Journal of VLSI Signal Processing, , 1–18 (1996)

Instruction Level Power Analysis and Optimization of Software

VIVEK TIWARI, SHARAD MALIK, ANDREW WOLFE

vivek@ee.princeton.edu, sharad@ee.princeton.edu, awolfe@ee.princeton.edu

Department of Electrical Engineering, Princeton University, Princeton, NJ 08540

MIKE TIEN-CHIEN LEE

lee@fla.fujitsu.com

Fujitsu Labs of America Inc., 3350 Scott Blvd., Bldg. 34, Santa Clara, CA 95054

Abstract. *The increasing popularity of power constrained mobile computers and embedded computing applications drives the need for analyzing and optimizing power in all the components of a system. Software constitutes a major component of today's systems, and its role is projected to grow even further. Thus, an ever increasing portion of the functionality of today's systems is in the form of instructions, as opposed to gates. This motivates the need for analyzing power consumption from the point of view of instructions - something that traditional circuit and gate level power analysis tools are inadequate for. This paper describes an alternative, measurement based instruction level power analysis approach that provides an accurate and practical way of quantifying the power cost of software. This technique has been applied to three commercial, architecturally different processors. The salient results of these analyses are summarized. Instruction level analysis of a processor helps in the development of models for power consumption of software executing on that processor. The power models for the subject processors are described and interesting observations resulting from the comparison of these models are highlighted. The ability to evaluate software in terms of power consumption makes it feasible to search for low power implementations of given programs. In addition, it can guide the development of general tools and techniques for low power software. Several ideas in this regard as motivated by the power analysis of the subject processors are also described.*

1. Motivation

The increasing popularity of power constrained mobile computers and embedded computing applications drives the need for analyzing and optimizing power in all the components of a system. This has forced an examination of the power consumption characteristics of all modules - ranging from disk-drives and displays to the individual chips and interconnects. Focussing solely on the hardware components of a design tends to ignore the impact of the software on the overall power consumption of the system. Software constitutes a major component of systems where power is a constraint. Its presence is very visible in a mobile computer, in the form of the system software and application programs running on the main CPU. But software also plays an even greater role in general digital applications, since an ever growing fraction of these applications are now being implemented as embedded systems. Embedded systems are characterized by the fact that their functionality is divided between a hardware and a software component. The software component usually consists of application-specific software running on a dedicated processor, while the hardware component usually consists of application-specific circuits. In light of the above, there is a clear need for considering the power consumption in systems from the point of view of software. Software impacts the system power consumption at various levels of the design. At the highest level, this is determined by the way functionality is partitioned between hardware and software. The choice of the algorithm and other higher level decisions about the design of the software component can affect system power consumption in a big way. The design of system software, the actual application

source code, and the process of translation into machine instructions - all of these determine the power cost of the software component. In order to systematically analyze and quantify this cost, however, it is important to start at the most fundamental level. This is at the level of the individual instructions executing on the processor. Just as logic gates are the fundamental units of computation in digital hardware circuits, instructions can be thought of as the fundamental unit of software. This motivates the need for analyzing power consumption from the point of view of instructions. Accurate modelling and analysis at this level is the essential capability needed to quantify the power costs of higher abstractions of software, and to search the design space in software power optimizations.

In spite of its importance, very little previous work exists for analyzing power consumption from the point of view of software. Some attempts in this direction are based on architectural level analysis of processors. The underlying idea is to assign power costs to architectural modules such as datapath execution units, control units, and memory elements. In [1], [2] the power cost of a module is given by the estimated average capacitance that would switch when the given module is activated. More sophisticated statistical power models are used in [3], [4]. Activity factors for the modules are then obtained from functional simulation over typical input streams. Power costs are assigned to individual modules, in isolation from one another. Thus, these methods ignore the correlations between the activities of different modules during execution of real programs.

Since the above techniques work at higher levels of abstraction, the power estimates they provide are not very accurate. For greater accuracy, one has to use power analysis tools that work at lower levels of the design - physical, circuit, or switch level [5], [6], [7]. However, these tools are slow and impractical for analyzing the total power consumption of a processor as it executes entire programs. These tools also require the availability of lower level circuit details of processors, something that most embedded system designers do not have access too. This is also the reason why the power contribution of software and the potential for power reduction through software modification has either been overlooked or is not fully understood.

1.1. Instruction Level Power Analysis

The above problems can be overcome if the current being drawn by the CPU during the execution of a program is physically measured. An instruction level power analysis technique based on physical measurements has recently been developed [8]. This technique helps in formulating instruction level power models that provide the fundamental information needed to evaluate the power cost of entire programs. This technique has so far been applied to three commercial, architecturally different processors - the Intel 486DX2 (a CISC processor), the Fujitsu SPARClite 934 (a RISC processor), and a Fujitsu proprietary DSP processor. The purpose of this paper is to provide a general description of the instruction level power analysis technique, based on its application for these three different processors.

The power models for the subject processors are described and interesting observations resulting from the comparison of these are highlighted. Other salient observations resulting from the analysis of these processors are summarized and these provide useful insights into power consumption in processors in general. Instruction level analysis of each processor helps to identify the reasons for variation in power from one program to another. These differences can then be exploited in order to search for low power alternatives for each program. The information provided by the instruction level analysis can guide higher-level design decisions like hardware-software partitioning and choice of algorithm. But it can also be directly used by automated tools like compilers, code generators and code schedulers for generating code targeted towards low power. Several ideas in this regard as motivated by the power analysis of the subject processors are also described.

2. Applications of Instruction Level Power Analysis

The previous section described the primary motivation for power analysis at the instruction level.

There are several additional applications of this analysis and it is instructive to list the important ones here:

- The information provided by the analysis is useful in assigning an accurate power cost to the software component of a system. For power constrained embedded systems, this can help in verifying if the overall system meets its specified power budget.
- The most common way of specifying power consumption in processors is through a single number - the average power consumption. Instruction level analysis provides additional resolution about power consumption that cannot be captured through just this one number. This additional resolution can guide the careful development of special programs that can be used as power benchmarks for more meaningful comparisons between processors.
- The proposed measurement based instruction level analysis methodology has the novel strength that it does not require knowledge of the lower level details of the processor. However, if micro-architectural details of the CPU are available, they can be related to the results of the analysis. This can lead to more refined models for software power consumption, as well as power models for the micro-architecture that may potentially be more accurate than circuit or logic simulation based models.
- The additional insight provided by an instruction-level power model also provides directions for modifications in processor design that lead to the most effective overall power reduction. Instructions can be evaluated both in terms of their power cost as well as frequency of occurrence in typical compiler or even hand-generated code. This combined information can be used to prioritize instructions that should be re-implemented to be less expensive in terms of power.

3. Analysis Methodology

This section describes in greater detail the measurement based technique that was referred to in the previous sections. This technique has so far been applied to three commercial processors:

- Intel 486DX2-S Series, 40MHz, 3.3V (referred to as the 486DX2). A CISC processor based on the x86 architecture. It is widely used in mobile and desktop PCs [9], [10].
- Fujitsu SPARCliteMB86934, 20MHz, 3.3V (referred to as the '934). A 32-bit RISC processor based on the SPARC architecture. It has been specially designed for embedded applications [11], [12].
- Fujitsu proprietary DSP, 40MHz, 3.3V (referred to as the DSP). A new implementation of an internal Fujitsu DSP architecture. It is used in several embedded DSP applications.

The basic idea that allows the use of the measurement based technique in the development of instruction level power models of given processors will also be described in this section. But first, we have to clarify the distinction between "power", a term that we have been using so far, and the term "energy". The average power consumed by a processor while running a certain program is given by: $P = I \times V_{CC}$, where P is the average power, I is the average current, and V_{CC} is the supply voltage. Power is also defined as the *rate at which energy is consumed.* Therefore, the energy consumed by a program is given by: $E = P \times T$, where T is the execution time of the program. This in turn is given by: $T = N \times \tau$, where N is the number of clock cycles taken by the program and τ is the clock period.

Energy consumption is the primary concern for mobile systems, which run on the limited energy available in a battery. Power consumption, on its own, is of importance in applications where cooling and packaging costs are a concern. Energy consumption is the focus of attention in this paper. While we will attempt to maintain a distinction between the two terms, we may sometimes use the term power to refer to energy, in adherence to common usage. It should be noted, nevertheless, that power and energy are closely related, and the energy cost of a program is simply the product of its average power cost and its running time.

3.1. Current Measurement

As can be seen from the above discussion, the ability to measure the current drawn by the CPU dur-

ing the execution of the program is essential for measuring the power/energy cost of the program. The different current measurement setups used in our work point to some of the options that can be used.

Board Based Measurements In the case of the 486DX2 study, the CPU was part of a mobile personal computer evaluation board. The board was designed for current measurements and thus the power supply connection to the CPU was isolated from the rest of the system. A jumper on this connection allows an ammeter to be inserted in series with the power supply and the CPU. The ammeter used is a standard off the shelf, dual-slope integrating digital ammeter. Programs can be created and executed just as in a regular PC. If a program completes execution in a short time, a current reading cannot be visually obtained from the ammeter. To overcome this, the programs being considered are put in infinite loops. The current waveform will now be periodic. Since the chosen ammeter averages current over a window of time ($100ms$), if the period of the current waveform is much smaller than this window, a stable reading will be obtained. The limitation of this approach is that it cannot directly be used for large programs. But this is not a limitation, since the main use of this technique is for performing an instruction-level power analysis. As discussed in the next section, short loops are adequate for this. This inexpensive current measurement approach works very well here. The current drawn by the external DRAM chips is also measured in a similar way. A similar measurement technique is also used in the case of the Fujitsu DSP. However, the DSP board had not been laid out with current measurements in mind. Therefore, the power pins of the CPU had to be lifted from the board in order to create an isolated power supply connection for them.

Tester Based Measurements A suitable board was not available for the '934. Therefore, an alternative experimental setup, consisting of a processor chip and an IC tester machine was used. The program under consideration was first simulated on a VERILOG model of the CPU. This produces a trace file consisting of vectors that specify the exact logic values that would appear on the pins of the CPU for each half-cycle during the execution of the program. The tester then applies the voltage levels specified by the vectors on each input pin of the CPU. This recreates the same electrical environment that the CPU would see on a real board. The current drawn by the CPU is monitored by the tester using an internal digital ammeter.

It should be stressed that the main concepts described in this paper are independent of the method used to measure average current. The results of the above approaches have been validated by comparisons with other current measurement setups. But if sophisticated data acquisition based measurement instruments are available, the measurement method can be based on them, if so desired. Interestingly, instruction level power power analysis can be conducted even for un-fabricated CPUs. Instead of physical current measurements, current estimates can be obtained through simulations on low level design models of the CPU.

4. Instruction Level Power Models

The instruction level analysis scheme described in the previous section has been applied to all three subject processors. Instruction level power models have been developed based on the results of these analyses. The key observations are summarized in this section. Separate references provide greater detail for each individual processor [13], [14], [15]. The basic components of each power model are the same. The first component is the set of *base costs* of individual instructions. The other component is the power cost of *inter-instruction effects*, i.e., effects that involve more than one instruction. This includes the effect of circuit-state, and other effects like stalls and cache misses. These components of the power models are described below:

4.1. Instruction Base Costs

The primary component of the power models is the set of *base costs* of instructions. The base cost of an instruction can be thought of as the cost as-

Table 1. Subset of the base cost table for the 486DX2 and the '934

	Intel 486DX2				Fujitsu SPARClite '934			
No.	**Instruction**	**Current** (mA)	**Cycles**	**Energy** $(10^{-8}J)$	**Instruction**	**Current** (mA)	**Cycles**	**Energy** $(10^{-8}J)$
1	`nop`	276	1	2.27	`nop`	198	1	3.26
2	`mov dx,[bx]`	428	1	3.53	`ld [%l0],%i0`	213	1	3.51
3	`mov dx,bx`	302	1	2.49	`or %g0,%i0,%l0`	198	1	3.26
4	`mov [bx],dx`	522	1	4.30	`st %i0,[%l0]`	346	2	11.4
5	`add dx,bx`	314	1	2.59	`add %i0,%o0,%l0`	199	1	3.28
6	`add dx,[bx]`	400	2	6.60	`mul %g0,%r29,%r27`	198	1	3.26
7	`jmp`	373	3	9.23	`srl %i0,1,%l0`	197	1	3.25

sociated with the basic processing needed to execute the instruction. The experimental procedure used to determine this cost requires a program containing a loop consisting of several instances of the given instruction. The average current drawn during the execution of this loop is measured. The product of this current and V_{CC} is the base power cost of the instruction. The base power cost multiplied by the number of non-overlapped cycles needed to execute the instruction is proportional to its base energy cost. Table 1 presents a sample of the base costs of some instructions for the 486DX2 and the '934. The measured average current, number of cycles, and the base energy costs are also shown. The base energy costs are derived from the formula shown in Section 3.

There are some points to be noted with regard to the assignment of base costs to instructions:

- The definition of base costs follows the convention that the base costs of instructions should not reflect the power contribution of effects like stalls and cache misses. The programs used to determine the base costs have to be designed to avoid these effects. The power costs of these effects are modelled separately.
- The program loops used to determine the base costs should be large enough to overcome the impact of the jump instruction at the bottom of the loop. But they should not be so large so as to cause cache misses. Loop sizes of around 200 have been found to be appropriate.
- It has been observed that, in general, instructions with similar functionality tend to have similar base costs. This observations suggests that similar instructions can be arranged in classes, and a single average cost can be assigned to each class. Doing so speeds up the task of power analysis of the given processor. Table 2 illustrates the application of instruction grouping in the case of the DSP. The commonly used instructions have been grouped into 6 classes as shown.
- The base cost of an instruction can vary with the value and address of the operands used. While appropriate measurement experiments can give the exact cost if the operand and address values are known, in real situations these values are often unknown until runtime. The alternative is to assign a single average cost as the base cost of an instruction. This is justified, since extensive experimentation reveals that the variation in operands leads to only a limited variation in base costs. The DSP, which was the smallest of the three processors, exhibited the maximum variation. But even this was less than 10% for most instructions. Therefore, the inaccuracy due to the use of averages will be limited.

4.2. Effect of Circuit State

The switching activity, and hence, the power consumption in a circuit is a function of the change in circuit state resulting from changes in two consecutive sets of inputs. Now, during the determination of base costs, the same instruction executes each time. Thus, it can be expected that the change in circuit state between instructions would be less here, than in an instruction sequence in which consecutive instructions differ from one another. The concept of *circuit state overhead* for a pair of instructions is used to deal with this effect. Given any two instructions, the current for a

Table 2. Average base costs for instruction classes in the DSP

	LDI	LAB	MOV1	MOV2	ASL	MAC
Current range (mA)	15.8 - 22.9	34.6 - 38.5	18.8 - 20.7	17.6 - 19.2	15.8 - 17.2	17.0 - 17.4
Average energy ($10^{-8}J$)	0.160	0.301	0.163	0.151	0.136	0.142

loop consisting of an alternating sequence of these instructions is measured. The difference between the measured current and the average base costs of the two instructions is defined as the circuit state overhead for the pair. For a sequence consisting of a mix of instructions, using the base costs of instructions almost always underestimates the actual cost. Adding in the average circuit state overhead for each pair of consecutive instructions leads to a much closer estimate.

While the above effect was observed for all the subject processors, it had a limited impact in the case of the 486DX2 and the '934. In the case of the 486DX2, the circuit state overhead varied in a restricted range, 5-30mA, while most programs varied in the range of 300-420mA. In the case of the '934, the overhead was less than 20mA between integer instructions, and in the range 25-34mA between integer and floating point instructions. In contrast, most programs themselves vary in the range 250-400mA. The explanation for the limited impact may lie in the fact that in large complex processors like the 486DX2 and '934, a major part of the circuit activity is common to all instructions, e.g., the clocks, instruction prefetch, memory management, pipeline control, etc. Circuit state can certainly result in significant variation within certain control and data path modules. But the impact of the variation on the net power consumption of the processor will be masked by the much larger common cost.

It should also also follow from the above that if instruction control and the data path constitute a larger fraction of silicon, the impact of circuit state should be more visible. This indeed happens in the case for the DSP, a smaller, more basic processor. Table 3 shows the average overhead costs between different classes of instructions. Considering the fact that for most programs the average current is in the range 20-60mA, several numbers in the table are significantly large.

4.3. Other Inter-Instruction Effects

The final component of the power model is the power cost of other inter-instruction effects that can occur in real programs. Examples are prefetch buffer and write buffer stalls [10], other pipeline stalls, and cache misses. Base costs of instructions do not reflect the impact of these inter-instruction effects. Separate costs need to be assigned to these effects through specific current measurement experiments. The basic idea is to write programs where these effects occur repeatedly. This helps to isolate the power costs of these effects. For example, in the case of the 486DX2, an average cost of 250mA per stall cycle was determined for prefetch buffer stalls [8]. The average cost for a cache miss was 216mA per cache miss cycle. Multiplying the power cost of each kind of stall or cache miss by the number of cycles taken for each, gives the energy cost of these effects.

4.4. Overall Instruction Level Power Model

The previous subsections described the basic components of the instruction level power models of the subject processors. These models form the basis of estimating the energy cost of entire programs. For any given program, P, its overall energy cost, E_P, is given by:

$$E_P = \sum_i (B_i \times N_i) + \sum_{i,j} (O_{i,j} \times N_{i,j}) + \sum_k E_k \quad (1)$$

The base cost, B_i, of each instruction, i, weighted by the number of times it will be executed, N_i, is added up to give the base cost of the program. To this the circuit state overhead, $O_{i,j}$, for each pair of consecutive instructions, (i,j), weighted by the number of times the pair is executed, $N_{i,j}$, is added. The energy contribution, E_k, of the other inter instruction effects, k, (stalls and cache misses) that would occur during the execution of the program, is finally added.

Table 3. Average pairwise circuit state overhead costs for the DSP (in mA)

	LDI	LAB	MOV1	MOV2	ASL	MAC
LDI	3.6	13.7	15.5	6.3	10.8	6.0
LAB		2.5	1.9	12.2	20.9	15.0
MOV1			4.0	18.3	10.5	3.8
MOV2				25.6	26.7	22.2
ASL					3.6	8.0
MAC						12.5

The base cost and overhead values are obtained as shown in the previous sections. As described in Section 4.2, circuit state varies in a limited range in the case of the 486DX2 and the '934. This suggests a more efficient and yet fairly accurate way of modelling this effect for these processors. Instead of a table of pairwise overhead values, a *constant* value is used for all instruction pairs. For e.g., $15mA$ and $18mA$ in the case of the 486DX2 and the '934 respectively. A table is still needed for the DSP, since this effect has a significant impact and greater variation, in the case of this processor.

The other parameters in the above formula vary from program to program. The execution counts N_i and $N_{i,j}$ depend on the execution path of the program. This is dynamic, run-time information. In certain cases it can be determined statically but in general it is best obtained from a program profiler. For estimating E_k, the number of times pipeline stalls and cache misses occur has to be determined. This is again dynamic information that can be statically predicted only in certain cases. In general, this information is obtained from a program profiler and cache simulator. A software power/energy estimation framework based on the above model is described in [8].

The 486DX2 program shown in Table 4 will be used to illustrate the basic elements of the estimation process. The program has three basic blocks as shown in the figure (A basic block is defined as a contiguous section of code with exactly one entry and exit point. Thus, every instruction in a basic block is executed as many times as the basic block.). The average current and the number of cycles for each instruction are provided in two separate columns. For each basic block, the two columns are multiplied and the products are summed up over all instructions in the basic block. This yields a value that is proportional to the base energy cost of one instance of the basic block. The values are 1713.4, 4709.8, and 2017.9, for `B1`, `B2`, and `B3` respectively. `B1` is executed once, `B2` four times, and `B3` once. The `jmp main` statement has been inserted to put the program in an infinite loop. Cost of the `jl L2` statement is not included in the cost of `B2` since its cost is different depending on whether the jump is taken or not. It is taken 3 times and not taken once. Multiplying the base cost of each basic block by the number of times it is executed and adding the cost of the unconditional jump `jl L2`, we get a number proportional to the total energy cost of the program. Dividing it by the estimated number of cycles (72)gives us an average current of $369.1mA$. Adding the circuit state overhead offset value of $15.0mA$ we get $384.0mA$. This program does not have any stalls, and thus, no further additions to the estimated cost are required. If in the real execution of this program, some cold-start cache misses are expected, their energy overhead will have to be added. The actual measured average current is $385.0mA$. Thus, the estimate is within 0.26% of the measured value.

An interesting extension of the above ideas is the development of power profilers for given processors. The above instruction level power model suggests that this can easily be done by enhancing existing performance based profilers with the power costs of instructions and inter-instruction effects. Using this data, the profilers can generate a cycle by cycle profile of the power consumption of given programs.

When average values are used for base costs etc., the accuracy of the energy estimate given by the model described in Equation 1 is limited to some extent by the range of variation in the average and the actual costs. However, the accuracy of the energy estimate is primarily limited by the accuracy in determining the dynamic information regarding the program. Other than this the model

Table 4. Illustration of the Estimation Process

Program	Current(mA)	Cycles
; Block B1		
main:		
mov bp,sp	285.0	1
sub sp,4	309.0	1
mov dx,0	309.8	1
mov word ptr -4[bp],0	404.8	2
;Block B2		
L2:		
mov si,word ptr -4[bp]	433.4	1
add si,si	309.0	1
add si,si	309.0	1
mov bx,dx	285.0	1
mov cx,word ptr _a[si]	433.4	1
add bx,cx	309.0	1
mov si,word ptr _b[si]	433.4	1
add bx,si	309.0	1
mov dx,bx	285.0	1
mov di,word ptr -4[bp]	433.4	1
inc di, 1	297.0	1
mov word ptr -4[bp],di	560.1	1
cmp di,4	313.1	1
jl L2	405.7(356.9)	3(1)
;Block B3		
L1:		
mov word ptr _sum,dx	521.7	1
mov sp,bp	285.0	1
jmp main	403.8	3

is very accurate. For example, for the 486DX2 and the '934, for instruction sequences where the dynamic information was fully known, the maximum difference between the estimated and the measured cost was less than 3%.

It should also be mentioned that in certain applications, e.g., speech processing, some statistical characteristics of the input data are known [16]. Incorporating this knowledge into the power model can lead to more accurate power estimates. This may be specially beneficial in the case of the DSP, which shows greater sensitivity to data based power variations than the other two processors.

4.5. Impact of Internal Power Management

An examination of the base costs of the '934 in Table 1 reveals that the cost for different operations like `OR`, `SHIFT`, `ADD`, or `MULTIPLY` does not show much of a variation. It may well be the case that the differences in the circuit activity for these instructions are much less relative to the circuit activity common to all instructions. Thus, these differences may not reflected in the comparisons of the overall current cost. Nevertheless, the almost complete lack of variation is somewhat counter-intuitive. For instance, it is expected that the logic for an `OR` should be much less than that for a `MULTIPLY`, thus leading to some variation in the overall current drawn for these instructions. The reason for the similarity of the costs most likely has to do with the way ALUs are traditionally designed. A common bank of inputs feeds all the different ALU modules, and thus all the modules switch and consume power, even though on any given cycle, only one of the modules computes useful results. This observation motivates a power reduction technique called *guarded evaluation* [17]. Under this, the modules that are not needed for the current ALU operation are disabled. Thus, it can be expected that if this technique were to be used, the power costs of the different ALU operations will show a variation depending upon their functionality.

The above idea is actually an extension of the principles of *power management*, which refers to the dynamic shutting down of modules that are not needed for a given computation. Power management is gaining popularity as an effective power reduction technique, and has been implemented in recent processors like the Low Power Pentium, PowerPC 603 [18] , and others [19]. Logic level techniques based on the power management idea have also been proposed recently [17], [20], [21]. An aggressive application of power management in a processor may have interesting ramifications for the instruction level power analysis of the processor. First, the base costs of different instructions may show greater variation than they do now. Variations due to differences in data may also increase, both due to the presence of data dependent power management features and due to a general decrease in the overall power consumption. The overall reduction in power may also make the effect of circuit state overhead more prominent. Some power management features may get activated depending on the occurrence of specific sequences of instructions, and these may require special handling.

A related effect was observed in the case of the DSP. The inputs to the on-chip multiplier on the DSP are latched. Thus, the change in the circuit state in the multiplier occurs only for multiply instructions. This change in circuit state is observed even if multiply instructions are not consecutive, and due to the relatively large power contribution of the multiplier for this processor, this effect can actually get reflected in the power cost of instruction sequences. An accurate way to deal with the effect is to add in the exact circuit state overhead for consecutive multiply instructions, even when they are not adjacent in the instruction execution order. An easier but approximate alternative is to enhance the base cost of the multiply instruction with an average value for this overhead. This assumes an unkown state for the multiplier on each multiply instruction, but eliminates the need to keep track of the preceding multiply. While this effect was observed only in the specific case of multiply instructions in the DSP, and for none of the larger processors, aggressive use of power management may mean that the basic power model described in Section 4.4 may need to be adapted in certain cases. And finally, if the mechanism of the major power management features is not described in public domain data books, greater experimental effort may be needed in order to conduct a comprehensive power analysis of the processors. These issues will be investigated further as part of future work.

5. Software Energy Optimization Techniques

It is generally accepted that there is a great potential for energy reduction through modification of software. However, very little has been done to effectively exploit this potential. This has largely been due to the lack of practical techniques for analysis of software energy consumption. The instruction level analysis technique described in the previous sections overcomes this deficiency. Application of this technique provides the fundamental information that can guide the development of energy efficient software. It also helps in the identification of sources of energy reduction that can then be exploited by software development tools like compilers and code generators and schedulers. Several ideas in this regard as motivated by our analysis of the subject processors are described below. Some of these ideas have general applicability for most processors. Others are based on specific architectural features of the subject processors.

5.1. Reducing Memory Accesses

An inspection of energy costs reveals an important fact that holds for all three processors - instructions that involve memory accesses are much more expensive than instructions that involve just register accesses. For example, for the 486DX2, instructions that use register operands cost in the vicinity of $300mA$ per cycle. In contrast, memory reads cost upwards of $400mA$, even in the case of a cache hit. Memory writes cost upwards of $530mA$. Every memory access can also potentially lead to caches misses, misaligned accesses, and stalls. These increase the number of cycles needed to complete the access, and the energy cost goes up by a corresponding factor. The energy consumption in the external memory system adds

Table 5. Energy Optimization of `sort` and `circle` for the 486DX2

Program `sort`	`hlcc.asm`	`hfinal.asm`
Avg. Current (mA)	525.7	486.6
Execution Time (μsec)	11.02	7.07
Energy ($10^{-6}J$)	19.12	11.35
Energy Reduction		40.6%

Program `circle`	`clcc.asm`	`cfinal.asm`
Avg. Current (mA)	530.2	514.8
Execution Time (μsec)	7.18	4.93
Energy ($10^{-6}J$)	12.56	8.37
Energy Reduction		33.4%

an additional energy penalty for cache misses, and for each write in case of write-through caches (as in the 486DX2 and the '934).

These observations point to the large energy savings that can be attained through a reduction in the number of memory accesses. This motivates the need for development of optimizations to achieve this reduction at all levels of the software design process, starting from higher level decisions down to the generation of the final assembly code. At the higher level, some ideas for control flow transformations [22] and data structure design for signal processing applications have been proposed [23] by other researchers. Our experiments provide physical data to analyze these ideas quantitatively.

Attempts can also be made to reduce memory operations during generation of the final code. This can be done automatically if compilers are used, but the basic ideas are applicable even if the assembly code is created manually. This is the level that we explored further using the instruction level analysis technique. The technique provides the guiding information as described above, and is also used to quantify the effectiveness of different ideas.

During compilation, the most effective way of reducing memory operands is through better utilization of registers. The potential of this idea was demonstrated through some experiments in the case of the 486DX2 [24] and the results are also shown in Table 5. The first program in the table is a *heapsort* program ("`sort`" [25]). `hlcc.asm` is the assembly code for this program generated by `lcc`, a general purpose `ANSI C` compiler [26]. The sum of the observed average CPU and memory currents is given in the table above. The program execution times and overall energy costs are also reported. The generated code for the main routine is shown on the left in Table 9. While `lcc` generates good code in general, it often makes tradeoffs in favor of faster compilation time and lesser compiler complexity. For example, register allocation is performed only for temporary variables. Local and global variables for the program are normally not allocated to registers. Optimizations were performed by hand on this code, in order to facilitate a more aggressive use of register allocation. The final code is shown on the right in Table 9. The energy results are shown in Column 3 of Table 5. There is a 40% reduction in the CPU and memory energy consumption for the optimized code. Results for another program (`circle`) are also shown in Table 5. Large energy reduction, about 33%, is observed for this program too.

It should be noted that register allocation has been the subject of research for several years due to its role in traditional compilation. The results of our study show that this research also has an immediate application in compilation for low energy. Further, it also motivates the aggressive use of known techniques, and the development of newer techniques in this direction.

On a related note, an interesting RISC vs. CISC power tradeoff is suggested by the following observation. In the 486DX2, a memory read that hits the cache is about $100mA$ more expensive than a register read. This difference is only $10mA$ in the case of the '934 (compare entries 2 and 3 for the two processors in Table 1). The smaller difference can be attributed to the larger size of the register file in the '934, which leads to a higher power cost for accessing registers. The '934 has 136 registers, as opposed to only 8 in the

Table 6. Effect of instruction reordering in the '934

No.	Instruction	Register contents
1	fmuls %f8,%f4,%f0	%f8=0, %f4=0)
2	andcc %g1,0xaaa,%l0	(%g1=0x555)
3	faddd %f10,%f12,%f14	(%f10=0x123456, %f12=0xaaaaaa)
4	ld [0x555],%o5	
5	sll %o4,0x7,%o6	(%o4=0x707)
6	sub %i3,%i4,%i5	(%i3=0x7f, %i4=0x44)
7	or %g0,0xff,%l0	
	Sequence	**Current (mA)**
a	1,2,3,4,5,6,7	227.5
b	1,3,5,7,2,4,6	224
c	1,4,7,2,5,3,6	226
d	2,3,7,6,1,5,4	228
e	5,3,1,4,6,7,2	223.5

486DX2. A large register file is characteristic of RISC architectures. Availability of more registers can help to reduce memory accesses, leading to power reduction. But on the other hand, a larger register file also means that each register access itself will be costlier.

5.2. Energy Cost Driven Code Generation

Code generation refers to the process of translating a high-level problem specification into machine code. This is either done automatically through compilers, or in certain design situations, it is done by hand. In either case, code generation involves the selection of the instructions to be used in the final code, and this selection is based on some cost criterion. The traditional cost criteria are either the size or the running time of the generated code. The main idea behind energy cost driven code generation is to select instructions based on their energy costs instead. The instruction energy costs are obtained from the analysis described in the previous sections.

An energy based code generator was created for the 486DX2 using this idea. An existing tree pattern based code generator selected instructions based on the number of cycles they took to execute. It was modified to use the energy costs of the instructions instead. Interestingly, it was found that the energy and cycle based code generators produced very similar code.

This observation provides quantitative evidence for a general trend that was observed for all the subject processors. This is that energy and running times of programs track each other closely. It was consistently observed that the difference in average current for sequences that perform the same function is never large enough to compensate for any difference in the number of cycles. Thus, the shortest sequence is also invariably the least energy sequence. Since this observation holds for all the subject processors, each of which represents a distinct architecture style, it is reasonable to expect that it will also hold for most other processors that exist today.

This is a very important observation, and something that has not been addressed in previous literature. It can be considered as empirical justification for a powerful guideline for software energy reduction for today's processors - as a first step towards energy reduction, do what needs to be done to improve performance. Potentially large energy reductions can be obtained if this observation is used to guide high-level decisions like hardware-software partitioning and choice of algorithm. It should be noted that this guideline is motivated and justified by the results of our instruction level analysis. Without the physical corroboration provided by the results, we would not have been able to put forth this guideline.

It also bears mentioning that it is possible that there may be certain application specific processors where this observation may not hold in general. It is also possible that aggressive use of use

Table 7. Results for Different Energy Optimization Techniques for the DSP

Benchmark		Original	Packing	Scheduling	Swapping
FJex1	Energy ($10^{-8}J$)	2.79	2.46	2.12	
	Energy Reduction		12.0%	24.0%	
FJex2	Energy ($10^{-8}J$)	3.91	3.14	2.83	
	Energy Reduction		19.7%	27.7%	
LP_FIR60	Energy ($10^{-8}J$)	57.60	30.80		25.60
	Energy Reduction		46.6%		55.6%
IIR4	Energy ($10^{-8}J$)	10.10	7.47	6.78	6.37
	Energy Reduction		26.3%	33.1%	37.2%
FFT2	Energy ($10^{-8}J$)	9.59	9.35	8.97	8.64
	Energy Reduction		3.4%	7.4%	10.9%

of power management and other low power design optimizations may also lead to situations where the fastest code may not always be the least energy code. While these cases remain to be identified, code generation based on energy costs will be useful in its own right for these cases.

5.3. Instruction Reordering for Low Power

Reordering of instructions in order to reduce switching between consecutive instructions is a method for energy reduction that does not involve a corresponding reduction in the number of cycles. An instruction scheduling technique based on this idea has been proposed in another work [27]. In this, instructions are scheduled in order to minimize the estimated switching in the control path of an experimental RISC processor. Our experiments, however, indicate that in terms of net energy reduction for the entire processor, instruction reordering may not always be effective. It has been observed to have very limited impact in the case of the 486DX2 and the '934. Table 6 illustrates this with an example. As can be seen, different reordering of the given sequence of instructions lead to very little change in the measured average current. The idea behind reordering instructions can be seen as an attempt to reduce the overall circuit state overhead between consecutive instructions. But as seen in Section 4.2, this quantity is bounded in a small range and does not show much variation in the 486DX2 and the '934.

In the case of the DSP, however, this quantity is more significant and does show relatively greater variation (refer to Section 4.2 and Table 3). Thus, instruction reordering is more beneficial for this processor. A scheduling algorithm that uses the measured overhead costs was developed for this processor [15]. The data in Table 7 illustrates the effectiveness of this algorithm. This table shows the impact of different software energy optimization techniques that are applicable for the DSP ("packing" and "swapping" will be discussed later). Five standard signal processing programs were used for the experiment. `FJex1` and `FJex2` are real Fujitsu applications for vector preprocessing. `LP_FIR60` is a length-60 linear phase FIR filter. `IIR4` is a fourth-order direct form IIR filter, and `FFT2` is a radix-2 decimal-in-time FFT butterfly. The last three programs were taken from the TMS320 embedded DSP examples in [28] and translated into native code for the target DSP processor. Column 2 shows the initial energy consumption of the programs. Columns 3, 4, and 5 show the energy consumption and the overall percent energy reduction after the application of each technique. The three techniques are applied one after the other, from left to right. The percent by which the values in Column 4 are lower than those in Column 3 quantifies the effectiveness of instruction scheduling alone. As shown, up to 14% reduction in energy (for `FJex1`) has been observed using this algorithm. Table 10 shows the initial code for `IIR4`, and the final code after all three optimizations. For this example, instruction scheduling alone leads to a 9.3% reduction in energy.

Switching on the address and data pins is a specific manifestation of the effect of circuit state. Software transformations to reduce this switching are believed to be a possible energy reduction method. The large capacitance associated with these pins can indeed lead to greater current

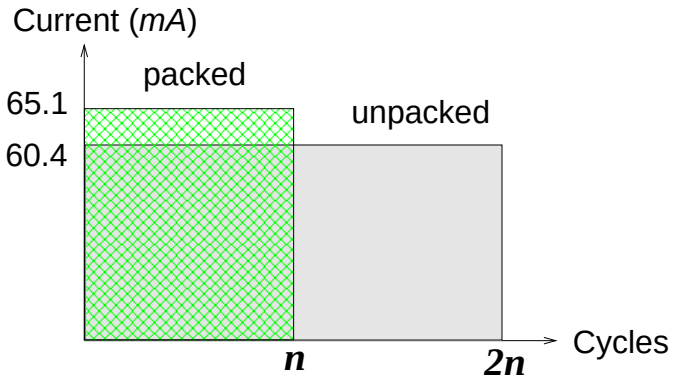

Fig. 1. Comparison of energy consumption for packed and unpacked instructions.

when these pins switch. However, there are some practical considerations that should be noted in this regard. First, the presence of on-chip caches greatly reduces external traffic. In addition the traffic becomes unpredictable making it harder to model the correlation between consecutive external accesses. Second, real systems often use external buses and memories that are slower than the CPU, necessitating the use of "wait states". This implies that, on the average, pins switch less often. Thus, for instance, in the case of the 486DX2 system, switching on the address and data pins had only a limited impact for most programs - even for back to back writes, the impact of greater switching on the address lines was less than 5%. Finally, even for processors without caches, it is difficult to model this switching for general programs. The necessary information is fully available only at run-time. However, reasonable models may be feasible for more structured applications like signal processing, and this bears further investigation.

5.4. Processor Specific Optimizations

Instruction level power analysis of a given processor can lead to the identification of features specific to that processor that can then be exploited for energy efficient software. We identified such specific features for each of the subject processors. Some of the more noteworthy examples are briefly described below.

Instruction Packing The DSP has a special architectural feature called *instruction packing* that allows an ALU instruction and a memory data transfer instruction to be packed into a single instruction. The packed instruction executes in one cycle, as opposed to a total of two for the sequence of two unpacked instructions. Interestingly, we found that the use of packing always leads to large energy reductions, even though a packed instruction represents the same functionality as a sequence of two unpacked instructions. Figure 1 illustrates this graphically. The average current for a certain sequence of n packed instructions is only marginally greater than for the corresponding sequence of $2n$ unpacked instructions. Therefore, since the unpacked instructions complete in twice as many cycles, their energy consumption (proportional to the area under the graph) is almost twice that of the packed instructions. Thus, instructions should be packed as much as possible.

Table 10 illustrates the application of packing for the example `IIR4`. Instructions with two opcodes separated by a colon are packed instructions, e.g., `MUL:LAB`. The use of packing leads to large energy savings for real programs (e.g. 26% for `IIR4` and 47% for `LP_FIR60`, as shown in Column 3 of Table 7). The substantial savings attainable also make it worthwhile to develop program transformation and scheduling techniques that can lead to better utilization of instruction packing.

Dual Memory Loads The Fujitsu DSP has two on-chip data memory banks. A special dual

Table 8. Software controlled power management in the '934

Instruction: `or %i0,0,%l0`		
Units powered down	**Current (mA)**	**% Energy Reduction**
None	198	0.0
SDI	185	6.6
FPU	176	11.1
DMA,FPU	172	13.1
FIFO,FPU	163	17.7
SDI,DMA,FIFO,FPU	154	22.2

load instruction can transfer two operands, one from each memory, to registers in one cycle. The same task can also be attained by two single load instructions over two cycles. However, we found that the average current for the latter was only marginally lower, and thus, doubling of execution cycles implies a corresponding increase in energy consumption. The large energy difference also justifies the use of memory allocation techniques that can lead to better utilization of dual loads. A static memory allocation technique based on *simulated annealing* was developed for this purpose [29]. Application of this technique led to a 47% energy reduction over the case where data is assigned to only bank for `LP_FIR60`. Our observations also suggest that other memory allocations techniques developed from the point of view of improving performance can also find direct application for energy reduction [30].

It should be noted that both the above features are not unique to the Fujitsu DSP, but are also provided by several other popular DSP processors, e.g., the Motorola 56000 series. The above observations are likely to be valid for these other processors too.

Swapping Multiplication Operands The results of our analysis of the Fujitsu DSP indicate that the on-chip multiplier on this processor is a major source of energy consumption for signal processing applications. This motivated a more detailed analysis of power consumption for multiply instructions. It was discovered that similar variations in the values of the two operands lead to different degrees of variations in the power consumption of multiply operations. This is reasonable, since the multiplier is based on the Booth multiplication algorithm, which treats the two operands in very different ways. We found that an appropriate swapping of the operands, in order to exploit this asymmetry, leads to up to 30% reduction in multiplication energy costs. This can translate into appreciable energy reduction for entire programs, as shown in Column 5 of Table 7. For example, for `LP_FIR60`, the use of operand swapping reduces the energy consumption of the packed code by an additional 16%.

Software Controlled Power Management The '934 provides a software mechanism for powering down parts of the CPU. By setting appropriate bits in a system control register through a specified sequence of instructions, the clock inputs to certain modules can be enabled or disabled. We were able to quantify the effectiveness of this mechanism by using our analysis technique. Table 8 shows the measured power reductions attained for an `OR` instruction, when some combinations of the SDRAM interface (SDI), DMA module, floating-point unit (FPU), and floating-point FIFOs are powered down. It is evident from the results, that power management, i.e., powering down of unneeded modules can lead to significant power savings. It should also be noted that *automatic* power management will be a more effective and more generally applicable power reduction technique. The energy overhead associated with power management will be much less if it is controlled by logic internal to the CPU, rather than through a sequence of instructions. The temporal resolution of the power management strategy will also be much finer, since it can then be applied on a cycle by cycle basis.

Table 9. 486DX2 Software Energy Optimization Example: `sort.c`

Compiler Generated Code

```
sort:
push    ebx
push    esi
push    edi
push    ebp
mov     ebp,esp
sub     esp,24
mov     edi,dword ptr 014H[ebp]
mov     esi,1
mov     ecx,esi
mov     esi,edi
sar     esi,cl
lea     esi,1[esi]
mov     dword ptr -20[ebp],esi
mov     dword ptr -8[ebp],edi
L3:
mov     edi,dword ptr -20[ebp]
cmp     edi,1
jle     L7
mov     edi,dword ptr -20[ebp]
sub     edi,1
mov     dword ptr -20[ebp],edi
lea     edi,[edi*4]
mov     esi,dword ptr 018H[ebp]
add     edi,esi
mov     edi,dword ptr [edi]
mov     dword ptr -12[ebp],edi
jmp     L8
L7:
mov     edi,dword ptr 018H[ebp]
mov     esi,dword ptr -8[ebp]
lea     esi,[esi*4]
add     esi,edi
mov     ebx,dword ptr [esi]
mov     dword ptr -12[ebp],ebx
mov     edi,dword ptr 4[edi]
mov     dword ptr [esi],edi
mov     edi,dword ptr -8[ebp]
sub     edi,1
mov     dword ptr -8[ebp],edi
cmp     edi,1
jne     L8
mov     edi,dword ptr 018H[ebp]
mov     esi,dword ptr -12[ebp]
mov     dword ptr 4[edi],esi
jmp     L2
L8:
mov     edi,dword ptr -20[ebp]
mov     dword ptr -16[ebp],edi
lea     edi,[edi*2]
mov     dword ptr -4[ebp],edi
jmp     L12
L11:
mov     edi,dword ptr -4[ebp]
mov     esi,dword ptr -8[ebp]
cmp     edi,esi
jge     L14
lea     edi,[edi*4]
mov     esi,dword ptr 018H[ebp]
mov     ebx,edi
add     ebx,esi
mov     ebx,dword ptr [ebx]
mov     edi,dword ptr 4[edi][esi]
cmp     ebx,edi
jge     L14
mov     edi,dword ptr -4[ebp]
lea     edi,1[edi]
mov     dword ptr -4[ebp],edi
L14:
mov     edi,dword ptr -12[ebp]
mov     esi,dword ptr -4[ebp]
lea     esi,[esi*4]
mov     ebx,dword ptr 018H[ebp]
add     esi,ebx
mov     esi,dword ptr [esi]
cmp     edi,esi
jge     L16
mov     edi,2
mov     esi,dword ptr 018H[ebp]
mov     ebx,dword ptr -16[ebp]
mov     ecx,edi
sal     ebx,cl
add     ebx,esi
mov     ecx,dword ptr -4[ebp]
mov     dword ptr -24[ebp],ecx
mov     ecx,edi
mov     edi,dword ptr -24[ebp]
sal     edi,cl
add     edi,esi
mov     edi,dword ptr [edi]
mov     dword ptr [ebx],edi
mov     edi,dword ptr -4[ebp]
mov     dword ptr -16[ebp],edi
mov     esi,edi
add     esi,edi
mov     dword ptr -4[ebp],esi
jmp     L12
L16:
mov     edi,dword ptr -8[ebp]
lea     edi,1[edi]
mov     dword ptr -4[ebp],edi
L12:
mov     edi,dword ptr -4[ebp]
mov     esi,dword ptr -8[ebp]
cmp     edi,esi
jle     L11
mov     edi,dword ptr -16[ebp]
lea     edi,[edi*4]
mov     esi,dword ptr 018H[ebp]
add     edi,esi
mov     esi,dword ptr -12[ebp]
mov     dword ptr [edi],esi
jmp     L3
L2:
mov     esp,ebp
pop     ebp
pop     edi
pop     esi
pop     ebx
ret
```

Energy Optimized Code

```
sort:
push    ebp
mov     edi,dword ptr 08H[esp]
mov     esi,edi
sar     esi,1
inc     esi
mov     ebp,esi
mov     ecx,edi
L3:
cmp     ebp,1
jle     L7
dec     ebp
mov     esi,dword ptr 0cH[esp]
mov     edi,dword ptr [edi*4][esi]
mov     ebx,edi
jmp     L8
L7:
mov     edi,dword ptr 0cH[esp]
mov     esi,dword ptr 4[edi]
mov     ebx,dword ptr [ecx*4][edi]
mov     dword ptr [ecx*4][edi],esi
dec     ecx
cmp     ecx,1
jne     L8
mov     dword ptr 4[edi],ebx
jmp     L2
L8:
mov     edi,ebp
mov     edx,edi
add     edi,edi
mov     eax,edi
jmp     L12
L11:
cmp     eax,ecx
jge     L14
mov     esi,dword ptr 0cH[esp]
mov     edi,dword ptr [eax*4][esi]
cmp     edi,dword ptr 4[eax*4][esi]
jge     L14
inc     eax
L14:
mov     esi,dword ptr 0cH[esp]
cmp     ebx,dword ptr[eax*4][esi]
jge     L16
mov     edi,dword ptr [eax*4][esi]
mov     dword ptr [edx*4][esi],edi
mov     edx,eax
add     eax,eax
jmp     L12
L16:
mov     eax,ecx
inc     eax
L12:
cmp     eax,ecx
jle     L11
mov     esi,dword ptr 0cH[esp]
mov     dword ptr [edx*4][esi],ebx
jmp     L3
L2:
pop     ebp
ret
```

6. Future Directions

There are several directions in which we would like to extend this work. The first of these would be to extend the analysis methodology to processors whose architecture and implementation style is significantly different from the processors studied here. We would specially like to analyze processors that are based on superscalar and VLIW architectures. These seem to be the architectures of choice for high performance processors in the near future, and with ever increasing integration and clock frequencies, the power problem will become even more acute for these processors. We would also like to continue to work on avenues for power reduction through software optimization, and development of automated tools where applicable. The results of our work show that a number of ideas from existing literature on traditional software optimization can be used here, but new techniques will also be developed. The ability to evaluate the power cost of the software component of an embedded system can also be used as a first step towards ideas and tools for hardware-software co-design for low power. Finally, the software perspective is essential in understanding the power consumption in processors. This additional perspective can help guide us in the search for more power efficient architectures, and this issue will be explored in the future.

7. Conclusions

The increasing role of software in today's systems demands that energy consumption be studied from the perspective of software. This paper describes a measurement based instruction level power analysis technique that makes it feasible to effectively analyze software power consumption. The main observations resulting from the application of this technique to three commercial processors were presented here. These provide useful insights into the power consumption in processors. They also illustrate how a systematic analysis can lead to the identification of sources of software power consumption. These sources can then be targeted through suitable software design and transformation techniques. The ability to quantitatively analyze the power consumption of software makes it possible to deal with the overall system power consumption in an integrated way. A unified perspective allows for the development of more effective power reduction techniques that are applicable for the entire system.

References

1. T. Sato, M. Nagamatsu, and H. Tago. Power and performance simulator: ESP and its application for 100MIPS/W class RISC design. In *Proceedings of 1994 IEEE Symposium on Low Power Electronics*, pages 46–47, San Diego, CA, Oct. 1994.
2. P. w. Ong and R. H. Yan. Power-conscious software design - a framework for modeling software on hardware. In *Proceedings of 1994 IEEE Symposium on Low Power Electronics*, pages 36–37, San Diego, CA, Oct. 1994.
3. P. Landman and J. Rabaey. Black-box capacitance models for architectural power analysis. In *Proceedings of the International Workshop on Low Power Design*, pages 165–170, Napa, CA, April 1994.
4. P. Landman and J. Rabaey. Activity-sensitive architectural power analysis for the control path. In *Proceedings of the International Symposium on Low Power Design*, pages 93–98, Dana Point, CA, April 1995.
5. L. W. Nagle. SPICE2: A computer program to simulate semiconductor circuits. Technical Report ERL-M520, University of California, Berkeley, 1975.
6. A. Salz and M. Horowitz. IRSIM: An incremental MOS switch-level simulator. In *Proceedings of the Design Automation Conference*, pages 173–178, 1989.
7. C. X. Huang, B. Zhang, A. C. Deng, and B. Swirski. The design and implementation of PowerMill. In *Proceedings of the International Symposium on Low Power Design*, pages 105–110, Dana Point, CA, April 1995.
8. V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: A first step towards software power minimization. *IEEE Transactions on VLSI Systems*, 2(4):437–445, December 1994.
9. Intel Corp. *Intel486 Microprocessor Family, Programmer's Reference Manual*, 1992.
10. Intel Corp. *i486 Microprocessor, Hardware Reference Manual*, 1990.
11. Fujitsu Microelectronics Inc. *SPARClite Embedded Processor User's Manual*, 1993.
12. Fujitsu Microelectronics Inc. *SPARClite Embedded Processor User's Manual: MB86934 Addendum*, 1994.
13. V. Tiwari, S. Malik, and A. Wolfe. Power analysis of the Intel 486DX2. Technical Report CE-M94-5, Princeton Univ., Dept. of Elect. Eng., June 1994.
14. V. Tiwari and Mike T.-C. Lee. Power analysis of a 32-bit embedded microcontroller. Accepted for publication in the *VLSI Design* Journal.
15. T. C. Lee, V. Tiwari, S. Malik, and M. Fujita. Power analysis and low-power scheduling techniques for embedded DSP software. In *Proceeding of the* International Symposium on System Synthesis, Cannes, France, Sept. 1995.

Table 10. DSP Software Energy Optimization Example : IIR4

Portion of Original Code	After Energy Optimizations
LDI coefa,X0	LDI coefa,X0
LDI xn,X2	LDI coefb,X1
MOV (X2),C	LDI xn,X2
LDI datanm1,X3	LDI datanm1,X3
LAB (X3+1),(X0+1)	LDI datan,X4
MUL:	MOV (X2),C
LAB (X3+1),(X0+1)	LAB (X0+1),(X3+1)
MSMC:	MUL:LAB (X0+1),(X3+1)
LAB (X3+1),(X0+1)	MSMC:LAB (X0+1),(X3+1)
LDI datan,X4	MSMC:LAB (X0+1),(X3+1)
MSMC:	MSMC:
LAB (X3+1),(X0+1)	MSMC:
MSMC:	MOV C,(X4)
LDI coefb,x1	RESC:LAB (X1+1),(X4+1)
MSMC:	MUL:LAB (X1+1),(X4+1)
MOV C,(X4)	MSMC:LAB (X1+1),(X4+1)
RESC:	MSMC:LAB (X1+1),(X4+1)
LAB (X4+1),(X1+1)	MSMC:
MUL:	MSMC:
LAB (X4+1),(X1+1)	MOV C,(X4)
MSMC:	
LAB (X4+1),(X1+1)	
MSMC:	
LAB (X4+1),(X1+1)	
MSMC:	
MSMC:	
MOV C,(X4)	

16. P. Landman and J. Rabaey. Power estimation for high level synthesis. In *Proceedings of the European Design Automation Conference*, pages 361–366, Paris, Feb. 1993.
17. V. Tiwari, S. Malik, and P. Ashar. Guarded evaluation: Pushing power management to logic synthesis/design. In *Proceedings of the International Symposium on Low Power Design*, pages 221–226, Dana Point, CA, April 1995.
18. S. Gary et al. PowerPC 603, a microprocessor for portable computers. *IEEE Design & Test of Computers*, pages 14–23, Winter 1994.
19. A. Correale. Overview of the power minimization techniques employed in the IBM PowerPC 4xx embedded controllers. In *Proceedings of the International Symposium on Low Power Design*, pages 75–80, Dana Point, CA, April 1995.
20. M. Alidina, J. Monteiro, S. Devadas, A. Ghosh, and M. Papaefthymiou. Precomputation-based sequential logic optimization for low power. *IEEE Transactions on VLSI Systems*, pages 426–436, December 1994.
21. L. Benini and G. De Micheli. Transformation and synthesis of fsms for low power gated clock implementation. In *Proceedings of the International Symposium on Low Power Design*, pages 21–26, Dana Point, CA, April 1995.
22. S. Wuytack, F. Franssen F. Catthoor, L. Nachtergaele, and H. De Man. Global communication and memory optimizing transformations for low power systems. In *Proceedings of the International Workshop on Low Power Design*, pages 203–208, Napa, CA, April 1994.
23. S. Wuytack, F. Catthoor, and H. De Man. Transforming set data types to power optimal data structures. In *Proceedings of the International Symposium on Low Power Design*, Dana Point, CA, April 1995.
24. V. Tiwari, S. Malik, and A. Wolfe. Compilation techniques for low energy: An overview. In *Proceedings of 1994 IEEE Symposium on Low Power Electronics*, pages 38–39, San Diego, CA, Oct. 1994.
25. Press et al. *Numerical Recipes in C*. Cambridge Univ. Press, 1988.
26. C. W. Fraser and D. R. Hanson. A retargetable compiler for ANSI C. *SIGPLAN Notices*, pages 29–43, Oct. 1991.
27. C. L. Su, C. Y. Tsui, and A. M. Despain. Low power architecture design and compilation techniques for high-performance processors. In *IEEE COMPCON*, Feb. 1994.
28. Texas Instruments. *Digital Signal Processing Applications - Theory, Algorithm, and Implementations*, 1986.
29. T. C. Lee and V. Tiwari. A memory allocation technique for low-energy embedded DSP software. In *Proceedings of 1995 IEEE Symposium on Low Power Electronics*, San Jose, CA, Oct. 1995.
30. A. Sudarsanam and S. Malik. Memory bank and register allocation in software synthesis for ASIPs.

In *Proceedings of the International Conference on Computer-Aided Design*, San Jose, CA, Nov. 1995.

Vivek Tiwari received the B. Tech degree in Computer Science and Engineering from the Indian Institute of Technology, New Delhi, India in 1991. Currently he is working towards the Ph.D. degree in the Department of Electrical Engineering, Princeton University. He will be joining Intel Corporation, Santa Clara, CA, in fall 1996.

His research interests are in the areas of Computer Aided Design of VLSI and embedded systems and in microprocessor architecture. The focus of his current research is on tools and techniques for power estimation and low power design. He has held summer positions at NEC Research Labs (1993), Intel Corporation (1994), Fujitsu Labs of America (1994), and IBM T. J. Watson Research Center (1995), where he worked on the above topics.

He received the IBM Graduate Fellowship Award in 1993, 1994, and 1995, and a Best Paper Award at ASP-DAC '95.

Sharad Malik received the B. Tech. degree in Electrical Engineering from the Indian Institute of Technology, New Delhi, India in 1985 and the M.S. and Ph.D. degrees in Computer Science from the University of California, Berkeley in 1987 and 1990 respectively.

Currently he is on the faculty in the Department of Electrical Engineering, Princeton University. His current research interests are: design tools for embedded computer systems, synthesis and verification of digital systems.

He has received the President of India's Gold Medal for academic excellence (1985), the IBM Faculty Development Award (1991), an NSF Research Initiation Award (1992), a Best Paper Award at the IEEE International Conference on Computer Design (1992), the Princeton University Engineering Council Excellence in Teaching Award (1993, 1994, 1995), the Walter C. Johnson Prize for Teaching Excellence (1993), Princeton University Rheinstein Faculty Award (1994) and the NSF Young Investigator Award (1994). He serves/has served on the program committees of DAC, ICCAD and ICCD. The is on the editorial boards of the Journal of VLSI Signal Processing and Design Automation for Embedded Systems.

Andrew Wolfe received a B.S.E.E. in Electrical Engineering and Computer Science from The Johns Hopkins University in 1985 and the M.S.E.E and Ph.D. in Electrical and Computer Engineering from Carnegie Mellon University in 1987 and 1992 respectively. He joined Princeton University in 1991, where he is currently an Assistant Professor in the Department of Electrical Engineering. He served as Program Chair of Micro-24 and General Chair of Micro-26 as well as on the program committees of several IEEE/ACM conferences. He has received the Walter C. Johnson award for teaching excellence at Princeton. His current research interests are in embedded systems, instruction-level parallel architectures and implementations, optimizing compilers and digital video.

Mike Tien-Chien Lee received his B.S. degree in Computer Science from National Taiwan University in 1987, and the M.S. degree and the Ph.D. degree in electrical engineering from Princeton University, in 1991 and 1993, respectively.

He has been working at Fujitsu Laboratories of America, Santa Clara, CA, as a Member of Research Staff since 1994. Before then he was a Member of Technical Staff at David Sarnoff Research Center, Princeton, NJ, working on video chip testing. His research interests include low-power design, embedded DSP code generation, high-level synthesis, and test synthesis. He received a Best Paper Award at ASP-DAC'95.