Improving ARM Code Density and Performance

New Thumb Extensions to the ARM Architecture

Richard Phelan June 2003

One of the most important challenges facing today's system designers is to balance the competing requirements for high-performance embedded systems at low cost with long battery life. ARM has led the industry in finding innovative solutions that help the designer find optimum solutions to this equation.

Thumb®-2 core technology is the latest enhancement to the ARM® architecture, building on existing ARM solutions to further extend the reach of ARM solutions in low-power, high-performance systems. This paper examines the issues influencing designers' choices in balancing the performance/power/cost equation, and provides details of how Thumb-2 core technology will help to produce better solutions.

ARM and Thumb Instruction Sets

The ARM instruction set provides the definitive and complete 32-bit instructions for the ARM architecture.

The Thumb instruction set is an extension to the 32-bit ARM architecture that enables very high code density.

The Thumb instruction set features a subset of the most commonly used 32-bit ARM instructions which have been compressed into 16-bit wide operation codes. On execution, these 16-bit instructions are decoded to enable the same functions as their full 32-bit ARM instruction equivalents.

An improvement in code density of around 30 percent (compared with the full ARM instruction set) is possible with Thumb technology. This efficiency is however typically at the expense of performance, as although a single Thumb instruction is equivalent to a single ARM instruction, more 16-bit Thumb instructions are needed to accomplish the same overall function.

Where there is no difference in the time taken to fetch an instruction, ARM code will, therefore, show a performance advantage over Thumb.

Design Trade-Offs

The important parameters that designers focus on include cost, performance and power. Being able to use a combination of ARM and Thumb code within an application enables designers to balance the cost, performance and power characteristics of the system.

Code density impacts program memory, hence memory size. With memory forming one of the major components of system cost, and the proportion of cost attributable to memory increasing with system complexity, a reduction in memory size is desirable.

While minimizing power consumption is always important, for designers of portable applications it is an essential to maximize battery life and, therefore, power considerations have a strong influence all design decisions.

Where performance is the primary concern, generally the fewer instructions required the better, therefore, using ARM instructions alone will usually give the best results.

Performance and power dissipation are closely linked. By using more efficient instructions (ARM instruction set), it may be possible to clock the design at a lower speed to meet the specified functionality and, therefore, significantly reduce the power dissipated in the design. Alternatively, by completing the required operations earlier then putting the system into a low-power mode, it may be possible to reduce the overall system power consumption.

Less obvious is the relation between code density and power. Achieving high code density can also help to improve power dissipation in systems where differing memory types have differing power implications.

Any off-chip data or instruction transaction will tend to dissipate more power than operations that keep everything on-chip as generally more gates will be clocked and more clock cycles will be required, however, on-chip memory is limited in quantity.

Writing applications using Thumb instructions will enable more of the most frequently used code to be stored in on-chip memory and so it can be said that higher code density can also be compatible with the aim of achieving low power.

The conclusion from this is that the performance-density-power trade-off is not straightforward. The consequence of this is that manually tuning code to reach the desired balance between these parameters can take a considerable amount of time.

Code Development Process

As well as the fundamental design trade-offs, overall development time is an extremely important consideration.

In order to achieve the desired balance of power, performance and code density to produce an optimized design, designers tend to use a mixture of both ARM and Thumb instructions.

Identifying performance critical code enables the fast routines to be developed using ARM instructions. Where possible, Thumb instructions are used for the remainder of the code to minimize memory footprint.

This is a practical solution to code development using ARM and Thumb. However, there are some limitations in using this mixed ARM-Thumb approach.

Not all ARM instructions have Thumb equivalents, so some ARM instructions must still be used even when the target is the highest code density.

For example, there are no Thumb equivalents for access to special functions (such as SIMD), access to co-processor registers or use of privileged instructions. To access these, Thumb code must call an ARM code function which performs the operation before returning to the Thumb code.

During development, the boundary between ARM code and Thumb code implementation can change as the memory limits are explored. However, the true picture only emerges as the design nears completion and the real hardware environment becomes available.

These factors can complicate the development process. Fine-tuning code to optimize between ARM and Thumb usage can take many iterations, especially when aggressive memory limits are set.

Performance, Density and Efficient Development: ARM Thumb-2 Core Technology

Thumb-2 core technology extends the ARM architecture to add enhancements to the Thumb ISA that benefit code density and performance. The resulting ISA consists of the existing 16-bit Thumb instructions augmented by:

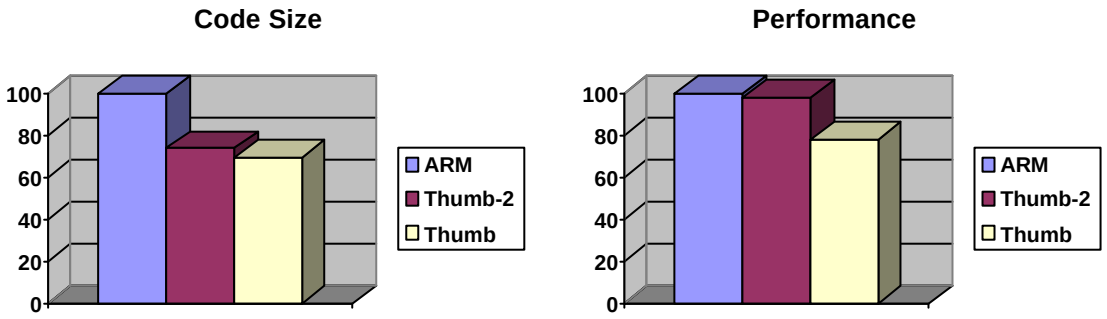
- New 16-bit Thumb instructions for improved program flow
- New 32-bit Thumb instructions derived from ARM instruction equivalents
- At the same time the ARM 32-bit ISA has been improved with the addition of new 32-bit ARM instructions for improved performance and data handling

The addition of 32-bit instructions to Thumb overcomes the previous limitations as

among these new 32-bit instructions are those for co-processor access, privileged instructions and special function instructions such as SIMD. Thumb can now access all of the instructions it needs to enable both high performance and exceptional code density.

The new Thumb-2 core technology provides:

- Access to equivalents for virtually all ARM instructions
- 12 completely new instructions which change the performance-code size balance
- Performance which reaches 98 percent of C code developed using the ARM instruction set alone
- Memory footprint which is just 74 percent of an ARM–equivalent implementation.
- Better than existing Thumb high density code:
    - 5 percent smaller
    - 2-3 percent faster



Where previously there might have been uncertainty in selecting between the use of Thumb and ARM instruction sets, Thumb-2 core technology will now become the default instruction set for the majority of applications.

Using Thumb-2 core technology will considerably simplify the development process, especially when the trade-off between performance, code density and power is not straightforward.

In addition, code ‘blending’, changing the mix between ARM and Thumb instruction usage, will no longer be required.

For developers with a significant investment in ARM and Thumb code for previous generations of the ARM architecture, Thumb-2 core technology provides a means for developers to gain significant benefits in code density by focusing on a few areas of performance code, leaving the majority of code untouched. This enables new features to be added to existing applications and for those enhancements to be brought to market quickly.

Thumb-2 core technology will get developers closer to the optimal end solution – faster.

Balancing Performance and Code Size

New instructions have been developed for Thumb-2 core technology as a result of analyzing code generated by both the ARM and Thumb compilers, across a wide range of examples.

By hypothesizing new instructions, it is possible to see the effect on both performance and code density.

Instruction Highlights

Bit Field Instructions

To improve the handling of packed data structures, instructions to insert and extract bitfields have been added to both the Thumb 32-bit instructions and the ARM 32-bit instructions. This reduces the number of instructions required to insert or extract a number of bits to or from a register making it more attractive to use packed structures and, thereby, also reducing the data memory requirements.

ARM (v6 or earlier)	Thumb-2 (ARM or Thumb)

`AND r2, r1, #bitmask`
`BIC r0, r0, #bitmask << bitpos`
`ORR r0, r0, r2, LSL #bitpos`
	`BFI r0, r1, #bitpos, #bitwidth`

The above shows that for simple ARM cases where the mask and shifted mask do not exceed the limits of the fields within the ARM instructions, three instructions are required. If the field widths are larger, then more instructions would be required. Thumb-2 core technology does not have this limitation. Also the ARM code requires an extra register for the intermediate value.

Bit Reversal Instruction

This instruction transfers each bit of the source register from bit [n] to bit [31-n] in the destination register. There are a number of ways in which this can be achieved without using the bit reversal instruction, for example a sequence of fifteen instructions that successively swaps smaller units until all bits are reversed; however this also requires an extra working register. The bit reversal instruction therefore saves a considerable number of instructions. Bit reversal is useful in DSP algorithms such as FFT.

16-bit constants

To provide greater flexibility in handling program constants, two new instructions have been added to the ARM 32-bit instruction set and the Thumb 32-bit instruction set. One instruction, MOVW, loads a 16-bit constant into a register and zero extends the result. The other, MOVT, loads a 16-bit constant into the top half of a register. In combination, the pair can be used to load a 32-bit constant into a register. This is commonly used to load the address of a peripheral prior to accessing one or more of its registers and is currently accomplished by using literal pools, which are 32-bit constants embedded in the instruction stream and accessed relative to the program counter.

Literal pools are useful for storing constants and keeping down the size of the code needed to access them, however, they can incur an overhead in cores implementing a Harvard architecture. This overhead is the number of extra cycles needed to make the constant held in the instruction stream available to the data port of the core. This means either loading the constant into data cache or being able to access program memory from the data port on the processor.

To embed the constant in two halves in two instructions means that the constant is already in the instruction stream and, therefore, does not require a data access. This is useful for small literal pools as reducing the cycles needed to access the constant improves performance and effectively reduces the power consumed in accessing the constant.

Table Branch instruction

The use of tables to control program flow based on the value of a variable is a common feature of high level languages, which translates well to the ARM and Thumb instruction sets. As ARM tends to be used to high-performance code, the compiler will tend to choose a code sequence that favors speed at the expense of size. Conversely the Thumb compiler will tend to favor a code sequence that uses packed data tables to minimize the memory used by the combined code and table.

Thumb-2 core technology introduces a table branch instruction that combines the best of both techniques, enabling a minimal number of instructions to be used with packed data, so producing maximum performance at minimal code and data footprint.

IT – if then

The ARM instruction set includes the ability to conditionally execute every instruction. This feature is useful when the compiler is generating code for short conditional clauses in code. However, the Thumb 16-bit encoding space does not have sufficient space to retain this ability and, therefore, this feature is not available to the Thumb compiler.

Thumb-2 core technology, however, introduces an instruction which provides a similar mechanism. The IT instruction predicates the execution of up to four Thumb instructions on the basis of one of the condition codes, which are based on one or more of the condition flags in the status register. This enables Thumb code to achieve similar levels of performance as the ARM code.

ARM	Thumb	Thumb-2
`LDREQ r0,[r1]`	`BNE L1`	`ITETE EQ`
`LDRNE r0,[r2]`	`LDR r0, [r1]`	`LDR r0, [r1]`
`ADDEQ r0, r3, r0`	`ADD r0, r3, r0`	`LDR r0, [r2]`
`ADDNE r0, r4, r0`	`B L2`	`ADD r0, r3, r0`
	`L1`	`ADD r0, r4, r0`
	`LDR r0, [r2]`	
	`ADD r0, r4, r0`	
	`L2`	

In the example above, the ARM code will occupy 16 bytes, the Thumb code 12 bytes, and the Thumb-2 core technology code will occupy 10 bytes. The ARM code will take 4 cycles to execute, the Thumb code will take between 4 and 20 cycles and the Thumb-2 core technology code will take either 4 or 5 cycles. The Thumb cycle count figure depends upon accurate prediction of the branches. In the case of Thumb-2 core technology, the IT instruction can be folded in a similar manner to branch instructions, reducing the cycles from 5 to 4.

Compare Zero and Branch – CZB

This instruction can be used to replace a commonly-used sequence of a compare with zero followed by a branch instruction. This is typically used to test address pointers. In addition to the program flow control, data processing and load/store instructions, the new 32-bit Thumb instructions include co-processor access instructions, which makes it possible for the first time to write Thumb code for the Vector Floating-Point (VFP) unit, amongst other co-processors. Along with instructions to access the system registers, this enables entire applications to be written for Thumb state, without needing to switch to ARM state to access special functions.

Case Study: Application Example

Table 1. illustrates the effect of compiling an existing ARM-Thumb implementation for Thumb-2 core technology.

The 1Mb application was originally implemented with 80 percent of the code compiled for Thumb to achieve the best code density, and the remaining 20 percent targeting the ARM instruction set to get the required performance. The overall saving when compiling all of the code to Thumb-2 core technology is 90k bytes, or 9 percent.

ARM	**Thumb-2**
200k	150k
Thumb	**Thumb-2**
800k	760k

Table 1. Compilation for Thumb v4

However, ARM code, tuned for performance, is more likely to reside in cache or tightly coupled memory. Analyzing the effect of Thumb-2 core technology compilation on just the code residing in the tightly coupled memory demonstrates even greater efficiency for the use of the on-chip memory. Assuming 128k of instruction TCM is available, with 50 percent occupied by ARM code and 50 percent occupied by Thumb code yields a 15 percent memory saving.

ARM	**Thumb-2**
64k	48k
Thumb	**Thumb-2**
64k	61k

Table 2. Cache/TCM memory saving

Summary

Thumb-2 core technology is a significant enhancement to the ARM architecture, which provides performance at higher code densities than previously achievable with the ARM architecture. In addition, Thumb-2 introduces a number of new features to further improve program flow, program efficiency and code size. Together these benefits will enable designers to pack more features into devices while obtaining improved power and performance characteristics, providing a more complete base for feature-rich, end-user devices.