Power Analysis of a 32-bit Embedded Microcontroller

Vivek Tiwari
Dept. of Electrical Engineering
Princeton University, Princeton, NJ 08544
vivek@ee.princeton.edu

Mike Tien-Chien Lee
Fujitsu Laboratories of America
77 Rio Robles, San Jose, CA 95134
lee@fla.fujitsu.com

Abstract

A new approach for power analysis of microprocessors has recently been proposed [1]. The idea is to look at the power consumption in a microprocessor from the point of view of the actual software executing on the processor. The basic component of this approach is a measurement based, instruction-level power analysis technique. The technique allows for the development of an instruction-level power model for the given processor, which can be used to evaluate software in terms of the power consumption, and for exploring the optimization of software for lower power. This paper describes the application of this technique for a comprehensive instruction-level power analysis of a commercial 32-bit RISC-based embedded microcontroller. The salient results of the analysis and the basic instruction-level power model are described. Interesting observations and insights based on the results are also presented. Such an instruction-level power analysis can provide cues as to what optimizations in the micro-architecture design of the processor would lead to the most effective power savings in actual software applications. Wherever the results indicate such optimizations, they have been discussed. Furthermore, ideas for low power software design, as suggested by the results, are described in this paper as well.

1 Introduction

A very large fraction of the applications in all segments of the electronics industry are being implemented as embedded computer systems. The basic characteristic of these systems is the presence of both a hardware and a software component. The hardware component consists of application-specific circuits, while the software component consists of application-specific software running on dedicated microprocessors. The role of the software component is actually projected to grow in the future. A large number of embedded computing applications are power critical, i.e., power constraints form an important part of the design specification. In light of the growing role of the software component, it is imperative to consider the power consumption of this component when analyzing the total system power consumption. However, most power analysis tools available today, work at only the lower levels of the design – at the circuit and logic level. These tools are not suited to accurately analyze the total power consumption in a microprocessor as it executes entire programs. These tools also requires the availability of lower level circuit details of microprocessors, 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.

A recent work [1] overcomes these deficiencies by developing a methodology that provides a means for analyzing the power consumption of a given microprocessor as it executes a given program. The idea is to use a measurement-based analysis technique for developing and validating an instruction level power model for any given processor. Such a model can then be provided by the processor vendors for both off-the-shelf processors, as well as embedded cores. This can then be used to evaluate embedded software, much as a gate level power model has been used to evaluate logic designs. The ability to evaluate software in terms of the power metric helps in verifying if a design meets its specified power constraints. In addition, it can also be used to search the design space in software power optimization [2].

The initial work in this direction has been in the context of the Intel 486DX2, a general-purpose CISC architecture. This paper describes the application of this power analysis methodology for the Fujitsu SPARClite MB86934, a 32-bit RISC microcontroller [5, 6, 7] targeted for embedded applications. A comprehensive power analysis of this processor has been performed and an instruction level power model has been developed. The salient results of the analysis are described here. Interesting observations and insights based on the results are also presented. The successful application of the analysis methodology for two different processors provides validation for the general applicability of this methodology. This is reinforced by a recent work based on the application of this analysis technique for a specialized embedded DSP processor [3].

2 Processor Overview

The SPARClite MB86934 is a SPARC-based microprocessor optimized for use in embedded applications. A full description of the SPARClite family and of MB86934 (referred to as the '934 from here on) is available from other references [5, 6, 7]. However, some of the features that are relevant for the remainder of this paper are briefly mentioned below:

- Technology: 0.5 micron, 3 level metal CMOS technology. There are three separate power pins for the on-chip phase locked loop (PLL), internal logic, and I/O, respectively. All power supply connections can be at 3.3V.
- On-chip floating point unit (FPU): A high-performance on-chip FPU executes single/double precision operations.
- On-chip FIFOs: FPU instructions can get their operands from a 32-bit FPU register file, or 6 on-chip FIFOs, which are fed directly from main memory through DMA.
- On-chip caches: A 8K, 32-byte line, instruction cache, and a 2K, 16-byte line, data cache. Both caches are 2-way set associative and employ a write-through policy and a LRU replacement algorithm. Cache entries can be locked and the caches can also be disabled.

- Large integer register file: The integer register file consists of 136, 32-bit registers, which are organized into 8 overlapping windows.
- Software controlled power management: A software mechanism is provided to disable the clocks to various functional units in order to conserve power.

3 Experimental Method

The instruction level power analysis technique relies on the ability to measure the average current drawn by the processor. This is motivated by the formulas for the power and energy cost of a program. The average power, P, consumed by a microprocessor while running a certain program is given by: $P = I \times V_{CC}$, where I is the average current, and V_{CC} is the supply voltage. Since power is the rate at which energy is consumed, the energy, E, 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.

For the experimental setup used in this study, V_{CC} was 3.3V and τ was 50ns, corresponding to the 20MHz system clock. Thus, if the average current for an instruction sequence is I Amperes, and the number of cycles it takes to execute is N, the energy cost of the sequence is given by: $E = I \times V_{CC} \times N \times \tau$, which equals: $(16.5 \times 10^{-8} \times I \times N)$ Joules. Throughout the rest of the paper, in order to specify the energy cost of an instruction (instruction sequence), the average current will be specified. The number of cycles will either be explicitly specified, or will be clear from the context.

3.1 Current Measurement

From the above discussion it is evident that to measure the energy cost of a program, the average current drawn by the CPU during the execution of the program has to be measured. The measurement method employed was based on the test and measurement capabilities of a commercial IC tester. 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. Now, the current drawn by the CPU varies over the execution of a program, and so the ammeter may not yield a steady visual reading. To overcome this, the method used in the case of the 486DX2 is applied [1]. The programs being considered are put in infinite loops. Thus, the resulting current waveforms are now periodic. The ammeter averages current over a window of time (about $100ms$) for the purpose of analog to digital conversion. If the period of the current waveform is much smaller than this window, a stable reading is obtained.

3.2 Instruction Level Power Analysis

The above method make it feasible to measure the power cost of a given program. By designing special programs and measuring their power cost, it is possible to obtain the basic information needed for an instruction level power analysis of the processor, based on the following hypothesis. Consider a program consisting of several instances of a certain instruction. Since the CPU is executing the same instruction over and over again, it seems intuitive that the entire activity in the CPU can be attributed to that instruction. The power cost of the CPU for that program can be considered as the basic power cost of the given instruction. In real programs there may be other effects involving more than one instruction that can impact the power cost, e.g., the effect of circuit state, pipeline stalls and cache misses. By designing programs where these effects occur repeatedly can similarly provide a way for assigning power costs to these effects too.

This hypothesis has been validated for the Intel 486DX2. It has also found to be applicable for the ‘934, as the subsequent sections will show. The instruction level power model that has been developed for the two processors has the same basic components. The first of these is the set of *base costs* of instructions. The base cost of a given instruction is obtained by creating a program consisting of several instances of the instruction executing in a loop. The other component of the power model is the power costs of *inter-instruction effects*. The first of these is the effect of change of circuit state between consecutive instructions. During determination of the base costs, the same instruction is executed again and again. It can be expected that the change in circuit state between consecutive instructions will be less here, than for the case in which consecutive instructions differ. The quantity *circuit state overhead* is introduced to account for this effect. Given a pair of instructions, the current for an alternating sequence of the two instructions is measured. The difference between this current and the average of base costs of the two instructions is defined as the circuit state overhead for the pair. This effect is illustrated in some of the later sections and is discussed in detail in Sections 10 and 11. The power cost of other inter-instruction effects like pipeline stalls and cache misses can also be obtained through appropriate experiments.

4 Power Analysis of the ‘934

In the subsequent sections, the specifics of the instruction level power model for the ‘934 are presented. Other results that highlight the characteristics of the power consumption, as it relates to instructions and software are also reported. For the sake of clarity, the experiments are divided into several categories, each of which is treated in a separate section. The results include the power costs of the important instructions, and examples that illustrate the power model that is used for the estimation of power consumption of instruction sequences. The power costs of external memory accesses, and the effect of the caches on the overall power cost is also explored. Results are also provided for the impact of software controlled power management on the power cost of instructions. The salient observations and interesting insights based on the results of each sections are also briefly discussed. One of the benefits of an instruction-level analysis is that it provides cues as to what optimizations in the micro-architecture design would lead to the most effective power savings in actual software applications. Wherever the results indicate such optimizations, they have been discussed. Furthermore, ideas for low power software design, as suggested by the results, are also described.

The following observations are valid for all experiments reported in this paper. Repeated runs of an experiment at different times resulted in only a very small variation in the observed average current values. The variation was in the range $\pm 1\ mA$. The current drawn by the power pin connected to the on-chip PLL was

Table 1: Sample integer ALU instructions: caches enabled.

no.	instruction	register contents	$I1$ (mA)	$I2$ (mA)
1	`or %g0,0, %l0`		177	21
2	`or %g0,0xfff,%l0`		174.5	21
3	`or %g0,%i0,%l0`	(%i0=0)	177.5	21
4	`or %g0,%i0,%l0`	(%i0=0xfff)	173.5	21
5	`add %i0,%o0,%l0`	(%i0=0, %o0=0)	178	21
6	`add %i0,%o0,%l0`	(%i0=0, %o0=0xfff)	174	21
7	`add %i0,%o0,%l0`	(%i0=0xfff, %o0=0)	174	21
8	`add %i0,%o0,%l0`	(%i0=0xfff, %o0=0xfff)	173	21
9	`add %o0,%i1,%l2`	(%o0=0, %i1=0x555)	174.5	21
10	`srl %i0,%o0,%l0`	(%i0=0xfff, %o0=0x1)	179	21
11	`srl %i0,1,%l0`	(%i0=0xfff)	176	21
12	`srl %i1,%o5,%g3`	(%i0=0x555, %o5=1)	174.5	21
13	`or %g0,%r16,%i0`	(%r16=0)	178	21
14	`orcc %i1,%o0,%l1`	(%i1=0x555, %o0=0xaaa)	173	21
15	`subx %g0,%r16,%i0`	(%r16=0)	172	21
16	`xor %g0,%r16,%i0`	(%r16=0)	177.5	21
17	`xor %g0,%r17,%i0`	(%r17=0)	176	21
18	`andcc %g1,0xAAA,%l0`	(%g1=0x555)	179	21
19	`sll %o4,0x7,%o6`	(%o4=0xf0)	173.5	21
20	`umul %i0,0x2,%o3`	(%i0=0xaaa)	174.5	21
21	`mul %g0,%r29,%r27`	(%r29=0)	177	21

Table 2: Integer ALU instructions: caches disabled vs. enabled.

no.	instruction	register contents	disabled $I1$ (mA)	enabled $I1$ (mA)
1	`or %g0,0,%l0`		187.5	177
2	`or %g0,0xfff,%l0`		184	174.5
3	1&2		196	192
4	`or %g0,%i0,%l0`	(%i0=0)	188	177.5
5	`or %g0,%i0,%l0`	(%i0=0xfff)	184	173.5
6	4&5		192	187.5
7	`srl %i0,%o0,%l0`	(%i0=0xfff, %o0=0x1)	188.5	179
8	`srl %i0,1,%l0`	(%i0=0xfff)	184	176

very small and below the measuring range of the tester. The current drawn by the power pins connected to the internal logic and I/O circuitry is denoted by $I1$ and $I2$, respectively. The symbol & used in the tables below for an instruction pair i&j denotes an instruction sequence where instructions i and j are executed alternately.

5 Integer ALU Instructions: Caches Enabled

Table 1 shows the base costs for some integer instructions. The caches, prefetch, and write buffers are enabled. The base costs are shown in terms of the $I1$ and $I2$ current. All the instructions shown execute in one cycle, except entry 20, which executes in 2 cycles.

Observations and comments

Integer ALU instructions tend to have very similar costs, as shown in the above table. They vary in the range of 170 - 180 mA, in terms of $I1$ current. $I2$ current is mostly stable around 21 mA. The reason for the low $I2$ current is that the caches are enabled, and thus, after one iteration of the loop, the instructions are always available in the instruction cache, and there is no traffic on the I/O pins.

The $I1$ current shows a limited variation depending upon the actual value of the data operands used. Variation due to the use of different registers is not significant. Entries 1 and 2 show the cost for an OR instruction for two different immediate operand values. Entries 3 and 4 show the costs for the OR instruction when only register operands are used, but the content of one of the registers is different. Entries 5 to 9 show the costs for an ADD instruction for different combinations of the operands. There seems to be a correlation between the number of 1's in the binary representation of operands and the base cost – more the 1's, lesser the cost. However, the overall range of variation is very limited, and thus the use of average base costs for instructions should suffice for program energy estimation purposes. This in fact is the only option in cases where the exact value of operands cannot be determined until runtime.

- An interesting observation leading from the above results is that the cost of the ALU instructions doesn't seem to depend much on the ALU operation that is being performed. The cost of an OR, SHIFT, ADD, or MULTIPLY all seems to be about the same. This is somewhat counter-intuitive. For instance, it is expected that the logic for an OR should be much less than that for an ADD, thus leading to a variation in the current drawn for these operations.

  The reason for the similarity of the costs may have to do with the way ALUs are traditionally designed. All the different ALU sub-functions are fed by a common bank of inputs, and the outputs of the appropriate module are selected by a multiplexor structure. Now, in any given cycle, the results of only one sub-function are needed. Thus, the circuit activity in the other sub-functions is a waste of power. The design can be modified for low power by extending the principles of automatic power management. If the inputs of the sub-functions that are not needed are prevented from switching, the power consumed in these sub-functions can be saved. This observation motivates the concept of *guarded evaluation*, which has been explored in detail in another reference [4].

6 Integer ALU Instructions: Caches Disabled

Table 2 shows some of the same instructions as the previous table. However, in this case, the on-chip caches have been disabled. Prefetch and write buffers are also disabled. These modules can be enabled or disabled by writing into a specific system control register. The number of memory wait states is zero. Column 4 shows $I1$ current in this case. The $I2$ current was 134 mA for all entries. Column 5 shows the $I1$ current for the case when the caches are enabled. The $I2$ current in this case was 21 mA.

Observations and comments

Since the instruction cache is disabled, every instruction access goes to the external I/O pins. The $I2$ current is therefore higher than when caches are enabled. The $I1$ current (internal logic current) is also about 10 mA higher. Entries 3 and 6 show what happens when different instructions are executed together. This will be discussed in greater detail in Section 10.

In terms of overall current, disabling the instruction cache leads to a total CPU current increase of about 123 (= 10 + (134-21)) mA, i.e., about 64%. However, when the cache is disabled in the '934, every instruction fetch takes *two* cycles, even for a zero wait state system. Thus, in terms of energy, disabling the instruction cache leads to at least about a 124% increase in the energy consumption. This points to two things:

- Accessing the cache is much more energy efficient than accessing external memory. Thus, attempts to increase the cache hit rate through software modifications will be very beneficial. It is further indicated that attempts to increase the hit rate through architectural transformations may also help reduce the overall energy consumption.
- In certain embedded applications, the designer may choose to disable the caches. This is usually done to improve the

Table 3: Load and store instructions: caches enabled and locked.

no.	instruction	register contents	$I1$ (mA)	$I2$ (mA)
1	`ld [0x0],%i0`	(%i0=0)	191.5	21
2	`ld [0xffc],%i0`	(%i0=0)	187	21
3	`ld [%l0],%i0`	(%i0=0, %l0=0)	192	21
4	`ld [%l0],%i0`	(%i0=0xfff, %l0=0)	189.5	21
5	`ld [%l0],%i0`	(%i0=0xffffff, %l0=0)	187.5	21
6	`ld [%l0],%i0`	(%i0=0xffffffff, %l0=0)	185	21
7	`ld [%l0],%i0`	(%i0=0, %l0=0xffc)	191	21
8	`ld [%l0],%i0`	(%i0=0, %l0=0xfffffc)	188	21
9	`ld [%l0],%i0`	(%i0=0, %l0=0xfffffffc)	185	21
10	`st %i0,[0x0]`	(%i0=0)	173	21
11	`st %i0,[0xffc]`	(%i0=0)	169	21
12	`st %i0,[%l0]`	(%i0=0, %l0=0)	175	21
13	`st %i0,[%l0]`	(%i0=0xfff, %l0=0)	173.5	21
14	`st %i0,[%l0]`	(%i0=0, %l0=0xffc)	172	21
15	`ldub [%l0],%i5`	(%i5=0xaaa, %l0=0)	192.5	21
16	3&4		206	21
17	3&5		213	21
18	3&6		216	21
19	3&7		202.5	21
20	3&8		207	21
21	3&9		211	21
22	12&13		185	21
23	12&14		183	21

performance predictability for real-time systems. However, this will lead to a penalty in terms of the system energy consumption, and thus, the battery life. This fact has to be understood and weighed in, when deciding on whether the caches should be disabled.

7 Load and Store Instructions: Caches Enabled and Locked

Table 3 shows the cost of some instructions that reference memory. Since the '934 is a RISC, load-store machine, the only instructions that explicitly reference memory are the loads and the stores. The above results are for the specific case when the caches are active and the entries in the data cache are locked. This implies that every data access is a cache hit. In addition, since the cache entries are locked, the store (write) instructions also don't go out to external memory. Note that the '934 has a write-through cache, and thus in the normal case, each data write also goes out to the external bus. Since there is no traffic on the I/O pins, the $I2$ current is low. Each instruction also executes in one cycle in this case.

Observations and comments

Entries 1 and 2 are direct loads and entries 10 and 11 are direct stores. The rest of the instructions utilize the indirect addressing modes. The results indicate that there is not much difference between these two addressing modes, in terms of base current. Entries 3 to 6 show the variation in the cost of a load for a fixed address but differing data operands. Entries 3, 7, 8, and 9 show the variation for a fixed data operand but differing addresses. Entries 12, 13, and 12, 14, show the corresponding variation in the case of stores. The general trend points to the correlation between base cost and the number of 1's in the binary representation of the data operand and the memory address. More the 1's, lower the cost. The variation in the costs, though, is limited. Entries 16 to 23 show what happens when different instructions execute together. The data and address registers used for each instruction in the pair were different, but the register contents were the same as shown in the individual instruction entries. Entry 16 is for the case when the instructions in entry 3 and 4 execute alternately. The current is higher than the average of the two base costs. This is due to the effect of circuit state overhead. 12 data operand bits flip between entries 3 and 4. The entries 17 and 18 show the results for greater data flips. Entries 19 to 20 show the results when the address bits flip between adjacent instructions. The results indicate a positive correlation between the number of bit flips, and the increased effect of circuit state.

The results also lead to the following interesting observations:

- A comparison between Tables 2 and 4 shows that cache accesses aren't much more costly than register accesses. Cache reads are about 10 mA more costly, and cache writes are about the same cost as register accesses. Since both cache and register accesses take one cycle, the energy comparison shows the same relation. This observation is in stark contrast to what was observed in the case of the Intel 486DX2, where cache accesses were much more costly than register accesses. The reason for the similarity in the cost of cache accesses and register accesses in the '934 is most likely due to the large size of its register file. The '934 is a RISC, load-store architecture, and it is characteristic for this architectural style to use a large number of registers. The register file has 136 registers. In addition, it is multiported, and is windowed. In contrast the 486DX2 has a simple register file with only 8 registers.

  This observation illustrates an interesting CISC vs. RISC trade-off with regards to power. On one hand, the availability of a larger number of registers can help reduce the use of memory operands, leading to power reductions. But on the other hand, the larger register file causes each register access itself to be costlier.

- The data also points to the fact that micro-architectural or circuit transformations to optimize the register file for low power, will be very beneficial in terms of overall power reductions. The load-store design of the '934 involves very heavy usage of the registers, and a lower power cost of accessing registers will translate into power reductions for all programs.

- It should be noted that the use of memory operands does have a high cost even in the '934, due to the possibility of cache misses. Also, if the cache is unlocked, stores will incur additional cost in terms of $I2$ current (as shown in the next section), and memory system current. Thus, the use of memory operands should certainly be avoided. This also points out that the cache locking feature should be exploited as far as possible, for applications where energy consumption is a design constraint.

8 Store Instructions: Caches Enabled and Unlocked

Table 4 shows the costs of some store instructions when the caches are enabled but are unlocked. Since the data cache is write-through, all the stores also reference the external main-memory. The number of memory wait states is zero. However, the design of the '934 imposes an extra cycle for every bus transaction. During this cycle the bus is idle.

Observations and comments

Most typical applications do not lock the data cache. Thus, the stores in these applications will go out to the external bus, leading

Table 4: Store instructions: caches enabled and unlocked.

no.	instruction	register contents	$I1$ (mA)	$I2$ (mA)
1	`st %i0,[%l0]`	(`%i0`=0, `%l0`=0)	198	148
2	`st %i0,[%l0]`	(`%i0`=0, `%l0`=0xffc)	191	115
3	`st %i0,[%l0]`	(`%i0`=0, `%l0`=0xffffc)	185	71
4	`st %i0,[%l0]`	(`%i0`=0, `%l0`=0xfffffffc)	181.5	46
5	1&2		198	137
6	1&3		199	116
7	1&4		200	106
8	`st %i0,[%l0]`	(`%i0`=0xfff, `%l0`=0)	193	148
9	`st %i0,[%l0]`	(`%i0`=0xffffff, `%l0`=0)	191	150
10	`st %i0,[%l0]`	(`%i0`=0xffffffff, `%l0`=0)	189	150
11	1&8		203	173
12	1&9		207	193
13	1&10		211	206.5

to higher $I2$ current, as shown in the table. This table, therefore, reflects the more typical cost of memory writes. Note that there will also be an additional system energy penalty due to the current being drawn by the external memory.

Entries 1, 8, 9, and 10 show the variation in the cost of the stores for a fixed address but varying data. There is a minor decrease in both the $I1$ and $I2$ current for increasing number of 1's. Entries 11 to 13 consider instruction sequences where different stores alternate. For example, entry 11 shows the cost for a sequence consisting of the instructions in entry 1 and 2 appearing in succession. The $I1$ and $I2$ currents are greater than the average of the current costs for the individual instructions. This is another illustration of the effect of circuit state overhead. The $I1$ overhead represents the effect of the circuit state in the internal logic circuits, while the $I2$ overhead represents the effect of switching on the data pins. Entry 11 involves 12 bit flips at the data lines, while entries 12 and 13 involve 24 and 32 bit flips respectively. As expected, greater number of bit flips, result in greater current. The increase in current is greater in the case of $I2$ current. This too is expected, since the I/O pads typically involve larger capacitive loads.

Entries 1 to 4 show the variation in the cost of stores for a fixed data value but varying addresses. The $I1$ current decreases with an increase in the number of 1's in the binary representation of the address. The $I2$ current also decreases, but the decrease is very drastic. For example, consider entries 1 and 4. Entry 1 has no 1's in the address, and entry 4 has 30 1's. The difference in the $I2$ current is 102 mA. This translates into about 3.3 mA per each occurrence of "0" in the binary representation of the address. Comparisons between the other entries also yields the same result. This observation seems strange, since if the same address is being put on the bus for every instruction, the address pins should not switch. However, the address pins do switch due to the following reason. Every memory transaction involves an extra cycle during which the bus is idle. During these bus idle cycles, the '934 pulls up the address pins, i.e., the pins go to logical value 1. Thus, even if back to back store instructions use the same address, there is an intervening cycle when the address pins are all 1's. This means that the pins corresponding to the address bits that are 0 will switch each time. More 1's in the address value means less switching, and thus lower $I2$ current.

Entries 5 to 7 show another illustration of this effect. The $I2$ value for a pair of stores is about the same as the average of the $I2$ values of the individual stores. There is no circuit state overhead. The reason being that the intervening bus idle cycle, in which the address pins are pulled up, isolates the two stores from each other.

The above results lead to the following observations:

- The above results show that occurrence of 0's in the address values means greater current cost, on the order of 3.3 mA of $I2$ current for each occurrence of a 0. This suggests that if data and instructions are stored at the higher end of memory, the program energy cost may be reduced. The reason being that, in that case, on the average, addresses will have lesser 0's in them. The power reduction can potentially be very significant.

- It should be noted that the higher current cost of 0's in the address is a manifestation of the effect of circuit state (in other words, switching) on the address pins. Now, most real systems utilize wait states, since memory access times are usually slower than the CPU clock period. If the number of wait states increases, there will be a greater number of bus idle cycles. We know that during the bus idle cycles, the address pins are at a constant 1. Thus, more wait states means that on the average, the address pins will switch less often, leading to a lesser impact on the overall system energy cost. This is in line with the observation in the case of the 486DX2, where it was noted that for real systems, switching on the external pins had limited impact on the overall energy cost of programs.

Table 5: Floating point instructions: caches enabled.

no.	instruction	register contents	n	$I1$ (mA)
1	`fitos %f4,%f0`	(`%f4`=0)	1	177.5
2	`fitos %f4,%f0`	(`%f4`=0xfff)	1	177.5
3	`fmovs %f4,%f0`	(`%f4`=0)	1	175
4	`fmovs %f4,%f0`	(`%f4`=0xff)	1	175
5	`fmovs %f4,%f0`	(`%f4`=0xffff)	1	174
6	`fmovs %f4,%f0`	(`%f4`=0xffffff)	1	175
7	`fitod %f4,%f0`	(`%f4`=0)	1	178
8	`fadds %f8,%f4,%f0`	(`%f8`=0, `%f4`=0)	1	175.5
9	`fadds %f8,%f4,%f0`	(`%f8`=0xffff, `%f4`=0xffff)	1	176
10	`fadds %f8,%f4,%f0`	(`%f8`=0x555555, `%f4`=0xaaaaaa)	1	177
11	`fadds %f8,%f4,%f0`	(`%f8`=0xffffff, `%f4`=0xffffff)	1	178
12	`faddd %f8,%f4,%f0`	(`%f8`=0, `%f4`=0)	1	177
13	`faddd %f8,%f4,%f0`	(`%f8`=0xffff, `%f4`=0xffff)	1	177.5
14	`faddd %f8,%f4,%f0`	(`%f8`=0x555555, `%f4`=0xaaaaaa)	1	177.5
15	`ld [0x0],%f8`	(`%f8`=0)	1	205
16	`ld [0x0],%f8`	(`%f8`=0x4b7ff)	1	198
17	`ld [0x0],%f8`	(`%f8`=0x4b7fffff)	1	193
18	`ldd [0x0],%f8`	(`%f8`=0,0)	1	214
19	`ldd [0x0],%f8`	(`%f8`=0x416fffff, e0000000)	1	200
20	`ldd [0x0],%f8`	(`%f8`=0x4b7fffff, 4b7fffff)	1	192
21	`fmuls %f8,%f4,%f0`	(`%f8`=0, `%f4`=0)	1	174
22	`fmuls %f8,%f4,%f0`	(`%f8`=0xfff, `%f4`=0xfff)	1	175
23	`fmuls %f8,%f4,%f0`	(`%f8`=0x555555, `%f4`=0xaaaaaa)	1	175
24	`fdivs %f8,%f4,%f0`	(`%f8`=0xaaaaaa, `%f4`=0x555555)	13	167.5
25	`fdivs %f8,%f4,%f0`	(`%f8`=0xffff, `%f4`=0xffff)	13	168
26	`fdivs %f8,%f4,%f0`	(`%f8`=0, `%f4`=1)	13	167.5
27	26 & 1 `nop`		13	181.5
28	26 & 4 `nop`'s		13	181.5
29	26 & 12 `nop`'s		13	182
30	26 & 1 `add`		13	179
31	26 & 12 `add`'s		13	177.5
32	`fsqrts %f4,%f0`	(`%f4`=0xfe01)	13	173
33	`fsqrts %f4,%f0`	(`%f4`=0xaaaaaa)	13	173.5
34	`fsqrts %f4,%f0`	(`%f4`=0)	13	174
35	34 & 1 `nop`		13	184
36	34 & 4 `nop`'s		13	184.5
37	34 & 12 `nop`'s		13	185
38	34 & 1 `add`		13	181
39	34 & 12 `add`'s		13	180.5

9 Floating Point Instructions

Table 5 shows the costs for some typical floating point instructions. The caches were enabled and unlocked in all experiments. The FPU is pipelined and Column 4 shows the throughput for each instruction. Column 5 shows the $I1$ current. The $I2$ current was 21 mA for all cases. The energy cost of an instruction is proportional to the product of the total current and the number shown in Column

4.

Observations and comments

The results indicate that most instructions that involve the FPU have similar cost. For example, consider entries 1 to 14, and 21 to 23, all of which take one cycle, and don't cause any FPU pipeline interlocks. The dependence of the value of operands is almost negligible, and this may have to do with the circuit design of the FPU. In addition, the dependence of current on the type of FPU operation is also not exhibited. This maybe due to the same reason as discussed in Section 5. Instructions for loading values into floating point registers (entries 15 to 20) result in costs that are similar to those seen in the case of integer registers. The trend with respect to the current cost and the number of 1's in the data operands is also similar.

Entries 24 to 26 show floating point divide instructions, and entries 32 to 34 show square root instructions. The current variation for different operand values is negligible. These instructions take 13 cycles in a particular FPU pipeline stage. This leads to 12 pipeline interlocks. This means that an FPU instruction that immediately follows one of these instructions will have to wait for 12 cycles. However, the integer pipeline may not be held up in most cases, and can continue to execute. Entry 27 shows what happens when a NOP instruction (internally treated as an integer instruction) appears after a divide instruction. The execution of this instruction is hidden within the 12 interlock cycles of the divide. Entries 28 to 31 show other examples when integer instructions follow a divide instruction, and entries 35 to 39 show the same for the square root instruction. These entries show that the current cost in this case isn't much more than when no integer instructions are executed in the FPU interlock cycles. This leads to the following insights:

- When integer instructions are executed during the FPU interlock cycles, the current doesn't increase much beyond what it is when the interlock cycles are completely idle. This suggests that during the FPU interlock cycles, switching activity doesn't completely stop in the other parts of the CPU. If this activity is useless, then eliminating it can result in power reduction during the interlock cycles. This represents another opportunity where automatic power management or *guarded evaluation* may be useful.

- The results also show that for the current implementation of the '934, it is beneficial to execute integer instructions during the FPU interlock cycles. The current cost is not much higher than when the integer instructions are executed independently. Therefore, the decrease in the number of execution cycles translates into actual energy reduction. Execution cycles are reduced, since the cycles required to execute the integer instructions are overlapped with, and thus, hidden in the FPU interlock cycles. Software optimizations to achieve this can therefore be considered as both performance as well as energy optimizations.

10 Effect of Circuit State Overhead and Instruction Reordering

The purpose of Table 6 is to illustrate and quantify the effect of the circuit state on the energy cost of instruction sequences. Base costs of several instructions as well as the costs for pairs of instructions are shown in the table. Only the $I1$ current is shown. The

Table 6: Effect of circuit state overhead.

no.	instruction	register contents	$I1$ (mA)	Ovh. (mA)
1	`or %g0,0,%l0`		177	
2	`or %g0,0x001,%l0`		174	
3	`or %g0,0x00f,%l0`		174	
4	`or %g0,0x0ff,%l0`		174.5	
5	`or %g0,0xfff,%l0`		174.5	
6	1&2	1 opcode flip	178	2.5
7	1&3	4 opcode flips	184.5	9
8	1&4	8 opcode flips	191	15
9	1&5	12 opcode flips	192	16
10	`or %g0,%i0,%l0`	(%i0=0)	177.5	
11	`or %g0,%o0,%l0`	(%io=0xfff)	173.5	
12	10&11	12 data flips	187.5	12
13	`or %g0,%r16,%i0`	(%r16=0)	177.5	
14	`or %g0,%r17,%i0`	(%r17=0)	176	
15	`or %g0,%r15,%i0`	(%r15=0)	175.5	
16	`or %g0,%r23,%i0`	(%r23=0)	176	
17	13&14	1 opcode flip	177.5	1
18	13&16	3 opcode flips	180	3
19	13&15	5 opcode flips	180.5	4
20	`subx %g0,%r16,%i0`	(%r16=0)	172	
21	`xor %g0,%r16,%i0`	(%r16=0)	177.5	
22	`xor %g0,%r17,%i0`	(%r16=0)	176	
23	20&21	4 opcode flips	191.5	17
24	20&22	4 opcode flips	192	18
25	`or %i1,%o0,%l1`	(%i1=0x555, %o0=0)	174.5	
26	`add %o0,%i1,%l2`	(%i1=0x555, %o0=0)	174.5	
27	`or %o5,0x555,%l4`	(%o5=1)	170	
28	`srl %i1,%o5,%g3`	(%i1=0x555, %o5=1)	174.5	
29	25&26		185	10.5
30	26&27		182	9.5
31	27&28		191	19
32	28&25		184.5	10.5
33	`fmuls %f8,%f4,%f0`	(%f8=0,%f4=0)	174	
34	`nop`		176	
35	33&34		202	27
36	`andcc %g1,0xaaa,%l0`	(%g1=0x555)	179	
37	33&36		210	33.5
38	`ld [0x555],%o5`		192.5	
39	`sll %o4,0x7,%o6`	(%o4=0x707)	173.5	
40	38&39		202.5	19.5
41	`or %g0,0xff,%l0`		176	
42	33&41		207	32
43	`fadd %f10,%f12,%f14`	(%f10=0x123456, %f12=0xaaaaaa)	176	
44	38&43		212	28

$I2$ current was 21 mA in all cases. For entries with pairs of instructions, Column 4 shows the current for the combined sequence, and Column 5 shows the circuit state overhead. This value is the difference between the actual current cost of an instruction, and the average of the base costs of the individual instructions.

Observations and comments

The existence of circuit state overhead can be attributed to the fact that each instruction executes in the context of the circuit state set by the previous instruction. The greater the change in the circuit state between instructions, greater should be this overhead. While the change in the circuit state can result from any part of the processor, the common notion is that it is basically due to the change in the opcodes of adjacent instructions. Entries 6 to 9 in Table 6 show this quantitatively with an example. With increasing number of bit flips in the opcodes of adjacent instructions, the overhead increases. However, Table 6 also shows that switching of the opcodes is not the only source of circuit state overhead. For example, entries 18, 19, and 23, 24 involve almost the same number of opcode flips. However, entries 23 and 24 have a much higher overhead cost.

Table 6, and some of the examples presented in Section 7, as well as the next section, quantify several instances of circuit state overhead. These and several other examples indicate that the overhead varies between 0 and about 34 mA. The overhead between integer instructions is typically below 20 mA. The overhead be-

Table 7: Example of instruction reordering.

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	$I1(mA)$
a	1, 2, 3, 4, 5, 6, 7	206.5
b	1, 3, 5, 7, 2, 4, 6	203
c	1, 4, 7, 2, 5, 3, 6	205
d	2, 3, 7, 6, 1, 5, 4	207
e	5, 3, 1, 4, 6, 7, 2	202.5

Table 8: Program energy estimation example.

no.	instruction	register contents	$I1$ (mA)	$I2$ (mA)
1	`st %i0,[%l0]`	(`%i0`=0xaaa,`%l0`=0)	176	21
2	`orcc %i1,%o0,%l1`	(`%i1`=0x555,`%o0`=0)	173	21
3	`ldub [%l0],%i5`	(`%i5`=0xaaa,`%l0`=0)	192.5	21
4	`umul %i0,0x2,%o3`	(`%i0`=0xaaa)	174.5	21
5	1&2		203	21
6	2&3		196.5	21
7	3&4		192.5	21
8	4&1		187.5	21
	Measured current		196	21
	Estimate using base costs		178.1	21
	Overhead estimate		16	0
	Final estimate		194	21

tween floating point and integer instructions is higher, typically in the range 25 - 34 mA. The most important aspect of this observation is that the range of the variation in this overhead is small, compared to the overall CPU current cost.

A recent idea in the area of software design for low power is to reorder instructions to reduce the power cost of a program. This can be seen as an attempt to reduce the average current cost of a program by minimizing the circuit state overhead. Our experiments based on actual energy measurements on the '934, however, reveal that this technique does not translate into significant overall energy reduction. The reason is that the circuit state overhead is bounded in a small range and does not show very significant variation. Thus, different instruction schedules will not vary significantly in their current costs.

Table 7 illustrates this with an example. Entries 1 to 7 show a set of instructions. Entries *a* to *e* show the current cost of different sequences consisting of these instructions. The order of the instructions in the sequences is shown in Column 2 and the $I1$ current cost is shown in Column 3 (I2 current was 21 mA in all cases). The maximum variation in the current is only 4 mA, or about 1.9%. Since all the sequences take the same number of cycles, the energy variation is also 1.9%.

Similar observations were also made in the case of the Intel 486DX2 [2]. It appears that this is characteristic of large, complex CPUs, where a major part of the circuit activity is common to all instructions, e.g., instruction fetch, pipeline control, clocks, etc. However, it may be the case that instruction reordering can result in significant variation in smaller processors, like DSPs, and processors with complex power management features. This bears further investigation.

11 Program Energy Estimation Example

The results of the previous section quantify the parameters of the instruction-level energy model of the '934. It consists of base energy costs of individual instructions, and energy costs of effects that involve more than one instruction, e.g., effect of circuit state, pipeline, and write buffer stalls. This model forms the basis for estimating the energy cost of a given program. For this, the base cost of each instruction, weighted by the number of times it will be executed, is added up to give the base cost of the program. To this the circuit state overhead for each pair of consecutive instructions, weighted by the number of times the pair is executed, is added. The energy contribution of other inter-instruction effects is finally added to give the final energy estimate. The purpose of this section is to illustrate the validity of this model with the help of a very simple example shown in Table 8.

Entries 1 to 4 show a program consisting of a sequence of four instructions. The caches were enabled but the entries were locked in. Columns 4 and 5 show the base current costs of instructions. Instruction 4 takes 2 cycles to execute while instructions 1 to 3 take 1 each. The measured current for the full sequence is also shown. The first step is to estimate the cost for the sequence using just the base costs. Multiplying the $I1$ currents by the number of cycles for each instruction, summing these up, and then dividing by the total number of cycles, gives an estimate for the $I1$ current for the sequence:

$$(176.0 * 1 + 173.0 * 1 + 192.5 * 1 + 174.5 * 2)/5 = 178.1mA$$

The estimate for $I2$ is obviously 21 mA. The inter-instruction effects should now be accounted for, which in this example only include the effect of circuit state. This effect is modeled by considering the circuit state overhead between each pair of consecutive instructions. The measured cost for each pair is shown in entries 6 to 8. The circuit state overhead between instructions 2 and 3 is given by (196.5-(173.0+192.5)/2) = 13.75 mA. Between 3 and 4 it is given by (192.5-(192.5+2*174.5)/3)*3/2 = 18.0 mA. The reason for multiplying by 3/2 is that for an alternating sequence of instructions 3&4, the overhead occurs twice during 3 cycles. In a similar way, the overheads between 1&2 and 4&1 are seen to be 28.5 mA and 18.75 mA, respectively. The average overhead is 16 mA for $I1$ and 0 mA for $I2$. When these overheads are added to the base cost estimates, we obtain 194 mA for $I1$ and 21 mA for $I2$. A comparison of entries 9 and 12 shows the close correspondence of the estimate predicted by the instruction-level power model and the actual estimate. Such a close correspondence was also obtained for other experiments involving other instruction sequences.

The basic energy model developed in the precious sections and demonstrated above is remarkably similar to the model for the 486DX2. In fact, like in the case of the 486DX2, a quick yet efficient way of dealing with circuit state overhead would be to use a constant value for it (18 mA is a good candidate), since the variation in this quantity is limited. Therefore, the program energy estimation methodology that was developed for the 486DX2 [1] can be directly applied to the '934.

12 Software Controlled Partial Power Down

One of the unique features of the '934 is a facility for powering down certain CPU modules that are not needed. This is achieved by setting the appropriate bits in a system control register known as the Power-Down Register. The modules that can be powered down are the SDRAM interface (SDI), the DMA module, the floating-point unit (FPU) and the floating-point FIFOs. Bits in the Power-Down Register that are set to 1 cause the clock input of the corresponding module to be disabled. Clearing the bits re-enables the clock input.

Table 9: Software controlled partial power down.

		`or %i0,0,%l0`		`ld [%l0],%i0`	
3-6 no.	powered-down units	$I1$ (mA)	% saved	$I2$ (mA)	% saved
1	None	177	0.0	192.5	0.0
2	SDI	164	7.6	188.5	2.1
3	DMA	164	7.6	188	2.3
4	FPU	155.5	12.1	180	6.5
5	FIFO	155.5	12.1	180	6.5
6	DMA, FPU	151.5	14.4	174	9.6
7	DMA, FIFO	151.5	14.4	175	9.1
8	FIFO, FPU	142	19.8	166	13.8
9	DMA, FIFO, FPU	138	22.0	162.5	15.6
10	SDI, DMA, FIFO, FPU	133	25.0	157	18.4

Table 9 shows the results of experiments that study the effect of powering down specific modules, or combinations of modules. The $I1$ currents for two different instructions, an OR and a LD are shown in Columns 3 and 5, respectively. The percent reduction in current for each entry is shown in Columns 4 and 6. Entry 1 shows the results when nothing is powered down. Powering down different modules leads to different power savings, the maximum being the case when all the four modules, SDI, DMA, FIFO, and FPU are powered down.

Observations and comments

- As can clearly be seen, powering down of unneeded modules can lead to significant power savings. However, powering down and later powering up a module through software involves the execution of a certain number of control instructions. These instruction themselves consume energy. Thus, powering down of modules will lead to energy savings only if the modules are powered down long enough to compensate for the overhead involved in powering them down and then up.

- The previous observation also indicates that automatic power management of modules will be more effective in saving power. The overhead of powering up and down will be negligible if it is controlled by logic internal to the CPU. Plus, the temporal resolution of the power management strategy can then be much finer - it can even be performed on a cycle by cycle basis.

- An interesting observation leading from the above table is that the power saving achieved by powering down a combination of modules doesn't necessarily equal the sum of the individual power reductions. For example, for the OR instruction, power savings in entry 10 are not equal to the sum of the savings shown in entries 2 to 5. This is because circuit activity in different modules may be correlated for this instruction. Powering down one module also eliminates some activity in another module. Thus, powering down these modules together, results in different savings than what is expected from the savings achieved by powering then down individually.

  This actually illustrates the fact that power estimation/analysis methods have to account for correlations between the activities in various modules for each instruction. A power estimation method based on summing up typical power consumptions of separate modules, while disregarding correlations, may be very inaccurate. Thus, either the exact correlations have to be known, or methods which estimate/analyze the power consumption of the entire CPU as a whole, should be used. The measurement based analysis method described in this paper is such a method. It implicitly accounts for all correlations between internal modules, since it is based on measurements made at the boundaries of the processor.

13 Conclusions

This paper describes the application of a new power analysis technique for analyzing the power consumption of the Fujitsu SPARClite MB86934, a RISC processor. This technique had earlier been applied to the Intel 486DX2, a CISC processor. The successful application of this technique for both these processors points to its general applicability for other processors. This study reveals that the basic instruction-level power model of the '934 is very similar to that of the 486DX2. This power model can be used to effectively evaluate the power cost of software, without requiring knowledge of the proprietary lower level details of the processor. The results of the analysis also provide valuable information about the power consumption in the '934. Besides suggesting several ideas for the design of power efficient software, this information reveals other avenues for power reduction in the processor's design.

Acknowledgment: The authors would like to thank D. Maheshwari, H. Kotcherlakota, M. Somasundaram, A. Watanabe, and B. McKeever of Fujitsu Microelectronics Inc. for their help with the experimental setup and for technical discussions.

References

[1] V. Tiwari, S. Malik, and A. Wolfe. "Power analysis of embedded software: A first step towards software power minimization". In *IEEE Trans. on VLSI Systems*, vol. 2, no. 4, pages 437–445, Dec. 1994.

[2] V. Tiwari, S. Malik, and A. Wolfe. "Compilation techniques for low energy: An overview". In *Proc. Symp. on Low Power Electronics*, pages 38–39, San Diego, CA, Oct. 1994.

[3] Mike T.-C. Lee, V. Tiwari, S. Malik, and M. Fujita. "Power analysis and low-power scheduling techniques for embedded DSP software". To appear in *8th Int'l Symp. on System Synthesis*, Cannes, France, Sept. 1995.

[4] V. Tiwari, S. Malik, and P. Ashar. "Guarded evaluation: Pushing power management to logic synthesis/design". In *Proc. Int'l Symp. on Low Power Design*, pages 221–226, Dana Point, CA, April 1995.

[5] Fujitsu Microelectronics Inc. *SPARClite Embedded Processor User's Manual.* 1993.

[6] Fujitsu Microelectronics Inc. *SPARClite Embedded Processor User's Manual: MB86934 Addendum.* 1994.

[7] C. Feigel and M. Enfield. "Fujitsu extends SPARClite family". In *Microprocessor Report*, June 1994.