Microprocessor Power Analysis by Labeled Simulation[1]

Cheng-Ta Hsieh, Lung-sheng Chen, Massoud Pedram
Dept. of EE-Systems
University of Southern California
Los Angeles, CA 90089

Abstract

In many applications, it is important to know how power is consumed while software is being executed on the target processor. Instruction-level power microanalysis, which is a cycle-accurate simulation technique based on instruction label generation and propagation, is aimed at answering this question for a superscalar and pipelined processor. This technique requires the micro-architectural details of the CPU and provides the power consumption of every module (or gate) for each active instruction in each cycle. To validate this approach, a Zilog digital signal processor core was designed by using a 0.25μ TSMC cell library, and the power consumption per instruction was collected using a Verilog simulator specially written for the DSP core.

1 Introduction

Given the micro-architectural description of a target processor and some application program to be executed, it is usually useful to know which modules (or gates) consume the most power and under what input data or internal state conditions. For example, a common question is how to automatically identify and eliminate unwanted power consumption during the program execution by hardware control (e.g., clock gating) and/or software optimization (e.g., compilation). To fully answer the question, we need to know the cycle-accurate power consumption of each individual module (or gate) in the processor due to the execution of each instruction. We refer to this kind of analysis as *power microanalysis*, and present a simulation-based strategy to achieve it. Microprocessor designers can use the power microanalysis report to improve the power efficiency of a proposed instruction set architecture. Similarly, compilers can use power microanalysis to reduce the energy cost of an application program running on the target microprocessor by performing high-level transformations or low-level code generation.

Power microanalysis reports can also be useful in generating an accurate *power macromodel* of a processor [1][2]. A power macromodel is usually trained by running a number of instruction traces and studying the resulting power dissipation profile in the target circuit. Without an accurate power consumption breakdown for each instruction in the pipeline, the various power dissipation effects have to be averaged out. These power effects include, for example, the power consumption caused by pipeline stalls, pipeline flushes, and cache misses. Furthermore, in some cases, power may be dissipated due to unwanted operations (this is mainly because of poor design practices). For example, the input operands of the multiplier may change even when the executed instruction is not a multiplication instruction, which in turn causes extra power consumption. If this kind of effect is not accurately modeled during the power macromodel construction, it will be treated as a random statistical variation at best, which will then increase the error of the power macromodel. The power microanalysis technique proposed here can be quite valuable in constructing an accurate instruction-level power macromodel because it provides information about the power consumption caused by each instruction in each gate in the circuit while accounting for pipeline stalls, pipeline flushes, and cache misses.

The instruction execution in a modern CPU has the following characteristics:

- Multiple instructions are executed concurrently in the processor e.g., very large instruction word (VLIW) and superscalar architectures.
- Interactions between the instruction and the architecture can cause significant power consumption e.g., branch misprediction.
- Interactions among the instructions greatly contribute to the overall power consumption of the CPU (e.g., data dependency and resource contention).

Because of this complexity, it is very difficult to automatically generate the equation form of the instruction–level power macromodel or even perform the calibration process (i.e., calculate the macromodel equation coefficients) for a given power macromodel equation form. For example, in [2], the macromodel equation is manually designed and then automatically calibrated by measuring the power dissipation of a set of specially designed instruction traces.

Running an application program that is simply a loop with only one or at most two types of instructions typically generates the trace. The measured instruction power is

[1] This research is sponsored in part by contract number DAAB07-00-C-L516 from the DARPA PAC/C program.

called base cost, which is used for instruction-level macromodel training. The inter-instruction temporal effects can also be calculated and included in the model equation using these training traces. However, the model is still too simple to capture the actual CPU power dissipation. More precisely, because of the lack of detailed (module-level or gate-level) knowledge about the power consumption of each individual instruction in each clock cycle, the following difficulties arise:

- The initial power macromodel equation form (i.e., the number and meaning of different terms and the way they are combined) has to be input by the designer based on his experience and knowledge about the microprocessor architecture. If the initial form is incomplete or inappropriate, the accuracy of the power macromodel predictions will be adversely affected.
- It is very difficult to ensure proportionate coverage of the various power consumption factors in the processor (e.g., instruction mix and order, pipeline effects, and branch handling policy) with the macromodel equation. The calibration step requires a detailed simulation of a very large number of complex instruction traces (i.e., with a number of instruction types and exercising different hardware conditions in the pipeline) to ensure correct calibration of the macromodel coefficients in order to cover instruction correlations, data dependencies, various architectural effects and scenarios. In contrast, with the aid of a power microanalysis report, the macromodel calibration process would be a lot simpler since the required information would be available.

Our technique handles both super-scalar and pipelined processors. However, it is not intended to replace the works that are exemplified by [2] and [3]. Please refer to [4][5] for detailed reviews of high-level (including software-level) power estimation and optimization.

An instruction is *active* if it is being executed in the instruction pipeline of a given microprocessor. The power microanalysis for the microprocessor can be defined as identifying what active instructions cause the power consumption for each gate in a register transfer-level (RTL) description of the processor. A naïve approach simply assumes that the power consumption of every gate is caused by all of the active instructions. In this paper, we present a more sophisticated and significantly more accurate simulation-based technique called *Labeled Simulation* for evaluating the power consumption of the microprocessor. Note that although a detailed RT-level description of the micro-architecture is assumed in this paper, power microanalysis can be performed even when some parts of the processor are behaviorally specified as long as the complete model can be simulated.

This paper is organized as follows. Section 2 provides details of our proposed power microanalysis technique. Section 3 describes the DSP core used a design example. Section presents our experimental setup and results. Conclusions are given in Section 5.

2 Power Microanalysis

2.1 Problem formulation

Assume that there are n gates, $g_1 \dots g_n$, in the circuit description of the target processor, and k instructions, $I_1 \dots I_k$, are active in the processor in a certain clock cycle. We find a labeling $L_i = \{I'_1, I'_2, I'_3 \dots\}$ for each gate g_i, $i = 1 \dots n$, such that the energy consumption of g_i in the current clock cycle is caused by instructions in L_i. If L_i is empty, the energy consumption of g_i is not caused by any particular instruction and is considered the *intrinsic* energy consumption of the processor (e.g., the energy consumption of the instruction cache is not caused by an individual instruction). If L_i contains multiple instructions, the energy consumption of g_i is caused by and equally attributed to all of the instructions in L_i.

Define $G(I)$ as a set of gate indices such that instruction I belongs to the label of each gate according to the indices in $G(I)$.

$$G(I) \equiv \{ i \mid I \in L_i \}$$

The energy consumed by instruction I in the current clock cycle is:

$$E(I) = \frac{1}{2} \sum_{j \in G(I)} \frac{1}{|L_j|} C_j V_{dd}^2 sw_j$$

where $sw_j = 1$ if wire j switches, otherwise $sw_j = 0$; and C_j is the effective capacitance of gate j. The total energy dissipation of an instruction I for the program being evaluated is calculated by the summation of $E(I)$ over clock cycles when I is active (non-empty $G(I)$). Note that the labels need to be updated every clock cycle while the instruction is propagating through the pipeline.

Consider a simple MIPS-like instruction pipeline with five stages, and assume that there is no feedback path between any two pipelines. In this case the labeling problem is solved by propagating the labels from one pipeline stage to the next through the labeling network, which is equivalent to RT or gate level logic network of the processor. The on-chip memory is treated in the same way as the flip-flops because its functionality is the same as that of the flip-flops (registers). The labeling can be derived by labeling the wires connected to the instruction memory (IM) as newly fetched instruction I_i and propagating the labels in the network according to these rules:

Combinational gate: If we assume that the instruction pipeline has no feedback, the input labels of a gate will not contain different instructions. We simply pass the input label to the output.

Flip-flop: At the positive or negative clock edge, we label the flip-flop itself with its input instruction label.

2.2 Labeling network

To initialize the labeling propagation, the first task to undertake is identifying the label sources and sinks for label propagation.

2.2.1 Source and sink

Definition: The source refers to the set of gates (or wires) from which the labels are originated.

Definition: The sink refers to the set of gates (or flip-flops) where the instruction label is dropped.

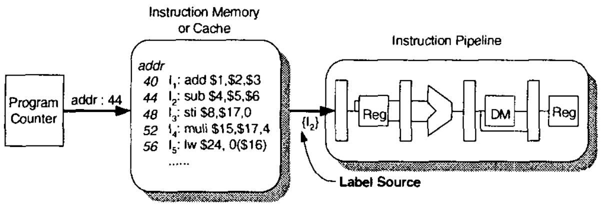

Figure 1 Instruction memory as the label source

When a processor fetches an instruction *I* from the external memory, cache, or on-chip memory, the set of wires connected to the read port are labeled as *L={I}*. In Figure 1, for example, the instruction addressed by the program counter, whose content is 44, is fetched, and the instruction bus is labeled as $\{I_2\}$. Sometimes, the instruction fetch unit is designed to fetch *k* instructions in one clock cycle (e.g. VLIW machine), and then the read port of the IM (or cache) is labeled by those instructions, $\{I_1,\ldots,I_k\}$. Note that for some advanced processors, there may be multiple IMs in the system. Therefore, the label source may not be unique. The new instruction labels continuously flow into the system from the label sources in every clock cycle.

The next question is when we should stop propagating an instruction label or drop an instruction from a label in the network. The instruction label, which is stored in a flip-flop, is only removed when it is not transferred to any other flip-flop in the processor (including the flip-flop where it is stored). For example, if an instruction label is propagated to the last stage of the pipeline and if this label is not transferred to any of the data paths in the processor, it will be overwritten by another label in the next clock cycle.

When an instruction label is transferred into the on-chip memory or register file, the question of whether we should label the memory elements inside the memory file or the register file arises. Note that if the labels are not removed in these memory elements (flip-flops or memory cells), the number of distinct instructions in all of the labels in a given clock cycle may be larger than the number of pipeline stages. As an example, in Figure 2, a 'mov' instruction (denoted as I_{mov}) finishes its job after writing the immediate value 100 to a register. Then instead of propagating $\{I_{mov}\}$ after we write to the register file, the label should be dropped because the 'mov' instruction never uses the written data again. After a number of clock cycles, the register content may be used by another instruction 'add \$3, \$1, \$n.' However, the energy consumption induced by the newly fetched register content (\$1) should be attributed to the instruction that fetched the register (i.e., *add \$3, \$1, \$n*), not to the instruction that wrote it (i.e., *mov \$1,100*). Similarly, when a 'store' instruction writes some data into the data memory, it never uses the memory content again, and the label should be dropped in the memory.

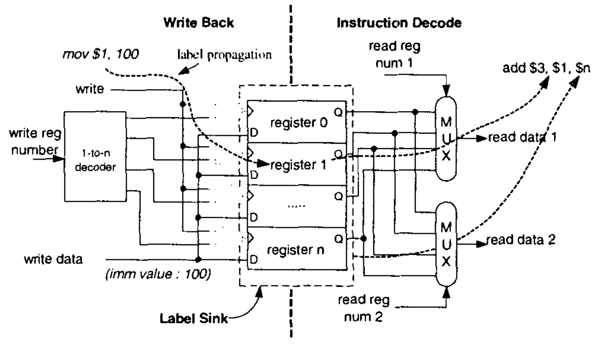

Figure 2 A 2-read, 1-write register file

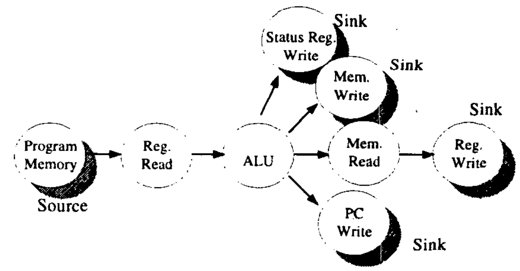

Figure 3 Instruction label flow chart for MIPS

In MIPS architecture, the register file, data memory, status register, and program counter are marked as the label sinks. Note that the contents of the flip-flops or memory elements that are marked as label sinks may affect the power consumption of other modules in the system. In general, labels are dropped one clock cycle after when they reach a sink. It is also possible that instructions require different definitions for label sink locations that may conflict with each other. If such a conflict occurs, instruction *I* is dropped from the label if it reaches a node where the node is defined as a sink for *I*.

Figure 3 shows the journey of an instruction in the pipeline of a MIPS architecture. The lifetime of an instruction starts from the source and ends at the sink (if it is not discarded in the middle, e.g., due to a pipeline flush). At each clock cycle, the instruction label moves toward the label sink and activates some control signals or simply stays in the same place in the case of encountering a control or a data hazard.

2.2.2 Propagation rule

After synthesizing and mapping an RTL design to a standard cell net-list, the instruction label starts from instruction memory and propagates through nets and cells under a specific propagation rule. Each type of standard cell should have an associated propagation rule. For a simple inverter, we propagate its input label to its output.

The notation for a 2-input gate is shown in Figure 4; in_1 and in_2 denote the *logic values* of the inputs. The rule is: $L_{out}=L_1$ if $L_2=\{\}$ and $L_{out}=L2$ if $L_1=\{\}$. In the case in which both L_1 and L_2 are non-empty, the rule is as follows: L_1 will be propagated if in_2 has a non-controlling value, and L_2 will be propagated if in_1 has a non-controlling value.

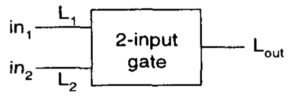

Figure 4 Labeling for 2-input gates

For an OR gate, the propagation rule is summarized below:

in1	*in2*	L_{out}
0	0	L_1+L_2
1	0	L_1
0	1	L_2
1	1	L_1+L_2

Notice that L_{out} can be statically decided if $L_1=\{\}$, $L_2=\{\}$ or $L_1=L_2$. The propagation rule table for an AND gate is similar except that the '10' input combination causes propagation of L_2 whereas "01' combination propagates L_1. For an XOR gate, L_1+L_2 is propagated to the output for all input combinations. For a combinational circuit cell like a multiplexer (MUX), the propagation rule table derived from its equivalent Boolean implementation in terms of AND and OR gates should be consistent with the table constructed by a direct derivation. The logic values of the multiplexed inputs are not important in the case of the MUX cell. Instead, the select signal plays a major role. Consider a 2-to-1 MUX with select logical value zero for label zero (L_0) input and one for label one (L_1) input. The following table shows the propagation rule for the MUX:

select	$L_s==\phi$	$L_0==\phi$	$L_1==\phi$	L_{out}
0	X	1	X	L_s
0	1	0	X	L_0
0	0	0	X	(L_s+L_0)
1	X	X	1	L_s
1	1	X	0	L_1
1	0	X	0	(L_s+L_1)

where "$L_i==\phi$' is '1' if L_i is empty, and '0' if not empty. 'X' denotes a don't-care condition. By observation, L_{out} can be statically decided if exactly one of the $\{L_0,L_1,L_s\}$ is not empty or L_0 equals L_1.

For the '+' operation, we give two different definitions.

***Definition:** Priority Rule (Time-Stamp Rule)*

If $L_1=\{I_i\}$ and $L_2=\{I_j\}$, then $L_1+L_2=\{I_{max(i,j)}\}$. Only the instruction that is fetched later (i.e., it has a larger time stamp) is kept in the merged label. Therefore, the labels after the merge contain at most one instruction. In this rule, the instruction that is fetched later always assumes the responsibility for the power consumption when multiple instructions are propagated to the same wire.

***Definition:** Union Rule*

$L_1+L_2=L_1\cup L_2$. In this rule, instructions that run into each other assume equal responsibility for the power consumption.

As mentioned in the problem formulation, the input labels of a gate do not contain different instructions because of the assumption that there is no pipeline feedback. However, for a modern microprocessor, a resource hazard is resolved automatically with a hazard detection unit, e.g., the pipeline-stall and flush mechanism or a data-forwarding unit. Those abilities require feedback information between different pipeline stages. Hence, the input labels should be annotated with different instructions. Several architectural patterns must be defined and analyzed for a specific microprocessor in order to make sure that the propagation rules of the cells satisfy all the architectural patterns.

2.3 Architecture patterns

We define an architecture pattern to have three fields as follows:

1. **Name** is a handle that we can use to describe the intended architecture effect (e.g., control hazard).
2. **Description** explains how the pattern is caused and how the processor reacts to the pattern.
3. **Required Rule** specifies how the propagation rule should work in response to the pattern.

The most common architecture patterns, pipeline-stall, data forwarding, and pipeline-flush, will be given as examples. Each pattern is caused by a certain architectural effect, and the related control circuitry will be explained. The required rule is given based on the specific control circuitry. The example is, however, representative, and other causes of an architecture pattern will give rise to similar rules. Furthermore, the circuit implementation may vary for different processors, but the underlying structure for the instruction dispatch and routing will be similar.

2.3.1 Pipeline-stall pattern

Name*:* Pipeline-stall

Description: A data hazard usually occurs when an operation needs operands that are not computed or have been computed but are not yet available to the instruction. This is also called the "read-after-write" hazard. There are many other types of data hazards, depending on the target architecture. In particular, the super-scalar processors that perform speculative execution have complex control logic or architecture to make sure that the program works the same as when it is run on a scalar machine. Such complex architectures usually generate a lot of data hazards.

Figure 5 shows how the pipeline stall architecture injects bubbles into the instruction pipeline. If no hazard is detected, the MUX1/MUX2 select line is '0,' and the instruction pipeline works as a streamlined pipe. If a hazard is detected, the MUX1/MUX2 select line equals '1,' and I_4 is retained in flip-flops FF1, and a bubble is injected to flip-flops FF2. The hazard detection logic can be implemented as in Figure 6, where each of the '==' gates compares the inputs and produces '1' if the two inputs are equal. Figure 6 shows only part of the circuit; a complete hazard detection unit should compare both source operands of I_4 with the destination operands in the pipeline.

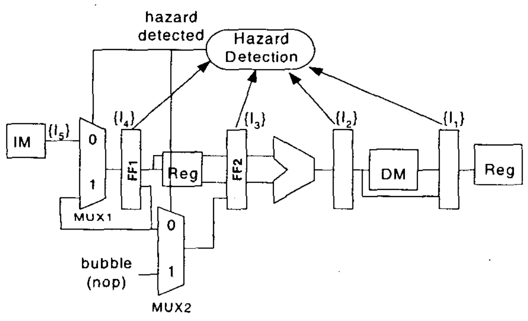

Figure 5 Pipeline-stall architecture

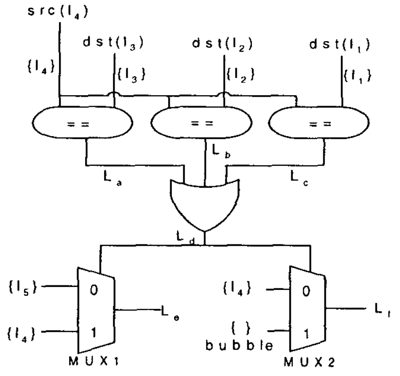

Figure 6 Hazard detection logic

Required Rule (c.f. Figure 6):

L= '1' denotes the labeled wire with logic value '1.'

- L_a={I_4,I_3}. L_b and L_c follow similar rules.
- L_d should be the minimal set while satisfying the following rules:
  - o $L_d \supseteq L_a$ if L_a= '1.'
  - o $L_d \supseteq L_b$ if L_b= '1.'
  - o $L_d \supseteq L_c$ if L_c= '1.'
- L_e={I_4}+L_d and L_f=L_d if L_d='1'. Otherwise L_e={I_5}+L_d and L_f={I_4}+L_d.

2.3.2 Data-forwarding pattern

Name: Data-forwarding

Description: Instead of stalling the pipeline to avoid data hazards, a data-forwarding architecture can be used to reduce the "read-after-write" hazard. In Figure 7, such an architecture for the MIPS pipeline is shown. When there is read-after-write dependency between I_3 and I_2 or I_3 and I_1, the operands required by I_3 can be directly forwarded from the computed result of I_2 or I_1. A forwarding unit can be implemented as shown in Figure 8.

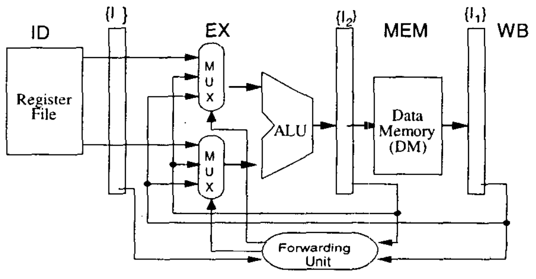

Figure 7 Data-forwarding architecture

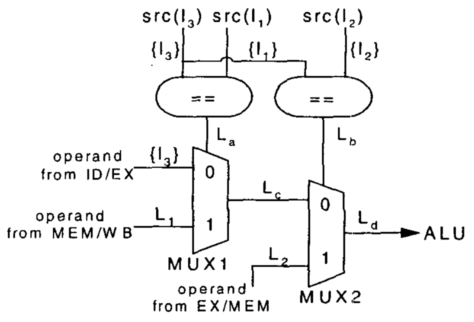

Figure 8 Data-forwarding control circuitry

Required Rule: (c.f. Figure 8):

- L_a={I_3,I_1}
- L_b={I_3,I_2}

- $L_c=\{I_1\}+\{I_3\}+L_1$, if $L_c=$ '1', otherwise $L_c=\{I_3\}+\{I_1\}$
- $L_d=\{I_3\}+\{I_2\}+\{I_1\}$, if $L_a=$ '0' and $L_b=$ '0'
  $L_d=\{I_3\}+\{I_2\}+\{I_1\}+L_1$, if $L_a=$ '1' and $L_b=$ '0'
  $L_d=\{I_3\}+\{I_2\}+L_2$, if $L_b=$ '1'

Please note that the feedback path will not cause an infinite L_d length because of the *priority rule* or *union rule* applied to the '+' operation.

2.3.3 Pipeline-flush pattern

Name: Pipeline-flush

Description: A control hazard is usually caused by branch instructions. A branch instruction may change the target instruction address to be fetched next. The target address may not be known at the time that the next instruction is fetched. Therefore, the control logic needs to monitor these situations to make sure that the processor works correctly with or without the branch hazards.

Figure 9 shows an example of the branch hazard taken from [6]. The instruction at address 40 compares the register content of \$1 and \$3 and jumps to address 72 (40+28) if \$1=\$3. There are two ways to handle the control hazard: Always Stall and Assume Branch Not Taken.

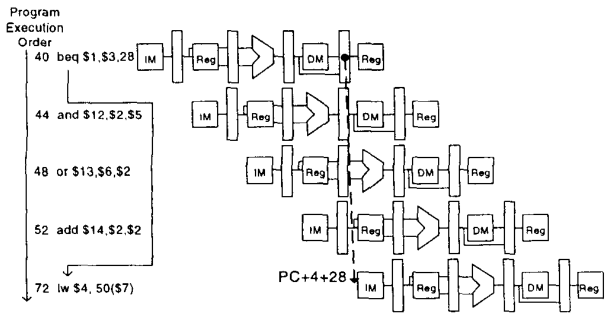

Figure 9 Branch hazard example

Always Stall: This is the simplest way to handle the branch hazard. Each time a branch instruction is encountered, the control unit simply stalls the instruction pipeline by injecting a bubble. The control circuit can be implemented similarly to the one shown for data hazard detection. The Always Stall strategy does not cause pipeline flush.

Assume Branch Not Taken: Instead of stalling the pipeline immediately, we continue the execution by assuming that the branch will not be taken. If the branch is untaken, the instruction pipeline keeps running without any interruption. If the branch is taken, the instructions that are being fetched and decoded must be discarded. To discard the instructions, we need to change the control code of the instruction in IF, ID, and EX stages (see Figure 10) in such a way that the instruction will not write back any result to the register file or the memory. The control circuit can be implemented as shown in Figure 11. Note that the status register, which decides whether the branch is taken or not, can be set by an earlier instruction and is marked by an empty label, or it is set by the branch instruction and is labeled as $L_I=\{I_{branch}\}$ where I_{branch} is the branch instruction in the memory stage.

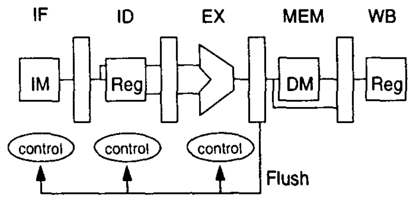

Figure 10 Branch hazard control circuit

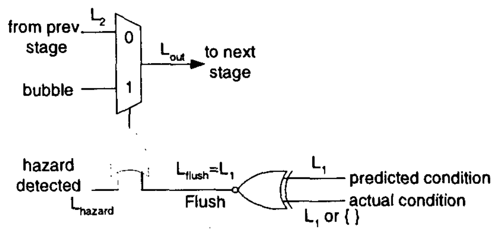

Figure 11 Control circuit for pipeline flush

Required Rule: (c.f. Figure 11)

- $L_{out}=L_1$ if pipeline is flushed due to the branch misprediction.
- $L_{out}=L_2$ if both "hazard detected" and "Flush" are de-asserted.
- $L_{out}=L_{hazard}$ if "hazard detected" is asserted.

3 Design of a DSP core

Instruction-flow-driven power analysis is also useful for power analysis in digital signal processors. Usually the computational resources in a digital signal processor are distributed, and multiple on-chip buses are used to maximize the throughput. Consequently, it is even more difficult to manually perform the labeling. In this section, we use a Zilog voice processor [7] as the DSP example for microanalysis.

In this processor, there are two on-chip RAM banks: RAM0 and RAM1, a stack, and several distributed registers: X, Y, P, and an Accumulator. The lower 64 words of the on-chip RAM can also function as registers. To perform a multiplication, two operands are simultaneously loaded from RAM0 and RAM1 and then stored in X and Y registers within one clock cycle. In the instruction set, X and Y can also function as general-purpose registers to move data around. Note that the data outputs of X and Y registers are directly tied to the inputs of the multiplier. Therefore, if a 'mov' instruction moves the data from Accumulator to X without the need to perform a multiplication, then the multiplier will still dissipate (waste) power because its inputs change. Our labeling scheme could simply propagate non-multiple instruction to X and Y and capture the wasted power. A similar problem can be automatically detected for the ALU inputs. For example, if we want to perform the multiplication instruction and the result is written into register P, then the value change in P may be passed on to the ALU and subsequently cause unnecessary power consumption in the ALU. This can also be detected by label propagation.

Another potential problem is that the select line of the MUX may change value even when no ALU instruction is being executed. This problem may be caused by, for example, poorly designed decoder logic. By labeling, we can easily identify the specific part of the instruction decoder that causes this problem. This last case also shows that the instruction-labeling scheme can help debug and verify the hardware early in the design process.

We have designed a DSP core, which is compatible with the Z89C00 instruction set [8], in Verilog language. The Z89C00 DSP instruction set, consisting of 30 basic instructions, is optimized for high code density and reduced execution time. Single-cycle instruction execution is possible on most instructions, including multiplication and I/O operations. There are 9 different addressing modes, which enables high code density.

4 Experimental Results

4.1 DSP core mapping

The DSP core is mapped to a TSMC Process-Perfect Library [9] with Synopsys Design Compiler v. 1999.05 [10]. The RAM0, RAM1, and instruction memory are not mapped and remain in behavioral model for the purpose of fast RTL simulation. The power consumption inside memory may be captured or estimated separately if a more accurate power model is needed. We construct the propagation rules with *Union Rules* for all of the library cells and verify them with several architectural patterns.

4.2 RTL simulator and label propagation engine

A Verilog simulator [11] is used for RTL simulation. A label propagation engine is built with the Verilog Procedural Interface [12], which provides a mechanism to access the internal simulation data of the Verilog simulator. The engine performs label propagation and generates an instruction power consumption profile (cf. Figure 12). We first simulate one clock cycle and record the switching activity of each wire in the mapped net list. We do label propagation at the end of the clock cycle. Note that the logic value of each net, which is utilized to perform label propagation, should therefore remain unchanged until the end of the current clock cycle.

The energy consumption is then calculated in the third step. Energy dissipation is dependent on the power-supply voltage, the switching activities, and the internal and output load capacitances. The energy dissipated in each cell in each cycle is calculated by the following equation:

$$E = E_{Internal} + E_{External}$$

where $E_{Internal}$ denotes the energy dissipation in the internal capacitances of the cell due to input transitions and $E_{External}$ denotes the energy dissipation due to transitions at the output of the cell. This includes the effects of both input pin capacitances of the fanout gates and the routing capacitance of the net connecting the cell and its fanout gates. Obviously the power dissipation is the product of the energy consumption and the clock frequency.

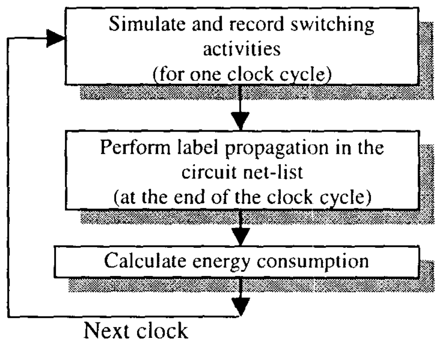
Simulate and record switching activities (for one clock cycle)
Perform label propagation in the circuit net-list (at the end of the clock cycle)
Calculate energy consumption
Next clock

Figure 12 Simulation workflow

The TSMC data book provides $E_{Internal}$ and input capacitance values for all cells. For the wire capacitance, we simply assume that it is proportional to the fanout count of the driver. Note that the first part of the equation is the power consumption of the library cell, which is made up of all the instruction labels of its output pins (nets). The second part is the total power consumption of the output nets of the cell in the current clock cycle. By iterating the cell instances and summing up their power dissipation P, we calculate the total circuit power consumption.

- Target application on Zilog DSP processor

Currently, we do not have a C/C++ compiler and assembler for the Z89C00 DSP instruction set. Because of the lack of high-level language utilities, it is impossible for us to build complex DSP applications for our testing purposes. Instead, five simple programs were written in assembly language and directly translated into the binary code. This process was cumbersome but served our objective.

4.3 Simulation results

The Zilog Z89C00 Instruction Set is categorized into 5 instruction classes, NON, SL, MAC, CTRL, CAS and ALF. The NON-instruction is for the background power consumption, which cannot be attributed to any instruction class. SL is for load and store instructions including different addressing modes. MAC is for simultaneous multiplication and addition instructions. CTRL is for control related instructions. CAS is for comparison and integer arithmetic instructions. ALF is for logical operation instructions. Five simple programs are used as target applications on the Zilog DSP core. The energy consumption of an instruction class for each program is given in Figure 13 to Figure 17. The energy consumption does not include the dissipation in the instruction memory and data memory. The average energy for the 'NON' class is the background energy divided by the instruction count.

Instruction Class	**Average Energy(10^{-8}J)**	**Instruction Count**
NON	0.0053	-
SL	0.0262	83
MAC	0.0513	132
CTRL	0.0101	30
CAS	0.0147	7
ALF	0.0198	14

Figure 13 Instruction class energy for program 1

Instruction Class	**Average Energy(10^{-8}J)**	**Instruction Count**
NON	0.0046	-
SL	0.0128	70
MAC	0.0649	35
CTRL	0.0155	125
CAS	0.0111	8
ALF	0.0124	27

Figure 14 Instruction class energy for program 2

Instruction Class	**Average Energy(10^{-8}J)**	**Instruction Count**
NON	0.0071	-
SL	0.0193	20
MAC	0.0674	98
CTRL	0.0138	58
CAS	0.0265	142
ALF	0.0136	33

Figure 15 Instruction class energy for program 3

Instruction Class	**Average Energy(10^{-8}J)**	**Instruction Count**
NON	0.0054	-
SL	0.0210	61
MAC	0.0477	10
CTRL	0.0144	78
CAS	0.0120	17
ALF	0.0188	104

Figure 16 Instruction class energy for program 4

Instruction Class	**Average Energy(10^{-8}J)**	**Instruction Count**
NON	0.0058	-
SL	0.0190	56
MAC	0.0520	42
CTRL	0.0171	70
CAS	0.0201	33
ALF	0.0179	42

Figure 17 Instruction class energy for program 5

5 Conclusions

An instruction-flow-based power analysis technique was proposed to accurately calculate power dissipation induced by a certain instruction running on a target processor. The proposed algorithm attributes the power consumption of each gate within the processor to the instructions that are executed in the instruction pipeline. As a result, the power microanalysis enables the processor architect or designer to identify the instructions that consume a lot of power or, more importantly, waste power. When some component does not work as expected from a power, performance, or functionality perspective, the instruction-labeling scheme can help designers trace the problem back to the instructions that caused the problem. The proposed technique also helps in synthesizing an instruction-level power macromodel, which can later be used by a compiler to generate power-efficient executables.

REFERENCES

[1] C-L. Su, C-Y. Tsui, and A.M. Despain, "Low Power Architecture Design and Compilation Techniques for High Performance Processors," *Proc. of IEEE COMPCON*, pages 489-498, 1994.

[2] V. Tiwari, S. Malik, A. Wolfe, and T.C. Lee, "Instruction Level Power Analysis and Optimization of Software," *J. of VLSI Signal Processing*, Vol. 13, No. 2/3, pages 223-238, Aug. 1996.

[3] W. Ye, N. Vijaykrishnan, M. Kandemir and M.J. Irwin, "The design and use of SimplePower: a cycle-accurate energy estimation tool," *Proc. of 37th Design Automation Conference*, pages 340-345, Jun. 2000.

[4] E. Macii, M. Pedram and F. Somenzi, "High level power modeling, estimation and optimization," *IEEE Trans. on Computer Aided Design*, Vol. 17. No. 11, pages 1061-1079, Nov. 1998.

[5] K. Roy and S. C. Prasad, *Low Power CMOS VLSI Circuit Design*, John Wiley and Sons, 2000.

[6] D. A. Patterson and J. L. Hennessy, *Computer Organization & Design, The Hardware/Software Interface*, Morgan Kaufmann, 1994.

[7] Z89223/273/323/373 16-Bit Digital Signal Processors with A/D Converter: Production Specification, URL: http://www.zilog.com.

[8] Zilog Z89C00 Instruction Set, URL: http://www.zilog.com.

[9] TSMC Process-Perfect Library Data Book 2.5-Volt Standard Cell TSMC 0.25μm Process, Jan. 1999.

[10] Design Compiler Reference Manual, v. 1999.05, Synopsys, 1999.

[11] Verilog-XL Reference Manual, Cadence, 1998.

[12] VPI User Guide and Reference, Cadence, 1998.