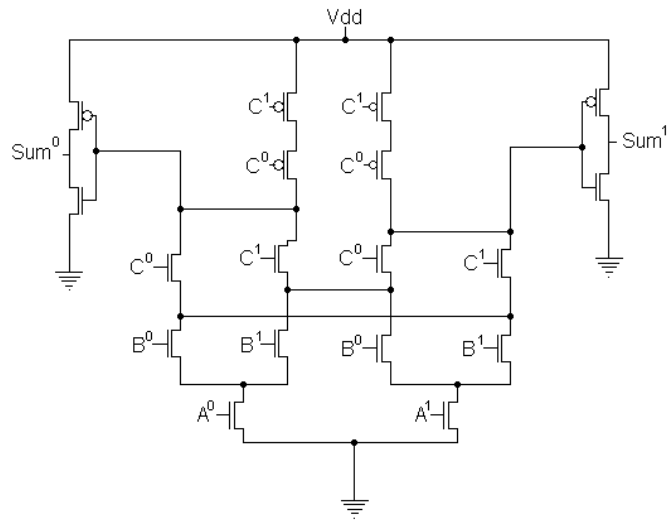


Figure 7: Self Timed Sum Logic

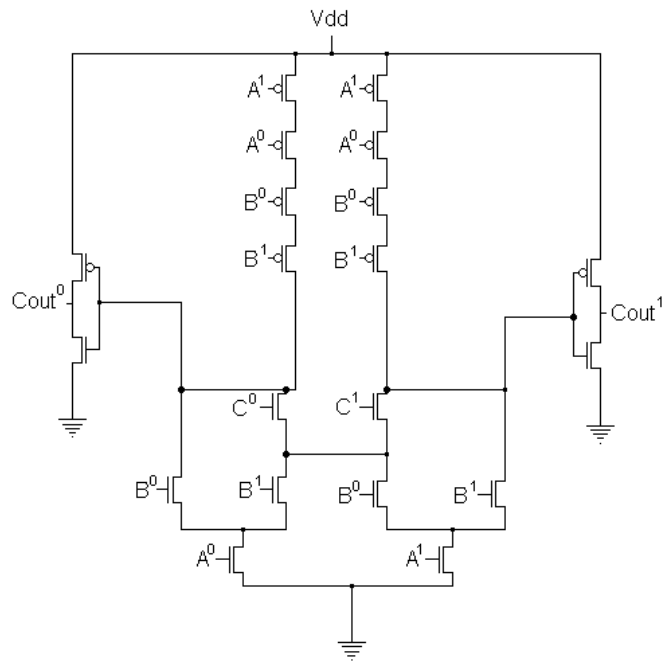


Figure 8: Self Timed Carry Logic

State Holding Circuitry, (Static C Element types): The operation of the PCHB is very similar to that of the Muller C element. In fact a PCHB implementation of the AND function, employing the optimization methods explained earlier, results in a Muller C element for the Z^1 line. Therefore it seems as though the circuits used in static C elements could be applied to the precharge half buffer. Research presented in [10] compares the implementations of the

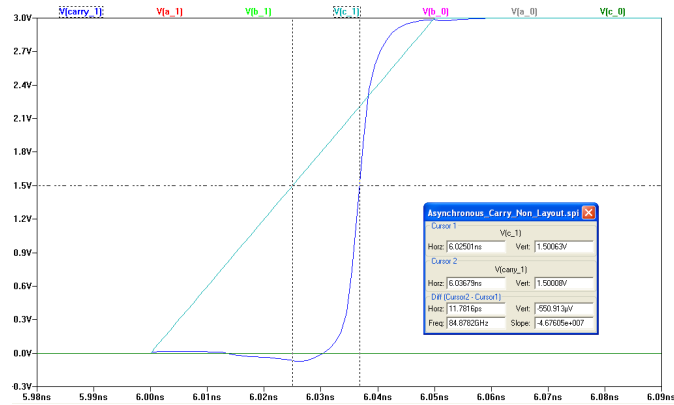


Figure 9: Self Timed Carry Signal Timing

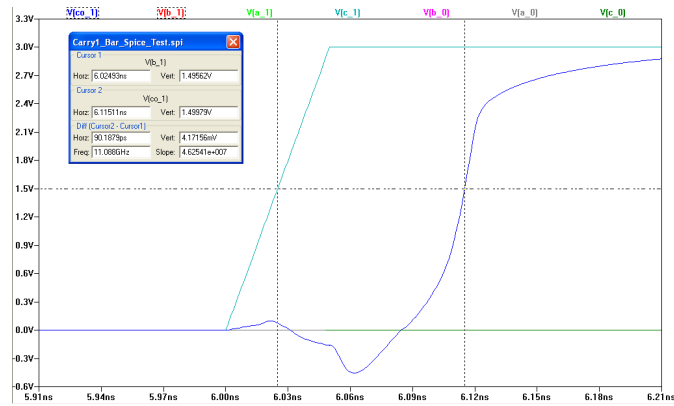


Figure 10: Self Timed Carry Signal Timing for Circuit from [3]

Martin, Sutherland [13] and Van Berkel [15]. However, [10] does not include a comparison with the dynamic implementation. Similar application of the state holding circuitry of the Sutherland have been presented in [12], and this also briefly explores the idea of using the state holding circuit proposed by Martin. However [12] does not include the Van Berkel method, which was shown in [10] to be both the fastest and most energy efficient implementation.

We implemented each of these circuits in a VLSI layout for MOSIS technology. The circuits have been simulated using SPICE and a comparison of their propagation delays and energy consumption is presented here. The layouts of these circuits are shown in Figure 12, Figure 13, Figure 14 and Figure 15.

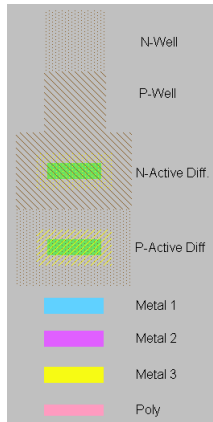


Figure 11: VLSI Layout Key

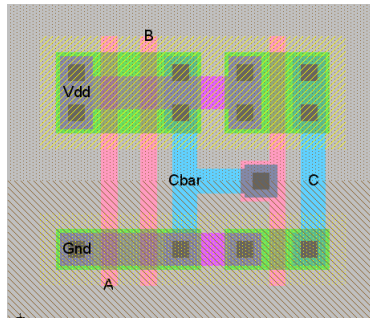


Figure 12: VLSI Layout of a Dynamic C Element

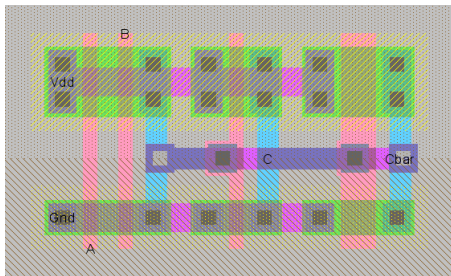


Figure 13: VLSI Layout of a Static C Element Proposed by Martin

The results of the simulations can be summarized by the following. Of the three static gates, the Van Berkel implementation was found to be the fastest and most energy efficient, despite the more complicated layout. The dynamic C element was found to be faster and more energy efficient than all the static implementations but did come quite close to the performance of the Van Berkel C element.

Some attempts were made at applying the Van Berkel C element's state holding circuitry to general PCHB circuits. However, it was found that circuit

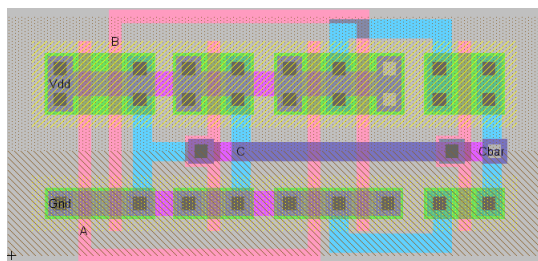


Figure 14: VLSI Layout of a Static C Element Proposed by Sutherland

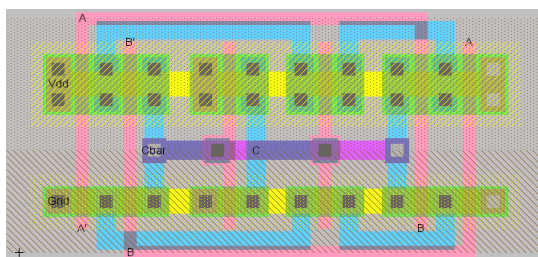


Figure 15: VLSI Layout of a Static C Element Proposed by Van Berkel

sizes increased exponentially with this circuitry. Even a circuit as simple as a four input C element requires 66 transistors, compared with 18 for the Sutherland implementation and 10 for the dynamic implementation. Because of this large transistor count, the Van Berkel C element method becomes unfeasible for circuits with more than 2 or 3 inputs. The method proposed by Sutherland is well documented and is used for Null Convention Logic self timed gates [12]. This method adds, at most, an additional $U + 2D + 4$ transistors where U is the number of transistors used in the pull up tree and D is the number of transistors used in the pull down chain. An example of a two input OR function, using the Sutherland style feedback is shown in Figure 16. Effectively an additional calculation tree is created and the reset logic is replicated in the pull up and pull down chain of the feedback inverter. Despite the larger number of transistors the performance of the circuit is still much better than using the weak inverter feedback method proposed by Martin. Figure 17 shows the rise time of the circuit shown in Figure 16. Figure 18 shows the rise time of a self timed OR gate with a weak inverter feedback.

The above circuit is the basic circuit with no reset circuitry optimizations. Six transistors can be removed from the Z^0 circuitry by applying the second optimization to the reset circuitry. Hence all A^1 and B^1 transistors can be removed from the Z^0 circuitry. In addition this optimization means the A^0 and B^0 pull up chain is not needed in the Z^0 circuit and hence one more transistor can be removed. Even with these optimizations the transistor count is quite high, 38 compared with the equivalent synchronous circuit, 26 (6 transistors for the OR gate and 20 for a static edge triggered flip flop with no pass transistors). The dynamic self timed gate has 16 transistors, compared with the equivalent synchronous circuit (dynamic data storage and no pass transistors) which has 18 transistors. In both cases, the dynamic implementations are also faster. For

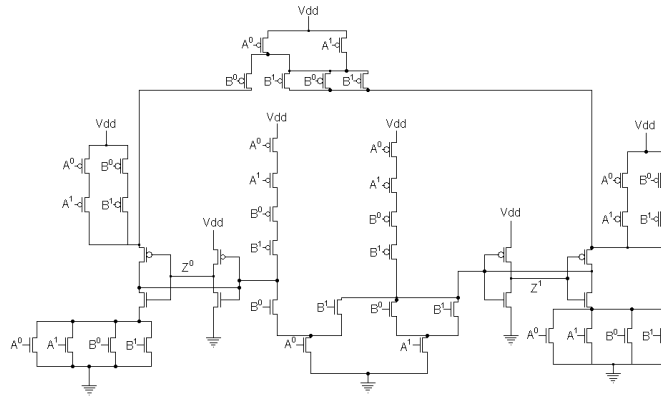


Figure 16: Self Timed Or Gate with Sutherland Style Feedback

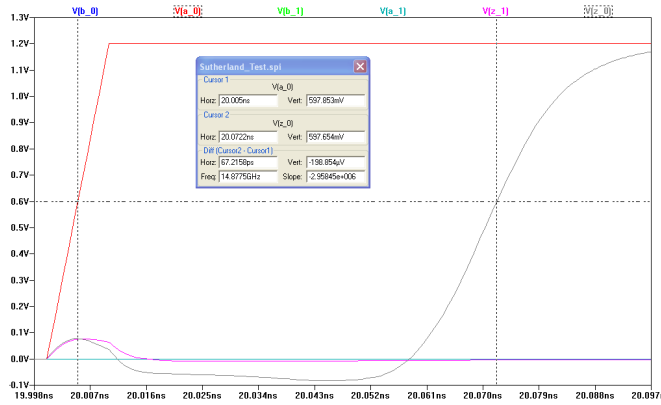


Figure 17: Sutherland Or Gate Timing

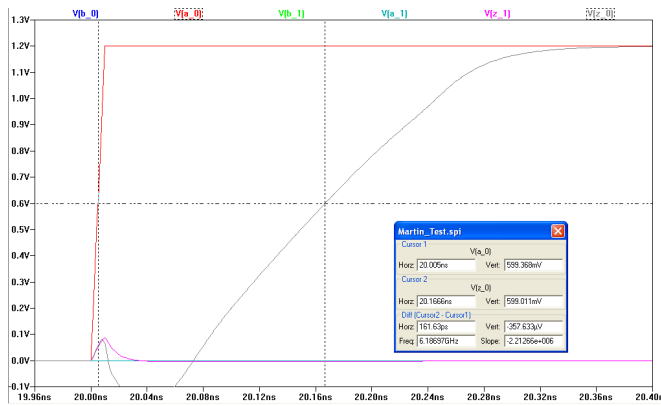


Figure 18: Martin Or Gate Timing

the self timed circuit, the dynamic implementation is approximately twice as fast as the static implementation. The rise time of the dynamic implementation

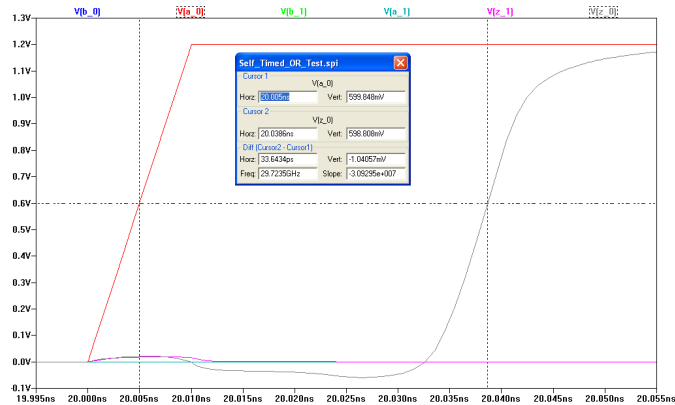


Figure 19: No Feedback Self Timed Or Gate Timing

is shown in Figure 19.

4.1.2 Comparison with Equivalent Synchronous Circuits

Due to the nature of the problem, it is difficult at this early stage to fairly compare synchronous designs to asynchronous designs. To fairly compare the two methodologies an entire system would have to be created and then the performance of the two could be compared. Currently simulations have been run on the basic self timed full adder circuit and the equivalent synchronous full adder (combinational full adder followed by flip flops). The results indicate that under the right circumstances the asynchronous system has the potential to run significantly faster than the synchronous system even before average case running times are taken into account. It should be noted, however, that these figures are only simulated estimates of maximum operation speed and should be treated as such. Many aspects of the overall design have not been taken into account which would further reduce both figures.

Circuits: The equivalent two circuits to be compared are an implementation of a self timed full adder, very similar to Martin's [3] and the conventional combinational full adder referred to in [9]. These circuits were laid out in VLSI and the SPICE simulation files were created by Electric. The self timed circuits are shown in Figure 7 (Sum) and Figure 8 (Carry). The self timed circuit layouts are shown in Figure 20 (Sum) and Figure 21 (Carry). The synchronized full adder is shown in Figure 22 and the layout in Figure 23.

Simulation Results: Simulation results have indicated that the asynchronous system can perform faster than the synchronous system given as long as the input is provided correctly. The control logic required to provide the input has not yet been developed in this research and so is not included in the simulation. Similarly the logic required to provide the inputs to the synchronous system has not been developed and hence is not included in the simulation. Table 3 is a summary of the results. The processing delay for the self timed system is the time required for the circuit to change state from undefined output

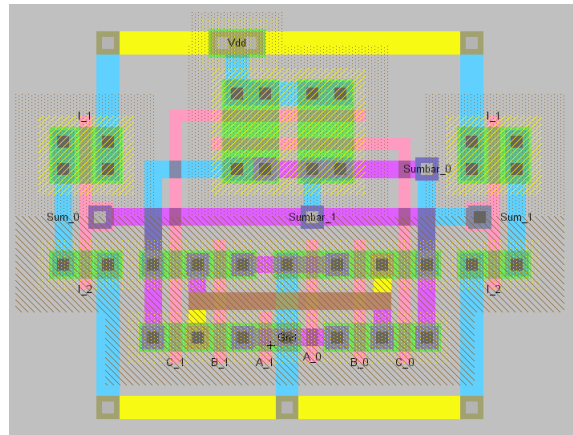


Figure 20: Self Timed Sum Logic Layout

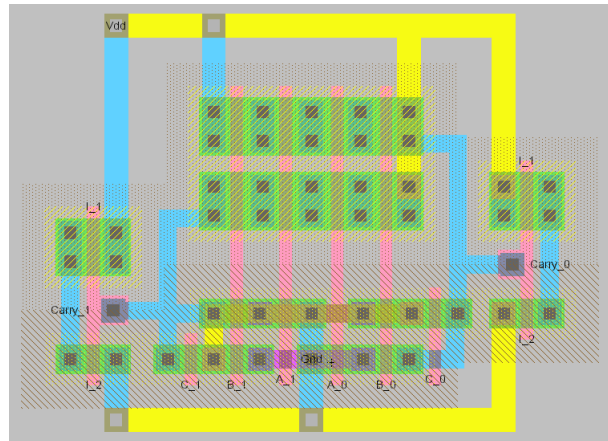


Figure 21: Self Timed Carry Logic Layout

to defined output. The reset delay is the amount of time required to change state from defined output to undefined output. The processing delay for the synchronous system is the maximum delay of the conventional full adder (Figure 24). The external delays for the synchronous system are the delays caused by set up times and propagation delays through the edge triggered flip flops surrounding the adder. The results indicate that the self timed circuit is both faster and smaller than the synchronous equivalent. Furthermore, these results are only for a single bit and hence are not an indication of the self timed circuit's ability to take advantage of algorithms that have a faster average time than worst case time. Therefore, if both designs were implemented in a ripple carry adder, the self timed circuit would have a significantly faster average time than the synchronous system's worst case time. Additionally, the average time of the self timed circuit would reflect the general throughput of the self timed multiple-bit adder and hence be much faster than the worst case delay of the equivalent synchronous adder.

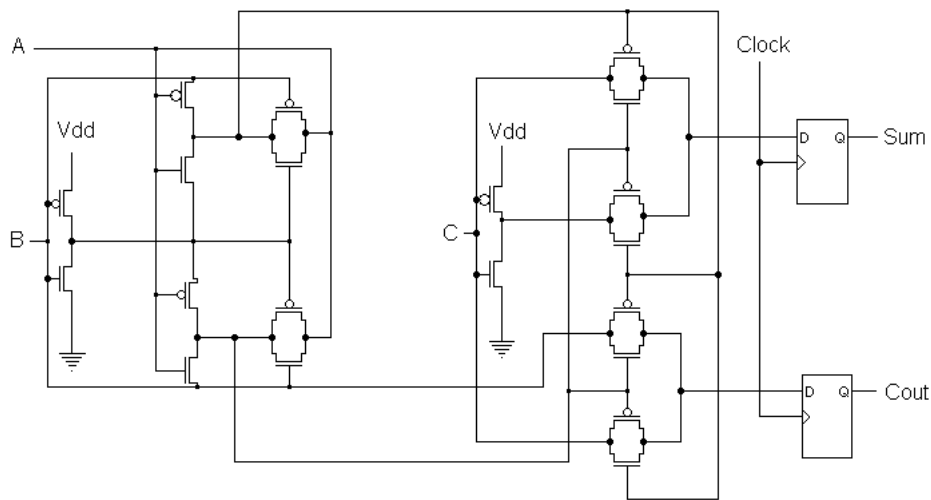


Figure 22: Synchronized Conventional Full Adder

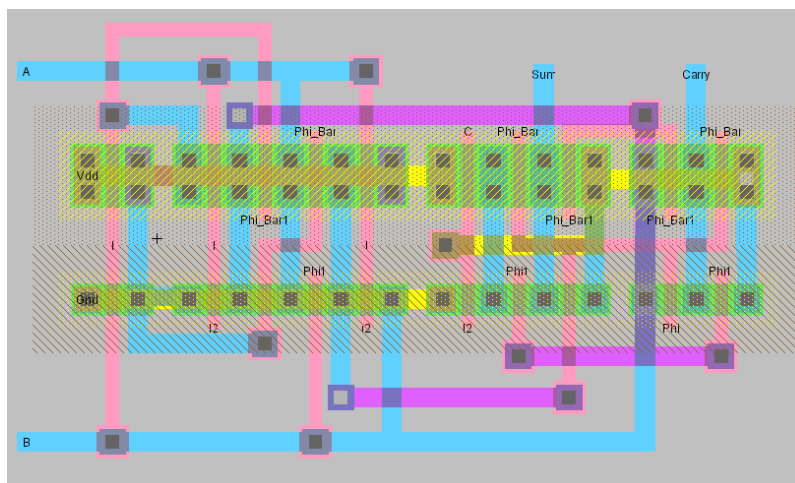


Figure 23: Conventional Full Adder Layout

Table 3: Summary of Comparison of Synchronous and Self Timed Full Adder Designs

Circuit	Processing Delay	Reset Delay	External Delays	Total Delay	Transistor Count
Self Timed	59ps (Sum)	106ps (Carry)	0ps	165ps	38
Synchronous	83ps (Max)	0ps	148ps	231ps	44

4.1.3 Large Scale Simulator

For a very large scale design, repetitive SPICE simulations become unfeasible. Some other simulation system would need to be used to reduce processing time. There are several switch level simulators available; unfortunately these simula-

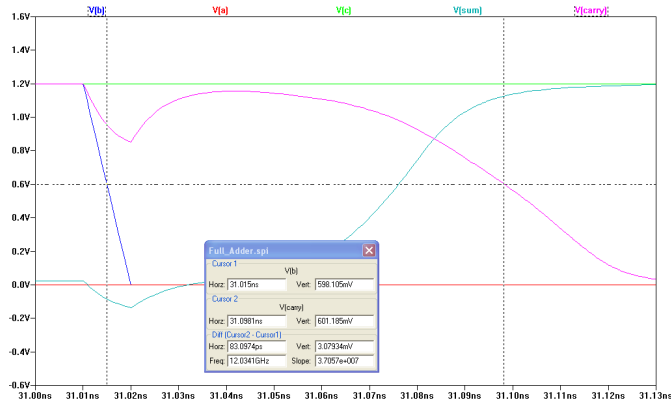


Figure 24: Conventional Adder Longest Propagation Delay

tors generally break down when feed back paths are introduced to the circuit, or do not keep track of charges used for dynamic data storage. Further more, these simulators tend to be inaccurate in terms of real propagation delays. To overcome this problem, a simulator was written that uses user defined models to simulate the behavior of a large scale design.

The models are written to estimate propagation delays in the circuit. The idea being that precalculating these propagation delays would keep the simulation accurate, while calculating the answers quickly. Unlike traditional CMOS designs, the propagation delays vary depending on the combination of inputs to the gate. In the case of the self-timed carry logic circuit, an input of $A = 1, B = 1$ will result in a significantly different propagation delay to an input of $A = 1, B = 0, C = 1$, due to the longer pull down chain. In a traditional gate there would normally be a difference in delay depending on whether or not the output was switching to a one or zero.

Each gate that is to be instantiated in the final circuit requires a model file. This model file has the following template (Table 4).

Each cell is given a name, in the `< name >` section, and the inputs and outputs are defined. “TT:< d >” denotes a start of a “truth table” for the output with name < d >. “Def:< a >,< b >” means that for this truth table only inputs < a > and < b > are defined. Each line of the truth table is then written, starting with inputs < a >=< b >=< c >= 0, then < a >= 0, < b >= 0, < c >= 1 right through to < a >= 1, < b >= 1, < c >= 1. Each line has the output value, either 0,1,x (null output) or h (hold current value) followed by the propagation delay, in ps, for that value. For the reset of the output < d >, a truth table would need to be defined for no defined inputs. Currently, the delays need to be estimated from SPICE simulations. As an example of a complete gate, Table 5 definition shows the definition of a two input AND gate.

In this definition, the propagation delay for an input of $A = 1, B = 1$ is 10ps faster than all other inputs. The reset delay of this circuit is 40ns, and requires all inputs to be undefined in order to reset.

The definition of the self-timed full adder circuit is quite long (see appendix B).

The interconnection of several gates is described in a separate file to the gate

Table 4: Large Scale Simulator - Gate Input File Template

```

Cell: < name >;
Inputs:< a >,< b >,< c >,...;
Outputs:< d >,< e >,< f >,...;
TT:< d >;
Def:< a >,< b >,< c >,...;
Table:
0,30;
0,30;
1,20;
1,20;
...;
TT:< d >;
Def:< a >,< b >,...;
0,40;
h,0;
h,0;
1,40;
...;
End:
< Comments >

```

Table 5: Large Scale Simulator - Example AND Gate Input File

```

Cell:AND2;
Inputs:A,B;
Outputs:Out;
TT:Out;
Def:A,B;
Table:
0,30;
0,30;
0,30;
1,20;
TT:Out;
Def;
Table:
x,40;
End:
This is a simple self timed,
2 input, AND gate

```

definitions. The definition of this file is not particularly important in regard to the accuracy and performance of the simulator, hence only an example of a four bit adder is shown here (Table 6).

Table 6: Large Scale Simulator, Four Bit Adder Circuit File

Inputs:A3,A2,A1,A0,B3,B2,B1,B0,C0;

Outputs:Sum3,Sum2,Sum1,Sum0,C4;

New:Full_Adder;
Name:adder_n=0;
A0,B0,C0,Sum0,C1;

New:Full_Adder;
Name:adder_n=1;
A1,B1,C1,Sum1,C2;

New:Full_Adder;
Name:adder_n=2;
A2,B2,C2,Sum2,C3;

New:Full_Adder;
Name:adder_n=3;
A3,B3,C3,Sum3,C4;

End: This test circuit implements
a 4 bit full adder

One of the major problems with this simulator is that it does not easily model isochronic forks within the internals of each gate. Rather it assumes that all forks within the gate are isochronic, and hence the gate will operate correctly. However, it may be possible to describe the circuit at the transistor level, through more complicated gate definitions and interconnections. In the form described here the simulator should only be used to gauge the performance of a system, and not the functionality of the system.

4.1.4 Automated Circuit Creation

To improve design to implementation time of self timed circuits, a program was created to automatically generate a net list of transistors using some of the optimizations mentioned earlier. Currently the program only works for single output gates, hence the optimizations relating to the weak conditions 1, 2 and 4 do not apply. In addition, the current version only implements path sharing at the last level of transistors. All other optimizations are implemented in the program. The program currently requires only an input of a truth table. In future versions that implement all of the optimizations, it is expected that some user intervention will be required.

For example, in the case of the full adder the user will have to specify which of the inputs is expected to be later than the others, otherwise the automatic creation program may make the optimizations on the B input rather than the carry input. Since it is not evident in the truth table specification that the carry input would arrive later, some additional information is obviously needed to improve the performance of the circuit.

The program was originally designed to create the static feedback paths

as well as the behavioral logic. However, this was removed due to errors in the method that have since been rectified and hence only dynamic circuits are created. Future versions will implement the Sutherland style feedback, with the option of creating dynamic cells.

The dynamic circuits created by the program have been tested and confirmed to be functionally correct given the optimizations currently implemented. Currently the circuits do not perform as well as the hand optimized circuits, i.e. the automatically created full adder will not have logarithmic average performance and also the pull up chain is longer.

4.2 FPGA

The main advantage of FPGAs is the fast design to implementation time. Since the design is simply downloaded onto the FPGA, there are no fabrication issues to worry about. However, for self timed designs, FPGAs can be restrictive. The layout of the FPGA is usually optimized for heavily pipelined synchronous designs. In traditional FPGAs each logic cell usually contains a lookup table followed by an edge triggered flip flop and/or a level triggered latch. In contrast, a self timed circuit is usually some logic, surrounded by Muller C elements or is entirely described by a net list of MOS transistors. Neither of these two descriptions simply map to the logic elements found in FPGAs.

However, all is not lost. FPGAs that contain level triggered latches and/or RS latches can be used to implement self timed circuits. The difficulty becomes the control of timing restrictions such as isochronic forks. In addition the synthesis and mapping tools do not recognize the processing paths in the self timed circuits and hence synthesis and placement of logic is sub-optimal, and in some cases will cause the system to fail. All these problems can be overcome through careful design and only place small restrictions on what can be implemented successfully in an FPGA.

The main disadvantage in FPGAs is that they can not be easily used to compare the performance of the synchronous design with the asynchronous design. Using the standard synthesis tools, many standard operations are heavily optimized for synchronous designs. For example, some FPGAs have dedicated carry logic, which can not be utilized by a self timed circuit, and which greatly increases the performance of synchronous adders. Therefore it becomes difficult to fairly compare the performance of a synchronous adder and a self timed adder on an FPGA.

4.2.1 Implementation of Self-Timed Circuits Using an RS latch with precedence

This section demonstrates how to implement a self timed circuit using an RS latch with precedence. The circuit remains relatively small and is more robust in regard to timing constraints than an equivalent circuit using either an RS latch without precedence, or using a Muller C element. This section will also demonstrate one scenario where the circuit will fail if some forks are not implemented as isochronous forks. Finally some aspects of implementing the circuit in an FPGA are analyzed.

Motivation: FPGAs generally do not map directly to the sorts of circuits commonly found in Quasi-Delay insensitive (QDI) designs. Most designs of quasi-delay insensitive circuits are either made up of Muller C elements, logic surrounded by Muller C elements or are transistor circuits in the form of a precharge half buffer (PCHB) circuit. Though an FPGA can implement any Boolean function in combinational logic, they do not, generally, contain Muller C elements. In order to produce QDI circuits something else on the FPGA must be used in order to replace the Muller C element.

Though a Muller C element could be described and implemented using a high level hardware description language, it was decided that it was likely a better implementation method could be found that more simply uses the resources available on FPGAs. The problem became whether or not the logic elements available could be used to replace the function of QDI circuit primitives.

The result of this effort is a methodology for creating QDI circuit with a very similar operation to the PCHB. This methodology has been used and successfully implemented on an FPGA (using synthesized control signals) and results in circuits that can outperform synchronous equivalent circuits. Though not fully examined in this section, it appears as though these techniques could be applied in a more general way to create entirely self timed processing systems on an FPGA.

Method of Implementation: The circuits created follow the same general layout of the PCHB. The calculating circuitry and reset circuitry have been replaced with logic. The state-holding circuitry has been replaced with an RS latch with precedence. Which input of the RS latch has precedence is not important, the only requirement is that one of them have precedence and that the precedence does not change. We will see later that the precedence property of the RS latch is important in removing a timing constraint that would otherwise have to be designed for.

For the purposes of simplicity we will create the simplest circuit possible in a PCHB, the non-inverting buffer, with input X and output Y . The operation of the circuit is the same on both X^0 and X^1 inputs and hence we will only look at half the circuit, the X^1 circuit producing the Y^1 output. Though this circuit can be replaced by a single wire, we will still implement it to demonstrate the operation of the circuit.

The calculating circuitry has the following PR:

$$\begin{aligned} X^1 &\mapsto S^1 \uparrow \\ \neg X^1 &\mapsto S^1 \downarrow \end{aligned}$$

The reset circuitry has the following PR:

$$\begin{aligned} X^1 &\mapsto R^1 \downarrow \\ \neg X^1 &\mapsto R^1 \uparrow \end{aligned}$$

The RS latch with precedence has the following PR:

$$\begin{aligned} R^1 \wedge \neg S^1 &\mapsto Y^1 \downarrow \\ S^1 &\mapsto Y^1 \uparrow \end{aligned}$$

It can be seen from these PR that when X^1 is asserted the output Y^1 will become asserted. The PR are the same for the X^0 circuitry.

Key Differences between this Style of Circuit and Standard PCHB:

The main difference between this and a traditional PCHB circuit is the precedence of the latch. The precedence of the latch means that there can be an increased differential delay between the operation of the reset and set circuitry. A small differential delay is not going to effect the operation of the circuit at all, while in the traditional PCHB the output of the circuit would become a value that is neither 0 nor 1 for a significant amount of time, making it difficult to predict how connected circuits would respond to this value.

Isochronic Fork Constraint: There is a constraint on the input signals to the circuit. There is a fork on the input signals that are sent to the reset circuitry and the calculating circuitry. Because of the precedence of the latch, the circuit will still operate correctly even if this fork is not implemented as an isochronic fork. The logic in the calculating circuitry contains a fork, these forks must be implemented as isochronic forks. In addition, the forks on the inputs between the Y^0 and Y^1 circuits, where Y is the output of the gate, otherwise the circuit may again behave incorrectly.

Example of Failure due to non isochronic fork: Imagine the negative rail of an XOR gate has been implemented as shown in Figure 25 below. In the calculating circuitry the fork on the input A^1 is not isochronic. Imagine that the input is set to $A = B = 1$, the set circuitry is therefore producing a one and the circuit produces a value of 1. A and B are then set to 0, while $*A^1$ remains 1 because of the large delay on that line. The set signal goes low and the reset signal goes high, hence the output switches to low. The input can now be changed to the next value. The next value is $A = 0, B = 1$. Since $*A^1$ is still one, the output of the circuit goes high even though the input indicates that it should not. Therefore the circuit has failed because of the non isochronic fork on A^1 .

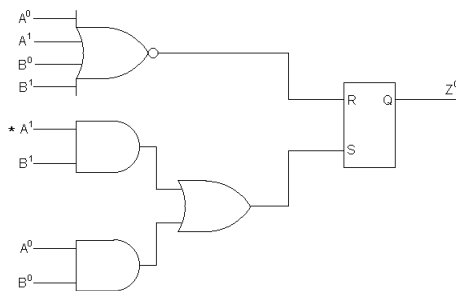


Figure 25: Negative Rail of a Self Timed XOR gate using and RS latch

Key Constraints in an FPGA: The problem with FPGAs is that delays along routed wires can become large quickly, especially when signals have a high fan out. An advantage of FPGAs is that since look up tables implement the logic functions, fan out can be reduced over implementing the function with the interconnection of logic gates. This also means that the number of controllable forks on signals can be reduced. Controllable forks are those introduced by the

interconnection of logic elements, these are controlled by the router and logic specification. Non-controllable forks are those forks that are internal to the logic elements, since the structure of logic elements is generally unknown we can only assume these forks are isochronic.

Effects of "Fan In" in an FPGA implementation: As mentioned earlier, FPGAs implement required logic functions using look up tables. This means that the number of controllable forks is reduced over that of conventional gate logic. Consider the logic function $(A \wedge B) \vee (A \wedge C)$ implemented as two AND gates and one OR gate. The input A has a fork on the input of the logic, and if this function was to be used as part of a PCHB circuit this fork would have to be isochronic. However, if this function were implemented as a lookup table, then there would be no (controllable) fork on the input A and hence the fork would not have to be designed.

However, one problem with FPGAs is that there is a limit to the number of inputs to any lookup table. On most FPGAs the limit is four inputs; in dual rail encoding this equates to only two bits. For more than four inputs, the function must be split into multiple lookup tables. This then becomes a problem for the designer, as a number of (controllable) forks may be created by the implementation of the required function using lookup tables. Luckily, for the designer, the synthesis software should try to reduce the fan out of signals, and hence the number of forks will be reduced as much as possible. However, the designer will have to make sure that any controllable forks that are not eliminated are isochronic and unfortunately the creation of a fork on a signal makes the control of the delay much more difficult.

4.2.2 Current Work

A number of self timed circuits designs have been completed and implemented on an FPGA. Unfortunately these circuits simply cannot compete with the equivalent synchronous circuits.

Large, optimized, calculating circuitry has been created, such as adders (up to 64 bits in size), multipliers (8x8 bit) and dividers (8/8 bit), as well as other boolean function gates.

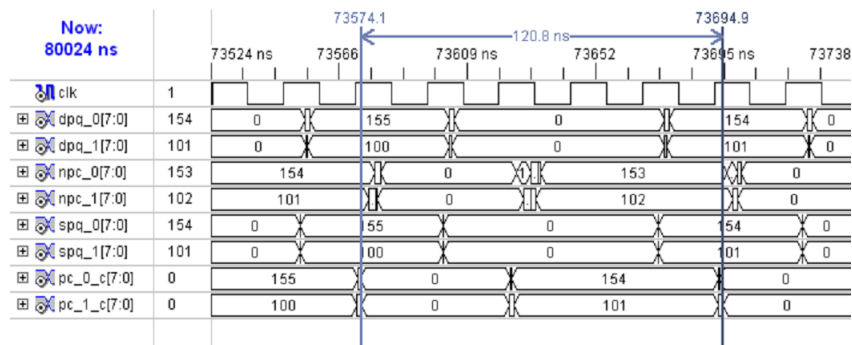


Figure 26: Timing of Asynchronous Counter with RAM

Currently the largest performance obstacle has been the creation of a self timed RAM block. If the self timed description of the RAM is synthesized a large amount of logic element resources are required. With a RAM larger than a few words the number of resources required become too large to implement. To avoid this problem the onboard synchronous RAM is “wrapped” with logic such that the signalling of the outputs is identical to the expected operation of a self timed RAM block. The correct operation of any self timed that uses this “wrapped” RAM, will function regardless of the clock frequency, provided it is low enough that the synchronous logic operates correctly.

The largest processor like structure that has been designed so far is an 8 bit program counter that writes the value of the counter to RAM at the location of the program count. The average performance of this system is around 8Mhz when implemented in a Virtex II pro FPGA of speed grade -5, however the RAM is by far the biggest bottleneck in the system. A timing diagram of a full calculation cycle is shown in Figure 26. It should be noted that there is a clock input to this circuit, this clock signal is only used to clock the asynchronously wrapped synchronous RAM.

5 Appendix A: Limitations to CMOS DI Circuits

Hypothesis: There can be no DI circuit in CMOS other than a single wire.

Proof: It has been shown that only inverters, non-inverting buffers and C elements admit a delay insensitive implementation [2]. However the PR rules to describe these circuits are insufficient as they do not include the forks internal to the gates. These forks are required to operate the complementary transistor arrangements in CMOS.

Below is a circuit of a C element (Figure 27), assume that the fork for the input B has not been implemented isochronously, that is the delay along B^1 is longer than the switching delay of the element. Assume that the input to A and B will only change once the last transition has been detected (ie four phase hand shaking). Now, A and B remain 0 for a long time. The output is zero and hence the input can change. The input changes such that $A = B = 1$. The output switches to one, so the input can change again. A switches to zero and B remains one. B^1 is still zero at this point, hence the two pull up transistors switch and the output goes low when the input signals are 0 and 1.

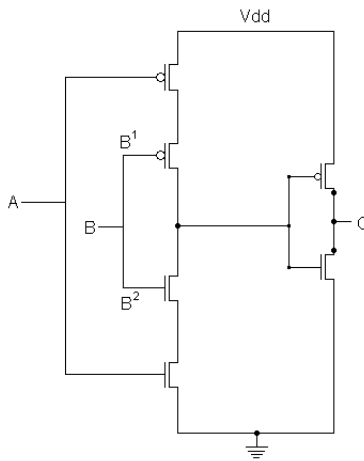


Figure 27: CMOS Implementation of a Dynamic C Element

Since the correct operation of the circuit requires that a fork be implemented as an isochronous fork, the circuit is in fact quasi-delay insensitive. The same can be done for the inverter; however, instead it will reach a forbidden state, ie a voltage that is neither zero nor one. If the voltage is close to the threshold of other gates, it may be difficult to determine how any connected circuits react to this value, making the circuit non delay insensitive. The complementary nature of CMOS means that all inputs require a fork and hence since the C element cannot be implemented with non isochronic forks, there is no DI CMOS implementation of a Muller C element. Since [2] shows that there are no possible implementations for gates other than the C element and Inverter that are DI, then therefore there is no CMOS delay insensitive gate other than a wire.

However, really this finding is insignificant since it is so simple to create the isochronic fork required for the operation of the C element, that it is nearly impossible to make it non-isochronic. It is unlikely a designer would make such a design flaw.

6 Appendix B: Full Adder Circuit Definition

Table 7: Large Scale Simulator, Full Adder Gate File

```
Cell:Full_Adder;  
Inputs:A,B,Cin;  
Outputs:Sum,Cout;
```

```
TT:Sum;  
Def:A,B,Cin;  
Table:  
0,56;  
1,56;  
1,56;  
0,56;  
1,56;  
0,56;  
0,56;  
1,56;  
TT:Sum;
```

```
Def:A,B;  
Table:  
x,90;  
x,90;  
x,90;  
x,90;
```

```
TT:Sum;  
Def;  
Table:  
x,90;  
TT:Cout;  
Def:A,B,Cin;  
Table:  
0,40;  
0,40;  
0,56;  
1,56;  
0,56;  
1,56;  
1,40;  
1,40;
```

Table 8: Large Scale Simulator, Full Adder Gate File Continued

```
TT:Cout;  
Def:A,B;  
Table:  
0,40;  
h,0;  
h,0;  
1,40;
```

```
TT:Cout;  
Def:Cin;  
Table:  
x,106;  
x,106;
```

```
TT:Cout;  
Def;;  
Table:  
x,106;
```

```
End: This is a simple Full Adder
```

References

- [1] S. Hauck, *Asynchronous design methodologies: an overview*, Proceedings of the IEEE **83** (1995), 69–93.
- [2] A.J. Martin, *Limitations to delay-insensitivity in asynchronous circuits*, Sixth MIT conference on Advanced research in VLSI (Boston, Massachusetts, United States), 1990, pp. 263–278.
- [3] ———, *Asynchronous datapaths and the design of an asynchronous adder*, Formal Methods in System Design **1** (1992), no. 1, 117–137.
- [4] A.J. Martin, S.M. Burns, T.K. Lee, D. Borkovic, and P.J. Hazewindus, *The design of an asynchronous microprocessor*, ARVLSI: Decennial Caltech Conference on VLSI (C.L. Seitz, ed.), MIT Press, 1989, pp. 351–373.
- [5] A.J. Martin, A. Lines, R. Manohar, M. Nystrom, P. Penzes, R. Southworth, U. Cummings, and Tak Kwan Lee, *The design of an asynchronous mips r3000 microprocessor*, Advanced Research in VLSI, 1997. Proceedings., Seventeenth Conference on, 1997, TY - CONF, pp. 164–181.
- [6] Carver Mead and Lynn Conway, *Introduction to vlsi systems*, Addison-Wesley, Reading, Mass., 1980, Carver Mead, Lynn Conway. Includes bibliographical references and index.
- [7] T. Nanya, Y. Ueno, H. Kagotani, M. Kuwako, and A. Takamura, *Titac: design of a quasi-delay-insensitive microprocessor*, Design & Test of Computers, IEEE **11** (1994), no. 2, 50–63.
- [8] C.L. Seitz, *System timing*, Introduction to VLSI systems, 1980.
- [9] A.T.M. Shafiqul Khalid, *A fast optimal cmos full adder*, Circuits and Systems, 1996., IEEE 39th Midwest symposium on (Ames, IA, USA), vol. 1, 1996, pp. 91–93.
- [10] M. Shams, J.C. Ebergen, and M.I. Elmasry, *A comparison of cmos implementations of an asynchronous circuits primitive: The c-element*, International Symposium on Low Power Electronics and Design (Monterey, CA, USA), 1996, pp. 93–96.
- [11] S. C. Smith, R. F. DeMara, R. F. Yuan, D. Ferguson, and D. Lamb, *Optimization of null convention self-timed circuits*, Integration, the VLSI Journal **37** (2004), no. 3, 135–165.
- [12] G.E. Sobelman and K.M. Fant, *Cmos circuit design of threshold gates with hysteresis*, IEEE International Symposium on Circuits and Systems, ISCAS '98 (Monterey, CA, USA), vol. 2, 1998, pp. 61–64.
- [13] I.E. Sutherland, *Micropipelines*, Communications of the ACM **32** (1989), no. 6, 720–738.
- [14] J.A. Tierno, A.J. Martin, D. Borkovic, and T.K. Lee, *A 100-mips gaas asynchronous microprocessor*, Design & Test of Computers, IEEE **11** (1994), no. 2, 43–49.

- [15] K. van Berkel, *Beware the isochronic fork*, Integration, the VLSI Journal **13** (1992), no. 2, 103–128.
- [16] C.G. Wong, A.J. Martin, and P. Thomas, *An architecture for asynchronous fpgas*, Field-Programmable Technology (FPT), 2003. Proceedings. 2003 IEEE International Conference on, 2003, TY - CONF, pp. 170–177.
- [17] J.V. Woods, P. Day, S.B. Furber, J.D. Garside, N.C. Paver, and S. Temple, *Amulet1: an asynchronous arm microprocessor*, IEEE Transactions on Computers **46** (1997), no. 4, 385–398.