FEATURE ARTICLE

Robert Martin

ARMs to ARMs

Part 1: Welcome to the World of ARM

Assimilating yourself to the world of ARM can be like moving to a new country: it's difficult if you're not familiar with the language and landscape. But with Robert as your guide and translator, you're sure to gain the confidence needed to use the architecture in your own projects.

Chances are you've used an ARM-based device within the last week and quite possibly the last hour. ARM CPUs have been in PDAs since the days of Newton. Currently, ARM is the only supported processor architecture for the latest version of Microsoft's PocketPC. And as I write this article, it's apparent that ARM chips will soon be appearing in PalmOS devices, having displaced the venerable 68K.

Most mobile phones use ARM cores either in ASICs or standard CPUs. They may be controlling your hard disk, printer, or network switch. ARMs were one of the root causes for the transfer of the semiconductor division of the last generation's giant to this generation's giant. ARM processors range from small, embedded microcontrollers all the way up to those intended for the multimedia market.

It's clear that many industry leaders have voted with their orders in favor of ARM. There are reasonable options for hobbyists and smaller players, too. Many ARM chips are still available with wires instead of balls (it's hard to believe that 1.0-mm pitch packages are now the convenient option). Numerous reasonably priced ARM evaluation boards are available on the market. For those of you who want to write software for higher-end embedded ARM processors but don't have an evaluation board, removing PocketPC from an iPAQ is a reasonably priced option. Compaq sponsored a successful open-source project based on the idea that embedded Linux is targeted for iPAQs.

ARM, Ltd. is an IP vendor that's licensing the architecture, processor cores, and system cores. In addition, ARM has its own development tools, including an instruction set simulator and JTAG in-circuit emulator.

TERMINOLOGY AND NOMENCLATURE

Discussing ARM products presents you with many "secret handshake" opportunities. In the ARM world, there is a differentiation between the instruction set architecture, the basic core, and the core including all of the embedded macrocells. The terminology for these items is similar enough to easily trip up the product sales representatives. So, knowing the secret handshake will show you which FAEs are really knowledgeable enough to help you.

When someone talks about the ARM architecture version, they're talking about the instruction set. Most ARM chips in embedded devices and current designs are based on either the ARM v4 or v4T architecture; however, the most advanced ARM chips from both ARM and Intel are based on the v5T architecture, which is a superset of the v4T architecture.

The "T" in the architecture's name means that the architecture in the chip supports the 16-bit Thumb instruction mode, which we'll discuss later. There is also an "E" variant of the v5 architecture for DSP extensions. You can implement one architecture in multiple core families. If you're just using an ARM processor in your design or an ARM core in your ASIC, then you will only care about the architecture version when you're poring through instruction set manuals. Companies like the former Digital Semiconductor and Intel are concerned with the architecture as they implement their own core that is compliant with a given architecture.

Discussions of architecture versions will cause many engineers' eyeballs to

glaze over as they try to use an ARM chip in a design. When the discussions move to the core that's used in the chip, the glazed-look should disappear.

The core family is the actual implementation of the architecture. The common core families from ARM are: ARM7 Thumb, ARM9 Thumb, ARM9E Thumb, and ARM10 Thumb. There are also ARM core families from Intel: the StrongARM (originally from Digital Semi) and XScale Microarchitecture.

The core that you use determines the maximum performance you'll get from the CPU. Different cores within the same architecture version may have different pipeline and data path structures. In addition, with the exception of the few cores that can be synthesized, ARM licenses the physical core. The feature sizes and voltages available were determined by ARM when the core was initially implemented.

Having all of the SoCs with the same ARM core and roughly the same core CPU performance is beneficial for those of us who consume the chips. The chip manufacturers compete with each other on peripheral sets in the microcontroller rather than with benchmark numbers that are only meaningful to the marketing department.

Most of the ARM chips on the market use a CPU core that's the combination of the basic core and embedded macrocells that provide debug (D), 64-bit multiplication results (M), DSP extensions (E), and in-circuit emulation (I). The most common system cores are the ARM7TDMI and ARM9TDMI (recall that the "T" stands for Thumb instruction mode).

You can combine a CPU core with memory system units to create a system core. These system cores are licensed by ARM and identified by a name that uses an extended core number. For example, the ARM720T core contains separate instruction and data caches, write buffers, and a memory management unit. With a better memory hierarchy, it's no surprise that CPUs with system cores tend to perform better than ones with only a CPU core.

FEATURES

With a few exceptions, the code for ARM processors is written in a higher-level language. Like all RISC machines, the architecture is well suited to higher-level languages like C. So, why should we discuss the architecture and instruction set?

First, when writing or porting code to a new embedded platform, you'll need assembly to do the initial start-up code, to set up some of the on-chip peripherals, and to take advantage of architectural features like fast interrupts. Second, many of you are bit pushers who may appreciate some of the features in the ARM instruction set.

That being said, I will limit the discussion of the architecture and instruction set to a quick overview, as well as to those things that differentiate the ARM from its competitors, in an effort to reduce the risk of losing those of you still reading this article. Note that you'll need to refer to ARM's *The ARM Architecture Reference Manual* for the actual instruction documentation.

Obviously, ARM processors are RISC machines. The only interactions with memory are to load or store registers. Arithmetic, logic, and control instructions operate only on registers. To accommodate this, a large general-purpose register set is available. Unlike other RISC processors, the ARM instruction set has instructions that take longer than one execution cycle. The instructions to load or store multiple registers may deviate from theoretical RISC design, but it does reduce code space, improve data throughput, and provide a way to perform atomic reads or writes of up to 15 32-bit words. These advantages were enough for the ARM designers to include the instructions and lose a level of RISC purity.

PROCESSOR MODES

An ARM processor has seven potential operating modes, including a standard user mode and six privileged modes that are used to handle interrupts, exceptions, or system tasks. The privileged modes are: System, Supervisor, Interrupt, Fast Interrupt, Abort, and Undefined Instruction. All of the privileged modes, with the exception of System mode, have their own stack pointer, link register, and current status register. The System mode uses User mode register set.

The Supervisor, Interrupt, Fast Interrupt, Abort, and Undefined Instruction modes are entered through hardware or software exceptions. Exceptions with their own mode and registers minimize exception-handling latency and reduce the problems of nested exceptions. You only need to save registers that you're going to use in your exception handler without concern about the stack pointer, link register, and SPSR. You must save these registers if you're going to re-enable the exception in your exception handler.

REGISTERS

The ARM architecture defines 37 registers. Of the 37 registers, 31 are

Register	APCS name	Use	Saving convention
r0	a1	Arg 1/integer result/scratch	Caller
r1	a2	Arg 2/scratch	Caller
r2	a3	Arg 3/scratch	Caller
r3	a4	Arg 4/scratch	Caller
r4	v1	Register variable 1	Callee
r5	v2	Register variable 2	Callee
r6	v3	Register variable 3	Callee
r7	v4	Register variable 4	Callee
r8	v5	Register variable 5	Callee
r9	sb/v6	Static base/register variable 6	No change/callee
r10	sl/v6	Stack limit/register variable 7	No change/callee
r11	fp	Frame pointer	
r12	ip	Scratch/new sb in interlink-unit calls	
r13	sp	Stack pointer	
r14	lr	Link register/scratch	
r15	pc	Program counter	

Table 1—*The ARM procedure calling standard (APCS) defines how registers should be used. This standard is followed by ARM higher-language compilers. Assembly language routines that interface with higher-level languages must at least follow the convention for the caller or callee saving of the registers.*

Exception	Processor mode	Vector address
Reset	Supervisor	0x00000000
Undefined instruction	Undefined	0x00000004
Software interrupt	Supervisor	0x00000008
Prefetch abort	Abort	0x0000000C
Data abort	Abort	0x00000010
Interrupt	IRQ	0x00000018
Fast interrupt	FIQ	0x0000001C

Table 2—*When one of the seven ARM exceptions occurs, the processor switches into one of the privileged modes and vectors to a well-known address. With the exception of the fast interrupts, the vector address for another exception follows in the next word. For all non-FIQ exception handlers, the instruction in the vector address must be a jump to the actual exception handler.*

general-purpose registers and six are status registers. Each program status register (PSR) contains the current processor mode, interrupt and fast interrupt enable flags, a Thumb mode flag, and conditional flags. The flags—negative, zero, carry, and overflow—are used for the conditional execution of ARM instructions.

One of the PSRs, the current program status register (CPSR), is used for the current state of the processor. The five exception processor modes each have their own saved program status register, or SPSR. These are referred to as SPSR_mode.

When the processor is in a mode other than User mode, the SPSR contains the CPSR from when the processor left the previous mode. You need to use the SPSR in some exception handlers, especially if you are using a Thumb-enabled processor.

The ARM architecture includes 31 32-bit-wide, general-purpose registers. At any time, only 15 general-purpose registers, the program counter, and CPSR are visible. The first eight registers, r0 to r7, are referred to as unbanked registers. Registers r8 through r14 are referred to as banked registers, because different registers may be visible under the same name in different processor modes.

All of the exception modes use a banked r13 and r14 for the mode stack pointer and link register. Only the FIQ mode banks registers r8 through r12. The banked registers for a given mode are designated rN_mode. When in a given mode, you'll see the registers for that mode without using the "_mode" designation. The internals of the processor use the _mode registers for processor mode swapping.

Register r15 is the program counter, and it must be treated differently than the other 15 general-purpose registers. Care must be exercised when writing to the program counter, because this will cause the processor to execute a branch to the new address. To return from a function call, you must move the return address into the program counter. The simplest way to do this is to execute the `mov pc, lr` command, which will do a register-register transfer of the return address in the link register into the program counter.

If the link register was saved on the stack as it entered into the function, then the load multiple register command can be used. For example, if you're saving registers r4 through r6 on the stack along with the link register using the `stmfd sp!, {r4-r6, lr}` command, then executing:

```
ldmfd sp!, {r4-r6,pc}
```

will restore registers r4 through r6 and cause a branch back to the return address.

ARM has defined the convention for the usage of the registers in the ARM procedure call standard (APCS). [1] This convention sets the ground rules for compiler writers and those of you writing assembly code that will interface with higher-level languages.

Because the ARM instruction set is meant to be used with higher-level languages, all of the ARM assembly language routines should at least follow the convention for which registers need to be saved by the callee routine. The register usage in APCS is shown in Table 1.

The first four arguments in a function call are placed in registers a1 to a4. APCS allows these registers to change inside the called function. The function caller should assume these values are returned in an unpredictable state. If the registers need to be preserved, they should be pushed onto the stack prior to the function call. A single-word return value comes back from the function in register a1.

The registers reserved for register variables, v1–v5/v7, must not change during a function call. The called function must save these values prior to use and restore them prior to return. Because C compiler writers follow this standard, this lets you know the maximum number of optimized variables in any section of code.

EXCEPTIONS

There are seven types of exceptions for ARM processors: reset, interrupt, fast interrupt, software interrupt, data abort, prefetch abort, and undefined instruction. When an exception condition occurs, the ARM hardware vectors to a specific address for that exception. The vector addresses are listed in Table 2.

Because there is only one machine word available for all exceptions apart from fast interrupts, the instruction at the vector address should be a jump to the actual exception handler. The

Exception	Priority	Banked registers	Link register	Return example
Reset	1	Undefined	Undefined	Undefined
Data abort	2	r13, r14, CPSR	Aborted instruction + 8	subs pc, r14, #8
Fast interrupt	3	r8–r14, CPSR	Next instruction + 4	subs pc, r14, #4
Normal interrupt	4	r13, r14, CPSR	Next instruction + 4	subs pc, r14, #4
Prefetch abort	5	r13, r14, CPSR	Aborted instruction + 8	subs pc, r14, #8
Undefined	6	r13, r14, CPSR	Next instruction	movs pc, r14
Software interrupt	6	r13, r14, CPSR	Next instruction	movs pc, r14

Table 3—*Take a look at the exception priorities, banked registers, and return mechanism for the seven ARM exceptions. The priority is used if there are multiple exceptions pending at the completion of an instruction. The undefined exception and software interrupt are the same priority because they are mutually exclusive.*

exception to this is the fast interrupt. As the highest exception vector, there is no need for a jump, and there are no restrictions on the FIQ handler starting at address 0x0000001C.

When an exception occurs, the registers that will be banked for the new mode are transferred to the appropriate _mode register. Then, the CPSR is copied into the exception mode's SPSR, normal interrupts are disabled, and the program counter is set to the exception vector.

If the exception is a reset or fast interrupt, FIQ is also disabled. Otherwise, the FIQ disable bit in the CPSR is not modified. The instruction pointed to by the banked link register is determined by the exception type. The link register for reset is undefined because it doesn't make any sense to return from a processor reset. The link registers stored for each exception are listed in Table 3. To return from the exceptions, special forms of the `mov` or `sub` commands with an "`s`" suffix are required to notify the processor that the saved registers must be restored. This is also shown in Table 3.

You can use a store multiple and load multiple combination with a couple of modifications from what is done to return from a function. First, you must perform the appropriate subtraction on the link register prior to issuing the Store Multiple command. Second, you must add a ^ suffix to notify the processor that a mode change must occur. Listing 1 shows an example.

There are two separate interrupt mechanisms: normal interrupts and the fast interrupt. There can be multiple sources of IRQs, but only one source for the FIQ. As you can tell by the name, fast interrupts are meant to occur with a minimum of interrupt handling time. The ARM architecture helps this in three ways. First, the limitation on only one source of the FIQ exception removes the need for software vectoring. Second, there is no need for an additional jump, because the FIQ handler is at the highest exception vector address. Third, the processor will bank registers r8–r14 on an FIQ exception (see Table 3). This may provide enough working registers so that stack operations are not necessary.

Normal IRQs can be caused by on-chip peripherals or interrupt requests external to the chip. Each implementation can handle multiplexing differently. Some implementations require external hardware for interrupt multiplexing, while others provide internal registers for this purpose. In general, software vectoring is required for IRQs. Even here there are potential differences between implementations. The AT91 ARM7TDMI family implements hardware IRQ vectoring through the Advanced Interrupt Controller, which is an on-chip peripheral.

The two abort exceptions are initiated by the memory system when an instruction fetch, data load, or store is requested from an invalid address. These will normally occur in systems with a memory management unit or memory protection unit. The exception handler will generally resolve the abort condition and then replay the instruction that caused the abort. In a system with an MMU, especially a virtual memory system, the abort exception handlers will be used to appropriately set up the MMU registers. The breakpoint (BKPT) instruction in v5T

Listing 1—*When entering an exception handler, you must push the registers that you will use onto the stack. When you use the store multiple register instruction for your working registers, the final register pushed on the stack should be the link register. This allows you to use one instruction to restore your working registers and to perform a return. This listing shows you how to store r0–r6 for an IRQ handler.*

```
sub   r14, r14, #4          //Set return address properly
stmfd sp!, {r0-r5,r14}      //Store r0-r5 and return address on
                              the stack
;
;Execute main part of IRQ handler
;
ldmfd sp!, {r0-r5, pc}^     //Restore registers and return
                            //Note ^ for changing modes
```

architecture is another way to generate a prefetch abort exception.

The software interrupt (SWI) is used to provide User mode access to functions that are only accessible to privileged modes. The execution of the `swi <argument>` command causes the software interrupt exception. The system will enter Supervisor mode to service the request. The exception handler uses the argument, which is encoded in the instruction, to determine the system function that's being requested. This argument is 24 bits if the SWI instruction was executed in ARM mode; it's only 8 bits in Thumb mode.

If you're implementing a system that will use Thumb instructions or that's set up for Thumb Interworking, you should consider limiting the number of system commands to 255. The exception handler will need to extract the argument from the instruction prior to the one pointed to by the link register.

The undefined instruction exception occurs when an undefined opcode is executed or a coprocessor doesn't accept a coprocessor command. This exception can be used to expand the ARM instruction set. It can be used, for instance, in a system without a coprocessor. The undefined instruction exception handler would be used to emulate coprocessor instructions.

To implement an undefined instruction handler, you have to decode the instruction prior to the one pointed to by the link register. Both the undefined instruction and software interrupt handlers require knowledge of instruction machine code encoding.

The timing of the exception handling depends on the individual exceptions. When the reset input to the processor is asserted, the processor immediately enters the reset state, interrupting the current instruction. The other one that's probably of most interest to you is the timing of FIQ and IRQ exceptions. The state of the interrupt inputs is checked at instruction boundaries. Individual instructions are not interrupted. This means that load multiple, store multiple, and coprocessor transfer instructions can delay the processor recognition of an interrupt.

For load multiple and store multiple instructions, this is up to the time required to read or write memory with 16 words or 512 bytes. The coprocessor designer determines the maximum number of bytes transferred by coprocessor transfer instructions. For this reason, ARM recommends that coprocessor designer not transfer more than 512 bytes in a single instruction.

WHAT'S NEXT?

In Part 2 of this series, I'll explain what I believe is the best reason for using ARM in your embedded designs: the ARM architectural features that reduce the code size compared to other RISC processors. In addition, I'll introduce you to the various CPU and system cores, toolchains, and OSs that are available in the market today.

Robert Martin received a Ph.D. in Physics from The College of William and Mary. He's been working with embedded and real-time systems for over 10 years. Currently, Robert is an engineering manager directing a team of embedded software engineers near Phoenix, Arizona. You may reach him at rmartin@sonoranfoothillseng.com.

REFERENCE

[1] ARM, Ltd., "The ARM-THUMB Procedure Call Standard," SWS ESPC 0002 B-01, 2000.

RESOURCES

ARM, Ltd., *ARM Architecture Reference Manual*, DDI 0100E, 2000.

S. Furber, *ARM System-On-Chip Architecture*, 2nd ed., Addison-Wesley, Harlow, England, 2000.

SOURCES

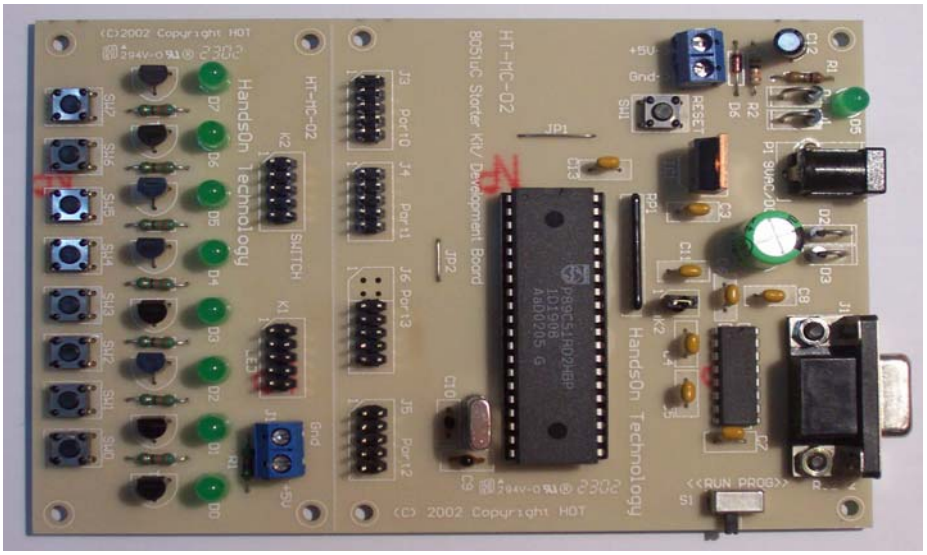
ARM7 Thumb, ARM9 Thumb, ARM9E Thumb, ARM10 Thumb
ARM, Ltd.
www.arm.com

AT91 ARM7TDMI family
Atmel Corp.
www.atmel.com

StrongARM, XScale
Intel Corp.
www.intel.com

Low Cost 8051μC Starter Kit/ Development Board *HT-MC-02*

HT-MC-02 *is an ideal platform for small to medium scale embedded systems development and quick 8051 embedded design prototyping.* ***HT-MC-02*** *can be used as stand-alone 8051μC Flash programmer or as a development, prototyping and educational platform*



Main Features:

- 8051 Central Processing Unit.
- On-chip Flash Program Memory with In-System Programming (ISP) and In Application Programming (IAP) capability.
- Boot ROM contains low level Flash programming routines for downloading code via the RS232.
- Flash memory reliably stores program code even after 10,000 erase and program cycles.
- 10-year minimum data retention.
- Programmable security for the code in the Flash. The security feature protects against software piracy and prevents the contents of the Flash from being read.
- 4 level priority interrupt & 7 interrupt sources.
- 32 general purpose I/O pins connected to 10pins header connectors for easy I/O pins access.
- Full-duplex enhanced UART **–** Framing error detection Automatic address recognition.
- Programmable Counter Array (PCA) & Pulse Width Modulation (PWM).
- Three 16-bits timer/event counters.
- AC/DC (9~12V) power supply – easily available from wall socket power adapter.
- On board stabilized +5Vdc for other external interface circuit power supply.
- Included 8x LEDs and pushbuttons test board (free with ***HT-MC-02*** while stock last) for fast simple code testing.
- Industrial popular window ***Keil*** C compiler and assembler included (Eval. version).
- Free ***Flash Magic*** Windows software for easy program code down loading.

PLEASE READ ***HT-MC-02 GETTING STARTED MANUAL*** BEFORE OPERATE THIS BOARD
INSTALL ACROBAT READER (AcrobatReader705 Application) TO OPEN AND PRINT ALL DOCUMENTS

FEATURE ARTICLE

Robert Martin

ARMs to ARMs

Part 2: Delving Deeper into the World of ARM

Now that you're familiar with the world of ARM processors, you should be thinking about how you can use the ARM architecture in your own embedded systems. In this article, Robert digs a little deeper and discusses the ARM features that will reduce code size.

Last month, I introduced you to the world of ARM, including an overview of ARM terminology and architecture. Now, we'll dive deeper into one of the selling points for using ARM in embedded systems: code space reduction compared to other RISC architectures.

As you'll see, the Thumb processor mode is beneficial for reducing code size with some loss of flexibility. Next month, I'll complete this series of articles by showing you how the architecture is implemented in CPU and system cores. I'll also talk about the tools and tool chains that are available.

CODE SPACE REDUCTION

The ARM architecture includes three additional features that work to reduce the code space required. Because code space is a concern when using RISC processors in embedded systems, it's nice to see that there is some progress being made in keeping code storage costs down. The ultimate way to reduce code space in an ARM system is to use the Thumb instruction set. I'll explain this topic later in this article.

All ARM instructions are encoded with a 4-bit conditional code. This allows for the conditional execution of every ARM instruction based on the flags in the program status register. One time, I heard an FAE give a presentation on ARM and declare that this appears to be a nice feature, but that he didn't know anyone who uses the bits. I am still trying to figure out what he was talking about, because every ARM assembly routine I've either written or seen takes advantage of the conditional execution. All ARM compilers will do the same and rely heavily on the conditional bits.

These conditional executions reduce the code space required in comparison to architectures without this feature. The conditional executions are encoded such that each condition comes in a pair with the complimentary conditional. This allows if-then-else constructs to be compiled into as little as three instructions (see Listing 1). The space efficiency provides a real advantage to using ARM processors in embedded systems and counters some of the arguments against RISC in deeply embedded environments. If you're looking for more information on RISC, you should read Jim Turley's January article, "Is RISC Good Embedded" (*Circuit Cellar* 138).

You may be concerned about the flag bits changing if you want to use the conditional bits on non-branch instructions. Take a look at the last portion of code in Listing 1. If the flag bits changed during the execution of the first `add` instruction, then you would lose the ability to perform the second `add` because of an intervening jump. Fortunately, the flag bits do not change by default on arithmetic or memory access instructions.

But, what if you wanted to check for overflow on the `add`? To do that, you must specify that you want to update the flag bits. This is accomplished by using the set flag (i.e., putting an `S` suffix on the instruction).

In this example, `add` becomes `adds` if you want the flag bits to be updated. Data comparison instructions set the flags by default; arithmetic instructions do so only with the set flag. For data movement instructions, check *The ARM Architecture Reference Manual* to see if the set flag is available.

Another feature that reduces ARM code size is the ability to use the shifter and ALU in the same instruction with the shifter result available for use by the ALU. Logical shifts left or right, arithmetic shifts left or right, and rotate left or right are available. The shift or rotate amounts can be either a literal between zero and 31 or specified in a register.

The good news is that these operations are efficient. The bad news, however, is that this efficiency allowed the ARM designers to leave out an instruction to multiply by a constant. All multiplication by constants must be done with addition, subtraction, or data movement instructions. For example, to multiply r0 by 15 and put the result in r1, the assembly is written as:

```
rsb r1, r0, r0, LSL #4
```

That is a reverse subtraction (i.e., the second operand is subtracted from the third operand with the result placed in the first operand) with a logical shift left of the third operand by four. In other words, r1 = 16 × r0 – r0. This may be good for code space, but it doesn't make writing the assembly routines that need a multiply by a constant particularly easy.

To increase the efficiency and reduce the code space of loop processing, ARM instructions have autoincrement and autodecrement addressing modes. This is performed without a penalty in execution time because of the pipeline structure of ARM cores. The incrementing can be implemented prior to or after the instruction is executed. To increment a pointer prior to the execution of a load, use the following code:

```
ldr r0, [r1, #4]!
```

First, load the value in r1+4 to r0, and then write r1+4 to r1. The `!` causes the instruction to write the literal offset back into the base register.

To apply the post-execution incrementing, use the following:

```
ldr r0, [r1], #4
```

First, load the value in r1 to r0, and then increment r1. This syntax is slightly less tricky than the pre-execution incrementing. A different register could have been used instead of the literal in both examples. To decrement, just use a negative literal or a negative value in the "increment by" register.

The ARM instruction set provides a useful set of instructions that are not within the RISC ideal but are quite useful in the real world. Some of the utility of the Load Multiple and Store Multiple instructions was demonstrated by describing the return from functions and exceptions. Because the interrupt state is polled on instruction boundaries, the `ldm` and `stm` instructions are atomic. The `ldm` and `stm` are used for stack operations, context switching, and other applications where atomically moving registers off to a contiguous block of memory is required.

The Load and Store Multiple register commands can operate in any of four different indexing modes: Full Ascending; Full Descending; Empty Ascending; and Empty Descending. Full or empty indicates whether the index register points to the last word written or the next destination address. Ascending or descending indicates whether the address in the indexing register increases or decreases with each load or store. Any of the four modes can be used. However, you must store and load using the same mechanism, otherwise you can get into some rather interesting off-by-one error scenarios.

An exclamation point on the index register causes the value in the index register to be autoincremented or autodecremented. The choice becomes easier if you're just using the `ldm` and `stm` instructions for stack manipulation and you're following the APCS. The APCS defines the stack as a full, descending stack. The stack pointer is also incremented or decremented during the operations. The com-

Listing 1—*The ARM assembly for a simple C if-then-else demonstrates the use of conditional execution of all ARM instructions. Notice that five instructions are reduced to three through the use of conditional execution on non-branch instructions. Eight bytes of code were saved and the overall complexity and maintainability of the code has been improved.*

```
//Trivial C code
if (a>b)
{
      ++a;
}
else
{
      ++b;
}

//Use conditional execution only on branch instructions
; a in reg v1
; b in reg v2
     cmp    v1, v2
     ble    INCB         //jump if a<=b
     add    v1, v1, #1   //++a
     b      DONECOND     //jump to done
INCB:
     add    v2, v2, #1 ; ++b
DONECOND:
…

//Use conditional execution on regular instructions
; a in reg v1
; b in reg v2
cmp       v1, v2
addgt     v1, v1, #1    //++a if a>b
addle     v2, v2, #1    //else ++b
…
```

mands used to push and pop registers on the stack are:

```
stmfd sp!, {<register list>}
ldmfd sp!, {<register list>}
```

The APCS convention is only for stack operations. You're free to use any indexing mode for other block transfers to and from memory.

THUMB

The ultimate way to reduce code size on a 32-bit processor is to compress the instructions down to 16 bits. In essence, this is the Thumb mode in the ARM architecture. Thumb instructions originally appeared after architecture V.4 in architecture V.4T, and they continued into V.5 of the architecture. Processors that only conform to V.4, such as the Intel StrongARM, do not support Thumb mode.

When you take a rich set of 32-bit wide instructions and compress them to 16-bits wide, you have to make certain compromises and sacrifices. Conditional execution of every ARM instruction is lost in Thumb mode. Only conditional branches use condition codes in the Thumb instruction set. This is the way most architectures operate, but it means that there may have to be more instructions used in a Thumb routine than there would be in an equivalent ARM routine.

All Thumb instructions can use registers r0 through r7, but only a limited subset can operate on registers r8 through r15. Note that the limited set includes the program counter. In the nomenclature of the Thumb instruction set, registers r0 through r7 are called the low registers. Registers r8 through r15 (PC) are referred to as the high registers. In addition to limiting the register set, immediate values encoded in the instructions are limited. Similarly, PC-relative branches are restricted to an 8-bit signed offset for conditional branches and an 11-bit offset for unconditional branches.

The Load Multiple and Store Multiple register commands are limited to the Full Descending index mode. The instructions do not use the `fd` suffix; instead, they use the non-stack alias `ia` (i.e., `ldmia` and `stmia`). These instructions operate only on the low registers. But the stack pointer is not one of the low registers. The Thumb instruction set provides `push` and `pop` instructions for saving and restoring registers on the stack. The `push` and `pop` instructions operate only the on low registers and link register.

With the compromises necessary to implement Thumb mode, the question becomes: "How is the performance?" Furber reports the following results. Thumb code will take roughly 70% of the code space of ARM code. This is more than half because Thumb code contains 40% more instructions than equivalent ARM code. In 32-bit memory systems, ARM code is faster by 40%; within 16-bit memory systems, Thumb code is 45% faster than ARM. In addition, Thumb code typically requires 30% less external memory power than ARM code. [1]

You can enter Thumb mode by executing the branch and exchange instruction, `bx`. If bit 0 of the address in the destination register is set, the processor enters Thumb mode. You do the same thing to leave Thumb mode and return to ARM mode. In that case, bit 0 of the address in the destination register will be zero. After entering Thumb mode, the T bit in the CPSR is set. Don't set the T bit in the CPSR directly; instead, you may set this bit in the SPSR and then restore the CPSR from the SPSR. This is how you should handle returns from exceptions.

Exceptions are always executed in ARM mode, even if the processor was in Thumb mode prior to the exception. Exceptions are then handled the same way they are in pure ARM mode: the processor makes appropriate adjustments to the link register when it enters the exception mode so that the return from the exception is identical for interrupted ARM code or Thumb code.

There is no need for the exception handler to determine if the size of two instructions is 4 or 8 bytes. Exception handlers that decode instructions do need to determine if the interrupted mode was Thumb or ARM. In particular, software interrupt and undefined instruction exception handlers need to pay particular attention to the T bit in the SPSR register. This allows them to know the size and format of the instruction to be decoded. If the T bit in the SPSR is set, the processor will return to Thumb mode when it exits the exception handler.

Calling ARM routines from Thumb code or calling Thumb routines from ARM code is called "interworking." When compiling a routine written in a higher-level language like C that requires interworking, the compiler and linker will insert the appropriate veneer to allow for the transition between the two modes.

For assembly language routines, you have to pay attention to getting the calls and returns correct in interworking routines. In an interworking routine, you cannot perform a branch and link call, `b`. Instead, you need to implement the following:

```
mov r0,  <subroutine address>
mov lr, pc
bx  r0
```

`pc` is the latter instruction plus eight or the instruction after `bx`. Similarly, to return from a function, you cannot implement a `mov pc, lr` instruction. Instead, you have to execute two instructions: first, move the `lr` into another register, and then implement a `bx`.

In architecture V.4T, you cannot simply use a load multiple or pop with the program counter as the destination. You need to extract the saved return address from the stack and then execute a `bx`. Version 5T of the architecture does not require the use of the `bx` instruction to change modes. In V.5T, the branch, link, and exchange instruction (`blx`) behaves in the same way as setting up the link register and then executing a `bx` does in V.4T. You can also return and exchange modes using the `ldm` or `pop` instructions.

At compile time, you need to specify that you're targeting a Thumb system and that you want interworking support. Because interworking requires some sacrifices, you should specify interworking only on those routines

that require it. It's probable that you will not want to compile in the interworking support for a routine that's called from its own mode and is either a leaf function or one that merely calls functions that are also in the same mode. Libraries can get around this by designating separate libraries for ARM mode, Thumb mode, and interworking. The linker then determines with which library to link.

This all sounds excessively complicated. In reality, though, things are much simpler on most Thumb systems. You'll probably use Thumb mode if you have your program stored in 16-bit memory or you're using a processor with a 16-bit data bus. You will probably just compile all of your User mode code to use Thumb instructions. Note that it's essential to have start-up code that knows how to call Thumb code and exception handlers compiled to use ARM instructions. There are situations where you will want some code compiled for Thumb and some others for ARM. In those special cases, you will have to pay attention to all of the details of interworking.

NEXT STOP

Next month, I'll explain the implementation of the architecture in CPU and system cores, tool chains, and tools. You'll see ARM processors targeted toward replacing legacy microcontrollers and those intended for high-performance multimedia systems. In addition, I'll tell you about tool chains with prices that are acceptable to the hobbyist, as well as some that are priced at levels that make even large corporations uncomfortable. These topics will close out my introduction to the world of ARM processors.

Robert Martin received a Ph.D. in Physics from The College of William and Mary. He's been working with embedded and real-time systems for over 10 years. Currently, Robert is an engineering manager directing a team of embedded software engineers near Phoenix, Arizona. You may reach him at rmartin@sonoranfoothillseng.com.

REFERENCE

[1] S. Furber, *ARM System-On-Chip Architecture*, 2d ed., Addison-Wesley, Harlow, England, 2000.

RESOURCE

ARM, Ltd., *ARM Architecture Reference Manual*, ARM DDI 0100E, 2000.

SOURCES

ARM7 Thumb, ARM9 Thumb, ARM9E Thumb, ARM10 Thumb
ARM, Ltd.
+44 01223 400400
www.arm.com

StrongARM, Xscale Microarchitecture
Intel Corp.
(408) 765-8080
www.intel.com

FEATURE ARTICLE

Robert Martin

ARMs to ARMs

Part 3: Working in the World of ARM

Now that you've been fully assimilated into the world of ARM, it's time to get to work. In this final installment of Robert's series, he demonstrates how to implement the ARM architecture in CPU and system cores, tool chains, and tools. Now, you don't have to rely on the x86 or 68K.

In my last two articles, I described the ARM architecture, instruction set, and Thumb mode, but I barely mentioned anything about its implementation. Although I find much of the implementation independence to be a nice feature of the ARM architecture, it won't help you drop a specific chip into a design. Therefore, in this final article, I'll cover the practical topics of cores, tools, and tool chains.

ARM CORES

The cores are the physical implementation of the ARM architecture. Although there were ARM cores prior to the ARM7TDMI, I'll discuss only those that you're likely to use in your embedded designs. If you're using an old Apple Newton, please use the references at the end of this article to find more information about the ARM6.

The ARM7 core is an implementation of the V.4T architecture that contains a three-stage pipeline with a single memory port without an inherent internal memory hierarchy. This core is the basis for the ARM7TDMI.

The processor core supports Thumb instructions (the "T" in the name) and 32 × 32 bit multiplication with a 64-bit result (the "M" in the name). In addition, the core has the JTAG debug module and Embedded ICE JTAG in-circuit emulator module, which provides a mechanism for hardware breakpoints and watch points. Additional coprocessors, such as a memory management unit (MMU) or memory protection unit (MPU), can be included with the ARM7TDMI core. You shouldn't assume that these coprocessors are present if they aren't specified, because the core exists without them.

The core is used mainly in low-cost, deeply embedded designs. ARM7TDMI cores tend to be in CPUs that have microcontroller-like peripheral sets, and they're used in CPUs meant for low-power designs where battery life is more important than performance. This core is not limited to deeply embedded designs. ARM7TDMI is supported in Palm OS 5, the first non-68K version of Palm OS.

The ARM720T is a CPU core instead of a processor core like the ARM7TDMI. It has an ARM7TDMI processor core. This core adds a memory hierarchy, including write buffers, an MMU, and an 8-KB, four-way, set-associative unified cache. The write buffer has a capacity of eight data words in four unique addresses. The exception vectors can be remapped to start at 0xFFFF0000. Note that this is a Windows CE operating system requirement.

The ARM720T is used in systems with resource-intensive operating systems (e.g., WindowsCE and Linux). These systems tend to be heavier than those that use a straight ARM7TDMI. You'll find that designs that require good memory-system performance but not the fastest processor on the block will do well with the ARM720T.

The ARM740T has the memory performance features of the ARM720T, but it doesn't have the MMU. Instead, the '740T has an MPU that allows you to restrict access to memory and memory-mapped I/O regions without the overhead of the MMU. Memory system safety and performance with greater predictability than an MMU system is a combination that meets the needs of many real-time and embedded systems.

In comparison to the ARM7 family, the ARM9 processor core contains var-

ious performance improvements while staying within the V.4T architecture. The core has a five-stage pipeline and separate instruction and data ports. The pipeline improvements and changes that were incorporated to increase memory bandwidth allow the ARM9TDMI core to run at a higher clock frequency than the ARM7TDMI core.

The ARM920T and ARM940T cores are analogous to the ARM720T and ARM740T cores, respectively. With read and write ports to the ARM9TDMI core, the ARM920T and ARM940T contain separate data and instruction caches rather than the unified cache in the ARM7 cores. The interface to external memory is still unified.

INTEL CORES

All of the cores that I've described so far are licensable from ARM, Ltd. There are high-end cores available from Intel where the architecture is licensed from ARM but not the core itself. Digital Semiconductor initially did this, but Intel assumed the relationship with ARM after acquiring Digital Semiconductor.

The StrongARM core (SA1), which was originally designed by Digital Semiconductor, incorporated performance improvements that were later adopted by ARM, including a five-stage pipeline. This core was developed prior to the development of the Thumb instruction set and the Debug and Embedded ICE macrocells. Essentially, this means that StrongARM processors have performance numbers at or exceeding ARM9 cores but don't have all of the features available in the ARM7TDMI. Specifically, the JTAG port on chips based on the SA1 core can be used only for boundary scan and ROM loading.

Until recently, the StrongARM was the ARM System-on-a-Chip technology of choice for embedded designs that needed processing power, memory system performance, and low-power usage. The SA1110 has been used widely in Windows CE devices, especially in high-end, PocketPC-based PDAs such as the Compaq iPAQ and now defunct HP Jordana.

The SA1110 comes in 206- and 133-MHz models. As part of its power-management features, the clock rate can be reduced from the maximum for the model.

Intel acquired the StrongARM core in its acquisition of Digital Semiconductor. The age of the design, the design decisions that limited productions to former Digital Semiconductor fabs, and the lack of a StrongARM upgrade path aided Intel in deciding to move forward with the new XScale microarchitecture. XScale is based on the ARM V.5TE architecture; therefore, it contains the Thumb instruction set and DSP functionality. This specific implementation of the ARM architecture is an Intel-only design.

XScale uses many of the latest advancements in RISC processor design to maximize performance and limit pipeline stalls and other mechanisms that increase the cycles per instruction from the theoretical minimum. The design is interesting, but it would require another article to do it justice. If you're interested in learning more about the design, you should refer to the documentation on Intel's web site.

In February 2002, Intel released the first two XScale processors, PXA250 and PXA210. The former, which is intended for high-end designs, is apparently targeted at the same market as the StrongARM. The latter has a 16-bit data bus and is intended for designs that don't require the performance of the PXA250.

The PXA210 has a maximum clock frequency of 200 MHz. According to Intel marketing material, the PXA250 product line's clock frequency will increase to 600 MHz. Extensive information is available on the Intel developer web site. Since its release, the 400-MHz version of the PXA250 has replaced the StrongARM as the processor of choice among the high-end PocketPC PDAs.

The PXA250 and PXA210 have a high level of system-level peripheral integration. Integrated LCD, PCMCIA/Compact Flash, and USB client controllers have removed the need for the companion chip that was required for the StrongARM. Both processor families have the usual array of serial controllers as well. This level of integration has become a must in the hand-held and wireless worlds.

Another requirement in those worlds is multilevel power management. The XScale processors take an

Mode	Description	Entered	Exited
Run	Standard mode. Clock frequency is selectable. All other modes are entered from Run mode; other modes must exit into Run mode.	Software	Software or power fail
Turbo	Clock speed is a multiple of the Run clock speed. Intended for times when extremely fast processing of instructions is required with few accesses of external memory. External memory accesses may cause pipeline stalls because of the high internal clock speed.	Software	Software
Idle	CPU clock is disabled. On-chip peripherals still receiving nominal clock frequency.	Software	Interrupt, generally coming from a peripheral.
Sleep	Only the real-time clock and power manager receive input strobes. SDRAM is placed in Self-Refresh mode. The internal processor state is lost.	Software or power fail	Preselected group of interrupts. System must reboot when exiting Sleep mode; essentially all CPU state information was lost.

Table 1—*Although the clock frequency of the Run and Turbo modes is selectable, it's best to change the frequency only when booting the system. Changing the frequency during normal running requires several additional steps because of the latency of frequency change and the effect it has on system stability.*

Module mode	GCC options
ARM only, no interworking	No additional options necessary, can use `-marm`
Thumb only, no interworking required	`-mthumb`
ARM with interworking	`-mthumb-interwork -marm`
Thumb with interworking	`-mthumb -mthumb-interwork`

Table 2—*A GCC version later than V.2.95 is required to have support for Thumb mode and ARM-Thumb Interworking. Note that for ARM modes,* `-marm` *can be specified on the command line but isn't necessary.*

interesting approach to power management with a four-level system (see Table 1). The addition of Turbo mode is quite telling; embedded processors have come now that they're capable of running faster than 100-MHz SDRAM.

For those of you writing code, the PXA210 and PXA250 have two advantages over the StrongARM. First, the support for Thumb mode is a nice addition to systems that use low-cost 16-bit memory. This is especially important for the PXA210 because its external memory bus is only 16-bits wide. The second advantage is expansion. Both the PXA210 and PXA250 have JTAG ports that you can use for software debugging.

TOOLS AND TOOL CHAINS

With a large number of chip offerings, ranging from low-end processors for deeply embedded work to high-end processors for demanding PDA applications, it's not surprising that there's a great selection of tools for ARM. You can find many different tool chains, JTAG emulators, ROM monitors, and evaluation boards to help you go from a concept to a deployable product.

There are numerous commercial tool chains available for ARM processors. ARM has its own set of tools, as well as its own JTAG emulator. Green Hills, Metaware, IAR, and others also have tools for ARM.

The tool chains normally include some sort of professional IDE, an assembler, C compiler, source code debugger, and possibly a simulator. Of course, these all come with a professional price tag to match. Most of the development tools have time-limited evaluation versions that are available free of charge. The evaluation versions generally give you a month or two to decide whether or not you're going to invest the money in the tool chain.

GNU TOOLS

The GNU tool chain is available for ARM; it includes the GCC compiler, the GNU binutils package, and a library package. The newlib open-source library package, currently sponsored by Red Hat, is meant for embedded applications, so it fits well. The newlib license is different from the GNU general public license (GPL); it's considered to be friendlier toward commercial projects than the GPL.

Although the commercial tool chains generally provide a warm and comfortable development environment, the GNU tool chain requires more of an investment of your time in order to be fully set up for development. I've found the GNU tools to be worth the effort, but they're truly not for everyone.

In December 2001, Bill Gatliff wrote an excellent article that provides a terrific starting point for setting up the GNU tool chain with the newlib C library ("Why Not GNU?" *Circuit Cellar* 137). If you're only using ARM code and don't want Thumb or Thumb Interworking, then you don't need to look farther than Bill's article. But, if you do want Thumb or Interworking capabilities, there are additional steps that you must take.

The 2.95.x versions of GCC don't include Thumb support. To use Thumb or Interworking, you must implement either one of the 3.x versions or a recent development snapshot. Using a recent version still will not give you interworking capability without some additional effort on your part.

First, you must take out the comments in the multilib sections for interworking in the $GCC_SOURCE/gcc/config/arm/t-arm-elf file in the GCC source distribution. Then, go through the tool chain compilation steps described in Bill's article. If you have an earlier version of newlib compiled for ARM, you'll need to recompile it with the GCC version that supports interworking. Otherwise, you will not get the multilib support in newlib.

To determine if you have the proper interworking support in a tool chain, type `gcc --print-multi-lib`. If all went well in the earlier steps, you'll see lines containing `thumb-interwork`. The additional GCC command line arguments for multilib support are shown in Table 2.

JTAG EMULATORS

One of the advantages of using one of the cores with the Debug and Embedded ICE units is the JTAG debug port. You can hook a JTAG in-circuit emulator to the processor to assist in all of the stages of development.

There are JTAG emulators available for ARM in all price ranges. The Wiggler and Raven are within the hobbyist's price range. More expensive emulators from ARM, Agilent, Green Hills, Abatron, and others are accessible using Ethernet interfaces. When buying an emulator, verify that it will work with the debugger in your tool chain.

ROM MONITORS

The ARM standard ROM monitor that's available on most development boards is the ARM Angel debugger. Angel provides a standard interface for debuggers. The debuggers in both commercial and GNU tool chains support this connection mechanism.

You can use Angel in a stand-alone mode or, in later stages of development, in a more limited role, applying the Angel library to provide start-up code, entry points, and raw device drivers. ARM provides a porting guide that will help you get Angel running on your custom hardware. [1]

Red Hat's RedBoot debug monitor is available for some ARM platforms. RedBoot is based on the eCos operating system; therefore, if eCos is available for your platform, RedBoot probably is too. RedBoot provides both a debugger interface for gdb and a command line interface for downloading and flashing applications on your board. RedBoot operates over a serial line or Ethernet.

Angel and RedBoot work well for general application debugging. One thing you need to remember if you're

using either Angel or RedBoot is that both require their own exception handlers. You will need to chain the ROM monitor's exception handlers after your own exception handlers. So, how can you step through your interrupt handlers? Unfortunately, you'll need to use either a JTAG ICE or ROMulator, or you'll have to set LEDs.

DEVELOPMENT BOARDS

There's quite a selection of development boards available from ARM and the silicon manufacturers themselves. For ARM7TDMI development aimed at deeply embedded targets with limited resources, the Atmel AT91EBxx series and ARM Evaluator-7T fit the bill. Both boards are inexpensive and contain systems appropriate for starting deeply embedded designs.

The price tag will increase substantially if you start working on an evaluation board base on an ARM720T or ARM9 family processor. These are higher performance chips meant for systems with greater resources, so these evaluation boards come with much more memory and Compact Flash than the straight ARM7TDMI boards. Most non-Intel development boards come without a screen.

Currently, the situation for finding development boards for the Intel ARM offerings is in a state of flux. The Intel StrongARM development boards are no longer available from the company. Third-party evaluation boards are available from Applied Data Systems. There is at least one XScale development board currently available from Intel.

In addition, Applied Data Systems has added an XScale development board to its product line. The boards for the Intel offerings tend to be aimed at PDAs, wireless consumer devices, Internet appliances, or set-top boxes. They usually have PDA or Internet appliance-sized LCD screens and are loaded with memory and Compact Flash. Even the form factor of the Intel Assabet StrongARM board is PDA-sized.

WHAT NOW?

Start-up code and a mixture of C and assembly code for the Atmel AT91EB40 eval board are on the *Circuit Cellar* ftp site. This code will demonstrate much of what has been discussed here. If you're new to ARM, this should provide a starting point for future work.

Next time a project comes your way that requires something more than a simple microcontroller, think about ARM instead of just reaching for an embedded x86 or 68K. There are a number of OS choices available for ARM processors, ranging from RTOSs to WindowsCE or Linux.

Robert Martin received a Ph.D. in Physics from The College of William and Mary. Currently, Robert is an engineering manager directing a team of embedded software engineers near Phoenix, Arizona. You may reach him at rmartin@sonoranfoothillseng.com.

PROJECT FILES

To download the code, go to ftp.circuitcellar.com/pub/Circuit_Cellar/2003/150/.

REFERENCE

[1] ARM, Ltd., "Application Note 54: Angel Porting Guide," ARM DAI 0054A, 1998.

RESOURCES

ARM, Ltd., "ARM Architecture Reference Manual," DDI 0100E, 1996.

eCos, sources.redhat.com/ecos/.

S. Furber, *ARM System-On-Chip Architecture,* 2nd ed., Addison-Wesley, Harlow, England, 2000.

Intel developer information, developer.intel.com.

SOURCES

Angel Debug monitor, ARM
ARM, Ltd.
www.arm.com

AT91EB40 Evaluation board
Atmel Corp.
www.atmel.com

PXA210/250, StrongARM, Xscale
Intel Corp.
www.intel.com

RedBoot Debug monitor
Red Hat, Inc.
www.redhat.com