A First-step Towards an Architecture Tuning Methodology for Low Power

Greg Stitt, Frank Vahid*, Tony Givargis
Department of Computer Science and Engineering
University of California, Riverside
{gstitt, vahid,givargis}@cs.ucr.edu
www.cs.ucr.edu/~dalton

*Also with the Center for Embedded Computer Systems at UC Irvine.

Roman Lysecky
Conexant, Department of IP Management
roman.lysecky@conexant.com

ABSTRACT

We describe an automated environment to assist a system-on-a-chip designer to tune a microprocessor core to a particular application program that will run on the microprocessor, and vice-versa, with the goal of reducing embedded system power consumption. We limit such tuning to modifications that do not change the microprocessor instruction set, thus avoiding the large costs that would come with such a change. Our tuning environment for the 8051 microcontroller is freely-available on the web.

Keywords: system-on-a-chip, embedded systems, parameterized architectures, cores, low-power, tuning.

1. Introduction

Today's silicon chips can implement a system-on-chip (SOC), where a single chip may contain components like a microprocessor, memory, and perhaps tens of peripherals like DMA (direct memory access) controllers, UART's (universal asynchronous receiver/transmitters), encoders/decoders, analog-digital converters, and protocol interfaces. A designer may acquire these components in the form of an intellectual property core. A core may be a soft core, which is a behavioral description written in a hardware description language (HDL), where that description can be automatically synthesized down to a structural description. It may be a firm core, which is a structural description captured in an HDL. Or it may be a hard core, which is a technology-specific layout description. In any case, the various cores can be mixed together to form one large chip description, from which an actual silicon chip is fabricated.

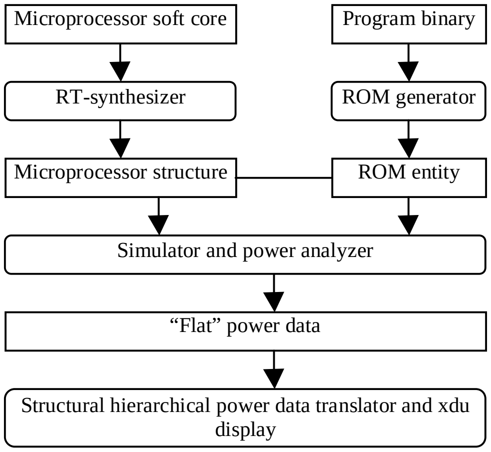
Designers are increasingly using SOC's to implement embedded systems, and embedded systems have a unique feature that provides a unique but untapped optimization opportunity. A unique feature of embedded systems that distinguishes them from desktop computer systems is that an embedded system typically executes a single application program for its lifetime. In contrast, a desktop system will have hundreds of different programs loaded onto it and executed. This single-program feature of embedded systems provides the opportunity for us to optimize the SOC components such that they execute that one particular program very efficiently. We refer to such optimization as *architecture tuning.* For example, a particular program may have an inner loop that accounts for most of its execution time, and that inner loop may include an operation that moves register R0's contents to register R1. On a regular instantiation of a microprocessor core, that operation might consume power as the data is read out of the register file, through the ALU (arithmetic-logic unit), through the shifter, and back into the register file. However, we could tune the microprocessor to this particular program, by creating a shortcut within the register file for that particular move from R0 to R1, thus reducing power. Furthermore, we could then tune the program to the microprocessor, by modifying the program to replace other moves by moves from R0 to R1 when possible.

We have begun to investigate techniques for tuning a microprocessor core to its particular embedded system application, with a focus on tuning for low power. Towards this end, we have developed an environment for automatically providing extensive power data to a designer, data that can be used to guide tuning decisions. We describe this environment in this paper, and highlight how it can be used for tuning purposes.

2. Problem description and previous work

We are given a microprocessor soft core, and a single application program that will run on that core. Our goal is to modify the soft core description such that the synthesized core will consume the minimum power possible while executing that program. A secondary goal is to modify the program into a functionally-equivalent program that consumes minimum power when executing on the microprocessor. The microprocessor modifications are restricted to those that *maintain exact instruction set compatibility,* meaning that the modified microprocessor executes exactly the same instruction set. No

Figure 1: Architectural view tool.



instructions may be omitted from the original instruction set. Furthermore, no instructions may be added either.

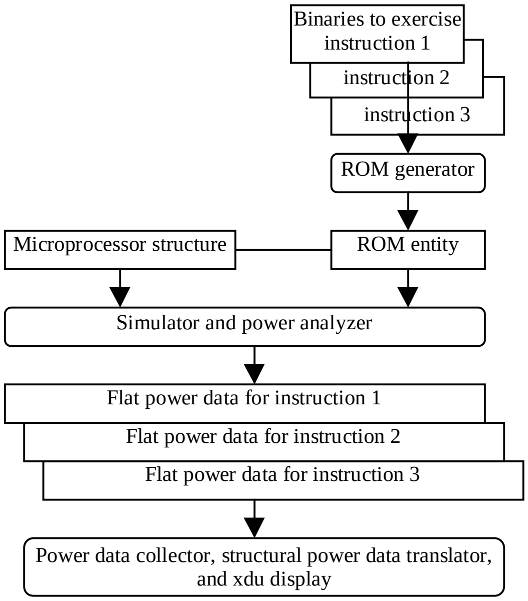
The restriction to exact instruction set compatibility is necessary since no modifications to compilers should be necessary, since any required changes to current embedded system design software would greatly limit the acceptance of the approach. Furthermore, the binaries that execute on the modified microprocessor must be portable to other versions of the microprocessor in the future, and thus should have no modifications made to them by special post-compilation passes that would result in non-standard instructions. Because we wish to make modifications to the structure of a microprocessor, our investigations focus mainly on soft cores, since new structure can be synthesized easily for soft cores.

Much previous work has focused on creating a microprocessor that is tuned to a particular program [1]. However, these approaches focus on actually choosing an instruction set that is best for a particular program or set of programs, thus resulting in an application-specific instruction-set processor, or ASIP. Our focus is instead on standard microprocessor cores, since in many cases standard cores provide tremendous advantages in terms of high-quality and low-cost tool support, designer familiarity, reduced design time, low-cost parts, and ease of future upgrades.

Some previous work focused on developing a compiler that would generate code optimized for a particular microprocessor, given power-per-instruction data of the microprocessor [7]. Some recent work [5] has focused on tuning a microprocessor and application by creating an architectural description language to describe architectural features (e.g., register file size, number of ALU’s, etc.), and creating a compiler able to read that language and hence optimize for the given features. This tuning approach could be used for standard processors as well as ASIP’s.

We have previously focused on tuning the parameterized components external to a microprocessor, including cache, bus, and peripheral cores, to a particular application, e.g., [4], and have shown that order of magnitude improvements in power and or performance are possible through such tuning. In this paper, though, we focus on the microprocessor internals.

Figure 2: Instruction-set view tool.


Binaries to exercise instruction 1
instruction 2
instruction 3
ROM generator
Microprocessor structure
ROM entity
Simulator and power analyzer
Flat power data for instruction 1
Flat power data for instruction 2
Flat power data for instruction 3
Power data collector, structural power data translator, and xdu display

3. Environment

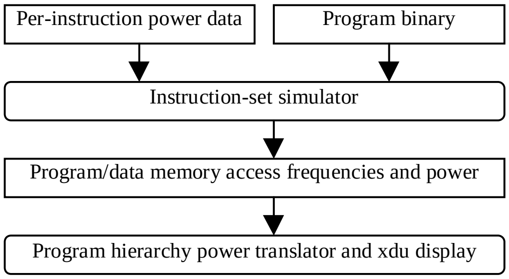
To enable a designer to tune an architecture and program, we have found three tools to be particularly useful. Each provides a unique view into the power consumption of architecture/program pair. The first is the architecture view, which gives a hierarchical summary of which microprocessor components are consuming power as the program executes. The second is the instruction-set view, which for a given architecture tells us the power-per-instruction for every instruction in the instruction-set. The third is the program/data memory view, which tells us the power per memory location being consumed by the executing program. We describe each in more detail in the following sections.

We have implemented all three tools for the 8051 microcontroller. The 8051 is one of the most popular 8-bit microcontrollers, and is used extensively in soft-core form as part of larger embedded systems. We have developed our own complete synthesizable soft core of the 8051 [8]. However, all the tools could be implemented for any other microprocessor for which a soft core is available. In the following sections, we’ll use the term “binary” to refer to a program file executable by the 8051 – such a file is actually known as a hex file in the context of 8051’s, and is in the form of a text file.

3.1 Architectural view

The architectural view tool provides a hierarchical view of where power is consumed in the microprocessor architecture when it executes the application program. The tool’s components are shown in Figure 1. The *ROM generator* translates a program binary into an HDL ROM entity, and synthesizes this entity to structure. The RT-synthesizer converts the microprocessor soft core HDL description into structure. The microprocessor and ROM entities are connected. The *Simulator and power analyzer* simulates the microprocessor executing the program in ROM. During this time, the simulation records

Figure 3: Program/data memory view tool.


Per-instruction power data
Program binary
Instruction-set simulator
Program/data memory access frequencies and power
Program hierarchy power translator and xdu display

switching activity and determines the power for each net in the synthesized system. The *Structural hierarchical power data translator and xdu display* converts this "flat" power-per-net data into a hierarchical set of power data, where the hierarchy corresponds to the hierarchy of entities in the synthesized structure. The hierarchical output is formatted with the same format of the Unix utility "du". This allows the power results to be used as input to the program "xdu", which will display an interactive tree representation of the power usage. All the synthesis, simulation and power tools are Synopsys tools [6]. The remaining functionality is achieved using scripts.

3.2 Instruction-set view

The instruction-set view tool provides a view of the power consumed by each instruction in the microprocessor's instruction-set, independent of the actual application program. The tool's components are shown in Figure 2. It contains a binary test program for each instruction. Each binary program consists of hundreds of occurrences of the corresponding instruction, using random data and addresses if the instruction has such fields. Those occurrences are contained in a loop that executes tens of times. For each binary, the tool generates a ROM entity, connects it with the existing microprocessor structure (which, if it doesn't exist yet, must be synthesized from the soft core), and runs simulation and power analysis. The resulting power data is collected per instruction, with the power for the loop subtracted from this data, and a power-per-instruction table is created. Another table is created hierarchically according to the classes of instructions (arithmetic, moves, branches, etc.), so that xdu can again be used to explore the data hierarchically.

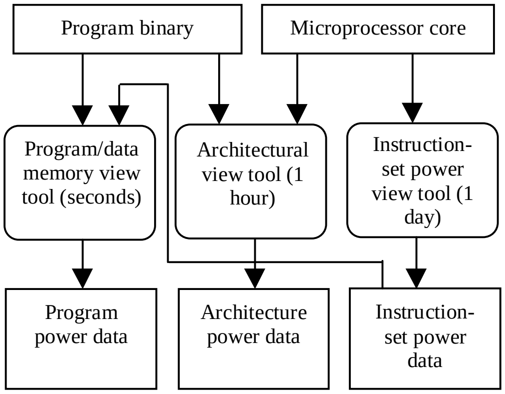
Furthermore, the tool executes the structural hierarchy power data translator, as was done in the architectural view tool, to provide data on how each individual instruction consumes data in the microprocessor.

All functionality outside the Synopsys synthesis, simulation and power analysis tools is again achieved using scripts.

3.3 Program/data memory view

The program/data memory view tool provides the power consumed by each instruction in the application program itself (i.e., by each address in program memory corresponding to the beginning of an instruction), and by each address in data memory, as the program executes on the given microprocessor. The tool's components are shown in Figure 3. The heart of the tool is an *instruction-set simulator*, which reads and executes the program binary. In addition to creating the simulator for the 8051, we have extended the simulator to lookup the per-instruction power data, obtained by the instruction-set view tool, as each instruction is simulated. Upon completion, the instruction-set simulator outputs a file describing the frequency of access of every memory location in program and data memory, and the associated power of each (access frequency times power-per-access). The *program hiererarchy power translator* converts this data into hierarchical data, according to the program block hierarchy, for examination using xdu. (This translator currently only provides a flat view). The data memory power data can also be grouped hierarchically according to various regions of the memory space. For example, the 8051 has a region holding four register sets, a bit-addressable region, regions corresponding to external ports, etc.

Figure 4: Tuning environment.


Program binary
Microprocessor core
Program/data memory view tool (seconds)
Architectural view tool (1 hour)
Instruction-set power view tool (1 day)
Program power data
Architecture power data
Instruction-set power data

3.4 Overall tuning environment

These three tools can automatically generate new power views whenever we modify the soft core or program binaries, to assist a designer detect candidate changes for tuning, and to examine the effectiveness of such changes. The overall environment is shown in Figure 4.

The instruction-set power view tool must be run whenever there is a change to the microprocessor architecture. Because this tool gathers data for every instruction in the instruction-set, it takes about one day to run in our implementation.

The architectural view tool must be run whenever there is a change in the program binary or the microprocessor core. It requires about an hour in our implementation.

The program/memory data view tool must be run whenever there is a change in the program binary or microprocessor core (since it relies on the per-instruction power data generated by the instruction-set power tool). This tool only requires seconds or minutes to execute.

The design flow is shown in Figure 5. In response to the provided power views, the designer can modify the architecture, the binaries, or both. Changing the architecture requires about a day for new power views, since all three tools must be re-run. Changing the binaries requires only about an hour, since the instruction-set view tool need not be re-run.

4. Experimental results

The following sections provide results from an experiment of using the tuning environment on a particular fabricated testbench application. The testbench consists mainly of several arithmetic instructions followed by a square root calculation. When synthesized, it produces a ROM entity consisting of 1162 bytes.

4.1 Architectural view

The first analysis tool that we ran was the architectural tool, which generates a ROM from the given hex file, synthesizes and simulates for a given period of time, and then outputs the results in a format readable by xdu. For our model, ROM is separated from all other components. This allows these two groups to be synthesized separately, so that when multiple programs are tested, only the ROM needs to be synthesized multiple times. The synthesis of the ROM takes between a minute and an hour depending on the size of the application. Synthesizing the rest of the microprocessor takes approximately an hour. The results are shown in Figure 6. These results show that most of the power is consumed by the RAM, controller, and ALU. In this case, the controller would be a good choice for first examination. Although this figure shows a very simple component configuration, any possible tree configuration can be handled.

Figure 5: Design flow when using the tuning environment.

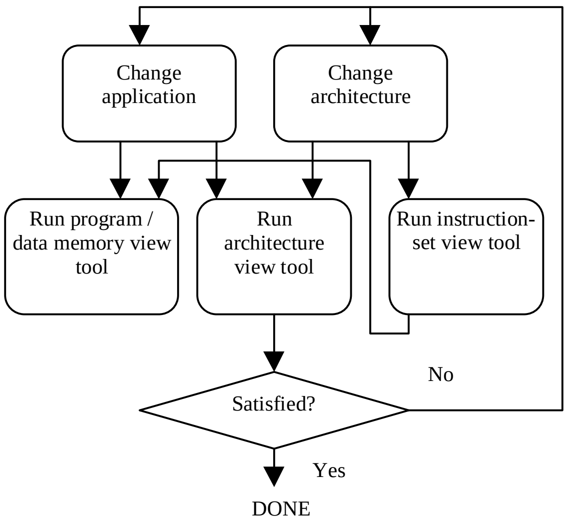
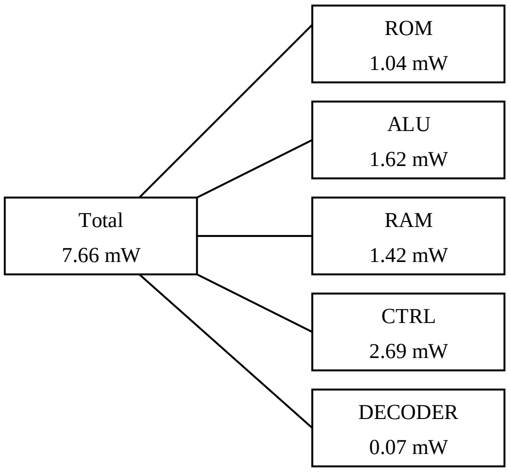

Change application
Change architecture
Run program / data memory view tool
Run architecture view tool
Run instruction-set view tool
No
Satisfied?
Yes
DONE

Figure 6: Hierarchical architectural view of power for the testbench application.



4.2 Instruction-set view

After creating the architectural view of power usage, we ran the next tool in order to determine the average power per instruction for our 8051 model. For each instruction, we ran the ROM generator and then synthesized, simulated, and computed average power. Not all the jump instructions are being tested at

Table 1: Average power per instruction

Instruction	**Power** (mW)
ADDC_1	7.340834
ADD_1	7.350741
ANL_1	6.631394
CLR_1	3.76228
CPL_1	5.481627
DA	5.28897
DEC_1	5.368807
DIV	7.716592
INC_1	4.662862
MOVC_1	6.078014
MOVC_2	5.021021
MOV_1	5.577664
MOV_2	6.164267
MUL	5.522886
NOP	4.900275
ORL_1	6.954121
POP	8.103867
PUSH	8.7116
RL	4.302023
RLC	5.521254
RR	4.23862
RRC	5.307967
SETB_1	3.810065
SJMP	6.392
SUBB_1	7.740368
SWAP	3.671953
XCHD	9.479995
XCH_1	5.635007
XRL_1	6.792233

this time because they require extra considerations. The instruction-set view tool is somewhat time consuming. It takes approximately 15 minutes for ROM synthesis and system simulation of each instruction. There are 111 instructions in the 8051 instruction set, thus leading to a running time of about a day. A small sample of the output can be seen in Table 1.

In this example, the difference in power between instructions ranges from approximately 3.8 mW to 9.5 mW. If the higher power instructions are used frequently, it may be beneficial to optimize those instructions in the architecture, or to change the application to use other, lower power instructions if possible.

4.3 Program/data memory view

The power per instruction results become very useful when used with the next step of analysis. At this point, the instruction set simulator is run with the testbench application and statistics are logged regarding both the program and data memory. A small sample of the output for program memory statistics can be seen in Table 2. This table shows data for two blocks of program memory (as mentioned earlier, the LJMP and RET instruction power has not yet been implemented). The first block (addresses 0 through 11) is executed quite frequently, whereas the second block (12 through 22) is only executed approximately one fourth as often. This view is obviously very effective in determining what code blocks and areas of program memory are consuming the most power. At this point, either the application could be changed to use less power, or the microprocessor could be changed to more efficiently implement one type of instruction (in this case MOV_9).

The statistics logged for RAM are also useful. Table 3 shows the areas of RAM that are accessed the most. The most frequently accessed locations in this example are register bank 0 (Addr 0-7), location 208, and location 224. If accesses to these regions of memory are optimized, it is obvious that a great deal of power can be saved.

Unlike the other tools, the simulator doesn’t need to track actual signal activity. Instead it uses the earlier-generated power-per-instruction lookup table – similar in idea to [7]. Therefore, the simulator is much faster than both the architectural and instructional power tools. Simulation of 1000 instructions with the previous tools takes approximately 15 minutes. With the instruction set simulator, almost 50,000 instructions can be simulated every second on a 550 MHz Intel Pentium III machine.

4.4 Sample tuning optimizations

The data in the previous sections can be used to implement several obvious optimizations. First, we note that RAM consumes much power. Second, we note that most accesses to RAM are to locations 208 and 224. Thus, if we pull those locations out of the RAM and instead use internal registers, we may be able to reduce power. To test this hypothesis, we modified the core model by creating a new register inside the CTRL module, and replaced accesses to location 224 by accesses to that new register. We then re-ran the three view tools, requiring about a day. The results showed that this one change decreased overall power from 7.67 mW to 7.27 mW – a 5% decrease from just this one change. We observed that RAM power was decreased from 1.42 mW to 0.8 mW, accompanied by a smaller increase in CTRL power.

Table 2: Program memory view

Addr	Ins	Freq	Pwr	Freq*Pwr
00000	LJMP	1	0	0
00003	MOV_9	108	5.46067	589.752
00005	MOV_9	108	5.46067	589.752
00007	MOV_9	108	5.46067	589.752
00009	MOV_9	108	5.46067	589.752
00011	RET	108	0	0
00012	MOV_9	27	5.46067	147.438
00014	MOV_9	27	5.46067	147.438
00016	MOV_9	27	5.46067	147.438
00018	MOV_9	27	5.46067	147.438
00020	MOV_4	27	4.83507	130.547
00022	LCALL	27	0	0

Table 3: Data memory view

Addr	Purpose	Accesses
00000	RegBank0	38765
00001		16127
00002		7562
00003		25012
00004		25737
00005		35160
00006		32750
00007		35143
00008	RegBank1	136
00009		136
00010		136
00011		136
00012		82
00013		82
00014		82
00015		109
00128	P0	1311
00129	SP	70317
00130	DPL	31189
00131	DPH	7977
00144	P1	161
00208	PSW	413527
00224	ACC	360949
00240	B	2598

Many more tuning modifications are of course possible. For example, we could partition the controller into two finite state machines. One of these can handle the majority of instructions and the other can be designed to efficiently handle instructions that are most frequently executed and hence consume most power. A good controller partitioning has been shown to reduce controller power by over 50% [3][2] in many cases, and can be implemented in a short amount of time using straightforward techniques for rewriting the controller HDL behavioral description.

We plan to use this environment to perform extensive tuning optimizations on a number of applications, to determine the range of power savings possible through such tuning.

5. Conclusion

We have developed a tool to assist a designer to tune a microprocessor with its application. This is useful in design methodologies that use a microprocessor soft core, when that core will execute a single application for the core's lifetime. An important feature of our approach is that it does not modify the microprocessor instruction set. Our environment provides enough views to enable a designer to decide whether to focus on modifying the architecture, application program, or both, as well as to determine which part of each to focus on. The environment is automated and mostly uses standard tools along with scripts. The tuning environment for the 8051 is available, along with a synthesizable 8051 soft core and an 8051 instruction-set simulator, at the UCR Dalton project web page at http://www.cs.ucr.edu/~dalton. The UCR Dalton project focuses on issues related to IP-based system-on-a-chip design. Future work may include speeding up iterations in the environment, developing tuning strategies, and perhaps automating certain tuning transformations.

6. Acknowledgements

This work was supported by the National Science Foundation under grants CCR-9811164 and CCR-9876006.

References

[1] J. Fisher. Customized Instruction Sets for Embedded Processors. Design Automation Conference, pp. 253-258, 1999.

[2] L. Benini, P. Vuillod, G. De Micheli and C. Coelho, Synthesis of Low-Power Selectively-Clocked Systems from High-Level Specifications. International Symposium on System Synthesis, pp. 57-63, Nov. 1996.

[3] E. Hwang and F. Vahid and Y.C. Hsu. FSMD Functional Partitioning for Low Power. Design Automation and Test in Europe (DATE) Conference, pp. 22-28, March 1999.

[4] T. Givargis, J. Henkel and F. Vahid. Interface and Cache Power Exploration for Core-Based Embedded Systems. International Conference on Computer-Aided Design (ICCAD), pp. 270-273, November 1999.

[5] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, A. Nicolau. EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability. Design Automation and Test in Europe (DATE) Conference, pp. 485-490, March 1999.

[6] Synopsys, http://www.synopsys.com.

[7] V. Tiwari, S. Malik, A. Wolfe. Power Analysis of Embedded Software: A First Step Toward Sofware Power Minimization. IEEE Transactions on VLSI Systems, vol. 2, no. 4, pp. 437-445, 1994.

[8] The UCR Dalton Project, http://www.cs.ucr.edu/~dalton.