Keywords: MAXQ, RISC, microcontroller, maxq risc architecture, micros

APPLICATION NOTE 3222

Introduction to the MAXQ Architecture

May 10, 2004

Abstract: The MAXQ RISC architecture combines high performance and low power with a variety of complex analog functions.

Introduction to the MAXQ™ Architecture

Microcontroller system designers today have a myriad of choices when it comes to selecting a microcontroller for a project - 8-bit, 16-bit, RISC, CISC, or something in between. As a rule, many criteria are considered during the selection process. These can include price, performance, power, code density, development time, and even future migration-path alternatives. To complicate the selection process, tight demands for one criterion generally influence the options in other areas. Factors critical in one application may have little importance in another. Consequently, there is no one microcontroller that is perfect for all projects. But to be successful, a modern microcontroller must excel in many of the areas under consideration.

When world-renowned analog chipmaker, Maxim Integrated Products, joined forces with the industry-leading high-performance microcontroller supplier, Dallas Semiconductor, an opportunity to integrate superior analog functionality with leading-edge microcontrollers was created. One result of this partnership is the MAXQ RISC architecture, a new microcontroller core that combines high performance and low power with a variety of complex analog functions.

When integrating complex analog circuitry with high-performance digital blocks, the operating environment should be kept as quiet and noise-free as possible. However, the clocking and switching that occur in the digital circuits of a microcontroller core inject noise into the sensitive analog section. Therein lies the difficulty facing the mixed-signal designer: to achieve high microcontroller performance, but minimize clock noise that can affect sensitive analog circuits.

The MAXQ architecture reduces noise through intelligent clock management and utilization. This means that the MAXQ core enables clocks only to those circuits that require clocking at any instant, thus reducing power consumption and providing a quiet environment optimal for analog integration. Additionally, the MAXQ architecture performs many functions on each clock to maximize its performance. This article provides an overview of the MAXQ architecture and highlights its competitive advantages.

No Wasted Cycle Clocks

The MAXQ architecture was designed to achieve a high performance-to-power ratio. The first requisite in generating a high-efficiency machine is to maximize utilization of the clock cycles for user code execution.

The most fundamental way that the MAXQ achieves high utilization is through single-cycle instruction

execution. Single-cycle instruction execution benefits the end user by increasing instruction bandwidth that leads to higher performance, and/or reduced power consumption made possible by the ability to reduce clock frequency. All MAXQ instructions execute in a single clock cycle except long jump/long call and certain extended register accesses. While many RISC microcontrollers claim to support single-cycle execution, this often applies to a small subset of instructions or addressing modes. With the MAXQ, single-cycle execution is the norm.

Secondly, the MAXQ architecture achieves increased clock-cycle utilization because it does not require an instruction pipeline (common to many RISC microcontrollers) to achieve single-cycle operation. The MAXQ instruction decode and execution hardware is so simple (and timing so fast) that these operations are moved into the same clock cycle as the program fetch itself, with minimal impact to the maximum operating frequency. To illustrate the benefit of eliminating the instruction pipeline, consider the generic RISC CPU that executes from a pipeline. When a program branch occurs, the CPU uses one or more clock cycles (depending upon pipeline depth) to divert program fetching to the target branch address and discards the instruction(s) already fetched. Clearly, using clock cycles to discard instructions, versus executing them, is wasteful and undesirable as it reduces performance and increases power consumption. While the operation is undesirable to the user, the clocks stolen by the CPU to reload the pipeline are an artifact of the architecture and are unavoidable. The MAXQ architecture distinguishes itself from other 8-bit and 16-bit RISC microcontrollers by offering single-cycle execution without an instruction pipeline (and the wasted clock cycles that accompany it).

The MAXQ Instruction Word

The MAXQ instruction word is unique because there is only one instruction in the classical sense, the "MOVE" instruction. The source and destination operands for the "MOVE" instruction are the basis for creating instructions and memory accesses, and triggering hardware operations. Dissecting the 16-bit MAXQ instruction word reveals only two components: a 7-bit destination field and an 8-bit source field accompanied by a source format bit. The source format bit, when coded as 0, allows any immediate or literal byte value (i.e., #00h-#FFh) to be supplied as a source operand. Unrestricted support for any immediate byte source within a single instruction word can be very valuable during register initialization routines and when performing ALU operations. The nonliteral source and destination possibilities are subdivided into smaller groups, or modules. **Figure 1** illustrates the 16-bit MAXQ instruction word.

FORMAT	DESTINATION	SOURCE
f	ddd dddd	ssss ssss
1	INDEX MODULE	INDEX MODULE
0	INDEX MODULE	IMMEDIATE BYTE DATA (i.e., 00h–FFh)

Figure 1. The MAXQ instruction word is simple, yet very powerful.

All machine instructions reduce to source and destination operands for a transfer operation. These operands can be used to select physical MAXQ device registers. This type of transfer is the most basic and quite easy to imagine. In the MAXQ machine, however, the source and destination operands are not rigidly associated with physical registers.

The MAXQ architecture uses this same source-to-destination transfer construction when performing

indirect memory access. Certain destination and/or source encodings are identified as indirect access portals to physical memories such as the stack, accumulator array, and data memory. These indirect memory access portals use physical pointer registers to define the respective memory address locations for access. As an example, one way that the data memory can be accessed indirectly is using the "@DP[0]" operand. This operand, when used as a source or destination respectively, triggers an indirect read or write access to the data memory location addressed by the Data Pointer 0 (DP[0]) register.

The MAXQ architecture also uses special destination and/or source encodings to trigger underlying hardware operations. This trigger mechanism serves as the basis for creating MAXQ instructions that are implicitly linked to certain resources. For example, math operations (ADD, SUB, ADDC, and SUBB) are implemented as special destination encodings that implicitly target one of the working accumulators, with only the source operand supplied by the user. Conditional jumps implicitly target the instruction pointer (IP) for modification and are implemented as separate destination encodings for each status condition that can be evaluated.

The indirect memory access and underlying hardware-operation triggers are combined whenever possible to create new source/destination operands, which provide dual benefits. The autoincrement/decrement indirect-access mnemonics for the data pointers demonstrate this combination. When reading from data memory with DP[0], the user can optionally increment or decrement the pointer following the read operation using the "@DP[0]++" or "@DP[0]--"source operand, respectively.

Numerous advantages come as a result of the MAXQ instruction word. The instruction word contains modularly grouped source and destination operands, which allow simple and fast instruction-decoding hardware and limit signal switching for those modules not involved in the transfer, thus reducing dynamic power consumption and noise. The instruction word uses its full 16 bits to specify source and destination operands, producing an abundant address space for physical registers, indirect memory access, and hardware-triggered operations. Ultimately, coupling the abundant source/destination address space with minimal restrictions on sourcedestination combinations gives rise to a highly orthogonal machine.

MAXQ System Highlights

The MAXQ system not only provides the basic hardware resources and capabilities expected by today's microcontroller users, but it also enhances these resources and adds new features to expand device functionality and utility. While it is not feasible to document all the MAXQ system resources, some are discussed here.

Working accumulators

The MAXQ architecture thus far has been addressed as a single entity. However, two slightly different versions, the MAXQ10 and MAXQ20, will be implemented in the initial MAXQ product family launches. The primary difference between the MAXQ10 and MAXQ20 options is the standard width of the working accumulators and supporting arithmeticlogic unit (ALU). The MAXQ10 supports 8-bit (byte-wide) accumulators and ALU operations, while the MAXQ20 supports 16-bit (word-wide) accumulators and ALU operations. The MAXQ devices come equipped with a minimum quantity of eight accumulators and, depending on the application, can have as many as 16 accumulators. In the source/destination transfer map, these accumulators are located in a system register module and are each directly accessible as A[n], where *n* corresponds to their respective index. So, a MAXQ device equipped with 16 accumulators would contain accumulators A[0], A[1]...A[14], and A[15]. Any one of the accumulators can be designated as the active accumulator and indirectly accessed through the Acc mnemonic by setting the accumulator pointer register, AP, to its specific index (i.e., Acc = A[AP]). The AP register implements only the number of bits necessary to provide a binary decode into the accumulator array, so four bits are required in MAXQ devices having 16 accumulators. All ALU operations implicitly specify the active accumulator as the destination for the operation being performed. Take, for example, the "ADDC src" instruction. This

instruction always performs the addition operation between the active accumulator, the carry flag, and the source (src) operand specified. Awealth of bit manipulation and shift/rotate instructions surround the active accumulator.

Additional hardware is attached to the accumulator pointer to expedite ordered and predictable accesses to the accumulator file. The accumulator-pointer control (APC) register provides bits for resetting AP and for streamlining increment, decrement, and modulo operations on the accumulator-pointer register.

The processor status flag (PSF) register contains five status flags, which have special meaning in relation to the active accumulator status and ALU operations. These are the (C)arry, (Z)ero, (S)ign, (E)qual, and (OV)erflow status flags. Some of these flags can be evaluated for performing conditional jumps and returns. The PSF register also provides two additional generalpurpose flags (GF1 and GF0) for user software needs.

Dedicated hardware stack

The MAXQ architecture contains a dedicated hardware stack. The stack depth for any MAXQ device is product dependent. A dedicated hardware stack has two distinct advantages. Firstly, it allows data memory to be preserved for other application uses instead of being consumed by stack, and secondly, it supports fast PUSH/POP operations because a dedicated read/write port exists, which need not be shared with data memory. If the hardware stack depth is insufficient for the context storage needed, then the stack-like operation of the data pointers (pre-increment/decrement for writes, postincrement/ decrement for reads) is ideal for creating software stacks in data memory.

Flexible interrupt architecture

The MAXQ10 and MAXQ20 support a single, user-configurable, interrupt-vector address register. This scheme allows placement of the interrupt identification and servicing routines according to user preference. There is no natural priority forced upon any interrupt source. In addition to the normal individual and global interrupt enables and flags, masking and identification flags are provided at the module level. The individual source enabling, module-toglobal level masking, and prioritization of interrupt sources is under the control of user code. The described interrupt support structure can be advantageous. First, no code space is left unused. This generally cannot be said for microcontrollers having dedicated interrupt-vector addresses per source, as code space associated with unused interrupt vectors is often left unused. Second, the user has increased control over which interrupts are enabled and over interrupt prioritization.

Hardware-loop counters reduce overhead

The MAXQ architecture implements a DJNZ instruction that can operate with either of two 16-bit loop counter (LC[0] or LC[1]) registers. In a single-clock cycle, the "DJNZ LC[n], src" instruction decrements the loop counter register and, if the counter has not reached 0, it conditionally branches program execution to the specified address. For competing RISC microcontrollers, updating a counter register and testing for a loop-terminating condition are generally two separate operations. Merging the two actions in the MAXQ means that software loops, commonplace in microcontroller application code, require less code and cycle overhead to manage the loop counter. The singlecycle, DJNZ-triggered loop-counter decrement and conditional branch operation exactly follow our objective of maximizing utilization of clock cycles.

Enhanced data pointers

The MAXQ comes equipped with three 16-bit data pointers (DP[0], DP[1], and BP[Offs]). All three data pointers are individually configurable for either word- or byte-access mode via the Word/Byte Select (WBSn) register bits in the Data-Pointer-Control (DPC) register. All three datamemory pointers support

single-cycle indirect memory access with pre-increment/decrement for write operations and post-increment/decrement for read operations. One of the data pointers, the Frame Pointer (FP=BP[Offs]), is generated by the unsigned additive combination of a 16-bit basepointer (BP) register and an 8-bit offset (Offs) register. This type of pointer is especially important to C compiler development tools, and more specifically, in the handling of stack frames.

Harvard memory architecture with Von Neumann benefits

The MAXQ architecture uses a Harvard memory organization, one in which the program and data memory buses are separate, so that simultaneous access to an instruction word and a data word can occur in the same clock cycle. This style of memory organization is necessary to achieve maximum performance and support single-cycle execution of instructions that access data memory. Microcontrollers that use a Von Neumann memory interface experience performance bottlenecks associated with sharing bus bandwidth among accesses to program memory, data memory, I/O, and peripherals.

Advocates of the Von Neumann memory architecture cite the inability to access program space as data memory and vice versa as a weakness. Having accessibility can simplify constant storage, look-up tables, and in-system or in-application programming alternatives. The MAXQ architectural solution to this weakness is insertion of a memory-management unit (MMU) and fixed-utility ROM that provide logical memory mappings and fixed utility-code routines to support in-system programming and the desired access modes.

Centralized access to resources

Another important feature of the MAXQ architecture is the presence of a single transfer map that contains access points to all resources. The reason for calling this a transfer map and not simply a register map is the transfer-trigger concept upon which the MAXQ architecture is based.

The transfer map is partitioned into 16 modules. Within each module are 32 indexes or individual access points. It should be emphasized, once again, that these access points can be used for direct read/write access to registers, but they may also be used for indirect access to memory or to trigger hardware operations. Of the 16 modules, the first six modules (M0–M5) are allocated for device-specific peripheral functions. This provides a generous amount of space (6 x 32 = 192 locations) in the transfer map for peripheral registers and access. These modules, based upon the specific MAXQ device option, are populated with registers to implement functions such as digital I/O, timers, serial ports, hardware multiplier, LCD driver, ADC, and in-circuit debugger. The last 10 modules (M6–M15) are reserved for MAXQ system functionality. The system modules contain registers that are vital to MAXQ system operation, such as those used for watchdog, system clock, and interrupt control. The system modules additionally contain the working accumulator file, data pointers, and source/destination encodings that trigger indirect memory access and/or special machine operations. The basic system register space is intentionally kept as common as possible among MAXQ device options. **Figure 2** presents an example MAXQ source and destination transfer map.

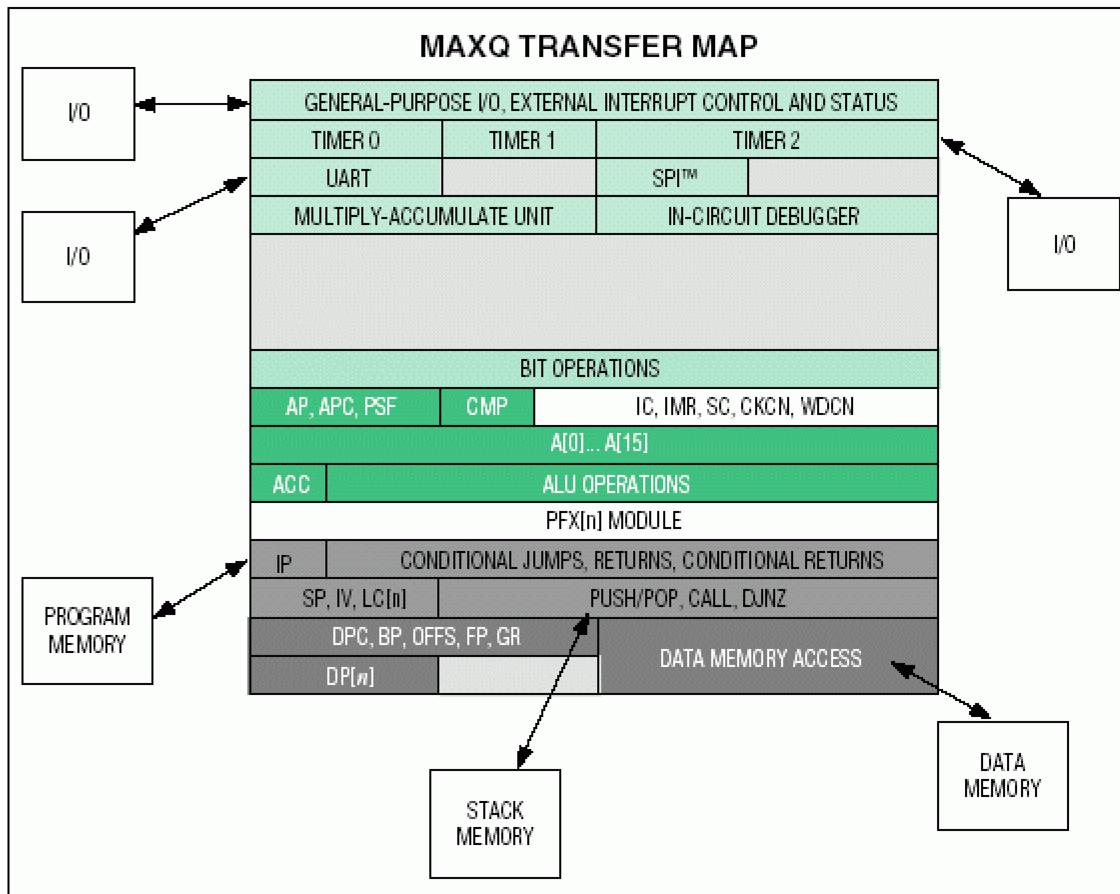

Figure 2. All MAXQ resources are accessible through a central transfer map.

The prefix register module is a feature of the MAXQ architecture that deserves special mention. There exists a single prefix register in which the data (default = 00h) is used for those transfer operations requiring it. This prefix register, when loaded, holds data for one clock cycle before being returned to a 00h state. An index (n) must accompany the prefix-register (PFX[n]) selection. Since there are 16 modules and 32 indexes per module in the transfer map, certain locations cannot be directly accessed using the source/destination encoding bits available in a single-instruction word. This is true of the latter 16 source indexes and the latter 24 destination indexes in a module. The prefix register solves this problem by opening an access window to these locations, which lasts for one cycle. When the PFX[n] register is loaded, its index "n" supplies the high-order source and destination bits to the instruction immediately following, where n = dds. In this respect, the prefix-register module is a means through which additional decoding bits can be supplied to access extended (and/or protected) registers. Operations and accesses that require loading the prefix register are automatically generated by the assembler and need not be manually coded by the user. The prefix-register module can also be used to concatenate source bytes when writing to 16-bit destinations. Although transparent to the user, the prefix register is used exactly in this fashion for jumps and calls to 16-bit absolute addresses. For those interested in future enhancements to the MAXQ architecture, the prefix-register module provides a seamless mechanism for MAXQ instruction-set expansion or extension into currently unused system-module space.

To summarize, the complete transfer map on any MAXQ device contains all system and peripheral registers defined for the device. The same map provides indirect access points to the data memory, stack memory, and accumulator array. The same map contains access points that trigger MAXQ

machine instructions and underlying operations, and a mechanism for simple extension of the instruction set in future MAXQ families. With access points to all resources aggregated into a central transfer map, the number of source-to-destination transfer opportunities is very large. The centralized access also simplifies clock distribution to only those resources needing a clock. This promotes a very quiet environment (hence the "Q" in MAXQ) that is advantageous when integrating analog peripherals. The MAXQ architecture allows maximum modularity and portability of peripheral functions. This strategy, intentionally adopted to align with rapid product development cycles and ever-changing peripheral requirements of the end user, promotes flexibility and reuse. The modularity of peripheral functions minimizes the design time required to duplicate, add, or remove standard MAXQ peripheral modules when creating new MAXQ devices for certain markets or applications.

Conclusion

The MAXQ architecture is truly an innovation in today's microcontroller industry. The MAXQ exploits a transfer-triggered architecture to achieve the objectives of high bandwidth, high efficiency, and high orthogonality. Furthermore, the modular organization of the MAXQ system and peripheral resources leads to compiler optimizations and allows portability of modules for rapid creation of new MAXQ derivatives. Looking to the future, the MAXQ architecture incorporates a built-in mechanism for instruction-set expansion suitable for next-generation products. These compelling benefits make the MAXQ architecture an ideal solution for existing and future projects, as it will inevitably rank high no matter the selection criteria for the project.

MAXQ is a registered trademark of Maxim Integrated Products, Inc.

Related Parts		
MAX1460	Low-Power, 16-Bit Smart ADC	
MAXQ2000	Low-Power LCD Microcontroller	Free Samples

More Information
For Technical Support: http://www.maximintegrated.com/support
For Samples: http://www.maximintegrated.com/samples
Other Questions and Comments: http://www.maximintegrated.com/contact

Application Note 3222: http://www.maximintegrated.com/an3222
APPLICATION NOTE 3222, AN3222, AN 3222, APP3222, Appnote3222, Appnote 3222
Copyright © by Maxim Integrated Products
Additional Legal Notices: http://www.maximintegrated.com/legal