Reed Electronics Group

MICROPROCESSOR REPORT

www.MPRonline.com

THE INSIDER'S GUIDE TO MICROPROCESSOR HARDWARE

ARM Strengthens Java Compilers

New 16-Bit Thumb-2EE Instructions Conserve System Memory

By Tom R. Halfhill {7/11/05-01}

In the 10 years since Sun Microsystems introduced Java, the dragon of slow run-time performance has pretty much been slain by better virtual machines, faster microprocessors, and extensions like ARM's Jazelle. Today, Java is successfully running on cellphones and other embedded systems.

Now, ARM is taking up another challenge: reducing code bloat when compiling Java bytecode with just-in-time (JIT) compilers or static native compilers. Either type of compilation can dramatically improve on the performance of a Java run-time interpreter, but it usually exacts a high price in memory, as the code expands to several times its original size.

ARM's solution is a new variation of Jazelle that enhances the 16/32-bit Thumb-2 instruction set t o assist compilers. Java isn't the only beneficiary. The extensions are also useful for other cross-platform programming languages that rely on run-time interpreters or virtual machines, such as Perl, Python, and Microsoft's .NET languages.

The new Jazelle extensions will appear in all of ARM's future Cortex-A embedded-processor cores and will be optional in Cortex-R processors, but they will be absent from the Cortex-M family. Cortex-A is intended for applications processors, and Cortex-R is for real-time embedded systems; both families are likely to run Java software. The smaller Cortex-M cores are for microcontrollers constrained by stringent cost and power requirements. (See *MPR 11/29/04-01*, "ARM Debuts Logical V7.") ARM plans to deliver the first Cortex-A and Cortex-R processor cores with the new extensions this fall, and the first chips should appear in 1H06.

ARM's marketing department has been busy, too, concocting a host of new brand names for the technology. The new extension package is called Jazelle RCT (Run-time Compilation Target) because it's aimed primarily at compilers. The original Jazelle extensions that ARM introduced in 2000 will henceforth be known as Jazelle DBX (Direct Bytecode eXecution) because they are designed for accelerating Java bytecode interpreters. (See *MPR 2/12/01-01*, "Java to Go: Part 1" through *MPR 6/4/01-01*, "Java to Go: Part 4.") The subset of 16-bit instructions in Jazelle RCT will be called Thumb-2EE (Execution Environment). And when an ARM processor enters the state enabling Thumb-2EE instructions, the new execution environment will be called ThumbEE.

It sounds more complicated than it really is. To put it more simply, Jazelle RCT adds 12 new 16-bit instructions to the ARMv7 architecture, and it modifies the operation of a few existing Thumb-2 instructions while the processor runs in ThumbEE state. The new Thumb-2EE instructions address some idiosyncrasies of virtual-machine languages and compilers. Although the instructions appear arcane, they can significantly reduce the size of compiled executables to conserve memory and improve overall performance.

Compilers Aren't Just Compilers Anymore

In ancient times—say, the 1960s—all compilers were static native compilers. They were design-time tools used by programmers to convert source code into native executable code that ran on only one processing architecture. To target a different architecture, programmers had to recompile the source code, creating a new executable program. Most modern compilers still work that way, but Java has popularized the concept of a run-time compiler. Although Sun didn't invent cross-platform interpreted languages or run-time

compilation, Java made those technologies popular in the mass market.

Run-time or JIT compilers generate the executable program from an intermediate language when the program launches, just in time for execution. They aren't quite the same as run-time interpreters, which typically translate the source code or an intermediate language into executable code line by line as the program runs. A run-time compiler converts all or some of the code at once and caches it in system memory for fast execution. Run-time compilers almost always work alongside a traditional interpreter, because some parts of the program aren't worth compiling. For example, Java applets contain initialization code that executes only once, when the applet launches, so there's no advantage in compiling and caching that code.

Some run-time compilers—often called dynamic adaptive compilers (DAC)—continue compiling parts of a program during execution, using information gathered at run time to focus on the most frequently used or time-consuming parts of the program. As run-time compilation technology keeps evolving, there is less and less distinction between JIT compilers and DACs, because a DAC is really just a smart JIT. All run-time compilers should be smart enough not to spend more time compiling than they save by avoiding line-by-line interpretation, so they all perform some run-time analysis. To prevent hiccups during execution, they must also be quick and dirty compilers. In contrast, a static native compiler—also known as an ahead-of-time (AOT) compiler—can spend much more time analyzing the source code and building an optimized executable file.

Most CPU architectures were created with design-time, not run-time, compilers in mind. Consequently, their instruction sets aren't optimized for JITs. In addition, the software targeted by JITs is usually written in a programming language that produces an intermediate format halfway between source code and native code. Java bytecode is one example of an intermediate executable format; another is the Microsoft Intermediate Language (MSIL), the foundation of .NET languages such as Visual Basic and C#. Few CPU architectures were created with those intermediate languages in mind, either.

ARM's original Jazelle extensions, now called Jazelle DBX, were ARM's first step toward optimizing the ARM architecture for Java interpreters and JITs. Jazelle DBX consists of native ARM instructions that map more directly to Java bytecode instructions. The latest Jazelle RCT extensions add more new instructions catering to design-time and run-time compilers for Java and other high-abstraction programming languages.

Jazelle RCT addresses some other features of these programming languages as well. Languages such as Java and C# are object oriented from the ground up (unlike C++) and have many built-in safeguards against common programming errors and hacker exploits (unlike C and C++). For instance, Java blocks a program's attempt to reference memory beyond the bounds of an array, and it enforces more-rigorous type checking and type casting for variables. These safeguards require more bounds checking and exception handling, both at compile time and at run time.

Instruction	Description	Notes
New Thumb-2EE Instructions		
ENTERX	Enter ThumbEE state	Enables new Thumb-2EE instructions and disables some Thumb-2 instructions
LEAVEX	Leave ThumbEE state	Disables Thumb-2EE instructions and returns to Thumb-2 mode
CHKA Rn, Rm	Array bounds check	Throws an exception if array index is below or above the array length
HB{L} #handler	Branch to handler	Branches to a specified handler routine (up to 256); option {L} saves return address
HB{L}P #handler, #parameter	Branch to handler, pass parameter	HBP branches to specified handler routine (up to 8) while passing a 3-bit integer; HBLP branches to specified handler routine (up to 32) while passing a 5-bit integer
LDR Rd, [R9, #offset]	Load register from address	Load a low register (R0–R7) from specified register address with offset
STR Rd, [R9, #offset]	Store register at address	Store a low register (R0–R7) to specified register address with offset
LDR Rd, [R10, #offset]	Load register from address	Load a low register (R0–R7) from specified register address with offset
LDR{S}H Rd, [Rn, Rm, LSL#1]	Load register from 16-bit array	Load register from a 16-bit array without using 32-bit instr or additional shift instr
STR{S}H Rd, [Rn, Rm, LSL#1]	Store register from 16-bit array	Store register into a 16-bit array without using 32-bit instr or additional shift instr
LDR Rd, [Rn, Rm, LSL#2]	Load register from 32-bit array	Load register from a 32-bit array without using 32-bit instr or additional shift instr
STR Rd, [Rn, Rm, LSL#2]	Store register from 32-bit array	Store register into a 32-bit array without using 32-bit instr or additional shift instr
Thumb-2 Instructions Disabled or Modified in ThumbEE state		
LDMIA	Load multiple (16-bit instruction)	Disabled in ThumbEE state to free opcode space for Thumb-2EE instructions
STMIA	Store multiple (16-bit instruction)	Disabled in ThumbEE state to free opcode space for Thumb-2EE instructions
LDR Rd, [Rn, Rm]	Load register	Replaced by LDR with left-shift of Rm
STR Rd, [Rn, Rm]	Store register	Replaced by SDR with left-shift of Rm
LDR{S}H Rd, [Rn, Rm]	Load register	Replaced by LDR{S}H with left-shift of Rm
STR{S}H Rd, [Rn, Rm]	Store register	Replaced by SDR{S}H with left-shift of Rm

Table 1. Jazelle RCT extends Thumb-2 with these new Thumb-2EE instructions in the ARMv7 architecture. All Thumb-2EE instructions are 16 bits long. Thumb-2EE instructions can execute only when a program enters the new ThumbEE state with the ENTERX instruction. LEAVEX exits ThumbEE. Note that a few existing Thumb-2 instructions shown in this table are not available or will behave differently in ThumbEE state.

Java will always be an odd fit for traditional CPU architectures, because the bytecode is actually native code for a virtual CPU architecture not intended for inscription in silicon. Nevertheless, tweaking existing CPU architectures to make them more Java friendly has proved more successful than the numerous and mostly sorry attempts to sell Java processors that natively execute bytecode. (See *MPR 3/17/03-02*, "Octera Throws a Javalon"; *MPR 8/14/00-04*, "Imsys Hedges Bets on Java"; *MPR 8/7/00-02*, "Embedded Java Chips Get Real"; *MPR 3/27/00-04*, "JSTAR Coprocessor Accelerates Java"; *MPR 11/17/97-02*, "MicroJava Pushes Bytecode Performance"; and *MPR 4/15/96-01*, "New Embedded CPU Goes ShBoom.")

Jazelle RCT: Only a Dozen Instructions

Table 1 lists all the new Thumb-2EE instructions that Jazelle RCT adds to the ARMv7 architecture. There are only a dozen, because Jazelle RCT is a tightly focused extension package requiring only about 8,000 gates in the processor core. The table also lists some existing Thumb-2 instructions disabled or modified while the processor runs in the ThumbEE execution environment.

Because ThumbEE is a different execution environment (processor state) than Thumb, existing Thumb-2 code remains compatible and can continue using the instructions disabled or modified in ThumbEE. A program must explicitly enter and leave ThumbEE using the new ENTERX and LEAVEX instructions shown in the table. The processor distinguishes among different execution environments by reading the T and J bits in the current program status register (CPSR), as follows: setting the T-bit indicates Thumb state; setting the J-bit indicates Jazelle DBX state; setting both the T-bit and J-bit indicates ThumbEE state; and clearing both the T-bit and J-bit indicates normal 32-bit ARM state.

Jazelle RCT is subtle and highlights the idiosyncrasies of Java and similar virtual-machine platforms. Consider the new CHKA instruction, which checks whether an array index is above or below the size of the array. CHKA throws an exception if an index is out of bounds. It's a simple instruction that at first glance seems hardly worth the trouble of adding to a well-established CPU architecture like ARMv7. However, CHKA is valuable for languages like Java, which frequently uses bounds checking to trap null-pointer bugs and to prevent the buffer-overflow exploits that haunt other platforms. CHKA can replace the two or three compare and branch instructions required to check the array bounds and call the appropriate exception handler.

ARM's CHKA instruction is reminiscent of a feature in Digital Communication Technologies' (DCT) Lightfoot processor core, which is also optimized for Java. Lightfoot has an array-bounds cache that stores the base address and length of each array. If an array reference exceeds the array bounds, Lightfoot throws an exception and calls a Java security handler. Although the CHKA instruction is a different implementation of the same concept, both ARM and DCT have found ways to optimize their processors for an extremely common operation in Java and other modern programming languages. (See *MPR 1/28/02-04*, "DCT Marches Into Java Processors.")

A Utility Infielder for Catching Exceptions

Similarly, ARM's new HB{L} and HB{L}P instructions can jump to an exception handler while optionally passing a parameter. HB{L} can jump to any of 256 predefined handlers. That may seem like an excessive number of handlers, but it allows a Java program compiled with this instruction to quickly call frequently executed pieces of code, such as the routine that allocates memory for a new object. In addition, one of Java's safety features is that any method capable of throwing an exception requires an exception handler, or else the source code won't even compile. As a result, Java programs often have numerous TRY–CATCH statements that try to perform a task and catch any exceptions thrown if the attempt fails.

The remaining extensions in Jazelle RCT optimize load/store operations. These new instructions focus on Java-centric features, too. For instance, the LDR and STR instructions that access a memory address calculated from an offset to register R9 are intended primarily for manipulating the Java stack—particularly the local variables of methods. Likewise, the LDR instruction that loads from an address offset of register R10 is intended for loading method constants.

Other LDR and STR instructions allow access to elements of 16- or 32-bit arrays without using a 32-bit-long instruction or multiplying the index to reach deeply into such large arrays. All loads and stores check the base register for a null pointer and call an exception handler if that is the case. These instructions modify the behavior of some LDR and STR instructions normally available in Thumb state, as indicated in the table.

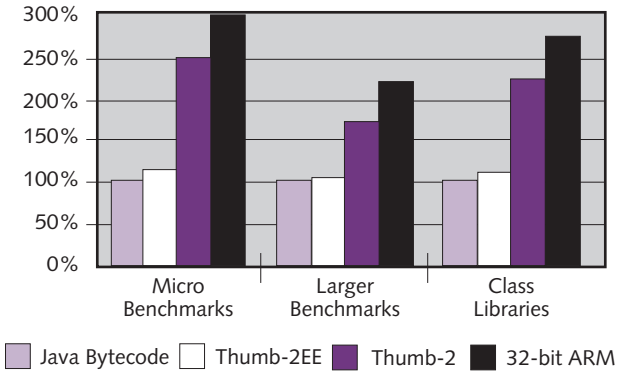

Figure 1. At best, a static native compiler enhanced with Jazelle RCT generated executable code only 7% larger than the original Java bytecode, according to these ARM internal benchmark tests. At worst, the same compiler generated code 44% larger—still impressive, considering that most static compilers inflate Java bytecode to several times its original size. Micro benchmarks: short sequences of typical Java code used by ARM's engineers to test the first implemention of a processor with Jazelle RCT. Larger benchmarks: proprietary Java benchmarks from third parties. Class libraries: Connected Limited Device Configuration (CLDC) 1.0.4 class libraries and the Mobile Information Device Profile (MIDP) 2.0 reference implementation libraries from Sun.

Price & Availability

Jazelle RCT is a new subset of the ARMv7 architecture that will be included with all ARM Cortex-A processor cores and will be optional in Cortex-R processor cores. ARM plans to release the first of those cores this fall. ARM doesn't publicly disclose licensing fees for its processors. For more information, visit this (case-sensitive) URL: *www.arm.com/products/solutions/Jazelle.html*.

When an ARM processor with Jazelle RCT enters ThumbEE state, it disables two existing Thumb-2 instructions altogether: LDMIA (load multiple) and STMIA (store multiple). These instructions load or store any subset of general-purpose registers in a single operation; the substitute is to use individual load or store instructions. ARM sacrificed LDMIA and STMIA in ThumbEE to make room in the ARMv7 opcode map for the new Jazelle RCT instructions, but they remain available in Thumb state.

Java Source	Java Bytecode	Compiled With Thumb-2	Code Size (Bytes)	
			Java	Native
X = new int[50];	bipush 50	MOV R0, #50	2	2
	newarray int	MOV.W R8,#T_INT	2	4
		BL.W DoNewArray		4
	astore 5	STR.W R0,[R9,#20]	2	4
X[index] = data	aload 5		2	
	iload 4	LDR.W R1,[R9,#16]	2	4
	iload 6	LDR.W R2,[R9,#24]	2	4
	Iastore	CMP R0,#0	1	2
		BEQ.W NullPtrHandler		4
		CMP R1,#50		2
		BHS.W ArrayIndexHandler		4
		STR.W R2,[R0,R1,LSL#2]		4
		TOTAL BYTES	**13**	**38**

Java Source	Java Bytecode	Compiled With Thumb-2EE	Code Size (Bytes)	
			Java	Native
X = new int[50];	bipush 50	MOV R0, #50	2	2
	newarray int	HBLP.X #T_INT, #NewArray	2	2
	astore 5	STR.X R0,[R9,#20]	2	2
X[index] = data	aload 5		2	
	iload 4	LDR.X R1,[R9,#16]	2	2
	iload 6	LDR.X R2,[R9,#24]	2	2
	iastore		1	
		MOV R7,#50		2
		CHKA.X R7,R1		2
		STR.X R2,[R0,R1,LSL#2]		2
		TOTAL BYTES	**13**	**16**

Figure 2. These code snippets from a Java program demonstrate how Jazelle RCT conserves memory. The first example shows two lines of Java source code, the corresponding intermediate bytecode generated by a Java source compiler, the assembly-language code generated by an ARM bytecode-to-native compiler using Thumb-2 instructions, and the code size in bytes for the Java bytecode and compiled native code. (Mnemonics with a .W suffix are 32-bit Thumb-2 instructions.) The second example shows the same two lines of Java source code and bytecode, the assembly code generated by an ARM bytecode-to-native compiler using Thumb-2EE instructions, and the code size in bytes for the Java bytecode and compiled native code. (Mnemonics with a .X suffix are Thumb-2EE instructions.) In the first example, the native code is nearly three times the size of the bytecode. In the second example, the native code is only three bytes larger. However, note that ARM simplified this example for clarity, so it may not be typical of real compiler output. In particular, register usage is the same in both code snippets, whereas a real compiler might use the registers more efficiently with Thumb-2.

Evaluating the Efficiency of Jazelle RCT

ARM says a Java compiler enhanced with Jazelle RCT can produce ARMv7 executable code that's within 10% of the size of the original Java bytecode. If true, that's a remarkable achievement. Compilation often bloats a Java program to several times its original size. Sun offers a dynamic adaptive compiler as part of its Connected Limited Device Configuration (CLDC) HotSpot Implementation, and code expansion with this compiler is typically about 6×.

One reason for this expansion is that Java wasn't designed for static compilation. Sun designed Java for native translation at run time and optimized the bytecode for compactness so Java applets could travel rapidly over large networks. The ARM architecture lends itself to generating compact code, too, especially when using Thumb, so Jazelle RCT makes for a happy marriage.

To back up its claims of minimal code expansion, ARM used its own Java compiler (not a commercially available product) to compile some internal benchmark code. ARM notes that this static native compiler is a research tool optimized for high code density, not high throughput. Figure 1 compares the code size of the original Java bytecode with the compiler's output when using standard ARM instructions, Thumb-2 instructions, and Thumb-2EE instructions (Jazelle RCT).

Of course, the results produced by different compilers will vary widely. One important factor is how aggressively the compiler optimizes the Java bytecode for performance, because speed optimizations often require a trade-off in code density. For instance, a common performance optimization is method inlining, which replicates the code of entire methods to eliminate branches to distant memory addresses. Extensive inlining quickly inflates a program to several times its original size. By using Jazelle RCT, a compiler could offer a better compromise. It could keep the program relatively small while still allowing some inlining, or it could apply aggressive inlining without bloating the program beyond the size that an unenhanced compiler would generate. To illustrate how Thumb-2EE instructions reduce code bloat, ARM provided the before-and-after code examples in Figure 2.

Effect on Throughput Is Uncertain

ARM also benchmarked Jazelle RCT to measure the effect on throughput. As Figure 3 shows, one internal benchmark test found that Java bytecode compiled with Thumb-2EE instructions ran slightly

faster than the same bytecode compiled with Thumb-2 instructions and only slightly slower than regular 32-bit ARM code. However, this test didn't include all the benchmark tests shown in Figure 1—it omits the proprietary third-party benchmarks and Java class libraries. ARM says its early pipeline simulator for a Cortex processor with Jazelle RCT isn't fast enough to boot up the operating system and Java platform required to run those system-level tests.

When Cortex processors with Jazelle RCT become available, we hope ARM runs more tests with an independent benchmark suite and a commercial Java compiler. Although ARM is a member of EEMBC, which has a Java benchmark suite, and even chairs the Java subcommittee, ARM has never published certified EEMBC benchmark scores for Java performance.

One reason for ARM's reluctance may be that almost no one has published scores for EEMBC's Java suite, so there's no meaningful context for comparison. In March 2004, EEMBC published the first benchmark results for the suite, which uses Java 2 Micro Edition (J2ME) and produces a composite score known as the GrinderMark. Sun Microsystems used the GrinderMark tests to measure the performance of two Java virtual machines on a Sharp Zaurus SL-5500 PDA, which has an Intel StrongARM processor. More than a year later, no other vendor has published GrinderMark scores. (See *MPR 8/30/04-01*, "Benchmarking the Benchmarks.")

To remedy the situation, EEMBC wants to essentially give away the GrinderMark suite for free. Currently, only paying EEMBC members enjoy access to it. EEMBC is planning to post the executable files of the GrinderMark suite (but not the Java source code) on a new website from which anyone can freely download and run the tests on a Java cellphone. Furthermore, EEMBC may relax its restrictions preventing members from publishing benchmark results for other members' products. If all those plans come to pass, it will soon be possible to benchmark Java performance on cellphones built with ARM and other processors and then publicize the scores.

Java Is Vital for ARM

Despite the lack of independent benchmark tests with a Java compiler that real-world software developers can use, *MPR* doesn't doubt that Jazelle RCT is a good addition to the ARMv7 architecture. Only the degree of improvement is in question, and even the best available benchmarks wouldn't provide a blanket answer—compilation varies too widely. We trust that ARM wouldn't have troubled to modify its architecture and processors if Jazelle RCT were valueless. Java performance is simply too important for ARM's business, which is powerfully driven by cellphones and other embedded-Java systems.

Over the past 10 years, numerous companies have tried to accelerate Java in numerous ways, but Jazelle RCT is the first attempt we've seen to limit the code bloat of compilation by modifying an existing microprocessor architecture. It's logical that ARM would lead the charge. ARM processors rule the worldwide cellphone market and are extremely popular in all kinds of low-power embedded systems. Having already introduced Jazelle DBX to conserve clock cycles, ARM is now introducing Jazelle RCT to conserve memory.

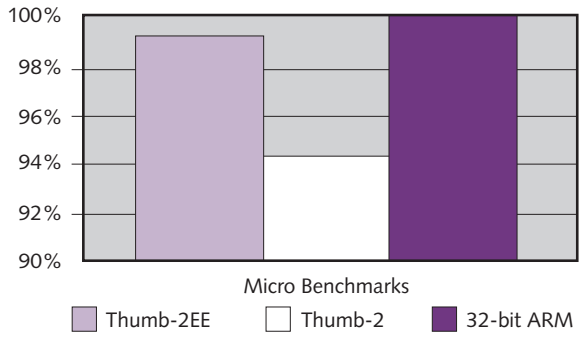

Figure 3. ARM's internal benchmarking indicates that an ahead-of-time (AOT) Java compiler can limit code bloat without significantly hampering performance. This test indicates that bytecode compiled with Thumb-2EE instructions can outperform the same code compiled with Thumb-2 instructions.

Not everyone is convinced that saving memory is worth adding gates to the processor. MIPS Technologies says memory is a "nonissue" now that Java cellphones typically have at least 2MB or 3MB. The preferred MIPS solution for boosting Java performance is Esmertec's JBed dynamic adaptive compiler, which needs only 235KB. MIPS argues that graphics performance on cellphones with color screens is becoming a larger issue than Java performance or system memory. There is some truth in what MIPS says, but *MPR* believes that conserving memory is still important, especially given the trends in cellphone design and other consumer electronics.

We note that ARM endorses Esmertec's JBed platforms and dynamic compiler, too. In fact, ARM and Esmertec last year announced a collaborative relationship in which Esmertec will use Jazelle DBX extensions to accelerate JBed. Although ARM and Esmertec haven't announced a similar deal for Jazelle RCT yet, it's almost a sure bet that Esmertec will adopt the new extensions in a future version of JBed for ARM.

Memory will be an issue in cellphones and other small embedded-Java systems as long as designers keep cramming more and more functions into the devices. The consumer appetite for extra features seems insatiable, and users are eagerly downloading games, applications, utilities, and other software written in Java. Conserving memory with Jazelle RCT is worth the trifle of 8,000 additional gates, which require a mere speck of silicon in a deep-submicron fabrication process.