An Evaluation of Asynchronous Stacks

Jo Ebergen, Daniel Finchelstein[*], Russell Kao, Jon Lexau, David Hopkins
Oracle Labs
Redwood Shores, USA

OL #: 2011-0143

May 18, 2011

Abstract

We present an evaluation of some novel hardware implementations of a stack. All designs are asynchronous, fast, and energy efficient, while occupying modest area. We implemented a hybrid of two stack designs that can contain 42 data items with a family of GasP circuits. Measurements from the actual chip show that the chip functions correctly at speeds of up to 2.7 GHz in a 180 nm TSMC process at 2V. The energy consumption per stack operation depends on the number of data movements in the stack, which grows very slowly with the number of data items in the stack. We present a simple technique to measure separately the dynamic and static energy consumption of the complete chip as well as individual data movements in the stack. The average dynamic energy per move in the stack varies between 6pJ and 8pJ depending on the type of move.

Keywords: asynchronous circuits, LIFO, stack, static energy consumption, dynamic energy consumption.

1 Introduction

Two often claimed benefits of asynchronous circuit design are the potential for high average-case performance and low power consumption. We try to demonstrate these benefits by the design of a small example: a fast, energy-efficient stack. Our asynchronous circuit consists of a control path and a data path. The control path steers the data, by means of splits and merges, and generates the signals that dictate the data movements in the data path, just like a clock dictates the data movements in a synchronous circuit. In the asynchronous circuit, however, the control path generates the move signals only when needed and where needed. To obtain speed, we use GasP circuits [12] in our control circuit. GasP circuits are energy efficient and offer a fast cycle time of 6 gate delays (FO4 delays). GasP circuits are often used to implement the

[*]Currently at NVIDIA

handshakes between pipeline stages. In general, GasP circuits can be used to implement compositions of general finite state machines. To reduce energy consumption, we try to minimize the number of data movements and any large fanouts, which have large loads associated with them. We present our designs in an informal way. For more technical details, we refer the reader to [4].

The only actions that the stack can perform are *put* and *get* actions, where a put action puts a data item on top of the stack and a get action gets the data item from the top of the stack. Furthermore, we require that the stack has overflow and underflow protection.

An obvious implementation is a SRAM with a top-of-stack pointer as illustrated in Figure 1(a). Because of the high densities of SRAMs, such an implementation consumes very little area per data item. For each data access, the address decoders of SRAMs decode a k-bit pointer address into charging 1 out of 2^k word lines, each of which has a high capacitive load. Because of the potentially large fanouts and the large loads in the SRAM, the access time and energy consumption per read or write of an SRAM, however, can be high [1]. Furthermore, the access time and energy consumption often grows with the size of the RAM. Although the access time can be large, one can pipeline several accesses to SRAM, thereby keeping the cycle time small as was done in the asynchronous SRAM of [3]. We try to have one complete access within one cycle.

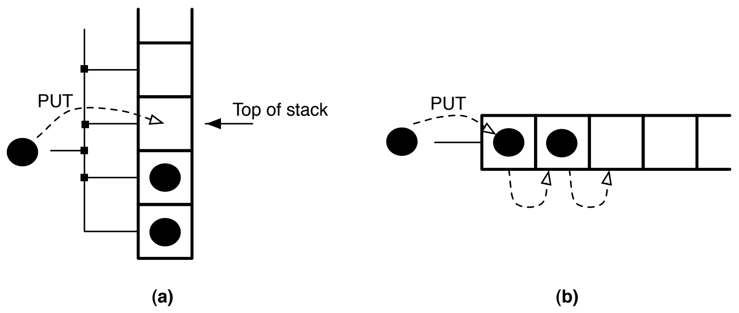


Figure 1: Two simple stacks: (a) a pointer stack (b) a linear stack

An alternative implementation is a linear array of cells as in Figure 1(b), where the first cell of the array always contains the data item on top of the stack. We can operate this linear array of cells as a shift register, where for each put and get action all data items move simultaneously. This approach obviously has the disadvantage of a large fanout for the control, which may lead to a large cycle time and energy consumption. We can obtain a shorter cycle time if cells communicate only with their neighbors. Each put and get action ripples down the linear array, where multiple ripples can occur concurrently. For a put action, each cell receives a new data item and then pushes the previous data item into the substack. For each get action, each cell pops the current data item and then pulls a new data item from the substack. A potential disadvantage of such an implementation is that the total energy consumption per put or get action is high, because each put or get action causes all items in the array to move. In other words, the energy consumption per put or get action is proportional to the number of items in the stack. Furthermore, a simple implementation may fail to prevent overflow or underflow of the stack: a put action on a full stack loses the data item at the bottom of the stack, and a get on an empty stack will yield an unknown value. Finally, for a put action, each cell must temporarily store a second data item, thereby wasting storage space and area. We show that there are stack implementations with a short cycle time independent of the number of data items in the stack and data width, low energy consumption, and modest area.

Related work on asynchronous stacks includes Alain Martin's lazy stack [8] and stack designs by Mark Josephs et al. [6, 10]. These designs, however, have a longer cycle time and the cycle time depends on the size of the stack.

2 A Linear Three-Place Stack

The stack implementation shown in Figure 2 is a combination of the pointer stack and the linear stack. The implementation consists of a linear array of cells, where each cell has three storage locations and each storage location can hold a separate data item.

Two ideas form the basis for this stack implementation. First, puts and gets on the stack and substack rotate through the storage locations of the cell in a round-robin fashion, like the stack pointers in a circular pointer implementation. Furthermore, the cell propagates actions to the substack only when necessary. More precisely, only when the cell becomes full, the cell performs a put action on the substack, and only when the cell becomes empty, the cell performs a get on the substack.

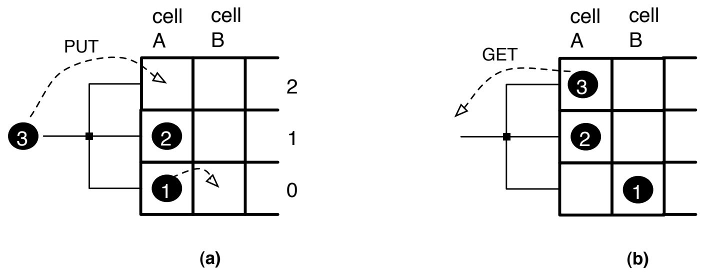


Figure 2: A linear three-place stack: (a) A put of item 3 results in a move of item 1 into the substack consisting of cell B and following cells (b) A get of item 3 has no effect on the substack

Second, in order to protect against overflow and underflow, we introduce *unsuccessful* put and get actions. An unsuccessful put action notifies the environment that a put action failed, since the stack is full. An unsuccessful get action notifies the environment that a get action failed since the stack is empty. As a consequence of an unsuccessful put or get action, no data items move, are lost, or are created. To implement the overflow and underflow protection, each cell records whether the substack becomes full or empty respectively. The linear three-place stack has the property that it becomes full only after its substacks have become full and it becomes empty only after its substacks have become empty.

Figure 2(a) illustrates the data movements when the environment puts item 3 on the stack. First cell A accepts item 3 in location 2 and then cell A puts item 1 on the substack. Figure 2(b) illustrates a get action on the stack. Cell A immediately acknowledges with popping item 3 from the stack and no further actions are necessary on the substack.

Besides, overflow or underflow protection, the linear three-place stack has a few more attractive properties. First, notice that a put action propagates down the stack as long as each cell has 2 items. As soon as a cell has anything less than 2 items, the cell stops the propagation of a put action. Similarly, a get action propagates down the stack as long as each cell has 1 item. The get action stops propagating at the first cell with more or less than 1 item. Thus, put and get actions only cause a fraction of all data items to move resulting in better energy consumption per put or get action. Second, each cell contains three storage locations, each of which may store a separate data item. Thus no storage location is wasted. Third, data moves are only between neighboring storage locations, which simplifies the implementation by avoiding fanout from and fanin to storage locations.

3 A Tree Stack

A different type of stack, offering different advantages, is the tree stack. Every node in a tree stack consists of an n-place cell, like the nodes in a linear stack. The number of places in a cell is at least 2. Every cell has n substacks, one for each place. Three nodes from a tree stack with $n = 2$ appear in Figure 3.

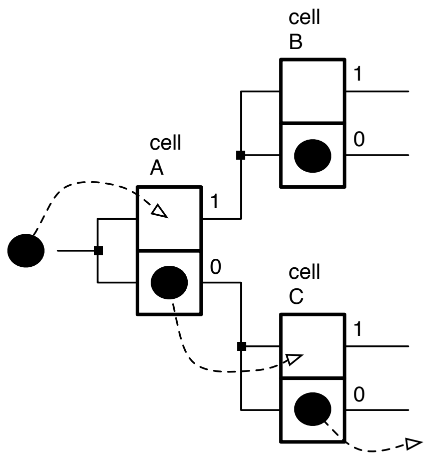


Figure 3: A two-place tree stack: A put action on the root of the tree propagates down a path in the tree

The behavior of each tree cell with respect to puts and gets is similar to that of the linear stack cell. When using a two-place cell, puts and gets rotate through the places 0 and 1. After the cell puts a data item in place i, the cell puts the data item residing in place $i+1$ (modulo 2), if any, into the substack of place $i+1$, as illustrated in Figure 3. Similarly, after the cell gets a data item from place i and place $i-1$ (modulo 2) is empty, the cell gets a data item for place $i-1$ from the substack of place $i-1$.

Like in the linear stack, we can implement overflow and underflow protection for the tree stack by using unsuccessful put or get actions. Thus, a tree stack becomes full or empty only after each substack becomes full or empty respectively. Furthermore, by making sure that each place in a cell can contain a data item, the tree stack also wastes no storage locations. Finally, the tree stack has one more attractive property. Note that for each put or get action, only the data items in one path of the tree are pushed down into or pulled up from the tree stack. Consequently, the total number of moves per put or get action for the tree stack is logarithmic in the number of items in the stack, rather than linear.

4 GasP Implementation

In our design method we specify each cell in a stack design with a finite state machine, where the complete stack is the parallel composition of these finite state machines. The finite state machines and their composition have implementations consisting of GasP circuits. We briefly summarize the main points of the implementation method by means of an example. For more information, see [4].

The implementation consists of a data path and a control path as shown in Figure 4. The data path is a network of pulsed latches, and the control path is a network of GasP modules. The network of GasP modules implements the correct sequences of events, where an event is a data move, and generates the pulses for the pulsed latches. The pulses for the latches implement the data moves.

We propose a simple translation of a finite state machine into a GasP implementation: map every event to a GasP module and map every state to a connection between GasP modules. We call such a connection a state conductor. The implementation of a composition of finite state machines is simply the superposition of all state-machine implementations, where GasP modules with the same label overlap. For small finite state machines, this translation is simple and efficient. For finite state machines with many states, this translation may become inefficient, because you need one connection per state. Fortunately, the specifications for our stack consist of small finite state machines.

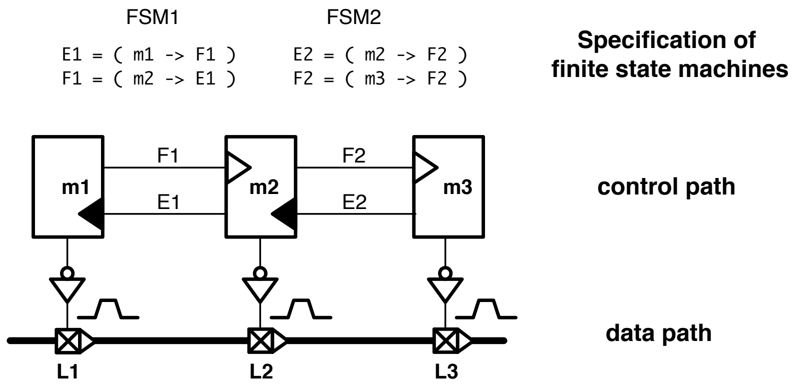


Figure 4: Two finite state machines implemented by a connection between GasP modules. Each GasP module implements a data move.

Figure 4 shows how the implementation of two finite state machines, together implementing a two-stage FIFO. FSM1 specifies the repetition of events `m1` followed by `m2`, FSM2 specifies the repetition of events `m2` followed by `m3`. FSM1 alternates between states `E1` and `F1`, for “Empty1” and “Full1” respectively, and FSM2 alternates between states `E2` and `F2`. The GasP modules for events `m1`, `m2`, and `m3` dictate the data movements in the data path. Latches L1, L2, and L3 are pulsed latches, normally opaque. When GasP module m1 "fires," it produces a pulse for latch L1, upon which data moves through latch L1 to latch L2.

Each GasP module has some input ports and some output ports. For example, F1 and E2 are input ports for module m2 and E1 and F2 are output ports. When all inputs of a GasP module are "set," the GasP module will "fire." When a GasP module fires, three things happen: The GasP module resets each input, sets each output, and generates a pulse for a pulsed latch. Setting and resetting a port involves driving the state conductor connected to the port HI or LO for a brief fixed period. Because state conductors are not actively driven HI or LO all the time, they need to have a keeper to keep their state when the wire is not driven. To avoid clutter, state conductors in schematics appear as lines between GasP modules, and keepers are omitted. In order for these implementations to work properly, all transistors must be carefully sized. Sizing the transistors can be done with the Theory of Logical Effort [5].

In a composition of finite state machines many events can occur concurrently. An event common to two machines, however, can occur if and only if both machines permit the event. In the superposition of machine implementations, the GasP module for the common event has at least two input ports, one for each finite state machine. The GasP module fires only when both inputs of an event are set, i.e., when both finite state machines permit the event. For example, GasP module m2 can fire if and only if FSM1 is in state F1 and FSM2 is in state E2. This firing rule is analogous to the firing rule for transitions in Petri Nets [11].

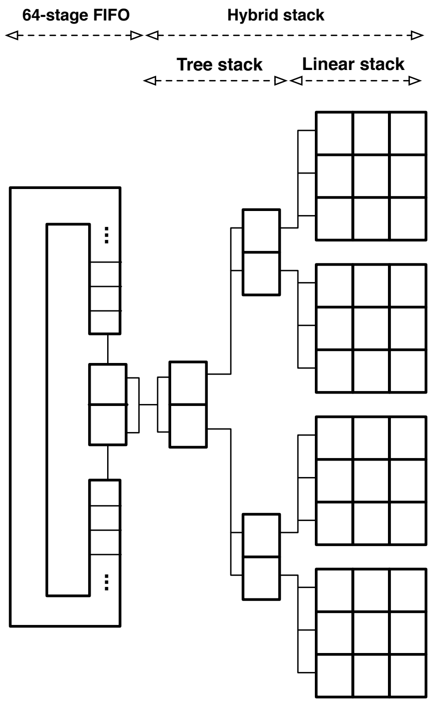
5 A Hybrid Stack Implementation

In order to benefit from the advantages offered by the linear three-place stack and the tree stack, we have constructed a hybrid stack as shown in Figure 5(a). The complete stack contains 2+2+2+ 4*3*3 =42 storage locations. Although this stack is small in terms of storage locations, this implementation is large enough to demonstrate the main characteristics with respect to cycle time and energy efficiency of each move in the tree stack and linear stack.

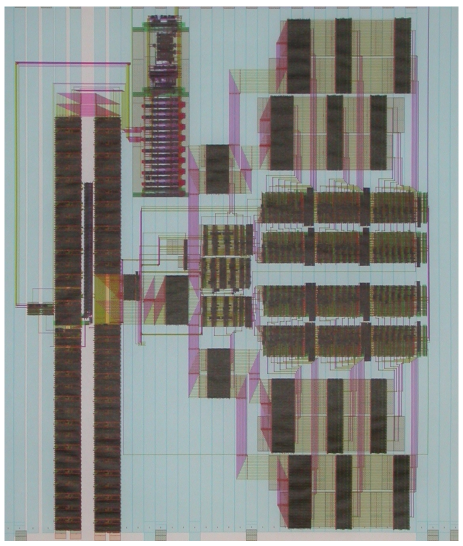
We implemented the hybrid stack in the TSMC 180nm process. In order to be able to test the functionality and the speed of the stack circuit, we built a 64-stage FIFO ring of which two stages communicate with the hybrid stack. Two other adjacent stages in the ring serve as loading and unloading stages. For each experiment we preload the ring with a specific sequence of puts and gets, run the ring with the stack to do the measurements, and then stop the ring and check the contents of the ring and the stack. The FIFO ring can be loaded with any sequence of puts and gets, where every put has a data value. For our implementation we chose word widths of 18 bits, because we could reuse data latches of a data path of 18 bits that was used for a different chip. The actual chip layout appears in Figure 5(b).

The two FIFO stages that interface with the stack behave such that any put (P) with a data value will execute a put action of that data value on the stack, and a get (G) will execute a get action on the stack. Furthermore, the two interface stages turn each put action into a get action and each get action into a put action. This allows us to run the FIFO ring continuously and to monitor the throughput and energy consumption of the FIFO ring and the stack. We must load the ring with a proper sequence of puts and gets. For example, a sequence of $(P^k G^k)^n P^k$ after the first revolution will turn into the sequence $(G^k P^k)^n G^k$ and a stack loaded with k values, independent of the value of n. After the second revolution we are back in the initial state with the initial sequence of puts and gets and an empty stack.

When we tested our stack chip with different values for k and n, we found that the FIFO and stack were fully functional at voltages of 1.6V and up. Although other tests indicate that the FIFO and stack are also functional at voltages below 1.6V, we could not verify any data values, because the scan chain malfunctioned for voltages below 1.5V. The scan chain does not contain any GasP circuits.



(a) Test setup for the stack chip

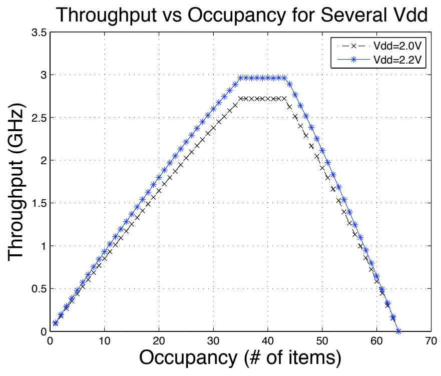


(b) A plot of the layout of the stack chip

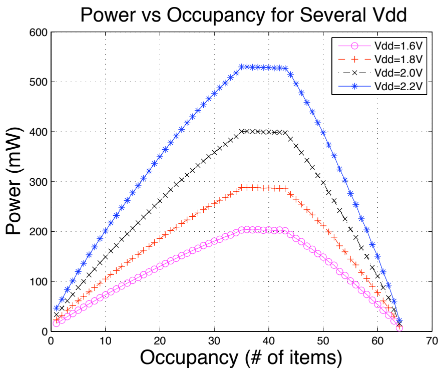
Figure 5: Test setup of hybrid stack and a picture of the chip

6 Throughput and Power Consumption of the Stack

To verify the speed and power consumption of the stack, we created plots for throughput versus occupancy and power consumption versus occupancy, where occupancy is the number of items in the FIFO. The items in the FIFO are puts and gets, in an alternating sequence. A plot of the throughput versus occupancy appears in Figure 6(a). A plot of the power consumption as a function of the occupancy appears in Figure 6(b). There are no throughput plots for 1.8V and below, because the counters to measure throughput malfunctioned at voltages of the nominal 1.8V and below. The counters were reused from a different chip design and do not contain any GasP circuits. For the throughput and power consumption measurements, all data values are the same. Consequently, the control path consumes all dynamic energy and the data path consumes no dynamic energy, since no data values are changing.



(a) Throughput as a function of occupancy



(b) Power consumption as a function of occupancy

Figure 6: Measuring throughput and power consumption

The throughput plots show a plateau at the maximum throughput of the stack, because we designed the ring such that the hybrid stack forms the bottleneck of the ring. By extrapolating the slopes of the throughput plots we can derive what the maximum throughput is of the FIFO rings. The plots for 2.2V and 2V show plateaus at 2.96GHz and 2.72GHz respectively. These frequencies correspond to cycle times of 338ps and 368ps. The FIFO frequencies for these voltages are approximately at 3.4GHz and 3.1 GHz respectively, which correspond to cycle times of 294ps and 322ps.

The plot for power consumption shows that the stack operates at 1.8V and 1.6V. Although we don't have good numbers on the maximum throughput for the stack and FIFO for 1.8V, the data for 2.2V and 2V seem to suggest that the stack operates at a maximum throughput of just under 2.5GHz, i.e. a cycle time of just below 400ps, and the FIFO operates at a maximum frequency of around 2.75GHz, or a cycle time of around 363ps. The predicted cycle times for the FIFO and stack are 370ps and 454ps respectively at 1.8V, which indicates that our FIFO operates as predicted but our stack operates a bit faster than predicted.

7 Static and Dynamic Energy Consumption

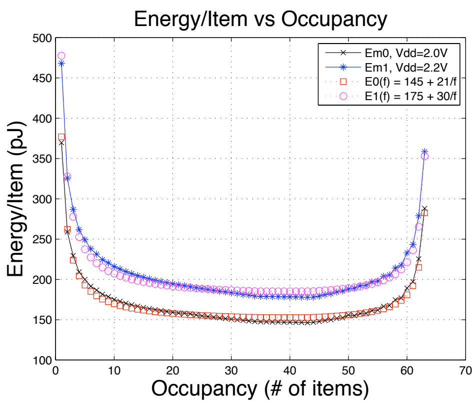
When we divide the power consumption by the throughput, we get the energy per item for traveling through the 64-stage FIFO. Figure 7(a) shows the energy per item in the ring as a function of occupancy. The plot for 2V shows that at maximum throughput each item spends 150pJ to travel through the 64-stage ring.

The plot also shows that the energy per item increases as the occupancy moves away from the occupancy for maximum throughput. This increase in energy is due to the increase in static energy per item. As the cycle time of each stage increases, more time of the cycle time is spent idle wasting static energy, thus increasing the total energy per item per cycle. This observation led us to try to measure separately the static and dynamic energy of the FIFO from the data in Figure 7(a).

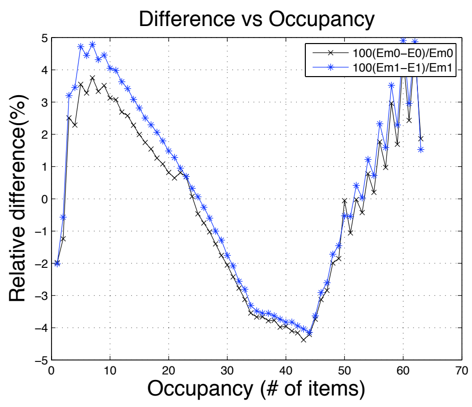
If we assume that the static energy per item is proportional to the cycle time and cycle time is inversely proportional to throughput, then static energy per item can be expressed as $E_{static}(n) = C1/f(n)$ where $C1$ is a constant and $f(n)$ is the throughput for occupancy n. The dynamic energy per item per cycle is constant $C0$, because the same gates fire each time the item passes a stage. Consequently, the total energy per item is the sum of the dynamic and static energy and can be expressed as

$$E_{tot} = C0 + C1/f(n)$$

In Figure 7(a) we plotted the functions $E0(f) = 145 + 21/f(n)$ and $E1(f) = 175 + 30/f(n)$. These functions are in good agreement with our measured results for energy per item for 2V and 2.2V as indicated by $Em0$ and $Em1$ respectively. These observations lead us to conclude that the dynamic energy per item spent on moving an item through the 64-stage FIFO ring is approximately 145pJ for 2V and 175pJ for 2.2V. The static power consumption for the complete circuit is approximately 21pJ per nanosecond, or 21mW, for 2V and 30pJ per nanosecond, or 30mW, for 2.2V. Note that the static power consumption includes not only the static power spent in the ring, but also in the stack and the peripheral circuitry for testing and (un)loading. Recall that the dynamic energy consumption represents the energy spent in the control circuit only and includes the charging and discharging of the 18 latches. The dynamic energy per move in the FIFO is 145/64=2.27pJ for 2V and 175/64=2.73pJ for 2.2V.



(a) Energy per item versus occupancy



(b) Difference between measured and predicted total energy

Figure 7: Measuring average dynamic energy per item in FIFO ring

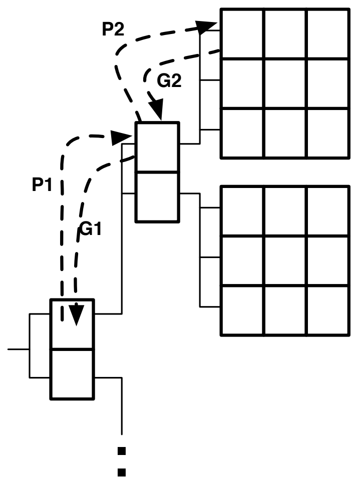
We were surprised that the measured and fitted graphs for energy per item matched so well. The difference is within 5% for all measurements, as can be seen from Figure 7(b). This result suggests that measuring energy with this technique provides a very good way to measure separately the average dynamic energy per item and average static power of asynchronous FIFOs. You only need to measure throughput and power versus occupancy, and by some simple computations and graph fitting you obtain the average dynamic energy per item and static power.

8 Energy per Move in the Stack

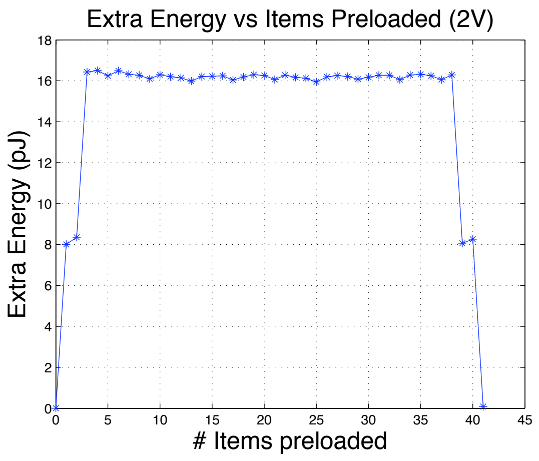
We measured the dynamic energy spent for the various moves in the tree stack and in the linear three-place stack. To find the dynamic energy per move in the tree stack, we used a sequence of 37 puts and 36 gets, i.e., $(PG)^{36}P$, such that the ring operates at maximum throughput. Before we start this sequence, we preloaded the stack with k items, where $k = 0, 1, .., 42$. As a result of these action sequences and preloaded values, data moves only in the tree stack. No moves occur in the linear three-place stacks. Consequently, the extra energy spent in addition to the energy spent in the FIFO ring is the dynamic energy per put or get spent on data moves in the tree stack. Figure 8(a) illustrates the moves in the tree stack. When only 1 item is preloaded in the stack, each put causes a P1 move in the tree stack and each get causes a G1 move in the stack. When only 2 items are preloaded in the stack, each put also causes a P1 move in the tree stack, but now into the other (empty) branch of the tree stack. Similarly, each get causes a G1 move in the stack. When k items are preloaded in the stack, with $3 \leq k \leq 38$, each put causes a P1+P2 move in the tree stack and each get causes a G1+G2 move in the stack. When the stack is almost full at 39 and 40 items preloaded, there are only 3 and 2 places empty respectively in the tree stack. Each put then causes only a P1 move in the tree stack and each get causes only a G1 move in the stack. For each number of preloaded items k we measured the energy per item $E(k)$ and subtracted the energy of the FIFO, $E(0)$, when no items are preloaded. Figure 8(b) shows the extra energy spent on moves in the tree stack obtained from measured data. Figure 8(b) shows that the dynamic energy per move (P1, P2, G1, or G2) in the tree stack is fairly constant at about 8pJ when operating at 2V.

Finally we measured the energy spent per move in the linear three-place stack. For this purpose we fill the stack with n items and then empty the stack, for various values of n. We first calculated the average number of moves in the hybrid stack per put or get action for each value of n. As can be seen from Figure 9(a), the average number of moves rises very slowly as a function of n. The function slowly decreases after $n = 35$, because then the stack fills up the last free spaces nearer to the root of the stack. Note that the increase in average number of moves from $n = 13$ to $n = 30$ is about 1. This increase of 1 is entirely due to moves in the linear stack.

To see if we can observe the behavior of Figure 9(a) from an experiment we used the the sequence of actions $(P^nG^n)^kP^n$ for some value of k and an empty stack initially. We operated the FIFO ring as close as possible to maximum throughput, because then the energy per put or get spent in the FIFO ring is fairly constant and we can subtract this energy from the total energy cost. If we operate outside the range of maximum throughput the influence of static energy becomes larger and the results may become inaccurate. Figure 9(b) shows the result of our experiment. Although we failed to get a good match, the trends in both figures are similar. Part of the reason that the match is not that good this time may be that the occupancy of the FIFO ring varied significantly for different values of n and that the average energy values are the result of differences between values about 10 times as large. Despite the inaccuracies in this graph, we may conclude that the difference in energy between $n = 13$ and $n = 30$ is about 6pJ at 2V which is the average dynamic energy consumed per move in the linear three-place stack.



(a) Types of move in the tree stack: P1, P2, G1, G2



(b) Dynamic energy spent in tree stack at 2V

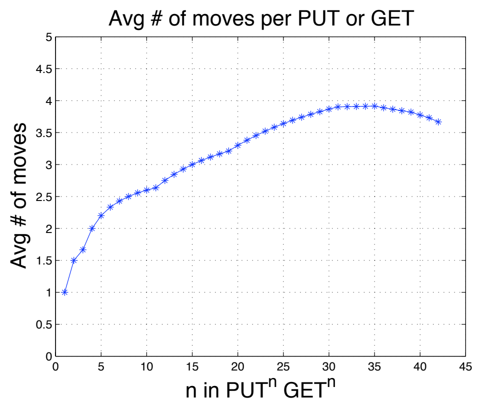
Figure 8: Energy per move in tree stack

9 Concluding Remarks

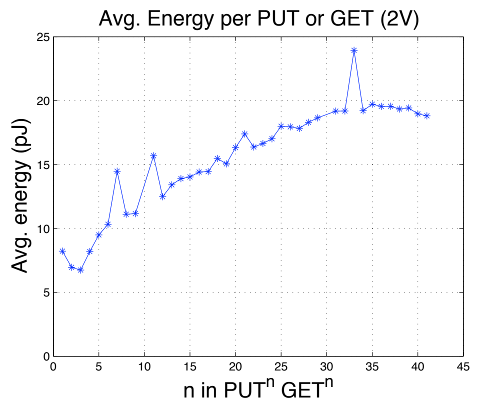
The hybrid stack has a few attractive properties. First, the cycle time of this implementation is only about 6 FO4 delays. In other words, the implementation completes each put and get action within 6 FO4 delays. We believe that the cycle time is independent of the number of storage locations. Just as you can add stages to an asynchronous ripple FIFO implementation without increasing its cycle time, so can you add cells to the linear stack implementations without increasing its cycle time. Second, the average number of moves in the hybrid stack per action at its interface increases very slowly with the number of items in the stack. This slow increase is due to the tree topology and the potential annihilations of puts and gets in the linear three-place stack. Chip measurements show that the average dynamic energy consumption of a single move in the tree stack is about 8pJ and the average dynamic energy consumption of a single move in the 3-place linear stack is about 6pJ. These numbers should be compared with the average dynamic energy of 2.26pJ for a single move in a FIFO.

There is another attractive property of the hybrid stack that we discovered after tape-out. The cycle time for all cells in the linear stack after the first cell can be quadrupled without affecting the cycle time at the interface of the hybrid stack. We discovered this property after a careful study of all sequences of actions at the interface of the hybrid stack that result in two successive actions for the second cell in a linear stack. This property can be used to reduce the energy consumption of the stack even further, for example by reducing transistor sizes or lowering the voltage supply for certain cells. This last property only applies to the hybrid stack. The property fails to hold for any linear stack or for any tree stack.

How would our asynchronous stack implementation compare to a synchronous stack implementation? Unfortunately, the only synchronous stack implementation we found is in [9]. This stack implementation consists of a simple shift register of eight 8-bit words. A simulation operates at 50MHz in 2μ CMOS.



(a) Average number of moves as a function n in P^nG^n



(b) Average energy as a function of n in P^nG^n

Figure 9: Finding the average dynamic energy of a move in the linear stack

Obviously, linear shift registers suffer from a high energy consumption per put or get, since at least all occupied registers will be activated with each clock tick. No energy numbers are given in [9].

In absence of any other synchronous stack implementation, the closest comparison we can make is with a synchronous single-ported SRAM implementation, where the SRAM would have to be expanded with some pointer arithmetic for implementing the LIFO function. In [2] we found the lowest energy consumption per access of an SRAM in a technology comparable to ours. The authors in [2] report on a fabricated low-power SRAM of 4K words of 16 bits each in 180nm CMOS with an energy per access of 9.5pJ at 1.6V, which converts to about 14.8pJ at 2V. The access time, which is equal to the cycle time of this SRAM, is 4ns at 1.6V. Slightly higher numbers are reported in [7]. Obviously, this energy consumption and cycle time do not yet include any energy or delay spent on LIFO pointer arithmetic. The cycle time of our stack implementation is 368ps, at least a factor 10 better. The energy per put or get depends on the contents of the stack and can be expressed as $8t + 6l + D$ pJ, where $t = 1, 2$ represents the number of tree moves, l represents the number of moves in the linear stack, and D represents the energy spent in the data path. For a stack with 4K locations, the value of l ranges between 0 and $4000/(3 * 4) \approx 333$, where the average value of l is probably much smaller than 333. Although this comparison is fraught with problems, it is clear that our stack implementation has a much smaller cycle time but a higher energy consumption per put or get when compared with the low-power SRAM implementations in [2, 7].

10 Acknowledgements

We thank Bill Coates for his careful comments on an earlier draft of the paper. Charles Molnar and Huub Schols contributed to earlier designs of a stack.

References

[1] B. Amrutur and M. Horowitz. Speed and power scaling of sram's. *Solid-State Circuits, IEEE Journal of*, 35(2):175 –185, Feb. 2000.

[2] S. Cosemans, W. Dehaene, and F. Catthoor. A low-power embedded SRAM for wireless applications. *Solid-State Circuits, IEEE Journal of*, 42(7):1607 –1617, July 2007.

[3] J. Dama and A. Lines. Ghz asynchronous SRAM in 65nm. In *Asynchronous Circuits and Systems, 2009. ASYNC '09. 15th IEEE Symposium on*, pages 85 –94, May 2009.

[4] J. Ebergen, D. Finchelstein, R. Kao, J. Lexau, and D. Hopkins. A fast and energy-efficient stack. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 7–16. IEEE Computer Society Press, Apr. 2004.

[5] J. Ebergen, J. Gainsley, and P. Cunningham. Transistor sizing: How to control the speed and energy consumption of a circuit. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 51–61. IEEE Computer Society Press, Apr. 2004.

[6] M. B. Josephs and J. T. Udding. Implementing a stack as a delay-insensitive circuit. In S. Furber and M. Edwards, editors, *Asynchronous Design Methodologies*, volume A-28 of *IFIP Transactions*, pages 123–135. Elsevier Science Publishers, 1993.

[7] K. Mai, T. Mori, B. Amrutur, R. Ho, B. Wilburn, M. Horowitz, I. Fukushi, T. Izawa, and S. Mitarai. Low-power sram design using half-swing pulse-mode techniques. *Solid-State Circuits, IEEE Journal of*, 33(11):1659 –1671, Nov. 1998.

[8] A. J. Martin. A synthesis method for self-timed VLSI circuits. In *Proc. International Conf. Computer Design (ICCD)*, pages 224–229, Rye Brook, NY, 1987. IEEE Computer Society Press.

[9] M. McDonnell and K. Winters. A dynamically allocated CMOS dual-LIFO register stack. *Solid-State Circuits, IEEE Journal of*, 25(5):1287 –1290, Oct. 1990.

[10] F. Pessolano and M. B. Josephs. A low-power, high-speed stack controller designed using asynchronous circuit techniques. In A.-M. Trullemans-Anckaert and J. Sparsø, editors, *Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 387–396, Oct. 1998.

[11] J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall, 1981.

[12] I. Sutherland and S. Fairbanks. GasP: A minimal FIFO control. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 46–53. IEEE Computer Society Press, Mar. 2001.

Author Information

Jo Ebergen has a Ph.D. in Mathematics and Computing Science from Eindhoven University of Technology (1987). He taught at the University of Waterloo from 1988 until 1996. Since 1996, he works at the VLSI Research Group at Sun Labs, which has now become Oracle Labs. Jo is a member of IEEE and a distinguished engineer of the ACM. His research interests include asynchronous circuit design, VLSI design, and performance analysis of digital systems.

Daniel Finchelstein received the Ph.D. and M.S. degrees in 2009 and 2005, respectively, from MIT, and the B.A.Sc. degree in 2003 from the University of Waterloo. He is currently working in the 3-D graphics performance group at Nvidia Corporation. His research interests include energy-efficient and high-performance digital circuits and systems.

Russel Kao received the Ph.D. Degree in Electrical Engineering from Stanford University in 1992. He has worked at the research labs of Hewlett Packard, Digital Equipment, and Sun Microsystems. His interests include computer architecture, computer design, software performance tools, and CAD. His CAD interests include circuit simulation, netlist comparison, ASIC physical design, routing, and layout generation.

Jon Lexau received his M.S. degree in Electrical Engineering from Stanford in 1994. He works at the VLSI Research Group in Sun Labs (now Oracle Labs) since 1993. Jon has been an IEEE member since 2006. His research interests include circuits for optical interconnects, analog circuits, and computer architecture.

David Hopkins received his M.S. degree in electrical engineering from Stanford University in 2008 and his Master and Bachelor of Engineering degrees from Harvey Mudd College in 2002 and 2001, respectively. He joined the VLSI Research Group at Sun Labs (now Oracle Labs) in 2002. His interests include high-performance and energy-efficient computing and communication.