Automatic Insertion of Low Power Annotations in RTL for Pipelined Microprocessors

Vinod Viswanath, Jacob A. Abraham
Computer Engineering Research Center
University of Texas at Austin
vinod,jaa@cerc.utexas.edu

Warren A. Hunt, Jr.
Department of Computer Sciences
University of Texas at Austin
hunt@cs.utexas.edu

Abstract

We propose instruction-driven slicing*, a new technique for annotating microprocessor descriptions at the Register Transfer Level (RTL) in order to achieve lower power dissipation. Our technique automatically annotates existing RTL code to optimize the circuit for lowering power dissipated by switching activity. Our technique can be applied at the architectural level as well, achieving similar power gains. We demonstrate our technique on architectural and RTL models of a 32-bit OpenRISC processor (OR1200), showing power gains for the SPEC2000 benchmarks.*

1. Introduction

We propose a new technique for low power microprocessor design, a fine-grained clock gating scheme implemented at the RTL or the architectural level which utilizes the program structure of the model. Our algorithm automatically identifies fine-grained blocks of circuit which are not used on any given cycle during the execution of a particular instruction, and shuts them down. This scheme of slicing the circuit based on the instruction being executed is termed *instruction-driven slicing*.

All prior approaches towards analyzing and optimizing RTL and architectural models for lower power dissipation suffer from modeling the power dissipation for very specific hardware structures. Besides, they do not make use of the program structure or the dataflow information (statically or dynamically) available at the architecture and RT-levels.

1.1. Contributions of this work

We propose an algorithm to statically reduce switching power dissipation of a microprocessor. For any given instruction, when it is decoded, we have sufficient information to recognize what resources are required to execute that instruction. We introduce the concept of an *instruction-driven slice*. An *instruction-driven slice* of a microprocessor design, is all the relevant circuitry of the design (a slice of the RTL program) required to take the life cycle of the instruction to completion (decode, execute, writeback *etc.*).

This material is based upon work supported by the Defence Advance Research Projects Agency (DARPA) under Contract NBCH30390004.

The primary idea is that, given a microprocessor design, depending on the instruction, we identify the instruction-driven slice, and shut off the rest of the circuitry. This might include gating out fine-grained parts of various processor blocks depending on the instruction, or gating out the floating point execution units during integer ALU execution, or gating out the memory units during ALU operation, or turning off certain FSMs in various control blocks because the instruction-driven slice provides exact value constraints on their inputs, and so on.

One way of implementing this idea is to add the instruction-driven slice identification and turning off operations to the RTL code as annotations. At the netlist level, we do not have sufficient information to identify an instruction-driven slice. The advantage of annotating the RTL is that the circuitry relevant to perform these tasks is automatically generated by the synthesis tool along with the rest of the netlist.

We have implemented this on OR1200, a Verilog implementation of the OpenRISC [8] architecture. This technique and the same annotations can also be inserted at the architectural level. We have implemented an architectural model of the same OR1200 processor and simulated it with and without the annotations in the SimpleScalar tool set [3] and estimated the power dissipation using SimWattch [7].

1.2. Prior work

In VLSI circuits that use well-designed logic-gates, switching activity power accounts for a substantial amount of total power dissipation (sometimes close to 90% [5]). Over the years, a host of optimization methods at various levels of circuit abstraction have been implemented to reduce switching activity power dissipation. We direct the interested reader to [9] for a survey of many of these optimizations. In the rest of this article, we will concentrate on minimizing switching activity power at the RTL and higher (architectural) levels of abstraction.

An important aspect of optimizing power at the RT-level is to first develop a framework for analyzing the power dissipation at an architectural level. The traditional way has been to translate the given high level architecture description to gates (netlists) and then use reasonably accurate low-level power analysis engines. This method is infeasible if we want to evaluate a large number of design choices. Brooks *et. al* in [2] present a framework for analyzing power dissipation at the architecture level.

3-9810801-0-6/DATE06 © 2006 EDAA

We use this framework to estimate the efficacy of our optimizations at the architectural level.

Most initial work in optimization for low power at the RT-level is focussed on power analysis and reduction in caches [16], [21]. This is chiefly because embedded microprocessors, historically the reason for low power design, used to devote nearly 40% of their power budget to caches. Besides, caches are regular structures and are more easily modeled than other circuits. Other attempts include power reduction by reducing needless speculation in branch predictors [6], gated clocks in integer ALUs [1], *etc.*

Inserting annotations using instruction-driven slicing is explained in detail in Section 2. Our overall methodology and integrated tool flow is detailed in Section 3. We also give an algorithm for automatically inserting the annotations into the RTL code. Section 4 explains instruction-driven slicing in the RTL as well as the architectural models of OR1200, a pipelined implementation of OpenRISC. Results from running SPECINT2000 benchmarks on these models and some comparison metrics are also presented. Some conclusions and future directions are discussed in Section 5.

2. Instruction-Driven Slicing

Program slicing has been well studied in the context of software engineering [11], programming languages [18], and more recently, in the context of slicing hardware description languages [10], [20]. We define a new notion of program slicing for microprocessor descriptions, *viz.,* slicing based on the instruction which is being executed. Given a microprocessor design and an instruction, *instruction-driven slicing* identifies a slice of the abstract program graph of the microprocessor design corresponding to all the relevant circuitry needed to execute that instruction.

The cone of influence of a variable in a program is the set of all program statements which depend on the variable. Any hardware design written in Verilog (at the RT-level) can be thought of as a control flow graph, henceforth referred to as program graph. Traditional program slicing on a variable can be thought of as reducing the program to retain only the slice of the program graph which is within the cone of influence of the variable. Instruction-driven slicing on the other hand, identifies the slice of the program graph which is within the cone of influence of an instruction. The cone of influence of an instruction is the slice of the microprocessor circuitry required to execute that instruction from start to finish. In terms of the RTL program graph, it is a slice of the program which is in the cone of influence of the semantics of an instruction. More specifically, it is the the union of the cone of influence of each of the variables affected by the instruction.

There are two parts to instruction-driven slicing. First, we need to identify the instruction-driven slice. Next, we need to isolate the rest of the circuit by identifying the flops governing the rest of the circuit and gating them out. Since we are slicing based on an instruction or a type of instruction (for example, an ALU instruction, or an LSU instruction, *etc.*), we can obtain a slice both at the RTL and the architectural level models.

Turning off sub-circuits is a well-researched topic [13], [19], and in addition to instruction-driven slicing. We can implement more sophisticated algorithms to determine the sub-circuits to be turned off and the logic required to perform the disabling. The innovation is to automatically identify the sub-circuits in the context of the execution of a particular instruction by leveraging available high level information about instructions and functional units at the RT-level, which is not available to the traditional transistor level optimizing tools. In fact, it turns out that our algorithm to automatically identify an instruction-driven slice introduces much more fine grained gated clocks than prior art automatic methods. We report these findings in Section 4.

Instruction-driven slicing at the architectural level is carried out exactly the same way as at the RT-level. The overall structure available at the architectural level is the same as the RTL model. The key difference is that the architectural model is more abstract than the RTL model. This in turn means that the clock gating due to instruction-driven slicing is more coarse-grained in the architectural model than in the RTL model.

3. Our Technique

3.1. Instruction-Drive Slicing Algorithm

We give an algorithm for automatically identifying instruction-driven slices, given a set of instructions and a microprocessor design. The instruction-driven slicing algorithm for RTL models is given in Figure 1.

The inputs to the algorithm are an RTL model (vRTL) of the microprocessor, and a set of instructions insts which will be executed on that model (given by the ISA of the microprocessor). First, we parse the vRTL model to generate an abstract graph of the program, called the Abstract Syntax Program Graph (ASPG). The nodes of an ASPG are the data computing/modifying statements of the design, whereas the edges of the ASPG define the control flow of the RTL program. We have modified the Verilog parser from vl2mv code distributed with VIS-2.0 [17] to generate our ASPGs.

Now, we traverse the ASPG for each instruction and slice the ASPG. The graph traversal algorithm is a two-pass algorithm. In the first pass we identify variables affected by the instruction driving the slicing and the cone of influence of those variables. Along with this, we compute the condition predicates that are true for every pipeline stage. In the second pass, we identify parts of the ASPG governed by flops[1] which are outside the identified cone of the first pass. These parts of the ASPG are gated out[2]. If there is already gating logic on any of these

1 We use the term *flop* loosely to mean a single-bit storage element with an enable signal.

2 We implement the gating out, not by preventing clock from switching, but setting and unsetting the enable on every flop. We assume that all flops have an enable signal. Also, because of such a gating out mechanism, there is no

Algorithm instruction-driven-slicing (input: vRTL, insts; output: aRTL).
[1] Parse vRTL to obtain the ASPG (Abstract Syntax Program Graph).
[2] For each instruction *i* in insts repeat
Begin loop
[3] Slice the ASPG for instruction *i*
[4] Traverse the ASPG
[5] Add annotation variables if such a block is found
[6] If a particular flop is already gated by a previous annotation, then
add the current annotation as an additional signal
[7] Return the annotated ASPG
End loop
[8] Generate Verilog code for the annotated ASPG (aRTL).
End.

Figure 1. Overview of the Instruction-driven Slicing Algorithm for RTL.

flops, then the algorithm adds to the existing logic in an optimized fashion.

Lastly, we reverse the process of the parser, and generate Verilog code for this annotated ASPG (aRTL).

The time complexity of our algorithm is linear in the size of the program graph. A point of note is that the algorithm may not be able to identify every flop outside the slice. The computing the cone of influence part of our algorithm is based on prior algorithms with guaranteed correctness [20]. The correctness result guarantees that the generated slice is always an over-approximation, *i.e.*, the annotation insertion is guaranteed to be a functionality preserving transformation.

We have implemented our instruction-driven slicing algorithm on the ASPGs. The advantage of this is that without any loss of generality, we can apply the algorithm on any ASPG, irrespective of whether the ASPG was generated from Verilog RTL or from SimpleScalar architectural C models. The algorithm for instruction-driven slicing on the architectural model remains the same, except for parsing the model into ASPGs and generating annotated models from ASPGs. We have implemented our instruction-driven slicing algorithm in C.

The advantage of decoupling the algorithm from the model is that the algorithm can now be treated as a transform engine, which is a part of the tool chain.

3.2. Methodology

We have implemented a methodology to incorporate instruction-driven slicing into the design flow. Figure 2 describes the overall implementation strategy of our technique. We have designed the tool-flow in order to incorporate instruction-driven slicing as a part of the traditional design flow.

In order to demonstrate our technique we have built the following tool-chain. We start with the Verilog RTL (`vRTL`) and the architectural models. The RTL code is annotated with

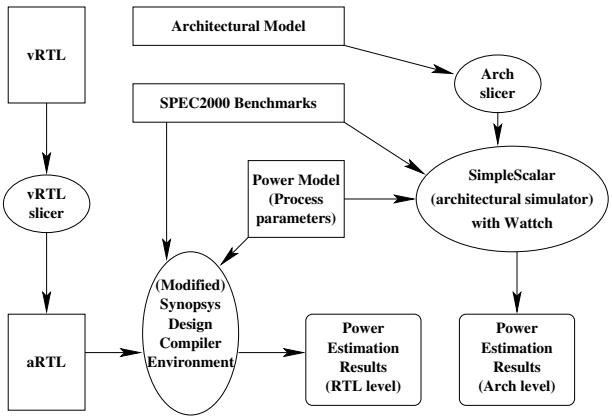


Figure 2. Incorporating *Instruction-driven slicing* **into the design flow.**

instruction-driven slicing annotations to obtain the `aRTL`, by the previously described algorithm (Figure 1). The `aRTL` code, process parameters for power estimation, as well as the benchmark SPECINT2000 files are fed to the Synopsys Design Compiler Environment. We have modified and set up the Synopsys Design Compiler Environment as an integrated tool which can take SPEC benchmarks and RTL code, synthesize the RTL code and determine the power consumption due to switching activity. All SPEC benchmarks were run for 5 hours, and not to completion.

Along a parallel path, we start with the architectural model of the design. Our model is written compatible with the SimpleScalar Tool Set [3]. The model is annotated with instruction-driven slicing annotations and fed as input to the SimpleScalar with Wattch environment. Wattch is an architecture level power estimator [2], [7]. We also modified the `power.h` file in this environment to reflect the same process parameters as used for the RTL power estimation.

Our aim in building this parallel power estimating setup at two levels of design hierarchy is two-fold. First, we wish to

added clock skew, and at the same time, there is no dynamic power saved in the clock distribution network.

show that the dataflow and structure information available at these levels can be usefully exploited to optimize the design for lower power consumption. Second, our technique of automatically adding annotations is scalable to many levels of design hierarchy. The only caveat is that since the architecture model is more abstract than the RTL model, the slicing induced clock gating is coarser for the architecture model than the RTL model.

4. OR1200 - a Pipelined OpenRISC Processor

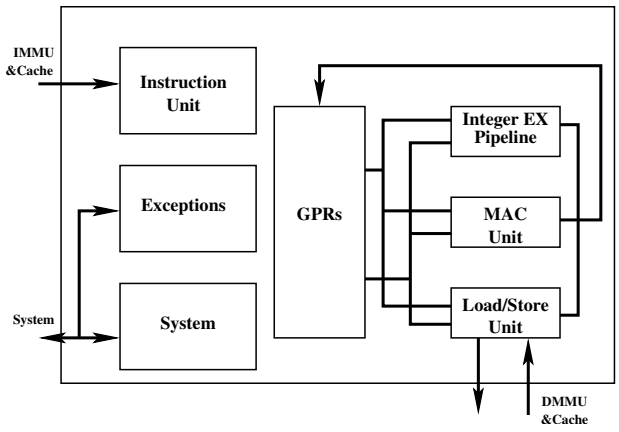


Figure 3. OR1200 Processor Block Diagram.

In order to demonstrate the efficacy of our technique, we have chosen a very complicated, state-of-the-art microprocessor as our example. OR1200 is a pipelined microprocessor implementing the OpenRISC instruction set architecture. We have implemented the architectural model of OR1200 compatible with SimpleScalar (`sim-or1200`). In the rest of this section, we first give a description of the processor itself, and then give our results from running our technique on these models.

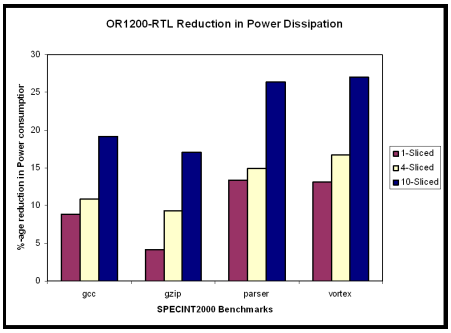
4.1. OR1200

We use the OR1200, a publicly available processor for our experiments. The specification manual of the OR1200 is at [8] and the source code of its implementation in Verilog RTL can be obtained from [15]. The OR1200 is a 32-bit scalar RISC with Harvard microarchitecture, 4 stage integer pipeline, virtual memory support (MMU) and basic DSP capabilities. OR1200 is intended for embedded, portable and networking applications.

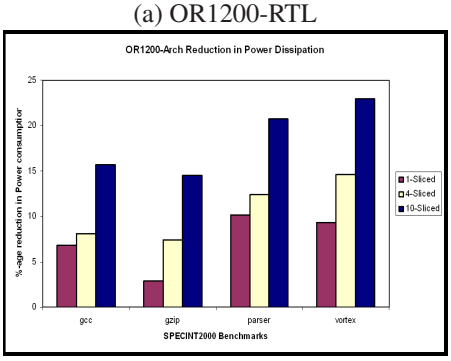
Figure 3 shows the block diagram of the CPU of the OR1200 processor. The instruction unit implements the basic instruction pipeline, fetches instructions from the memory subsystem, dispatches them to available execution units, and maintains a state history to ensure a precise exception model and that operations finish in order. The execution unit must discern whether source data is available and has to ensure that no other instruction is targeting the same destination register. OpenRISC 1200 implements 32 general-purpose 32-bit registers. The load/store unit (LSU) transfers all data between the general purpose registers and the CPU's internal bus.

In this experiment we use TSMC CLO18G [14], a `0:18` m generic process technology to estimate the power dissipation.

4.2. Results for OR1200-RTL



(a) OR1200-RTL



(b) OR1200-Arch

Figure 4. OR1200 reduction in power dissipation for SPECINT2000 benchmarks.

The RTL annotations were automatically generated and inserted in the OR1200 RTL in this experiment. The results are shown in Figure 5(a). It is important to note that these numbers are on models of the processor, and were not originally designed to be power-efficient. The key result therefore, is the percentage reduction in power dissipation. The results are summarized in Figure 4(a). In the best case, we see a 25% reduction of dynamic power dissipation and 20% on an average.

Our power estimation tool (Synopsys power compiler) also automatically gates the clock. However, there is not much overlap between those and the ones we add automatically. Primarily this is because our flop disable logic is extremely fine grained and is not on the clock distribution network. Both the unsliced and the sliced RTL go through the same additional clock gating and hence the percentage reduction we obtain is in addition to what was automatically added by the synthesis tool.

Figure 6 depicts the power-vs-timing and power-vs-area tradeoffs. The normalized `Energy Area` product decreases consistently with increased slicing. This means that as far as increase in area is considered because of additional logic, it is

SPECINT2000 Benchmarks	Unsliced Power Dissipation	1-Sliced Power Dissipation	4-Sliced Power Dissipation	10-Sliced Power Dissipation
gcc	1.89 mW	1.72 mW	1.69 mW	1.53 mW
gzip	1.44 mW	1.38 mW	1.31 mW	1.19 mW
parser	2.12 mW	1.84 mW	1.80 mW	1.56 mW
vortex	2.33 mW	2.02 mW	1.94 mW	1.70 mW
Average	1.95 mW	1.74 mW	1.68 mW	1.49 mW

(a) OR1200-RTL Power dissipation results for SPECINT2000 benchmarks

SPECINT2000 Benchmarks	Unsliced Power Dissipation	1-Sliced Power Dissipation	4-Sliced Power Dissipation	10-Sliced Power Dissipation
gcc	2.04 mW	1.90 mW	1.87 mW	1.72 mW
gzip	1.67 mW	1.62 mW	1.55 mW	1.43 mW
parser	2.32 mW	2.08 mW	2.03 mW	1.84 mW
vortex	2.51 mW	2.28 mW	2.14 mW	1.94 mW
Average	2.14 mW	1.97 mW	1.90 mW	1.73 mW

(b) OR1200-Arch Power dissipation results for SPECINT2000 benchmarks

Figure 5. OR1200 Power dissipation results after slicing on `1`, `4` and `10` instructions.

not a problem since we are gaining substantially in terms of reduced power. The same result shows up from the `Energy Delay`^{2} product as well. [12] introduced `Energy Delay`^{2} product as an efficient measure of energy-vs-delay tradeoff since it represents a voltage independent metric. Therefore, independent of device supply voltage, gains because of lower power dissipation consistently offset the increased area and delay. Also, in certain timing critical blocks, our algorithm can be tuned to slice more coarsely to meet the timing requirements of that block.

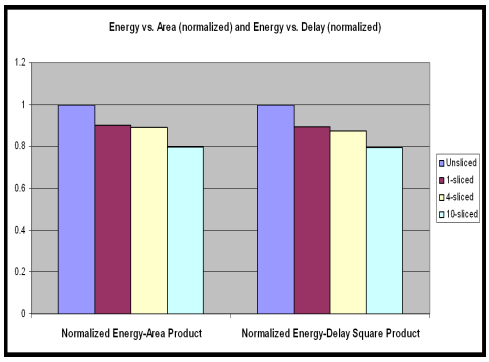


Figure 6. OR1200-RTL Power reduction compared to increase in area and delay due to slicing. The first set of comparisons depicts the normalized `Energy Area` product. The second set depicts the normalized `Energy Delay`^{2} product.

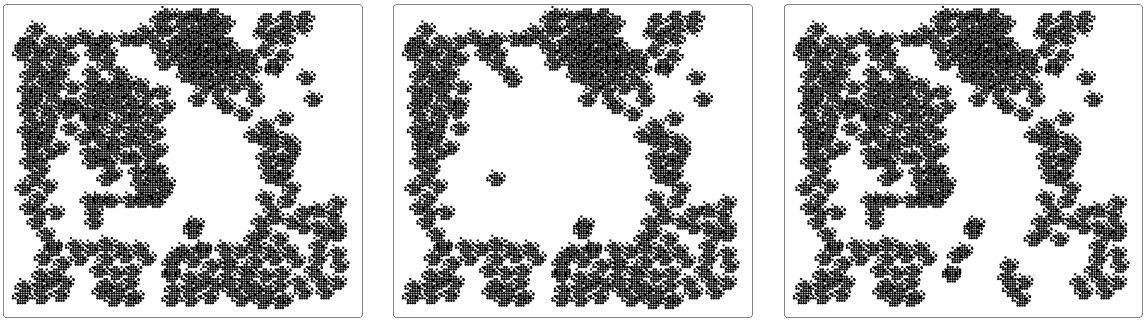
We also synthesized our design and ran it through a place-and-route tool [4], both before and after the slicing. The design contained `3287` flops before slicing. In the unsliced version, all `3287` enables are treated as on as shown in Figure 7(a). After instruction-driven slicing on `l.add`, `1413` flops are disabled during the course of execution of the `l.add`. Figure 7(b) shows the flop distribution after slicing on the `l.add` instruction. The parts of the chip that are lit are all the enables on the flops which are on during the execution of the `l.add`. In the unsliced layout, the entire chip is on, as opposed to the sliced layout where we can clearly see the fine-grained clock gating induced by our algorithm. Figure 7(c) shows the same comparison for a load (`l.lw`) instruction (`831` flops are disabled in this case). The flop distribution in Figure 7 is based on preliminary floor plan estimate, whereas, the number of enables in each case is accurate.

The inserted annotations introduce additional flops into the design. For the OR1200 RTL we found that the number of additional flops was less than `1%` of the total number of flops. On the same count, the additional switching power due to the additional logic is also less than `1%` of the total power dissipated. The additional logic will also cause increased leakage power. Since we have no model to measure the static power dissipation, we do not have a measure of this. However, since the percentage of additional logic is so low compared to the overall power reduction, we do not expect this to be a problem.

4.3. Results for OR1200-Arch

We ran the same benchmarks on our architectural model `sim-or1200`. The results are shown in Figure 5(b). `sim-or1200` absolute power dissipation estimations were more than the RTL estimations. The percentage improvement observed was also lesser than in the RTL model. We believe this behavior is a direct correlation of how fine-grained the clock gating is. Also, since the architectural model is more abstract than the RTL model, it is natural to expect lesser gains on the architectural model. The results are summarized in Figure 4(b).

Instruction-driven slicing is unique because it tries to enforce a semantics (the semantics of the instruction being executed, as given by the program graph) on the flops one is trying to shut-off. This does not preclude the use of netlist level power optimization techniques. To what extent a netlist level



(a) Unsliced flop distribution (3287 flops enabled)

(b) After slicing on `l.add` (1874 flops enabled)

(c) After slicing on `l.lw` (2456 flops enabled)

Figure 7. Flop distribution effect of instruction-driven slicing on `l.add` and `l.lw` in the OR1200 RTL.

optimization is anticipated by our method is not clear. On the contrary, it is clear that our algorithm by virtue of operating at the RTL and architectural levels employs a host of optimizations which are not visible at the netlist level.

5. Conclusions

In this article, we have proposed *instruction-driven slicing*, a new technique to automatically annotate RTL for reducing power dissipation by switching activity. We have implemented the *instruction-driven slicing* algorithm and have incorporated it into the design flow tool-chain. We have automatically sliced the RTL and architectural models for OR1200, a pipelined implementation of the OpenRISC instruction set architecture. We have used our tool-chain to test our methodology on this processor and have obtained encouraging results.

Our algorithm is particularly suited for in-order pipelined processor designs. It can be applied to out-of-order superscalar processors too. However the reduction in power will be substantially less than the OR1200 case since there might be multiple instructions in flight in any pipeline stage, thereby reducing the amount of logic we can actually shut off.

Although our algorithm is conservative, it automatically identifies a close-to optimal set of flops. Our instruction-driven slicing algorithm can be thought of as a wrapper to implement more sophisticated methods of identifying flops which control the circuitry outside the slice.

All previous program slicing algorithms slice the program graph syntactically. The key innovative idea in this technique which separates it from previous techniques is that it computes a cone of semantic influence of an instruction. It is noteworthy that this information is available only at the RTL and architectural level of description, but is lost at the netlist level where most prior power optimization techniques reside.

References

[1] D. Brooks and M. Martonosi. Dynamic thermal management for high-performance microprocessors. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture (HPCA'01)*, page 171. IEEE Computer Society, 2001.

[2] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th annual international symposium on Computer architecture*, pages 83–94. ACM Press, 2000.

[3] D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25(3):13–25, 1997.

[4] Cadence placement-and-routing (PNR) tool.

[5] A. Chandrakasan, S. Sheng, and R. Brodersen. Low-power cmos digital design, 1992.

[6] D. Chaver, L. Piñuel, M. Prieto, F. Tirado, and M. C. Huang. Branch prediction on demand: an energy-efficient solution. In *Proceedings of the 2003 international symposium on Low power electronics and design*, pages 390–395. ACM Press, 2003.

[7] J. W. Chen, M. Dubois, and P. Stenström. Integrating complete-system and user-level performance/power simulators: The simwattch approach. *Proceedings of International Symposium on Performance Analysis of Systems and Software*, 2003.

[8] D. Lampret *et al*. OpenRisc 1000 Architecture Manual, *http://www.cs.utexas.edu/ vinod/patmos05/or1200_spec.pdf*. 2003.

[9] S. Devadas and S. Malik. A survey of optimization techniques targeting low power vlsi circuits. In *Proceedings of the 32nd ACM/IEEE conference on Design automation*, pages 242–247. ACM Press, 1995.

[10] Edmund M. Clarke and Masahiro Fujita and Sreeranga P. Rajan and Thomas W. Reps and Subash Shankar and Tim Teitelbaum. Program Slicing of Hardware Description Languages. *Conference on Correct Hardware Design and Verification Methods*, pages 298–312, 1999.

[11] M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

[12] A. J. Martin, M. Nystrom, and P. I. Penzes. Et2: a metric for time and energy efficiency of computation. pages 293–315, 2002.

[13] J. Monteiro, J. Rinderknecht, S. Devadas, and A. Ghosh. Optimization of combinational and sequential logic circuits for low power using precomputation. In *Proceedings of the 16th Conference on Advanced Research in VLSI (ARVLSI'95)*, page 430. IEEE Computer Society, 1995.

[14] Mosis-TSMC 0.18 m CLO18 Process. *http://www.mosis.org/products/fab/vendors/tsmc/tsmc018/*. 2004.

[15] OPENCORES. *http://www.opencores.org*.

[16] A. Sakanaka, S. Fujii, and T. Sato. A leakage-energy-reduction technique for highly-associative caches in embedded systems. *SIGARCH Comput. Archit. News*, 32(3):50–54, 2004.

[17] The VIS Group. VIS: A system for Verification and Synthesis. In R. Alur and T. Henzinger, editors, *Proceedings of the 8th International Conference on Computer Aided Verification*, pages 428–432. Springer Lecture Notes in Computer Science #1102, July 1996.

[18] F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[19] V. Tiwari, S. Malik, and P. Ashar. Guarded evaluation: pushing power management to logic synthesis/design. In *Proceedings of the 1995 international symposium on Low power design*, pages 221–226. ACM Press, 1995.

[20] V. M. Vedula, J. A. Abraham, J. Bhadra, and R. Tupuri. A hierarchical test generation approach using program slicing techniques on hardware description languages. *J. Electron. Test.*, 19(2):149–160, 2003.

[21] C. Zhang, F. Vahid, J. Yang, and W. Najjar. A way-halting cache for low-energy high-performance systems. In *Proceedings of the 2004 international symposium on Low power electronics and design*, pages 126–131. ACM Press, 2004.