A Comparison of Synchronous and Asynchronous Digital Design

Sohum Datta, EECS, UC Berkeley
sohumdatat@berkeley.edu

Abstract—A simple experiment is described which compares synchronous and asynchronous implementations of a three-stage integer pipeline. The aim of the experiment was to replicate some results of similar previous works and gain insights to the fundamental difference between the two paradigms. The asynchronous implementation was about 19% worse in area and 64% worse in throughput than the synchronous design. A derivation of the Muller C gate based on latches and a proof of linearity of the lower bound on energy scalability with the entropy of specification of asynchronous circuits is also provided.
This document is self-contained for the experimental setup. A summary of design methods for asynchronous processors is also provided. However, the proofs in appendices are brief and important results used are only stated without derivations.

Index Terms—Asynchronous Circuits, Field-Programmable Gate Arrays, Self-timed Pipelines, Quasi-Delay Insensitive Circuits, prefix codes, process algebra.

1 Introduction

Asynchronous circuits have no clock. The sequencing of dependent computations is done via *handshaking*.

The main advantage of asynchronous circuits is the inherent *slack-free* nature. The worst-case correctness behaviour for synchronous systems is replaced by *average*-case behaviour in asynchronous systems.

Synchronous circuits consume energy strongly related to the *largest combinational delay* in its path. By contrast, asynchronous circuits of large systems can be made to consume energy within a constant factor of the circuits entropy. A proof is outlined in Appendix B.

However, there are some serious pitfalls of this new paradigm. For asynchronous circuits, Computer Aided Design (CAD) infrastructure is still in its infancy. Estimating the average-case delay of a circuit is an NP-Hard problem [1]. It can also depend on component delays in unexpected ways – a faster gate can lead to overall slower circuit! (For an example, please refer [1], pg. 7). Finally, verification of asynchronous circuit can be very difficult [2]. A substantial family of circuit classes cannot be tested for certain types of errors [3].

2 Asynchronous Circuits on FPGA

Prototyping asynchronous circuits on Field-Programmable Gate Arrays (FPGAs) is a recent trend. As designers turn to explore asynchronous circuits, the cheaper platform which is widely deployed in today's data-centers serves better [4]. Moreover, many modern systems features (such as Multi-Processor SoC, ARM IP Cores) can be easily integrated with the asynchronous circuit implemented.

This is the final report for the term project of EE241B: Advanced Digital Integrated Circuits offered in Spring 2017 by Prof. Borivoje Nikolic at UC Berkeley, CA 94704, United States. Special thanks to Vignesh Iyer and Prof. Elad Alon, GSI and Instructor of EECS151/251 Spring 2017. The experiments would not have been possible without their equipment and support.

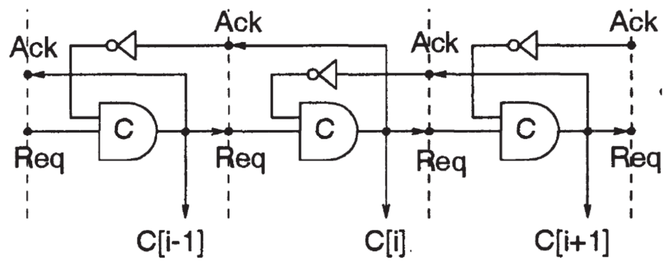

Fig. 1. Sutherland's basic micro-pipeline for asynchronous circuits.

There are several ways to design asynchronous circuits on commercial FPGAs [5] [6] [7] [8] [9]. An important choice is to design Muller C Gate using FPGA resources. Appendix A describes the design used.

3 Self-Timed Pipelines

Self-timed pipelines ensure progress without the help of external signals (clock). Unlike most modern processors where deep pipelines result in large control overhead (synchronous stall signals), free elasticity in asynchronous pipelines provide for data-driven throttling of speed and power.

The basic micro-pipeline was developed by Ivan Sutherland (fig. 1) [10]. Its main pitfall is the fact that the pipeline evaluates *at most half* of its stages at any instant.

Another family of self-timed circuits is composed of alternating computation and interconnection blocks [1]. Computational blocks use *Dynamic Cascode Voltage Switch Logic* (DCVSL), a four-phase logic style, and the interconnection blocks control handshaking.

A good reference for self-timed pipelines is [11].

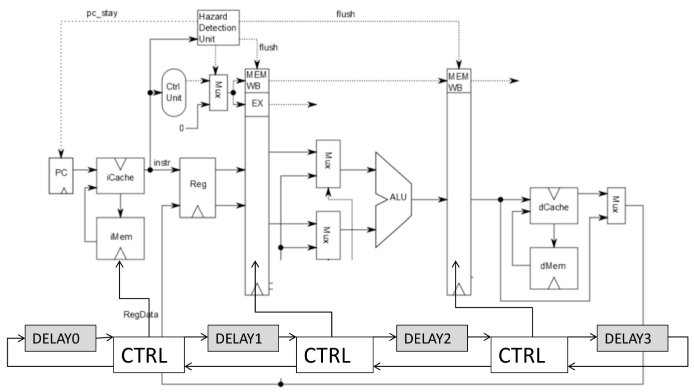

Fig. 2. The three-stage asynchronous pipeline implemented on Xilinx Virtex-5.

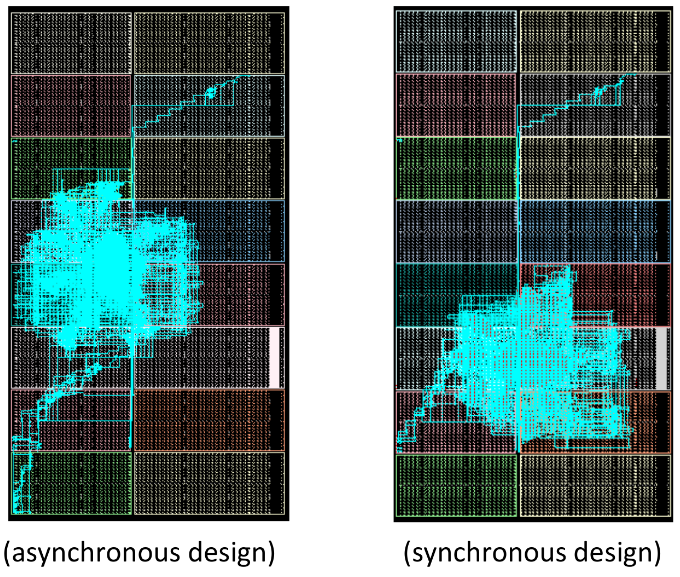

Fig. 3. The routed designs: asynchronous (left) and synchronous (right).

4 The Asynchronous Pipeline Setup

A three-stage pipeline of an in-order RISCV32UI integer processor is implemented. A synchronous and asynchronous version of the same pipeline was written and compared. The asynchronous pipeline (as in fig. 2) has stage registers implemented by flip-flops but clocked by the micro-pipeline (fig. 1). Since the stage delays may be mis-matched, *no data forwarding or speculative execution* is performed in either design.

The EECS151: FPGA Lab resource Xilinx Virtex-5 FPGA was used for the experiments. Its basic assembly tests were used for verification and debugging, and the `mmult` program for benchmarking and comparing designs.

In order to implement the delay elements (DELAY0, ... DELAY3 in fig. 2), the Virtex 5 primitive IODELAY was used. IODELAY can provide user-adjusted fixed delay to Input/Output ports or to the internal fabric. The resolution of the delay line used is about 31 ps. at 50 MHz control frequency. Fixed tap delay requires the use control primitive IODELAYCTRL (one each in 16 clock regions).

5 Simulation Methodology

Preliminary Static Timing Analysis (STA) without IODELAY primitives revealed that stage delays went up to 3.8 ns (i.e. about 3 IODELAYs). Virtex 5 requires *all clock signals to be buffered* before being clocked into a latch. Since the buffering is performed automatically (depending on the number of clock domain crossings), the user cannot limit the distance between micro-pipeline gates and IODELAYs without manually routing the entire design.

This was a significant problem: a change in the number of IODELAYs for a stage will change the routed design unpredictably. Hence, number of IODELAYs for *each stage* must be tuned correctly at once.

With a lack of alternatives, a manual iteration schedule was followed:

1) Set the number of IODELAYs for each stage.
2) Find out the stage delays after tool-flow using STA.
3) If the pipeline path is *within limits* of the delay provided by the current number of IODELAYs for all stages, continue. If for any path the STA delay is longer than the maximum delay provided by IODELAYS, increment the number of IODELAYs. If there are many such stages, increment for the stage with least number of IODELAYs. Go to Step 2.
4) If you have Static timing delays within the limits of IODELAYs, set the correct tap delay for each stage.
5) Perform assembly tests. If fail, return to Step 1 with the stage with smallest total IODELAY delay incremented. If the IODELAYs are already at their maximum delay for that stage, a new IODELAY must be added.
6) Benchmark `mmult` to obtain the effective throughput.

6 Results

Fig. 3 shows the routed wires and SLICEs for the design. The asynchronous design has more wires towards the left edge – to connect to IODELAYs forced to be at the chip (IO) boundary.

Resource	Sync.	Async.	Change
SLICE Regs.	1413	1446	+33 (23 %)
SLICE LUTs	2358	2395	+37 (16 %)
BUFG/BUFGCTRL	2	5	+3
Throughput	67.4 MIPS	24.1 MIPS	- 43.3 (- 64 %)

Clearly, the `mmult` throughput of asynchronous design was lower than 50 % of synchronous since only half of the micro-pipeline can be full. Rest of the loss can be accounted for by over-estimation of IODELAY values (due to manual tuning) and extra latency to IODELAY at the edge of chip.

Xilinx ML505/6/7 development board does not have supply resistances to measure current. Hence, power measurements could not be performed. The difficulties in designing the asynchronous system on a commercial synchronous machine highlights the mismatch between conventional tools and asynchronous design.

Appendix A The C Gate and the Virtex-5 FPGA

The Muller C (*"Completion"*) gate is a sequential circuit element which switches only when all its inputs have the same value. When inputs disagree, the gate output retains its previous value. Any multi-input C gate can be expressed as a cascade of 2-input C gates (henceforth MULLER). This appendix describes the MULLER gate and derives its realization on the Xilinx Virtex-5 FPGA.

Boolean expressions of sequential elements need special notation to distinguish between the current value and next value of signals. We denote both the signal and its current value by the signal name. Hence, a denotes the value of signal 'a' at the current instant. Let a' denote its value at the next instant and $\overline{a}$ its logical negation. a'' will be the value of 'a' after a'. Also, let $\oplus$, $\cdot$ and $+$ represent logical XOR, AND and OR respectively.

The MULLER gate can be modelled as a latch which is transparent to one of its inputs only when both of them agree:

$$\mathtt{MULLER}(a,b)' = \overline{(a\oplus b)}\cdot a \;+\; (a\oplus b)\cdot \mathtt{MULLER}(a,b) \quad (1)$$

A.1 Xilinx Virtex-5 Primitives

The Xilinx Virtex-5 FPGA contains 4 storage elements in each SLICE. A storage element can be configured as a level-sensitive latch with input driven by a LUT in the same SLICE. The latch is transparent when the control signal clock `CLK` is LOW. Any latch element can either be instantiated directly as a *Transparent Data Latch with Asynchronous Clear and Preset and Gate Enable* (LDCPE) primitive or inferred by the Xilinx Synthesis Tool (XST).

Although XST enables optimization across modules, inferred synthesis may combine handshake stages to produce unwanted behaviour. For guaranteed functionality, the LDCPE primitive must be used. Its inputs are asynchronous clear (`CLR`) and preset (`PRE`), gate enable (`GE`), gate (`G`) and data (`D`):

$$\mathtt{LDCPE}' = \overline{\mathtt{CLR}}\cdot(\mathtt{PRE}+(\mathtt{GE}\cdot\mathtt{G})\cdot\mathtt{D})+\overline{\mathtt{CLR}}\cdot(\overline{\mathtt{GE}\cdot\mathtt{G}})\cdot\mathtt{LDCPE} \quad (2)$$

A.2 C Gate using RS and SR Latches

A straightforward way of synthesizing MULLER on Virtex-5 would be to program a LUT for the XNOR gate $\overline{(a\oplus b)}$ and connect its output to `CLK`. However, this will be prone to spurious input transitions. One of the latch inputs ($\overline{a\oplus b}$ as `enable`) will involve additional delay of an LUT and arrive later than the other input – thus exhibiting wrong behaviour. It is mandatory for a latch `clk` signal to be driven by a buffer in Virtex 5; thus the latch and LUT are on separate slices exacerbating the delay mismatch.

Using two latches reduces the risk of an output transition due to glitching. A well known implementation of MULLER using RS and SR latches due to Murphy [?] can be used for this purpose.

Both SR and RS latches have two inputs `set` (`S`) and `reset` (`R`). SR latch is equivalent to the RS latch with `R` and `S` inputs interchanged and the output inverted.

$$\mathtt{RSLatch}(\mathtt{S},\mathtt{R})' = \overline{\mathtt{R}}\cdot(\mathtt{S}+\mathtt{RSLatch}) \quad (3)$$

$$\mathtt{SRLatch}(\mathtt{S},\mathtt{R})' = \mathtt{S}+\overline{\mathtt{R}}\cdot\mathtt{SRLatch} \quad (4)$$

For brevity $\mathtt{MULLER}(a,b)$ is denoted by c. One can easily simplify eqn. 1 (using identity $x+\overline{x}\cdot y = x+y$) to obtain:

$$\begin{aligned}\mathtt{MULLER}(a,b)' &= c' \\ &= (a\cdot b+\overline{a}\cdot\overline{b})\cdot a + (a\cdot\overline{b}+\overline{a}\cdot b)\cdot c \\ &= a\cdot(b+\overline{b}\cdot c)+\overline{a}\cdot b\cdot c \\ &= a\cdot c + b\cdot(a+\overline{a}\cdot c) \\ &= a\cdot b+(a+b)\cdot c\end{aligned} \quad (5)$$

To express MULLER in terms of SR and RS latches, observe that eqn. 5 contains only true values of inputs but `R` appears inverted in both SR and RS latches (eqs. 4 and 3). Hence, `R` of the latches will be $\overline{a}$ or $\overline{b}$. Only the expression for RS latch (eqn. 3) has a minterm containing both `R` and `S`, which will produce the minterm $a\cdot b$ in eqn. 5.

Therefore, the RS latch appears in the first stage producing the minterms $a\cdot b$ and $a\cdot c$ or $b\cdot c$ (depending on `R` being $\overline{a}$ or $\overline{b}$ respectively). The SR latch forms the second stage. Since `S` appears alone in its expression, we connect the RS output to `S` and the input other than `R` of previous stage to the `R` of the second stage.

Representing the RS output by p and the subsequent SR output by q, we have:

$$\begin{aligned}\mathtt{RSLatch}(\mathtt{S}=a,\mathtt{R}=\overline{b}) &= p' = a\cdot b+b\cdot p \\ \mathtt{SRLatch}(\mathtt{S}=p,\mathtt{R}=\overline{a}) &= q' = p+a\cdot q\end{aligned} \quad (6)$$

From above, we have $q'' = p' + a'\cdot q' = (a\cdot b+b\cdot p)+a'\cdot(p+a\cdot q)$. Assuming input steady state i.e. $a''=a'=a$ and $b''=b'=b$,

$$\begin{aligned}q'' &= a\cdot b+a\cdot p+b\cdot p+a\cdot q \\ &= a\cdot b\cdot(1+q)+(a+b)\cdot p+a\cdot q \\ &= a\cdot b+(a+b)\cdot p+a\cdot q+a\cdot b\cdot q \\ &= a\cdot b+(a+b)\cdot p+(a+b)\cdot a\cdot q \\ &= a\cdot b+(a+b)\cdot(p+a\cdot q) \\ &= a\cdot b+(a+b)\cdot q'\end{aligned} \quad (7)$$

This proves the equivalence of the latch pair (eqn. 6) to MULLER (eqn. 5).

It may appear that the heuristic used to connect the pair just happened to prove equivalent to MULLER. However, the circuit really was synthesized using reductions on a Signal Transition Graph (STG) specification [?]. The synthesis algorithm matches STG leaf nodes to the target specification – like our heuristic [1]. Note that the difference of SR and RS latch expressions (eqs. 4 and 3) helped using the heuristic: it would be much harder to conceive MULLER using 2 SR latches.

The LDCPE primitive can be easily configured as RS and SR latches. To obtain $\mathtt{LDCPE} = \mathtt{RSLatch}(\mathtt{S},\mathtt{R})$ by comparing eqs. 2 and 3, set $\mathtt{CLR}=\mathtt{R}$ and $\overline{\mathtt{GE}\cdot\mathtt{G}}=1$. Then, $\mathtt{PRE}=\mathtt{S}$. Similarly, for `SRLatch` set $\overline{\mathtt{CLR}}=1$, $(\mathtt{GE}\cdot\mathtt{G})=\mathtt{R}$, $\mathtt{D}=0$ and $\mathtt{PRE}=\mathtt{S}$.

Appendix B
Energy and Entropy of Asynchronous Circuits

This appendix sketches a proof of a lower bound on the scalability of energy dissipated by an asynchrnonous circuit in terms of the entropy of its specification.

B.1 Circuit Specification and Entropy

Analogous to a structural or behavioral description of a synchronous system, asynchronous circuits are specified using a process algebra formalism. Any circuit description requires an *alphabet* of signals $\mathbb{C} = \{\mathbb{I}, \mathbb{S}, \mathbb{O}\}$ composed of inputs ($\mathbb{I}$), outputs ($\mathbb{O}$) and internal *state* signals ($\mathbb{S}$).

An asynchronous circuit is described as a collection of communicating *processes* – sequential programs executing simultaneously on separate machines and synchronizing by passing messages.

A *statement*, denoted by block letters except G, is a boolean assignment where a state/output signal is assigned the value of a boolean expression of any number of signals. $A; B$ is a sequential process where statement B begins only after A completes. A *guard* ($G, G_1, G_2, ...$) is a boolean condition composed of any number of signals, used to implement control flow. There are two control structures:

- **Deterministic Selection** Denoted by $[G_1 \rightarrow S_1 \,\Box\, G_2 \rightarrow S_2 \,\Box\, ...\Box\, G_n \rightarrow S_n]$ where at most one guard $G_1 ... G_n$ is true at any instant. The program waits until a guard becomes true. If G_i is true, only S_i executes.
- **Non-Deterministic Selection** Denoted by $[G_1 \rightarrow S_1 \mid G_2 \rightarrow S_2 \mid ...\mid G_n \rightarrow S_n]$. Same as Deterministic Selection, except when more than one guards are true, any one of them is selected. The selection criteria is not necessarily random and can be assumed to be demonic for the circuit correctness.

A *repetition* on a control structure is denoted by $'*[...]'$: the control structure is repeatedly executed until no guards are true. $S_1||S_2$ denotes *concurrent* execution, where the statements are executed in any order and ensures *weak fairness* (i.e. any action that is enabled to execute and stays enabled will eventually execute).

Any of the guarded statements $S_1 ... S_n$ in the examples above could be control structures, their repetitions, sequential processes or concurrent processes themselves. Finally, for message passing, a special data structure *channel* (denoted by $\mathbf{1}, \mathbf{2}...$) is used. At any instant, at most one statement shall read from or write to a channel. A read statement $\mathbf{1}?v$ reads the value from the channel and stores it in signal v. A write statement $\mathbf{1}!G$ evaluates the boolean expression G and write the value to channel $\mathbf{1}$. A read and write statement waits till channel is full or empty respectively.

We postulate that any asynchronous circuit can be completely described by a process P of the form:

$$P = *[[G_1 \rightarrow A_1; X_1 \Box\, G_2 \rightarrow A_2; X_2 \Box\, ... \Box\, G_n \rightarrow A_n; X_n]] \tag{8}$$

where $A_1 .. A_n$ are either **skip** or message passing statements, $X_1 ... X_n$ are simple assignments where the value assigned is a boolean constant [14]. This essentially means all computation by the circuit is captured by evaluation of guard conditions alone.

Let Z_i be a random variable for the ith statement executed by P. The probability distribution for Z_i can be numerically calculated assuming a prior input distribution or estimated by profiling actual traces of execution.

Definition 1 (Entropy). Let $Z = (Z_1, Z_2, ... Z_m)$ be a sequence of m discrete random variables over range S_m with joint probability mass function $\Pr(Z_1, Z_2, ... Z_m)$. The entropy of the sequence is

$$H(Z_1, Z_2, ... Z_m) = -\sum_{(z_{1...m}) \in S_m} \Pr(z_1 ... z_m) \log_2 \Pr(z_1 ... z_m) \tag{9}$$

Definition 2 (Entropy of a Process). The entropy of a process P which executes sequence of statements $Z_1, Z_2, ...$ is

$$H(P) = \lim_{m \rightarrow \infty} \sup \frac{1}{m} H(Z_1, Z_2, ... Z_m) \tag{10}$$

P has about n statements $X_{1...n}$ ($A_{1...n}$ are mostly **skips** as circuits often have a lot of signal-level parallelism), hence $Z_1, Z_2, ...$ can have values n distinct values. Therefore, their entropies are bounded $H(Z_1), ... H(Z_2) \leq \log_2 n$. Hence, the entropy of the process P exists as its a limit of the supremum of a bounded sequence:

$$\frac{1}{m} H(Z_1 ... Z_m) \leq \frac{1}{m}(H(Z_1) + ... + H(Z_m)) \leq \log_2 n \tag{11}$$

B.2 Energy Index of Asynchronous Circuits

A CMOS circuit has three main sources of energy dissipation: leakage currents, short circuit currents and dynamic switching currents flowing between the rails. An *operation* is defined as a *finite* sequence of input transitions within a finite time. The input values may not be provided, only the number of transitions and their times must be specified [12]. Assuming most of the circuit is implemented using static CMOS (dynamic nodes are usually pulled to rail by weak feedback to prevent soft errors) and designed to have low leakage, we can estimate the total energy expended **in one operation** of the circuit:

$$E_T \approx E_{sw} = \sum_i n_i C_i V_{DD}^2 \tag{12}$$

Here, all nodes (indexed by i) have capacitance C_i and switch LOW to HIGH n_i times during one operation. The intermediate nodes of stacked transistors are ignored (they are usually smaller than drain capacitances and charge to values lower than V_{DD}).

It is worthwhile to note differences of this energy model to that of a synchronous circuit. The switching factor n_i is an integer rather than a probability. Also, a well-defined operation requires that outputs stabilize within finite time after the inputs switch. A special class of asynchronous circuits are required to avoid metastability [1].

The *energy index* of the circuit is defined as $K = \sum_i n_i C_i$. Energy indices are additive: the index of a circuit is a sum of energy indices of all its sub-circuits. The energy index of a CSP P is denoted by $C(P)$.

A central assertion of this model is that parallel sub-circuits with no synchronization require no extra energy. This is reasonable since no synchronization essentially means no extra circuitry, only extra wires for common signals.

Since asynchronous circuits compute only when required, the chances of a sub-circuit turning on and consuming switching energy depends on the inputs. If probability of sub-circuits P_1 and P_2 turning on during one operation are w_1 and w_2, then the total energy index is $K = w_1K_1+w_2K_2$.

The three possible ways of enforcing order in CSP are: sequential ordering (;), choice (deterministic and non-deterministic) and message-passing. Energy models of implementing each of these are supplied without proof, please refer [14] for the same.

Receiving and transmitting N-bit values over a channel requires energy index proportional to N. A guarded choice $P = *[[G_1 \rightarrow P_1 \square \, ... \, \square \; G_n \rightarrow P_n]]$ entails an extra energy index $\propto \log_2 n$ over executing the guarded statements concurrently $Q = *[[G_1 \rightarrow A_1]] \; || \; *[[G_2 \rightarrow A_2]] \; ... \; || \; *[[G_n \rightarrow A_n]]$. This is because we can implement the guarded choice using concurrent statements and extra $\log_2 n$ single-bit channels for message-passing [15].

Now consider the program P from from eqn. 8 with $n = 2$. The energy index can be evaluated as

$$\begin{aligned} C(P) = C(G_1) + C(G_2) + k\log_2 2 + \\ w_1 C(A_1; X_1) + w_2 C(A_2; X_2). \end{aligned} \tag{13}$$

Here k is a technology constant. Notice that the energy indices of all the guards are added irrespective of their probabilities being true.

Without loss of generality, assume $w_1 \geq w_2$. Now consider the following modification to the program:

$$Q = *[[G_1 \rightarrow A_1; X_1 \square \; [\overline{G_1} G_2 \rightarrow A_2; X_2]]] \tag{14}$$

The modified energy index is

$$\begin{aligned} C(Q) &= C(G_1) + w_1 C(A_1; X_1) + k\log_2 2 + \\ &(1-w_1)(C(G_2) + u + w_2 C(A_2; X_2)) \\ &\approx C(G_1) + (1-w_1)C(G_2) + k\log_2 2 + \\ &w_1 C(A_1; X_1) + (1-w_1) w_2 C(A_2; X_2) \\ &< C(P) \end{aligned} \tag{15}$$

Here the constant u is the fixed energy index to compute a logical negation and a logical AND of the result (usually negligible). Therefore, re-structuring the program P such that a more common guard executes before a less common guard saves overall energy.

For the general case (n guarded statements), assume P is re-structured to a tree of cascaded guarded statements Q as above. Let $Y_i = A_i; X_i$ denote the guarded statements of P. Then, for a particular Y_i of P there is a corresponding order of guard evaluation through the tree in Q which ends in the leaf node for Y_i. At a depth d in the tree of Q, energy will be dissipated by *all guard evaluations* at that depth. Let $S(Y_i) \subset \{G_1, G_2, ...G_n\}$ be the set of all guards that were evaluated by Q to reach leaf node for Y_i. Further, let $d(Y_i)$ denote the depth of the leaf node Y_i in the tree of Q, and let $n_1(Y_i), n_2(Y_i), ...n_{d(Y_i)}(Y_i)$ be the number of guards that were evaluated at depth 1, 2, ... $d(Y_i)$ respectively.

Then, the energy index for evaluating Y_i is given by

$$C_{Y_i}(Q) = \sum_{G \in S(Y_i)} C(G) + \sum_{j=1}^{d(Y_i)} k \log_2 n_j(Y_i) + C(Y_i) \tag{16}$$

and the energy index for evaluating $Y_{1...m} = Y_1, Y_2, ...Y_m$ by Q is $C_{Y_{1...m}}(Q) = \sum_{i=1}^{m} C_{Y_i}(Q)$.

Definition 3 (Energy Index per Operation). $\Pr(y_1, ...y_m)$ is the probability of the sequence of statements $(y_1, y_2, ...y_m)$ being executed by P. Then, the energy index **per operation** of Q is defined as the expected average energy index of a chain of operations

$$C(Q) = \lim_{m\to\infty} \frac{1}{m} \sum_{(y_1, ...y_m)} \Pr(y_1, ...y_m) C_{Y_{1...m}}(Q) \tag{17}$$

The above expression is really the average energy index in the limit of long chains. Assuming the statements $y_1, y_2, ...$ are independent and identically distrubuted, using the law of large numbers we have:

$$C(Q) = \mathbb{E}\Big[\sum_{G \in S(Y)} C(G) + C(Y)\Big] + k\,\mathbb{E}\Big[\sum_{j=1}^{d(Y)} \log_2 n_j(Y)\Big] \tag{18}$$

Here, $\mathbb{E}[\cdot]$ denotes the expectation with respect to the sequence of statements $(y_1, ...y_m)$ having joint probability distribution of $\Pr(y_1, ...y_m)$.

Now, the path to Y is determined by which of the guards resolved true at each depth. Since there are $n_i(Y)$ distinct guards at depth i, we can identify the true guard at depth i by a binary number $\log_2 n_i(Y)$ bits wide. The entire path is then a list of such numbers in total $\sum_{i=1}^{d(Y)} \log_2 n_i(Y)$ bits wide (call it $I(Y)$). Since all statements are leaf nodes, the binary nnumber $I(Y)$ cannot be a prefix to any $I(J), J \neq Y$. Hence, $I(Y)$ is a prefix code for Y, and $\mathbb{E}[\sum_{i=1}^{d(Y)} \log_2 n_i(Y)]$ is the average code-length. A fundamental result from by C. Shannon from Information Theory states that the average code length of any prefix code is atleast the entropy of the alphabet [13]. Hence $\mathbb{E}[\sum_{i=1}^{d(Y)} \log_2 n_i(Y)] \geq H(Y)$.

Therefore, we have

$$C(Q) \geq (\sum_{i=1}^{n} w_i C(Y_i) + \mathbb{E}[\sum_{G \in S(Y)} C(G)]) + kH(Y) \tag{19}$$

The first term in parenthesis is the *indispensable* cost for evaluating the guards and executing the statements. As such, asynchronous paradigm does not provide any major benefits.

The last term relates to the *scalability* of energy with the design size. Unlike synchronous systems where the energy scalability factor is linear (or polynomial) in size, an optimally designed asynchronous circuit scales linearly with the *entropy* of the system. This shows the *asymptotic* utility of asynchronous circuits for energy efficiency in large systems.

In this sketch, several details about methods of specification and transformation were not provided. This theory is very thoroughly developed in [15] [14].

References

[1] J. A. Brzozowski and C-J. H. Seger, *Asynchrnonous Circuits*, Monographs in Computer Science, Springer 1995.

[2] H. Hulgaard, S. M. Burns, G. Borriello, *Testing asynchronous circuits: A survey*, Integration, the VLSI Journal, Volume 19, Issue 3, 1995, Pages 111-131.

[3] J. A. Brzozowski and K. Raahemifar, *Testing C-elements is not elementary*, Proceedings Second Working Conference on Asynchronous Design Methodologies, London, 1995, pp. 150-159.

[4] J. Furushima, M. Nakajima, H. Saito, *Design of an Asynchronous Processor with bundled-data implementation on an commercial FPGA*, Informaticam Vol. 90, No. 4, 2016.

[5] M. Tranchero and L. M. Reyneri, *Exploiting synchronous placement for asynchronous circuits onto commercial FPGAs* , Proc. FPL, pp.622625, 2009.

[6] Q. T. Ho et al.,*Implementing Asynchronous Circuits on LUT Based FPGAs*, Proc. FPL, pp.3646, 2002.

[7] H. Saito et al., *A Floorplan Method for Asynchronous Circuits with Bundled-data Implementation on FPGAs*, Proc. ISCAS, pp.925928, 2010.

[8] K. Takizawa et al., *A Design Support Tool Set for Asynchronous Circuits with Bundled-data Implemen- tation on FPGAs*, Proc. FPL, pp.14, September 2014.

[9] N. Minas et al., *FPGA Implementation of an Asynchronous Processor with Both Online and Of- fline Testing Capabilities*, Proc. Async, pp.128137, 2008.

[10] I. E. Sutherland, *Micropipelines* Commun. ACM 32, 6 (June 1989), 720-738.

[11] T. E. Williams, *Analyzing and improving the latency and throughput performance of self-timed pipelines and rings,* [Proceedings] 1992 IEEE International Symposium on Circuits and Systems, San Diego, CA, 1992, pp. 665-668 vol.2.

[12] R. Manohar, *The entropy of traces in parallel computation* in IEEE Transactions on Information Theory, vol. 45, no. 5, pp. 1606-1608, Jul 1999.

[13] C. Shannon, *A mathematical theory of communication*, In Bell Systems Tech. J., volume 27, pages 379-423,1948.

[14] J. A. Tierno, R. Manohar, and A. J. Martin. *The Energy and Entropy of VLSI Computations* In Proceedings of the 2nd International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC '96). IEEE Computer Society, Washington, DC, USA, 1996.

[15] J. A. Tierno, *An energy-complexity model for VLSI computations* Dissertation (Ph.D.), California Institute of Technology. 1995.