Rethinking Instruction-Set Design for Energy-Efficient Computing

Jeff Butera
Haverford College
jbutera@haverford.edu

Sol Lutze
Haverford College
slutze@haverford.edu

David G. Wonnacott
Haverford College
davew@cs.haverford.edu

Abstract

Although the RISC revolution improved the regularity of instruction sets, and improvements in manufacturing moved floating-point arithmetic operations from software to co-processors to processor cores, the basic semantics of arithmetic operations have remained largely unchanged since the origins of the microprocessor. We have been exploring the possibility of adopting new semantics for arithmetic and other operations to facilitate low-power computing while retaining high execution speed.

We have initially explored our ideas in the context of integer arithmetic instructions implemented with two- or three-input logic gates. In this context, we believe it is possible to achieve significant improvements in energy use while maintaining throughput for some codes. Furthermore, our approach can exposes novel forms of instruction-level-parallelism which are not readily expressible in traditional instruction sets.

Low-energy computing is a new area for all of us, and we are far from certain about the benefits of this approach. Clearly they are highly sensitive to details of chip manufacture. Before moving forward, we hope to engage the WEED community in a discussion about whether our approach might have any value or simply be an interesting academic exercise. (In an attempt to avoid wasting WEED reviewers' time, we did run these ideas past several colleagues who work on computer architecture, and they were consistently rated "not obviously insane". If these ideas are deeply misguided, we ask that the WEED reviewers be direct and if possibly provide specific citations.)

1 Introduction

The fundamental semantics of microprocessor arithmetic operations have remained largely unchanged for many decades — a single instruction causes a mathematically meaningful operation such as addition or multiplication to be performed on two (or perhaps three) operands. This concept seems fundamental to the very idea of computer arithmetic, and has remained through the RISC revolution, and improvements in manufacturing that moved floating-point arithmetic operations from software to co-processors to processor cores, and even through the exploration of novel number representations for GPU's.

We have been exploring the possibility of adopting new semantics for arithmetic and other operations to facilitate low-power computing while retaining high execution speed. Specifically, we have begun to design an instruction set whose semantics are defined in terms of the evolving state of arithmetic circuits, rather than their ultimate result. As in the case of current high-end processors, achieving correctness and high-performance with our design would require sophisticated code generation based on reasoning about the program and the state of the elements within the processor (hopefully via an automated compiler). As in the case of current high-end processors, meaningful results can be produced slowly at low speed without sophisticated static analysis. However, when we cannot ensure high performance via static analysis, our processor design forces us to select slow execution (at low power) rather than expending power to attempt to re-gain performance.

The essence of our approach is to examine various of breaking in operation into a sequence of steps (essentially in the same way stages of a pipeline are laid out) and defining these as instructions in our instruction set. The approach is most advantageous when these intermediate steps have some useful semantic properties (often related to any invariants from which the correctness of the result follows). In some cases, the availability of these intermediate results at the instruction set level opens opportunities for exploitation of instruction level parallels some that cannot be expressed or captured in a traditional pipeline construction set.

Below, we explore a simple example of this approach and outline the questions we must answer to determine if the approach has value.

2 Motivating Example

The simplest example of our approach is the construction of a low-power, high-throughput adder from two- or three-input logic gates. This construction follows the design of a standard "carry-save adder", but exposes the intermediate registers at the instruction set level.

2.1 Adders and Carry-Save Adders

A "ripple carry" adder is probably the conceptually simplest way to construct an n-bit binary adder, and is familiar from any number of first-semester digital circuitry textbooks. Numbers A and B, with binary digits $A_{n-1\ldots 0}$ and $B_{n-1\ldots 0}$ are added by defining C_{i+1}, the carry from digit i into digit $i+1$, as true whenever the value in column i is $\geqslant 2$, i.e. when the majority of A_i, B_i, C_i are true ($C_{i+1} = m(A_i, B_i, C_i)$), with C_0 either signifiying incoming carry or being simply 0 for the simple sum $A + B$; The i^{th} digit of the sum, S_i, is true if the parity (3 or 1 inputs are true) of A_i, B_i, C_i is true ($S_i = p(A_i, B_i, C_i)$).

In a ripple-carry adder, the output of $m(A_i, B_i, C_i)$ in column i is simply wired directly to the C_{i+1} inputs of the m and p circuits in the next position. All digits of A and B are presented simultaneously, along with C_0, and the values of C_i and S_i stabilize in order of increasing i, with all bits of the result being ready within $n+1$ times the maximum propogation time through the majority circuit m.

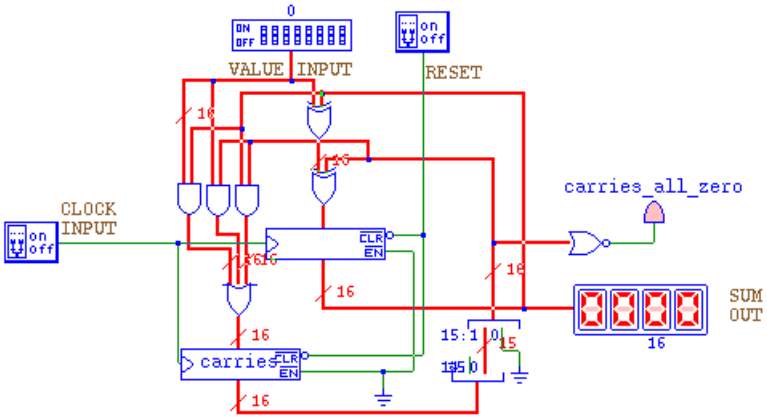

Figure 1. Carry-Save Adder Circuit

The "carry-save" variant simply adds registers R_C to hold each bit of C and R_S to hold each bit of S after a single delay of time max (delay(m), delay(p)). The values of A, B, and C_0 are then presented in the first clock cycle, after which A and C_0 are held at 0 and B is fed the value of S. After n clock cycles, C must be 0 and the final answer will be present in S. At first this seems to offer no advantage, as it adds at least n register-load delays to the time required to produce the final result. However, the semantics of the intermediate result are well-defined: at any time, $R_C + R_S = A + B$ (this invariant is what ensures we produce the correct sum). Thus, we can at any time select a non-zero value for A, *even after only one clock cycle*, and produce a sum of three values. In this fashion, k values can be added in $n + k$ quick clock cycles (long enough to produce one digit) rather than k slow clock cycles (long enough for all bits of a ripple carry to stabilize). A slightly simplified version of this circuit, in which B is always defined as S (initially reset to 0), is shown in Figure 1.

This circuit is, of course, most useful for algorithms that sum large numbers values; it is a classic way to implement a multiplier circuit in which two n-bit integers are multiplied in $2n$ quick steps with comparitively little hardware (just adding shift registers to Figure 1).

2.2 An Observation about Implementation

There are, of course, many other ways to implement quick addition circuits, as can be found in most textbooks on computer arithmetic, with various design tradeoffs.

A carry-lookahead adder (see [Par00], Chapter 6) uses additional gates to compute all bits of C_i very quickly (e.g., in $O(1)$ or $O(\log(n))$ time, e.g. using two-level-design to combine inputs A_j and B_j, for $j < i$). This is essentially a way of buying speedup with additional power, with the number of gates that may switch during a given addition growing more than linearly with n (with $O(n^2)$ gates to get $O(1)$ time).

A Manchester carry chain treats transistors as switches (rather than tools with which to create gates), employing a few quick computations for each bit and a sequence of n such switches to compute all bits of C_i very quickly (in time for power to flow through $O(n)$ such switches that are each set after a $O(1)$ time capacitor-charging step based on a simple computation on A_i and B_i). Like classic carry-lookahead, this approach uses power to buy speed, and (for CMOS manufacturing) best suited for $n \leqslant 8$ (see [Par00], Chapter 5.6).

Carry-save and carry-lookahead are not mutually exclusive. For example, 4-bit carry lookahead produces the carry-out from each group of four bits in each clock cycle, essentially producing a hexidecimal carry-save adder, and other speed/power tradeoffs are possible.

2.3 Instructions for a Carry-Save-Adder-Based Architecture

In a traditional ISA, carry-save addition might be used to produce a n-bit integer add and multiply operations requiring n and $2n$ clock cycles, respectively. Hybrid carry-save/carry-lookahead could improve the speed (e.g., to $\frac{n}{4}$ cycles for hexidecimal carry-save addition). Multiple such units could be implemented on a chip, and pipelined instruction execution could then be used to improve average IPC (instructions per cycle, i.e. throughput).

Pipelining, however, requires significant chip area and consumes significant energy. Consider a traditional `ADD(R1, R2, R3)` operation to place $R_2 + R_3$ into R_1 — the semantics of a traditional ISA require that subsequent uses of R_1 (until the next definition) yield $R_2 + R_3$, and significant design effort and execution power go into meeting this requirement when an instruction making use of `R1`, or another `ADD` instruction, arrives at the processor before this `ADD` writes its result into `R1` (see Section 4.1).

Instead, we define the semantics of instructions around the implementation, and require that programmers (or compilers) ensure that the semantics of addition are preserved in their result. In other words, the instruction stream must ensure that the final sum of the addition hardware is available when needed. Instructions, whether involving addition or other operations, can directly address the registers of each carry-save adder as the source or destination.

Consider the task of extracting certain bits (described by a mask in `R1`) from the values in registers `R3` and `R4`, arithmetically summing these values, and turning on other bits as per a value in `R2`, with the result being placed in `R5`, i.e., computing $R_5 \leftarrow (((R_1 \& R_3) + (R_1 \& R_4)) | R_2)$. A traditional ISA might express this computation as shown on the left of Figure 2, whereas we would express it as shown on the right. On the right, we assume `R8` corresponds to the "S" register of a carry-save adder, and a write to `R9` corresponds to provision of the "A" input for this adder, and that we know that this adder is idle ("C" is 0); on the left, we assume `R6` is available. A traditional superscalar architeture could simultaneously perform both `AND` operations and feed their results into the `ADD`; initiating these four instructions in fewer than few cycles. We believe a VLIW-style static combination of the two `AND` operations would be more suited to our low-power emphasis if we wish to allow multiple instruction issues per clock cycle.

An instruction set with multiple adder and multiplier units would statically assign register numbers to the relevant registers of each unit. Thus, two independent multiplications (or additions) could execute concurrently if they place their multiplicands in distinct registers. A fused-multiply-add instruction $XY + Z$ arises naturally from this design when Z is delivered to the carry-save adder that will receive the product XY before (or as) X and Y are delivered to the shift registers that feed this adder. A sum of three or more operands can be quickly computed with *one* adder unit by delivering two operands initially and then feeding the others into the A input in the following instructions.

2.4 Opportunities and Challenges for Program Optimization

Our approach presumes that all opportunites for instruction-level parallelism (ILP) would be detected and scheduled statically, as was the case with the very-long instruction word (VLIW) approach. While this approach was not widely adopted in the era when instructions-per-cycle drove chip design, be are optimistic that it may prove more valuable in optimizing instructions-per-joule, especially when combined with our ISA design.

By providing funtional units with different performance characteristics (e.g. both a "binary" and a "hexidecimal" carry-save adder), delays can be not just anticipated but mitigated by instruction ordering. An addition whose result is not needed for n cycles can be placed on a binary CSA, and one whose result is needed in $\frac{n}{4}$ placed on a hexidecimal CSA, unless we choose to optimize a given application for power use rather than speed, in which case the latter might also be placed on a binary CSA and no-operation used to produce the necessary delay.

3 Generalization and Other Opportunities

Integer addition provides a simple example with which to investigate our approach, but it is hardly sufficient for an architecture. We have given some thought to the semantics of other arithmetic, branches, memory operations, and multi-threading in this context.

```
AND(R5, R1, R3)                 AND(R8, R1, R3)
AND(R6, R1, R4)                 AND(R9, R1, R4)
ADD(R5, R5, R6)                  (delay slots filled with
   (possible independent code,     independent code
    or dynamic stall/out-of-order  or no-operations)
    execution if required)
OR(R5, R5, R2)                  OR(R5, R8, R2)
```

Figure 2. Code for $R_5 \leftarrow (((R_1 \& R_3) + (R_1 \& R_4))|R_2)$: Traditional vs. Our ISA

3.1 Arithmetic Beyond Addition/Subtraction

We have initially explored integer addition, but note that integer addition plays an important role in other arithmetic implementations. Integer multiplication requires repeated addition; as discussed above, carry-save adders are particularly well suited to this context. Floating-point addition requires addition of mantissas (essentially as integers, once they have been aligned), and floating-point multiplication requires addition of (integer) exponents and multiplication of mantissas (potentially also based on carry-save multiplication).

3.2 Flow Control and "Branch Kenning"

Branch instructions can interfere with ILP in pipelined processors, and we expect processors based our ISA would be subject to a similar effect. With our approach, this becomes a *correctness* issue rather than a performance issue — consider what happens if the code of Figure 2 braches if the result of the sum is zero, i.e. the `OR` is preceded immediately by `BZ(target)` on the left (assuming the instruction set relies on the traditional notion of global flags) or by `BZ(R8, target)` for our code (we do not believe processor status flags would fit well into our approach). In the absence of sufficient instructions above this branch, our code would branch incorrectly.

Pipelined systems devote a certain amount of hardware (and power) to "branch prediction" [HP03] (which should perhaps be called "branch forecasting", since in may predict incorrectly). We could envision similar hardware, but it is antithetical to our low-energy approach.

Instead of hardware directed toward such branch forecasting, we have devolped a technique we call "branch kenning", which branches once the direction of the branch is certain (even if the final arithmetic result is not known). Following our carry-save-adder example, note that we can prove a `BZ` instruction *must not* be taken if there is a non-zero value for any bit S_i for some if such that $C_{j \leqslant i}$ are all zero. Note that for a loop index that is counting down toward zero, most iterations require only a few of the rightmost bits to prove this is so. Similarly, we may be able to prove a non-negative result of a multiply long before the full value is known.

3.3 Memory Operations

Memory operations (`LOAD` and `STORE`) typically allow the addition of a constant offset to a base register; if we envision addition as a many-cycle instruction, this would greatly hinder performance of memory operations. One possible way around this would be to allow a bitwise or of a constant offset to a base (presumably ending with 0's). If records and stack frames were aligned in the proper way, the effect of addition of an offset could be achieved quickly in this way.

Given our focus on codes that can only be optimized with static analysis, an explicitly-managed cache, as used in the Cell processor, might make sense with our design. We have not explored this in detail. This would also help us statically predict which memory operations may take a long time, which would be helpful in deciding when to switch threads in a multithreading implementation.

3.4 Multithreading

The proposed instruction set architecture has the potential to further exploit parallelism and increase overall instruction throughput by adding multithreading capabilities. Theoretically, our design should should be able to handle multithreading with only minimal hardware additions, e.g. registers and program counters to hold the architectural state when threads are switched and a fetch unit that can schedule multiple threads[HH07]. The thread scheduler could be tuned to only switch threads when a long latency instruction, such as a LOAD that experiences a cache miss and must access main memory, is encountered.

The next logical optimization, in terms of increasing parallelism and throughput, would be to investigate simultaneous multithreading. Our design would not be compatible with the dynamic demands of SMT. SMT is most efficiently implemented on top of superscalar processors because multiple instructions must be simultaneously issued in the same clock cycle.

Özer et. al.[ÃCS01] outline a implementation for a VLIW processor with multithreading capabilities. They use a technique called operation welding to combine operations from different threads to more fully utilize the hardware resources. This multithreaded design experienced a 23% speed-up with two threads compared to a single-threaded VLIW design using the SPECint95 benchmark; at this point we can only guess how much multithreading would speed up our design.

4 Anticipated Costs and Benefits

Here we examine the relative costs of our approach versus the traditional costs of superscalar pipelined processors.

4.1 Costs of Superscalar Pipelining

Thus far we have been unable to determine exact costs of optimizations in pipelining; however, we have been able to determine which factors do cost, which we can compare to our own propposed design.

4.1.1 Out-of-order Execution

Out-of-order execution ([HP03], Chapter 3.2) refers to the reordering of instructions to work around potential hazards, such as overlapping operations that modify the same variable; in such a case, the apparent dependency may be resolved by shifting independent operations from later in the program to act as a buffer between the two instructions. Although some operation reordering can be performed at compile time, often conflicting instructions will be undetectable until runtime, preventing us from transfering the full burden of operation reordering off of the hardware.

4.1.2 Register Renaming

Similar to out-of-order execution, register renaming ([HP03], Chapter 3.7) allows for register assignments to be altered to eliminate apparent dependencies; if each new assignment generates a new variable (or reuses old variables only sparingly), fewer dependencies exist and thus more parallelization can be exploited. As with out-of-order execution, some of these dependencies are not visible until runtime.

4.1.3 Branch Prediction

Heuristics exist in both compiler and hardware designs to allow hardware to anticipate the results of branches in order to pre-load instructions from the correct destination, preventing pipeline stalls on branches ([HP03], Chapter 3.4). This prediction may incur power costs, especially in the event of misprediction, where incorrect operations must be discarded and old values restored to their proper registers.

4.1.4 Forwarding

Forwarding ([HP03], Appendix A.3), also called bypassing, allows one instruction to pass its results to more recent instructions before that result is written back to a register, thus allowing dependent instructions to be executed in succession more quickly. Obviously, this can only be performed in hardware since it relies on making data available quickly rather than on eliminating dependencies.

4.2 Anticipated Benefits and Limitations

One cost of this approach is, of course, that it "bakes in" implementation choices about high-level operations. A revolutionary new approach to, say, addition could be implemented cleanly into an existing instruction set, but would require rebuilding of the instruction set level with our approach. This, in turn, would require recompilation of all source code and rewriting (or automatic transcoding) of binaries and assembly language files. However, these approaches are already required to re-performance-tune applications for new implementations of an existing chipset.

5 Questions of Interest/Future Work

A number of important questions remain unanswered by our work so far, including:

5.1 Generalization

How well will this approach work for instructions that don't rely on a step equivalent to integer addition?

5.2 Other Relevant Related Work

Has there been other work in re-thinking ISA's for energy-efficient computing? What about other low-energy implementation techniques that support or undercut our approach (e.g., if the Manchester carry chain can be implemented on a large scale, it would make all examples discussed above irrelevant)?

5.3 Empirical Analysis with Benchmarks

Any credible argument for or against this approach must be based on empirical data about the speed achieved and energy required. What degree of implementation is needed? At what level of abstraction? What is the easiest way to accurately measure energy consumption on this scale?

5.4 Empirical Data for Current Processors

While research on pipelining and pipelining support optimizations (Section 4.1) present considerable empirical evidence of speed improvements, they are much quieter on the topic of energy consumed to produce these gains. In particular, some have argued[HS99] that some of these techniques are not worthwhile even in existing designs. How can we get more information about the energy costs of pipelining? What about other elements of (dynamic) superscalar processor implementation?

5.5 Objective Functions for Measuring Cost/Benefit

The driving motivation behind rethinking the semantics of arithmetic and other operation is to create an instruction set that promotes low-power computing without negatively impacting performance. There are several specific metrics that can be used to quantify the energy-efficiency of a microprocessor design. Gonzalez and Horowitz[GH96] outline the Energy-Delay Product (ED) as an effective metric for measuring power with respect to performance. The ED Product measures the product of the total energy consumed during a specific execution and the total end-to-end execution latency. This metric ensures that the balance between energy and performance is maintained because in order to improve the ED product, either performance must be increased or energy must be reduced without affecting the other.

Another common metric used to measure the energy-efficiency of microprocessors is the Energy-Delay^2 Product. This metric expands on the ED Product by placing more emphasis on performance. This means that more energy can be consumed in order to increase the performance of the processor. Conversely, measuring the Energy^2-Delay Product would allow performance to be sacrificed so that energy consumption can be decreased[ACP11].

We believe the ED product is the appropiate metric to use when comparing an implementation of our proposed ISA to existing designs. Is this right?

Bibliography

[ACP11] Massimo Alioto, Elio Consoli, and Gaetano Palumbo. From energy-delay metrics to constraints on the design of digital circuits. *International Journal of Circuit Theory and Applications*, 2011.

[GH96] R. Gonzalez and M. Horowitz. Energy dissipation in general purpose microprocessors. *IEEE Journal of Solid-State Circuits*, pages 1277–1284, September 1996.

[HH07] David Harris and Sarah Harris. *Digital Design and Computer Architecture*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[HP03] John L. Hennessy and David A. Patterson. *Computer architecture - a quantitative approach, 3rd Edition*. Morgan Kaufmann, 2003.

[HS99] S. Hily and A. Seznec. Out-of-order execution may not be cost-effective on processors featuring simultaneous multithreading. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, HPCA '99, pages 64–, Washington, DC, USA, 1999. IEEE Computer Society.

[Par00] Behrooz Parhami. *Computer arithmetic: algorithms and hardware designs*. Oxford University Press, Oxford, UK, 2000.

[ÃCS01] Emre ÃŰzer, Thomas Conte, and Saurabh Sharma. Weld: A multithreading technique towards latency-tolerant vliw processors. In Burkhard Monien, Viktor Prasanna, and Sriram Vajapeyam, editors, *High Performance Computing âĂŤ HiPC 2001*, volume 2228 of *Lecture Notes in Computer Science*, pages 192–203. Springer Berlin / Heidelberg, 2001. 10.1007/3-540-45307-5$_1$7.