A Detailed Study of Hardware Techniques That Dynamically Exploit Frequent Operands to Reduce Power Consumption in Integer Function Units *

Kaushal R. Gandhi and Nihar R. Mahapatra
Department of Computer Science & Engineering
University at Buffalo, The State University of New York
Buffalo, NY 14260-2000, USA
Email: {krgandhi, mahapatr}@cse.buffalo.edu

Abstract *The use of multiple pipelines in superscalar processors and increasing hardware support for intelligent compilers have resulted in processors becoming computationally intensive. There has been an increase in the number of function units, which are wider and operate at higher speeds. A high percentage of integer instructions executed in standard benchmarks reveal a high usage of integer function units.*

In this paper, we analyze three different techniques, namely, memoing (caching results that can be reused), narrow-width operand exploitation (limiting computation to low order bytes), and byte encoding (computation performed over significant bytes), that dynamically exploit operands to lower power consumption in integer function units. Each technique proposes avoiding redundant computation in a function unit and providing the output or a part of the output via alternate means. We analyze and compare these techniques in terms of power savings, area and delay overheads, and their effectiveness when applied to different integer function units. Previously, estimates of power savings based on switching activity were reported. Our implementation of integer function units (CLA, array multiplier, comparator, etc.) at the VLSI layout level and analysis using standard integer benchmarks from the SPEC CPU2000 suite provide realistic power savings, area, and delay overheads.

1 Introduction

VLSI research and development and advances in computer architecture have resulted in processors becoming computationally intensive. The use of multiple pipelines in superscalar processors and increasing hardware support for intelligent and efficient compilers has led to higher performance in processors, but at the same time increased the computation demanded from functional units. The increase in the performance of processors has led to greater power consumption and thrust low power to the forefront as an important design parameter. A high percentage of integer instructions executed in standard benchmarks reveal a high usage of integer function units. Hence the necessity of designing power-efficient integer function units.

Power dissipation in CMOS circuits falls into two categories: dynamic and static. Static power dissipation is due to leakage current. Dynamic power dissipation is due to: (1) short-circuit current during an output transition that is caused by a DC path between the supply rails and the ground line and (2) the charging and dis-charging of capacitive loads during logic changes [9]. Techniques to reduce static power dissipation aim to turn off functional units during idle cycles, e.g., through power gating and use of dual threshold voltages [12, 10, 14, 18]. At the circuit level, voltage scaling, transistor sizing, transformations, and innovative circuit design techniques have been proposed to reduce power consumption [5, 6, 17]. At a higher level of abstraction, compilers aim to reduce power by producing energy-efficient code. Most of these techniques do not exploit operand values dynamically. Only recently, researchers have looked into the possibility of exploiting operand values to reduce power consumption. The observation that many computations on operands are redundant for typical applications and the fact that operands that are inputs to functional units are not completely random provide opportunities for operand exploitation to reduce dynamic power dissipation. Three such techniques are: *memoization* (caching frequent operand sets and results for reuse), *narrow-width operand* (operands that can be represented in fewer bits) *exploitation*, and *byte encoding* (computation skipped over insignificant bytes). In this paper, we comprehensively analyze and compare these techniques in terms of their power savings, area and delay overheads, and effectiveness when applied to different functional units.

The organization of the remainder of the paper is as follows. In Section 2, we describe the above three techniques in detail and in Section 3, we analyze them. Following that, in Section 4, we describe the simulation setup. In Section 5, we analyze the results obtained and compare them. Finally, we conclude and propose future work in Section 6.

*This research was supported by US National Science Foundation Grant # 0102830.

2 Operand value exploitation for low power

We analyze memoization, narrow-width operand exploitation technique, and byte encoding technique [2, 3, 7, 8]. These techniques share some common features: (1) hardware schemes that exploit operands dynamically, (2) detection of frequent operands, (3) reduction of redundant computation by avoiding computation on frequent operands and providing the output via alternate means, and (4) power savings achieved by reduction in the switching activity of a function unit. Memoization exploits the temporal locality of operand sets [2], while the other two schemes exploit the magnitude of operand values at different levels of granularity [3, 7, 8].

2.1 Memoization

Reusing results that were previously calculated, instead of computing them again, could potentially lead to significant power savings in function units. The frequent reuse of data in many applications supports this claim. *Memoization*, a technique that stores results for reuse, has been used previously for improving performance [13]. Azam et al. use this concept to reduce power dynamically [2]. They proposed the use of an execution cache per function unit that can be used as a buffer to capture the temporal locality of operand sets in instruction and data streams. Each entry of the execution cache consists of a computed value and the operand set for which it was computed. The execution cache is either a direct-mapped cache or an s-way set associative cache with LRU replacement policy. An execution cache stage is introduced in the processor pipeline, which is placed between the instruction decode stage and the execute stage, to determine whether an operand set is being reused.

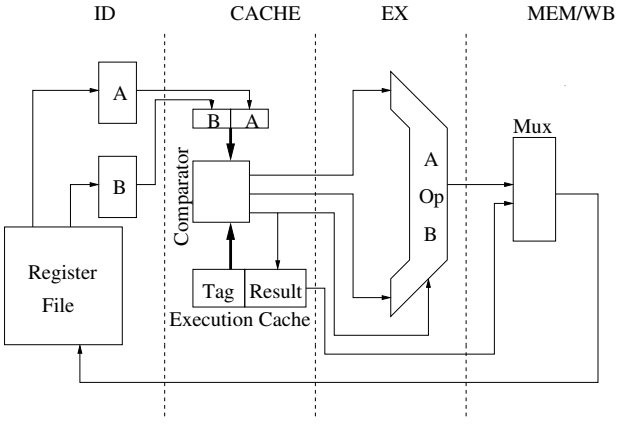

Figure 1: Memoization: Simple bypass via operand tagged caching [2].

During the execution cache stage, certain bits of an operand set ready to be issued to a function unit are used to index into the execution cache. The remaining bits of the operand set are compared to the corresponding bits of the operand set stored in the cache entry. The matching of these two operand sets implies that computation by the function unit can be bypassed as the result is already known and can be reused. If the operand sets do not match, the function unit performs computation over these operands and the result is stored in the execution cache. Using this technique, operand sets are exploited based on their their frequent occurrence for a particular function. The design is illustrated in Fig. 1 [2].

2.2 Narrow-width operand exploitation

Although modern microprocessors support 64-bit addresses and operations, most ALU operands have much smaller magnitudes that can be represented using 16, 8, or fewer bits. By dynamically shutting off that part of a function unit which operates over the high order bytes of frequent operands and sign extension of the result computed for the low order bytes results in significant reduction of switching activity. This approach to operate only a part of the original circuit by latching of certain inputs is termed *guarded evaluation* [16]. We study two implementations of this technique [3, 7].

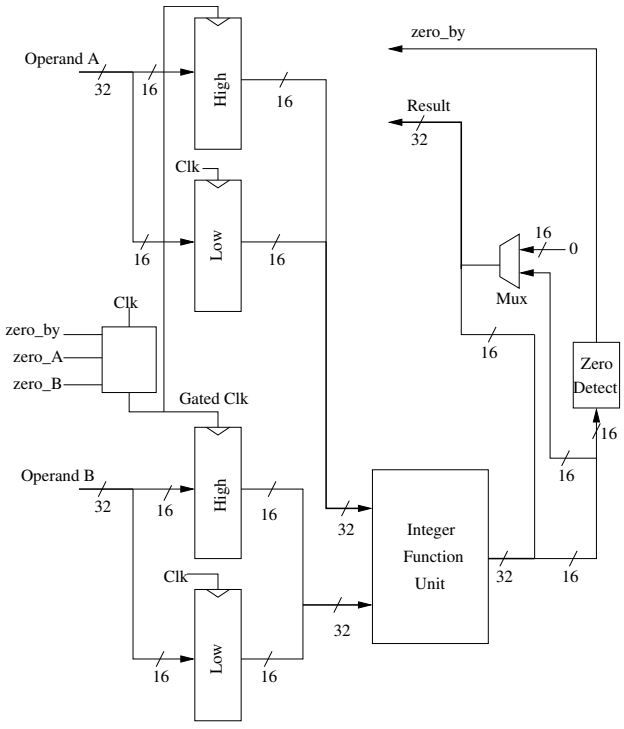

Figure 2: Exploiting narrow-width operands [3].

Fig. 2 illustrates the architecture proposed by Brooks et al. [3]. By disabling value changes in the high order part of a function unit when both operands are narrow-width operands, they are able to reduce switching activity and thus reduce the dynamic power dissipation. This disabling of the high order part of a function unit is achieved by selectively clocking the latches that feed the high order bits of the operands to the function unit. The low order bits are always latched. The selective latching of the high order part of the operand depends upon whether both the operands have leading ones/zeros in their high order bits. The high order bits are latched selectively based on the clock and a signal called *zero/one*. This signal, generated using a *zero/one* detection logic, is stored along with an operand in the register when the operand was either computed as a result or when it was loaded from the cache. The result for the high order part of the function unit is supplied by hardwired values through a multiplexor.

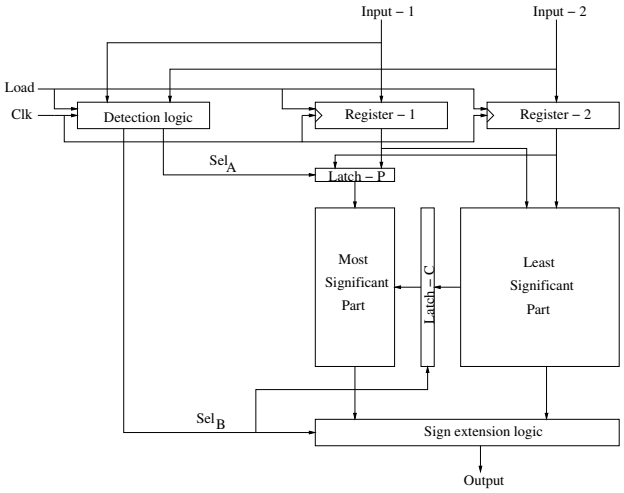

Figure 3: Exploiting narrow-width operands by partially guarded computation [7].

Fig. 3 illustrates the technique employed by Choi et al. to reduce dynamic power dissipation [7]. The high order bits of the inputs are selectively passed on to that part of the function unit that operates on them through latches. The latches are controlled by a signal produced by a detection logic which detects whether the high order bits of both the operands are either zeros/ones. The high order bits of the result are computed via a sign extension logic when both the operands are narrow-width operands.

2.3 Byte encoding

This scheme employs the same method of preventing the propagation of operand bits as described in the previous section, however, at a finer granularity [8]. Canal et al. propose exploitation of operands at various stages in a processor pipeline. We only discuss the application to integer function units here. Operands are coded at a byte level and this choice of dividing an operand into bytes is arbitrary. The least significant byte is always considered to be significant and a function unit always operates over this byte of the operand sets. One scheme of encoding is to add extra bits that represent the number of high order bytes that are merely sign extensions. Another scheme of encoding is to represent each high order byte by one bit. A byte is encoded as 1 if it is the sign extension of the adjacent byte and 0, otherwise. The overhead in terms of bits appended to each operand is higher in the latter scheme, but it exploits more cases of frequent valued operands than the former. The extension bits are computed using simple detection logic when an operand is loaded from the instruction or data cache and when a result is computed. The authors describe three different implementations for a 32-bit wide processor, namely, byte-serial implementation, in which the data path is one-byte wide, semi-parallel implementation, in which the data path is two-bytes wide, and fully parallel implementation, in which the data path is four-bytes wide.

3 Analysis of techniques

In this Section, we analyze the advantages of each scheme and also discuss some of the shortcomings. Each of these schemes involves detection of frequent operand values, a method to bypass or limit computation on such frequent-valued operands, and a method to provide the whole output or a part of it by alternate means.

3.1 Frequent-valued operand detection

Integer function units that have long latencies and cause extensive switching are targets for memoization. The power savings would be significant in such function units if they were to operate on the same operand set frequently. All operand sets that have temporal locality can potentially be exploited. Frequent valued operands as exploited by this scheme are operand sets that have temporal locality with respect to a particular function unit. The implementation by Azam et al. uses an execution cache per function unit [2]. Hence the area overhead is significant. In order to detect the temporal locality of an operand set, an operand set ready to be issued to a function unit is compared to an operand set stored in an entry of the execution cache. This involves a comparison each time an integer function unit is supposed to be used. This is an expensive overhead in terms of the power dissipated

while comparing operand sets. Thus a limitation of this technique is the kind of function that can be bypassed to achieve significant power savings. As the detection of frequent operands involves comparison of operands, it is most useful to apply this technique to functions that consume more power than addition, as comparison of operands, during the cache stage, incurs power consumption similar to that of addition.

Narrow-width operand exploitation targets all types of integer function units in the processor pipeline and results presented previously show significant power savings [3]. Their implementation exploits those narrow-width operands, whose high order bits are the same in both operands. They assume some processors have a zero-check for operands before they are loaded from memory. This may not be true in general and it may be required to detect narrow-width operands explicitly. Their implementation detects whether a result computed by an integer function unit can be classified as a narrow-width operand. Although the area and delay overhead is reduced, if the detection is carried out in the above manner, they lose out on exploiting narrow operands that are loaded from the instruction and data cache.

The implementation by Choi et al. exploits those narrow-width operands, whose high order bits are the same [7]. The detection of narrow operands is performed explicitly by a detection logic before operands are passed on as inputs to the integer function unit. Although this ensures exploitation of all narrow operand sets, there is a significant delay and area overhead due to the detection logic.

Byte encoding applies the concept of narrow-width operands at a finer granularity with regard to the operand value. A byte wise partitioning, although chosen arbitrarily, performs well and significant power savings are reported in terms of reduction in switching activity in the ALU. Frequent-valued operands are detected by a detection logic that encodes bytes of an operand when they are loaded from the instruction cache and the data cache.

3.2 Avoiding computation

Bypassing computation is achieved efficiently by memoization. If an operand set is found to be present in the execution cache, the inputs to the function unit remain unchanged and thus switching activity through the function unit is considerably reduced. The result, which is also stored along with the operand set in the execution cache, is forwarded as the output of the function unit. The area overhead is due to the storage of the operand set and the result in the execution cache. A shortcoming of the scheme as implemented by Azam et al. is that an execution cache is required for each function unit [2]. This is an expensive overhead in terms of area and power consumed. Sharing an execution cache across multiple function units by using multiple ports would be a possible solution to this. But this would cause a severe slowdown. Adding an extra stage to the processor pipeline is also an overhead, as the overall latency increases.

In the narrow-width operand exploitation technique, the partition point for a function unit is selected based on the number of operand sets that can be exploited and the size of the function that will be active during a narrow-width operation so as to achieve significant power savings [3]. Although, this is a reasonable way to determine the partition point, it does not ensure maximum power savings. By selectively latching only the high order bytes of narrow operands, switching activity in the function unit that operates over the high order bytes of the operands is reduced. They, however, do not discuss any integer function unit in detail with regard to its structure (for example, CLA has a tree structure, ripple carry adder has a linear structure, array multiplier has an array structure, etc.). The result for the high order bits is provided through a multiplexor for frequent operands.

The partition point as determined by Choi et al. aims to maximize power savings from estimations of power savings [7]. However, they assume that power dissipation scales linearly with the size of an integer function unit. This may not always be true for different structures of the same function unit. The output for the high order bits is provided by sign extending the low order part of the result computed for the narrow operands. They use a sign extension logic to sign extend the narrow result. The high order bits of the result are provided through a multiplexor and the selection is based on a select signal produced by the detection logic. This adds to the delay and area overhead.

The byte-serial implementation as described by Canal et al. enables independent operation of an integer function unit over individual bytes of operands [8]. Thus the byte encoding technique exploits more cases of frequent valued operands as compared to the narrow-width exploitation technique. But, several dependencies between operand bytes have to be checked for to determine whether computation over the operand byte can actually be avoided. The result for an operand byte is provided by either one of the operand bytes or multiplexing hard wired values. The dependencies between operand bytes of the operand set are resolved using simple logic that varies for different function units. This also contributes to the area

and delay overheads in addition to that incurred by latching of operand bytes and multiplexing of output bits.

3.3 Estimation of power savings

The energy savings reported for the *memoization* technique are based on the assumption that the 2n-bit comparator used to detect frequent-valued operand sets consume 75% of the energy of the n-bit carry-save adders in the multiplier and the energy required to drive the word and bit lines of the 2n-bit tagged cache is approximately the same as that dissipated in an n-bit adder of the multiplier. This may report reasonable power savings at a high level of abstraction, but it does not provide realistic power savings.

The narrow-width operand exploitation technique as implemented by Brooks et al. assumes that the power dissipation of function units scales linearly at the bit level. This may not necessarily be true as function units are implemented in various structures and thus the power savings reported by them, though reasonably accurate at an architectural level, may not be very realistic.

Choi et al. implemented their scheme on the array multiplier and report power savings at the VLSI level. However, the operands for which the array multiplier was tested by Choi et al. were application-specific [7]. Such operands may not arise in general purpose processors. Also, they do not implement their technique on various integer function units for comparison.

Canal et al. estimate power savings based on activity reduction, which is based on the extension bits [8]. This does not accurately capture true power savings. They also do not explain how such a design could be extended to multiplication and division, which are more complicated as compared to addition, comparison, and bit-wise logical operations. In the parallel compressed pipeline [8], they do not explain how dependencies over different bytes of the ALU are resolved in the same cycle of operation. The detection of dependencies would require more logic, which would be different for different functions as the dependencies would vary, and thus might not be shared across different integer function units.

4 Simulations

In this section, we first describe the simulator used to obtain operand traces and the benchmarks used in our simulations. Next, we describe the simulations that were performed at the VLSI level to obtain our results.

4.1 Simulation for operand analysis

We use a version of SimpleScalar's sim-outorder to collect operands [4]. SimpleScalar provides a simulation environment for out-of-order processors with speculative execution. We collected operands for 10 million committed instructions for 5 integer (INT) benchmarks from the SPEC CPU2000 suite after skipping through a warm-up window of 500 million instructions for most benchmarks [15]. We collect 32-bit operands just before they are issued to various integer function units. The configuration of the simulator we used is given in Table 1.

Benchmarks	gzip, bzip2, gcc, mcf, gap
Compiler	gcc 2.7.2.3 for the PISA version of SimpleScalar
Inputs	Reference inputs for each benchmark
Instruction Count	10 million after the warmup window for each benchmark
Processor core	RUU size 16 instructions, LSQ size 8, Fetch queue size 4 inst/cycle, Fetch width 4 inst/cycle, Decode width 4 inst/cycle, Issue width 4 inst/cycle, Functional units 4 integer ALUs, 1 integer multiply/divide unit
Branch prediction	2048-entry bimodal predictor
L1 instruction cache	512K, 32-byte blocks, direct-mapped
L1 data cache	128K, 32-byte blocks, 4-way associative
L2 cache	Unified, 1024K, 64-byte blocks, 4-way associative

Table 1: Simulation setup to collect operands.

4.2 Simulation of integer function units

We designed and simulated the integer function units using Cadence simulation tools and Spectre simulator. We used 0.18 micron technology to design the circuits and designed them as static CMOS circuits. This type of circuit level simulation had not been done previously to determine power savings in integer function units for most of the schemes. The integer function units we designed are carry look-ahead tree adder, array-multiplier, comparator and bit-wise logical function units (AND, OR, XOR, NOR). This choice of function units covers various functions implemented in various structures. We chose the carry look-ahead tree adder and comparator to represent the tree structure, the array multiplier to represent an array structure, and the bit-wise logical function units that are implemented in a linear fashion. Our aim was to see the effectiveness of different techniques when applied to different functions implemented in different structures.

We simulated each circuit for a conventional design of the functional unit for a set of 500 inputs per bench-

mark to obtain the energy consumption in each function unit for comparison. We then simulated function units with same set of operands after modifying the design according to each scheme. Ideally, a choice for the size of the set of inputs would be as large as possible, but this would lead to excessively long simulation times.

For the memoization technique, we determined the set of operands that would be bypassed and the set of operands that would not be bypassed for a direct-mapped cache. We simulated the integer function units for those operand sets that would not be bypassed owing to their availability in the execution cache. The execution cache that we used had 64 entries and was a direct-mapped cache. We also performed simulations for the comparisons made between operand sets to determine whether an operand set was present in the execution cache. We assumed that the energy required to drive the word and bit lines of the 2n-bit tagged cache is the same as the average energy dissipated in an n-bit adder of the multiplier based on the assumption made by Azam et al [2]. We do not simulate the bit-wise logical function units as the energy consumed by the overheads would be greater than the savings that would be made if the computation were to be bypassed. This is due to the simplicity of the structure of bit-wise logical function units.

For the narrow-width operand exploitation technique, we determined the partition point in two ways. A partition point is determined so as to exploit the maximum number of operands for all the integer function units. We also determined the partition point by estimating the maximum power savings that are possible for all the integer function units [7]. We obtained partition points using both techniques and partitioned operands so that the lower part of the operand was 16 bits and 14 bits wide, respectively. In order to determine these partition points we assumed that power scales linearly at the bit-level for all the function units. We perform operand detection before the operands are issued to a function unit. The basic building blocks for the integer function units such as the full adder, carry save adder, and basic logic gates were designed as static CMOS circuits [11].

In order to implement the byte encoding technique, we simulate the byte serial implementation as described by Canal et al. for the adder, comparator, and the bit-wise logical operations [8]. Dependencies between bytes of the operands are resolved using simple logic, which varies for different operations. We extend the *byte encoding* technique to exploit operands in the array multiplier. We divide the array multiplier into four parts, so that a byte in one of the operands (A) is multiplied by all the bytes in the other operand (B). The presence of insignificant bytes in operand B is also exploited. If a byte in operand B consists of all zeros, a logical zero value is hardwired to the output of that block. We only exploit those bytes whose bits are all zero. The encoding of bytes for these operands is also modified. Each byte whose constituent bits are all zero is encoded as 1, and 0, otherwise.

5 Observations and analysis of simulation results

From our simulations at the VLSI level, we observe that the estimated power savings are highly optimistic. Actual power savings are smaller and vary for the various techniques. One of the reasons for this could be that while estimating power savings, leakage power is not taken into consideration. Another reason could be that the estimations are based on the switching activity that could potentially be reduced. This may not accurately indicate realistic power savings. Also, the assumption that power scales linearly at the bit-level may not always hold true for different function units.

Our simulations show that memoization is not very effective when applied to different integer functions. It is effective only for long latency and complicated functions. Although the presence of operand sets that are reused by the same function units is abundant (Fig. 4), exploiting these operand sets for simple functions such as addition, comparison, and bit-wise logical operations does not prove to be beneficial. We only present plots for addition, comparison, and multiplication as the overheads would be too large to exploit bit-wise operations by this technique. Memoization provides power savings only for multiplication (Fig. 7). The area overhead is huge, about 65%. The delay overhead is due to the additional stage in the processor pipeline.

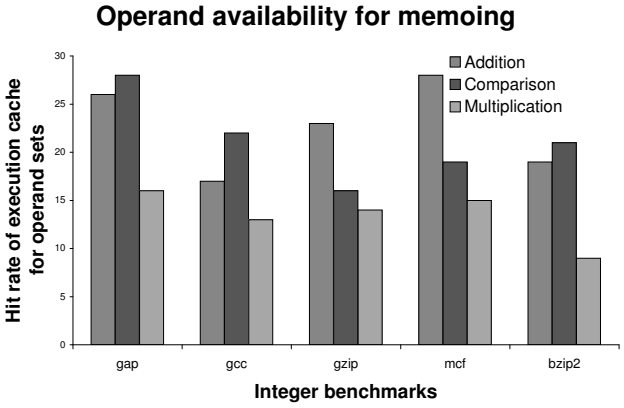

Figure 4: Availability of operands that can be exploited by memoization.

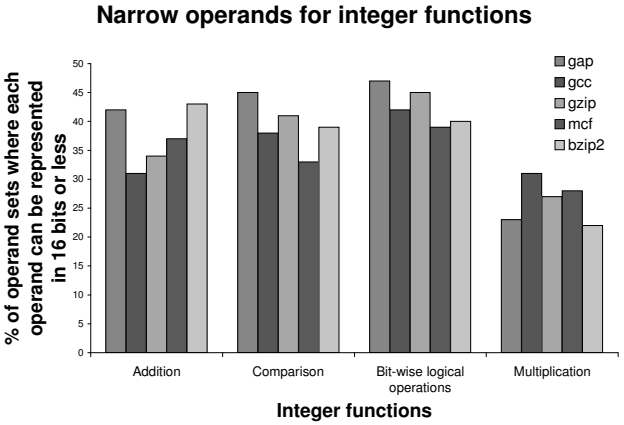

Figure 5: Availability of narrow-width operands.

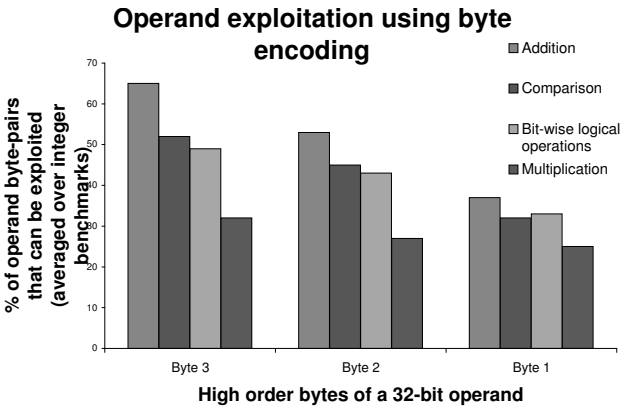

Figure 6: Availability of operands that can be exploited using byte-encoding.

The abundance of narrow-width operand sets for addition, comparison, and bit-wise logical operations is evident from Fig. 5. We also observe that narrow operand sets for multiplication are not as many. A possible explanation for this is that results from multiplication are reused as operands for multiplication frequently. In contrast, our simulations for actual power savings for the narrow-width operand exploitation technique as illustrated in Fig. 8 show that the array multiplier gives the greatest amount of power savings. This can be attributed to the fact that the array multiplier has the largest and most complicated structure as compared to the other function units under consideration. Power savings for the adder and the comparator are similar for this technique. The power savings for the bit-wise logical operations are not very significant, and this is probably due to the simplicity of the structure of these function units. We also observe that the power savings obtained by partitioning along the partition point obtained so as to maximize power savings proves to be more effective as compared to that obtained to exploit a large number of operands. The delay overhead is similar for all the operations except for multiplication. The same holds true for the area overhead.

We analyzed operand availability for exploitation by byte encoding at a byte level, since our implementation is of a byte-serial nature. Since it is possible to operate over bytes of an operand independently, we study the percentage of operands available for exploitation in individual bytes of the operand (Fig. 6). We observe that the opportunity to exploit operands is greatest in the most significant byte and decreases as we move toward the least significant byte. The power savings using byte encoding is again greatest for multiplication(Fig. 9). The area overhead is much greater compared to the narrow-width operand exploitation technique, as more logic is required to detect dependencies between operands. Also, additional logic is required to produce the output for frequent-valued operand bytes. The delay overhead is greater due to the excessive logic used.

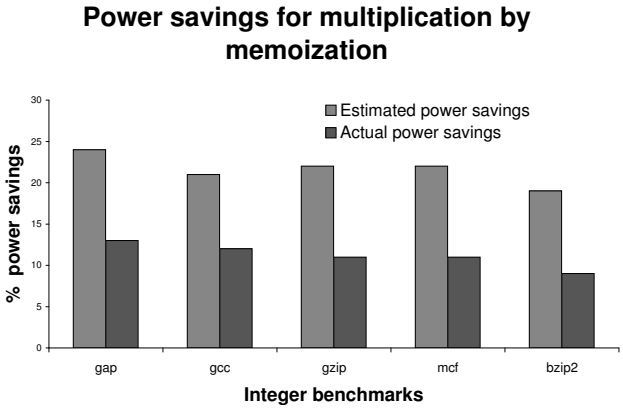

Figure 7: Power savings using memoization.

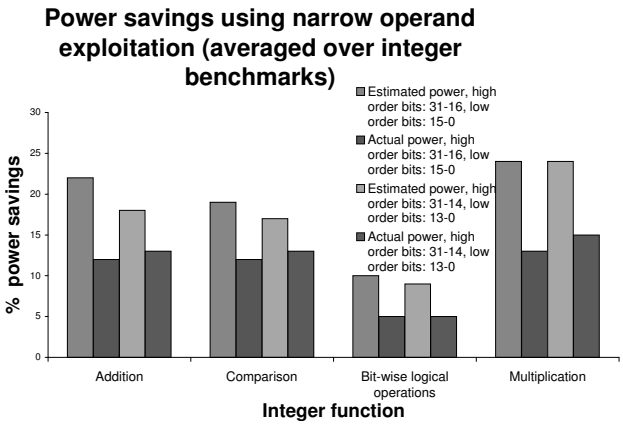

Figure 8: Power savings using narrow-width operand exploitation.

Our results show that the byte encoding technique is the most efficient technique to reduce power consumption. This is because byte encoding exploits the

Operand exploitation technique	Integer function unit	Area overhead	Delay overhead	Estimated power savings	Actual power savings
Memoization	CLA	-	-	-	-
	Comparator	-	-	-	-
	Bit-wise logical operations	-	-	-	-
	Array multiplier	65%	-	20.5%	11.2%
Narrow operand exploitation (16-bit)	CLA	22.5%	1.3%	22.2%	12.1%
	Comparator	22.2%	1.3%	19.4%	12.1%
	Bit-wise logical operations	21.9%	1.3%	10.3%	5.1%
	Array multiplier	27.4%	1.7%	24.1%	13.2%
Narrow operand exploitation (14-bit)	CLA	21.3%	1.3%	18.4%	13.4%
	Comparator	23.3%	1.3%	17.6%	13.3%
	Bit-wise logical operations	20.4%	1.3%	9.1%	5.3%
	Array multiplier	25.3%	1.7%	24.1%	14.9%
Byte encoding (byte-serial)	CLA	26.6%	2.2%	27.2%	14.2%
	Comparator	26.1%	2.1%	22.2%	14.2%
	Bit-wise logical operations	24.7%	2.2%	13.2%	6.1%
	Array multiplier	28.6%	2.4%	29%	16.4%

Table 2: Results averaged over integer benchmarks for different integer function units using various techniques to exploit frequent-valued operands.

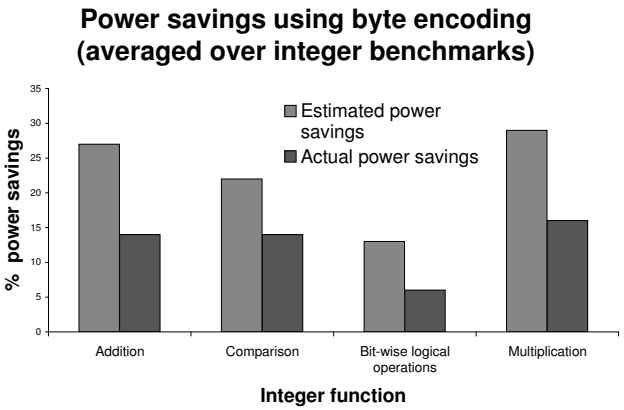

Figure 9: Power savings using byte-encoding.

most number of cases of frequent valued operands at a finer granularity. Table 2 tabulates the results. Our results show that power can be reduced by 10-15% for addition, comparison, and multiplication. The power savings for bit-wise logical operations are still lower.

6 Conclusions and future work

In this paper, we presented power savings for integer function units that were simulated at the VLSI level using various low power techniques that exploit frequent operands. We find from our results that the power estimates predicted by previous work are optimistic, whereas actual power savings are smaller. Another shortcoming to previous techniques is that they only target reduction of dynamic power dissipation. These techniques do not address the issue of leakage power, which becomes a significant part of the total power dissipated as design moves towards sub-micron technology. In order to gain considerable power savings, these techniques need to be applied more aggressively to design function units. We observe from our results that memoization, when applied to integer function units for general applications, does not perform too well. However, narrow-operand exploitation of operands and byte encoding prove to be effective. A better organized execution cache that is more efficient than the present implementation to buffer operand sets and their results could lead to greater power savings. Encoding of operand sets that are operated on repeatedly can also be considered to improve this technique. In order to exploit frequent-valued operands more effectively, a more refined partitioning of function units, based on operand statistics, that exploits more cases of frequent valued operands needs to be developed. An exhaustive enumeration of all the possible frequent valued operands that could be exploited by careful analysis of the structure of function units can also be considered for future work.

References

[1] M. Alidina, J. Monteiro, S. Devadas, A. Ghosh and M. Papaefthymiou. Precomputation-based sequential logic optimization for low power. In *IEEE Trans. Very Large Scale Integr. Syst.* 2, 4 (Dec. 1994), 426-436.

[2] M. Azam, P. Franzon and W. Liu. Low-power data processing by elimination of redundant computation. In *proceedings of the international symposium on low power electronics and design,* 1997.

[3] D. Brooks, M. Martonosi. Value-based clock gating and operation packing: dynamic strategies for improving processor power and performance. In *ACM Transactions on computer systems*, Vol. 18, 2. May, 2000.

[4] D. Burger, T. Austin. The simple scalar tool set. Tech. Rep. TR-1342. Computer Science Dept., Univ. of Wisconsin, Madison. 1997

[5] A. P. Chandrakasan, R. W. Brodersen. Minimizing Power Consumption in Digital CMOS Circuits. *Proceedings of the IEEE*, Vol. 83: 498-523, April 1995.

[6] A. P. Chandrakasan, M. Potomac, R. Mehra, J. Rabaey, R. W. Brodersen. Optimizing Power Using Transformations. In *IEEE Transactions on Computer-Aided Design*, 1995.

[7] J. Choi, J. Jeon, K. Choi. Power minimization of functional units by partially guarded computation. *ACM/IEEE International Symposium on Low Power Electronics and Design* (ISLPED), 2000.

[8] R. Canal, A. Gonzalez, J. Smith. Very low power pipelines using significance compression. *Proceedings of the 33rd Annual ACM/IEEE international symposium on Microarchitecture*, 2000.

[9] M. Pedram. Design technologies for Low Power VLSI. In *Encyclopedia of Computer Science and Technology*, Vol. 36, Marcel Dekker, Inc., 1997, pp. 73-96.

[10] M. D. Powell, S-H. Yang, B. Falsafi, K. Roy, and T. N. Vijaykumar. Gated Vdd: A Circuit Technique to Reduce Leakage in Deep-Submicron Cache Memories. In *ACM/IEEE International Symposium on Low Power Electronics and Design* (ISLPED), 2000.

[11] J. Rabaey. Digital integrated circuits, a design perspective. Prentice Hall, Inc. 1996.

[12] S. Rele, S. Pande, S. Onder, and R. Gupta. Optimization of Static Power Dissipation by Functional Units in Superscalar Processors. *International Conference on Compiler Construction*, LNCS 2304, Springer Verlag, pages 261-275, Grenoble, France, April 2002

[13] S. E. Richardson. Caching function results: faster arithmetic by avoiding unnecessary computation. Technical report, Sun Microsystems Laboratories, 1992.

[14] K. Roy. Leakage Power Reduction in Low-Voltage CMOS Design. In *IEEE International Conference on Circuits and Systems*, pages 167-173, 1998.

[15] K. Skadron , P. Ahuja , M. Martonosi , D. Clark. Branch Prediction, Instruction-Window Size, and Cache Size: Performance Trade-Offs and Simulation Techniques. In *IEEE Transactions on Computers*, v.48 n.11, p.1260-1281, November 1999

[16] V. Tiwari, S. Malik, P. Ashar. Guarded Evaluation: Pushing Power Management to Logic Synthesis/Design. *IEEE Trans. on CAD*, vol. 17, no. 10, Oct. 1998, pp. 1051-60.

[17] V. Tiwari , D. Singh , S. Rajgopal , G. Mehta , R. Patel , F. Baez. Reducing power in high-performance microprocessors. *Proceedings of the 35th annual conference on Design automation conference* p.732-737, June 15-19, 1998.

[18] Q. Wang and S. Vrudhula. Static Power Optimization of Deep Submicron CMOS Circuits for Dual V_T Technology. In *International Conference on Computer-Aided Design* (ICCAD), pages 490-496, 1998.