

Thumb: Reducing the Cost of 32-bit RISC Performance in Portable and Consumer Applications

Liam Goudge and Simon Segars

Advanced RISC Machines Inc (ARM)
email: info@arm.com

Abstract

This article discusses a RISC architectural innovation from ARM known as Thumb.

High-end embedded control applications such as cell-phones, disk drives and modems are demanding more performance from their controllers while still requiring low cost. By implementing a second "compressed" instruction set, Thumb reduces RISC code size and so provides 32-bit ARM RISC performance and power consumption at 8/16-bit system cost.

1.0 Introduction

Due to demand for additional product features, many high-volume consumer applications are now requiring more performance than 8 and 16-bit micros can offer. Portability requirements mean that power consumption has also become an important design criteria¹. Combined with system cost and time to market needs, the embedded control designer is faced with many challenges.

RISC processors² offer low on-chip power consumption³, high levels of performance and can be made small enough to allow integration within complex ASICs. However, the problem with RISC in embedded control today is code density. A processor upgrade from an 8-bit CISC may imply more memory just to implement the features of the old product and still more memory to implement the new application features that the processor upgrade allows. For cost-sensitive applications that are limited by the discrete sizes of memory systems, this is a severe drawback.

In addition, there is the need for 32-bit wide memory. A RISC architecture strives to execute one instruction per cycle and so must load one instruction every cycle to sustain the maximum execution rate. Loading from narrow 16 or 8-bit memories therefore requires extra cycles to build up the 32-bit instruction. This leads to a loss in the performance defeating the purpose of RISC in the first place.

Not only does code inefficiency increase system cost, but it also impacts system power consumption. External memory accesses demand power since an address bus must be driven, the RAM cycled and the resulting instruction driven back to the processor.

Therefore, in order to make RISC attractive to embedded designers upgrading from simple 8-bit micros, the code density issue had to be addressed.

2.0 The Thumb Solution-Compression

ARM^{4,5} needed an efficient code decompression technique that had no impact on the performance of the processor. Ideally the code compression should be to below the size of existing 8 and 16 bit code so that memory space could be liberated for the programmer to implement new features that the power of a RISC processor makes possible.

The solution was in effect to "reapply the rules of RISC" and create a "really reduced" instruction set drawn from original 32-bit ARM instructions. This second "Thumb" instruction set is a subset of the ARM instruction set encoded into 16 bits, each Thumb instruction having an exact equivalent ARM instruction (Figure 1).

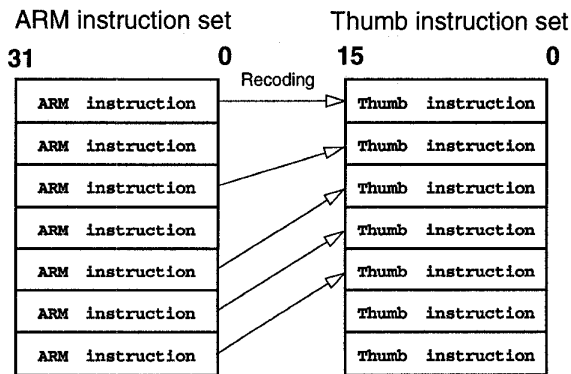


FIGURE 1. ARM to Thumb instruction set mapping

It is the fact that each Thumb instruction has an exact ARM equivalent which makes decompression simple as will be seen later.

Three types of ARM instruction were considered in the search for code compression; those which benchmarking customer code had shown to be the most frequently used and hence the most critical to shrink, those needed by the compiler to produce compact code and those that had some redundancy in the fixed length op-code.

Trade-offs were made between code size, execution speed, ease of implementation and architectural elegance. Small code was the primary goal, but not at the expense of a large loss in performance, a difficult design to implement (using too much die area) or architectural features that would not scale into the future.

The final Thumb instruction set contains a subset of 36 instruction formats drawn from the standard 32-bit ARM instruction set that have been re-coded into 16-bit wide op-codes. The net result of compressing code in this way is typically a 30% improvement in code density over native ARM code.

Due to the 16 bit instruction width limitation, not all of the features of the ARM instruction set are present in the Thumb instruction set. As a result, more 16-bit instructions than 32 bit instructions are required to execute a given function. However, as Thumb instructions are half the width of 32-bit ARM instructions, the 16 bit code is on average 35% smaller.

An example C routine demonstrates the compression effect. This routine returns the absolute value of the integer passed to it as a parameter:

```

if (X >= 0)
    return x;
else
    return -x;

```

The equivalent ARM assembly version is:

```

CMP    r0,#0;           compare r0 to zero
RSBLT  r0,r0,#0;       if r0<0 then do ro=0-r0
MOV    pc,lr;          Return

```

The Thumb version is:

```

CMP    r0,#0;           compare r0 to 0
BGE    return;         return if greater or equal
NEG    r0,r0;          if not, negate r0
MOV    pc,lr;         return

```

The ARM code requires 12 bytes (3 instructions at 4 bytes each), while the Thumb code only needs 8 bytes, a saving of 33%.

With a 32 bit wide memory system, 16 bit Thumb code has a lower performance than 32 bit ARM code because more 16-bit instructions must be executed to perform the same task. However, when using an 8 or 16 bit wide memory system, 16 bit Thumb code almost always out-performs the 32 bit code because twice as many memory accesses are required to load a 32 bit instruction than to load a 16 bit one. This is demonstrated in the example in section 5.

Since “Thumb-aware” cores are able to execute the standard ARM instruction set as well as the new Thumb instructions, the designer can therefore trade-off code size against performance, sub-routine by sub-routine, writing size-critical routines in Thumb code and performance-critical routines in ARM code.

3.0 The ARM7TDMI

The first implementation of Thumb in an ARM processor macrocell is on the ARM7TDMI, where the T indicates Thumb-awareness. The only difference between an ARM7TDMI and its predecessor, ARM7DMI, is the implementation of the Thumb “decompressor” in the instruction pipeline (shown as a shaded block in Figure 2). The ARM7 core itself is a 32-bit RISC processor noted for its small size and low power consumption while still pro-

viding RISC performance. The D and I extensions to the core allow use of JTAG debug⁶ while the M is a multiplier extension for DSP applications.

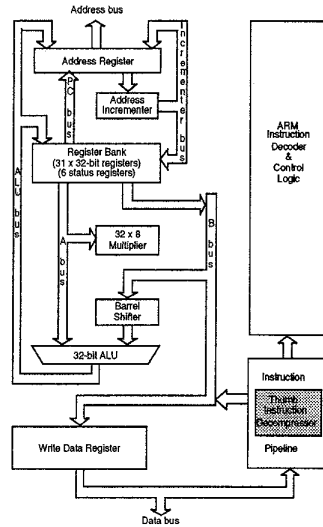


FIGURE 2. The ARM7TDMI core showing decompressor

The decompressor is a small amount of logic added to the processor macrocell pipeline which converts 16-bit Thumb instructions back to their 32-bit ARM equivalents in real time. The reconstituted 32-bit ARM instruction can then be executed as normal by the rest of the processor.

The decompressor has been designed for implementation in an unused clock phase of the pipeline so that compressed instructions can be executed at the same rate as normal instructions. This means that when executing Thumb instructions, there is no performance loss associated with decompression. In addition, since the task is simple, no extra complex circuitry is required.

Thumb-aware cores such as the ARM7TDMI still have ARM's full 32-bit architecture, so the designer retains 32-bit RISC performance and low power consumption.

3.1 Instruction decompression process

Since Thumb instructions map exactly onto an ARM equivalent, the process of decompression is simple. For example:

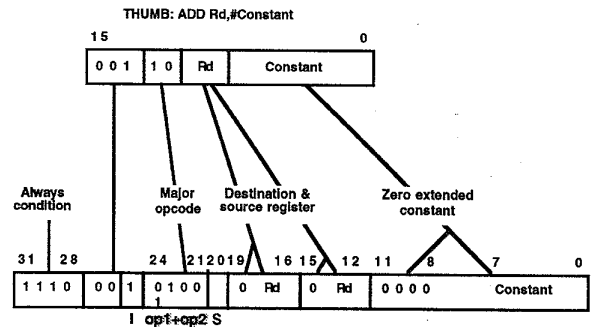


FIGURE 3. Example of instruction decompression

In this case, Figure 3 shows Thumb Add to ARM Add instruction decompression.

A feature of the ARM instruction set is the ability to specify conditional execution for each instruction. This is useful as it enables the designer to write conditional code that will not create wasteful pipeline flushes. Thumb's 16-bit op-codes did not leave space for this feature apart from in branch instructions which are useless without it. The first step in the translation process is therefore to explicitly set the condition codes of the ARM instruction to the always condition.

The major opcode of the Thumb instruction denotes the type of instruction, in this case an ALU arithmetic instruction. This is placed in the appropriate field of the ARM instruction. The minor opcode which specifies the type of ALU arithmetic operation i.e. an ADD between a register and constant is then translated via a lookup table.

The major opcode selects the operand routing from the Thumb opcode to the ARM opcode.

Since Thumb instructions have space for 3 register specifier bits as opposed to an ARM instruction's 4, the register specifiers are expanded from the Thumb opcode with zero extension to 4 bits.

The constant value is zero extended, specifying an unrotated 8 bit constant in the ARM opcode.

Rather than taking up a bit in the op-code to distinguish between instruction formats, use of a bit in the processor's status register leaves room in the 16-bit op-code for a richer instruction set; transfer between instruction sets is achieved using an instruction which toggles the state bit. This means that routines of Thumb and ARM code can

live together in the same memory space. The programmer can not interleave ARM and Thumb instructions, but can change instruction set “state” on a sub-routine by sub-routine level. The bulk of code-size critical routines can therefore be compiled for Thumb, while performance critical routines are compiled in ARM.

4.0 Software development tools

ARM has significantly enhanced the tools available to the programmer for software development in order to support the new Thumb instruction set as well as to ease the development task as a whole.

The toolkit primarily contains an ARM C compiler, assembler, linker, librarian, simulator and debugger. To this, ARM has added a new Thumb assembler and C compiler. All these components have been combined into a complete Windows Integrated Development Environment (IDE) incorporating a Project Manager, Editor, Debugger with full on-line manuals.

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 allowing code written for Thumb to call ARM code and vice-versa.

The Thumb Assembler, which can assemble either ARM or Thumb code, allows mixing of ARM and Thumb instructions in source files via new directives which switch between 16-bit Thumb and 32-bit ARM op-codes.

The Linker has also been enhanced to support both ARM and Thumb object types. ARM and Thumb routines can be freely mixed in an applications allowing the designer to trade off code-size against performance.

Debugging support is provided by a full windowing debugger on Windows platforms. These tools provide full C source or assembler level debugging and can either debug code running on an instruction accurate simulator (ARMulator) or transparently on target hardware by using JTAG protocols.

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

5.0 Low cost digital cellular phone RISC controller using 8-bit ROM

The example controller in Figure 4 integrates the Thumb aware microprocessor, a DSP and customer-specific on-chip peripherals as well as small amount of fast 32-bit ROM or RAM which is used to store speed-critical code. When the Thumb-aware ARM7TDMI core switches into ARM state for extra performance, such as in the case of an interrupt, ARM code is executed out of this area of fast memory. Slow external 8-bit ROM and RAM are used for code storage (mainly Thumb code) and scratchpad data.

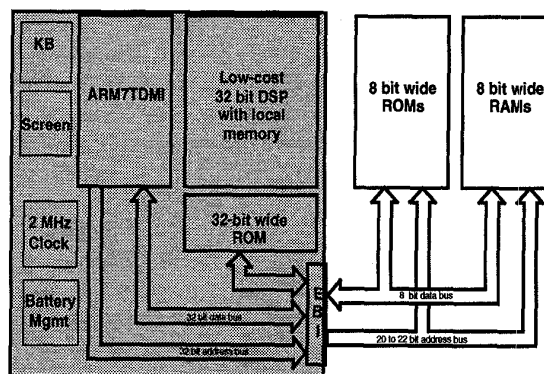


FIGURE 4. Digital cellular phone controller

This system benefits from the narrow external bus and memory that low-cost embedded applications demand. The data in Table 1 shows performance, expressed as Dhrystone 2.1 MIPS, against processor frequency for code running from 32, 16 and 8 bit wide memory. In all cases, the memory is the same speed as the processor.

Mode	Bus width	5 MHz	10 MHz	40 MHz
Thumb	8 bits	1.6	3.2	13.0
	16 bits	3.1	6.2	24.6
	32 bits	3.7	7.4	29.5
ARM	8 bits	1.2	2.4	9.7
	16 bits	2.4	4.7	18.8
	32 bits	4.4	8.8	35.3

TABLE 1. ARM7TDMI Performance with various bus widths and clock speeds

Thumb code executes at 85% of ARM performance from 32 bit memory due to the extra instructions required to complete a task. However, with narrower memories, Thumb code out-performs ARM code by 130% due to the reduction in the number of memory accesses required to fetch an instruction. Therefore in the cellular system shown above, any performance critical code should be compiled as ARM code and reside in 32-bit wide on-chip memory. The rest of the code can be compiled as Thumb code and stored in the narrow external memory for minimal system cost.

The benefits of Thumb are not just limited to code size. Low power is key in cellular telephony as it directly impacts battery life time. The ARM7TDMI Thumb-aware processor inherits the low power consumption of the ARM7 processor at approximately 1.8 mW/MHz when executing Dhrystone 2.1. However, a more system level view must be taken as external memory accesses dominate processor system power consumption due to on and off chip capacitances.

While more 16-bit Thumb instructions are needed to encode a task, since they are only half the width of an ARM instruction, 30% less energy is expended on average when accessing external 8-bit ROM. Reducing system power consumption in this way extends battery life.

In addition, the processing power and static design of the core helps reduce system power consumption and cost. Benchmarking on real GSM code has shown that while 8-bit CISCs will supply the 1 MIP of processor performance required for a digital cellular phone with clock speed greater than 10 MHz, the ARM7TDMI has the same performance at 800 KHz. The lower clock speed reduces power consumption, simplifies design and means that lower cost "glue" logic can be used.

ARM has also benchmarked Thumb code size against 8-bit CISC processors from the last generation of phones on real GSM code. In these tests, Thumb code size was 33% denser than the 68K, 10% denser than an H8 and 33% denser than the Z80. This is mainly due to the fact that the ARM7TDMI with its 4-Gbyte of flat address space does not need to page code as 8-bit processors do. The other major factor is the fact that a 32-bit RISC architecture is inherently a better target for modern C compilers to generate more efficient code.

The reduction in code size may enable the designer to eliminate a memory IC. However, it is unlikely that a product would be shipped with code size just breaking into a new memory device but rather that the spare memory

would be filled with software features. By using Thumb to reduce code size, the software engineer is given more memory space to add in more new features for the next generation product without any increase in system cost. In a cellular phone, these could be features such as hands free dialling, enhanced GUIs or organizer functions.

The next generation of phones which may well use processors embedded with flash memory would see a direct cost benefit of using Thumb thanks to a smaller die size.

In summary, not only does the ARM7TDMI improve performance for the same system cost, but power consumption is reduced and new software features are made possible.

6.0 Summary

The Thumb extension to the ARM architecture provides 32-bit RISC performance from the narrow (8 and 16-bit) memories common to cost-sensitive consumer applications.

The advantages are not only limited to narrow memory performance, but also include denser code enabling smaller amounts of memory to be used for reduced system cost.

The Thumb instruction set does not replace the ARM instruction set; a Thumb-aware processor can execute both instruction sets. This allows the designer to optimise code at a subroutine level for either ARM's extra 15% performance or Thumb's 30% better code size.

Code that requires the maximum performance of ARM can execute in ARM state from 32 bit wide (possibly on chip) memory; code that requires maximum code density can be run in Thumb state from low cost 16 bit or 8 bit wide memory.

A special branch instruction in both instruction sets is used to jump and switch to the other instruction set. This allows complete interworking of Thumb and ARM functions. The ARM Software Development tool kit has been fully extended to aid the user develop software for Thumb aware systems.

A Thumb-aware ARM is not a 16 bit architecture. The underlying features of a full 32 bit shifter, ALU, 32 bit registers, 32 bit address space, 32 bit memory data transfers and data processing instructions give the user the full power of a 32 bit processor for the cost of a 16 bit system.

7.0 References

For more information on ARM, please visit ARM's Web page at <http://www.arm.com>

- [1] T.E. Bell, "Incredible Shrinking Computers", IEEE Spectrum, May 1991, pp. 37-41.
- [2] D.A. Patterson and J.L. Hennessey, 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.