Embedded Control Problems, Thumb, and the ARM7TDMI

High-end embedded control applications such as cellular phones, disk drives, and modems demand more performance from their controllers yet still require low costs. By implementing a second, compressed instruction set, our architectural innovation Thumb reduces RISC code size, providing 32-bit RISC performance at 8-/16-bit system cost. We describe the problems of embedded control, discuss the Thumb solution and its implementation, and explore typical application areas and competitive benchmarking.

Simon Segars

Keith Clarke

Liam Goudge

Advanced RISC Machines

Designers of embedded control systems face many problems. Cost is almost always a prime concern since consumer markets such as mobile phones are fiercely competitive. In addition, ensuring speedy time to market means engineers must avoid complex design-in problems. For these reasons, 4-, 8-, and 16-bit microcontrollers still dominate the embedded control market, with over 1.2 billion 4- and 8-bit micros shipped in 1994. (Source: The Information Architects, Mar. 1995.)

However, many applications now require much more performance than simple 8- and 16-bit micros can offer due to demand for additional features in existing products such as in video recorders. Also, today's demand for portability[1] makes power consumption an important new design criteria; many 8- and 16-bit micros were not designed with power consumption in mind.

Cost is partly being reduced through integration: Yesterday's discrete solutions are today being integrated into single ASICs. This imposes a size limitation on the microprocessor, which must be small enough to fit in an ASIC and still leave room for the other system functions.

The ideal embedded control processor is thus one that is small and easy to integrate, easy to design with, features low power consumption to enhance battery life, and yet still provides enough performance for tomorrow's products.

CISCs or RISCs?

RISC processors[2] represent the way forward for embedded controllers in all of these areas, as conventional CISC processors are currently running out of performance. CISC complexity led to physically large chips that, once embedded in ASICs, left little room for anything else. Also, their high power consumption[3] reduces a product's battery life in portable applications. High levels of power consumption also lead to heat dissipation problems that require expensive packaging and heat sinks to solve.

RISC processors offer much higher levels of performance and can be made small enough to allow integration into complex ASICs. The drawback of most RISC processors, however, is code density. In many cost-critical applications, one of the most expensive system components is memory. Therefore, the less memory a program occupies, the better. CISC processors do well in this area. However since RISC processors have simple instructions, many of them may be needed to perform what may be done in a single CISC instruction.

Code inefficiency also impacts system power consumption. External memory accesses are costly in terms of power since an address bus must be driven, RAM cycled, and resulting instructions driven back to the processor. Single, complex CISC instructions may be implemented in multi-instruction RISC routines, increasing the number of memory accesses and system power consumption.

The ideal processor for embedded control is thus a small, inexpensive, simple, low-power RISC, with good code density. The current ARM7

0272-1732/95/$04.00 © 1995 IEEE

line of 32-bit RISC microprocessors meets all these criteria with the possible exception of code density. (See Furber for more information on these microprocessors.[4]) Though ARM7 products outperform other RISCs and many CISCs in this area, the designer looking to upgrade from a simple 8-bit micro often finds the extra memory expense too great. We therefore had to address code size.

Thumb solution

Our designers developed a new instruction set architecture called Thumb, which specifically addresses the code density issue. It contains a subset of 36 instruction formats drawn from the standard 32-bit ARM instruction and recoded into 16-bit-wide opcodes. A small amount of logic added to the basic ARM7DMI macrocell decompresses these 16-bit instructions to their 32-bit ARM equivalents in real time. The reconstituted 32-bit ARM instruction then executes as normal. The net result of compressing code in this way is typically a 30 percent improvement in code density over native ARM code.

The ARM7TDMI

Equipped with the Thumb instruction decompressor, the ARM7TDMI macrocell core can execute both the ARM and Thumb instruction sets while still retaining all of the processor's qualities of high performance, low power, and small die size. ARM7TDMI also contains debugging features (an EmbeddedICE macrocell containing two fully configurable watchpoint/breakpoint registers; see later debugging section) and a multiply unit producing 64-bit results for digital signal processing applications.

Instruction decompression

The macrocell's major functionality change to support the Thumb instruction set is the addition of the instruction decompressor. Each 16-bit Thumb instruction has a directly equivalent 32-bit ARM instruction that has precisely the same architectural definition. Rather than provide a completely new instruction decoder for the instruction set that would act in parallel with the existing ARM decoder, ARM7TDMI implements a serial decompressor. The decompressor performs a direct translation from the 16-bit instruction to the 32-bit ARM instruction.

Like its predecessors, the ARM7TDMI core achieves single-cycle throughput for all simple instructions using a three-stage pipeline with fetch, decode, and execution cycles. The low and high phases of the clock control each cycle so that two functions typically occur within each cycle. Since the ARM instruction decoding takes place in the second phase of the decode cycle, we could position the Thumb decompressor in the spare first phase. This ensures that the single-cycle decode stage is maintained and the maximum operating frequency has not been compromised. See Figures 1 and 2.

In Thumb state, the output of the Thumb decompressor is multiplexed onto the input of the ARM instruction decoder and internal instruction bus. Thus, the Thumb instruction expands into the equivalent ARM instruction, allowing the execution stage to continue as normal.

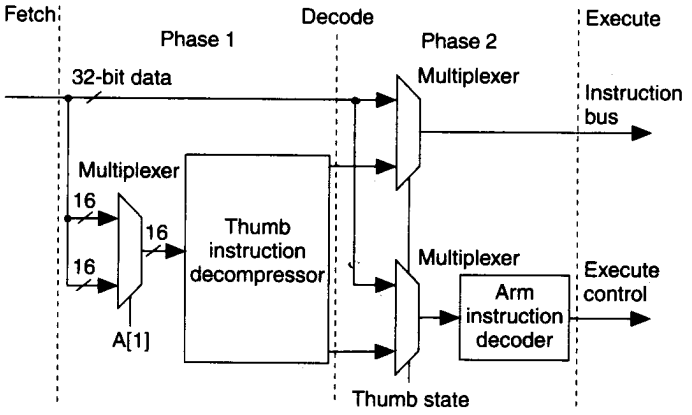

Figure 1. Instruction flow through the decode stage of the ARM7TDMI pipeline.

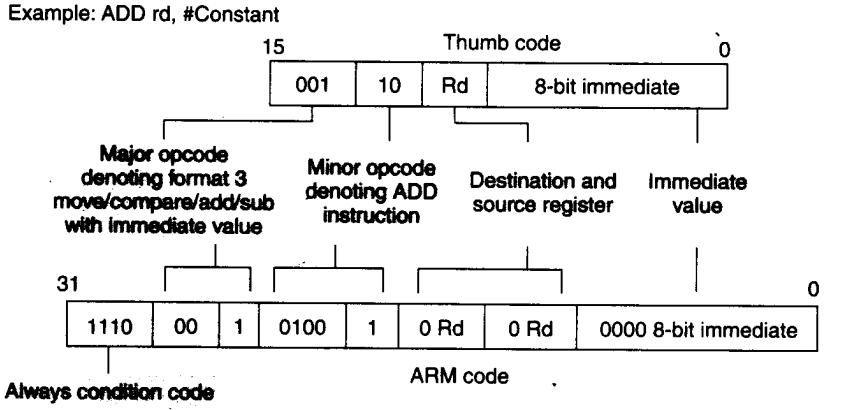

Figure 2. Expanding a Thumb ADD instruction into its ARM equivalent.

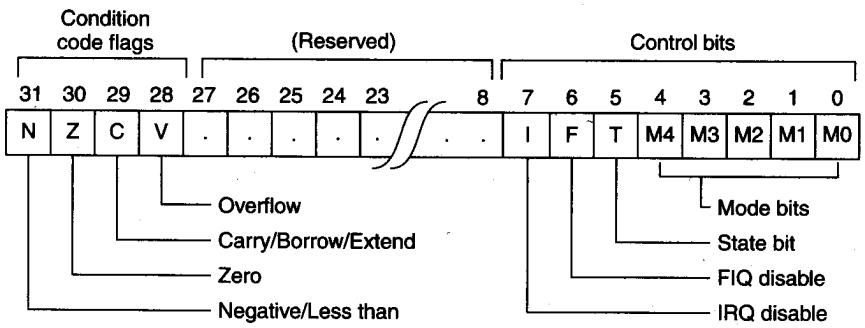

Figure 3. ARM7TDMI status register layout.

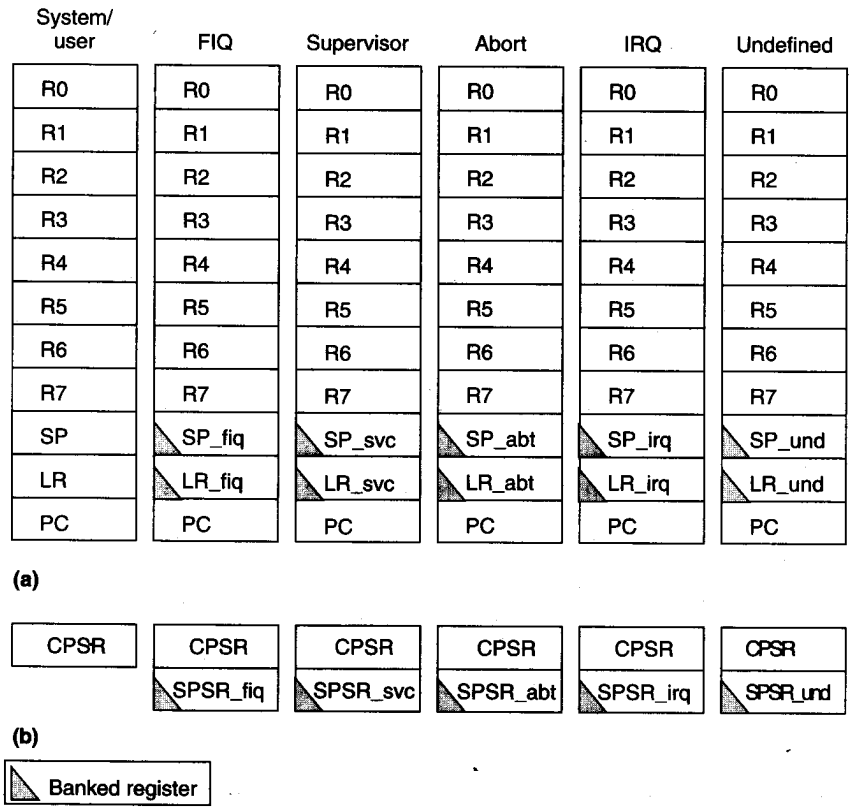

Figure 4. Programmer's model (a) with Thumb state program status registers (b).

Since only branches can be conditionally executed in the Thumb instruction set, we placed the Always condition in the expanded instruction's upper nibble. A simple lookup table produces major and minor codes, selected registers are zero extended into the 4-bit fields, the immediate value transfers, and a zero rotation value is inserted. This simplicity of decompression is key to the small size and low power consumption of the Thumb decompressor.

Switching between states

A Thumb-aware processor can execute both Thumb and original ARM instruction sets. Programmers make this choice via a state bit in the current processor status register (CPSR) known as the Tbit. Figure 3 displays the new layout of this register.

To switch between states, we added a new instruction to the ARM instruction set. This is one of the ways in which ARM7TDMI differs from mixed instruction set microprocessors. Code may be compiled as either native ARM code or Thumb code on a function-by-function basis.

In addition, since the ARM and Thumb instruction sets are separate rather than mixed, implementation is easier. We simply added the Thumb decompressor to make the ARM7 core Thumb aware; this enabled the silicon area to be kept small and hence maintains low-power and high-MIPS/W (million instructions per second/watt) performance.

Some other processors use a bit in the instruction to distinguish between instruction sets. This allows instructions within a function to be mixed between the two instruction sets but has the drawback of less space in the instruction field for the instruction itself. In a 16-bit encoding scheme, this is a serious drawback since the richness of the instruction set is limited.

Thumb instruction set

In defining the Thumb instruction set in 16 bits, we made some tradeoffs. Conditional execution is not supported since spending 25 percent of the instruction space on the condition code would not have left enough space for a useful mix of instructions. Thumb code tends, as a result, to look much more like conventional assembler with compares followed by short branches.

The other compromise made was in access to the register bank. ARM code has free access to 16 registers at once and

uses three or four operand instructions with 4-bit register specifiers. Again, to allow enough opcode space, we permitted free access to only eight registers. Instructions consist of the more conventional two or three operands and use 3-bit register specifiers. This has the hidden benefit of simplifying the migration to ARM assembler for programmers[5] accustomed to typical 8-bit assembly languages.

A typical ARM 3-operand instruction is

```
ADD  R0, R1, R2  ; R0 = R1 + R2
```

A typical Thumb 2-operand instruction is

```
ADD  R0, R1      ; R0 = R0 + R1
```

Move and add instructions permit restricted access to the rest of the registers. Figure 4 shows the programmer's model.

We limited the flexibility of other instructions (see Table 1) to their most common occurrences so they would fit into the allocated 16-bit Thumb instruction space. For example, data processing operations with one of the operands shifted are not available when in the Thumb state. To overcome this, we defined dedicated shift instructions such as

```
ARM instruction:
ADD       R0, R1, R2, LSL #3

Thumb sequence:
MOV       R0, R1
LSL       R2, R2, #3
ADD       R0, R2

Decompressed ARM instructions:
MOVS      R0, R1
MOVS      R2, R2, LSL #3
ADDS      R0, R0, R2
```

In this example, the 32-bit ARM ADD instruction, which would execute in a single cycle, has been converted into three 16-bit Thumb instructions, taking three cycles to execute. Other instructions fare much better by conversion into Thumb code. For example, MOV R0, #0 occupies 32 bits in ARM and 16 bits in Thumb and takes a single cycle to execute in both cases.

Generally, over a whole program, more Thumb instructions are required to execute a given function leading to a performance loss. However, since each Thumb instruction only occupies half the space, there is a significant net gain in code density.

What is more, since a Thumb-aware processor can execute both Thumb and ARM instruction sets, programmers can make trade-offs between performance and code density on a function-by-function basis. In typical embedded control code, one

Table 1. Thumb instruction set.

Title	Instruction	Example	ARM-code equivalent
ADC	Add with carry	ADC Rd,Rs	ADCS Rd,Rd,Rs
ADD	Add	ADD Rd,Rs,Rn	ADDS Rd,Rs,Rn
AND	AND	AND Rd,Rs	ANDS Rd,Rd,Rs
ASR	Arithmetic shift right	ASR Rd,Rs	MOVS Rd,Rd,ASR Rs
B	Unconditional branch	B Label	B Label
BCC	Conditional branch	BCC Label	BCC Label
BIC	Bit clear	BIC Rd,Rs	BICS Rd,Rd,Rs
BL	Branch and link	BL Label	BL Label
BX	Branch and exchange	BX Hs	BX Hs
CMN	Compare negative	CMN Rd,Rs	CMN Rd,Rs
CMP	Compare	CMP Rd,#offset8	CMP Rd,#offset8
EOR	EOR	EOR Rd,Rs	EORS Rd,Rd,Rs
LDMIA	Load multiple	LDMIA Rb!,[Rlist]	LDMIA Rb!,[Rlist]
LDR	Load word	LDR Rd,[PC,#Imm]	LDR Rd,[PC,#Imm]
LDRB	Load byte	LDRB Rd,[Rb,Ro]	LDRB Rd,[Rb,Ro]
LDRH	Load halfword	LDRH Rd,[Rb,#Imm]	LDRH Rd,[Rb,#Imm]
LSL	Logical shift left	LSL Rd,Rs,#offset5	MOVS Rd,Rs,LSL#offset5
LDRSB	Load sign-xtd byte	LDRSB Rd,[Rb,Ro]	LDRSB Rd,[Rb,Ro]
LDRSH	Load sign-xtd halfword	LDRSH Rd,[Rb,Ro]	LDRSH Rd,[Rb,Ro]
LSR	Logical shift right	LSR Rd,Rs	MOVS Rd,Rd,LSR Rs
MOV	Move register	MOV Rd,#offset8	MOVS Rd,#offset8
MUL	Multiply	MUL Rd,Rs	MULS Rd,Rs,Rd
MVN	Move NOT register	MVN Rd,Rs	MVNS Rd,Rs
NEG	Negate	NEG Rd,Rs	RSBS Rd,Rs,#0
ORR	OR	ORR Rd,Rs	ORRS Rd,Rd,Rs
POP	Pop registers	POP {Rlist}	LDMIA R13!,{Rlist}
PUSH	Push registers	PUSH {Rlist}	STMDB R13!,{Rlist}
ROR	Rotate right	ROR Rd,Rs	MOVS Rd,Rd,ROR Rs
SBC	Subtract with carry	SBC Rd,Rs	SBCS Rd,Rd,Rs
STMIA	Store multiple	STMIA Rb!,{Rlist}	STMIA Rb!,{Rlist}
STR	Store word	STR Rd,[Rb,Ro]	STR Rd,[Rb,Ro]
STRB	Store byte	STRB Rd,[Rb,Ro]	STRB Rd,[Rb,Ro]
STRH	Store halfword	STRH Rd,[Rb,Ro]	STRH Rd,[Rb,Ro]
SWI	Software interrupt	SWI value8	SWI value8
SUB	Subtract	SUB Rd,Rs,Rn	SUBS Rd,Rs,Rn
TST	Test bits	TST Rd,Rs	TST Rd,Rs

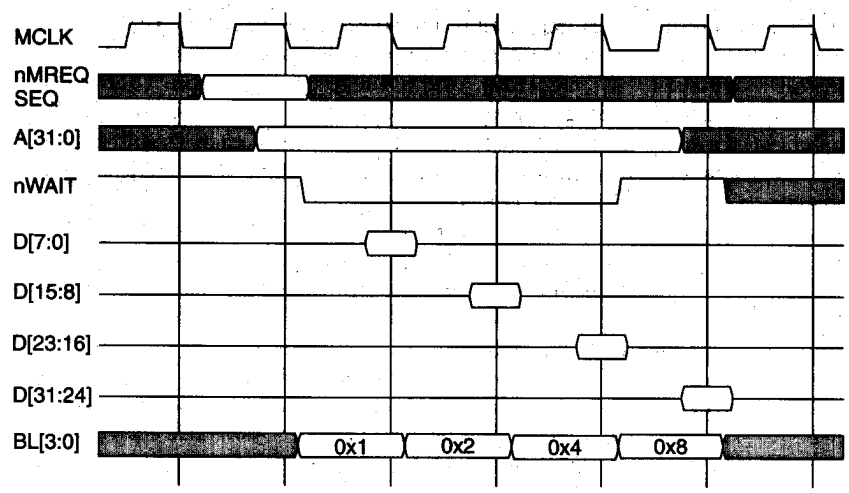

Figure 5. Word reads from 8-bit memory.

or two routines may be critical to performance. These routines should be compiled as ARM code, and the rest of the application compiled as Thumb code. The resulting code therefore takes full advantage of the code density offered by Thumb and the performance of the underlying ARM processor.

Narrow memory

An important factor in the overall cost of an embedded system is the width of the system data bus. When a processor core such as the ARM7TDMI is integrated into an ASIC, the width of the external bus may determine the number of pins on the package chosen. Using a 16-bit bus instead of a 32-bit bus may mean the difference between using an inexpensive, commonly used package and moving up to one that not only is more expensive but also requires more circuit board area.

Other benefits to using a narrower system bus can be just as important. Some types of memory devices are sold in 8- or 16-bit-wide configurations. Having to provide 32-bit-wide data to the processor directly from the memory may demand the use of two parts even though one could have sufficed. If a narrower memory is selected, a word read will require multiple read cycles and extra latches to hold the data until the final portion has been fetched, all adding to system cost.

The ARM7TDMI macrocell helps alleviate the problem of narrow memory systems by providing four signals to control directly the enables of each of the 4-byte latches, which hold data loaded from the 32-bit processor data bus. The ARM7 processor core already contained these latches, so we found it relatively simple to split each latch into four parts and allow the latch enables to be controlled externally. This reuse of existing circuitry helps maintain the very small die size of the ARM7TDMI while letting us remove circuitry that otherwise would have been required in the memory system data path.

Figure 5 shows an example of a word read from an 8-bit memory where the memory controller stalls the processor for the first three cycles using nWAIT and controls BL[3:0] to load the data byte by byte.

The ARM instruction set is much richer than the Thumb set, and so usually fewer ARM instructions are required to perform a given task. When running from ideal 32-bit memory, ARM code will almost always outperform Thumb code. However there are two circumstances in which Thumb code can outperform ARM code. These speed increases are in addition to the saving achieved in code density.

The first case is when the memory width is 16 bits or less and the code sequence is dominated by instruction fetches or data accesses that are mostly less than word width. In a 16-bit memory system, although there may be 30 to 40 percent more Thumb instructions to fetch from memory, each instruction requires half the number of memory accesses to load when compared with ARM instructions. By running benchmarks on compiled code in a 16-bit-wide memory system, we have shown that Thumb code could perform up to 50 percent faster than the equivalent ARM code.

The second situation in which Thumb code may outperform ARM code is, ironically, in a 32-bit memory system. In this case though, the benefit is only seen when the memory is slow and takes multiple wait states for each memory access. The memory controller can use ARM7TDMI's internal latches as a one-instruction prefetch buffer when Thumb instructions are being fetched, so that the second instruction does not need an extra memory access.

When the first Thumb instruction is fetched from a word-aligned address boundary, the memory access often returns the next 16-bit instruction in the other halfword of the data bus during the same cycle. The Byte Latch control signals can ensure that the core latches all 32 bits even though only 16 bits are required for this fetch. On the next cycle, the processor may require a sequential memory access. Then, the memory controller can allow the processor to continue immediately—since the data is already in the instruction latch—without performing another memory access. Figure 6 shows a typical sequence of such cycles and the use of BL[3:0] and nWAIT to control the processor timing.

If the memory takes N cycles to provide one 32-bit value, two simple Thumb instructions will have a throughput of $N + 1$ cycles, whereas two simple ARM instructions will require $2N$ cycles. The benefit will only be realized when N is large

enough to cancel out the effect of the extra Thumb instructions typically required for a task.

Software tools

Since Thumb-aware cores can execute ARM instructions, all existing ARM software continues to run on Thumb cores. However, to exploit fully the Thumb and ARM instruction sets, we significantly extended the ARM software tools. We developed a new compiler, assembler, and linker to work with the two instruction sets. These tools allow the user to compile routines as either ARM or Thumb instructions and to build an executable. This is known as interworking.

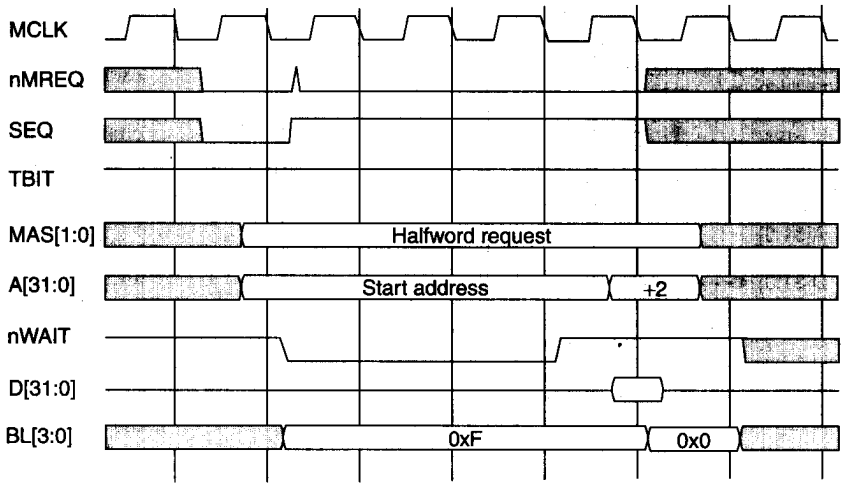

Figure 6. Typical cycle sequence using BYTE LATCH control signals.

The Thumb C compiler (tcc) compiles ANSI C to 16-bit Thumb instructions and may be used in conjunction with the standard ARM C compiler to allow code written for Thumb to call ARM code and vice versa.

The Thumb assembler can assemble either ARM or Thumb code. It allows mixing of ARM and Thumb instructions in source files via the new directives (CODE16 and CODE32), which switch between 16-bit Thumb and 32-bit ARM opcode translation.

The enhanced ARM linker supports both ARM and Thumb object types. Designers can freely mix ARM and Thumb routines in an application, allowing trade-offs of code size against performance. They can link objects across the fragmented memory maps common to many embedded applications.

Debugging support is provided by a full windowing debugger on Microsoft Windows platforms and by a command line debugger on Unix, Macintosh, and DOS. These tools provide full C-source or assembler-level debugging. ARM's debuggers can either debug code running on an instruction-accurate simulator (ARMulator) or on target hardware. Target hardware debugging support through ARM's EmbeddedICE IEEE Std 1149.1 (JTAG)[6] debugger is transparent.

ARMulator can be used to benchmark and develop code prior to the creation of target hardware. Users can configure the simulator to emulate target hardware with fragmented memory maps of differing speeds. When using the simulator in conjunction with ARM's C profiling tool, designers can choose optimal memory configurations that incorporate the three critical factors of speed, space, and memory cost.

Debugging

As systems become more complex and the level of integration increases, the problem of debugging development

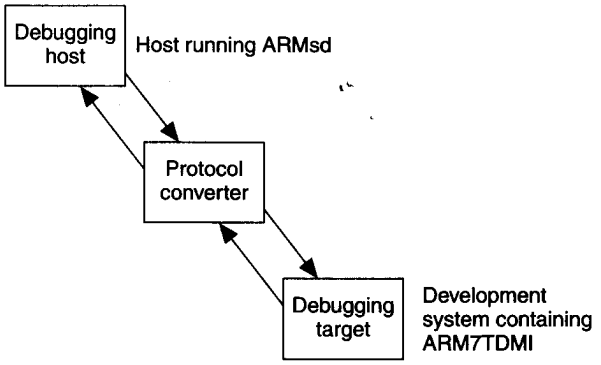

Figure 7. ARM debug route.

systems grows. To solve this problem, we introduced a combined hardware and software solution to embedded debugging. There are three components to the system: a symbolic debugger running on a debug host (such as a personal computer), a protocol converter, and a debug-compliant ARM processor. Figure 7 shows this system.

At the top of the system is the symbolic debugger, ARMsd. It allows the user to set breakpoints (on instruction fetches) and watchpoints (on data loads and stores), and examine and modify the state of the processor and memory. This is done in a high-level manner independent of the target being debugged. The target may be either a chip such as ARM7TDMI or the ARMulator software instruction emulator.

At the bottom of the system is a debugging-compliant ARM processor such as ARM7TDMI. This processor contains inte-

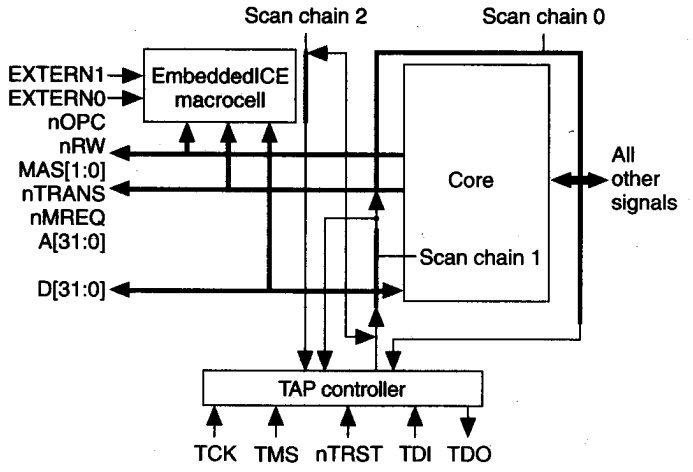

Figure 8. ARM7TDM core with Debug and EmbeddedICE macrocell extensions.

grated debugging and ICE functions, which allow the processor and memory system to be interrogated via a chip-level 1149.1 interface. In between them is a debug protocol converter. The converter takes the high-level commands issued by ARMsd (such as set a breakpoint on address X) and generates the low-level commands required to implement such a function by the target. The debugging extensions within ARM7TDMI consist of two additional units over ARM7TDM, as shown in Figure 8. These are the EmbeddedICE macrocell and a 1149.1 compliant test access port (TAP) controller.

EmbeddedICE contains a collection of breakpoint registers. These compare the value on the address, data, and control outputs against values programmed into the registers. EmbeddedICE is programmed through scan chain 2, via the 1149.1 interface. If a match occurs, EmbeddedICE generates a breakpoint signal that passes to the processor. For example, EmbeddedICE may be programmed to generate a breakpoint when, say, an instruction is loaded from a particular address or a particular data value is stored to a given location.

If the breakpoint occurs on an instruction fetch, and the instruction reaches the execute stage of the pipeline (that is, there is no preceding branch or exception), then instead of executing the instruction the processor enters the debugging state. If the breakpoint occurs on a data load or store, the processor enters the debugging state at the end of the current instruction. Once in this state, the processor state and memory system may be examined.

The processor and memory are examined through the 1149.1 interface via the TAP controller. Once the processor enters the debugging state, it stops fetching instructions from the data bus and isolates itself from the memory clock. The 1149.1 state machine now determines the action of the processor. The user may now scan instructions into the pipeline via scan chain 1 and clock the processor by moving into a particular state of the state machine. The user can scan in store instructions, which will cause the contents of the registers to be written into the scan chain and then scanned out.

Scan chain 1 is 33 bits long, 32 bits of data plus one control bit. When an instruction is scanned in and the control bit is low, the instruction executes under control of the debugging clock, and any data transfers to and from the scan chain. This is termed a debug speed instruction. When the control bit is high, before executing the instruction, the processor first synchronizes back to the memory clock, and data transfers to and from the memory system. At the end of the instruction, the processor isolates itself from the memory as before and returns to the debugging clock's control. This is termed a system speed instruction.

System speed instructions allow the user to interrogate memory from the debugging state. Typically, a load multiple instruction would be scanned into the pipeline with the control bit high. As this executes at system speed, data transfers from memory into the processor's registers. This data can then be passed back to the debugger via a debug-speed store multiple.

This operation may be repeated in reverse, also allowing the user to download code from the debugger into memory. Here, a debug-speed load multiple fills the processor's registers. This is then copied into memory via a system-speed store multiple. This feature is useful in the early stages of system software development. The basic operating system software may be downloaded, saving the inconvenience of having to program EPROMs.

The debugging extensions offered by ARM7TDMI allow system level debugging to be carried out in a system-independent manner. This means that porting the debug system is very easy since the only resource that is required on the development system is a 1149.1 connector for the protocol converter. No system memory is used, and no program ROM is required, as typically found with debugging monitors. Also, since the processor is accessed through the 1149.1 port, it is impossible for the system to hang up such that the user cannot determine the state of the processor.

Market impact

We expect to see applications in feature-hungry consumer applications where products are at the ceiling of their current 8-bit, CISC, embedded controller performance. Future product generations will need more processing power and a larger address space to accommodate new functionality, but will not be able to afford the significant system cost increase that would be associated with moving to a 32-bit system. A good example of this kind of application is the digital cellular telephone.

The next generation of phones will bring features such as an enhanced user interface and half-rate GSM (Groupe Speciale Mobile digital cellular standard) coding for improved speech quality. Ultimately, a GSM phone might even be combined with a personal digital assistant, or PDA. This will not only drive up the performance requirements for the digital signal processor and microprocessor, but will increase the size of software required and hence the amount of system memory. Current bottom-of-the-range GSM phones require 512 Kbytes of code. This could well reach 2 to 4 Mbytes in the near future.

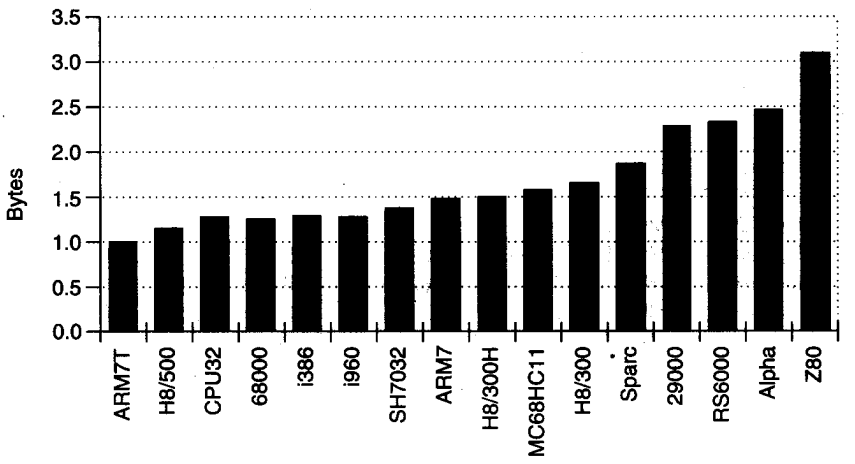

Figure 9. Normalized Dhrystone 1.1 code size for large memory model in bytes. (Source: Microprocessor Forum, 1993, and vendor data.)

Current 8-bit processors will fall short on processing power, power consumption (vitally important as it governs battery life), and addressing space. The switch to 16- or 32-bit architectures is inevitable to support the cellular telephone roadmap.[7]

ARM7TDMI solves many of these problems since the underlying 32-bit ARM7 processor provides the required performance and address space. The 16-bit-wide Thumb instruction set enables the designer to use a 16-bit-wide bus without losing performance. The traditional approach of using two 16-bit fetches for a 32-bit instruction falls over since performance is drastically reduced. As has been demonstrated, Thumb avoids the bus bottleneck by using 16-bit instructions that are decompressed to standard 32-bit ARM instructions in real time before being executed as usual on the full 32-bit ARM7 architecture.

Benchmarking tests using GSM code show that Thumb code is up to 10 percent smaller than commonly used CISC cores. Reducing code size enables either the elimination of a memory IC or the use of freed-up memory for new software features. In addition, Thumb-aware cores offer more MIPS per MHz; therefore a typical cellular phone using an 8-bit CISC clocked at +10 MHz to deliver 1 MIPS can, by using an ARM7TDMI, clock the processor at less than 1 MHz for the same performance. This reduction in clock speed brings a great saving in power consumption and also allows the use of slower logic.

Code-size benchmarking

To ensure a fair comparison, we used code size that is publicly available as Dhrystone 1.1 numbers in bytes for competing solutions and added data for the ARM7TDMI Thumb-aware core. However, this benchmark is not representative of long programs since it is less than 4 Kbytes and does not contain long branches. As code size is increased in complex applications such as PDAs, pagers, and cellular phones, 8- and 16-bit controller code size increases rapidly due to their lack of support for large code. See Figure 9.

The density of native ARM code comes close to traditional 16-bit CISC processors. This is due to novel features in the ARM instruction set such as conditional execution for every instruction and register write-back options.

With Thumb, designers who were previously considering 8- and 16-bit controllers to save system code memory can now migrate to 32-bit ARM cores and reduce the size of their system code. This will allow them either to eliminate a memory IC or use the freed memory space for new software features.

Performance benchmarks

We generated data for the ARM7TDMI core using the Thumb-aware ARMulator. This simulator provides a clock cycle count from which Dhrystone 1.1 values are calculated. Included is a cached version of the ARM7 processor for a fair comparison on power consumption. In Table 2 (next page), processors marked with an asterisk have an on-chip cache.

We also simulated performance and power consumption numbers at 3V for both the ARM7DMI and the ARM7TDMI running Dhrystones 1.1/2.1 at 20 MHz. See Tables 3 and 4.

The ARM7TDMI can execute both ARM and Thumb instructions. Therefore, in a 32-bit-wide memory system, it will deliver as many MIPS as the ARM7DMI if it runs in ARM state 100 percent of the time.

THE COST, PERFORMANCE, and power consumption of the ARM7TDMI processor makes it an excellent choice for portable applications. The ARM7TDMI has been licensed to

Table 2. Dhrystone 1.1 MIPS and MIPS per watt at 5V, 20 MHz for processors in 16-bit systems. (Source: Microprocessor Forum, 1993, and vendor data).

Processor	System	Power (W)	Dhrystone 1.1 (MIPS)	MIPS/W
ARM7TDMI	33 MHz, 5V	0.181	21.2	117
ARM7DMI	33 MHz, 5V	0.165	16.3	99
ARM710*	33 MHz, 5V	0.424	38.2	90
Z380	18 MHz	0.04	3.1	78
SH7032*	20 MHz, 5V	0.5	16.4	33
H8/500	10 MHz, 5V	0.1	1.0	10
486SLC *	33 MHz, 5V	2.25	18.0	8
H8/300H	16 MHz, 5V	0.25	1.9	8
386SLC	25 MHz, 5V	2.5	8.0	3

*Includes an on-chip cache

Table 3. ARM cores at 3V, 20 MHz in a 16-bit-wide memory system.

Processor	Benchmark	Power (W)	Dhrystone (MIPS)	MIPS/W
ARM7TDMI	Dhrystone 1.1	0.026	12.8	492
ARM7TDMI	Dhrystone 2.1	0.026	11.6	446
ARM7DMI	Dhrystone 1.1	0.033	9.9	300
ARM7DMI	Dhrystone 2.1	0.033	9.1	276

Table 4. ARM cores at 3V in a 32-bit-wide memory system.

Processor	Benchmark	Power (W)	Dhrystone (MIPS)	MIPS/W
ARM7TDMI	Dhrystone 1.1	0.026	15.6	600
ARM7TDMI	Dhrystone 2.1	0.026	14.0	538
ARM7DMI	Dhrystone 1.1	0.033	19.1	579
ARM7DMI	Dhrystone 2.1	0.033	18.0	545

several semiconductor manufacturers, and first silicon has been produced. Several embedded ASIC products featuring the ARM7TDMI will be released this year. We are now considering implementing Thumb in the other ARM processors.

References

1. T.E. Bell, "Incredible Shrinking Computers," *IEEE Spectrum*, May 1991, pp. 37-41.
2. D.A. Patterson and J.L. Henessey, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Mateo, Calif., 1990; ISBN 1-55860-188-0.
3. G.H. Forman and J. Zahorjan, "The Challenges of Mobile Computing," *Computer*, Apr. 1994, pp. 38-47.
4. S.B. Furber, *VLSI RISC Architecture and Organization*, Marcel Dekker, New York, 1989; ISBN 0-8247-8151-1.
5. A. Van Someren, *The ARM RISC Chip: A Programmer's Guide*, Addison-Wesley, Reading, Mass., 1993; ISBN 0-201-62410-9.
6. *IEEE Std 1149.1-1990, Test Access Port and Boundary-Scan Architecture*, (formerly the JTAG specification), IEEE, Piscataway, N.J.
7. S. Malhi and P. Chatterjee, "1V Microsystems-Scaling on Schedule for Personal Communications," *IEEE Circuits and Devices*, Mar. 1994.

Simon Segars works in ARM's Engineering Department as both project manager and technical leader of the Thumb project. He holds a BEng degree in electronic engineering from the University of Sussex and is currently studying part time for an MSc in low-power VLSI at the University of Manchester. He is a member of the IEEE Computer Society.

Keith Clarke has been with the Engineering Department at Advanced RISC Machines for two years, where he has held responsibility for the Thumb instruction set decoder.

Clarke graduated from Southampton University, England, with a BEng degree in electronic engineering. He is an associate member of the Institution of Electrical Engineers.

Liam Goudge is a product manager for the ARM7 product line and a segment manager for embedded control markets in ARM's Marketing Group. Previously, he was a marketer with Texas Instruments' Mixed-Signal Group in France and England.

Goudge holds a BEng degree in electronics from Nottingham University. He is a student member of the Chartered Institute of Marketing.

Direct questions concerning this article to Liam Goudge, Advanced RISC Machines, Fulbourn Road, Cambridge, England; lgoudge@armltd.co.uk.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Service Card.

Low 156 Medium 157 High 158