ASYNCHRONOUS RESEARCH CENTER
Portland State University

Subject: **Keynote for the Vail Computer Elements Workshop**
Date: **25 June 2012**
From: **Ivan Sutherland**
ARC#: **2012-is15**

The Tyranny of the Clock

INTRODUCTION – STATIC ELECTRICITY

I once experimented with a big CMOS power transistor. It was rated for about 5 amps and about 20 volts. To try it out I used a car battery for power and a light bulb from an automobile brake light as a load. The big CMOS transistor served as a switch to turn the light bulb on or off. I left the gate terminal of the transistor unconnected.

To turn the light on, I touched the positive terminal of the battery with one hand and the gate terminal of the transistor with my other hand. My body conducted positive charge to the gate, turning on the transistor and lighting the light. To my surprise, when I let go the light stayed on. The light would stay on for 15 seconds or so, but gradually drifted towards OFF as the static charge on the transistor gate gradually drained away. Touching the negative battery terminal and the gate turned the light out.

This experiment shows vividly that a CMOS transistor is a voltage controlled current source. The CHARGE on the gate of a transistor controls the flow of current between its source and drain. Moreover, unless removed, the charge will remain in place for a relatively long time.

Of course, this is the idea behind today's common DRAM memories. We store tiny charges on tiny wires where they remain for a few milliseconds. By refreshing them often enough we can store enormous numbers of bits in a very small space.

But logic designers tend to forget this important property of CMOS circuits. If you put a charge on a wire it will stay there, at least for a time that is very long compared to logic speeds. Moreover that charge will control the behavior of any transistor whose gate is attached to the wire. Instead, designers tend to think that only a flip-flop can store a bit of data. Only the electrical designers of DRAMs and CCDs regularly make good use of static electric charges. So remember: *EVERY WIRE IS A POTENTIAL STORAGE ELEMENT.*

WIRES CONSUME POWER

Every wire is a storage element. What does it store? It stores electric charge. One may think that a logic circuit sets the voltage of its output terminal. NOT SO, it merely delivers current to the output terminal to either add or remove charge from that output wire. To add charge, the logic circuit draws current, and therefore energy, from the power supply just as my body took charge from the car battery. Some of the energy from the power supply remains stored on the output wire in the form of charge; some is lost in the transistors of the logic gate, just as the resistance of my arms consumed a tiny amount of energy. To remove charge, the logic circuit drains the charge from the wire, thus consuming all the energy stored on the wire and dissipating it as heat.

Obviously it takes more energy to charge the greater capacitance of a longer wire than the smaller capacitance of a short one. In fact, the length of wire charged and discharged each second accounts for almost all of the energy the chip uses, and all that energy ends up as heat.

WIRES ARE SLOW

Designers, quite correctly, think of the charge on a “short” wire as spreading instantly throughout the wire. Even so, it takes time for the logic circuits to charge or discharge the wire and whatever may be connected to that wire. In well-designed circuits, the wires usually account for at least half of the delay. Novice designers often forget that their circuits must also put charge on the wire; the result is slow circuits.

But electricity moves through “long” integrated circuit wires relatively slowly. Integrated circuit wires have both resistance and capacitance all along their length. I think of the charge put into a wire as spreading through the wire much as heat spreads through a metal bar; indeed, the equations are the same. Imagine yourself holding one end of a silver spoon while I heat the other end with a candle. Eventually you’ll get uncomfortable as your end heats up. The delay in a “long” integrated circuit wire grows as the square of the length of the wire rather than linearly. Boosters or repeaters at intervals along the wire can hasten delivery.

WIRES COST MONEY

Wires cost money because they account for more than half of the processing steps in a “logic” chip. Such a chip may have about a dozen layers of wire. The logic transistors hide underneath the wires. It’s the wires that set the size of the chip. The very regular wiring patterns of DRAM chips permit cost savings by using only two or three wiring layers.

Wires cost design time because they must fit. After designing the logic, an abstract mathematical task, one then faces the geometric task of where to place the logic and how to route the wires. That task, called “place and route,” comes near the end of the chip design. It’s never pleasant, because it faces a little-recognized geometric growth law: as the number of things to connect increases, not only does the number of wires increase, but also the average length of each wire increases. This is the same truth that creates traffic congestion in big cities but not in country towns.

But it's even worse for the integrated circuit designer, because he must place his wires in such a way that each and every signal arrives "on time" to meet the tyranny of the clock. Because the delay in each wire increases with the square of its length, the designer must put "repeaters" in longer wires, and that adds area, making the chip larger. My notion is that the layout of a modern chip is an impossible task; a few exceptions to that impossibility do actually reach production.

Of course, these difficulties delay the schedule, and that costs money too.

REVOLUTION – AKA PARADIGM SHIFT

Like all tyranny, the tyranny of the clock stems from the range over which ***we choose*** to subject ***ourselves*** to the tyrant's authority. "Revolution" is the word used for the behavior of a people fed up with political tyranny. Recall "Taxation without representation," and "Liberté, Egalité, Fraternité."

"Paradigm Shift" is a way to escape from technical tyranny. The Defense Advanced Research Projects Agency (DARPA) is fond of seeking paradigm shifts, believing that their investment in high-risk, high-reward projects may lead to new ways of thinking about and solving problems. I thought you might like to see a "Paradigm Shift" right before your eyes. – Roll the film.

I do love a good pun. But seriously, what is the real problem? It's the tyranny of the clock. We impose that tyranny on ourselves by choosing to treat geometrically separated events as simultaneous. The "simultaneous" fiction helps designers think about complex concurrent tasks by dividing time into periods. It's the kind of help that students get from 55-minute class periods for scheduling classes. Is the 55-minute hour the right time increment for all teaching? The best class I ever took had a six-hour laboratory every Monday with lunch optional. Class periods are common because they make scheduling possible, not because they improve learning.

I promote the idea that we let things happen at their own natural pace. I'll start by describing one system I'm working on: *MERGE SORT***.** I want to sort long lists of numbers as fast as I can deliver them to the sorter.

I have found, empirically, that my logic takes about twice as long to decide which of two numbers is the larger than it takes to stream a data element to the sorter. In 90 nm CMOS, for example, I can make an on-chip network that will move data at 4 GHz, or about 250 psec per data element, but comparison takes about 400 psec. Were I to use a clocked circuit, 250 psec would be a very fast clock indeed. Typical 90 nm chips run at 1 GHz or slower.

Multiple comparator circuits are an obvious answer. But each comparison in a merge sort depends on the result of the previous comparison, so how can we do comparisons in parallel?

My plan is to do some comparisons that *might* be useful "on speculation." Instead of two comparison circuits, I suggest using four. The four circuits will compare the first two numbers in each list with the first two in the other list. Each comparison can start as soon as suitable data arrive. Only one of those comparisons produces an immediately useful answer, namely the one comparing the first number from each list. The other comparisons may or may not prove useful.

Choosing the first output advances one of the input lists. This renders useless the comparison of that element to the second member of the other list. Yes, we wasted that comparison, but we saved time because the next useful comparison got started before the choice was made. The next useful comparison is already half finished.

Of course, it takes N*logN comparisons to sort N numbers. I propose to put logN of my comparison units in series so that I can sort N numbers in linear time. I'll also need some local storage between stages to store the pairs of sorted lists that are inputs to each stage. The first comparator merges pairs of lists-of-one, producing sorted lists of two numbers each. The next stage merges pairs of lists-of-two, producing sorted lists of four numbers each, and so on. The intermediate storage holds the first output list and then feeds it on for comparison with the second output list. The final comparator merges a pair of lists-of-N/2 elements to produce the final sorted list of N elements. FIFO memories can form the intermediate storage.

The total storage required is, of course, N values, because all N values must have entered the sorting system before the greatest of them can emerge. The latency of the system is (N) + (logN) because the final element can enter only after the preceding N-1 have entered and the final element must then pass through logN stages. A system to sort 1000 numbers, for example, takes only slightly longer than the time it takes to put the 1000 numbers into the sorter.

Let us consider a fact about this sorter that I didn't at first notice. Suppose the comparison time is T: how much speedup do we get by using four sorting circuits in parallel? It's not a factor of four, because half of the steps are wasted. My first guess was that the time per element would be T/2. Surprisingly, the speedup turns out to be a bit more than a factor of 2. The reason is that we can abort the wasted comparisons before they complete, and so we waste less than half of the available comparison time.

SOME LESSONS IN ASYNCHRONY

Let us talk for a minute about self-timed First In First Out (FIFO) memories. Software people and many hardware designers think a FIFO has three parts: a block of addressable memory, a write pointer, and a read pointer. Here are some things such a FIFO must do: 1) Its read and write actions compete for memory cycles. 2) It must resolve simultaneous read and write requests. 3) Binary pointer values count up or down. 4) A decoder converts binary pointer values into one-out-of-N signals to address the memory. 5) Data values flow from input to memory cells. And 6) data values flow from memory cells to output. Have I got all that right? It's hard to design such a device and be sure it works correctly.

Now let's consider a self-timed FIFO. It has a series of storage locations through which data elements flow in sequence. There are no pointers. There is no decoder. There is no input vs. output contention: input and output are entirely concurrent.

The algorithm for such a FIFO is trivial. It's essentially the behavior I would have in a bucket brigade. I can act only if I have no bucket and my predecessor proffers me his bucket. When I act I do three things, 1) I capture the bucket of water, 2) I empty my predecessor's hands, and 3) I proffer the bucket to my successor. This is the basic pipeline equation. The algorithm requires only one single AND function to detect when to act and enough amplification to do the necessary actions.

My predecessor must be FULL and I must be EMPTY for me to accept fresh data. It's incredibly simple. Maybe we'll do a KLA at this point.

The important thing is to know the difference between FULL and EMPTY. The circuits we promote at the ARC use a single wire to "remember" this essential piece of state. Not surprisingly we call it a "state wire." A circuit we call GasP acts when it detects the FULL and EMPTY state of its two adjacent state wires. When it acts it does exactly the three things I mentioned before. 1) It causes data latches to capture the data, 2) it drives the predecessor state wire, which was FULL, to the EMPTY state, and 3) it drives the successor state wire, which was EMPTY, to FULL. Notice that this FIFO requires N+1 wires to carry N bits of data plus their FULL or EMPTY state.

Another way of looking at a GasP circuit is that it's part of a ring oscillator. Everybody knows that three or five, or any larger odd number, of CMOS inverters in a ring will oscillate. The AND logic in the GasP circuit couples adjacent rings to coordinate their actions. The state wires remember the state of a partial oscillation for later completion. All self-timed circuits use the interrupted actions of coupled ring oscillators.

GasP circuits take the unusual step of driving each state wire from both ends. Remember that the wire itself is the storage. I like the "HI is FULL" convention. With this convention, each GasP circuit drives its output state wire HI and its input state wire LO. Each state wire is driven HI at its input end and LO at its output end. We take care to avoid ever driving both directions at once. That's made easy by the simple fact that an AND function is always quicker turning off than it is turning on. A small "keeper" circuit prevents the charge on the state wire from gradually leaking away.

Our preferred GasP circuits run at the speed of a five-inverter ring oscillator. In 90nm technology that's about 4 GHz. We use a form we call 6/4 GasP that has forward latency of 6 gate delays and reverse latency of 4 gate delays. We choose to make forward latency greater than reverse latency because copying data forward is a real action whereas moving an empty space backwards changes only a state wire. The sum of forward and reverse latency is cycle time. We choose cycle time to be 10, the cycle time of a five-inverter ring oscillator. GasP circuits with a cycle time of 10 are possible with forward and reverse latencies of all values from 1 to 9; there are design cases where we might want to use another one of these over our default 6/4 GasP form.

MORE CONSERVATIVE LOGIC

Those who prefer to drive wires from only one end use a different FULL and EMPTY protocol. They need two state wires rather than one, and so sending N bits requires N+2 wires. The sender signals the presence of data on a "forward" or "request" wire. The receiver signals acceptance of data on the "reverse" or "acknowledge" wire.

Micropipelines, my 1988 Turing award paper, outlined a "two phase" or "transition" protocol for the request and acknowledge wires. Each *transition* of the request wire indicates the presence of a new data element. Up transitions have exactly the same meaning as down transitions. That's exactly what today is called "Double Data Rate" or DDR signaling. Today's DDR protocols with source clocking are exactly what you get if you omit the acknowledge wires. The acknowledge wires provide a double data rate acceptance signal that permits self-timed operation.

Because the request and acknowledge wires change just once for each data element, the state is FULL when the request and acknowledge differ in state and EMPTY when they match. The logic for a FIFO with this protocol is exactly the same as for GasP, but must first compute FULL or EMPTY from the two pairs of state wires. That requires an XOR gate that GasP can safely omit. Another form of asynchronous pipeline we use, called "Click," includes the XOR gates and uses N+2 wires. Click circuits are readily implemented using only standard cell libraries. Indeed the inventors of Click intended their circuits to fit as easily as possible into standard design flows. One of them, Willem Mallon, now works with us in the ARC.

Those who object to the transition signaling, double data rate, or two-phase protocols are free to use a four-phase protocol on the request and acknowledge wires. In the four-phase protocol, only rising transitions carry meaning; falling transitions are ignored. There's been much literature devoted to how best to get rid of the meaningless falling transitions with the least increase in the complexity of the logic.

CANOPY GRAPHS

It is handy to think about asynchronous pipelines in terms of a "Canopy Graph." The canopy graph is named after its tent-like shape. It plots throughput vertically and occupancy horizontally. We often measure pipeline circuits by making a racetrack ring pipeline and sending data elements around it like racecars.

I was surprised to figure out that 6/4 GasP rings work with any number of stages, even or odd. Because I knew that a ring oscillator must have an odd number of stages, I initially made my GasP rings with an odd number of stages, just in case. Only later did I realize that 6/4 GasP rings with either even or odd numbers of stages work equally well. Moreover, one can easily initialize a 6/4 GasP ring with any number of data elements from none to its full capacity. Some other FIFO control circuits introduce protocol polarity issues that limit either the number of stages that can form a ring or the number of initially full elements such a ring can hold.

Figure 1 shows measured and predicted canopy graphs for an 11-stage GasP ring pipeline. If the ring is empty, there's no throughput. If the ring is entirely full, there's likewise no throughput. A single data element, or one empty space, produces some throughput. Two data elements or two empty spaces provide twice as much throughput. Notice that this ring of 6/4 GasP elements gets more throughput from each vacancy than from each data element. The greater reverse throughput happens because 6/4 GasP can move an empty space backwards in four time units but takes six time units to copy data forward. In between completely empty and completely full there's an occupancy that achieves maximum throughput. Because bubbles move faster than data, maximum throughput for 6/4 GasP requires the pipeline to be about 60% full.

GasP NETWORKS

Building a pipeline network requires two special kinds of GasP modules: the addressable branch and the demand merge. The addressable branch GasP module has one input and two outputs. It sends data elements to the right or left according to an address bit carried with the data payload. I designed my addressable branch GasP module to run just as fast as an ordinary FIFO GasP module.

The demand Merge module has two inputs and one output. Like a freeway merge, it applies the rule of "first-come-first-served" to the two incoming channels. The demand merge module requires arbitration because the two inputs might arrive "exactly" at the same time. I designed my demand merge GasP module to run at the same speed as an ordinary FIFO GasP module except in the *very unlikely* case of a near tie, which is slower.

Figure 2 shows the canopy graph for two rings of 100 GasP stages each that share a 50-stage center section. The first stage of the center section is a demand merge module; the last stage of the center section is an addressable branch module. The two rings form a kind of infinity symbol; we named the test chip *Infinity*.

Notice that Infinity's canopy graph is flat on top. Usually a flat top indicates a slow stage, and suspicion naturally fell on my demand merge module. But it was a careful design and shouldn't cause such trouble. Careful examination revealed that a few long wires are responsible for the flat top. Prasad Joshi, then a Master's student at USC under Peter Beerel, back-annotated the long wire lengths to re-calculated the logical effort of the driving GasP modules and was thus able to predict the delay variation observed by the chip experiment.

COMPUTER ARCHITECTURE

Why, you may ask, did I spend seven pages on preamble before getting to the important subject of computer architecture? I wanted to set the stage by saying what I know to be true: we can build outstandingly fast self-timed on-chip data networks. They can branch and they can merge in whatever patterns you choose. In effect, we have the elements of a self-timed freeway system for data. Data elements can travel though such a network like trucks on the freeways, carrying routing information with them. In 90 nm technology the throughput is about 4 billion data elements per second. Because forward latency is only 6/10 of cycle time, each data element reaches about 6.6 billion branch points per second. I take as a given that on-chip communication can be fast and flexible.

The computer architect today is faced with the problem of how to fill a silicon chip with useful circuitry. The no-brainer choice is to replicate over and over what we already know how to build, namely a microprocessor *core*. This is called “multi-core.” I think multi-core is a bad idea. Why?

For one thing we can’t run all those processors concurrently because running them together will melt the chip. For another thing, even if we could run them all at once, the chip lacks sufficient memory bandwidth to keep them all busy. And for a final thing, our software friends haven’t a clue about how to use five processors together, let alone 50 or 100. The only good thing about multi-core is that step and repeat will fill the chip with circuitry – it’s truly a “no-brainer.”

A recent paper by a Stanford group led by Horowitz points out an obvious truth: special purpose hardware can do computations faster and use less energy than general purpose hardware. One easy way to see “why” is that general-purpose hardware must “fetch the algorithm from memory.” General-purpose hardware must read software instructions to control its actions. Special purpose hardware has a narrower scope of capability, but avoids the energy and time cost of generality.

My proposal is that we put relatively few general-purpose microprocessors on a chip. That will leave a big open space to fill. Let’s fill the remaining space with special purpose hardware. Now we can start the important discussion about *which* special purpose hardware to include. The machine will be very good at the things built into it, and no worse than a big multi-core chip at everything else.

WHICH SPECIAL PURPOSE DEVICES

It shouldn’t be hard to guess at the top ten special purpose choices. Graphics provides a rich set of algorithms that we already know how to build into silicon. Matrix arithmetic is a part of that, and again we know much that could be useful. I’m going to spend my time talking about some less obvious choices.

Sorting is my prime candidate. I remember the old days when tape-to-tape sorting speed was an essential feature of many machines. I believe that a low-energy, high-speed sorting system could have major impact on many tasks. If sorting were fast and inexpensive, at least some “pointer chasing” tasks could be converted to sorting tasks. I’ve talked already about fast sorting.

Crypto is another obvious candidate. Well-known algorithms could be built into the hardware – security should depend only on the key. Indeed, one might think of encrypting and decrypting as a routine part of memory access to minimize the time and area over which data and program are “in the clear.”

Memory access is a rather less obvious choice. There have been several proposals to provide more indirection in memory access to facilitate garbage collection. How much energy could we save by checking memory accesses on a block-by-block rather than on a word-by-word basis? Array access mechanisms with stride and extent built into hardware could reduce software time and energy for the low level tasks required to step addresses through memory, to say nothing of relieving pressure on register space. Think of a special purpose memory access device able to execute your choice of a few different access patterns at little or no cost to software.

Full-precision floating point is another option. Adding a million floating-point numbers accurately is difficult because the sum one computes may depend on the order of addition. A proper sum prevents a large value from overwhelming the contributions of many small values. Instead of sorting the numbers first, notice that the 11-bit exponent of a double-precision floating-point number is only 2048 positions. Double-precision floating-point numbers carry a 53-bit mantissa. Thus a fixed-point accumulator containing 2048 + 53 = 2101 bits can hold all double-precision floating-point values. Moreover, a 2101-bit accumulator is only a tiny part of a modern chip. It can add floating-point numbers with full precision by converting each to its fixed-point form before adding. The accumulator could use a carry-save form to conserve time and energy. Converting a sum back to floating-point notation will require carry resolution and normalization. If you allow me to do the easy cases fast and the harder cases more slowly, I’m sure I can produce any number of efficient ways to convert back to the standard floating-point form.

SELF-TIMING AND COMPUTER ARCHITECTURE

A notion of “data-absent” appears in many advanced programming languages. A String in Java can hold zero or more characters. A List of Names can have the type List<Name> and yet be empty. That’s a typed kind of emptiness. Yet the registers of my computer, r0 to r15, for example, have no concept of either type or emptiness. Might there be value in making “empty” a first class register value?

I think so. For example empty registers might serve for communication. Suppose the machine instruction: ***add r0, r1 -> r2*** finds r1 empty. Such an instruction might stall until r1 gets a value. Because memory fetch takes so long, many score boarding schemes have a way to mark a register as “pending” until a value returns from memory. A similar approach might let programs communicate if one program could fill and another program could drain the register in question.

Including EMPTY or FULL markers in a register is reminiscent of the FIFO discussion we shared earlier. Recall that a shared value that can be either EMPTY or FULL provides the essence of pipelining. I remind you that the UNIX pipe is one of the least painful forms of concurrency known to the programming world.

A critical part of computer architecture is how to access a large register file with only a few bits of register address. Sparc uses “register windows” for just that purpose. Groups of registers form a stack with push and pop operations that bring different groups into play. Register windows allow a 5 bit register address to choose from more than 2**5 registers, albeit not all at the same time.

Let us think of alternate register addressing schemes. Let us think of such schemes as providing virtual registers much as a memory map provides virtual memory. Such a scheme might map five-bit virtual register addresses into eight-bit real register addresses. Thought of as virtual addressing, would register windows be your scheme of choice?

My scheme of choice would put a FIFO of registers at each register address. If each such FIFO were eight registers long, a five-bit register address, for instance, could access 2**5 FIFOs, each 8 registers long, for a total of 2**8 registers, albeit not all at once. Instructions accessing a FIFO would require extra bits to copy, consume or recirculate values from a FIFO. I note that a FIFO register is ideal for register renaming in loops.

To implement FIFO registers requires a way to represent EMPTY or FULL. Such an architecture must offer a way to stall when attempting to write into a FULL FIFO or when attempting to read from an EMPTY FIFO. Such FIFO registers might be ideal for decoupling different phases of a software system, just as the UNIX pipe uses buffer memories to provide time independence to the piped processes.

A FINAL WORD ABOUT ENERGY

Clock gating is an important tool for reducing the energy consumption of a running computer. Clock gating avoids wasting energy on unnecessary clock signals to idle parts of the machine. Synchronous systems require elaborate logic to decide when to omit clock signals.

The pipelines I’ve been talking about here offer clock gating for free. They produce only those timing signals necessary for copying. When data enter a communication channel, a state wire changes state to announce that the data are ready. If and only if latches can capture the data, a GasP circuit produces a suitable timing signal that serves the role of a clock. Clock gating is truly free. Moreover, because we depend on timing only for local signals, clock skew is a non-problem.

In our laboratory at Portland State University we've begun to explore another power-saving measure called "power gating." Power gating avoids not only unnecessary energy consumption of unused clock signals, but also unnecessary energy loss to leakage of idle circuit elements. Power gating reduces the supply voltage to idle circuits to reduce their leakage and consequent energy loss.

We've found power gating remarkably easy to implement in GasP. Power need be applied to a circuit only when it is in use OR in anticipation of impending use. That OR function is remarkably like the AND function already in each GasP module. Moreover, one of our students has simulations showing that the OR function and some additional drivers can turn power on before it's needed. I love this idea because it's a remarkably simple solution to a problem that, at first, seemed hard.

SIMPLICITY

Simplicity is a necessary concomitant of reliability. Unfortunately some manufacturers think this too high a price to pay. Sir C.A.R. (Tony) Hoare, KBE.

The best Science and Engineering find simple answers. It has been my pleasure to observe close up the birth of a number of simple solutions. Here are five examples:

1) Henri Gouraud noticed that simple linear shading could hide the edges of a polygonal surface from the human eye thus producing an illusion of smooth surface. This simple idea is the basis of his PhD thesis.

2) Mark Raibert noticed that an automated pogo stick could use three entirely independent feedback systems to achieve stable operation. The first adds energy to keep the bouncing height constant. The second uses the hip joint during flight to ready the foot position for the next landing: placing the foot behind or in front of the balance point speeds or retards forward motion. The third uses the hip joint when the foot is on the ground to adjust the attitude of the body. Mark's promotion to a tenured faculty position followed from the simplicity of this idea.

3) I noticed a mathematical relationship between the complexity of a logic gate and its loss of ability as an electrical amplifier. I named this simple idea "Logical Effort." Because doing logic and driving electrical loads are mathematically equivalent, one can easily choose transistor strengths for near-optimum performance. David Harris, Bob Sproull and I published a book on the subject.

4) Scott Fairbanks and I noticed that self-timed pipelines require only a single AND function. We called the resulting circuit family GasP. Great performance

results from this simple observation because the Logical Effort of GasP circuits is very small.

5) It now appears that power gating may yield to an equally simple solution. If so, we may be able easily to save energy in a variety of pipeline circuits.

References:

I. Sutherland. “The Tyranny of the Clock,” Remarks for the ACM Turing Centenary, *Internal ARC Report ARC# 2012-is13, Portland State University*, 16 June 2012. Available at: http://arc.cecs.pdx.edu/publications.

ARC Films – "Paradigm Shift.” Available at: http://arc.cecs.pdx.edu/publications.

R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. “Understanding Sources of Inefficiency in General-Purpose Chips,” *Communications of the ACM, Vol. 54, No. 10*, pp.85-93, 2011.

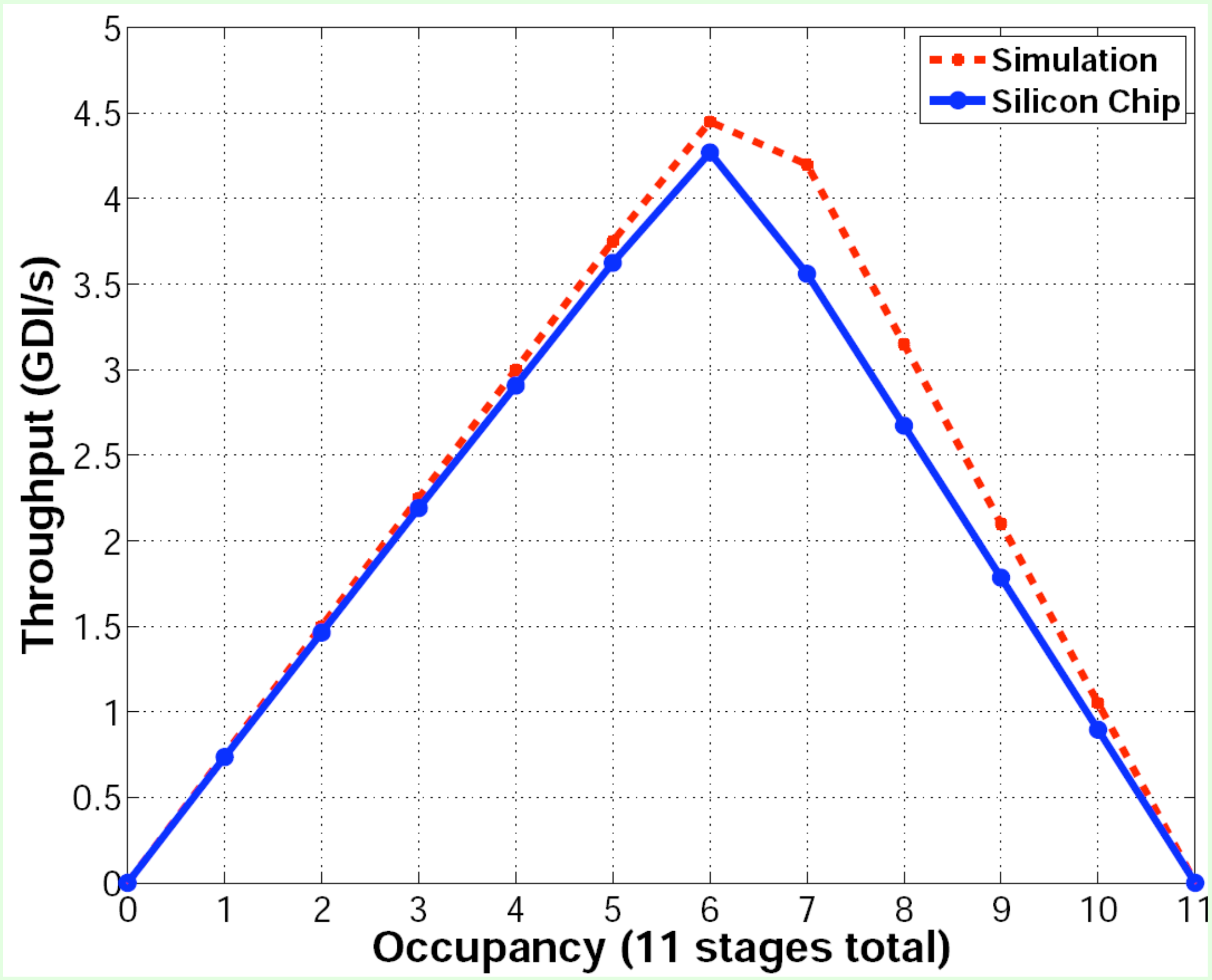
A. Peeters, F. te Beest, M. de Wit, and W. Mallon. “Click Elements: An Implementation Style for Data-Driven Compilation,” *Proceedings IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC),* pp. 3–14, 2010.

I. Sutherland. "Micropipelines," *Communications of the ACM, Vol. 32, No. 6*, June 1989, pp. 720-738.

P. Joshi, P. Beerel, M. Roncken, and I. Sutherland. “Timing Verification of GasP Asynchronous Circuits: Predicted Delay Variations Observed by Experiment.” In *D. Dams, U. Hannemann, M. Steffen (Eds.) Willem-Paul de Roever Festschrift, LNCS 5930*, pages 260-276. Springer-Verlag Berlin Heidelberg, 2010. Available at: http://arc.cecs.pdx.edu/publications.

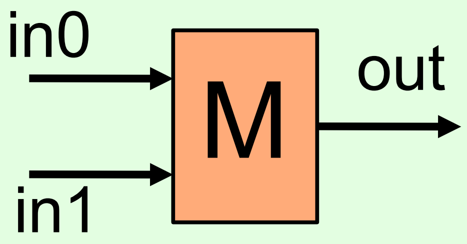
I. Sutherland, R. Sproull, D. Harris. “Logical Effort: Designing Fast CMOS Circuits,” *Morgan Kaufmann Publishers, Inc.*, 1999.

Ring: Throughput vs Occupancy

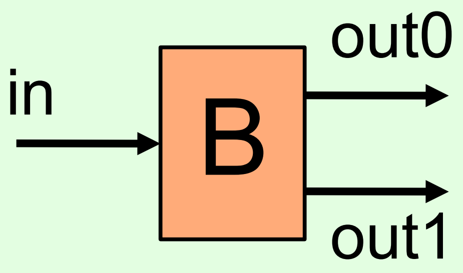


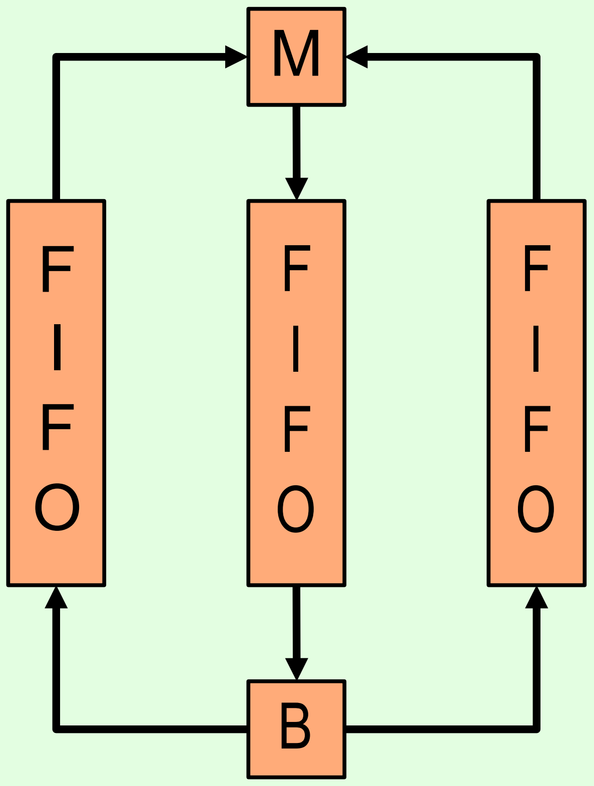
Infinity test

Merge

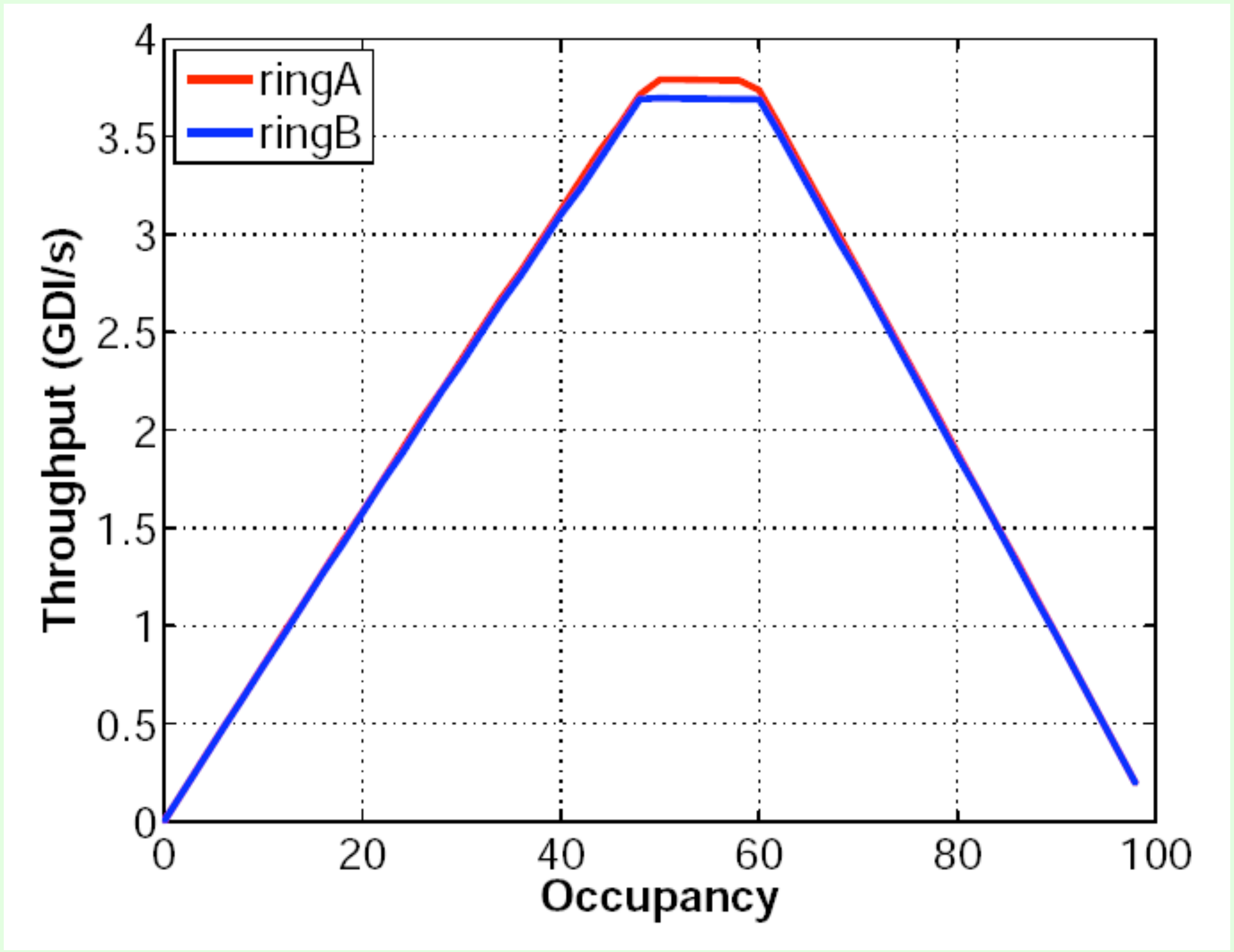


Data-directed Branch



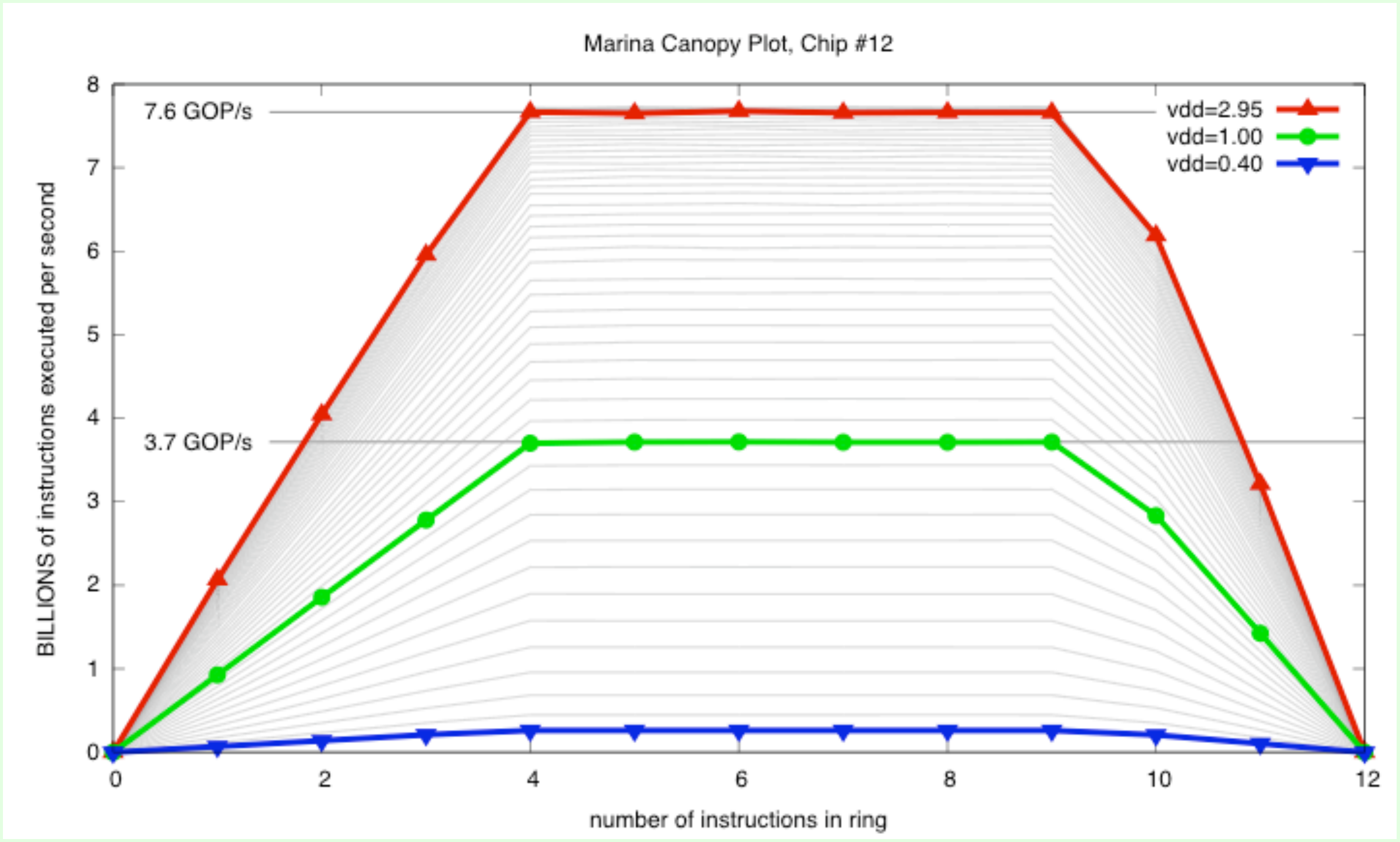


Infinity: Throughput vs Occupancy



Dock: Throughput vs Occupancy

1.0 volts (green), 0.4 volts (blue) and 2.95 volts (red & died)


Marina Canopy Plot, Chip #12
7.6 GOP/s
3.7 GOP/s
vdd=2.95
vdd=1.00
vdd=0.40
BILLIONS of instructions executed per second
number of instructions in ring