

SimpleDSP: A Fast and Flexible DSP Processor Model

(EXTENDED ABSTRACT)

Jeff Ringenberg, David Oehmke
Todd Austin, Trevor Mudge
{jringenb, doehmke, taustin, tnm}@eecs.umich.edu
The University of Michigan
Advanced Computer Architecture Lab

1.0 Introduction

When designing a future mobile microprocessor, the standard model of high performance at all costs does not apply. Power usage and chip costs are two major design points that must be minimized in order for the processor to be a viable product. Unfortunately, many current high performance designs satisfy neither of these requirements and, therefore, a separate class of machines has emerged that incorporates DSP functionality. As a tool to explore this design space, we have developed a simulator for a popular DSP, the Texas Instruments TMS320C6211 (C62x) [1], and incorporated it into the widely used SimpleScalar toolset [2]. With this simulator, we have run detailed, cycle-accurate simulations of the underlying architecture and have found several bottlenecks within the design. In addition, we have discovered the importance of appropriate code design and selection through the use of intrinsic instructions. However, much deeper studies remain to be performed since the purpose of this paper is to show the functionality and usage of the simulator.

2.0 Motivation

The importance of DSP systems doesn't need justification. However, until recently there has not been widely disseminated support for simulating such architectures in the academic community. The need for this type of simulator is apparent in the abundant usage of SimpleScalar, and many new papers in a wide range of microarchitecture research make use of this popular simulation environment. However, since it models the architecture seen in a superscalar processor, it is not useful for evaluating architectures that do not adhere to this type. This is especially true for DSP and VLIW processors. Therefore, in order to address this functional lacking, we have decided to modify the existing SimpleScalar structure to facilitate the simulation of both types of architecture with a focus on flexibility and accuracy.

3.0 Related Work

Due to the fact that DSP and VLIW processors are not currently the mainstream designs for modern high performance processors, there are not many simulators available that are free to use for research. Texas Instruments and other third party vendors offer several development environments for the C62x architecture, the most popular of which is the Code Composer Studio from TI [3]. However, these tools do not expose enough of the underlying architecture to allow for the detailed research of new design ideas.

An example of one tool, WETICS [4], created to simulate a DSP has been developed at the University of Texas. The tool is a web-based JAVA simulator that provides support for the TI-C30 and several Motorola MCx DSPs. However, the project is no longer under development. Unfortunately, this same status holds true for several other simulators of DSP, and VLIW, architectures. Our hope is that by building our model into SimpleScalar, we will garner much wider exposure and support than previous attempts.

4.0 Implementation

SimpleScalar was designed to simulate processors in which each instruction semantically had a latency of only one cycle and instructions were executed serially. However, a VLIW processor, which the TI-C62x is, includes both parallel execution and non-uniform latencies. For these processors, the compiler is responsible for statically scheduling the code using these latencies. Preserving the semantics of the generated code requires that correct timing be used for each instruction and that instructions be executed in parallel making the functional simulation of a VLIW processor much more difficult than that of a scalar processor.

The TI-C62x itself has some additional features, many geared towards DSP functionality, that further complicate the simulation. The pipeline is more complex because the processor has several techniques to improve code density including NOPs with multi-cycle latencies and the decoupling of fetch packets and execution packets. Instruction decode is also difficult because most instructions can be executed on either of the DSP's two clusters and on several different functional units, and most instructions also allow one or more sources to be a register, a constant, or a register from the other cluster. A final complication is that the TI compiler targets their test board and all the I/O system calls are implemented by the connected PC using a breakpoint and global buffer for data transfer.

The complications inherent in this architecture meant that it was not practical to merge this into the existing SimpleScalar simulators. Instead, we decided to create a new simulator based on the ideas in SimpleScalar and to use as much of the existing code as possible. This would allow us to take advantage of future improvements, allow people familiar with SimpleScalar to quickly learn our simulator, and to provide for possible future interoperability in areas like heterogeneous multiprocessing. We also decided to explicitly simulate each stage of the TI pipeline to guarantee accurate timing of all the instructions and to make the resulting simulator cycle accurate.

Similar to SimpleScalar, we use "def" files to decode and implement instructions. However, our simulator uses two separate def files. One def file is similar to SimpleScalar and decodes the instructions. The decode is done as one pass on the entire text section of the executable and an operation structure is filled out for each instruction. The other def file, called the operation def file, contains the timing information of the operation as well as its implementation. The basic execution of an operation is similar to that done in SimpleScalar with one notable exception. The reading and writing of registers is removed from the instruction specific implementation and done in a generic fashion using the information stored in the operation structure with the values stored into, or read from, the structure. This was done both to simplify instruction implementations and

also because some of these operations must occur in parallel. As an example, during each cycle all data reading must be complete before any writing can be done. Finally, the pipeline code was taken out and put into a separate pipeline file so that the bulk of the code could be shared across all the different versions of our simulator. The pipeline code provides macros that the various versions can use to hook into the pipeline. These were used to provide additional stats in sim-vliw-profile, to hook into the cache model in sim-vliw-cheetah and sim-vliw-cache, and to verify the simulation against a trace file in sim-vliw-verify.

5.0 Experiments

As mentioned previously, our simulator has allowed us to do detailed, cycle-accurate simulations of the TI-C62x architecture and subsequently find what features make it work well and what do not. Since the purpose of this paper is to show the functionality and features of the simulator, and not performance results, demonstrative experiments were run for a few DSP-like benchmarks, namely the GSM coder/decoder [5] and several components of a SmartCamera system [6].

For our first experiment, we found the number, and then analyzed the removal, of cycles that consist completely of NOP instructions. Due to the statically scheduled, VLIW nature of the C62x, NOP instructions are inserted after branches and any other multi-cycle latency instructions to ensure correct operation. Simply removing the NOP cycles from the code is not as straightforward as one would think, since the code would not function properly without them. Therefore, the results function as a best case scenario if the compiler did not need to insert these scheduling delays. As Figure 1 shows, there are many cycles consisting only of NOPs that could contribute to wasted execution and if they could be removed, execution time would greatly decrease.

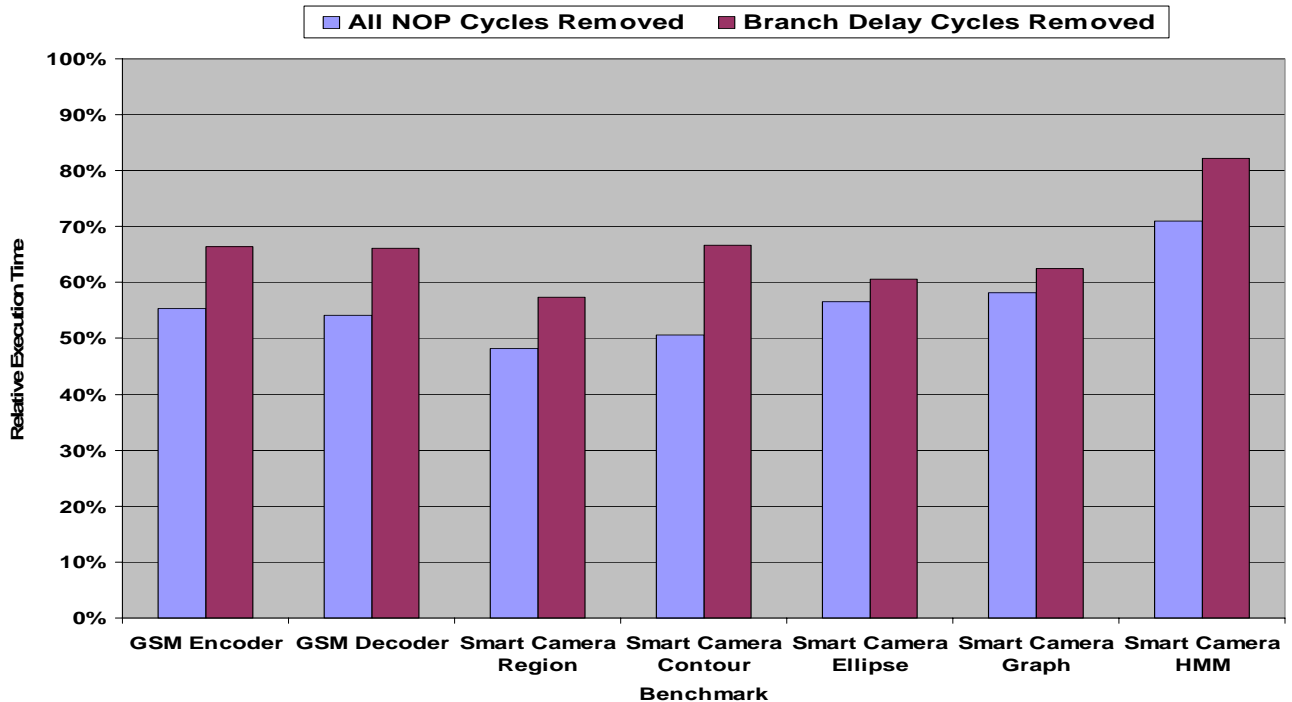


Figure 1: NOP cycles and the Effects of their Removal

For our second experiment, we looked at the effects on execution time of including intrinsic instructions in the GSM code. These intrinsic instructions, a saturated add is a good example, are tailor made for the C62x and are used as much as possible. Since TI provided us with the header files [7] containing the insertion of the intrinsic instructions into the GSM code, we were unable to get these results for the SmartCamera applications. It should be noted that these hand coded header files are required for appropriate intrinsic selection because the TI compiler is not efficient at discovering this information. Our third experiment explores the varying abilities of the compiler to create efficient code.

As Figure 2 shows, the insertion of the intrinsic instructions has a dramatic effect on the execution time of the GSM code. This is a perfect example of the importance of these types of instructions on the efficient generation and execution of code on a DSP or any other such architecture that includes these types of instructions. It also demonstrates the need for having a good compiler that can decide where these instructions should be placed or at least having hand coded header files that fulfill the same function.

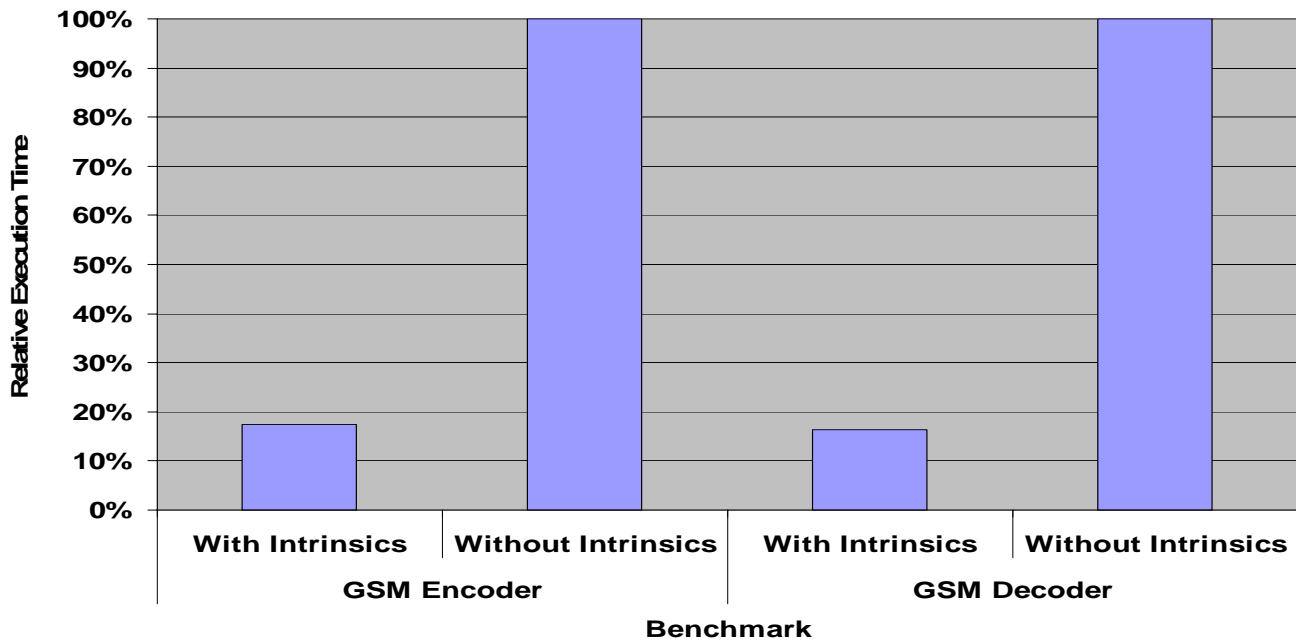


Figure 2: The Effects of the Addition of Intrinsic Instructions

For our third experiment, we decided to actually look at the TI compiler’s ability to create efficient code by exploring the effects of various compiler optimizations on execution time. Since VLIW processors are highly dependent on their compiler for optimal code generation and scheduling, it makes sense to explore which optimizations are the most beneficial.

In Figure 3, all relative execution times are normalized to defaults with `-O3` optimization. The first two optimizations turn off debug information and the graph shows that for some benchmarks quite a bit of performance is lost when this debug information left in. Forcing the stat counting function inline on the GSM code has quite a performance benefit as well. C62x function calls have a fair amount of overhead, so inlining

works well when a very small function is called often. For the next optimization, assuming no aliasing allows the compiler to be aggressive in register allocation and in instruction reordering and this works in a couple of the benchmarks. Next, using a large inlining threshold benefits those benchmarks that contain a lot of function calls. Finally, whole program analysis provides the compiler with more information to use when compiling for instance propagating constants through function calls. In several benchmarks, this allows the compiler to more efficiently software pipeline some of the major loops. As the graph shows, it is not always as simple as turning on all optimizations, since for a couple of the benchmarks turning on whole program analysis actually degrades the performance.

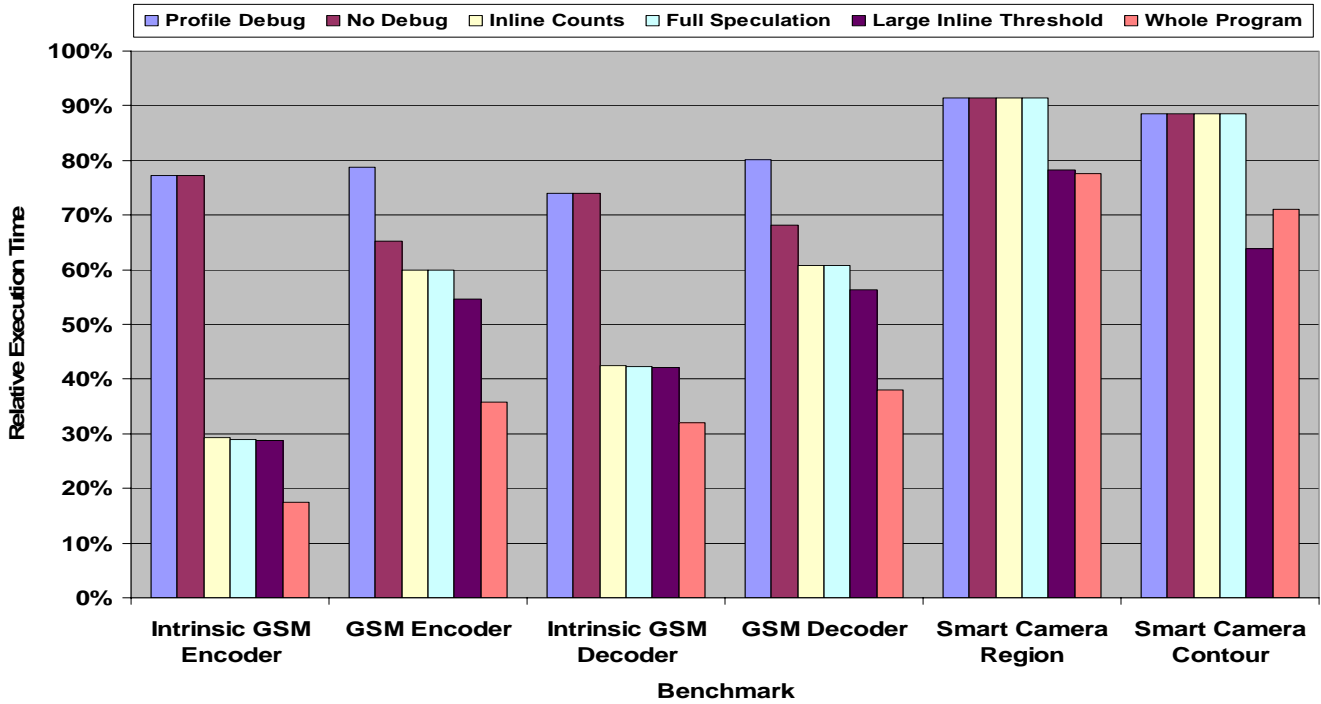


Figure 3: The Effects of Compiler Optimizations on Execution Time

In addition to these experiments, we have also run, and compiled statistics for, many other experiments that measure function unit usage, crosspath and cluster utilization, instruction class breakdowns, addressing mode types, branch and predicated instruction breakdowns, and many other statistics that measure the efficiency of the C62x.

6.0 Conclusion

In this paper, we have presented a simulator that can be used to do detailed analysis of a popular DSP, the TI TMS320C6211. In addition to being able to simulate this particular architecture, the simulator is general enough to allow the simulation of other VLIW machines with only minor changes to the infrastructure. With this simulator, it is now possible to explore both new architecture ideas and compiler ideas in a flexible and accurate manner. It is our opinion that this tool will make the design of future mobile devices easier and hopefully usher in a new era of design and simulation.

References

- [1] Texas Instruments. *TMS320C6000 CPU and Instruction Set Reference Guide*. SPRU189F. October 2000. Available from <http://focus.ti.com/lit/ug/spru189f/spru189f.pdf>.
- [2] D.C. Burger and T.M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin-Madison, June 1997.
- [3] Texas Instruments. Code Composer Studio IDE Version 2.2. August 2003. Available from <http://www.ti.com/tmwccs>.
- [4] D. Arifler and B. L. Evans. Web-Enabled Simulation and Debugging for Digital Signal Processors and Microcontrollers. Available from <http://anchovy.ece.utexas.edu/~arifler/wetics/>.
- [5] GSM 06.51 Encoder/Decoder Digital Cellular Telecommunications System (Phase 2+), Enhanced Full Rate Speech Processing Functions Version 8.0.1. Available from <http://www.etsi.org>.
- [6] T. Lv, B. Ozer, and W. Wolf. "Workload Characterization for Smart Cameras". *3rd Workshop on Media and Streaming Processors* (held in conjunction with the 34th International Symposium on Microarchitecture). December 2001.
- [7] Texas Instruments. *ETSI Math Operations in C for the TMS320C62x*. SPRA617A. November 2000. Available from <http://focus.ti.com/lit/an/spra617a/spra617a.pdf>.