Leakage power dissipation in LSI chips has been increasing exponentially with device scaling [14], and has grown to be a major component in the total power dissipation today. Among existing techniques to reduce leakage power, power gating is one of the most promising approaches. Power gating is a technique to shut off the power supply and put the target circuit into sleep mode during periods of inactivity by turning off an embedded power switch transistor.

So far, power gating control has been implemented at a coarse granularity both in terms of area and time. For example, IP cores such as CPU or DSP cores in an SoC are power gated and put into sleep mode depending on applications [7,15,16], for example, in an SoC for cell phone applications. IP cores only used at video telephony are powered off when the operation is switched to the voice call. More aggressively, several run-time leakage power reduction techniques have focused primarily on caches in the microprocessor, which occupy a large area on the processor die [3,11]. It is also relatively easy to apply...
power gating to the caches compared to other components in the microprocessor such as the functional units in the microprocessor pipeline.

In contrast, we have been studying more aggressive techniques to power gate internal circuits in the microprocessor in much finer granularity at run-time [9,19]. In [9], the authors present a technique to power gate functional units such as a fixed-point unit and a floating-point unit in a microprocessor. In [19], an approach has been proposed to power gate a group of combinational logic gates by employing an enable signal in a gated clock design. These fine-grained power gating techniques have more opportunities to reduce leakage at run time than coarse-grained power gating techniques.

In order to effectively apply fine-grained run-time power gating to functional units in the microprocessor, it is important to detect/predict when and which unit should be put into sleep mode. This should be done with the help of both hardware and software because there exist dynamically or statically determined idle periods. Thus, we propose an architectural (hardware) technique to tackle idle periods caused by dynamic events and a compiler (software) technique to predict and handle statically determined idle periods. That is to say, the architecture should be built in a way that it can detect the chance of power gating and appropriately put the target unit into sleep mode. Also, it should offer a suitable ISA interface to the software to enable controlling power gating from the software. Compiler should decide which functional unit to be put into sleep mode by analyzing the target source code and predict how long would the idle period be of each functional unit after its use. We show our techniques are effective in reducing leakage power by designing and implementing a real chip with fine-grained run-time power gating.

The next section in the chapter describes (1) a top-down design methodology to implement fine-grained run-time power gating, (2) architecture supports and enhancements to effectively apply power gating, and (3) compiler code generation by precisely analyzing the idle time of target functional units. Based upon these fundamental techniques, we design and fabricate Geyser, an MIPS R3000–compatible CPU with fine-grained run-time power gating, and introduce it as a case study of a real chip implementation in Section 3.3. Finally, Section 3.4 summarizes this chapter.

### 3.2 Fine-Grained Run-Time Power Gating for Microprocessors

In this section, we first describe a top-down design methodology to implement fine-grained run-time power gating, which is composed of the proposal of a design flow to use locally extracted sleep signals. Our technique utilizes the enable signals of gated clock design to automatically partition the target circuit into power gating domains. In Section 3.2.2, we introduce an efficient architectural method to apply fine-grained run-time power gating to the computational units of the classic five-stage RISC-style CPU pipeline. Also, the ISA extension to enable the control of power gating by sleep instructions is introduced. Finally, a code generation technique that analyzes the source code of the program and appropriately inserts the sleep instructions is described in Section 3.2.3.

#### 3.2.1 Design Approach for Fine-Grained Run-Time Power Gating

##### 3.2.1.1 Structure and Generation of Fine-Grained Run-Time Power Gating

#### 3.2.1.1 Exploiting Enable Signals of Gated Clock

Gated clock is a technique to reduce the dynamic power of the clock network by stop toggling the clock when data stored in flip-flops are not updated. During this period, combinational logic gates located at the transitive fan-in of the flip-flops are not required to compute new data. If outputs of the combinational logic gates are not used anywhere else, the logic gates are considered as “idle” and by detecting this period, we turn off the power switches provided to the combinational logic gates, which results in reducing active leakage power. Figure 3.1 shows the basic structure that we use for fine-grained run-time power gating. We fully exploit the enable signals of gated clock design to control both power switches provided to the
combinational logic gates and isolation cells. The isolation cell is composed of low leakage transistors (e.g., high-Vth and thicker gate oxide) and inserted between power-gated and non-power-gated circuits. When the enable signal is zero, power switches are turned off and active leakage current is cut off at the power-gated logic circuits. The isolation cells keep the input voltage of the non-power-gated circuits to avoid signal floating. When the enable signal is one, power switches are turned on and updated data are loaded into the flip-flop.

### 3.2.1.1.2 Power Gating Domain

In actual clock-gated designs, it is likely that more than one enable signals exist. To perform fine-grained run-time power gating for these designs, we propose an idea of “power gating domain” (PG-domain). PG-domain is defined as a group of circuits that are power gated together with a unique enable signal. We describe the PG-domain by using an example shown in Figure 3.2. In this circuit, there are two enable signals EN_A and EN_B which control clock-gating for multi-bit registers regA and regB, respectively. Combinational logic gates enclosed with a dotted line indicated as “Group_A” perform computation only for regA; logic gates in Group_A become idle if regA is not updated. This allows power gating the combinational logic gates in Group_A with the enable signal EN_A. Hence, we refer to Group_A as “PG-domain A.” Similarly, logic gates indicated as Group_B can be power gated using enable signal EN_B, which we refer to as “PG-domain B.”

In contrast, combinational logic gates indicated as “Group_X” influence not only regA but also regB. These logic gates become idle only when both regA and regB are not updated. Therefore, we refer to Group_X as “PG-domain AB” and power gate the domain using both EN_A and EN_B.
Logic gates indicated as “Group_Y” are not power gated because their transitive fan-outs are connected to the output pins. Data at the output pins may be used outside of this circuit, and hence should be kept updated. Thus we do not power gate the logic gates in Group_Y and they do not belong to any PG-domain. As an extension, if this scheme is applied to a coarse-grained run-time power gating where the entire circuit is put into sleep, we put the gates in Group_Y into an independent PG-domain. The PG-domain is controlled by a power switch that is turned off only when the entire circuit becomes idle.

### 3.2.1.1.3 Algorithms to Partition a Circuit into Power Gating Domains

We describe an algorithm to partition a given circuit into PG-domains. Let us assume the circuit depicted in Figure 3.3. First, we focus on a flip-flop and find an enable signal controlling it. For example, flip-flop FF1 is controlled by enable signal EN_A. Next, we traverse the combinational logic network backward until reaching input pins of the given circuit or an output terminal of a flip-flop from the data-input terminal of it. We put a label “A” to all the combinational logic gates that we meet during the traversal, which are the logic gates located at transitive fan-in of FF1. Then we move to the next flip-flop FF2 and find an enable signal of the flip-flop. In this case, the enable signal is identified as EN_A again, so a label “A” is put to logic gates located at transitive fan-in of FF2.

Since flip-flop FF3 is controlled by enable signal EN_B, label “B” is put to logic gates located at transitive fan-in of FF3 if unlabeled. It should be noted that we do not put a label “B” to gates G1 and G2 since they are already labeled “A.” Instead, we put a new label “AB” to G1 and G2 by ripping off the old label. Next, we focus on FF4 controlled by EN_C and put a label “C” to the extracted logic gates in the same way. Because label “AB” is already put to gate G2, we update the label to “ABC.” After we finish labeling logic gates located at transitive fan-in of all the flip-flops, we focus on the output pins of the circuit and perform a similar backward traversal. We put a label “N/A” to the extracted logic gates because they are not power gated in the fine-grained run-time power gating. If the extracted gates are already labeled, we update the label to “N/A.”

After we complete labeling all the combinational logic gates, we create PG-domains according to the labels and a power switch is connected to each PG-domain. For example, logic gates labeled “A” are put into the PG-domain A, while those labeled “AB” are put into the PG-domain AB. Note that logic gates labeled “N/A” are not put into any PG-domain because they are not going to be power gated.

### 3.2.1.1.4 Generation of Control Logic for Power Switches

Since PG-domains are built based on the labels we added, each power switch connected to the PG-domain is controlled by the enable signal corresponding to it. For example, the power switch connected to PG-domain A is controlled by enable signal EN_A. In contrast, the power switch connected to PG-domain AB

![FIGURE 3.3 Example circuit to partition into power gating domains.](image-url)
has to be controlled by both EN_A and EN_B. Since logic gates in PG-domain AB are idle when EN_A and EN_B are both “Low,” EN_A and EN_B are OR-ed and used to control the power switch. A three-input OR gate whose inputs are EN_A, EN_B, and EN_C is added for the power switch connected to PG-domain ABC and the output of the OR gate is connected to the power switch.

3.2.1.2 Implementation Methodology

3.2.1.2.1 Local Virtual Ground (VGND) Scheme

To implement fine-grained run-time power gating, conventional global VGND rail is not effective because partitioning the global rail is extremely difficult. Instead, we use a local VGND scheme in which logic cells and power switch cells within a PG-domain are connected with a local VGND line [13]. To implement this scheme, we modified the existing technology library of logic cells such that the source of NMOS transistor within the cell is disconnected from the real ground rail and is instead connected to a newly created VGND pin. Power switch cell contains an NMOS power switch transistor whose drain and gate are connected to a VGND pin and an enable pin, respectively. The VGND pin of the logic cells are connected to those of power switch cells through a local VGND line, which is routed as an inter-cell wire at the routing stage. We describe the design flow to implement the fine-grained run-time power gating from RTL down to the layout utilizing this scheme.

3.2.1.2.2 Design Flow

Gated clock design is performed in the synthesis step in which we synthesize the gate-level Verilog netlist using the conventional low-Vth standard cell library from RTL description. For the clock-gated netlist, we build a fine-grained run-time power gating structure by using a technique described in Section 3.2.1.1. The clock-gated design is partitioned into PG-domains based on the enable signals. The decision of whether to apply power gating or not is made by considering the break-even point. Break-even point is a certain point in time at which the leakage saving energy and the overhead of switching the power switch are equal. For the PG-domains, power switches are inserted and a control logic for them is generated by adding OR gates. In this way, a fine-grained power gated netlist is generated. The netlist is fed to a placement tool and initial placement is performed. The placement result is given to the power switch optimization engine where power switch sizing is performed (we used CoolPower [4] in our design flow). Power switches are sized such that the voltage bounce at each VGND line may not exceed the user-specified upper limit. Isolation cell insertion to avoid signal floating is also performed at this step. The result of this step is sent to a router and the final layout is generated.

3.2.2 Architectural Technique to Support Run-Time Power Gating

By applying the fine-grained run-time power gating design explained in Section 3.2.1.1, functional units can be shut off in a much finer granularity. Here, we consider applying such an aggressive dynamic control of power gating into a traditional RISC CPU pipeline. For example, multipliers and dividers occupy a large area in such CPU layouts but are not used in every instruction. Such components can be woken up only when they are required by checking the fetched instruction.

However, such an aggressive control has a problem: turning the power switches on and off requires energy, and it takes a certain amount of time for the leakage power to decrease after turning the components off. Thus, in order to reduce the leakage power, the time spent in sleep mode must be more than a certain break-even point. In other words, if the sleep time is less than the break-even point, the energy is increased by applying power gating. The main concern is how to control the sleep signals so that the sleep time will be longer than the break-even point to reduce the energy consumption of the target components.
3.2.2.1 Dynamic Fine-Grained Power Gating for CPU Pipeline

3.2.2.1.1 Target CPU and Its Functional Units

We propose a technique to apply our proposed fine-grained run-time power gating to a classic style in-order pipelined RISC CPU, such as MIPS [10], SPARC [18], Motorola 88000 [2], and DLX [8], which is today commonly used in embedded processors. Standard five-stage pipeline structure consists of Instruction Fetch (IF), Decode (ID), Execute (EX), Memory Access (MEM), and Writeback (WB). *

We select computational units in the EX stage as the functional units to be power gated. Computational units occupy a big portion of the total CPU area, and it is easy to identify their usage based on the fetched instructions. Other leakage-consuming units specific to each processor can also be considered as targets. In our implemented real chip Geyser, we additionally selected the coprocessor 0 (CP0) to be power gated, which will also be explained in this section. CP0 supports the operating system for processing exceptions or interrupts. So it only works in special conditions when the operating system is invoked.

Here, we explain the functional units selected as the target to be power gated.

- **ALU (common arithmetic and logic unit)**: General computational units including addition and subtraction. It can be put into sleep mode when branch, node occurrence probability (NOP), or memory access instructions without address calculation is fetched.
- **Shift unit**: Since the barrel shifter occupies a considerable area but is not frequently used, it is selected as an individual unit.
- **Multiplier unit**: The multiplier unit takes several clock cycles to complete a multiplication.
- **Divider unit**: The divider unit also takes several to tens of clock cycles to complete a division.
- **CP0**: A coprocessor, which is an interface between the CPU and the operating system for handling exceptions and controlling TLB entries. It can be put into sleep mode when the CPU runs in user mode.

3.2.2.1.2 Fundamental Control of Fine-Grained Run-Time Power Gating

Sleep signals for the target units are generated in the sleep controller shown in Figure 3.4.

* As will be introduced in the next section, we implemented a real chip Geyser, which is a MIPS R3000–compatible CPU, so some of the following description will be dedicated to MIPS architecture, but basically the fundamental idea can be applied to other classic style in-order CPUs.
By considering the usage of each computation unit, all units except ALU are put into sleep mode automatically after finishing the operation in EX stage. When an instruction is fetched in the IF stage, the sleep controller must check the fetched instruction and judge which computation unit is to be used for the fetched instruction. Since it takes a certain amount of time to wake (approximately 1 cycle is assumed here) the unit, the detection must be done in the IF stage. For this purpose, a high-speed simple decoder, which only detects which computational unit is to be used, is provided in the IF stage so that the unit can be available for use when required.

Figure 3.5 shows a high-speed detector for generating a wake-up signal considering the MIPS ISA. The detector checks the uppermost 6 bits of the instruction, and judges whether the instruction is a register–register operation or not. If so, the unit to be used is identified by the last 6 bits of the instruction. Otherwise, some immediate instruction or load/store instruction, which uses the ALU, is detected.

Since we are assuming an in-order processor, the pipeline is stalled during both an instruction and data cache miss (and also in the case of hit because our design takes multiple cycles to access the cache). Thus, all units are put into sleep mode when either cache miss/hit signals or a clock wake-up is detected until the data becomes ready. The sleep controller also detects the stall from the multiplier or the divider while waiting for data, and puts all units into sleep mode.

The sleep control for CP0 operates in a completely different manner; it is put into sleep mode when the CPU runs in user mode. The exception must be received in the sleep controller first, which then sends CP0 the wake-up signal, and also sends a signal to the pipeline to stall. Since the contents of registers in CP0 are lost by putting into sleep mode [12], the values of registers that need to be saved are also handled appropriately by the sleep controller.

### 3.2.2.2 Sophisticated Power Control Methods

#### 3.2.2.2.1 Instruction with Power Gating Direction

The fundamental method described earlier has a potential problem that it can possibly increase the leakage power. For example, when multiple multiply operations are executed within an interval of few cycles, the multiplier unit will be woken up as soon as it is put into sleep mode. In such a case, the time spent in sleep mode might be less than the break-even point, and the power overhead of power gating increases the total power. Such a combination of instructions can be detected by the compiler, and we can save such overhead by retaining the power of the multiplier unit after the first computation is finished. For this purpose, we introduce an instruction set with power gating direction encoded in it. The technique to effectively utilize this instruction by the compiler is explained in the next section.

Figure 3.6 shows an example of instruction format with power gating direction. In the MIPS ISA, when the uppermost 6 bits (ope-code) are all zeros, the operation is applied with two registers shown in operand fields, and the type of operation is indicated with the last 6 bits. Instead of all zeros, we use “100111,”
which is not defined in the original ISA, as the uppermost 6 bits to indicate the power gating direction. After executing such instruction, the functional unit is not put into sleep mode, but stay in active mode. The functional unit is kept awake until the next instruction which uses it with the ope-code of all zeros is executed.

### 3.2.2.2 Power Gating Policy Register

The leakage power is sensitive to the temperature and will increase drastically as the temperature rises. Thus, the break-even point is also influenced by the temperature, and tends to be short when the chip becomes hot. In such a situation, it is more beneficial to aggressively apply run-time power gating because power saving is possible in a short period of sleep. Conversely, if the chip is cool, a less-aggressive power gating policy suppresses the overhead. Thus, the policy of power gating should be changed based on the temperature that can be measured by a thermal sensor, or by the amount of leakage current itself, which can be measured by a leak monitor. Both devices can be integrated on chip and the values can be read out by the operating system.

Here we introduce three policies that can be applied:

1. **Policy-1**: The fundamental policy that always puts every unit into sleep mode dynamically after using it or after a stall, including a cache miss/hit.
2. **Policy-2**: A more conservative policy that puts every unit into sleep mode only when a cache miss/hit is detected.
3. **Policy-3**: A policy that does not apply power gating and no unit is ever put into sleep mode.

We introduce a power gating policy register that stores the data that indicates one of the aforementioned policies, and they can be written only in the kernel mode. The operating system controls the sleep mode depending on the data of the thermal sensor, as shown in Figure 3.7. Note that a policy can be selected for each functional unit, because the break-even point differs among each of them. Two bits are needed to represent the policy for computational units, and 1 bit is needed to indicate whether to sleep or not for CP0.

### FIGURE 3.6 Instruction with power gating directive.

### FIGURE 3.7 Power gating policy register.
3.2.3 Compiler-Assisted Run-Time Power Gating Technique

As mentioned in the previous section, compiler can predict the idle period of each functional unit by analyzing the source code of a program. According to this information, each functional unit is selected to be kept awake or put into sleep mode after execution. More precisely, if the predicted idle period of the functional unit is shorter than the target break-even point, the compiler sets the opcode to 100111 so that the target functional unit will not be put into sleep mode after execution.

One of the reasons for long idle periods are long latency events such as cache misses that occur dynamically. As mentioned in the previous section, cache miss/hit events are detected and handled by hardware. On the other hand, compiler analysis is effective for predicting idle times that are statically determined by instruction sequences. Because there are both short and long idle periods, we need to perform a global code analysis that focuses not only inside the basic block or the procedure but also across the procedure calls. By applying a hybrid technique that combines the compiler-assisted power gating and the cache miss/hit triggered power gating, it becomes possible to capture both statically and dynamically determined idle times very effectively. The details of the compiler analysis will be described in Section 3.2.3.1.

3.2.3.1 Overview of the Compiler Analysis

In this section, we describe the compiler analysis to predict the idle time of functional units. Our analysis is analogous to data-flow analysis [1], which is conventionally used in compiler optimization techniques. We analyze a control flow graph (CFG) and obtain the expected idle time for each functional unit. For example, if we want to analyze the idle time of a multiplier, we count the expected number of nodes that lie between the target multiply instruction and the succeeding multiply instruction in the CFG. By assuming each instruction is executed in a single cycle, the aforementioned number of nodes gives the expected idle cycles of the multiplier.

Usually, there are many procedure calls and loop structures in a program. The accuracy of the expected usage interval of functional units can be strongly affected by them. Therefore, we need to analyze across branches or procedure calls for precise prediction. We construct a call graph (CG), which illustrates the relation between procedure calls for interprocedural analysis.

3.2.3.2 Intrprocedural Analysis

First, we describe the analysis within a procedure that has no procedure call in it. The basic framework is similar to a data flow analysis scheme [1]; however, we define real number variables (RNV), which express the expected usage interval of functional units, for each node in the CFG instead of typical data-flow values such as reaching definitions. This analysis depends on the information that is computed in the reverse order of the control flow in a program, because we want to know where the following instruction next uses the target functional unit.

We define RNV for the nodes in a CFG as follows:

\[ \text{IN}_D[s], \text{IN}_P[s], \text{OUT}_D[s], \text{OUT}_P[s]. \quad (3.1) \]

\( \text{OUT}_D[s] \) expresses the expected number of instructions between the node \( s \) and the next instruction that uses the target functional unit. Therefore, \( \text{OUT}_D[s] \) indicates the predicted idle time of the functional unit with the assumption that every instruction is executed in a single cycle. \( \text{IN}_D[s] \) represents the same meaning value as \( \text{OUT}_D[s] \) defined for the point right before node \( s \). \( \text{OUT}_P[s] \) expresses the probability of reaching to the exit point of the procedure from the point after node \( s \) without executing the instruction that uses the target functional unit. \( \text{IN}_P[s] \) has the same value as \( \text{OUT}_P[s] \) defined for the point right before node \( s \).

Next, we give the data-flow equation, which gives the constraint between the variables of nodes. For preparation, we define two constant values \( T_D[s] \) and \( T_P[s] \). These variables are defined for each node in...
the CFG, and their values are determined whether the node uses the target functional unit or not.

\[ T_D[s] = \begin{cases} 0 & \text{if } s \text{ uses the target unit} \\ 1 & \text{otherwise} \end{cases} \quad (3.2) \]

\[ T_P[s] = \begin{cases} 0 & \text{if } s \text{ uses the target unit} \\ 1 & \text{otherwise} \end{cases} \quad (3.3) \]

\( T_D[s] \) expresses the expected idle time of the functional unit from the program point right before the node \( s \) to the program point right after the node \( s \). \( T_P[s] \) expresses the probability of reaching the program point right after the node \( s \) from the program point right before the node \( s \) without executing the instruction that uses the target functional unit. When analyzing a procedure that has no procedure call in it, these two values seem to be trivial because there is always only one instruction between the program point right before the node \( s \) and the program point right after the node \( s \). However, there can be several instructions between the two program point right before and after the node \( s \) in the interprocedural analysis, which we describe later, because of the presence of the procedure call instructions.

By using these values, the data flow equations are given as follows:

\[ I_{N_D}[s] = T_P[s]O_{UT_D}[s] + T_D[s] \]

\[ I_{N_P}[s] = T_P[s]O_{UT_P}[s] \quad (3.4) \]

\[ O_{UT_D}[s] = \begin{cases} q[s] \ast I_{N_D}[s_{suc1}] + (1 - q[s]) \ast I_{N_D}[s_{suc2}] & \text{if } s \text{ is a branch instruction} \\ I_{N_D}[s_{suc}] & \text{(otherwise)} \end{cases} \quad (3.5) \]

\[ O_{UT_P}[s] = \begin{cases} q[s] \ast I_{N_P}[s_{suc1}] + (1 - q[s]) \ast I_{N_P}[s_{suc2}] & \text{if } s \text{ is a branch instruction} \\ I_{N_P}[s_{suc}] & \text{(otherwise)} \end{cases} \quad (3.6) \]

\( s_{suc} \) indicates the following node after the node \( s \) when the node \( s \) is not a branch instruction. \( s_{suc1} \) and \( s_{suc2} \) indicate the following two nodes after the node \( s \) when the node \( s \) is a branch instruction. \( q[s] \) is the probability that the branch node \( s \) jumps to node \( s_{suc1} \). Values \( q[s] \) are control parameters and can be set for each branch node respectively. \( q[s] \) values can be obtained in several ways such as dynamic profiling technique or static branch prediction techniques.

Finally, we give the initial and boundary values for the iterative calculation by equations (Equations 3.4 through 3.6). Initial values for each node \( s \) except the exit node in the CFG are given as below:

\[ I_{N_D}^{(0)}[s] = T_P[s] \ast T_D[s], \quad I_{N_P}^{(0)}[s] = T_P[s], \]

\[ O_{UT_D}^{(0)}[s] = 0, \quad O_{UT_P}^{(0)}[s] = 0. \quad (3.7) \]

Let \( s_{exit} \) be the exit node of a procedure in the CFG, the boundary values are given as below:

\[ I_{N_D}^{(b)}[s_{exit}] = 0, \quad I_{N_P}^{(b)}[s_{exit}] = 1. \quad (3.8) \]

### 3.2.3.3 Interprocedural Analysis

Here, we describe how to handle the procedure calls in a procedure, which is often the case in a program. A pessimistic way to handle the call is to assume that the instruction that uses the target functional unit exists at the entrance of the called procedure. However, it is easy to imagine that it would result in a very conservative and rough prediction. Therefore, we need to apply interprocedural analysis.
In the following, we use a pseudo instruction “jal” as the instruction calling the procedure for explanation. Now we describe the overview of the interprocedural analysis by the following three steps: (1) preprocessing and analyzing the call graph (CG), (2) seeking the transfer constant values of each procedure (described later), and (3) passing the information from the exit point of the program to each procedure.

First, we analyze each procedure as described in the intraprocedural analysis. In steps (2) and (3), the analysis order of procedures is important because we need to exchange information properly between procedures. Therefore, we first analyze the call graph in step (1) to determine the proper analysis order.

In step (1), we decompose the CG into strongly connected components to handle the loops that are formed by recursive procedures or procedures that call each other and make a new graph CG’. Next, we give the post-order label to the nodes in the CG’ through depth first searching. We analyze each strongly connected component in the labeled order in step (2), and analyze them in reverse order in step (3). We analyze the procedures in the same component several times. For example, assume that two procedures A and B are in the same component. Then, we analyze the procedures in the order of A-B-A-B. Here we set the number of iterative times to two, which is the number of procedures in the component, to assure that the information will be passed over the components appropriately.

Next, we describe how to exchange the information between procedures in step (2). The values are given as below:

\[
T_D[\text{prc}] := \text{IND}[\text{sent}], \quad T_P[\text{prc}] := \text{INP}[\text{sent}]
\]  

(3.9)

where

\text{prc} \text{ is the procedure}

\text{sent} \text{ is the entrance instruction of procedure \text{prc}}

We assume that the procedure \text{prc} has been already analyzed in order to obtain the proper \text{IND}[\text{sent}] and \text{INP}[\text{sent}] values. We redefine the constant values \(T_D\) and \(T_P\) for the instructions (nodes) as below:

\[
T_D[s] = \begin{cases} 
0 & \text{(if } s \text{ uses the target unit)} \\
T_D[\text{prc} \text{ called}] & \text{(if } s \text{ is “jal”)}
1 & \text{(otherwise)}
\end{cases}
\]  

(3.10)

\[
T_P[s] = \begin{cases} 
0 & \text{(if } s \text{ uses the target unit)} \\
T_P[\text{prc} \text{ called}] & \text{(if } s \text{ is “jal”)}
1 & \text{(otherwise)}
\end{cases}
\]  

(3.11)

\text{prc} \text{ called} \text{ is the procedure that is called by the “jal” instruction. In step (2), we analyze the procedures according to the order obtained in step (1). We use the definition Equations 3.10 and 3.11 instead of Equations 3.2 and 3.3 used in the intraprocedural analysis. With the same initial and boundary values given by Equations 3.7 and 3.8, we seek the solution through an iterative calculation in each procedure. From the result, we can obtain the constant values \(T_D\) and \(T_P\) of the procedures, which will be used in the analysis of other procedures that call the procedure.}

Finally in step (3), we decide the expected usage interval of functional units at each point by passing the boundary values from the exit point of the program to the exit point of each procedure in reverse order of the control flow direction. In order to analyze a certain procedure, we set \(OUT_D\) values of the “jal” instruction \(s'_{\text{call}}\), which call the procedure to the boundary value \(\text{IND}[s_{\text{sent}}]\) instead of setting the boundary values given by Equation 3.8. Note that the instruction \(s'_{\text{call}}\) usually exists in another procedure. Therefore, we have to analyze the procedures in the proper order to pass the boundary values to each procedure.

In summary, we seek the expected usage interval of functional units through passing the information between procedures appropriately. We analyze each procedure with iteratively using Equations 3.9 through 3.11 and the data flow equations (Equations 3.4 through 3.6).
3.3 Case Study: Geyser-1

In this section, we introduce Geyser [17], a MIPS CPU in which fine-grained run-time power gating is applied to computational units in the execution stage, and is able to control the state of each unit. Putting into sleep mode and waking up from sleep mode are controlled via architecture and software, and each unit can change their state every cycle. Although the fundamental techniques to develop Geyser is introduced in the previous section, the compiler that utilizes the sleep instruction is still under development and results are not shown in this evaluation. Geyser has applied fine-grained power gating in terms of both area and size, and has shown the ability of effectively reducing leakage power in a real chip with real-world applications.

3.3.1 Geyser-1: An MIPS R3000 CPU Core with Fine-Grained Run-Time Power Gating

3.3.1.1 Design Policy

Our first chip implementation trial with fine-grained run-time power gating Geyser-0 [17] did not work because of problems on the layout. To avoid those problems, the second prototype Geyser-1 was designed with the following policies: (1) Only the CPU core with power gating is implemented on a chip. Cache and TLB provided in Geyser-0 chip have been moved outside the chip. (2) The design flow is improved so that no manual edit on the layout is needed. (3) 65 nm Fujitsu’s low power CMOS process is used instead of 90 nm standard process used in Geyser-0. Policy (1) gives serious impact to the CPU design. Because of the pin-limitation problem, part of address/data signals must be multiplexed. Additional delay of such multiplexers, long wires, and I/O buffers to access the cache outside the chip severely degrades the operation clock frequency. Moreover, the electric characteristics of the supported package sets are another problem for high-speed operation. As a result, the maximum clock frequency of Geyser-1 was set to be 60 MHz at the layout stage, which is a very conservative value for operation frequency.

3.3.1.2 Implementation and Required Number of Gates

Geyser-1 was designed in Verilog XL, synthesized with Synopsys Design Compiler 2007.03-SP4, and generated its layout by using Synopsys Astro 2007.03-SP7, like common CPUs. The cell library CS202SN 65 nm 12-metal-layer CMOS from Fujitsu is used as a basic cell design. The core voltage is 1.2 V while the I/O requires 3.3 V. For fine-grained run-time power gating, a limited set of standard cells were modified manually to separate the VGND. Such PG cells are replaced with the standard cells between the place and route. In this implementation, we could not modify all standard cells into PG cells because of the limitation of design time.

Optimum number of sleep transistors are inserted in the post layout netlist by the Sequence Designs tool. Cool Power 2007.3.8.5. The condition of optimization is as follows: when all gates that share a sleep transistor work together in the active mode, the maximum level of VGND is less than 0.2 V. The optimized netlist is read by Astro to generate the final mask patterns. Table 3.1 shows the area of each target component. PS shows the area of power switches while IsoCell stands for the area of isolation cells. It appears that 56% of cells are occupied with the multiplier and divider. The overhead of sleep transistors and isolation cells are about 5.4%–12.6%.

Figure 3.8 shows the layout of Geyser-1. The chip size is 2.1 mm × 4.2 mm. The black boxes are target components of fine-grained run-time power gating. The four small black boxes located near each corner are leakage monitors.
TABLE 3.1 Area Overhead of Fine-Grained Run-Time Power Gating

<table>
<thead>
<tr>
<th></th>
<th>Total (μm²)</th>
<th>PS (μm²)</th>
<th>IsoCell (μm²)</th>
<th>Overhead (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>3,752.8</td>
<td>296.4</td>
<td>79.2</td>
<td>10.3</td>
</tr>
<tr>
<td>Shifter</td>
<td>3,078.0</td>
<td>298.8</td>
<td>76.8</td>
<td>12.6</td>
</tr>
<tr>
<td>Multiplier</td>
<td>23,863.6</td>
<td>1762.0</td>
<td>153.6</td>
<td>8.5</td>
</tr>
<tr>
<td>Divider</td>
<td>27,918.4</td>
<td>1301.2</td>
<td>153.6</td>
<td>5.4</td>
</tr>
<tr>
<td>Others</td>
<td>46,304.4</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

3.3.2 Evaluation of Geyser-1

3.3.2.1 Operation Frequency of Geyser-1

First of all, we evaluated the maximum clock frequency of Geyser-1. Here, we define “PG (power gating) mode” in which the fine-grained run-time power gating is applied in policy (1), which is the fundamental policy. By using the instructions with the direction, all units can work without being put into sleep mode. This mode is called the “ACT mode.” A simple benchmark program is executed, and it appears that the maximum operation clock frequency is 60 MHz in both modes. The computation results were confirmed to be correct. This shows that the wake-up mechanism can correctly work at 60 MHz clock frequency.

3.3.2.2 Break-Even Point Analysis

In order to evaluate the effect and overhead of run-time power gating mode, we evaluated the break-even point of each component on the real chip. Figure 3.9 shows the break-even point of the multiplier unit. The test program is a simple loop that consists of a multiplication, some interval cycles, and a return to the loop entrance. The longer the interval becomes, the less frequently the state transition overhead occurs, i.e., the larger the power savings by power gating grows. So we could investigate the break-even point by changing the interval cycles and comparing the power difference between PG mode and ACT mode. The curve in the graph shows the difference. Positive value means that the state transition energy is larger than the leakage energy saved by power gating. When the operation clock frequency is 50 MHz and the temperature is 25°C, the break-even point becomes 44 clock cycles. That is, the module should be in the sleep mode if the interval of two consecutive “multiply” instructions is larger than 44 clock cycles. The compiler must use the instruction directive if the interval of two instructions is predicted to be smaller than the break-even point.

FIGURE 3.8 Layout of Geyser-1.
3.3.2.3 Leakage Power Reduction in Power Gating Mode

First, we evaluated the power consumption with all target units set in the sleep state and ACT mode when the clock is stopped. In this case, dynamic power is not consumed and only the leakage power is consumed. The results are shown in Figure 3.10. It shows that the power gating can reduce the leakage power for about 5% at 25°C. When the temperature grows, the leakage power savings of power gating becomes larger.

3.3.2.4 Evaluation Using Benchmark Programs

To evaluate Geyser-1, we used two programs from MiBench [5] benchmark suite: Quick Sort (QSORT) from mathematics package and Dijkstra from the network package. Also, DCT (discrete cosine transform) from the JPEG encoder program was selected as an example of media processing. Unfortunately, the delay of the Block RAM equipped inside the FPGA becomes large, so the evaluation was done with a 10 MHz clock frequency.

Figures 3.11 through 3.13 show the power consumption of three benchmark programs. The leakage power savings of run-time power gating mode is the largest in Dijkstra, which does not use the multiplier and divider. The savings are 8%–24% of the total power consumption, which is more than the values shown in Figure 3.10. However, it is not strange since the computational units put into sleep mode also reduce the dynamic power that is accounted in the ACT mode. The power reduction of QSORT, which
FIGURE 3.11  Power consumption of Dijkstra.

FIGURE 3.12  Power consumption of QSORT.

FIGURE 3.13  Power consumption of DCT.
sometimes uses the multiplier and divider, is from 4% to 29%, while that of DCT, which uses multiplier, frequently is 3%–17%. A large portion of the power reduction in DCT comes from the dynamic power reduction in run-time power gating mode, since most intervals between instructions using multiplier are not solong as the break-even point. The power reduction numbers are not as worse when compared with coarse-grained power gating technique [6], which fundamentally cannot be applied to computational units of the CPU. That is, it is demonstrated that our fine-grained run-time power gating can effectively reduce the computation power in the CPU.

As shown in this section, Geyser-1, a prototype MIPS R3000 CPU with a fine-grained run-time power gating is available. The evaluation results with the real chip reveals that the fine-grained run-time power gating mechanism works without electric problems. It reduces the leakage power by 7% at 25°C and 24% at 80°C. The evaluation results using benchmark programs show that the power consumption can be reduced from 3% at 25°C to 30% at 80°C.

3.4 Conclusion

This chapter presented fundamental techniques to enable fine-grained run-time power gating in a microprocessor: design methodology to implement fine-grained run-time power gating in the circuit level, an architectural technique to handle dynamically occurring idle periods with run-time power gating and the ISA extension to enable the control of power gating from the software, and a sophisticated compiler analysis to predict the length of statically determined idle periods together with the code generation. By utilizing the aforementioned techniques, Geyser-1, an MIPS R3000–compatible CPU with fine-grained run-time power gating is developed. The evaluation of Geyser-1 with real-world applications showed that fine-grained run-time power gating works without problems, and we can effectively reduce the leakage power of computational units in the CPU.

References


