AVR32007: UC3 C-coding Guidelines for ARM7 Developers

1 Introduction

This document describes differences between the AVR®32 UC3 and ARM7™ CPU architectures, and gives guidelines on developing applications on the AVR32 UC3 architecture for developers with ARM7 experience. The reader should refer to the AVR32 Architecture Manual, Application Note AVR32006 “Getting Started with GCC for AVR32” and datasheets for respective AVR32 microcontrollers for more information.

32-bit AVR® Microcontrollers

Application Note

Rev. 32075B-AVR-03/08

1.1 Overview of AVR32 UC3 CPU

AVR32 is a new, high-performance 32-bit RISC microprocessor architecture, designed for cost-sensitive embedded applications, with particular emphasis on low power consumption and high code-density.

Through a quantitative approach, a large set of industry-recognized benchmarks has been compiled and analyzed to achieve the best code density in its class. In addition to lowering the memory requirements, a compact code size also contributes to the core's low power characteristics. The processor supports byte and half-word data types without penalty in code size or performance.

In order to reduce code-size to a minimum, some instructions have multiple addressing modes. As an example, instructions with immediates often have a compact format with a smaller immediate, and an extended format with a larger immediate. In this way, the compiler is able to use the format giving the smallest code size.

Another feature of the instruction set is that frequently used instructions, like add, have a compact format with two operands as well as an extended format with three operands. The larger format increases performance, allowing an addition and a data move in the same instruction, in a single cycle.

Load and store instructions have several different formats in order to reduce code size and speed up execution.

The register file is organized as sixteen 32-bit registers and includes the Program Counter, the Link Register, and the Stack Pointer. In addition, register R12 is designed to hold return values from function calls and is used implicitly by some instructions.

1.2 Overview of the AVR32 MCU bus architecture

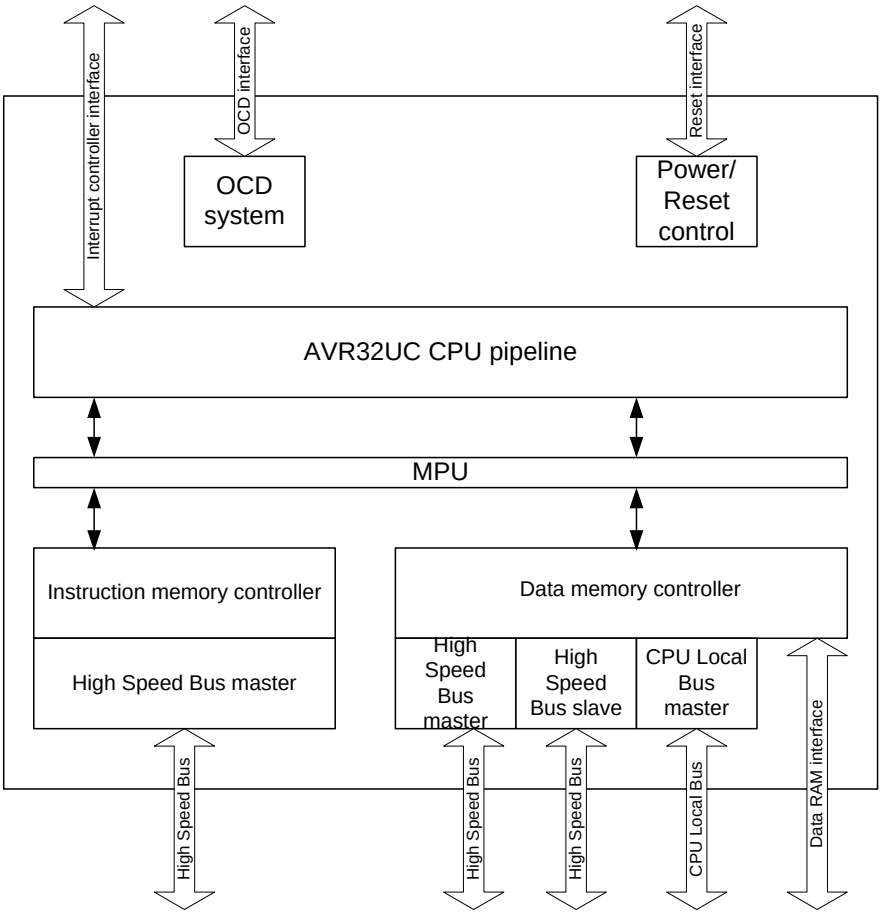
AVR32 UC3 MCUs provide an advanced OCD system, no instruction- or data-cache, and a Memory Protection Unit (MPU).

The CPU has three memory interfaces, one High Speed Bus master for instruction fetch, one High Speed Bus master for data access, and one High Speed Bus slave interface allowing other bus masters to access data RAMs internal to the CPU. Keeping data RAMs internal to the CPU allows fast access to the RAMs, reduces latency and guarantees deterministic timing.

In addition to the increased performance, power consumption is reduced by not requiring a complete High Speed Bus to access memory. A High Speed Bus slave interface is provided for other bus masters may access the internal data RAMs.

A local bus interface is provided for connecting the CPU to device-specific high-speed systems, such as floating-point units and fast GPIO ports. The local bus is able to transfer data between the CPU and the local bus slave in a single clock cycle. The local bus has a dedicated memory range allocated to it, and data transfers are performed using regular load and store instructions.

Figure 1-1. AVR32 UC3 MCU block diagram.

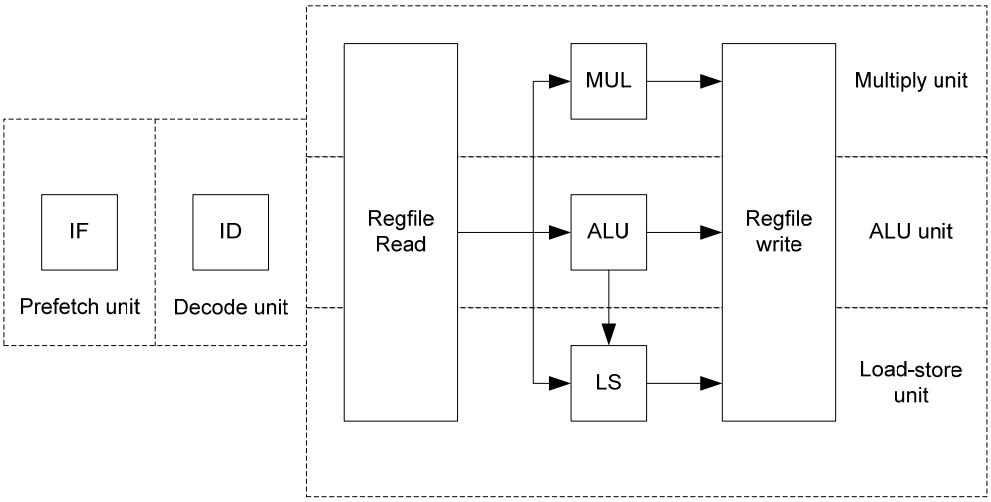


1.3 Pipeline Overview

The AVR32 UC3 CPU has three pipeline stages, Instruction Fetch (IF), Instruction Decode (ID) and Instruction Execute (EX). The EX stage are split into three parallel subsections, one arithmetic/logic (ALU) section, one multiply (MUL) section and one load/store (LS) section.

Instructions are issued and complete in order (*in-order execution)*. Certain operations require several clock cycles to complete, and in this case, the instruction resides in the ID and EX stages for the required number of clock cycles. Since there are only three pipeline stages, no internal data forwarding is required, and no data dependencies can arise in the pipeline.

Figure 1-2. AVR32 UC Pipeline.



2 Comparison between AVR32 UC3 and ARM7

The main difference between the AVR32 UC3 CPU and ARM7 is listed in the table below.

Table 2-1. Comparison between AVR32 UC3 and ARM7

Features	AVR32 UC3	ARM7
Hardware divider	Yes	No
DSP Instructions	Yes, single-cycle	No
Endianess	Big	Little
Bus architecture	Harvard	Von Neumann
Ram Location	Closely coupled to the CPU with additional bus slave interface	Ordinary bus slave
Memory Protection Unit	Yes	No
IRQ Latency	Max 16 cycles	Max 42 cycles
IRQ Levels	4+NMI	2 (IRQ and FIQ)
Autovectored IRQs	Yes	No
IRQ contexts	Stacked	Banked
Atomical bit bang support	Yes	No
Modeless instruction set	Yes	No, ARM® or Thumb mode
Memory map	Main regions defined by architecture	Unspecified
High-speed CPU local bus	Yes	No
Dhrystone MIPS/MHz	1.38	0.5 – 0.7

2.1 Digital Signal Processing

Hardware divider and DSP instructions greatly improve the computational power of AVR32 UC3 devices. The DSP instructions provide optional saturation and rounding, and are able to produce a 48-bit MAC result in a single clock cycle. This is useful in modern microcontroller applications requiring more signal processing power than traditional applications.

2.2 Endianness

AVR32 is a big-endian architecture, whereas the ARM7 is a little-endian architecture. This is usually transparent to the user, but can be an issue when porting applications from an ARM processor to an AVR32. Endianess conversion problems arise when the application accesses the same data as different data types, for example byte and word. Such applications will behave differently on a big-endian machine than on a little-endian one.

2.3 Bus architecture

AVR32 UC3 uses a Harvard bus architecture with a closely coupled RAM, whereas ARM7 uses a von Neumann bus architecture with the RAM placed on the system bus. A Harvard system is more efficient, since separate buses for instruction and data are provided. In an ARM7, a single memory interface is shared between instruction fetch and data access, causing bus contention and pipeline stalls. Furthermore, placing the RAM close to the CPU reduces memory access latency and power consumption, since the system bus does not need to be driven for ordinary memory accesses.

2.4 Memory Protection Unit

AVR32 UC3 has a Memory Protection Unit (MPU) that allows the programmer to set up different access privileges in different regions of memory. This allows the programmer to make more robust code. ARM7 has no such mechanism. The MPU can be left disabled if no special memory protection is needed.

2.5 Interrupt controller

AVR32 UC3 has a dedicated interrupt controller supporting a large number of interrupt sources. Each source is assigned a programmable priority and auto vector address. This allows the CPU to jump directly to the correct IRQ handler address, saving the cycle and code size overhead for jump tables needed by CPUs that do not support auto vectoring. The IRQ latency in AVR32 UC3 is shorter than in an ARM7. When an IRQ is received by an AVR32 UC3, registers R8-R12 and LR are automatically stored to stack so that they are available for the interrupt routine. AVR32 also provides support for more IRQ levels than ARM7, so that making time-critical systems with many interrupt sources is easier.

2.6 Atomic bit-operations

AVR32 UC3 also provides instruction for setting, clearing and toggling bits in memory atomically. Such atomic memory access instructions are not provided by the ARM7 instruction set.

2.7 16- / 32-bit execution modes,

ARM7 provides two execution modes, ARM and Thumb, each with a dedicated instruction set. ARM mode uses 32-bit instructions, and Thumb uses 16-bit instructions. Special mechanisms are used to switch between ARM and Thumb mode. ARM instructions are typically used in applications tuned for maximum performance, and Thumb code is usually used in applications tuned for minimal code size. But since Thumb mode only provides a limited subset of the ARM instruction set, applications such as DSP and floating-point libraries usually exclusively use ARM code.

In AVR32, there is only one instruction set, consisting of both 16- and 32-bit instructions that can be mixed freely without any mode-changing instructions between them. The compiler can therefore select an optimum mix of 16 and 32-bit instructions, resulting in both fast and code-compact code.

Switching between Thumb and ARM-mode in an ARM7 architecture includes a cycle-penalty, AVR32 does not have this limitation.

2.8 Memory Map

ARM7 has a flat and unstructured memory map. AVR32 UC3 separates the memory map into 4 regions with slightly different properties:

- CPU RAM
- CPU local bus
- Program memory (flash)
- High-speed system bus

Consult the AVR32 UC3 Technical Reference Manual for details

To summarize, all these differences makes the AVR32 UC3 twice as fast as an ARM7 in the commonly used Dhrystone benchmark. For real applications, in silicon with code executing from Flash, the results are often higher than x2. Read application note *AVR32006: "Getting started with GCC for AVR32"* for more information on compiler optimizations and the Dhrystone benchmark in particular.

3 Software structure

Usually, code is written in a layered fashion. A set of driver routines encapsulates and abstracts the processor hardware from the user software. As an example, a driver written in C can abstract the register interface and logic of a USART peripheral. These low-level drivers can be provided in libraries made by compiler or device vendors. The drivers can again be used by operating systems (OS) that provide another level of abstraction to the user application.

Porting applications running on an OS from ARM7 to AVR32 UC3 usually is simple: Just get the AVR32 UC3 version of the operating system and recompile the application written in C-code. The OS usually provide a hardware-independent API that the user application interfaces to. Usually, a small amount of machine-specific code must also be set up, such as PLL configuration and clock systems.

For applications not relying on an OS, the low-level device drivers must be ported when migrating from an ARM7 to an AVR32 UC3.

Atmel provides a free of charge, AVR32 Software Framework, which includes drivers for all peripherals in the UC3 devices. Refer to application note AVR32xxx “AVR32 Software Framework”.

This software framework greatly simplifies the work of porting software to AVR32, providing pre-made peripheral drivers and many assembly-tuned library routines for DSP and other applications.

3.1 Non-portable code constructs

Usually, most of an application is written in a high-level language (HLL) like C. Porting such code is usually easy, it just has to be recompiled. However, code using compiler intrinsics or pragmas, or inline assembly code may have to be rewritten when porting them from an ARM7 to an AVR32. In most cases, the mapping from ARM7 to AVR32 intrinsics or inline assembly is trivial.

3.2 Tuning code for the AVR32 UC3

Compiling HLL code written in C usually generates very good code. There are however applications where writing critical parts of the code in assembly language results in large performance gains. This chapter describes AVR32 instructions and tricks that can be used to increase the performance of your application. Some of these are exploited automatically by a compiler, others may have to be used manually by writing assembly code.

3.3 Arithmetic

AVR32 UC3 has more powerful arithmetical capabilities than an ARM7. AVR32 UC3 has a DSP instruction set that supports single-cycle multiply and multiply-accumulate with optional saturation. Saturating add and sub instructions are also provided, as well as instructions for reformatting numbers.

3.4 Digital Signal Processing

Assembly code must usually be written in order to use the DSP instructions, as they are hard for a compiler to infer from C-code.

Atmel provides a free-of-charge DSP software library in the “AVR32 UC3 Software Framework” containing a large number of hand-optimized DSP library routines like filters and FFTs that can be called directly from your C-code.

3.5 Hardware Divide

AVR32 UC3 also provides a divide instruction. This directly replaces the software divide function used in ARM7 code. ARM7 assembly code, performing software division should be rewritten to use the AVR32 hardware divider.

In ARM mode, arithmetical instructions executed by an ARM7 can use a shifter operand, i.e. one of the operands may be shifted an arbitrary amount. Such function is generally not provided in the AVR32 instruction set, therefore an ARM instruction with shifter operand may have to be replaced with one shift instruction and one arithmetical instruction in AVR32 UC3. Note that the AVR32 instruction set has a

subset of arithmetical instructions (like add and sub) that provide a limited shifter operand. This usually covers the most frequent needs for a shifter operand.

3.6 Bit manipulation

AVR32 UC3 has powerful instructions for performing bit manipulations. The bitfield instructions bfins and bfext allows extraction of a bitfield from one register and inserting it into a specified position in another register. This is very useful for example in protocol handling, where bitfields must be extracted or inserted into packets.

The swap instruction allows swapping of bytes in a register, and the set bit and clear bit in register (sbr, cbr) instructions allows setting and clearing of a bit in a register.

The bit load and bit store (bld, bst) instructions are useful for copying a bit in a register or a status register flag to a bit in another register.

An ARM7 will have to perform such bit manipulations using multiple ARM or Thumb instructions.

3.7 Data transfer

AVR32 UC3 provides the usual mix of byte, halfword and word data transfer instructions found in an ARM7. AVR32 UC3 also has instructions for load and store of multiple registers, similar to ARM7. AVR32 UC3 additionally provides a set of instructions not found in ARM7.

3.8 Endianness

Endianess-convertion instructions like load-and-swap and store-and-swap (ldswp, stswp) swaps bytes in a word before storing them to memory, or swaps them after reading them from memory. This is useful as many systems contain peripherals of different endianess.

The load-and-insert instruction (ldins) loads a byte or halfword from memory and inserts it into a bitfield in a register. This is useful for packing data for example in protocol handling.

Instructions for doubleword access, both load and store.

3.9 Conditional instructions

In the full ARM instruction set, most instructions are conditionally executed based on a condition code field in the instruction opcode. Such conditional execution is not supported when in Thumb mode.

In AVR32, the most frequently used instructions, like add, sub and, or, eor, loads and stores, have conditional variants. Conditional ARM code can usually be mapped to conditional AVR32 code with little effort. However, in some cases, a single conditional ARM instruction may have to be synthesized by two AVR32 instructions: One unconditional and one conditional one. As an example, AVR32 provides no conditional multiply instruction. A conditional ARM multiply will therefore have to be mapped to an unconditional AVR32 multiply placing the result in a dummy register, and a conditional move from the dummy register to the target register.

3.10 Atomical memory access

AVR32 UC3 provides instructions for atomically setting, clearing and toggling a bit in a memory location. No equivalent exists in the ARM7. AVR32 also provides a conditional store instruction, useful for semaphores between processes and in multiprocessor systems.

3.11 CPU local bus

AVR32 UC3 has a CPU local bus, operating at CPU speed. The local bus allows peripherals and subsystems to be closely connected to the CPU and accessed very efficiently. Which systems are placed on the local bus varies from device to device, but typically at least the General-Purpose IO (GPIO) ports are placed on the local bus. This allows the programmer to bit bang GPIO pins at the same frequency as the CPU clock.

3.12 Hardware configuration

The bus system should be set up so that the AVR32 UC3 can operate optimally. Make sure that the bus matrix is set up so that the flash controller bus slave uses the “Last Default Master” arbitration mode (set the DEFMSTR_TYPE field in the SCFG register associated with the flash controller slave to Last Default Master). This optimizes bus accesses from the instruction fetch interface to the flash program memory.

Since the CPU RAM is shared between the CPU and the system bus, arbitration hardware is implemented. CPUCR should be programmed to suit the arbitration scheme to suit different applications, balancing CPU and bus accesses to the CPU RAM.

3.13 Compiler

Make sure that the correct compiler and linker switches are used when compiling and linking the code. Compilers and toolchains are continuously changed and improved, so switches may be added between versions.

Refer to application note AVR32006 “Getting started with GCC for AVR32” for C-code, compiler and linker optimization techniques.

4 Pitfalls when porting code to AVR32 UC3

This chapter lists some of the most common pitfalls when porting code from ARM7 to AVR32 UC3.

4.1 Endianess

ARM7 is a little-endian architecture, whereas AVR32 is big-endian. This is usually transparent to the programmer, but is important when accessing the same data with accesses of different data size, e.g. byte and word. Consider storing the word 0xaabb_ccdd to address 0 in memory. The following data is returned when reading this data back as 4 bytes.

Table 4-1.

Byte load from address	AVR32 UC3	ARM7
0	0xaa	0xdd
1	0xbb	0xcc
2	0xcc	0xbb
3	0xdd	0xaa

Endianess issues can also complicate accesses to aggregate data structures as *union* and *struct* if they are accessed in creative ways.

4.2 Cycle correctness

Just to state the obvious, any ARM7 code that assumes a specific cycle behavior (such as bit banging for communication protocols) may not work directly out of the box when ported to other architectures such as AVR32 since the instruction and bus timings may be different..

5 Conclusion

Porting code from an ARM7 to an AVR32 UC3 is usually very simple. Atmel provides a complete software framework with drivers to make peripheral interfacing easy, and hand-optimized library routines to benefit from the powerful AVR32 instruction set.

Headquarters

Atmel Corporation
2325 Orchard Parkway
San Jose, CA 95131
USA
Tel: 1(408) 441-0311
Fax: 1(408) 487-2600

International

Atmel Asia
Room 1219
Chinachem Golden Plaza
77 Mody Road Tsimshatsui
East Kowloon
Hong Kong
Tel: (852) 2721-9778
Fax: (852) 2722-1369

Atmel Europe
Le Krebs
8, Rue Jean-Pierre Timbaud
BP 309
78054 Saint-Quentin-en-
Yvelines Cedex
France
Tel: (33) 1-30-60-70-00
Fax: (33) 1-30-60-71-11

Atmel Japan
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chuo-ku, Tokyo 104-0033
Japan
Tel: (81) 3-3523-3551
Fax: (81) 3-3523-7581

Product Contact

Web Site
www.atmel.com

Technical Support
avr32@atmel.com

Sales Contact
www.atmel.com/contacts

Literature Request
www.atmel.com/literature