X32V: A Design of a Configurable Processor Core for Embedded Systems

David Zier[1], Jumnit Hong[1], Savithri Venkatachalapathy[1], Jarrod Nelson[1], John Mark Matson[2], Ben Lee[1], Younghwan Bae[3], and Hanjin Cho[3]

[1]*School of Electrical Engineering and Computer Science*
Oregon State University
Corvallis, Oregon
{zier, hongju, venkasav, nelsonja, benl}@ece.orst.edu

[2]*Intel Corporation*
Hillsboro, Oregon
john.m.matson@intel.com

[3]*Electronics and Telecommunications Research Institute (ETRI)*
Daejeon, Korea
{yhbae, hjcho}@etri.re.kr

Abstract

This paper introduces the X32V configurable processor core. X32V is geared towards low-power, low-memory embedded systems, such as cell phones, PDAs, and digital cameras. X32V uses a feature that allows for variable length instructions that ultimately decrease the amount of program memory required for applications. In addition, X32V supports additional modules that increase flexibility. Currently, a multimedia extension module and a floating-point module have been developed and integrated into the X32V schema. An X32V prototype was developed as an execution based, cycle-accurate simulator. Preliminary testing of X32V has been performed using benchmarks based on portions of MPEG-4 decoding.

1. Introduction

Current trends in embedded applications are requiring more features and a shorter time-to-market. With the advent of System On Chip (SOC), the need to design a custom processor and secondary components has created enormous challenges for embedded system designers [1]. A synthesizable, configurable processor offers an effective way to improve time-to-market. A configurable processor allows the designer to create a custom microprocessor by configuring the processor core or adding specialized modules [2, 3]. These modules can provide the benefits of a coprocessor without the communication or area overhead usually encountered.

Currently, there are several companies offering configurable and extensible processor cores including Tensilica [2] and ARC cores [3]. The *eXpandable 32-bit Variable*, or X32V, processor core offers three modes with various instruction lengths, which allows added flexibility when balancing memory requirements and performance. At the core, X32V only supports an integer instruction set, which is useful in most situations, but X32V also provides extensive support for additional expansions through the use of add-on modules.

To prototype X32V and test the performance of various configurations, a cycle-accurate, execution-based simulator was developed. This simulator used the SimpleScalar Toolset [4] as the base framework for simulating memory, system calls, cache, resources, and statistics gathering.

One of the goals for the X32V processor core was to gear it towards small multimedia applications, such as those found in cell phones, digital cameras, and other handheld devices. In order for the microprocessor to be viable in a multimedia application, it would need to be able to very quickly decode and encode JPEG and MPEG type media. As a result, one of the first modules created for X32V was a multimedia extensions module, called *EM3*. Several multimedia benchmarks were run on the simulator to test the validity of X32V and EM3.

The paper is organized as follow: Section 2 describes the various modes and instruction formats that are supported by the X32V processor core. Section 3 discusses the configurable components of X32V. The compiler is described in Section 4. EM3 is described in Section 5 along with some performance results from using EM3. Finally, Section 6 concludes the paper.

2. X32V Instruction Modes

X32V supports three different modes of variable length instructions.

- Default (32-bit instructions)
- Light (32/16-bit instructions)
- Ultra-Light (32/24/16-bit instructions)

In default mode, a 32-bit instruction word is fetched from memory every clock cycle. In this mode, all the instructions are fetched on word aligned boundaries. The default mode provides large immediate values and room for expansion. Figure 1 illustrates the various instruction formats supported in Default mode.

Light mode can fetch both 32-bit and 16-bit instructions. The compiler will map any 32-bit instructions into their equivalent 16-bit formats if the instruction is compressible. Since all instructions are fetched 32-bit at a time, 32-bit instructions can span across a memory word. This incurs a 1-cycle penalty because the next 32-bit word needs to be fetched to get the complete instruction. Figure 2 illustrates the 16-bit instruction formats used by the Light mode.

Ultra-light mode allows instructions to be fetched as 32-bit, 24-bit, or 16-bit. The compiler will map any instructions that are compressible into 24-bit or 16-bit instructions. There can be up to a four-cycle penalty on a branch or jump that is made to a non-word aligned instruction in memory since there can potentially be three partial instructions in the fetched word. The benefit of the Ultra-light mode is that it provides the smallest program binaries and thus saves memory space. Figure 3 illustrates the 24-bit instruction format that is used in the Ultra-light mode.

To support multiple instruction lengths, the fetch unit has a buffer to hold any partial instruction fetched from memory, which is determined by the first byte in the opcode. This portion will be part of the next instruction fetched. On the next fetch cycle, the remaining part of the instruction will be combined with the portion of the instruction in the buffer to form the complete instruction.

A case study was done to determine the effectiveness of having default, light, and ultra-light modes. SimpleScalar [4] was used to conduct the study that required a one-to-one mapping of X32V instructions to SimpleScalar instructions. This mapping allowed an accurate estimation of code size from a compiled SimpleScalar binary. The benchmarks used in the study were from the MediaBench suite [5] and an MPEG-4 encoder/decoder provided by ETRI. Each benchmark is described below:

- ADPCM: Adaptive differential pulse code modulation. A simple form of audio encoding.
- EPIC: An experimental image compression utility based on a bi-orthogonal critically sampled dyadic wavelet decomposition and a combined run-length/Huffman entropy coder.

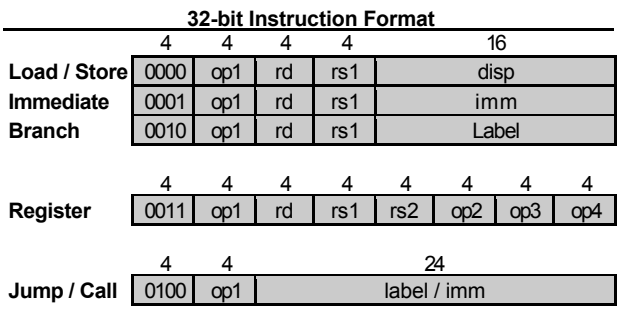

Fig 1. Instruction formats for Default mode.

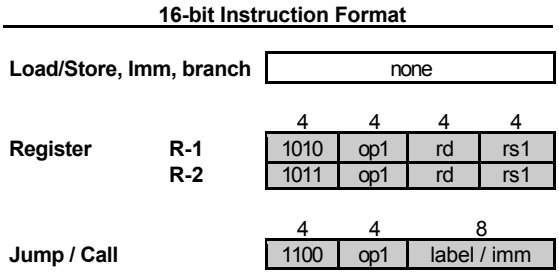

Fig 2. 16-bit format for Light mode.

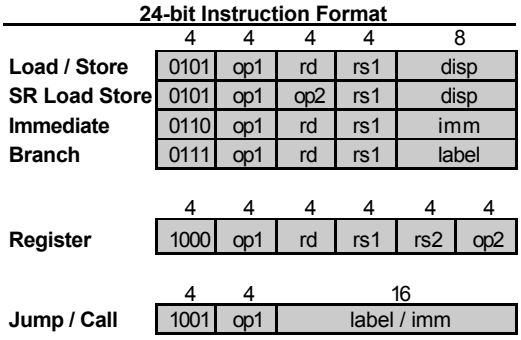

Fig 3. 24-bit format for Ultra-Light mode.

- G. 721 Enc/Dec: Voice compression standard.
- JPEG Enc: Standardized compression method for full color and gray-scale images.
- MPEG-2 Enc: Standard high quality movie compression.
- PEGWIT: A program used in public key encryption and authentication. It uses elliptic curve over GF(2^{255}), SHA1 for hashing, and a symmetric block cipher.

PERL scripts were created to analyze the output from the SimpleScalar compiled binaries. The first script analyzed the binaries for branches and jumps, determining penalties when branching to non-aligned word boundaries. The second script mapped SimpleScalar instructions onto X32V instructions in default, light, and ultra-light modes to show the compressed binary sizes.

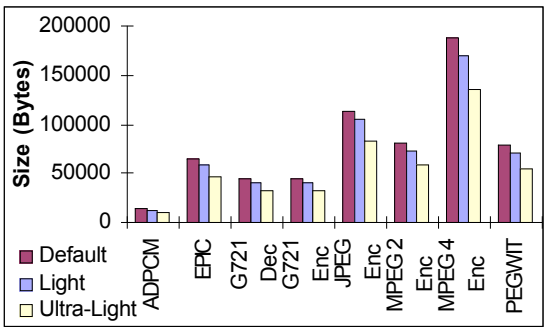

Fig 4. Binary size comparison of different modes.

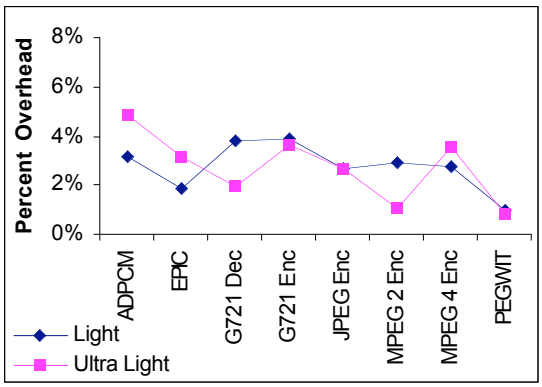

Fig 5. Overhead comparison of different modes.

These results are shown in Figures 4 and 5. X32V resulted in an 8% decrease in code size when compressing instructions into light mode and a 27% decrease in code size in ultra-light (see Figure 4). There is about a 3% cycle overhead in both light and ultra-light modes (see figure 5). The cycle overhead consisted of additional cycles introduced when reading misaligned memory instructions. These results show that the Ultra-light mode demonstrates the best combination of code compression with minimal performance loss.

3. Configurable Component Modules

X32V supports a configurable core that can be changed during the design phase by the inclusion of various modules. Currently, the floating-point and multimedia modules have been developed as cycle-accurate simulator components and can be added or removed depending upon the processor's intended purpose. The base simulator consists of several complex components that are needed to support component modules. Some examples are a complex forwarding mechanism, support for multicycle functional units, a fetch unit capable of fetching misaligned program memory, and a memory unit that is capable of reading data of any size and alignment from data memory. Additionally, a complex interface was created to include configurable modules.

3.1. Modular Support

A module is additional hardware that can interpret a group of new instructions. Depending on the module, this hardware consists of an extension to the decoder, functional units that will execute the instruction, and possibly an additional register file that can hold the module specific data. To simulate this hardware, additional source and header files containing cycle-accurate simulator components that prototyped these hardware modules were created. They were integrated into the simulator through the use of several interfaces shared by each module.

The basic underlying architecture of X32V is a 5-stage pipeline with several common data buses that are used for interfacing between the stages, write back, and forwarding. One of the key features for modularity is the support of multi-cycle functional units [6]. Since each functional unit shares the common data buses, it is very easy to include additional functional units.

In order to keep all of the functional units working seamlessly together, X32V maintains in-order execution through the use of a reservation shift register (RSR) [6]. The RSR simplifies scheduling by tracking the cycle on which any instructions will finish the execution stage.

3.2. Current Modules

Currently, there are only two modules that have been implemented and tested for performance: The floating-point modules that support both 32-bit and 64-bit floating-point numbers and the multimedia module (EM3) that performs SIMD type integer operations. Both of these modules share a common register file, which means that floating-point registers are aliased to the EM3 registers. Figure 6 illustrates the integration of both the floating-point and EM3 modules within the X32V pipeline. This integration is exactly how X32V is currently simulated. EM3 is further discussed in Section 5.

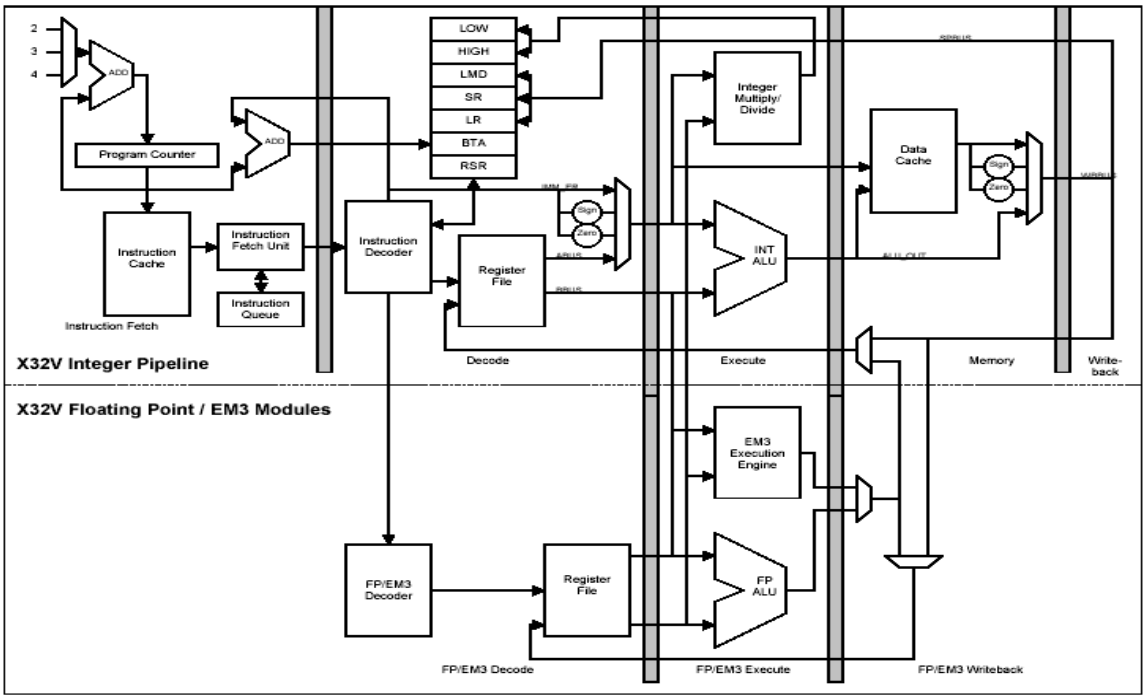

Fig 6. X32V architectural diagram with floating-point and EM3 modules.

4. Compiler Support

In order to facilitate the simulation and profiling of X32V instructions, a compiler was developed. The design principles of modularity and configurability influenced our approach while porting the egcs 1.1.2 compiler tool set for X32V [7]. At the time of this writing, the 32-bit Default mode is supported along with the provisions to support the 16- and 24-bit instructions for the Light and Ultra-light modes as well. Macro modules in the *fp-bit* floating-point library handle the floating-point operations while the *newlib* library provides the run-time support.

The porting process involved identifying an architecture similar to X32V and making the relevant modifications to the machine descriptions and calling conventions [8]. The Application Binary Interface (ABI) for X32V is defined with register R16 designated as the Stack Pointer and registers R5 and R6 designated as the Index and Frame registers. The parameter passing is done through registers R8 and R9, with the latter holding the return value.

The associated *makefile* defines certain flags for the target X32V configuration. When set appropriately, *egcs* compiles the binaries for Default, Light and Ultra-light modes as well as any modules that are included in the current configuration. At the moment when the compiler comes across an unsupported instruction, a NOP is inserted. This technique prevents the compiler from crashing and continues outputting binaries.

The EM3 instructions are implemented by using specific library routines and macros that mimic the behaviors of those instructions. Additional instructions can be supported with ease, as the entire machine dependent code is limited to a few specific files. This allows both the compiler and simulator to be highly configurable.

5. Multimedia Support

One of the major goals for X32V is to enhance embedded systems, such as cell phones or PDAs, with multimedia capabilities. In order to be competitive in this arena, X32V required a module that would enhance multimedia operations such as MPEG decoding and encoding. This module is known as the Expandable Multimedia Module (EM3).

EM3 takes advantage of Single-Instruction Multiple-Data (SIMD) operations. There are many successful microprocessors on the market today that rely, in one form or another, on the benefit of multimedia extensions; MMX, SSE, and SSE2 for Intel's Pentium processors [9], 3DNow for the AMD's Athlon processors [10], VIS for Sun's Ultra SPARC processors [11], and Vectra for Tensilica's Xtensa microcontroller [12]. EM3 has the advantage of using an in-house instruction set that is not restricted to any standards, thus allowing the EM3 instruction set to combine the best aspects of several existing multimedia extensions. EM3 currently supports 32-bit and 24-bit instruction sizes.

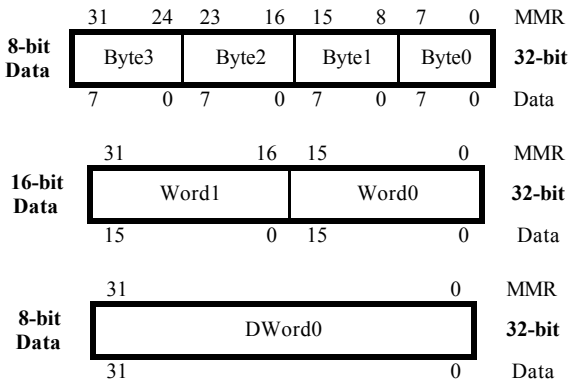

Fig 7. EM3 register formats.

5.1. EM3 Register Formats

X32V supports various floating-point operations, which require a floating-point register file that consist of 16 32-bit floating-point registers. Since floating-point operations and integer multimedia operations are rarely executed at the same time, the EM3 multimedia register file (MMR) is aliased to the floating-point register file. EM3 supports multiple data types including 8-, 16-, or 32-bit signed or unsigned integers. Figure 7 illustrates the various data formats that are available. It is important to note that the 16-bit and 32-bit formats both support signed integer values. Signed 8-bit integer data is not supported, since this data type is rarely used in multimedia applications.

5.2. EM3 Instruction Types

The entire EM3 instruction set can be broken down into five different categories; ALU operations, multiplication/division operations, data conversion operations, data movement operations, and special operations.

EM3 instructions that fall into the ALU operations category involve all of the basic integer arithmetic and logical operations. These instructions include addition, subtraction, AND, OR, XOR, NOT, compliments, and shifts. Each of these instructions can operate on all three of the register data formats. These instructions perform a one-to-one SIMD operation, meaning each segment is operated on independently of the other segments. The addition, subtraction, and compliment instructions can also operate on 16- and 32-bit signed integer data.

Multiplication and division operations involve long cycle times and are some of the most complicated operations in the EM3 instruction set. These instructions can operate on all the register formats, both signed and unsigned and can produce different output formats. For example, multiplication on a 16-bit unsigned register format can produce a 16-bit saturated result, a full 32-bit result, the upper 16-bits of the result, or the lower 16-bits of the result. These types of operations fall into the category of one-to-many SIMD operations, meaning each segment can generate a result consisting of many segments.

Data conversion operations are essential when handling SIMD data as they allow conversion from one register format to another register format. The conversion processes are done using the pack and unpack instructions that compress or expand data into the target register format. Being able to move between formats quickly and efficiently can improve both performance and precision.

Data movement operations consist of instructions that involve moving data to and from the EM3 register file. These instructions are not SIMD instructions, but serve the purpose of creating greater flexibility and performance by allowing data to be moved between the GPR and the MMR as well as providing special memory instructions to load and store data from/to memory. All the memory instructions move 32-bit data from memory, but allow the alignment to be based upon the format of the register. For example, a memory load instruction using an 8-bit format loads 32 bits of data from memory with byte alignment. This is accomplished by using a buffer in a manner similar to the one used when fetching instructions.

Special operations consist of experimental instructions that are used to increase the performance of multimedia applications. These instructions generally do not follow the same format as the aforementioned instructions but allow for the expandability of the EM3 instruction set. A good example of a special operation instruction is the swap instruction. The swap instruction allows a programmer to select four individual bytes from two EM3 registers and places those bytes in any order into another EM3 register. This instruction alleviates the need to call multiple pack and unpack instructions in order to move several bytes around and is useful when performing matrix multiplications or grabbing all of the 8-bit Red components from an RGB word. Future special operation instructions will include a Pixel Distance instruction and Fast Matrix Multiply instruction.

5.3. EM3 Performance

To test the performance of the EM3 instructions and X32V in general, two benchmarks were created based on specific algorithms within MPEG-4. The first benchmark models the color conversion process (YCC) and the second benchmark models both the Inverse Discrete Cosine Transform (iDCT) and the color conversion processes. Each benchmark has two versions, one version uses only the integer instruction set and the other uses the EM3 multimedia extensions. These benchmarks served two purposes;

Fig 8. Resulting image from the EM3 multimedia benchmark.

Fig 9. Close up of the original image.

Fig 10. Result from the EM3 simulation.

the first was to validate the simulator and the second was to study the benefits of the configurable architecture.

To make the benchmark as much like MPEG-4 decoding as possible, the benchmarks were written to use sub-sampled macro blocks. For each 16×16 block of 24-bit RGB pixels, there were 256 Y samples, 64 Cb samples, and 64 Cr samples. To allow

TABLE I
BENCHMARK CYCLE COUNTS

Benchmark	YCC	iDCT	BOTH
EM3	6,620,878	15,455,928	22,076,806
Integer	20,098,719	24,816,902	44,915,621

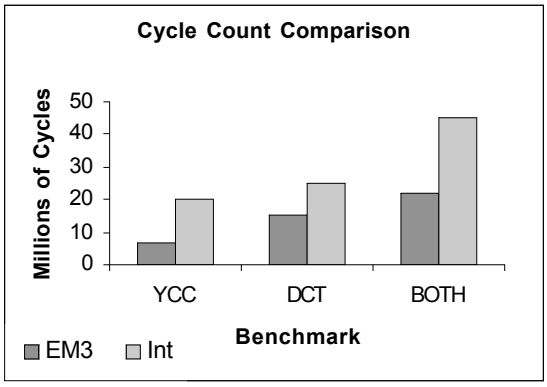

Fig 11. Media benchmark results

for quick validation, the benchmark writes the data to a file as a 24-bit color bitmap image. Figure 8 shows an example of the images created by the benchmarks. This image uses the standard Windows .bmp file format to allow for easy viewing. The effects of sub-sampling are illustrated clearly in Figures 9 and 10.

The most time consuming aspect of both benchmarks were the large number of fractional multiplications required. These multiplications were implemented using integer multiply instructions followed by a right shift. Within the iDCT benchmark, the 2-D iDCTs required in MPEG-4 decoding were performed as a 1-D iDCT on all the rows of the 8×8 sub-blocks followed by a 1-D iDCT on all the columns of the 8×8 sub-blocks. To maintain the required accuracy, 7 fractional bits were used in our cosine values and 4 fractional bits were kept for all our intermediate results. These fractional bits limited the final errors to within ±1 ulp, as required by most standards.

By using the EM3 operations, YCC used 3 times fewer cycles than when just using integer operations. The iDCT portion of the combined EM3 benchmark was about 50% faster than the same portion of the integer version. Table I contains the exact cycle counts from running each of the benchmarks for both the EM3 version and the integer version and Figure 11 illustrates these results. It should also be noted that the images created by the EM3 versions were identical to the images from the integer versions. The SIMD operations used by the EM3 version did not cause any additional round off or truncation errors.

By configuring the processor with additional functional units, it is possible to obtain a very large increase in performance. The additional functional units allow the processor to target a specific application without the additional cost or overhead associated with a dedicated co-processor.

6. Conclusion

Due to the advances in small, handheld computational devices, there has been a strong demand for cheap, low-power, configurable microprocessors. X32V was designed as a general-purpose microprocessor that could be configured for specific tasks and requirements.

One of the features that make X32V unique amongst other configurable microcontrollers is its ability to support variable instruction lengths that ultimately allow a designer to improve code density. X32V supports the use of modules that allow a designer to target X32V for a specific system. To support multiple configurations, a retargetable compiler was developed. With the compiler, the X32V becomes a highly cost effective method for modern applications since time is not spent on building unique compilers for each situation.

Of all the modules created for X32V, EM3 has proven to be a key element in improving the performance of running multimedia applications. As research continues, the EM3 module will continue to grow and improve the performance of MPEG-4 applications.

7. References

[1] Takaki, S. *et. al.*, "Hardware/Software Partitioning Methodology for Systems on Chip (SOCs) with RISC Host and Configurable Microprocessor," *International Workshop on IP based System-on-Chip Design*, 2003.

[2] Gonzalez, R.E., "Xtensa: A Configurable and Extensible Processor," *IEEE Micro*, Vol. 20, No. 2, March/April 2000.

[3] Sethia, A., "Solving System on Chip Design Challenges with the ARCform Development Platform," White Paper, ARC Cores Ltd., San Jose, CA, 2001.

[4] Burger, D., and T.M. Austin, "The SimpleScalar Tool Set, Version 2.0," *ACM SIGARCH Computer Architecture News*, Vol. 25, Issue 3, June 1997.

[5] Lee, C., M. Potkonjak, and W.H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," *30th Annual International Symposium on Microarchitecture (MICRO-97)*, Dec. 1997.

[6] Hennessy, J.L., and D.A. Patterson, *Computer Architecture: A Quantitative Approach, 3rd Edition*, Morgan Kaufmann Publishers, San Francisco, 2003.

[7] Stallman, R., *Using and Porting GCC*, Free Software Foundation, Boston, 1990.

[8] Aho, A.V., R. Sethi, and J.D. Ullman, *Compilers: Principles, Techniques and Tools*, Addison Wesley Publications, Boston, 1986.

[9] Mital, M., A. Peleg, and U. Weiser, "MMX Technology Overview," *Intel Technology Journal*, Q3, 1997.

[10] Advanced Micro Systems, *AMD Technology Manual*, Advanced Micro Systems, Sunnyvale, CA, 2000.

[11] Sun Microsystems, *VIS Instruction Set User's Manual*, Sun Microsystems, Santa Clara, 1997.

[12] Leibson, S., "SOC-Based Signal Processing: Meeting Performance Goals With Tailored DSPs," *Global Signal Processing Expo & Conference (GSPx 2003)*, 2003.