2006

A compiler-based leakage reduction technique by power-gating functional units in embedded microprocessors

Soumyaroop Roy
University of South Florida

A Compiler-Based Leakage Reduction Technique by Power-Gating Functional Units in Embedded Microprocessors

by

Soumyaroop Roy

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Engineering
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Co-Major Professor: Srinivas Katkoori, Ph.D.
Co-Major Professor: Nagarajan Ranganathan, Ph.D.
Soontae Kim, Ph.D.

Date of Approval:
October 20, 2006

Keywords: Supply-control, Sleep instruction, Low power design, Power-aware ARM processor, SimpleScalar ARM

DEDICATION

To my parents

ACKNOWLEDGEMENTS

I would like to thank Dr. Ranganathan for being a great mentor and Dr. Katkoori for all the support that he extended. I would like to acknowledge the helpful suggestions that Dr. Kim provided for this work. Finally, I would like to thank the members of VCAPP laboratory, especially Jared and Narendra, for indulging in brainstorming sessions involved during this research effort.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

A COMPILER-BASED LEAKAGE REDUCTION TECHNIQUE BY POWER-GATING FUNCTIONAL UNITS IN EMBEDDED MICROPROCESSORS

Soumyaroop Roy

ABSTRACT

Power-gating is a technique investigated widely for reducing leakage energy in the functional units of microprocessors at the architectural level. Effective power-gating involves deactivating idle functional units for sustained periods incurring little or no performance degradation. Accurate prediction of long idle periods is essential, which, in turn, depends on the application program characteristics. In this thesis, we propose a compiler-based leakage reduction technique for embedded architectures by exploiting the well-known attributes of embedded applications, namely, small code size and intensive loops. From the control flow graph (CFG) representation of the source program, we construct a forest of *loop hierarchy trees* (LHTs), which capture the nesting loop properties of the program. As an LHT satisfies the partial ordering on the loop nesting, we exploit this property to identify maximal subgraphs (of functional unit idleness) in the original program. For each subgraph so found, a *sleep* instruction is introduced at the entry point of the corresponding code segement, thus optimizing the number of sleep instructions. The sleep instruction has one operand, a bit-vector comprised of ON/OFF control bits for all functional units in the data path. Our target architecture is a modified ARM processor model comprising of functional units with power-gating ability. We obtained an average leakage energy reduction of 34.1% for 12 benchmarks chosen from the MiBench suite, with range of 19.5% and standard deviation of 6.5%.

CHAPTER 1

INTRODUCTION AND RELATED WORK

As the design complexity and the device density of the VLSI circuits have increased, the power consumption of the circuits has become one of the primary design concerns. High power consumption impacts the engineering feasibility of battery-powered portable devices to a large extent. The designers have to make a tradeoff between the size of the battery packs and the operating life of the devices. These issues have forced designers to pursue low-power desgin methodologies very aggresively [1].

Although techniques to reduce dynamic power in circuits have been worked on for long, with shrinking technologies, the leakage power dissipation in circuits has become substantially large. In fact, it has been established that a chip's leakage power increases 5 times each generation whereas the dynamic power remains constant [1]. Due to this, the energy consumed in circuits due to leakage power is becoming a large component of the total energy consumed by circuits in the current CMOS technologies. Since the total energy consumed by a circuit is indicative of its battery life, various leakage power reduction techniques have been employed to reduce the total leakage power consumption of VLSI circuits.

In the recent years, embedded microprocessor domain is a rapidly growing segment in the microprocessor industry [2]. Although many of the embedded microprocessors are small and inexpensive microcontrollers, their combined sales account for nearly half of all the microprocessor revenue. The applications range from sensor systems in household appliances to smart cellular phones that have the functionality comparable to that of a desktop machine. Since a lot of these applications require portability, the power consumption of these microprocessors is a major design constraint. Due to this factor, the design considerations

for embedded processors are focused on both performance and power instead of only on performance, as has been in traditional high performance desktop systems [3].

In this work, we exploit circuit level leakage power reduction techniques to reduce leakage energy consumption in embedded microprocessors at the architectural-level.

1.1 Leakage Power Reduction Techniques

In modern digital CMOS integrated circuits, there are three major components of power consumption: dynamic switching power, short circuit power, and leakage power. Dynamic switching power is the component that dominates the power consumption in these circuits. It is a result of charging and discharing of gate capacitances during signal switching and is given by:

$$P_{switching} = \sum_{i} \alpha C_{load_i} V_{dd}^2 f_{clk} \tag{1.1}$$

where α is the switching activity at node i, C_{load_i} is the load capacitance at node i, V_{dd} is the supply voltage, and f_{clk} is the clock frequency.

The subthreshold leakage current varies exponentially with the threshold voltage, V_T, which is given by the expression:

$$I_{subth} = Ae^{\frac{q}{n'kT}(V_G - V_S - V_T - \gamma' V_S + \eta V_{DS})}(1 - e^{\frac{-qV_{DS}}{kT}}) \tag{1.2}$$

where, $A = \mu_0 C_{ox} \frac{W_{eff}}{L_{eff}} (\frac{kT}{q})^2 e^{1.8}$, C_{ox} is the gate oxide capacitance per unit area, μ_0 is the zero bias mobility, n' is the sub-threshold swing coefficient of the transistor, V_T is the zero bias threshold voltage, γ' is the linearized body effect coefficient, and η is the Drain Induced Barrier Lowering (DIBL) coefficient [4].

The most effective way to reduce the dynamic power of a circuit is to reduce the supply voltage, V_{dd}. However, reduction in the supply voltage degrades the performance of the circuit. The propagation delay of a transistor is given by:

$$t_{delay} \propto \frac{C_{load} V_{dd}}{K(V_{dd} - V_T)^{\alpha}} \tag{1.3}$$

where, C_{load} is the load capacitance, K is a constant dependent on the process and the gate size, V_T is the threshold voltage of the transistor, and α models short channel effects and takes any value between one and two depending on the channel length [5].

To improve the performance of circuits at low supply voltages, both V_{dd} and V_T can be scaled down. The leakage current increases when V_T is scaled down and the dynamic power decreases with V_dd is scaled down. The increase in subthreshold leakage power is small compared to the quadratic reduction in dynamic power supply with small amounts of scaling. However, with extreme scaling of V_{dd} and V_T, the increase in leakage energy starts to dominate the reduction in switching energies. This indicates that there is an optimum energy point for a given target frequency. However, the V_{dd} and V_T, corresponding to the optimal energy point, are significantly lower than the typical threshold voltage levels in the

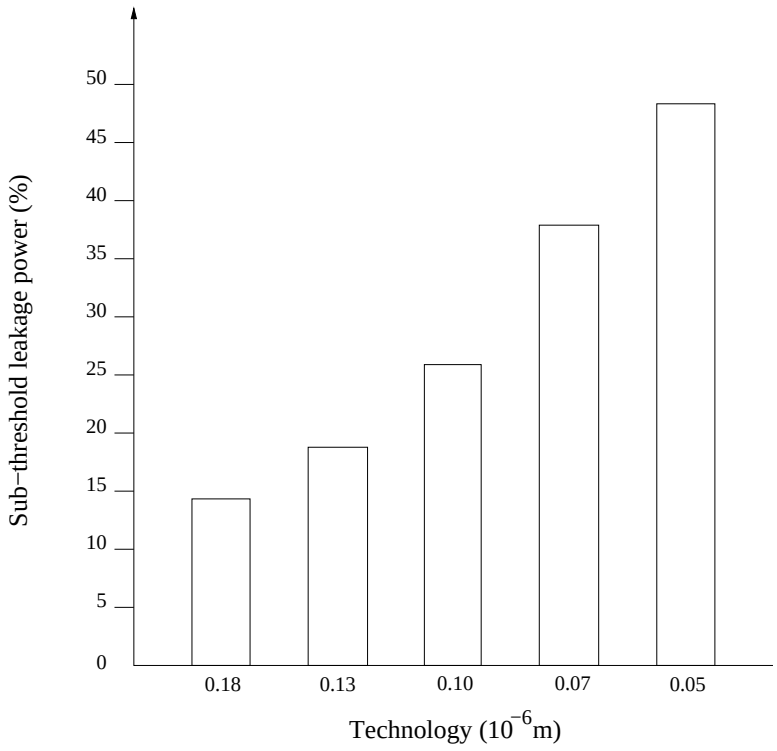


Figure 1.1 Increase in Leakage Power with Shrinking CMOS Technologies [1].

current technologies [6, 7]. Figure 1.1 shows the projection of the increase in subthreshold leakage power with shrinking technologies.

Several circuit-level leakage power minimization techniques have been proposed in the literature [8]. These techniques are briefly described below:

1.1.1 By Controlling the Input Vector

Several works [9–11] established the effect of input pattern on the circuit leakage behavior, which is a consequence of 'stacking effect' [12]. Since the inputs to the devices in the stack determine their state, which are determined by the primary inputs to the circuit, the leakage current is minimized by finding the input pattern that maximizes the number of disabled transistors in all the stacks across the design [13]. Figure 1.2 shows the leakage power histogram for one of the ISCAS-89 benchmark circuits for 100, 000 randomly chosen input vectors [14]. It can be seen for this circuit that the minimum leakage power is less than half that of the maximum leakage power.

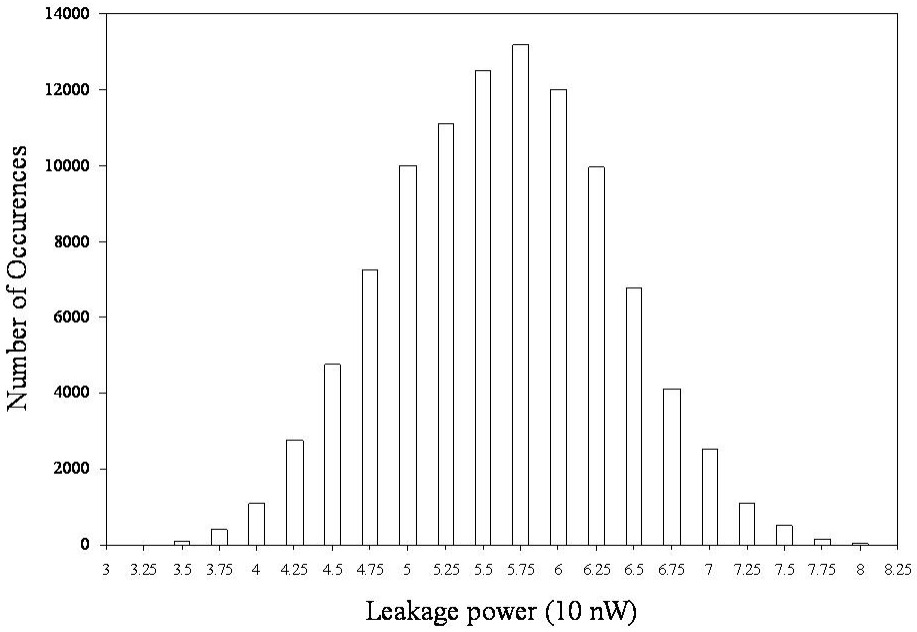


Figure 1.2 S298 Leakage Power Histogram [14].

One of the advantages of the input vector control technique is that its implementation requires minimal architectural support. In some cases, the sleep signal that determines whether the device is active may already be available [8].

1.1.2 By Increasing the Threshold Voltage

From Equation (1.2), it can be seen that there is an exponential relationship of the sub-threshold current, I_{subth}, with respect to the threshold voltage, V_T. There are multiple implementations proposed in the literature which achieve reduction in I_{subth} by increasing V_T. For all such implementations, the process technology is required to support the possibility of changing the threshold voltage of some (or all) transistors from the default value [8].

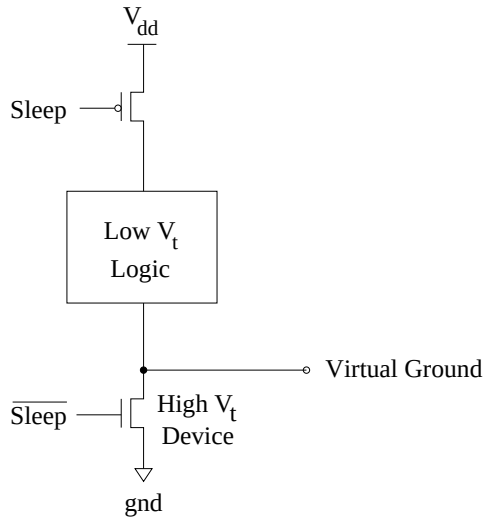


Figure 1.3 MTCMOS Circuit Structure [5].

- Multiple Threshold Voltage CMOS (MTCMOS) Technique - The MTCMOS technique is presented in [15] where a high-V_T device, known as a *sleep* transistor, is connected in series with low-V_T circuitry. Since the high-V_T transistor inserted has a very small on-resistance, virtual supply and/or ground rails are created with voltage levels very close to the real power lines. Figure 1.3 shows the MTCMOS circuit structure.

- Dynamic Threshold MOS (DTMOS) Technique - This technique [16], ties the body and the gate of each transistor such that whenever the device is turned off, low leakage is achieved. When the device is turned back on, higher current drives are possible.

- Dual V_T CMOS Technique - Yet another approach uses high-V_T devices on non-critical paths and low-V_T devices on the critical ones [17]. The standard approach for implementing this technique is to partition the circuit into critical and non-critical regions, and to use fast low-V_T devices when required to meet performance goals.

- Variable Threshold CMOS (VTCMOS) Technique - VTCMOS is one of the techniques that enables modifying the threshold voltage during runtime. The threshold voltage, V_T, is raised during standby mode by making the substrate voltage higher than V_{dd} in P devices and lower than ground in N devices. This technique is also called Body bias control (BBC) [8].

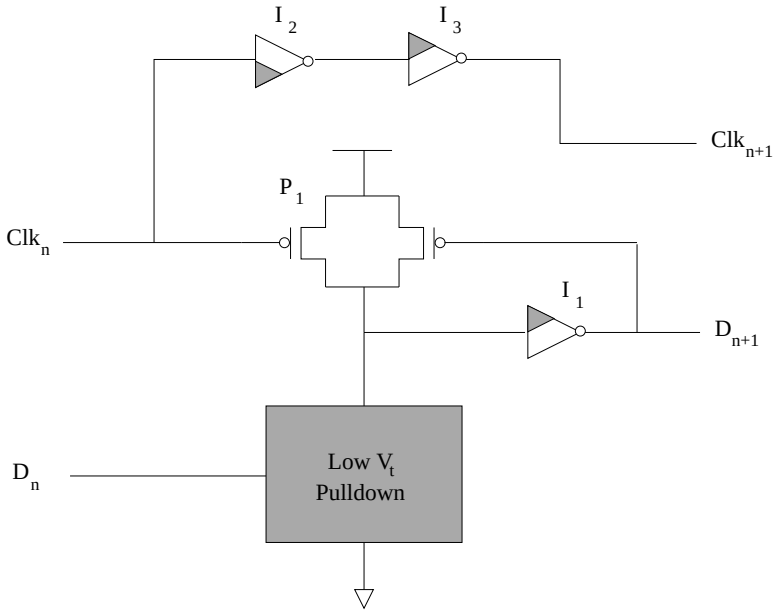


Figure 1.4 Dual-V_t Domino Logic Gate with Low-V_t Devices Shaded [5].

- Dual-threshold Domino Circuits - Dual-threshold voltage domino provides the performance equivalent of a purely low-V_T design with the standby leakage characteristic of a purely high-V_T implementation [5]. Since domino logic is characterized by fixed

transition directions, one can conveniently place the dual-V_T domino gate into a low leakage state, and can embed high-V_T devices in noncritical transition directions without impacting performance. Thus, with the dual-V_T domino gate, the designer has the option of trading off reduced precharge time for lower standby leakage currents. The advantage with dual-V_T domino methodology is that it utilizes high threshold voltages for all transistors that can switch during precharge modes and utilizes low threshold voltages for all transistors that can switch during the evaluate mode. Figure 1.4 illustrates a typical dual-V_T domino stage. It consists of a pulldown network, inverter (I_1), leaker device (P_1), and clock drivers (I_2, I_3). The low V_T devices are shaded in the figure.

1.1.3 By Gating the Supply Voltage

This technique, also called power-gating, reduces standby leakage by turning off the supply voltage to the circuit or parts of it which are idle and not involved in performing any worthwhile computation. This can be implemented using *sleep* transistors as in MTCMOS designs.

This technique was proposed for reducing leakage energy in SRAM cells for reducing leakage power dissipation in cache memories [18]. The fundamental reason for the reduction in leakage is the stacking effect of self reverse-biasing series-connected transistors. Power-gating maintains the performance advantages of low power supply and threshold voltages while achieving leakage reduction. In general, this technique does not require additional process technology support, in contrast to adjustable V_{TH} techniques [8]. Figure 1.5(a) and 1.5(b) show the footer and the header configurations for power-gating, respectively.

1.2 Power-Gating Technique for Architectural-Level Leakage Reduction

Power-gating technique is becoming popular for reducing leakage energy in circuits, in which, leakage energy dissipation is reduced by cutting off the power supply to the parts of the circuit which are not expected not to be involved in any useful computation for sustained

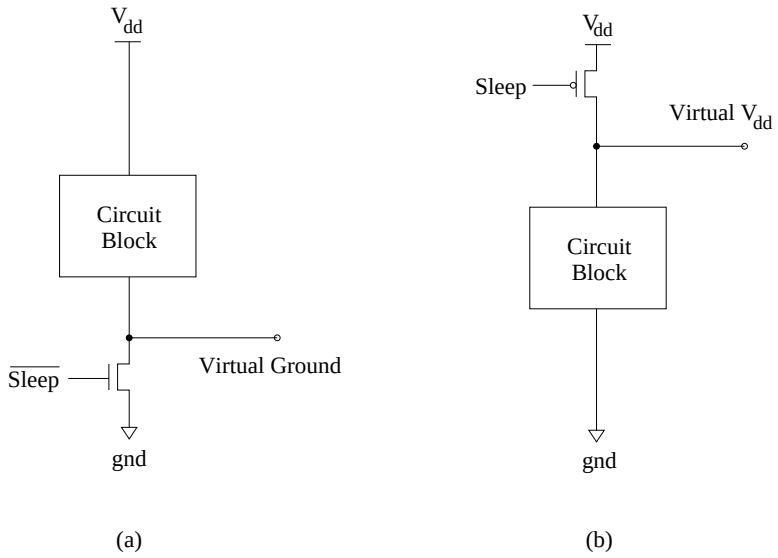


Figure 1.5 Power Supply Gating (a) Configuration with a Footer Sleep Transistor, (b) Configuration with a Header Sleep Transistor.

periods of time. Because of the simplicity of the implementation of this technique, power-gating has been applied to minimize leakage energy in circuits at the architectural level. The effectiveness of using power-gating requires the following:

- provide the switches for turning the functional units on and off at the circuit level (the power-gating capability).
- the controls for those switches to power-gate the various parts of the circuit can be provided as handles at the system level thereby giving the system software or the compiler the ability to control them.

Since there is an energy overhead associated with reestablishing the power supply to those parts of the circuit, the amount of time that the supply is cut off has to be long enough in order to achieve overall energy savings. For efficient utilization of this strategy, it is important to extract the information regarding the idle time periods to make sure that deactivation is beneficial in terms of energy savings versus the overhead to accomplish power-gating. In order to ensure the above, it is important to have accurate information regarding the idle time periods in the application code as well as the energy characterization

of the functional units. In other words, only by knowing accurately the leakage energy consumption of the hardware unit, one can determine the cut-off point that ensures beneficial power-gating.

1.3 Previous Work

A taxonomy of various works in architectural-level leakage reduction is given in Figure 1.6. As mentioned in the previous section, the circuit-level techniques for leakage reduction that have been implemented at the microarchitectural or the compiler-level are: (i) input vector control; (ii) body-bias control; (iii) dual-threshold domino circuits and (iv) power-gating.

Some earlier works on architectural-level leakage reduction have concentrated primarily on the memory subsystems, particularly caches, since they contribute upto 50% of the entire leakage consumption of the processor system with memory hierarchy [19, 20]. However, since the number of functional units in a superscalar processor is large to be able to exploit high degree of Instruction Level Parallelism (ILP), the superscalar processors become good candidates for saving leakage energy in functional units. Therefore, subsequently, there have been attempts at investigating leakage reduction in functional units in the datapath of a superscalar processor.

Among the circuit-level and microarchitectural techniques, reverse body-bias was investigated for leakage reduction in the Intel Xscale microprocessor by Clark et al. [21]. They choose to employ this technique because it is a circuit design approach which precludes any changes to the process technology. Moreover, it supports state retentive power-down, i.e. the state of the circuit is retained during all times, which is not the case with MTCMOS or power-gating methodology. Their scheme controls static leakage current while maintaining performance and active power.

An analytical energy model to achieve leakage energy optimization in functional units for superscalar processors, suitable for architectural-level analysis, is developed by Dropsho et al. in [22]. They explore the interaction of the application and technology, and study the effect that the parameters of their model has on the energy and performance of the system.

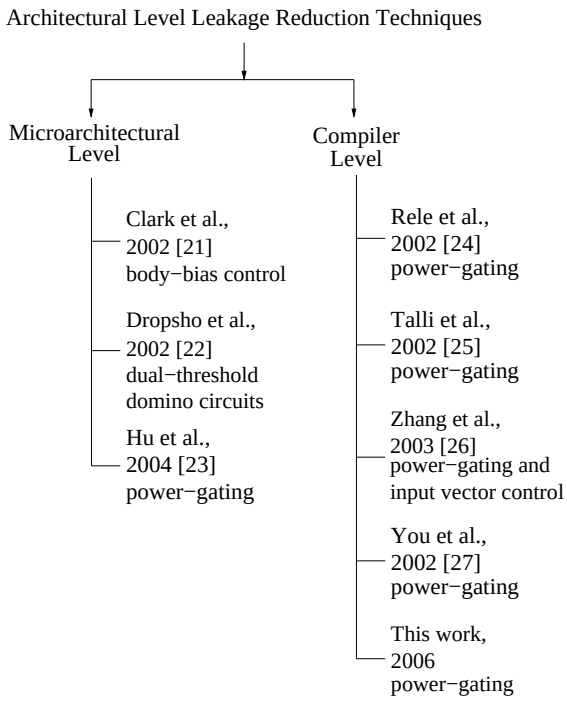


Figure 1.6 Taxonomy of Related Works on Architectural-Level Leakage Reduction in Microprocessors.

The static energy in the integer functional units is reduced by employing dual threshold voltage domino logic design technique. To achieve smaller idle time periods for functional units, they also propose a design technique called *gradual sleep* to reduce the energy impact of using the *sleep* mode.

Dynamic microarchitectural techniques for power-gating of execution units are investigated by Hu et. al [23] in which activation and deactivation of the functional units are guided by branch prediction techniques. They compare two different techniques for detection of power-gating opportunities of functional units. Initially, a perfect predictor is assumed that can predict the idle intervals of these units without any delay to evaluate the maximum power-gating potential of the functional units for a set of application programs. Following that, two different dynamic power-gating schemes: time-based and branch-misprediction guided, are studied, which are employed dynamically during program runtime.

There have been some compiler-level efforts in the literature which employ power-gating for leakage energy reduction [24–27]. These techniques perform data flow and component

usage analyses on the object code to insert *turn-off* and *turn-on* instructions to cut-off and resume power supply to the functional units, respectively.

Rele et al. in [24] employ power-gating at the compiler-level to reduce leakage power in functional units of superscalar processors. Corresponding to an application program, long idle periods for the functional units are predicted with the help of dynamically profiled information for typical sets of inputs. Based on this prediction, the compiler inserts instructions to turn off and turn on the functional units in the assembly code. Although the results are not reported in terms of the leakage energy savings, they indicate that most of the functional units can be kept off for more than 90% of the execution time at the cost of less than 1% performance degradation for all the benchmarks.

Zhang et al. in [26], propose a compiler-based technique for reduction of leakage energy for VLIW processors. They use data-flow analysis techniques and compare employment of various circuit-level leakage reduction techniques, including input vector control and power-gating, at the compiler-level for achieving leakage energy optimizations during the entire execution of the program. Their study shows that leakage energy savings upto 45% are obtained when both input vector control and power-gating are employed with the knowledge of dynamic profile information.

1.4 Motivation

The microarchitectural techniques discussed earlier [22, 23] have a dynamic energy overhead associated with the control block that is responsible for detecting the sleep periods. In superscalar architectures, sophisticated branch prediction methods are employed which assist in the implementation of the sleep period detection [23]. These architectures also consist of multiple number of multiple types of functional units, which is not the case in embedded processors. Due to this reason, the potential of leakage energy savings in superscalar architectures is much more than for embedded processors. This justifies the implementation of the microarchitectural techniques in superscalar.

In contrast to that, in a compiler-based technique, the runtime control overhead is not present since the *sleep* instructions are statically inserted into the object code. According to our understanding, the compiler-level techniques proposed in the works cited above, although reported only for superscalar [24, 25, 27] and VLIW architectures [26], in principle, should also work for embedded processors. However, in these works, the various energy components associated with application of the power-gating technique have been assumed based on some architectural-level power models as opposed to using their precise values. To the best of our knowledge there is no architectural leakage reduction work that exist for embedded processors.

In this paper, we propose a compiler-based technique for reducing leakage energy consumed by the functional units in an embedded microprocessors core. We investigate the program behavior of a set of benchmark applications for embedded systems. It is well known that the applications for embedded processors are characterized by relatively small code sizes when compared to those for general-purpose processors which are executed as part of iterative loops. Based on these observations, we focus on the iterative code structures in the program for detection of long idle regions in the program. Switches for power-gating the functional units are provided at the circuit level, which are controlled with special instructions inserted within the code during compile time based on idle time behavior analysis. Extensive simulations were performed based on the ARM processor as the target architecture model, and synthesizing a library of functional units with power-gating capability using 70nm MOSIS technology files. Experimental results are presented for two benchmarks for each category in MiBench suite [3] which indicate a leakage energy reduction of 34% on the average.

1.5 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 describes the framework for power-gating adopted in this work. The target architectural model with power-gating capability, and the modifications required to the compiler are discussed in detail. Chapter 3

presents the design of functional units with power-gating capability, the experimental setup, the details of simulations and energy computations, and the experimental results in terms of leakage reduction. Chapter 4 draws the conclusions of this work and discusses some future directions for this work.

CHAPTER 2

FRAMEWORK FOR POWER-GATING

In this chapter, we describe the framework used in this work for applying power-gating of functional units in embedded processors. First, we present a brief explanation of the entire framework. Then, we describe the modified ARM architecture used in this work which has the support for power-gating of the functional units in its datapath. Finally, we present the technique employed for finding out the idle regions in the program and inserting the *sleep* instructions. We also describe a robust mathematical infrastructure which enables us to devise that technique.

2.1 Compiler-level Leakage Reduction Framework for Power-Gating

The compiler-level leakage reduction framework adopted in this work is shown in Figure 2.1. A brief explanation of the various inputs to this framework is given below:

- Application Source Code - The source code for each application is compiled and is represented as a list of assembly-level instructions. In this work, we use benchmarks from the MiBench suite [3]. The precompiled executables for these benchmarks are available for little-endian machines running linux on them. The assembly-level instruction list for the benchmarks were obtained by the object dump of the precompiled executables.

- Power-Gated Architecture - The power-gated processor architecture is modeled on the existing ARMv7 processor [28]. A floating-point unit has been added to the datapath for the modified ARM architecture used in this work because the ARMv7 processor core has just integer functional units. The floating-point unit is modeled on

the VFP9-S vector floating-point coprocessor [29], which can be connected externally with a newer version of the ARM processor, ARM9E.

- Functional Unit Power-Gating Details - Since there is an overhead energy component associated with the power-gating technique, the *breakeven* period for each functional unit is calculated. The *breakeven* period, also called the *threshold* period, is defined as the minimum period of time for which a functional unit should be kept deactivated such the leakage energy savings during that period is equal to the sum of the overhead energies associated with its activation and deactivation.

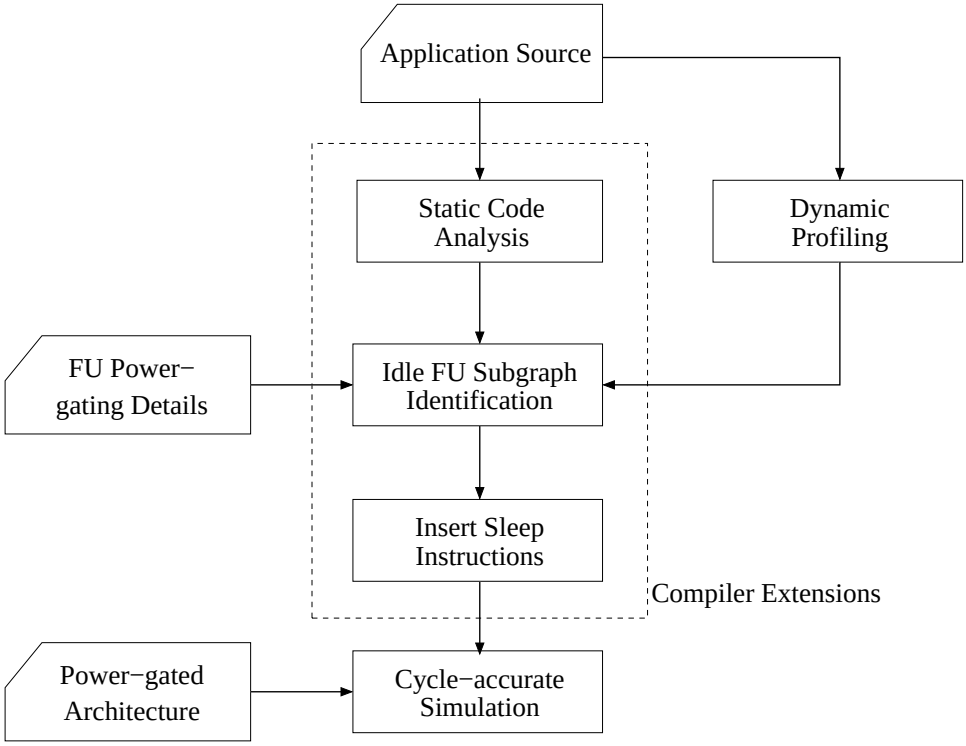


Figure 2.1 Framework Adopted for Power-Gating.

A brief description of the various tasks in the framework is as follows:

- Static Code Analysis - During this stage, the application source code is statically analyzed for functional unit requirements. A control flow graph (CFG) for the source

program is created with the vertices representing the basic blocks[1] and the edges representing the flow of program control. Along with the CFG, a data structure, called *loop hierarchy trees* (LHTs), is also created which captures the nesting structure of the iterative segments in the program.

- Dynamic Profiling - In the dynamic profiling stage, the application program is executed using test inputs which are supplied with the benchmarks. The basic blocks of the CFG are then annotated with the freqency of their execution from the program execution with the test inputs. This task is performed using the dynamic profiling tool available with the Simplescalar distribution [30], `sim-profile`. Section 3.2.2 describes this stage in more detail.

- Idle FU Subgraph Identification - At this stage, for each functional unit, maximal subgraphs are identified within the source program in which it is not used. We inspect entire functions and loop structures in the programs to identify such subgraphs. Loops not only enclose subgraphs in a program but they also dictate the program runtime behavior. The LHTs, which provides a partial ordering of the loops in a program, are used to identify the maximal subgraphs within loops. The *breakeven* periods of the functional units are used to decide the size of the subgraphs suitable for power-gating.

- Insert *Sleep* Instructions - At the *sleep* instruction insertion stage, the locations in the code for inserting the sleep instructions are decided. However, in this work, we do not regenerate code after finding out the regions where the *sleep* instructions are to be inserted. Instead, we pass the location of the *sleep* instructions to the cycle-accurate simulator to simulate execution of application code with power-gating instructions.

- Cycle-Accurate Simulation - The cycle-accurate simulation tool provided with the Simplescalar distribution, `sim-outorder`, is used to perform the computation of the leakage energy consumption of the power-gated functional units in the modified ARM

[1] A *basic block* is a straight-line code sequence with no transfers in or out, except at the beginning or the end

processor core. The Simplescalar ARM distribution is used in this work which simulates the Intel StrongArm processor core [31]. A detailed description of this stage is given in Section 3.2.3.

2.2 Modified ARM Architecture with Power-Gating for Functional Units

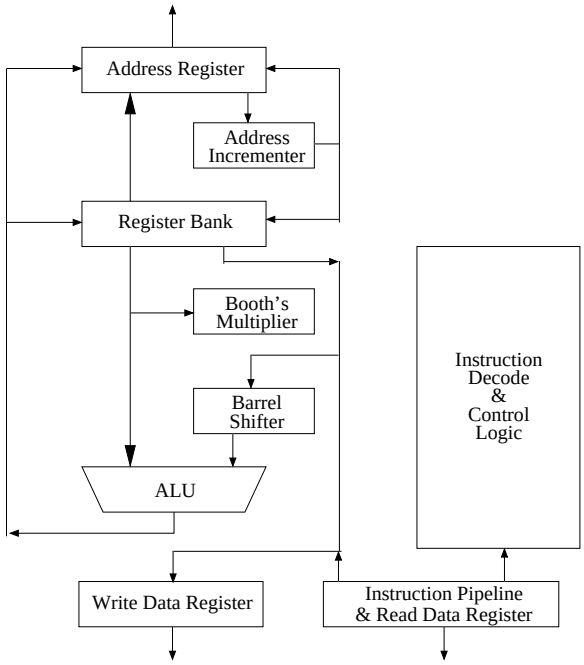


Figure 2.2 Standard ARM Architecture [28].

Figure 2.2 shows the block diagram of a generic ARM architecture (ARMv7) reproduced from [28]. As can be seen from the Figure, the processor core does not have any floating-point functional units. This architecture is modified to enable power-gating functionality for the functional units. The modified architecture is shown in Figure 2.3.

A floating-point unit has been added for the floating-point instructions supported by the ARM Instruction Set. The functional units are designed such that the latency in their activation[2] takes one clock cycle[3]. The details of the functional units are discussed in

[2] *Activation latency* is defined as the time required by a circuit to become operational after the sleep transistor is turned on

[3] Using HSPICE we verified that the activation latency for each module in our library is under one clock period of 10 ns

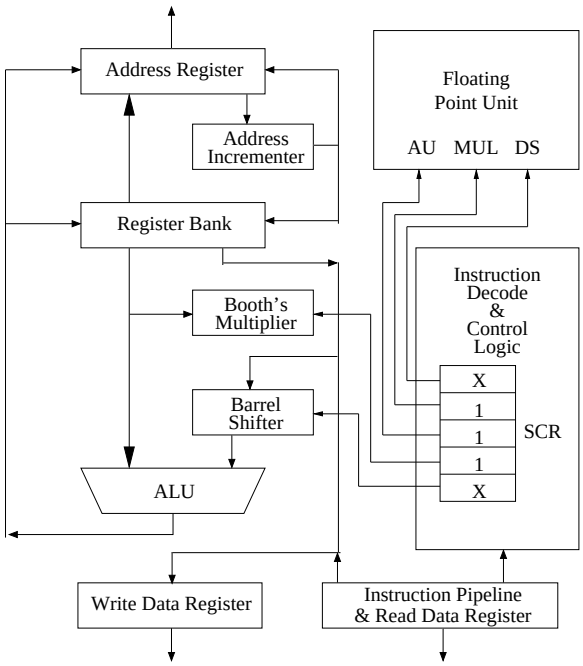


Figure 2.3 Modified ARM Architecture with Power-Gating for Functional Units.

Section 3.1. The instruction pipeline is considered to be a generic five-stage pipeline with a *diversified* execution pipe [32]. By a diversified execution pipe, we mean that there are parallel sub-pipelines employing different functional units in the execute stage(s).

Deactivating the functional units is carried out using the *sleep* instruction. A Sleep Control Register (SCR) is added to the instruction decode logic. Each functional unit that needs to be deactivated is supplied as an operand to the sleep instruction. Since each functional unit can be required to be either activated or deactivated, one bit is required to specify each operand. A '1' indicates that the functional unit has to be deactivated. A '0' indicates the status of the functional unit should *not* be changed. When a sleep instruction is decoded, a '1' is written into the corresponding bit location in the sleep control register and a '0' is ignored.

In the following example, the operands passed to the *sleep* instruction are - integer multiplier, floating-point arithmetic unit, and floating-point multiplier.

```
sleep     I-mul, FP-au, FP-mul
```

When the above instruction is decoded, '1' is written in the SCR entries corresponding to integer multiplier, FP arithmetic unit, and FP multiplier, indicating that these functional units are put to sleep. The entries corresponding to the barrel shifter and the FP division and square root unit do not change, indicated by 'X'. This scenario is shown in Figure 2.3.

The activation of the functional units happens at the *decode* stage itself. The functional unit gets activated during the cycle following the one in which the instruction enters the operand *fetch* stage or the *dispatch* stage in the pipeline [32]. Therefore, by the time the instruction enters the *execute* stage, the functional unit is active to perform useful computation. This avoids the need for separate *wakeup* instructions.

2.3 Compiler Support

In this section, we analyze the requirements for applying power-gating at the compiler-level mathematically and propose a technique for identifying maximal subgraphs in the program pertaining to idleness of the functional units. We describe the data structures that we use to implement this technique and present algorithms for identifying and inserting *sleep* instructions.

2.3.1 Program Analysis

The control flow semantics of a program are represented in the form of a control flow graph (CFG), in which the vertices represent the basic blocks and the edges represent transfer of control flow between the basic blocks [32]. Figure 2.4 illustrates a CFG with four basic blocks (denoted by rectangles with dashed lines) each containing a number of instructions (denoted by ovals). The directed edges represent flow of control between the basic blocks. These edges are induced by the branch instructions (denoted by rhombuses). The run-time execution of a program leads to the dynamic traversal of the vertices and the edges of its CFG. The branch instructions and their branch conditions (in case of conditional branches) dictate the actual path of traversal during program execution.

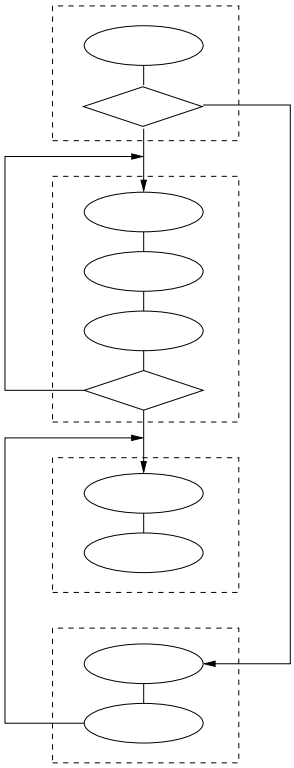


Figure 2.4 A Control Flow Graph (CFG) [32].

The main task of the compiler is to analyze the program behavior and predict regions in the program where certain functional units are not *expected* to be required during the execution of the program so that power-gating can be applied. Thus, given a CFG for a program, the objective is to find out maximal subgraphs in it pertaining to the idleness or non-usage of each functional unit. Then, *sleep* instructions should be inserted at the entry of these subgraphs so that the functional units are deactivated when the program control enters these subgraphs during program execution. Due to the presence of conditional iterative structures (*while* loops and *for* loops with dynamic iteration count), the run-time trace of the CFG of a program will be different for different run-time inputs. The program is, thus, dynamically profiled with some typical input data to characterize the program behavior that is not evident from the static analysis of the CFG. Based on these characteristics, the compiler inserts the appropriate sleep instructions.

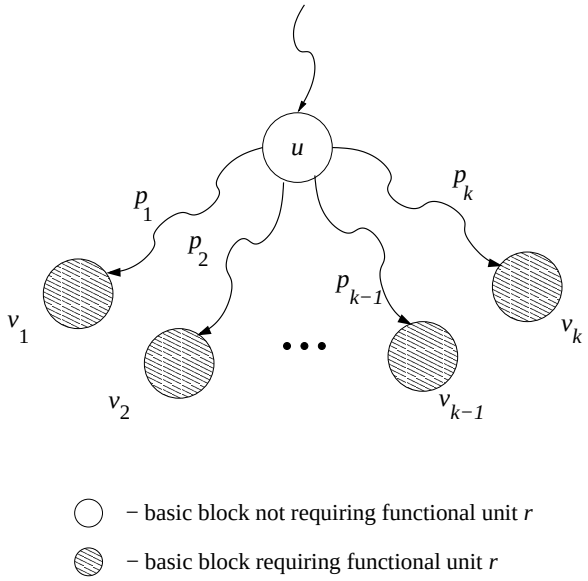


Figure 2.5 A Subgraph of the Run-Time Trace of a Control Flow Graph.

We investigate the problem of identifying the potential regions or subgraphs in the CFG where power-gating can be applied efficiently. We employ simple tree data structures representing loop hierarchy in a program to this end. In the remainder of this section, we analytically model the problem of inserting the sleep instructions and present a simple approach to solve it.

2.3.2 Identifying Potential Power-Gating Regions in the CFG

While generating the CFG for the source program, each vertex is annotated with the functional unit requirement of the basic block corresponding to that node. This information is required in finding subgraphs in the program CFG which are suitable for applying power-gating. Figure 2.5 illustrates a subgraph of the run-time trace of the CFG for a program. Vertex u represents a basic block that does not use a particular functional unit, say r. Vertices $v_1, v_2, ..., v_k$ represent basic blocks that use r. Let P represent the set of paths $\{p_1, p_2, ..., p_k\}$ where p_i is a path from u to v_i, such that the only vertex which uses r is v_i. None of the intermediate vertices in p_i use r.

Let t_{clk} be the clock period and $N(p)$ be the number of cycles required to execute the instructions in path $p \in P$. Then the total time spent in executing the instructions in p is given by,

$$T_p = N(p)t_{clk} \tag{2.1}$$

Let $L(p)$, the *length* of the path p, be described in terms of the number of instructions in p. If the maximum IPC (instructions per cycle) count, which is determined by the *fetch* and *decode* width of a processor, is λ, then

$$L(p) \leq \lambda N(p) \tag{2.2}$$

The inequality holds in Equation (2.2) because the IPC count for a code segment can be smaller than λ in account of cache misses, branch mispredictions, pipeline flushes, etc.

Eliminating $N(p)$ from Equation (2.1) and Equation (2.2), we obtain,

$$T_p \geq \frac{1}{\lambda} L(p)t_{clk} \tag{2.3}$$

For $\lambda = 1$, $T_p \geq L(p)t_{clk}$. This is intuitive in that, for a max IPC count at most 1, the minimum time spent in the execution of the instructions in p can be bounded by the number of instructions in p.

Let us now assume that the leakage energy saved per unit time by keeping the functional unit r deactivated is δ_r and that the dynamic energy overhead in activation and deactivation is Δ_r. Then the minimum time, t_{min} for which r should be deactivated before it is woken up is given by,

$$t_{min} = \frac{\Delta_r}{\delta_r} \tag{2.4}$$

So, for a path p to generate energy savings, the time spent in executing the instructions in p should be more than t_{min}, i.e., $T_p \geq t_{min}$. This is satisfied if,

$$\frac{1}{\lambda} L(p)t_{clk} \geq t_{min} = \frac{\Delta_r}{\delta_r} \tag{2.5}$$

which implies,

$$L(p) \geq \frac{\lambda \Delta_r}{\delta_r t_{clk}} \tag{2.6}$$

The quantity $\lambda \Delta_r / \delta_r t_{clk}$ is called the *threshold* length and is indicated by L_{th}. If the relation, specified by Equation (2.6), is satisfied for all $p \in P$, then

$$T_p \geq t_{min}, \ \forall \ p \in P \tag{2.7}$$

which ensures that the leakage energy saved in the subgraph shown in Figure 2.5 exceeds the dynamic energy overhead incurred, thereby, giving overall savings.

Although, the above approach ensures energy savings in every path in the subgraph G_i', it is a rather conservative approach. During the execution of the program, some of the paths in P may be greater in length and/or traversed more frequently than the rest. Thus, there is a possibility that the combined energy savings achieved along those paths exceeds the combined energy losses incurred along the rest of the paths, still resulting in overall energy savings.

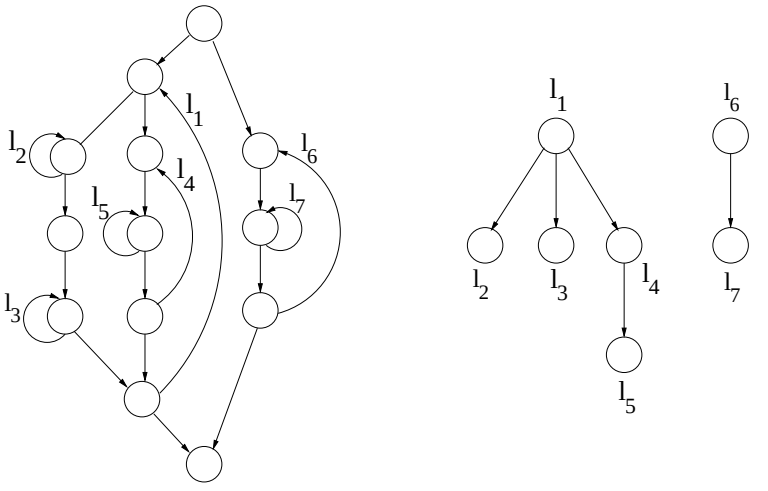
If n_p is the number of times the path $p \in P$ is traversed, then, using Equation (2.6), the leakage energy savings can be expressed as,

$$E_{savings} = \sum_{p \in P} n_p (\frac{L(p) \delta_r t_{clk}}{\lambda} - \Delta_r) \tag{2.8}$$

If $P' \subseteq P$ be the set of paths whose length is greater than L_{th} defined as, $P' = \{p \mid p \in P \text{ and } L(p) \geq L_{th}\}$, i.e., the set of paths whose length is greater than L_{th} then Equation (2.8) can be rewritten as,

$$E_{savings} = \sum_{p \in P'} n_p (\frac{L(p) \delta_r t_{clk}}{\lambda} - \Delta_r) - \sum_{p \in P - P'} n_p (\Delta_r - \frac{L(p) \delta_r t_{clk}}{\lambda}) \tag{2.9}$$

From the result in Equation (2.9), we can observe that if we identify subgraphs in the program CFG which can result in program execution paths of length at least L_{th} such that:



(a) CFG of a sample function (b) Loop hierarchy forest for the CFG

Figure 2.6 Loop Hierarchy Tree.

(i) a functional unit is not required in those paths; and (ii) the paths are also traversed relatively more number of times, we find potential regions in the program CFG which are good candidates for power-gating. Note that, in practice, the average program IPC can be smaller than λ. This will result in missing out various periods during the program execution where power savings could be achieved. However, even though the IPC may vary for different paths in the run-time trace of the CFG, our approach gives a consistent lower bound on the energy savings on the paths that result during program execution.

2.3.3 Subgraphs Enclosed within Loops

Following the analysis in the previous subsection, we use the loops in the source program to identify subgraphs which we hope would result in paths satisfying the conditions mentioned in the previous section. We introduce the notion of *loop hierarchy trees* (LHTs) to capture the nesting structure of the program segments. During the static code analysis phase, for each function in the source program, we create a forest of the LHTs. Each vertex of a LHT denotes a loop in the source program and its decendants denote the loops that are nested within that loop. Each vertex is annotated with the functional unit requirement of the loop corresponding to that vertex. Figure 2.6(a) shows the control flow graph of the

basic blocks in a sample function with the loops as indicated on the back edges. Figure 2.6(b) shows the corresponding loop hierarchy trees for the CFG shown in Figure 2.6(a). Loop l_1 has 3 nested child loops, l_2, l_3 and l_4. Among these, l_4 further has l_5 as a nested loop. Similarly, l_7 is a loop nested within loop l_6. For the sake of simplicity, the functional unit requirements for the basic blocks in the CFG and the loops in the LHTs are not shown in the figure.

An essential property of a LHT is that it gives a partial ordering[4] of the subgraphs to be considered in the CFG for the program. In Figure 2.6(b), loop l_4 encloses a bigger subgraph than l_5 does. However, a definite ordering of the sizes of the subgraphs enclosed by l_3 and l_4 cannot be established.

During the dynamic profiling of the source program, the following information is gathered:

- the vertices in the CFG of the program are annotated with the corresponding basic block execution frequencies.
- the vertices in the loop hierarchy tree are annotated with the corresponding loop execution frequencies.

We define the average length of a loop in terms of the average number of instructions that are executed from within the loop during the dynamic profiling stage. Let l_i be a loop in the CFG of the i^{th} function in the source program, $G_i = (V_i, E_i)$, such that $S(l_i) \in V_i$ denote the set of vertices in l_i and C_i denote the set of child loops of l_i. Then the average length of the loop l_i is defined as the average number of instructions executed as part of that loop during the dynamic profiling state and is given by the recursive relation,

$$L_{avg}(l_i) = \frac{1}{f(l_i)}\Big(\sum_{c_i \in C_i} L_{avg}(c_i) f(c_i) + \sum_{v \in S(l_i) - \bigcup_{c_i \in C_i} S(c_i)} L_{bb}(v) g(v) \Big), \tag{2.10}$$

where,

$f(l_i)$ = the frequency of execution of loop l_i,

[4] A partial order is a relation that is reflexive, antisymmetric, and transitive [33].

$L_{avg}(c_i)$ = average length of nested loop c_i,

$f(c_i)$ = the frequency of execution of nested loop c_i,

$L_{bb}(v)$ = length of basic block corresponding to vertex v,

$g(v)$ = the frequency of execution of the basic block corresponding to vertex v.

The average length values are used to make decisions about inserting sleep instructions in cases where the loop requires a functional unit but only a subset of the nested loops within that loop have the same functional unit requirement.

Thus, the L_{avg} values for all the loops in the CFG can be calculated by running a *breadth first search* from the root of each tree in the LHTs and applying *memoization*[5]. It can be noted from the definition of a nested loop that $f(l_i) \leq f(c_i)$, for all $c_i \in C_i$, i.e., the execution frequency of a loop is at least as high as that of its nested loops. For the functions that are called from within a loop, the entire function is considered as a basic block in the above formulation in Equation (2.10). Also, each function in the source program is separately analyzed for insertion of sleep instructions.

2.3.4 Insertion of Sleep Instructions

To find the locations in the program to insert *sleep* instructions, we perform a *depth first traversal* of the nodes in each LHT starting at its root. This traversal is done once for each functional unit and is terminated as soon as it is found that the entire loop corresponding to the node does not use the functional unit. We define the normalized average length of loop x, $L_{norm}(x)$, as,

$$L_{norm}(x) = L_{avg}(x)\frac{f(x)}{h(x)} \tag{2.11}$$

where,

$f(x)$ = execution frequency of loop x,

$h(x)$ = the number of times loop x is entered from a basic block outside of x. We perform the normalization to quantify the iterative degree of loops. Note that, in Equation (2.11), $f(x) \geq h(x)$. Therefore, $L_{norm}(x) \geq L_{avg}(x)$. This enables us to consider loops that are

[5] *Memoization* is a top-down dynamic programming strategy [33].

small (have a low instruction count) but are highly iterative to be potential subgraphs for power-gating.

This is illustrated by an example given in Figure 2.7.

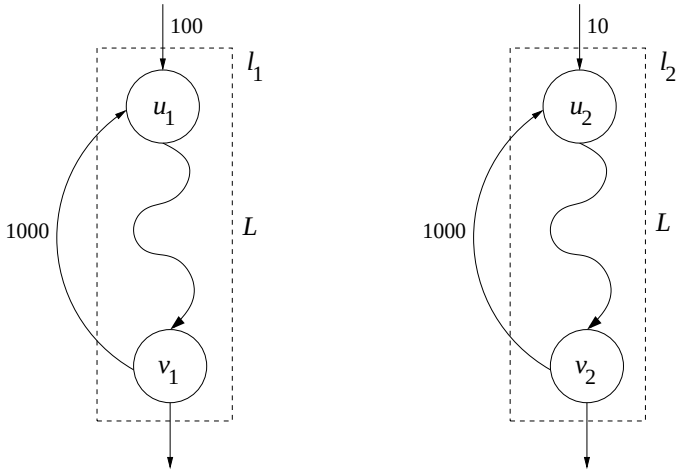


Figure 2.7 Example to Illustrate the Significance of Normalized Lengths of Loops.

There are two loops, l_1 and l_2, consisting of basic blocks bounded by the dashed rectangles, Their execution frequencies are the same, $f(l_1) = f(l_2) = 1000$ and their average lengths are also the same, $L_{avg}(l_1) = L_{avg}(l_2) = L$. However, loop l_1 is entered 100 times whereas loop l_2 is entered 10 times, as indicated by the edges entering the loops. Thus, on an average, each time loop l_1 is entered, it iterates $1000/100 = 10$ times while each time loop l_2 is entered, it iterates $1000/10 = 100$ times. Therefore, it can be observed that although both the loops have the same execution frequencies and the same average lengths, the program spends more time once it enters loop l_1 than it does in loop l_2.

Algorithm 1 INSERT-SLEEP(F) ▷ Algorithm to insert sleep instructions

```
1: S ← Φ
2: for all tree T ∈ F do
3:    for all functional unit r ∈ R do
4:       INSPECT(root(T), r)
5:    end for
6: end for
7: UNIQIFY(S)
```

Algorithm 1 presents the pseudocode for the routines INSERT-SLEEP. The set S maintains the sleep instruction locations. It is set to a NULL set at the beginning (line 1). In lines $2-6$, the two *for* loops iterate over all the LHTs in the LHT forest F for each functional unit in R and calls the routine INSPECT at the root of each tree. After all the LHTs are inspected, in line 7, the routine UNIQIFY is called to uniqify all the sleep instructions in set S.

Algorithm 2 INSPECT(x) ▷ Algorithm to inspect loops

```
1: if r ∉ res(x) then
2:    if L_norm(x) ≥ L_th then
3:       S ← S ∪ {e(x), r}
4:    end if
5: else
6:    for all y ∈ C_x do
7:       INSPECT(y, r)
8:    end for
9: end if
```

Algorithm 2 presents the pseudocode for the routine INSPECT which takes a vertex of a LHT, x, and a functional unit, r, as the arguments. In lines $1-4$, it checks whether r is used in x or not. The functional unit requirement for loop r is given by $res(x)$. If r is not used in loop x, it checks the normalized length of x, $L_{norm}(x)$. If $L_{norm}(x)$, is greater than the threshold number of instructions, L_{th}, the location is marked (added to the set S) for insertion of a *sleep* instruction for deactivating functional unit, r. The location is a two element tuple consisting of the basic block leading to the loop, $e(x)$, and the functional unit, r. If, however, r is used in the loop x, it calls INSPECT recursively on all the child vertices of x (lines 5-8). In other words, we inspect the loops nested in x to explore the possibility of power-gating r.

Algorithm 3 UNIQIFY(S) ▷ Algorithm to uniqify the sleep instructions

```
1: Sort the elements in S
2: for all unique locations do
3:    Merge all the FUs to form one sleep instruction
4: end for
```

Algorithm 3 describes the algorithm to create a unique sleep instruction for all the functional units which have the same location in the code. In line 1, the elements in S are sorted in ascending order by their locations. In lines $2-4$, a unique *sleep* instruction is generated for all the functional units which have the same location.

2.3.5 Time Complexity

Since the routine INSPECT implements a conditional DFS traversal in T, its worst-case time complexity is given by $O(|V|+|E|)$ [33]. However, since for a rooted tree, $|E| = |V|-1$, the worst-case time complexity of INSPECT is $O(|V|)$. Note that $|V|$ denotes that number of loops in a function. Therefore, the worst-case time complexity of INSPECT is linear in the number of loops in the function corresponding to the LHT T.

The time complexity of routine UNIQIFY is $O(|S| \lg |S|)$.

The routine INSERT-SLEEP makes calls to INSPECT for all the LHTs for each functional unit. Since there is one LHT for each function in the program, the worst-case time complexity of lines $1-6$ in INSERT-SLEEP is $O(n * |R|)$, where n is the total number of loops in the entire program across all functions and $|R|$ is the total number of power-gating enabled functional units. However, $|R|$ is constant since the number of functional units is constant. Thus, the worst-case time complexity of lines $1-6$ in INSERT-SLEEP is $O(n)$, which is linear in the number of loops in the program. However, the number of sleep instructions that can be inserted can be a maximum of the number of loops present and, therefore, $|S| \leq n * |R|$. Thus, the worst-case time complexity of the call to the routine UNIQIFY in line 7 of INSERT-SLEEP is $O(n \lg n)$. Therefore, the worst-case time complexity of routine INSERT-SLEEP is $O(n \lg n)$.

2.3.6 Handling Standard Library Functions

The standard library routines to which calls are made from the source program can either be treated as blackboxes so that no sleep instructions are inserted into them or they can be analyzed for the insertion of sleep instructions. In former case, the functional unit

requirement for the the entire library routine can be found out by analyzing the disassembled the object code for the library. However, in the latter case, the CFG for the function has to be analyzed just like any other function in the source code. Since a statically linked library is a relocatable object code, generation of a CFG from the disassembled object code is possible. But, insertion of sleep instruction would require regeneration of the object code and then linking the new object code instead of loading the standard library. In this work, we do not insert any sleep instructions within the standard library functions.

2.3.7 Summary

In this chapter, we described the framework that we use for applying power-gating of functional units in embedded processors. We presented the modified ARM architecture used in this work which has the support for power-gating of the functional units in its datapath. Finally, we described a mathematical infrastructure which enables us to devise the technique employed for identifying maximal subgraphs and present the algorithms to insert *sleep* instructions.

CHAPTER 3

EXPERIMENTAL RESULTS

In this chapter, we describe the experimental setup and the results. First, we present the specifications of the functional units created in this work. Then we describe the modifications made to the SimpleScalar-ARM simulator for this work. Next, we describe the calculations of various components of energy which are required by the compiler for the insertion of sleep instructions. Finally, we show the leakage energy savings obtained on a set of MiBench embedded benchmarks.

3.1 Design Details of Power-Gated Functional Units

We created functional units in compliance with the functional specifications of the functional units for the ARM processor. For the purpose of estimating power, these functional units are described structurally using lower level components like 4-bit carry lookahead (CLA) module, 8-bit registers, 8-bit shift registers, 8-bit multiplexers, etc. which are constructed in circuit level and are characterized for power using 70nm MOSIS technology model files. The power-gated version of all these components employ an appropriately sized footer sleep transistor. For example, the 32-bit adder in the integer ALU is constructed out of 8 stages of the 4-bit CLA module and the Booth multiplier is comprises primarily of 32-bit adder, 32-bit multiplexers and 32-bit shift registers. The power-gated functional units consist of the power-gated components while the regular functional units are comprised of components without any sleep transistors.

3.1.1 Arithmetic Logical Unit

The integer ALU performs integer arithmetic operations like addition, subtraction, comparison operations, and logical operations like inclusive or, and, not and exclusive or. The ALU unit does not have the power-gating feature because the integer adder is the most frequently used functional unit among all other units.

3.1.2 Barrel Shifter Unit

Under the ARM instruction set, the barrel shifter is used with the arithmetic and logical instructions and load store instructions. The instructions which use the ALU and the barrel shifter take two cycles in the execute stage. The barrel shifter performs the arithmetic right shift, logical left and right shift functions in a single cycle.

3.1.3 Booth's Multiplier Unit

The multiply instructions make use of a unit that implements 2-bit Booth's algorithm. The result is the least significant 32 bits of the product of the two 32-bit operands. The critical path of the unit has the delays of a 5-input multiplexer and that of a 32-bit adder delay. During the first cycle, the accumulator register is brought to the ALU, which either transmits it or produces a zero (depending on the instruction being MLA or MUL) to initialize the destination register. During the same cycle, the multiplier operand is loaded into the Booth's shift register. The multiplier operand is shifted right 2 bits per cycle, and, therefore takes 16 cycles to compute the final result.

The Booth's multiplier used in ARM version 7 employs early termination. However, the variability in the finishing time of the multiplication operation has not been modeled in the cycle accurate simulator and a worst case (16 cycle) latency has been used.

3.1.4 Floating Point Units

The floating point functional units have been modeled for supporting the IEEE standard 754 single-precision scalar operations supported by the VFP9-S Vector Floating-point Coprocessor [29]. The VFP9-S coprocessor has three separate pipelines:

- floating-point *multiply-accumulate* (FMAC) pipeline
- floating-point *divide and square root* (DS) pipeline
- *load and store* (LS) pipeline

These pipelines have common fetch and decode stages.

However, for our work, the floating point functional units have been separately implemented as:

- floating-point *arithmetic unit*
- floating-point *multiply unit*
- floating-point *divide and square root unit*

We construct single-precision functional units using the parameterized floating point unit designs presented in [34, 35]. However, since we construct these designs using components from our library, the latencies of the multiply unit and the divide and square root unit are different from those in the original work.

3.1.5 Floating Point Denormalizing and Normalizing Units

The normalized format of a floating point values is defined as the format in which exactly one non-zero digit forms the integer part of the mantissa which in case of binary representation is '1'.

The rounding and normalization unit supports two of the four rounding modes specified by IEEE standard: (i) round to zero and (2) round to nearest.

Figure 3.1 shows the schematic diagrams of the denormalizing and normalizing modules.

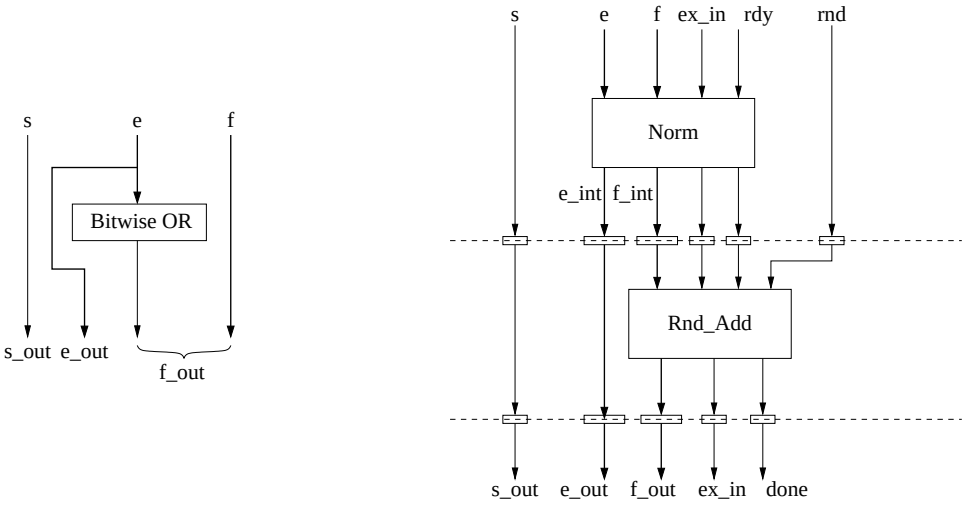


Figure 3.1 Schematic Diagram of Denormalizing and Normalizing Modules.

3.1.6 Floating Point Arithmetic Unit

The floating point arithmetic unit performs floating point addition and subtraction operations. The conversion modules - integer (both signed and unsigned) to floating point conversion and floating point to integer (both signed and unsigned) conversion, have not been constructed in this work. Figure 3.2 shows the schematic diagram of the floating point arithmetic unit.

3.1.7 Floating Point Multiply Unit

The floating point multiply unit performs floating point multiply operation. The first stage of the module is multicycled (6 cycles) because of the latency constraints of the available multiplier designs in our library. Figure 3.3 shows the schematic diagram of the floating point multiply unit.

3.1.8 Floating Point Divide and Square-Root Unit

The divide and square root unit perform single precision division and square root operations using a table lookup scheme [35]. Figure 3.4 shows the schematic diagram of the unit.

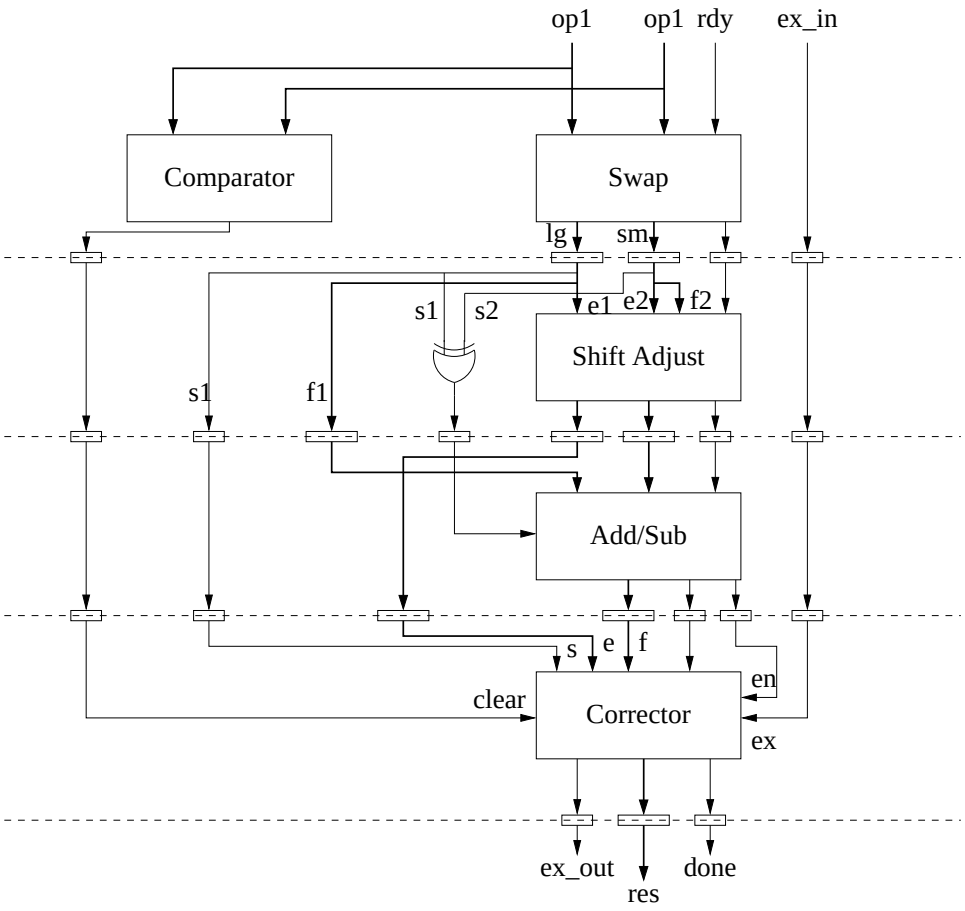


Figure 3.2 Schematic Diagram of Floating Point Arithmetic Unit.

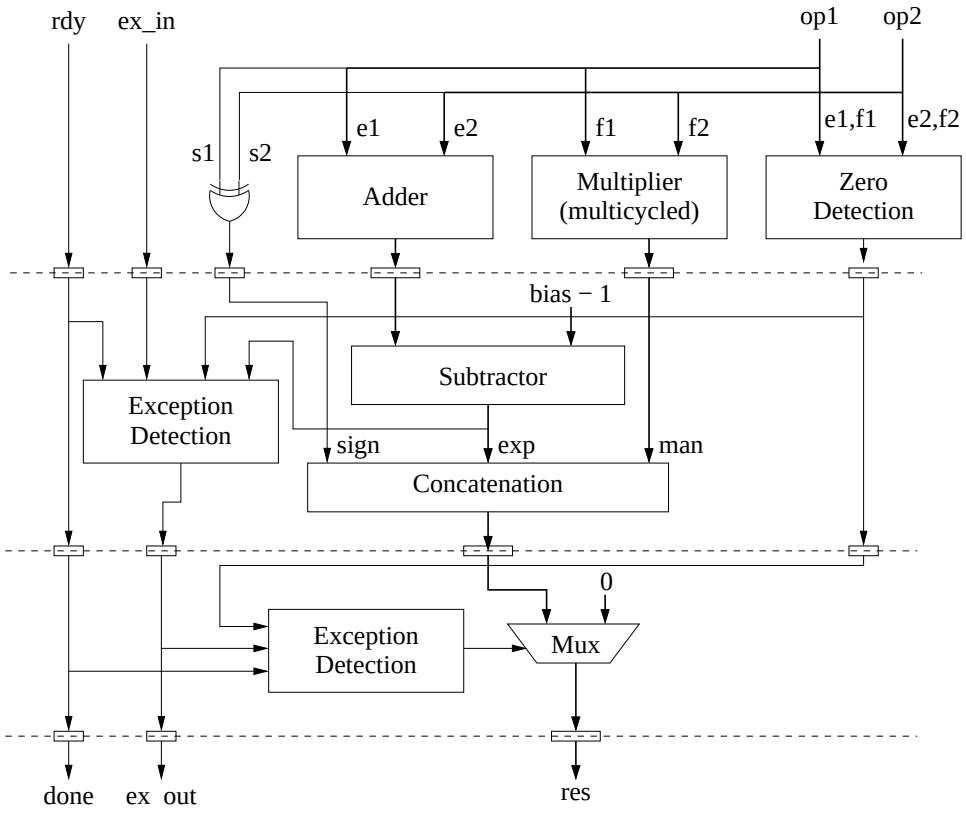


Figure 3.3 Schematic Diagram of Floating Point Multiply Unit.

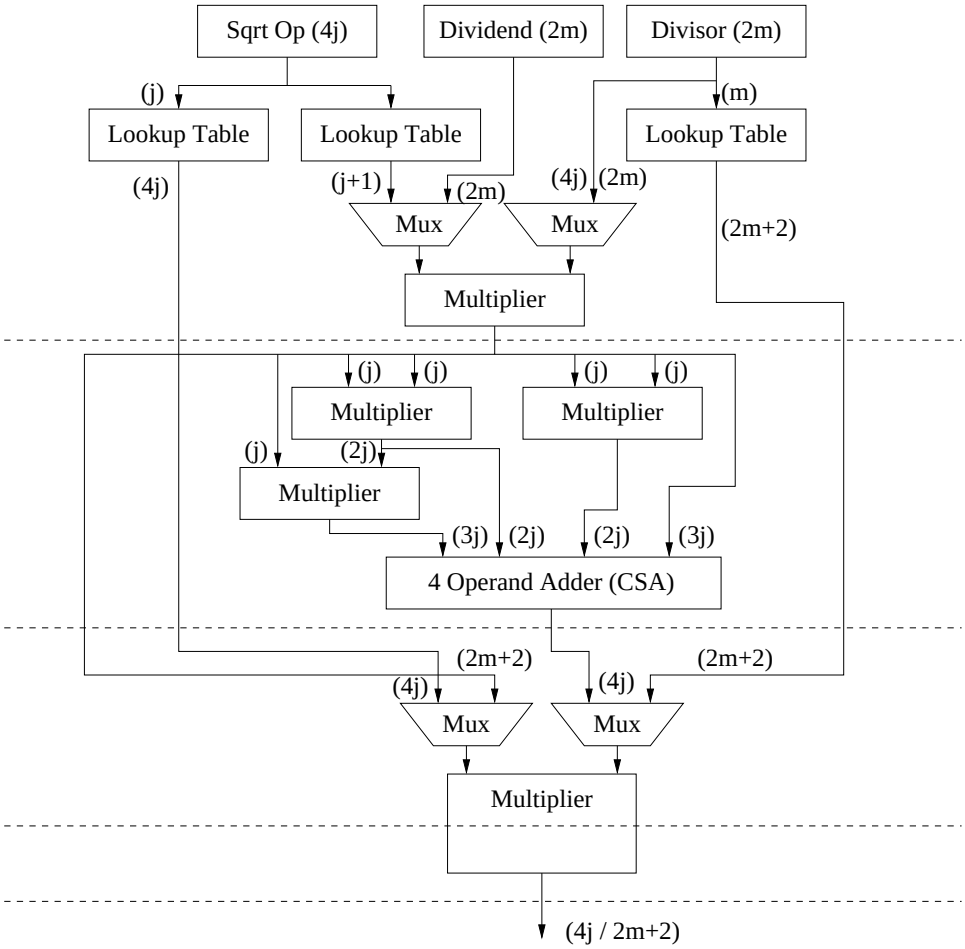


Figure 3.4 Schematic Diagram of Floating Point Divide and Square-Root Unit.

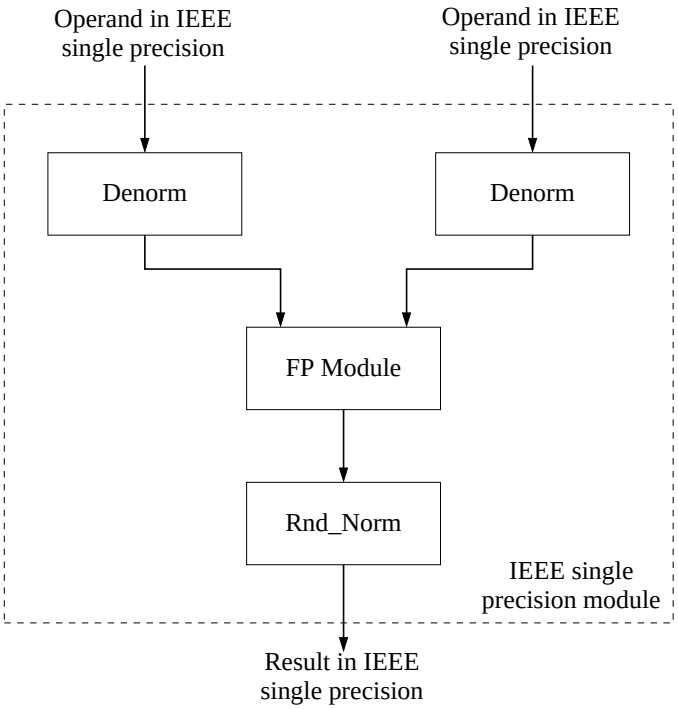


Figure 3.5 IEEE Single Precision Module.

For IEEE single precision format, $j = 7$ and $m = 12$. Stages 1, 2, and 3 have latencies of 8, 7, and 8 clock cycles, respectively, because of the multicycled multipliers. For square root operation all the stages of the pipeline are used. For division operation, the second stage is bypassed and only the multipliers in stage 1 and stage 3 are used.

3.1.9 Assembly of the Floating Point Modules

Figure 3.5 shows the assembly of the floating point functional units with the normalizing and denormalizing modules.

Table 3.1, describes the latency specifications of these components for a clock period of 10 ns. These latency values are used during cycle-accurate simulation with the embedded benchmarks.

Table 3.1 Latencies of Functional Units.

Functional unit	Pipeline stages	Latency (cycles)
Integer ALU	0	1
Barrel Shifter	0	1
Integer Multiplier	1	16
FP Arithmetic Unit	4	6
FP Multiply Unit	3	10
FP Div-Sqrt Unit	4	19

3.2 Modifications to SimpleScalar-ARM Simulator

In this section we describe the modifications that were done to the Simplescalar ARM infrastructure [30] that was used to implement the proposed technique.

3.2.1 Modifications to ARM ISA Definition Files

To provide the capability of accurately measuring the usage of the various functional units constructed earlier the ARM ISA definition files were modified.

The definition of resource classes for the the ARM architecture in the header file, `machine.h`, was modified. The class `IntBS` was added to provide the capability to perform cycle-accurate simulation considering the barrel shifter as a separate functional unit. The existing separate classes `FloatDIV` and `FloatSQRT` were merged into a single class `FloatDIV_SQRT`. The classes `IntDIV`, `FloatCMP`, and `FloatCVT` were removed since these classes were unused in the ISA definition file, `machine.def`.

The ARM ISA defines a subset of the data processing instructions to be able to use the barrel shifter for shifting one of its register operands before performing the intended operation. It also defines the load and store instructions to be able to affect the index value with the help of the shifter during indirect addressing. However, Simplescalar ARM includes the shifting functionality into the integer ALU and does not treat the barrel shifter

as a separate functional unit. To be able to to this, an instruction flag, `F_SHIFT`, is added to the existing flags in the header file, `machine.h`, to tag the instruction opcodes that may use the barrel shifter. This flag is added to the set of flags defined in the ARM ISA machine definition file, `machine.def`, for those instructions.

3.2.2 Modifications to Sim-profile

The code profiling tool with Simplescalar toolset, `sim-profile` is used for annotating dynamic profiling information with the control flow graph and the loop hierarchy trees for the benchmark program. The control flow graphs of all the functions in a benchmark and their loop hierarchy trees are generated from the disassembled executable binary for the corresponding benchmark and are passed to `sim-profile`. During the execution of the program with typical inputs, the frequencies of execution of all the basic blocks across all the functions are computed. This profiling information is then used to run Algorithm 1 as a post-processing step in `sim-profile`. However, we do not generate code with sleep instructions inserted. Instead, we generate a list of instruction addresses where the sleep instructions should be inserted as output by Algorithm 1. This list is passed to `sim-outorder` and is used to simulate the turning off of functional units as directed by the sleep instruction addresses during the cycle-accurate simulation. In this work, since we restrict ourselves to calculation of leakage energy savings of the functional units, this is an appropriate technique. If the energy calculations for all the components of the processor were required, then code generation would have been essential because the fetching an extra instruction from the memory, decoding it and retiring it would affect the overall energy consumption characteristics of the processor.

3.2.3 Modifications to Sim-outorder

The cycle-accurate simulation tool, `simoutorder`, maintains a resource manager whose interfaces are defined in `resource.h` and whose routines are defined in `resource.c`. The definitions of these interfaces were modified to incorporate the support for calculation of

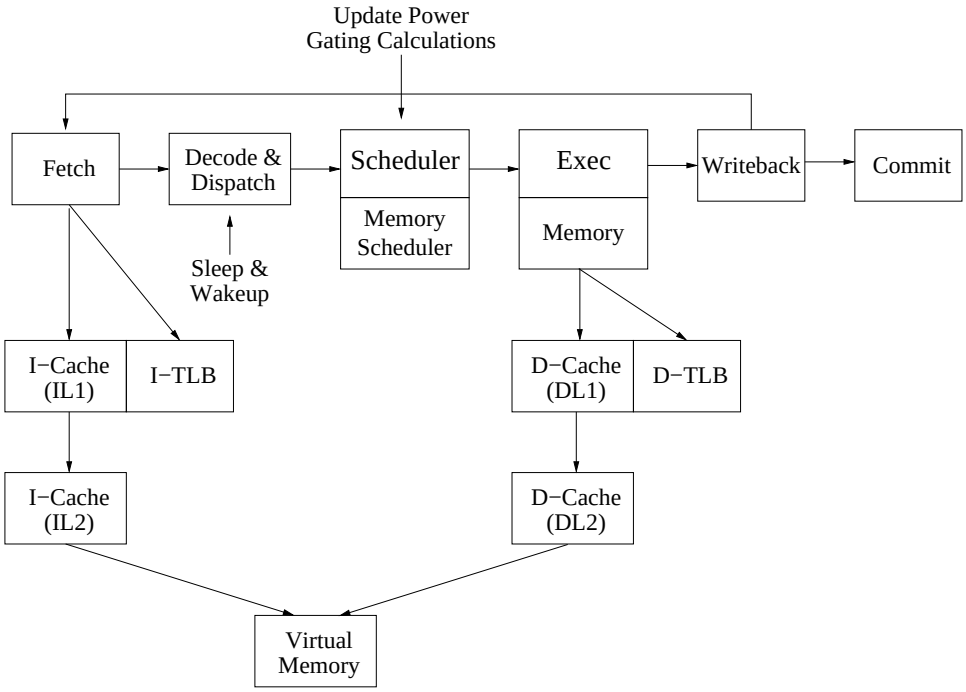


Figure 3.6 Modules Added to Sim-outorder.

leakage energy values during the execution of the benchmark program. Two definitions were added to the resource descriptor structure definition. One was the definition of the energy descriptor which incorporates the energy components tabulated for the functional units further in Table 3.2. The second was the definition of the power gating descriptor which maintains the details of the number of cycles that the functional units are idle and are turned off, number of cycles that the functional units are idle but are not turned off, number of times they are turned off, and number of times they are turned back on.

Figure 3.6 describes the modules added to the the cycle-accurate simulator and are implemented in `sim-outorder.c`. The list of sleep instructions with their addresses is obtained from the post-processing module in `sim-profile`. During the instruction fetch stage, if the address of the instruction matches with any of the sleep instructions, the actual instruction is stalled in the fetch stage for one cycle simulating the fetching of the sleep instruction one cycle before the fetching of the actual instruction. During the next cycle, the scheduling of turning off the intended functional unit is performed since the decoding

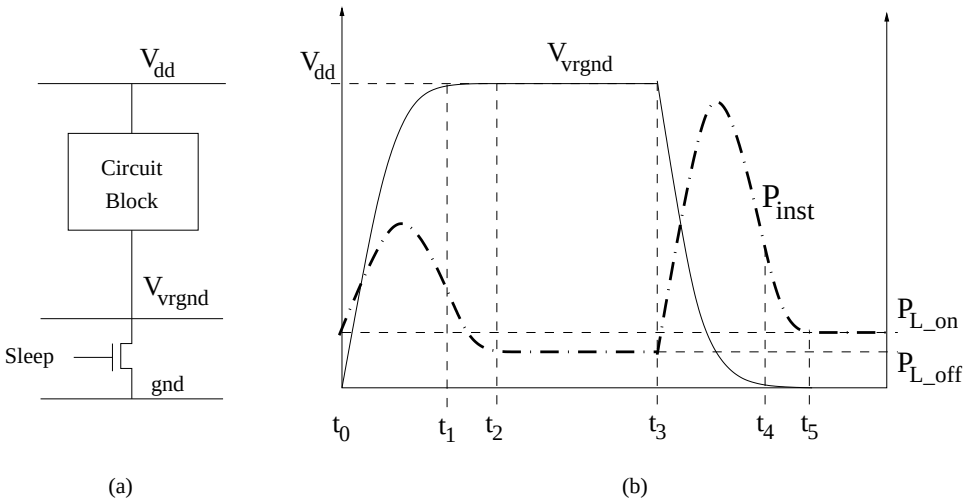


Figure 3.7 Overhead Energy Component Calculations (a) Footer Sleep Transistor Configuration, (b) Significant Time Intervals in Power-Gating.

of the sleep instruction triggers the turning off of the functional unit. The functional unit goes to the OFF status in the following cycle. Whenever an actual instruction is decoded, the functional unit required by that instruction is scheduled to be turned back on. The functional unit resumes the ON status in the following cycle. Each cycle, all the energy components are updated for all the functional units based on their power gated status.

3.3 Energy Component Calculations

Figure 3.7(a) shows the footer sleep transistor configuration and Figure 3.7(b) illustrates the significant time intervals for the calculations of the various components of energy for the power-gated structural components of the functional units. V_{vrgnd} refers to the voltage at the virtual ground. P_{inst} refers to the instantaneous power dissipated by the circuit. At time, t_0, when the sleep transistor is switched OFF, V_{vrgnd} rises to V_{dd} by time t_1. From time t_1 to t_2, all the capacitances in the circuit reach their final charge. During this interval the circuit still dissipates instantaneous power which slowly approaches the steady state leakage power in its OFF state, $P_{L_{OFF}}$. Thus, the overhead energy in deactivating the circuit is the total energy dissipated during the interval t_0 to t_2, which is denoted by $E_{t_0-t_2}$

and is given by the area under the curve for P_{inst} during the interval t_0 to t_2:

$$E_{t_0-t_2} = E_{t_0-t_1} + E_{t_1-t_2} \tag{3.1}$$

It is only after time t_2 that the circuit starts to dissipate $P_{L_{OFF}}$. Similar considerations are made for calculating the overhead energy required while activating the circuit. At time t_3, the sleep transistor is switched ON, V_{vrgnd} falls to V_{gnd} by time t_4. From time t_4 to t_5, all the capacitances reach their final charge. The overhead energy in activating the circuit is the total energy dissipated during the interval t_3 to t_5, denoted by $E_{t_3-t_5}$.

$$E_{t_3-t_5} = E_{t_3-t_4} + E_{t_4-t_5} \tag{3.2}$$

After time t_5 that the circuit starts to dissipate $P_{L_{ON}}$. The calculations of the energy components of the components is performed using HSPICE simulations on the 70nm model files available from MOSIS.

Since the leakage power is proportional to the number of transistors in a circuit, the energy components of each functional unit is estimated as the sum of the energy components of its constituent component instances. For a functional unit r, Δ_r is calculated as,

$$\Delta_r = E_{t_0-t_2} + E_{t_3-t_5} \tag{3.3}$$

Table 3.2 shows the energy values and the threshold lengths that are estimated for each of the functional units.

Table 3.2 Average Energy Components of Functional Units.

Functional unit	Leakage No power-gating (nJ/clock cycle)	Leakage OFF (nJ/clock cycle)	Leakage ON (nJ/clock cycle)	Overhead activation (nJ)	Overhead deactivation (nJ)	L_{th}
Int ALU	1.2687E-04	-	-	-	-	-
Barrel Shifter	3.0134E-04	1.6375E-04	3.1164E-04	5.6821E-02	2.5862E-04	415
Int Multiplier	2.6115E-04	1.4552E-05	2.6530E-04	6.0678E-02	2.3445E-04	530
FP Arithmetic Unit	9.1774E-04	4.9346E-04	9.1997E-04	1.9132E-01	8.8222E-04	453
FP Multiply Unit	8.1344E-04	4.2311E-04	8.1733E-04	1.6639E-01	6.7331E-04	428
FP Div-Sqrt Unit	1.6024E-03	8.7415E-04	1.6373E-03	3.8010E-01	1.3357E-03	523

3.4 Cycle-Accurate Simulation

We used the Simplescalar-ARM distribution [30] for modeling the proposed embedded architecture and Mibench embedded benchmark suite [3] for experimentation. We choose two benchmarks from each of the categories of applications in MiBench which are: Automotive and Industrial Control, Network, Security, Consumer Devices, Office Automation, and Telecommunications. The object code for the program is disassembled using the `gcc` tools for ARM available with the distribution. Static code analysis is performed on the CFG generated for the functions in the source program. We do not inspect standard library functions for insertion of sleep instructions. We used `sim-profile` to perform dynamic profiling of the program and, finally, used `sim-outorder` to perform cycle-accurate simulation after inserting the sleep instructions. The configuration used for `sim-outorder` is for Intel StrongArm-1 microarchitecture [31] which has a fetch and decode width of 1. Table 3.3 decribes the SA-1 processor configuration.

Table 3.3 ARM Processor Configuration.

Fetch Queue (instructions)	2
Branch Predictor	Not-taken
Fetch & Decode Width	1
Issue width	1
Instruction L1 Cache	16K, 32-way
Data L1 Cache	16K, 32-way
L2 Cache	None
Memory (bus width, first block latency)	4-byte, 12 cycle

We compare the leakage energy savings in a processor with power-gated functional units to one in which the functional units are not power-gated. The results are tabulated in Table 3.4. While reporting the energy savings for benchmarks that do not have floating point computations, we report savings for a processor core with and without the floating point units. Since only some of the benchmarks require floating point computations and

embedded system processor architectures are often designed with floating point units only if the targeted applications require floating point computations, we performed simulations for both cases. As can be seen in Table 3.4, the proposed methodology yields average energy savings of 34% for the select set of MiBench suite of benchmark programs. The deactivation of functional units using power-gating results in leakage energy savings without incurring any performance degradation. Since there was less than 0.1% performance degradation, we did not include those numbers in the table.

Table 3.4 Cycle-Accurate Simulation Results.

Benchmark	Category	Instruction Count (in Millions)	Fraction of total cycles for whichunit is power-gated (%)					Savings(%)	
			Barrel Shifter	Integer Multiplier	FP Arithmetic Unit	FP Multiply unit	FP Div-Sqrt unit	Without FP-unit	With FP-unit
bitcount	Auto/Industrial	49.6	16.3	100	100	100	100	31.2	41.3
qsort	Auto/Industrial	43.6	90.5	98.4	100	100	100	37.4	42.1
jpeg decode	Consumer	6.7	2.6	91.6	100	100	100	25.6	40.7
lame	Consumer	97.2	20.6	89.8	13.6	20.2	92.6	-	25.6
dijkstra	network	64.9	16.5	100	100	100	100	37.1	41.5
patricia	network	103.9	0.0	97.1	100	100	100	26.7	40.6
rsynth	office	57.9	0.0	99.7	52.9	14.2	98.7	-	22.6
ispell	office	8.4	3.3	99.2	100	100	100	30.3	41.1
fft	telecomm.	52.7	4.7	0.3	77.9	82.6	100	-	30.4
fft-inverse	telecomm.	65.8	5.7	83.7	62.7	68.6	99.8	-	28.7
rijndael-encode	security	30.7	0.0	93.7	100	100	100	29.6	38.9
sha	security	13.6	0.0	99.4	100	100	100	30.7	39.3

3.5 Summary

In this chapter, we explained the experimental setup performed and the results obtained in this work. We presented the specifications of the functional units with power-gating capability. We described the modifications made to the SimpleScalar-ARM simulator for this work. We also explained the calculations of various components of energy which are required by the compiler for the insertion of sleep instructions. Finally, we presented the results of the leakage energy savings obtained on a set of MiBench embedded benchmarks using our technique.

CHAPTER 4

CONCLUSIONS

In this thesis, we presented a detailed description of a compiler-based technique for applying power-gating in embedded processors. In this work, the calculations of the various energy components involved in the application of power-gating are modeled to achieve high degree of accuracy for each functional unit in the processor core. Thus, the final experimental results reported are more accurate (calculated using HSPICE) than those reported in earlier works. We also discuss how power-gating can be accomplished without any significant overhead or loss of performance by providing a robust mathematical apparatus.

An important aspect of our approach is that the design of functional units are based on assumption that the activation of a functional unit can be accomplished in a single clock cycle that is prior to the cycle when the unit is needed. The advantage is the reduction in the overhead since the above assumption precludes the need for a separate instruction to activate the functional units. Since embedded processors have much less hardware complexity than superscalar processors, this is a reasonable assumption. If the processor is an extremely fine-grained pipeline, then more than a single cycle may be needed for activation in such processors.

We chose to employ a compiler-based leakage reduction technique in this work to avoid the dynamic power overhead involved in implementing a microarchitectural techniques as presented in [8, 23] These techniques are more dynamic in nature and, thus, are more accurate in their predictions about the idle time durations than any static prediction techniques. However, since the task of prediction based on the profiling data is inherently stochastic in nature, a stochastic algorithm is expected to yield more accurate predictions compared to the deterministic approach adopted in this work.

REFERENCES

[1] S. Borkar. Design Challenges of Technology Scaling. pages 23–29. IEEE Micro, 1999.

[2] J Turley. Embedded Processors by the Numbers. Embedded Systems Programming, http://www.embedded.com/1999/9905/9905turley.htm, 1999.

[3] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. MiBench: A free, commercially representative embedded benchmark suite. pages 3–14. IEEE 4th Annual Workshop on Workload Characterization, 2001.

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

[5] J.T. Kao and A.P. Chandrakasan. Dual-Threshold Voltage Techniques for Low-Power Digital Circuits. pages 1009–1018. International Journal of Solid-State Circuits, 2000.

[6] A. Chandrakasan, I. Yang, C. Vieri, and D. Antoniadis. Design Considerations and Tools for Low-Voltage Digital System Design. pages 113–118. ACM/IEEE Design Automation Conference, 1996.

[7] R. Gonzalez, B. Gordon, and M. Horowitz. Supply and Threshold Voltage Scaling for Low Power CMOS. pages 1210–1216. IEEE Journal on Solid-State Circuits, Volume 32, 1997.

[8] D. Duarte, Y.F. Tsai, N. Vijaykrishnan, and M.J. Irwin. Evaluating Run-Time Techniques for Leakage Power Reduction. pages 31–38. Proceedings of 7^{th} Asia and South Pacific Design Automation Conference, 2002.

[9] A. Ferre and J. Figueras. Characterization of Leakage Power in CMOS Technologies. pages 185–188. IEEE International Conference on Electronics, Circuits and Systems, 1998.

[10] Z. Cheng, M. Johnson, L. Wei, and K. Roy. Estimation of Standby Leakage in CMOS Circuits Considering Accurate Modeling of Transistor Stacks. pages 239–244. Proceedings of International Symposium on Low Power Electronic Design, 1998.

[11] M. Johnson, D Somasekhar, and K. Roy. Models and Algorithms for Bounds on leakage in CMOS Circuits. pages 714–725. IEEE Transactions on CAD of Integrated Circuits and Systems, 1999.

[12] Y. Ye, S. Borkar, and V. De. A New Technique for Standby Leakage Reduction in High-Performance Circuits. pages 40–41. Digest of Technical Papers, Symposium on VLSI Circuits, 1998.

[13] S. Bobba and I. Hajj. Maximum Leakage Power Estimation for CMOS Circuits. pages 116–124. Proceedings of the IEEE Alessandro Volta Memorial Workshop on Low-Power Design, 1999.

[14] J.P. Halter and F.N. Najm. A Gate-Level Leakage Power Reduction Method for Ultra-Low-Power CMOS Circuits. pages 475–478. Proceedings of the IEEE Alessandro Volta Memorial Workshop on Low-Power Design, 1997.

[15] S. Mutoh, T. Douskei, Y. Matsuya, T. Aoki, S Shigematsu, and J. Yamada. 1-V Power Supply High-Speed Digital Circuit Technology with Multi-Threshold Voltage CMOS. pages 847–854. IEEE Journal of Solid-State Circuits, 1995.

[16] F. Assaderaghi, D. Sinitsky, S.A. Parke, J. Bokor, P.K. Ko, and C. Hu. Dynamic Threshold-Voltage MOSFET (DTMOS) for Ultra-Low Voltage VLSI. pages 414–422. IEEE Transactions on Electron Devices, 1997.

[17] L. Wei, Z. Chen, M. Johnson, K. Roy, and V. De. Design and Optimization of Low Voltage High Performance Dual Threshold CMOS Circuits. pages 489–494. Proceedings of the 35^{th} Design Automation Conference, 1998.

[18] M. Powell, S. Yang, B. Falsafi, K. Roy, and T. Vijaykumar. Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-Submicron Cache Memories. pages 90–95. International Symposium on Low Power Electronic Design, 2000.

[19] S. Kaxiras, Z. Hu, and M. Martonosi. Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power. pages 240–251. International Symposium on Computer Architecture, 2001.

[20] K. Flautner, Z. Hu, and M. Martonosi. Drowsy Caches: Simple Techniques for Reducing Leakage Power. pages 241–250. International Symposium on Computer Architecture, 2002.

[21] L. Clark, S. Demmons, N. Deutscher, and F. Ricci. Standby Power Management for a 0.18μm Microprocessors. pages 7–12. International Symposium on Low Power Electronic Design, 2002.

[22] S. Dropsho, V. Kursun, D.H. Albonesi, S. Dwarkadas, and E.G. Friedman. Managing Static Leakage Energy in Microprocessor Functional Units. pages 321–332. Proceedings of the 35^{th} IEEE/ACM International Symposium on Microarchitecture, 2002.

[23] Z. Hu, A. Buyuktosunoglu, V. Srinivasan, V. Zyuban, H. Jacobson, and P. Bose. Microarchitectural Techniques for Power Gating of Execution Units. pages 32–37. International Symposium on Low Power Electronic Design, 2004.

[24] S. Rele, S. Pande, S. Onder, and R. Gupta. Optimizing Static Power Dissipation by Functional Units Superscalar processors. pages 261–274. Proceedings of the 11^{th} International Conference on Compiler Construction, 2002.

[25] S. Talli, R. Srinivasan, J. Cook, and L. Eisen. Compiler-Directed Functional Unit Shutdown for Microarchitecture Power Optimization. http://domino.research.ibm.com/acas/w3www_acas.nsf/images/conf06/$FILE/jcook.pdf.

[26] W. Zhang, M. Kandemir, N. Vijaykrishnan, M.J. Irwin, and V. De. Compiler Suppport for Reducing Leakage Energy Consumption. pages 1146–1147. Design, Automation and Test in Europe Conference and Exhibition, 2003.

[27] Y. You, C. Lee, and J.K. Lee. Compiler Analysis and Supports for Leakage Power Reduction on Microprocessors. pages 147–164. ACM Transactions on Design Automation of Electronic Systems, 2006.

[28] Advanced RISC Machines Limited. ARM7 Processor Architecture Data Sheet. http://www.arm.com/documentation, 1994.

[29] Advanced RISC Machines Limited. VFP9-S Vector Floating-point Coprocessor Technical Reference Manual. http://www.arm.com/pdfs/VFP-S_Vector_Floating_Point_Tech_Manual.pdf, 2003.

[30] D. Burger and T. Austin. The Simplescalar Tool Set, version 2.0. Technical Report TR-97-1342, University of Wisconsin-Madison, 1997.

[31] Intel Corporation. Intel StrongARM SA-1110 Microprocessor Developers Manual. ftp://download.intel.com/design/strong/applnots/278240.pdf, 2001.

[32] J.P. Shen and M.H. Lipasti. Modern Processor Design: Fundamentals of Superscalar Processor. McGraw-Hill Science, 2004.

[33] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. Introduction to Algorithms. The MIT Press (Second Edition), 2001.

[34] P. Belanovic and M. Leeser. A Library of Parameterized Floating Point Modules and Their Use. pages 657–666. International Conference on Field Programmable Logic and Application, 2002.

[35] M. Leeser and X. Wang. Variable Precision Floating Point Division and Square Root. pages 47–48. Workshop on High Performance Embedded Computing, 2004.