ATMEL WHITE PAPER

MCU Architectures for Compute-Intensive Embedded Applications

Jo Uthus, Øyvind Strøm, PhD.

Atmel Corporation

Summary

This paper describes the challenges that developers face with embedded applications that require intensive computing tasks. Innovative architectural techniques that help solve those issues are discussed and a new 32-bit RISC architecture is described and evaluated through benchmark results.

Table of Contents

Computational Requirements for DSP-based Embedded Control Systems

The increasing use of complex algorithms in embedded control systems is substantially adding to processing overhead. Fast Fourier Transforms (FFTs), inverse Discrete Cosine Transformation (iDCTs) and other compute-intensive algorithms that require single-bit manipulation, matrix mapping in addition to byte and half-word arithmetic are becoming common in applications that were unimaginable just a few years ago. These include data compression, signal coding, vehicle passenger safety systems, and portable infotainment systems which use MP3 audio and MPEG-4 video codecs. All require advanced, computationally-intensive DSP algorithms.

Historically, the solution to increased processing overhead was to simply increase the processor clock rate. More recently, dual-core processors that include an MCU with one or two DSPs have been targeted at these applications. Unfortunately, the very applications that employ advanced DSP algorithms are increasingly being implemented in battery operated end-products, such as PDAs, cell phones, POS devices, portable media players.

Increasing the processor clock rate has a direct and proportional effect on power consumption. Using multiple processors (e.g. DSP and MCU) has a similar effect because it increases the capacitance and the total amount of gating activity. In addition, developing and debugging code for multi-processor systems is a nightmarish experience that can add months to the development cycle. How does an engineer determine what code for which processor is causing a system glitch?

The exceptionally small size of these end-products means that battery size and, therefore, battery capacity are severely limited with active power consumption budgets as low as 2 mW/MHz. So, how do you get the required throughput and the ultra-low power consumption required for these applications?

The traditional methods of turning up the processor clock or adding multiple processors simply don’t work when power consumption is important. New processor architectures must be developed that directly support the computational intensity of these applications with a fraction of the power consumption.

Processing Requirements for MPEG-4 Decoding.

MPEG-4 is a video encoding standard that allows universal, low bit-rate data transfer of video by reusing most of the data from the first video frame and transferring only those bits that have changed from one frame to the next

The basis of MPEG-4 video coding is a block-based predictive differential video coding scheme. The main techniques for compression are: division of the picture in 8x8 blocks or 16x16 macroblocks (MB), motion-compensated prediction, transform coding with discrete cosine transform (DCT), quantization, and run-length and Huffman coding for variable length codes (VLC).

Both spatial redundancy and irrelevancy are exploited with block-based DCT coding, quantization, run-length and Huffman coding. Only information from the picture itself is used, and thus, every frame can be decoded independently.

Additionally, in MPEG-4 inter-mode encoding, the temporal redundancy between the images in a video sequence is taken into account. A macroblock-based motion estimation between two successive images is done allowing a motion-compensated prediction of the current picture. The predicted image is then subtracted from the original image and the resulting difference picture is DCT-coded, quantized and VLC coded. The motion vectors describing the motion of the blocks in the picture are used later for decoding and are also encoded with VLC.

Both encoding and decoding are computationally intensive. During encoding, the sum of the absolute differences (SAD) must be calculated for all pixels in each frame. A qualitative measure of the distortion is assigned to each motion vector. The sum of absolute differences for an NxN block located at position *(x, y)* with a given displacement *(dx, dy)* in the reference Group of Video Objects (VOP), *ref*, is derived as follows:

$$SAD = \sum_{i=0}^{N-1}\sum_{j=0}^{N-1}\left|pred(x+i, y+j) - ref(x+dx+i, y+dy+j)\right|$$

A C-language representation of the algorithm is shown below:

```
/* From sad.c of the open source xvid codec */
uint32_t sad = 0;
uint32_t j;
uint8_t const *ptr_cur = cur;
uint8_t const *ptr_ref = ref;

for (j = 0; j < 8; j++) {
                //Compute SAD for 4 bytes
                sad += ABS(ptr_cur[0] - ptr_ref[0]);
                sad += ABS(ptr_cur[1] - ptr_ref[1]);
                sad += ABS(ptr_cur[2] - ptr_ref[2]);
                sad += ABS(ptr_cur[3] - ptr_ref[3]);
                //Compute SAD for next 4 bytes
                sad += ABS(ptr_cur[4] - ptr_ref[4]);
                sad += ABS(ptr_cur[5] - ptr_ref[5]);
                sad += ABS(ptr_cur[6] - ptr_ref[6]);
                sad += ABS(ptr_cur[7] - ptr_ref[7]);
                      ptr_cur += stride;
                      ptr_ref += stride;
                 }
```

Every line in this computation contains an addition, absolute value and a subtraction. That is the equivalent of three operations per line of code. Implementing this in a RISC architecture using standard arithmetical instructions would require 24 operations in addition to any non-arithmetic operations to align memory with load/store instructions. If the architecture is single-cycle, that equates to a minimum of 24 cycles for the 8x8 SAD algorithm in addition to the other parts of the algorithm. This algorithm is run approximately 60-70% of the time during MPEG-4 encoding putting a significant computational strain on most RISC architectures.

The decoding of MPEG-4 data streams includes a very compute-intensive 2-D 8x8 iDCT algorithm,

$$f(x,y) = \frac{2}{N}\sum_{u=0}^{N-1}\sum_{v=0}^{N-1} C(u)C(v)F(u,v)\cos\frac{(2x+1)u\pi}{2N}\cos\frac{(2y+1)v\pi}{2N}$$

$$C(u),C(v) = \begin{cases} \frac{1}{\sqrt{2}} & \text{for u, v} = 0 \\ 1 & \text{for u, v} \neq 0 \end{cases} \qquad (4.1)$$

where u, v, x, y = 0, 1, 2, … ,N-1.

Figure 1. iDCT Algorithm

The small screens on a phone or PDA do not require and cannot display 640 x 480 VGA resolution. The standard for small-screen devices is quarter-VGA (QVGA). Even at one-fourth the resolution, the computational load is about 2 billion operations per second. That’s a huge amount of processing. So the question is: How does an engineer get both the computational throughput *and* keep power consumption to a minimum.

The ARM926EJ-S™ core, used in the Freescale™ iMX21 application processor, requires a 266 MHz clock (full running operation) to decode a quarter-VGA (QVGA) encoded movie, and consumes 232 milliwatts of power per second while doing so. The ARM11™ core, which includes Single Instruction Multiple Data (SIMD) instructions, requires a 150 to 175 MHz clock frequency and is expected to consume between 60 and 70 mW in a 0.13 micron process technology.

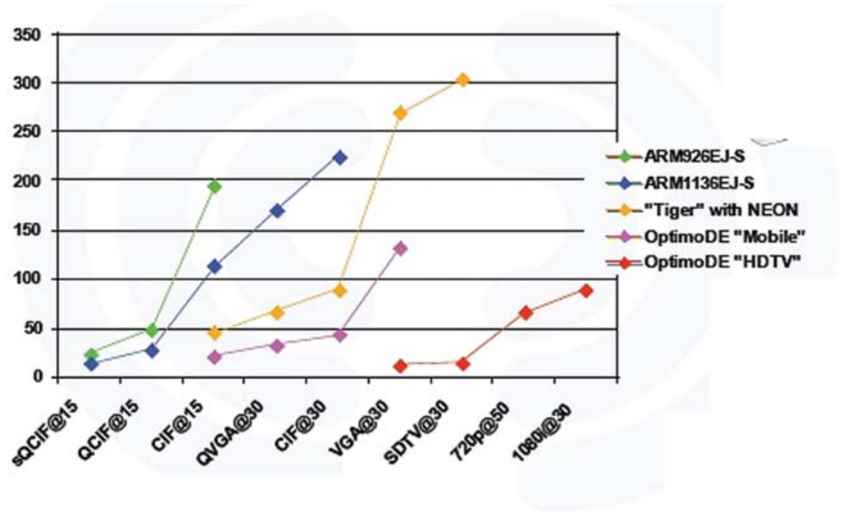


Figure 2. ARM® Processor Core MPEG-4 Decoding

When you consider that the systems are typically powered by a 750mAh, 3.6V battery which must also power the backlight LCD, amplifiers, regulators, RF modules and cameras, these levels of power consumption are less than ideal for power-constrained, portable applications.

Another solution is to use a DSP whose architectures are designed for very efficient computational processing with relatively slow clocks. For example, TI's older C5510 DSP can execute MPEG-4 QCIF (smaller than QVGA) decoding at slightly less than 200 MHz while consuming only 50 mW of power. The problem with DSPs, however, is that they are incapable of performing embedded control functions which are required for the system to function at all. Multi-functional end-products, such a PDAs or media players, execute many more functions than MPEG-4 or MP3 decoding. They must manage the user interface and juggle processor cycles among a variety of functions (audio, video, memory management, polling, user input, flash or disk-drive access, RF baseband decoding and more). DSPs cannot do these functions and as a result, virtually every system that uses a DSP for computation also has an external or on-chip 8- or 32-bit microcontroller to manage system functionality.

Using two processors increases power consumption and design complexity because it requires that code be written and debugged for *both* processors. This can add months to the product development cycle. How does an engineer develop and debug code for multiple processors in the same system? How does an engineer manage multiple in-circuit emulators (ICEs), different compilers and completely differing code development methodologies? How does an engineer interface these devices to the system bus and to the end-user?

Increased development time, increased test time, increased failure rates in production and reduced market share due to slipped delivery dates are the end results of this complexity. The bottom line is that for computationally-intensive embedded control applications such as MPEG-4, data compression and passenger safety systems, conventional MCUs and

DSPs consume too much power. Conventional MCUs require very fast clocks and DSPs require a second processor as well as a complex design cycle. A better solution would be a new processor architecture with an instruction set that supports single-cycle execution of frequently used operations and combinations thereof (FFT, SAD, iDTC, etc), has the command and control functionality of a microcontroller, and can employ advanced computational throughput to keep power consumption to a minimum all of which are required by these applications.

Architecture Enhancements to Increase Throughput

Traditionally, increased processing throughput has been achieved by increasing the processor clock frequency. Unfortunately, increasing the clock rate has a direct effect on power consumption which may be unacceptable in some portable applications. An alternative to increasing the clock frequency is to modify the processor architecture to increase the amount of computation that can be done with each cycle.

Some of the factors that affects a CPU efficiency to execute a particular function include: 1) the number of cycles used to move data into and out of the processor (load/store operations) 2) the execution efficiency of individual instructions in the pipeline 3) branches (4) the programming model; and 5) program code density.

Several steps can be taken to improve the computational throughput of the CPU.

Reduce the amount of load store cycles. More than 30% of the instructions executed in a RISC architecture are load or store instructions, each of which takes one or more cycles. Reducing the number of load/store cycles can have a substantial effect on processor throughput.

Streamline repetitive operations. Some algorithms, such as those for multimedia, contain operations that are repeated thousands of times on streams of data. For example, the 8x8 sum of absolute differences (SAD) algorithm contains 24 operations that must be executed on every pixel of an image during MPEG-4 encoding. Performing these operations on multiple data simultaneously (Single Instruction Multiple Data or SIMD) results in a linear reduction in cycles required to process the data stream.

Maximize utilization of pipeline resources. Some arithmetic operations take a single cycle while others take several cycles. For example, a division operation can take 32 cycles to execute. If the processor must wait for a multi-cycle operation to complete before issuing a new instruction, the other resources in the pipeline will be under-utilized. Allowing unused pipeline resources to be used for non-dependant calculations (out of order execution) increases the utilization of these resources and increases throughput per clock cycle.

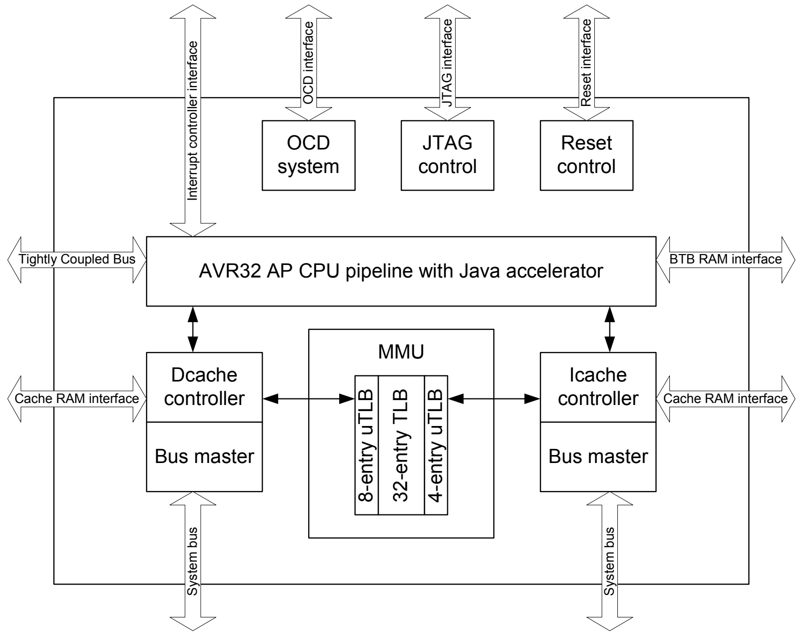
Different algorithms use operators in different ways. Providing a variety of instructions (e.g. multiplies) that fully exploit computational resources for target algorithms can increase throughput.

Minimize branch latency. Most multimedia and cryptography algorithms, consist of outer and inner loops. Branches for tight inner loops can consume three to five cycles each. In DSP algorithms some inner loops are executed tens of thousands of times a second and their branches can consume an enormous number of cycles. Implementing logic that predicts the branch and folds it into the instruction can reduce the branch penalty to zero cycles after the first loop iteration.

Improve code density. Since memory is relatively inexpensive, few people worry about code density. However, with processors that rely on an instruction cache for fast performance, code density can have a direct effect on performance. If the code is smaller, more instructions can be stored in the cache, resulting in fewer cache "misses" and fewer cycle-intensive fetches from external memory. Reducing the traffic on the main system bus can also significantly reduce power consumption.

AVR®32: High-performance, Low-power MCU Architecture

Atmel has developed a high-performance 32-bit RISC processor core with an instruction set architecture that vastly increases the computational throughput per cycle while also delivering ultra-low power consumption. The AVR32 core minimizes penalties from load/store and branch operations and maximizes pipeline throughput, allowing complex algorithms to be executed with a much lower clock frequency and power consumption than conventional processors.



On target algorithms, such as SAD and iDCT, the AVR32 achieves 35% more throughput per instruction cycle than an ARM11 core. It can execute 30 frame per second (fps) quarter-VGA MPEG-4 decoding with a clock frequency of just 100 MHz compared to the 150 to 175 MHz required in the ARM11 architecture.

Minimizes cycles expended for load/store operations. Atmel has extensively benchmarked target applications to find the optimal blend of load and store instructions. The AVR32 architecture has load/store instructions supporting byte (8 bit), half-word (16 bit), word (32 bit) and double word (64 bit) widths. The instructions are combined with various pointer arithmetic to efficiently access tables, data structures and random data. Instructions for loading bytes and half-words all have optional sign or zero extension of the data value.

Cryptography - Among the most popular algorithms for cryptography are the block cipher algorithms of which Blowfish, Triple-DES and Rijndael are examples. All of these algorithms use a special array addressing operation, which on current microprocessors, requires a long instruction sequence to execute. The AVR32 instruction set supports these algorithms with a new and innovative load instruction that loads a word with extracted index. The operation is as follows:

```
result =       pointer0[offset0 >> 24] ^ pointer1[(offset1 >> 16) & 0xff] ^
pointer2[(offset2 >>  8) & 0xff] ^ pointer3[offset3 & 0xff];
```

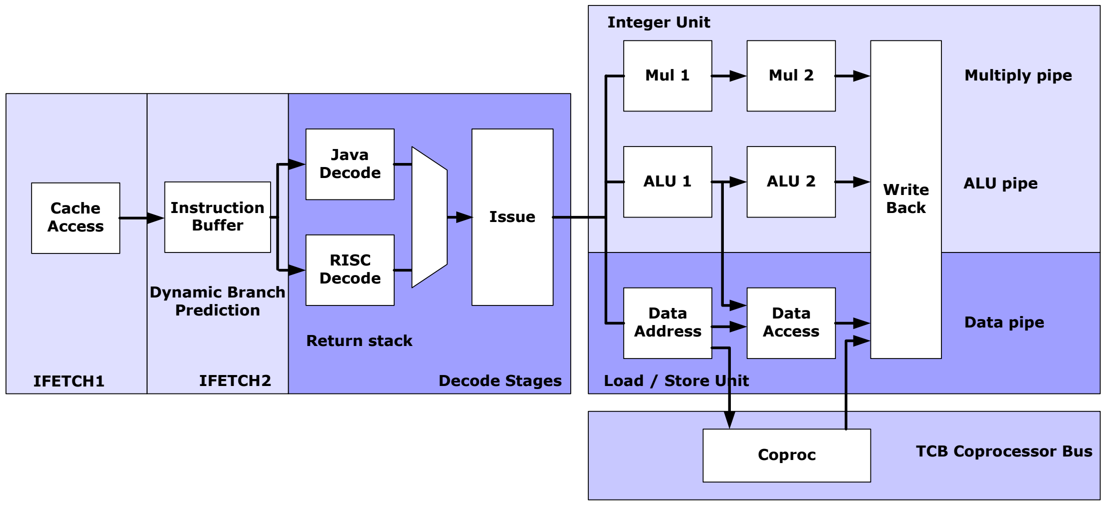
Four memory access operations are dominant in this operation which extracts one of the four bytes in a 32-bit word, zero-extends it and adds it to a base pointer. The result of this operation generates the memory address to be accessed.

A conventional architecture would need 14 cycles to execute this operation. The AVR32 can execute it in just seven cycles, The AVR32's load with Extracted Index instruction can perform all four memory accesses in four cycles while keeping all four offsets in one register.

Another example of an efficient load/store instruction is the Load Multiple Register instruction (ldm). Used in subroutine returns in combination with the Store Multiple instruction, it will fetch two and two registers from the instruction cache. The instruction can be used to return from a subroutine while the last register written is the program counter. This eliminates the need to execute a return instruction at the end of a subroutine.

Multiple pipelines support out-of-order execution. Generally, processors execute instructions one at a time, as they enter the pipeline. If a particularly complex instruction requires multiple clock cycles, the pipeline is stalled until that instruction is complete.

Available computational resources are left idle during this instruction. The AVR32 has



extensive logic in the pipeline that allows it to accomplish more processing per cycle.

The AVR32 has three pipelines (load/store, multiplier and ALU) that allow arithmetic operations on non-dependent data to be executed *out of order*. For example, if a 32-cycle divide operation enters the pipeline, instructions that follow it in the code may be executed in the ALU and/or load/store pipes during those 32-cycles. Rather than halting the code until the division is complete, as is the case in most processors, the AVR32 allows instructions to execute using available resources. *Hazard detection* logic detects and holds dependent instructions at the beginning of the pipeline until the dependent operation is complete.

Another cycle-saving feature of the AVR32 is the data forwarding between the pipeline stages. Instructions that finish execution before the writeback stage are immediately forwarded to the beginning of the pipelines if there are instructions waiting for the results. For instance when an add instruction finishes in the ALU1 pipeline stage, the result is forwarded back to the MUL1, ALU1 and Data Address stages. This allows the result to be used one cycle after the add was issued instead of waiting three cycles to pass it to the writeback stage. The AVR32 implements a full forwarding scheme meaning that not only selected instructions, but all instructions, forward their results as soon as they are finished.

Single-instruction multiple data (SIMD) in the AVR32 architecture can quadruple the throughput of DSP algorithms that require repetitive operation on a stream of data (e.g. motion estimation for MPEG decoding).

For example, in the AVR32, an 8-bit sum of absolution differences (SAD) is calculated by loading four 8-bit pixels from memory in a single load operation, executing a packed subtraction of unsigned bytes with saturation, adding together the high and low pair of packed bytes and unpacking them into packed half-words. The half-words are then added together to get the SAD value.

Branch prediction and folding can achieve zero-cycle penalty in loops. Although deep pipelines enable higher clock frequencies, they introduce significant cycle penalties when there are jumps in the program flow. These "branch penalties" are particularly harsh for all algorithms implementing small inner loops. To address this problem, the AVR32 pipeline has branch prediction logic that can accurately predict ***all*** change-of-flow instructions. In addition, branches are "folded" with the target instruction resulting in a zero-cycle branch penalty.

Exceptional code density. The AVR32 achieves exceptional code density due to extensive benchmarking and refinement of the instruction set together with the compiler vendor. The AVR32's high code density means that the code occupies less space in the cache, so a larger number of instructions can be stored in cache memory thus reducing the number of cache misses.

Instruction set support for advanced operating systems. The majority of CPU architectures were developed before operating system (OS) use became as pervasive as it is today. As a result, CPU cores tend to waste cycles calling the OS or external applications. The AVR32 architecture specifically supports the use of the Linux® OS with cycle-saving instructions. These include an Application Call (ACALL) instruction that calls sub routines from a jump table with an 8-bit index, allowing for more compact code, and a System Call (SCALL) instruction that issues a call to the operating system routine. The AVR32 comes fully-featured with an advanced MMU and security modes to support advanced operating systems such as Linux.

Ultra-low power consumption for portable applications. The high-throughput architecture of the AVR32 means that more work can be done in fewer clock cycles. The AVR32 can execute 30 fps QVGA MPEG-4 decoding at 100 MHz. The closest competitor, the ARM11 core, requires 150 to 170 MHz for this task.

As a rule of thumb, it can be said that 80% of the power features are decided when the instruction set architecture (ISA) is designed. Only 20% can be decided upon during the implementation phase. Keeping data close to the CPU core helps to minimize power consumption. Therefore, engineers partition algorithms to keep data as close to the CPU core as possible because the power cost of going to external, or even on-chip, memory is huge.

The AVR32 is designed with low power in mind. Choices were made specifically to minimize the unnecessary movement of data on buses that consume a lot of power. For example, the programming model includes a link register in the register file, so subroutine calls don't need to copy the return address to a stack in memory – something commonly done in conventional MCU architectures that unnecessarily burns power. Among other features, the status register and the return address for interrupts and exceptions are kept in system registers to avoid moving data to and from the stack which is costly with regards to power consumption.

Data is always kept as close to the CPU core as possible to minimize the power consumption on the main system bus.

In addition, AVR32 code is exceptionally dense – 50% denser than performance-optimized ARM code. Smaller code means more instructions can be stored in the instruction cache, reducing the need expend cycles or power on code fetches from external memory.

AVR32 Constently Outperforms In EEMBC® Benchmarks

In EEMBC benchmarks, AVR32 code consistently requires 5% to 20% less code than the ARM Thumb® instruction set to execute the same functionality. More significantly, for high-performance applications, when optimized for speed of execution, AVR32 code is 30% to 50% more compact.

The AVR32 consistently outperforms both ARM9™ and ARM11 cores in EEMBC's TeleMark™, OAMark™, AutoMark™, ConsumerMark™ and NetMark™ benchmarks. Its performance exceeds that of the ARM11 by over 35% on the ConsumerMark benchmark.

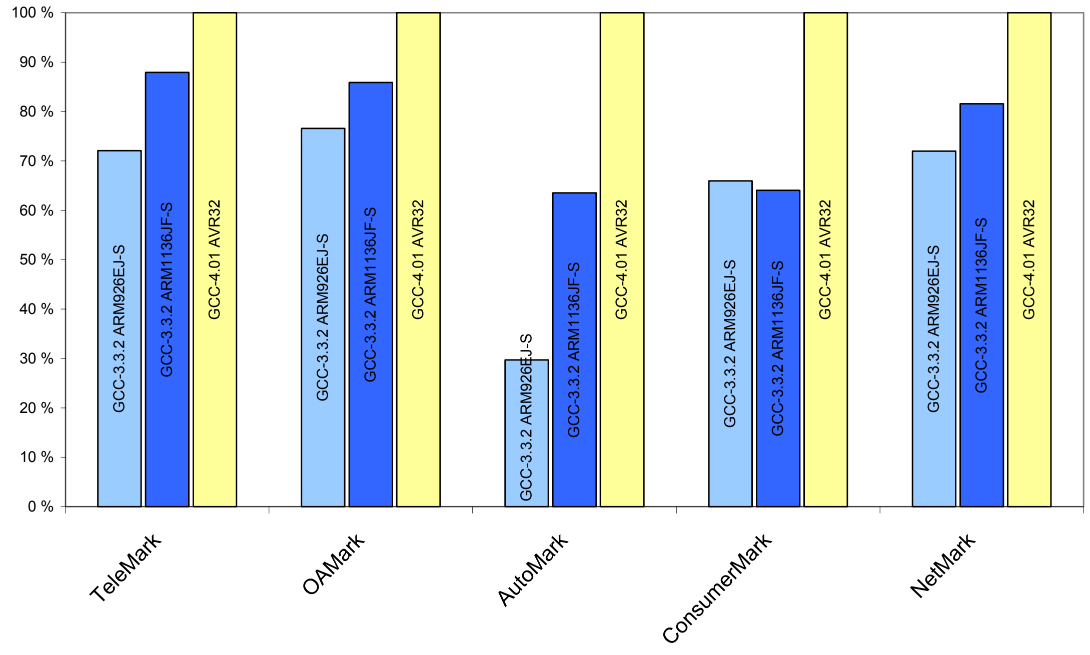


Figure 3. EEMBC normalized performance benchmarks for AVR32, ARM9 and ARM11.

The EEMBC benchmark numbers are normalized to the same clock frequency. The ARM926EJ and ARM1136JF benchmark data can be found at the EEMBC website (www.eembc.org) and are based on the Freescale i.MX21 and i.MX31 values. Figure 3 shows the architectural performance when comparing CPU cores with a higher value indicating a better performance

AVR32 performance-optimized code density is also consistently better than that of the ARM ISA in every EEMBC benchmark.

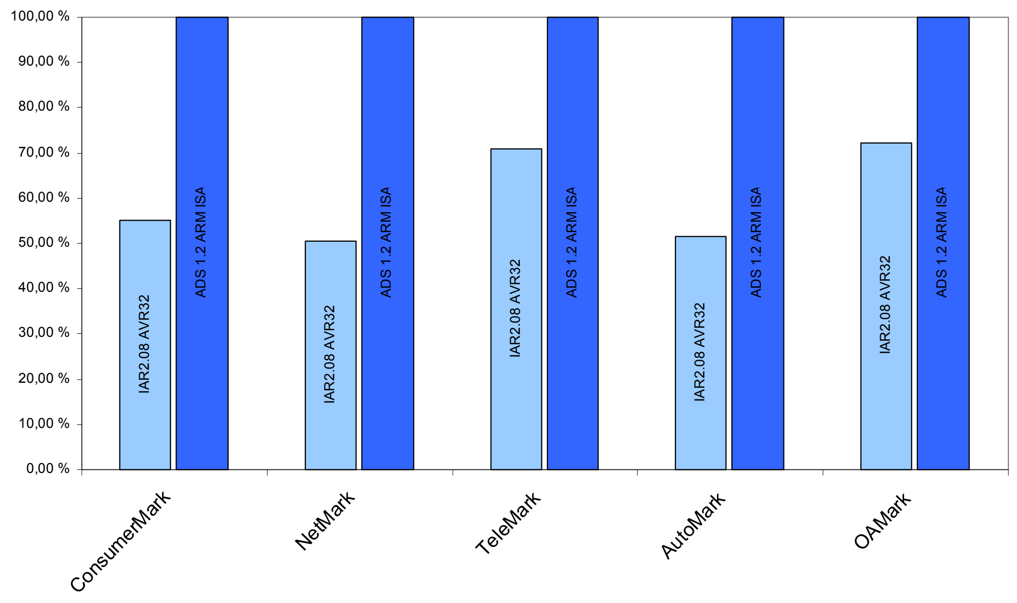


Figure 4. Code size on EEMBC Marks optimized for speed.

EEMBC code size benchmarks are based on publicly available numbers from EEMBC (www.eembc.org), based on ARM's simulated numbers for the ARMv5 Thumb ISA. AVR32 code was compiled with the IAR AVR32 compiler while ARM-code was compiled with ARM's ADS 1.2 compiler. Figure 4 shows the advantage of the AVR32 in regards to code size with a lower value indicating better performance.

Conclusion

For cost-sensitive, power-constrained, DSP-based embedded applications, the AVR32 core exhibits superior instruction execution throughput, code density and power consumption.

Atmel will launch a family of controllers targeted at these applications in 2006.

Editor's Notes About Atmel Corporation

Atmel is a worldwide leader in the design and manufacture of microcontrollers, advanced logic, mixed-signal, nonvolatile memory and radio frequency (RF) components. Leveraging one of the industry's broadest intellectual property (IP) technology portfolios, Atmel is able to provide the electronics industry with complete system solutions. Focused on consumer, industrial, security, communications, computing and automotive markets, Atmel ICs can be found Everywhere You Are®

Further information can be obtained from Atmel's Web site at www.atmel.com

Contact:

Jo Uthus, Marketing Manager, Tel: (+47) 72897593, e-mail: juthus@atmel.com
Øyvind Strøm, PhD, Project Manager, Tel: (+47) 72 89 75 15, e-mail: ostroem@atmel.com
Atmel Corporation, Vestre Rosten 79, 7075 Tiller, Norway