Dynamic Functional Unit Assignment for Low Power

Steve Haga, Natasha Reeves, Rajeev Barua
Dept of Electrical & Computer Engineering
University of Maryland, College Park

Diana Marculescu
Dept of Electrical & Computer Engineering
Carnegie Mellon University

Abstract

A hardware method for functional unit assignment is presented, based on the principle that a functional unit's power consumption is approximated by the switching activity of its inputs. Since computing the Hamming distance of the inputs in hardware is expensive, only a portion of the inputs are examined. Integers often have many identical top bits, due to sign extension, and floating points often have many zeros in the least significant digits, due to the casting of integer values into floating point. The accuracy of these approximations is studied and the results are used to develop a simple, but effective, hardware scheme.

1 Introduction

Power consumption has become a critical issue in microprocessor design, due to increasing computer complexity and clock speeds. Many techniques have been explored to reduce the power consumption of processors. Low power techniques allow increased clock speeds and battery life.

We present a simple hardware scheme to reduce the power in functional units, by examining a few bits of the operands and assigning functional units accordingly. With this technique, we succeeded in reducing the power of integer ALU operations by 17% and the power of floating point operations by 18%. In [4] it was found that around 22% of the processor's power is consumed in the execution units. Thus, the decrease in total chip power is roughly 4%. While this overall gain is modest, two points are to be noted. First, one way to reduce overall power is to combine various techniques such as ours, each targeting a different critical area of the chip. Second, reducing the execution units' power consumption by 17% to 18% is desirable, independent of the the overall effect, because the execution core is one of the hot-spots of power density within the processor.

We also present an independent compiler optimization called swapping that further improves the gain for integer ALU operations to 26%.

2 Energy modeling in functional units

A series of approximations are used to develop a simple power consumption model for many computational modules. To begin, it is known [15] that the most important source of power dissipation in a module is the dynamic charging and discharging of its gates, called the *switched capacitance*. This switched capacitance is dependent upon the module's input values [14]. It has been further shown [13, 6] that the *Hamming distance* of consecutive input patterns, defined as the number of bit positions that differ between them, provides a suitable measure of power consumption. In [13, 6], power is modeled as: $Power \approx \frac{1}{2}V_{dd}^2 f \sum_k C_k sw_k \approx \frac{1}{2}V_{dd}^2 f C_{module} h_{input}$, where V_{dd} = voltage, f = clock frequency, C_k = capacitance of output gate k, sw_k = average # transitions for output gate k (called *switching activity*), C_{module} = the total capacitance of the module, and h_{input} = the Hamming distance of the current inputs to the previous ones.

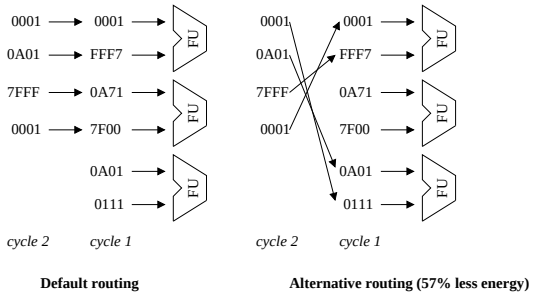

Figure 1. Alternative data routes for a 3-way processor

Since power consumption is approximately linearly proportional to h_{input}, it is desirable to minimize h_{input}. Such a minimization is possible because modern processors contain multiple integer arithmetic logic units, or IALUs, and multiple floating point arithmetic units, or FPAUs. IALU and FPAU are types of functional units, or **FU**s. Current superscalars assign operations to the FUs of a given type without considering power; a better assignment can reduce power, however, as illustrated by the example in Figure 1, which shows the input operands to three identical FUs, in two successive cycles. The alternate routing consumes less power because it reduces the Hamming distance between cycles 1 and 2. Figure 1 assumes that the router has the ability to not only assign an operation to any FU, but also to swap the operands, when beneficial and legal. For instance,

it is legal to swap the operands of an add operation, but not those of a subtract. (A subtract's operands can be swapped if they are first inverted, but the cost outweighs the benefit). Because superscalars allow out-of-order execution, a good assignment strategy should be dynamic. The case is less clear for VLIW processors, yet some of our proposed techniques are also applicable to VLIWs.

Our approach has also been extended to multiplier FUs; these units present a different set of problems. Even modern superscalars tend to only have one integer and one floating point multiplier. Some power savings are still possible, however, by switching operands. In the case of the Booth multiplier, the power is known to depend not only on the switching activity of the operands, but also on the number of 1's in the second operand [12]. Power aware routing is not beneficial for FUs that are not duplicated and not commutative, such as the divider FU.

3 Related work

A variety of hardware techniques have been proposed for power reduction, however these methods have not considered functional unit assignment. Existing hardware techniques are generally independent of our method. For example, implementing a clock-slowing mechanism along with our method will not impact the additional power reduction that our approach achieves.

Work has also been done to reduce the power of an application through compiler techniques [12], by such means as improved instruction scheduling, memory bank assignment, instruction packing and operand swapping. We in fact also consider the effect of compile-time operand swapping upon our results. But the difficulty of compile time methods is that the power consumed by an operation is highly data dependent [10], and the data behavior dynamically changes as the program executes.

The problem of statically assigning a given set of operations to specific functional units has also been addressed in the context of high-level synthesis for low power [6]. In [6], the authors present an optimal algorithm for assigning operations to existing resources in the case of data flow graphs (DFGs) without loops. More recently, [11] shows that the case of DFGs with loops is NP-complete, and proposes an approximate algorithm. The case of a modern superscalar processor employing out-of-order execution and speculation is much more complicated, however.

These difficulties point to the need for dynamic methods in hardware to reduce the power consumption.

4 Functional unit assignment for low power

In this section we present an approach for assigning operations to specific FUs that reduces the total switched capacitance of the execution stage. As shown in Figure 1, we can choose a power-optimal assignment by minimizing the switching activity on the inputs.

```
for i = 1 to Num(M)
 for j = 1 to Num(I) {
  Cost^i_j= Ham(OP1(I_j),OP1(M_i))+Ham(OP2(I_j),OP2(M_i))
  if (Commutative(I_j))
   Cost^i_j= Min (Cost^i_j, (Ham(OP1(I_j),OP2(M_i))+
                             Ham(OP2(I_j),OP1(M_i))))}
```

Figure 2. Finds the cost of every possible assignment.

Our main assumption is that whenever an FU is not used, it has little or no dynamic power consumption. This is usually achievable via power management techniques [16, 1, 2] which use transparent latches to keep the primary input values unchanged whenever the unit is not used. We will also use this hardware feature for our purposes, and thus, we assume that it is already in place for each of the FUs. To reduce the effect of leakage current when an FU is idle, techniques such as those in [9] may also be used, if needed.

Some nomenclature is helpful in the following discussion:
M_j: the j_{th} module of the given FU type.
I_j: the j_{th} instruction to execute this cycle on this FU type.
Num(M), Num(I): # of modules of type M, or # of instructions of the given type. The maximum value for Num(I) is Num(M), indicating full usage of that FU type.
OP1(M_j), OP2(M_j): the first and second operands of the *previous* operation performed on this module.
OP1(I_j), OP2(I_j): the first and second operands of the given instruction.
Commutative(I_j): indicates if I_j is commutative.
Ham(X,Y): the Hamming distance between the numbers X and Y. For floating point values, only the mantissa portions of the numbers are considered.

4.1 The optimal assignment

Although its implementation cost is prohibitive, we first consider the optimal solution, so as to provide intuition. On a given cycle, the best assignment is the one which minimizes the total Hamming distance for all inputs. To find this, we compute the Hamming distance of each input operand to all previous input operands. The algorithm to compute these costs is shown in Figure 2. Once these individual costs have been computed, we examine all possible assignments and choose the one which has the smallest total cost (defined as the sum of its individual costs).

This algorithm is not practical. Since the routing logic lies on the critical path of the execution stage, such lengthy computations are sure to increase the cycle time of the machine. Moreover, the power consumed in computing so many different Hamming distances is likely to exceed the power savings from inside the modules.

4.2 Approximate Hamming distance computation

To reduce the overhead of the above strategy, full Hamming distance computations must be avoided. We achieve this by exploiting certain properties of numerical data to represent an operand by a single *information bit.*

Integer bit patterns It is known [3], that most integer operands are small. As a consequence, most of the bits in a 2's complement representation are constant and represent the sign. It is likely that the Hamming distance between any two consecutive operand values will be dominated by the difference in their sign bits. Thus we can replace the operands by just their top bits, for the purposes of Figure 2.

Table 1 validates this technique. This table records the operand bit patterns of our benchmarks. (Section 6 describes the benchmarks and how they are run.) By combining the data from the table using probability methods, we find that on average, when the top bit is 0, so are 91.2% of the bits, and when this bit is 1, so are 63.7% of the bits.

Floating point bit patterns Although not as common, a reasonable number of floating point operands also contain only a few bits of precision, for three main reasons: (1) the casting of integer values into floating point representation, such as happens when incrementing a floating point variable, (2) the casting of single precision numbers into double precision numbers by the hardware because there are no separate, single-precision registers and FUs, and (3) the common use of round numbers in many programs.

Hence, the Hamming distance of floating point numbers may also be approximated. Floating points that do not use all of their precision will have many trailing zeroes. On the other hand, numbers with full-precision have a 50% probability of any given bit being zero. The bits of a full-precision number can be considered random, so the Hamming distance between a full-precision number and anything else is about 50% of the bits. The Hamming distance is only reduced when two consecutive inputs have trailing zeros.

OR-ing of the bottom four bits of the operand is an effective means of determining which values do not use the full precision. Simply examining the least significant bit of the mantissa is not sufficient, because this will capture half of the full-precision numbers, in addition to the intended numbers that have trailing zeroes. But using four bits only misidentifies $\frac{1}{16}$ of the full-precision numbers. We do not wish to use more than four bits, so as to maintain a fast circuit. Although we refer to an OR gate because it is more intuitive, a NOR gate is equally effective and slightly faster.

Table 1 describes the effectiveness of our approach. From Table 1 we can derive that 42.4% of floating point operands have zeroes in their bottom 4 bits, implying that 3.8% of these are full precision numbers that happen to have four zeroes ($(100 - 42.4) \div 15$) and that the remaining 38.6% do indeed have trailing zeroes ($42.4 - 3.8$). This table also shows that, on average, when the bottom four bits are zero, 86.5% of the bits are zero.

4.3 A lightweight approach for operand steering

In section 4.2, the Hamming distance computations of Figure 2 are reduced to just the Hamming distances of the information bits of the operands; this section examines avoiding the computation of Figure 2 entirely. This is achieved by predetermining the FU assignment for any particular set of instruction operands – without comparison to previous values. Some additional definitions are useful:

OP1	OP2	Commutative	IALU Freq (%)	IALU OP1 prob	IALU OP2 prob	FPAU Freq (%)	FPAU OP1 prob	FPAU OP2 prob
0	0	Yes	40.11	.123	.068	16.79	.099	.094
0	0	No	29.38	.078	.040	10.28	.107	.158
0	1	Yes	9.56	.175	.594	15.64	.188	.522
0	1	No	0.58	.109	.820	4.90	.132	.514
1	0	Yes	17.07	.608	.089	5.92	.513	.190
1	0	No	1.51	.643	.048	4.22	.500	.188
1	1	Yes	1.52	.703	.822	31.00	.508	.502
1	1	No	0.27	.663	.719	11.25	.507	.506

Table 1. Bit patterns in data The values in the first three columns are used to separate the results into 8 rows. Columns 1 and 2 show the information bits of both operands; (for integers, the top bit; for floating points, the OR-ing of the bottom four bits of the mantissa). Columns 4 and 7 are the occurrence frequencies for the given operand bits and commutativity pattern, as a percentage of all executions of the FU type. Columns 5, 6, 8, and 9 display, for the specified operand, the probability of any single bit being high.

bit(operand): the information bit of the operand.

case($\mathbf{I}_j$): the concatenation of $bit(OP1(I_j))$ with $bit(OP2(I_j))$. $case$ classifies the instructions into four possible tuples (00, 01, 10, 11).

least: the $case$ with the lowest frequency, as found by examining Table 1 for all four cases, where the commutative and non-commutative rows are combined into one value.

vector: the concatenation of (case(I_1), case(I_2), ..., case($I_{Num(I)}$)). The size of $vector$ is 2×Num(M). If Num(I) < Num(M), the remaining bit pairs of $vector$ are set to the $least$ case.

The insight behind an approach of just using present inputs without considering the previous ones is that, by distributing the various instruction cases across the available modules, subsequent instructions to that module are likely to belong to the same case, without needing to check the previous values. For example, if we consider a machine where Num(M)=4, and with an equal probability of each of the four cases (00, 01, 10, and 11), it is logical to assign each of the cases to a separate module. In cycles when no more than one instruction of each case is present, this strategy will place them perfectly, even without checking previous values. When multiple instructions of the same case are present, however, the assignment will be non-ideal.

The algorithm is implemented as follows. During each cycle, $vector$ is used as the input address to a look up table, or LUT. The output of that table encodes the assignment of the operations to modules of the given FU. Therefore, the LUT contains the assignment strategy. Although the algorithm is conceptually visualized as using an LUT, the actually implemented circuit may use combinational logic,

Num(I) =	1	2	3	4
IALU	40.3%	36.2%	19.4%	4.2%
FPAU	90.2%	9.2%	0.5%	0.1%

Table 2. Frequency that the functional unit uses a particular number of modules for a 4-way machine with 4 IALUs and 4 FPAUs. There is no Num(I) = 0 column because we only consider cycles which use at least one module – the other case being unimportant to power consumption within a module (ignoring leakage).

a ROM, or another method.

The contents of the LUT are determined as follows. The information from Table 1 – along with new information from Table 2, which lists the probabilities of multiple instructions executing on the given FU type – is used to compute the probabilities of different input patterns. For instance, in the IALU, case 00 is by far the most common ($40.11\% + 29.38\% = 69.49\%$), so we assign three of the modules as being likely to contain case 00, and we use the fourth module for all three other cases (our test machine has 4 modules). For floating point, case 11 is the most common ($31.00\% + 11.25\% = 42.25\%$), but because it is unlikely that two modules will be needed at once (see Table 2), the best strategy is to first attempt to assign a unique case to each module.

Whenever the number of instructions of a particular case exceeds the number of modules reserved for that case, then it is necessary to place some of them in non-ideal modules. These overflow situations are dealt with in the order of their probability of occurring. The strategy for making non-ideal assignments is to greedily choose the module that is likely to incur the smallest cost.

4.4 Operand Swapping

We also propose a method for swapping operands without considering the previous inputs; the intuition of our method is as follows. The most power-consuming integer computations are those where the information bits for both operands fail to match the bits from the previous operands to that FU. There are four ways that this may occur: case 00 follows case 11, case 11 follows case 00, case 10 follows case 01, or case 01 follows case 10. In the last two of these, swapping the operands converts a worst-case situation into a best-case situation, assuming that the operation is commutative.

Therefore, we propose always swapping the operands for one of these cases. To minimize mismatches, the case to swap from should be the one that has the lower frequency of non-commutative instructions. Only non-commutative instructions are considered, because these are the ones that will not be flipped, and will therefore cause mismatches. Table 1 shows that for the IALU, the 4_{th} row has a lower frequency than the 6_{th}; for the FPAU, the 6_{th} row is the smaller. Therefore, case 01 instructions will be swapped for the IALU, and case 10 instructions for the FPAU.

Compiler-based swapping An alternative strategy for swapping operands is to perform it in software, by physically changing the machine instructions in the binary executable, using profiling. This approach is not particularly novel, but is studied so as to avoid the hardware cost of dynamic operand swapping. It is also instructional, in that our results show that the benefit of our hardware method is fairly independent of this transformation.

Compiler-based swapping has three advantages over hardware swapping. First, it avoids the overhead of hardware swapping. In fact, it offers some power improvement even when our hardware mechanism is not implemented. Second, the compiler can afford to count the full number of high bits in the operands, rather than approximating them by single bit values. For example, a "1+511" and a "511+1" operation both look like a case 00 to our hardware method. A compile-time method, however, can recognize the difference and swap when beneficial. Third, certain operations are commutable by the compiler but not by the hardware. An example is the "$>$" operation, which can become the $\leq$ operation when the operands are swapped. The compiler is able to change the opcode, but the current hardware strategy cannot.

But the compiler approach also has three main disadvantages. First, each instruction's operands are either always swapped or always not. The decision is made based on the $average$ number of high bits for the two operands. In contrast, the hardware technique captures the dynamic behavior of the instruction. Hardware can choose to swap an instruction's operands on one cycle, and not to swap them on a later cycle. Second, since the program must be profiled, performance will vary somewhat for different input patterns. Third, some instructions are not commutative in software. One example is the immediate add. While this operation is commutative, there is no way to specify its operand ordering in the machine language – the immediate value is always taken to be the second.

Since both hardware and compiler swapping have their advantages, the best results are achieved with both.

5 Practical considerations

We now consider the costs of our method. The potential costs are (1) increased power and area due to additional buses and logic, and (2) increased cycle time due to more complex routing logic. In fact, additional buses are not needed, and the increase in logic is not large.

In appraising the costs of the method, it is helpful to review how FUs are currently assigned by a superscalar processor. The most common methods are loosely based upon Tomasulo's algorithm [8], where each FU type has its own $reservation\ station$, **RS**. On each cycle, those instructions whose operands become ready are removed from RS and assigned to the FUs on a first-come-first-serve basis. Figure

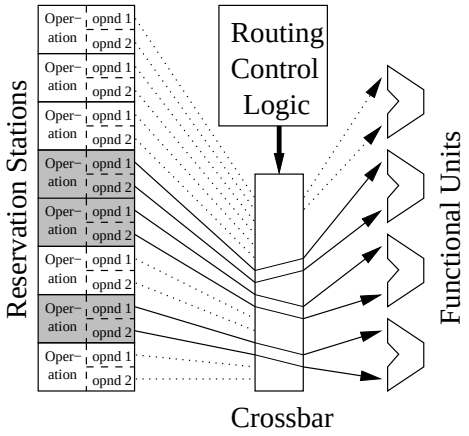

Figure 3. Typical Tomasulo hardware On a particular cycle, 3 operations are shaded to show they are ready to execute. Dotted lines indicate inactive buses, while solid lines represent active ones. Operations indicate their readiness to the routing control logic, which in turn schedules them, in order, to the FUs. The key observation is that the crossbar implies that no new buses will be needed by our method.

3 shows the basic hardware used in Tomasulo's algorithm. This algorithm allows instructions to execute out of order, so it requires routing logic and a crossbar. Since, on existing machines, most stations must already be able to map to any module, we do not need to add new wires.

It is unavoidable, however, that our method increases the complexity of the routing control logic. We replace the existing simple routing logic with the not-as-simple LUT described above. To make the LUT smaller and faster, we propose reducing the size of the vector. Since Num(I) $<$ Num(M) for most cycles, it is reasonable to consider a smaller vector that may not always contain all of the instructions. Reducing the size of the vector makes the proposed hardware modification faster and therefore more attractive. In the results we show that a 4-bit vector yields good performance. With a 4-bit vector, our fully implemented algorithm for the IALU, on a machine with 8 entries in its RS, requires 58 small logic gates and 6 logic levels. With 32 entries, 130 gates and 8 levels are needed. This is a negligible fraction of the many thousands of gates present in the IALU. Therefore, the power and delay introduced are small.

A third issue is that the crossbar currently used in Tomasulo's algorithm does not allow operand swapping. To perform operand swapping, it would be necessary to include additional wiring after the crossbar of Figure 3. This makes the compiler-based swapping methods more attractive.

6 Experimental Results

We have implemented the methodology described in section 4 using the *sim-outorder* simulator from the SimpleScalar 2.0 suite [5], with the default configuration of 4 IALUs, 4 FPAUs, 1 integer multiplier, and 1 floating point

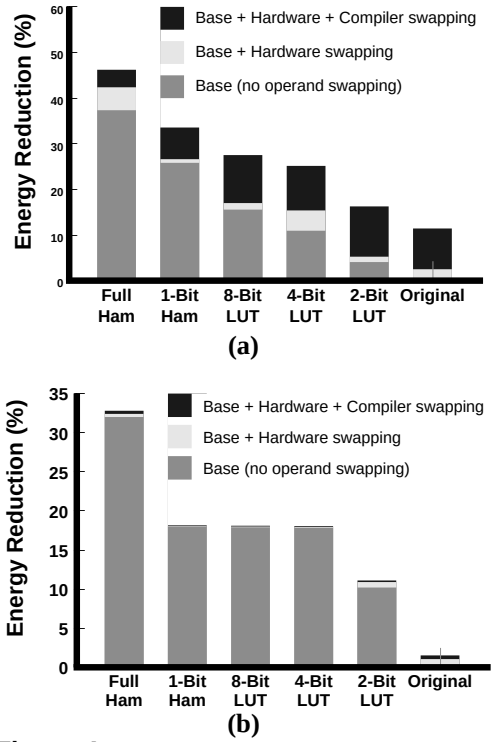

Figure 4. Results for the (a) IALU and (b) FPAU.

multiplier. SimpleScalar simulates a MIPS-like machine, with 32-bit integer registers and 64-bit floating point registers. The integer benchmarks used are: 88ksim, ijpeg, li, go, compress, cc1, and perl. The floating point benchmarks are: apsi, applu, hydro2d, wave5, swim, mgrid, turb3d, and fpppp. The benchmarks are from SPEC 95 [7], and were run to completion on large input files.

Figure 4 displays the power reduction of different schemes, as a fraction of the total power consumption of the FU type under consideration. Each bar is a stack of three values, so as to show the effect of operand swapping on the switching activity (which loosely approximates energy). Full Ham (section 4.1) and 1-bit Ham (section 4.2) are cost-prohibitive, but are included for comparison. Full Ham identifies the maximum possible improvement. 1-bit Ham is an upper bound on the improvement possible solely through the information bits. The 8 bit LUT represents the approach of section 4.3. The 4-bit and 2-bit LUTs represent shortened vectors, as considered in section 5. The Original column represents a first-come-first-serve assignment strategy. The gain for Original is not zero since swapping benefits it as well.

Figure 4 provides five insights. First, a 4-bit LUT is recommended, because its power savings are comparable to the upper bound (1-bit Ham), while being simple to implement. From Figure 4, the improvement for the 4-bit LUT with hardware swapping is 18% for the FPAU and 17% for the IALU. With compiler swapping, it is 26%, for the IALU.

	Integer			Floating Point		
Case	Freq (%)	OP1 prob	OP2 prob	Freq (%)	OP1 prob	OP2 prob
00	93.79	0.116	0.056	20.12	0.139	0.095
01	1.07	0.055	0.956	15.52	0.160	0.511
10	2.76	0.838	0.076	21.29	0.527	0.090
11	2.38	0.71	0.909	43.07	0.274	0.271

Table 3. Bit patterns in multiplication data (Multiplier power is related to how many 01 cases can become 10.)

If no swapping is provided by the hardware or the compiler, the still-rather-simple 8-bit LUT yields a very similar result to the 4-bit LUT with hardware swapping. Second, Figure 4 shows that the FPAU does not need operand swapping, due to differences between integers and floats. For integers, the majority of bits are usually the same as the information bit; for floating points, only an information bit of 0 has this property. Thus, for the IALU, a case 01 after a 10 causes most bits to switch; where as a case 01 after another 01 switches few bits. In contrast, for the FPAU, a case 01 after a 10 switches $\frac{1}{2}$ of the bits; where as a case 01 after another 01 still switches $\frac{1}{4}$ of the bits. Third, the FPAU is less sensitive to the size of the vector, because the floating point unit is less heavily loaded (Table 2). Fourth, profile-based swapping is more effective that hardware-based swapping, because the swap decision is based on the entire value, as opposed to the information bits alone. In fact, "Base + Compiler Swapping" (not shown) is nearly as effective as "Base + Hardware + Compiler". Fifth, implementing compiler swapping does not reduces the additional benefit of our hardware approach. Rather, the benefit of compiler swapping is slightly higher with an 8-bit LUT than it is for the original processor that has no hardware modifications.

Table 3 displays the bit patterns for multiplication data. This table, shows that 15.5% of floating point multiplications can be swapped from case 01 to case 10, certainly resulting in some additional power savings – though not quantifiable since we have no power model for multiplication.

7 Conclusions

We present a method for dynamically assigning operations to functional units, so as to reduce power consumption for those units with duplicated modules. We also examined operand swapping, both in conjunction with our assignment algorithm, and on its own merits for non-duplicated functional units like the multiplier. Our results show that these approaches can reduce 17% of the IALU and 18% of the FPAU switching, with only a small hardware change. Compiler swapping increases the IALU gain to 26%.

References

[1] M. Alidina, J. Monteiro, S. Devadas, and M. Papaefthymiou. Precomputation-Based Sequential Logic Optimization for Low Power. *IEEE Transactions on VLSI Systems*, 2(4):426–436, April 1994.

[2] L. Benini and G. D. Micheli. Transformation and Synthesis of FSMs for Low Power Gated Clock Implementation. *IEEE Transactions on Computer-Aided Design*, 15(6):630–643, June 1996.

[3] D. Brooks and M. Martonosi. Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performance. In *Proc of the 5th Int'l Symp on High Performance Computer Architecture (HPCA)*, pages 13–22, January 1999.

[4] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *Proceedings of the 27th International Symposium on Computer Architecture (ISCA)*, Vancouver, British Columbia, June 2000.

[5] D. Burger and T. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report TR 1342, University of Wisconsin, Madison, WI, June 1997.

[6] J.-M. Chang and M. Pedram. Module Assignment for Low Power. In *Proc of the European Conference on Design Automation (EDAC)*, pages 376–381, September 1996.

[7] S. P. E. Corporation. The SPEC benchmark suites. *http://www.spec.org/*.

[8] J. Hennessy and D. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann, Palo Alto, CA, second edition, 1996.

[9] M. Johnson, D. Somasekhar, and K. Roy. Leakage Control with Efficient Use of Transistor Stacks in Single Threshold CMOS. In *Design Automation Conference*, pages 442–445, June 1999.

[10] B. Klass, D. E. Thomas, H. Schmit, and D. E. Nagle. Modeling Inter-Instruction Energy Effects in a Digital Signal Processor. In *Power-Driven Microarchitecture Workshop, in conjunction with Int'l Symposium on Computer Architecture*, June 1998.

[11] L. Kruse, E. Schmidt, G. Jochenshar, and W. Nebel. Lower and Upper Bounds on the Switching Avtivity in Scheduling Data Flow Graphs. In *Proc of the ACM Int'l Symp on Low Power Design*, pages 115–120, August 1999.

[12] T.-C. Lee, V. Tiwari, S. Malik, and M.Fujita. Power Analysis and Minimization Techniques for Embedded DSP Software. *IEEE Transactions on VLSI Systems*, Mar. 1997.

[13] R. Marculescu, D. Marculescu, and M. Pedram. Sequence compaction for power estimation: Theory and practice. *IEEE Transactions on Computer Aided Design*, 18(7):973–993, 1999.

[14] J. Mermet and W. Nebel. *Low Power Design in Deep Submicron Electronics*. Kluwer Academic Publishers, Norwell, MA, 1997.

[15] M. Pedram. Power Minimization in IC Design: Principles and Applications. *ACM Transactions on Design Automation of Electronic Systems*, 1(1):1–54, January 1996.

[16] V. Tiwari, S. Malik, and P. Ashar. Guarded Evaluation: Pushing Power Management to Logic Synthesis/Design. In *Proceedings of the ACM/IEEE International Symposium on Low Power Design*, pages 139–142, April 1994.