# THUMB

## Introduction to Thumb

MS4417 - 3.0 March 1998

# Introduction

High-end embedded control applications such as cell-phones, disk drives and modems are demanding more performance from their controllers while still requiring low costs.

CISC cores are hitting their performance ceilings. Their large number of transistors tends to make them power-hungry, big and expensive as well as difficult to integrate, resulting in a high overall system cost.

RISC cores offer a potential solution to these problems. In the past RISC processors often lost out to CISC processors because of poor code density, which required larger memory sizes and a consequent high system cost.

*Extended architecture*

The ARM RISC architecture offers the low power consumption, small die size and high performance needed in embedded applications. ARM has extended this architecture in order to address the code size problem by developing **Thumb**, a new instruction set.

This overview describes Thumb, a major innovation from Advanced RISC Machines (ARM), which is the basis of a new series of microcontrollers from Mitel Semiconductor.

# Solutions to the code-size problem

There are several approaches to tackling the code-size problem:

- *hand code in assembler*

  The designer can consider hand-coding assembler for code-size optimization. However, this can take an impractical amount of time and may produce code that is difficult to support and only 10-20% more compact than that from a good compiler. The root of the problem, inefficient code, is still unresolved.

- *improve the compiler*

  Compiler technology could also be improved, but again the lower limit will be the code size achievable by hand-coding.

- *use compressed code*

  Another option is to use some form of compressed code that is expanded at run time. However, fast decompression that does not impact performance is difficult and requires additional system resources.

The ARM solution to the problem uses a combination of software and hardware. Elegant and simple, it builds on ARM’s already substantial advantages of:

- Industry-leading MIPS/Watt performance
- Excellent code-density
- Small die size for integration
- Global multi-vendor sourcing

# The Thumb Concept

Thumb is an extension to the ARM architecture. It contains 36 instruction formats drawn from the standard 32-bit ARM instruction set that have been re-coded into 16-bit wide op-codes. This brings very high code density, since Thumb instructions are half the width of ARM instructions. On execution, these new 16-bit Thumb op-codes are decompressed by the processor to their ARM instruction set equivalents, which are then run on an ARM core as normal.

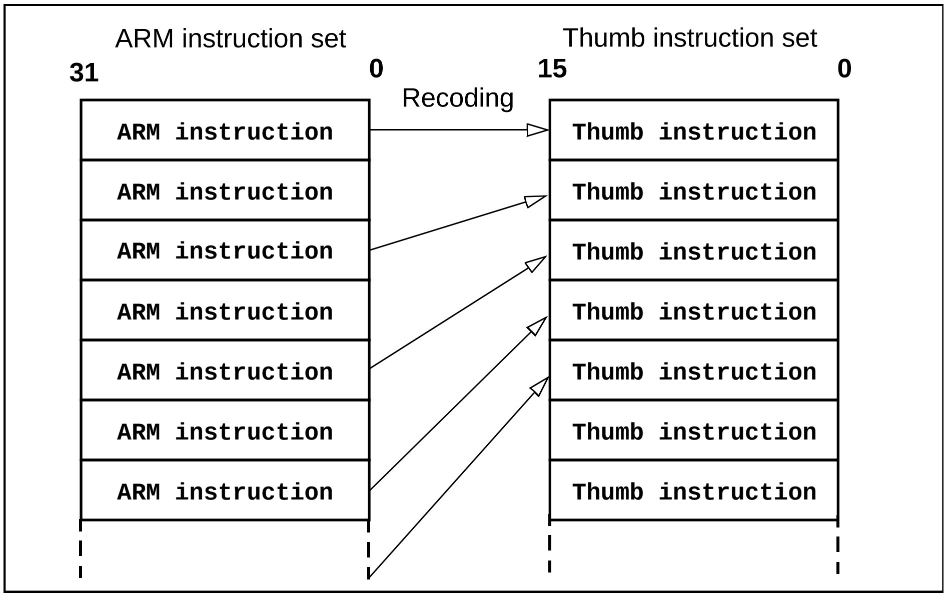


***Figure 1: instructions as a recoded subset of the ARM instruction set***

*Unique advantage*

Thumb is not just another mixed instruction set concept. Thumb-aware cores have two separate instruction sets—a unique advantage, since it allows the designer to keep all the power of ARM's 32-bit instruction set while benefiting from the code-size advantages of the Thumb instruction set. The fact that the two instruction sets are quite separate also means that decoding logic is extremely simple, and this in turn keeps silicon area small and maintains ARM's industry-leading low-power and MIPS/Watt performance.

*Size and performance-critical routines*

Since Thumb-aware cores are able to execute the standard ARM instruction set as well as the new Thumb instructions, the designer can 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.

*32-bit RISC performance*

Thumb-aware cores such as the ARM7TDMI still have ARM's full 32-bit architecture, so the designer retains 32-bit RISC performance. It is the combination of the two instruction sets running on a 32-bit Thumb-aware core that makes this an effective solution to the code-size and performance problems of 16-bit systems.

*30% code density improvement*

Results have shown around 30% code density improvement compared to ARM code, bringing Thumb-aware processors below traditional CISC cores for code size.

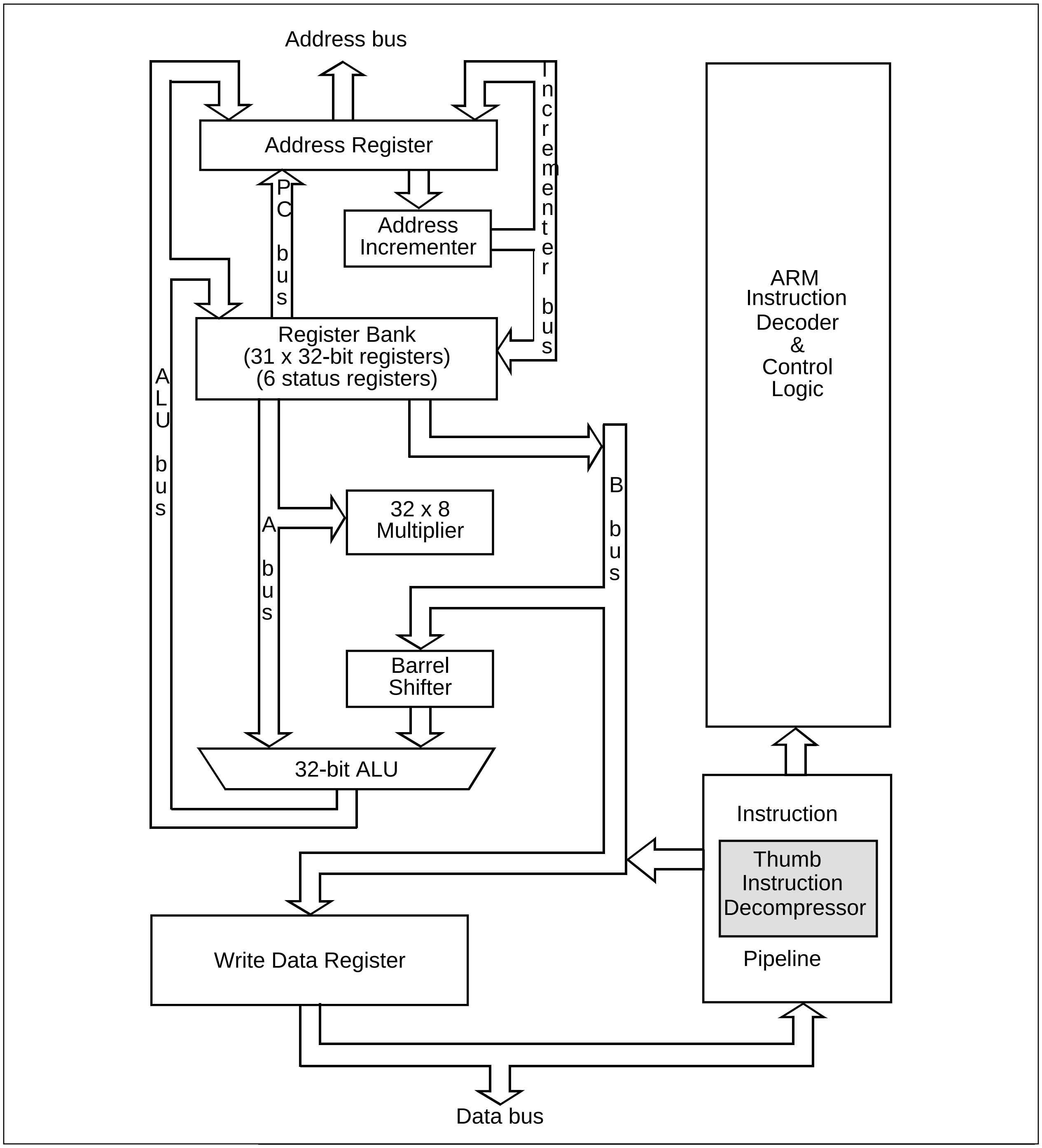


*Figure 2: ARM7TDMI core showing the Thumb instruction decompressor*

## An Introduction to Thumb

*Half-word support*

In addition to creating the new Thumb instructions, ARM has added half-word support (16-bit data) to both the Thumb and ARM instruction sets. ARM therefore now fully supports 8, 16 and 32-bit data. Optional sign-extension has also been added for Thumb and ARM cores to support 8 and 16-bit signed data operations.

*Enhanced ARM software toolkit*

The ARM software toolkit has also been enhanced to support the development of Thumb code. The programmer can use the toolkit to write ARM code, Thumb code or both, which will sit together in system memory.

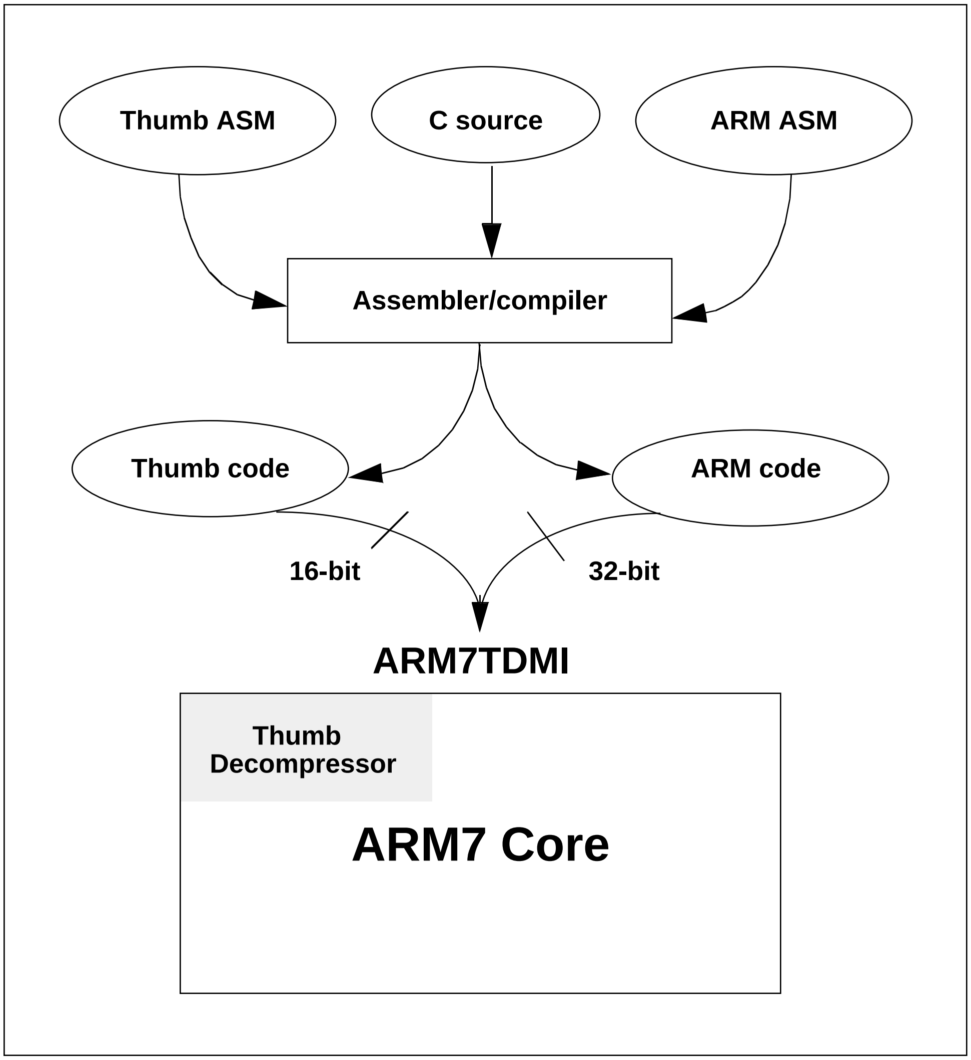


***Figure 3: Software development flow for Thumb-aware core***

# Example system configurations

The following three configurations demonstrate a Thumb-aware core in a system.

**Example 1**

This system benefits from the narrow external bus and memory that low-cost embedded applications demand.

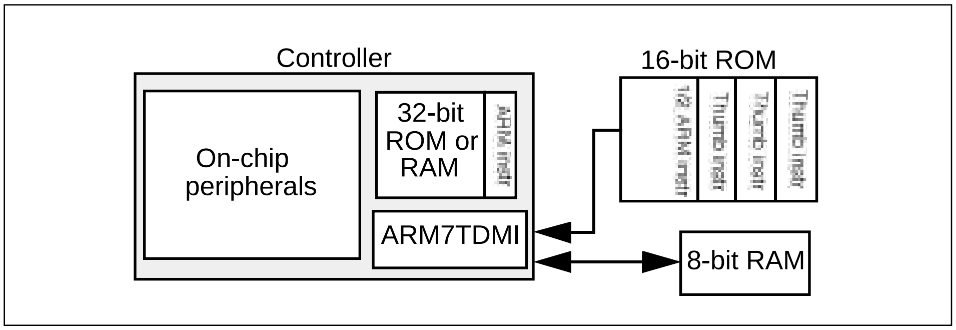


***Figure 4: Low-cost 16-bit controller and memory system***

The controller integrates customer-specific on-chip peripherals as well as small amounts of fast 32-bit ROM or RAM which is used to store speed-critical code. When the Thumb-aware 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. External 16-bit ROM is used for code and constants storage while 8-bit RAM contains scratchpad data.

**Example 2**

This configuration shows how a Thumb-aware core can be used with slow, low-cost 32-bit ROM.

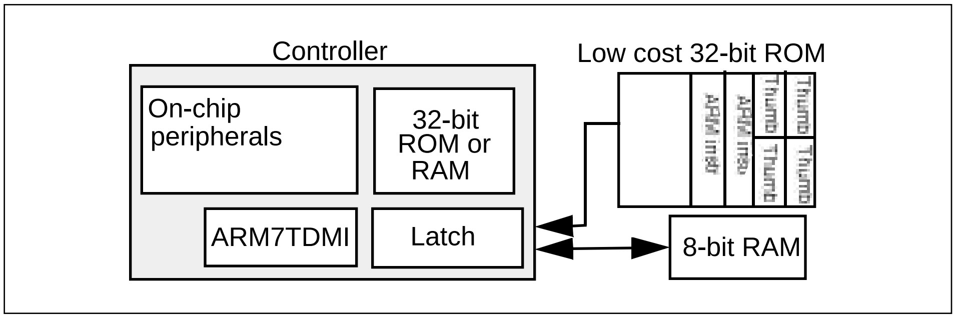


***Figure 5: 32-bit* system with low-cost ROM***

The ROM stores a mixture of routines of 32-bit ARM code with one instruction per 32-bit word and routines of Thumb code with two instructions per word. Each external fetch draws either one 32-bit ARM instruction or two 16-bit Thumb instructions. ARM instructions flow into the core pipeline in the usual way. However, in Thumb state, one Thumb instruction goes into the pipeline while the other is stored on a 16-bit latch, which is effectively a one-instruction prefetch buffer. At the next fetch, this stored instruction is immediately available to the core.

Simulations have shown that in this configuration with 200nS ROM, the Thumb solution will outperform a standard ARM core by between 10 and 20% depending on code and processor clock frequency. This is because the ARM solution incurs wait-states whilst fetching each instruction from the ROM, whereas the Thumb-aware solution only has to wait for one instruction in every two.

With high speed ROM that is clocked at the processor frequency, no wait states are incurred and hence the 32bit ARM mode will always outperform the Thumb-mode.

### Example 3

This solution represents the final Thumb-aware step before moving to a standard ARM core for its extra performance in 32-bit systems.

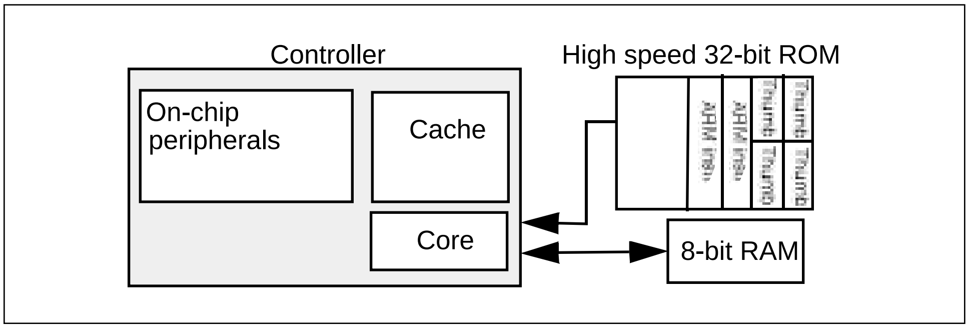


***Figure 6: High performance 32-bit system***

Using high-speed ROM and an on-chip cache, this system offers the highest performance of Thumb-aware solutions since 32-bit ARM instructions can be run straight out of fast memory. Code size and system cost are obviously greater than the low-cost 16-bit bus and memory systems.

# Summary of the Thumb Advantage

### Excellent code density

The Thumb instruction set gives excellent code density compared to both 32-bit cores and the 8 and 16-bit processors commonly used in embedded applications. Memory size and system costs are thereby reduced.

### 16-bit instructions

Thumb instructions are only 16-bits long, meaning that the system data bus need only be 16-bits wide. This reduces both power consumption and PCB area, leading to lower-cost, lower-power systems.

### Smallest core die size

Thumb-aware cores have amongst the smallest core die sizes in the industry (ARM7TDMI is less than 5 mm2 on 0.6µ). The ASSP and ASIC designer therefore gets a reduced system die size driven both by a core that is smaller than common 16/32 bit CISCs and by the reduced requirement for on-chip programme ROM. Combined with simplified—hence cheaper—testing compared to CISCs as well as low-power commodity plastic packaging, this brings a lower-cost product than today’s common solutions.

### Full 32-bit architecture

Thumb instructions execute on ARM’s full 32-bit RISC architecture. The designer is therefore able to exploit fast 32-bit maths and a simple unsegmented memory map that has a 4 GByte address space; room for the most complex of embedded control applications.

The standard architecture combined with new tools that can compile for ARM code, Thumb code, or a mixture of both, guarantees forward compatibility with the existing 32-bit ARM family. This provides the 16-bit system designer with a future migration route to an already-exisiting family of 32-bit cores.

### Code size and performance

Thumb-aware cores such as the ARM7TDMI execute both 32-bit ARM and the new 16-bit Thumb instructions. Designers can mix routines of Thumb and ARM code in the same address space. This allows the programmer to trade-off code size and performance, routine by routine, as required by the application.

### Enhanced ARM software toolkit

The new Thumb instructions are fully supported by an enhanced toolkit which is “Thumb-aware”. This toolkit include a Microsoft Windows Integrated Development Environment and seamless interworking between Thumb and ARM states.

### Protected investment

Investment in existing ARM software is protected, as Thumb-aware cores execute ARM code. For use in Thumb state, existing source code only needs recompiling.

### Building on the ARM advantage

Thumb-aware cores build on the standard ARM advantages of extremely low power consumption, industry-leading MIPS/Watt performance, small die size for integration/ low cost and global multi-vendor sourcing.

To summarise, the Thumb architecture gives designers of 16-bit systems access to the performance of ARM’s 32-bit cores at 16-bit system costs.

# Thumb-aware Cores and Roadmap

*High performance, superior code density*

The embedded control market is currently served by 8 and 16-bit offerings from multiple vendors. However, in higher-end applications, these products often no longer offer the required performance. What such applications need is 32-bit RISC processor performance combined with a code density superior to that of 16-bit CISC processors. Thumb offers both of these features, enabling the ARM architecture to bridge the gap between today's 16-bit systems and the 32-bit systems that will be required tomorrow.

*More performance without extra cost*

Mitel Semiconductor therefore believes that Thumb-aware cores will be especially successful in feature-hungry consumer applications which today are using 8 and 16-bit controllers and which are looking for more performance without extra cost.

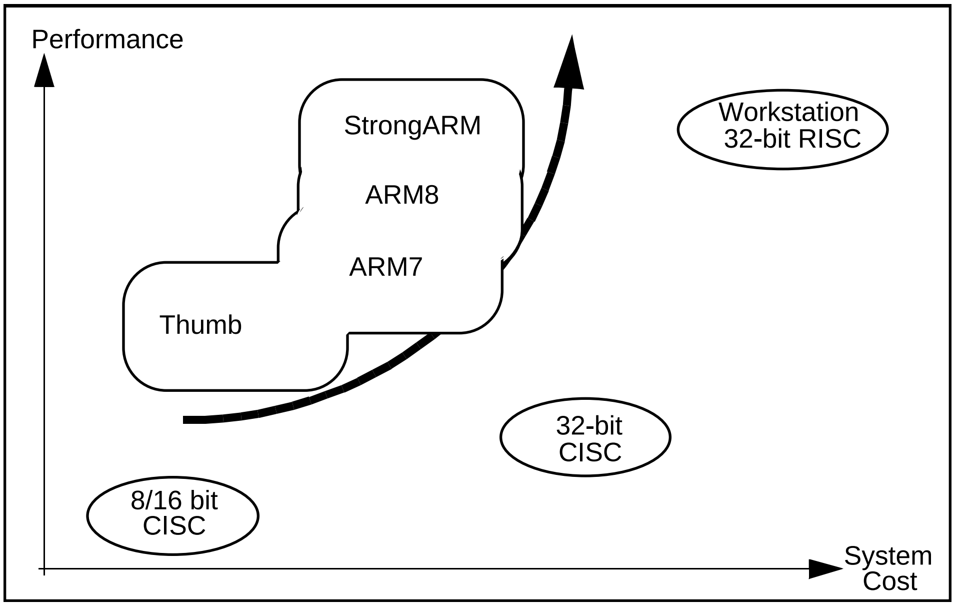


**Figure 7: Thumb-aware cores filling the performance gap**

*Source code compatibility*

Since Thumb-aware cores are simply an extension of the ARM architecture, the designer can compile for Thumb-code, ARM-code or a mix of both. This source code compatibility between Thumb-aware cores and ARM cores means that a seamless future upgrade path to a 32-bit system is already in place, making Thumb-aware cores a safe investment for the future.

The simplicity of the Thumb extension also means that future ARM cores with even higher performance, such as the ARM8 and StrongARM families, can also be available with Thumb-aware options.

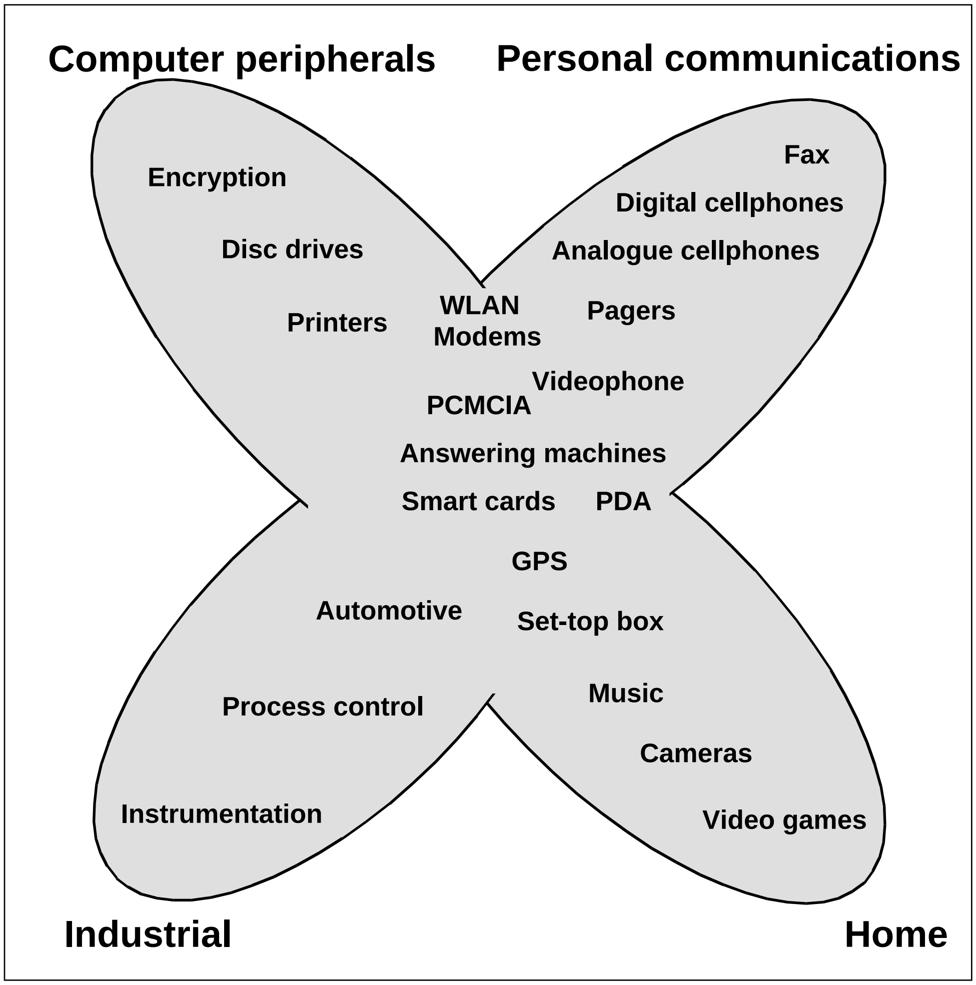


***Figure 8: Application areas for Thumb-aware cores***

*The ARM7TDMI*

The first core to feature Thumb-compatibility is the ARM7TDMI. This is an ARM7 core with:

- On-chip ICEbreaker debug support
- 32-bit hardware multiplier
- Thumb decompressor

*32-bit performance into 8 and 16-bit control applications*

The ARM7TDMI complements the original 32-bit ARM core range by giving low-end coverage of the embedded control market, reaching down from the 32-bit world and bringing 32-bit performance into 8 and 16-bit control applications.

Mitel Semiconductor will use the ARM7TDMI in a range of Thumb-based microcontrollers and ASSPs. Efficient performance with only a 16-bit external bus allows smaller, lower cost packages to be used and Mitel Semiconductor pays particular attention to optimising function-per-pin to exploit this advantage fully.

# Thumb Implementation

## (i) Hardware aspects

The major new addition to the ARM architecture to support the Thumb instruction set is the **Thumb decompressor**. ARM7TDMI is the first ARM core to implement this.

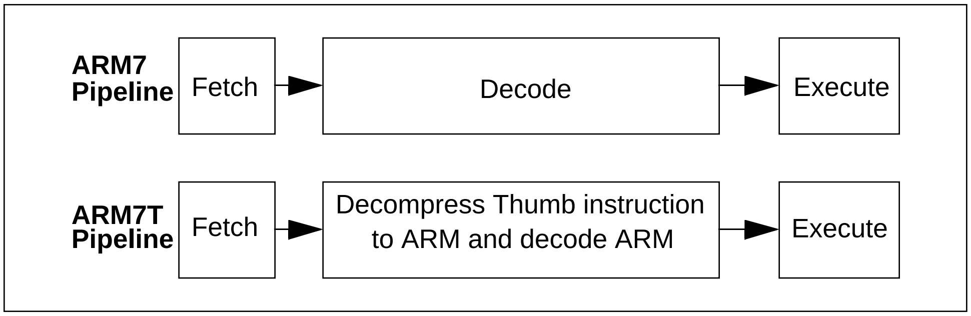


***Figure 9: Pipeline fetch, decode and execution***

*Single cycle decode and execution*

Both the ARM7 and ARM7T cores achieve single cycle execution using a 3-stage pipeline with Fetch, Decode and Execution phases. Instruction flow through each stage of the pipeline is controlled by high and low clock phases. The ARM7TDMI uses this to its advantage by decompressing the Thumb instruction during an unused phase of the clock in the Decode stage. There is therefore no additional timing overhead and single cycle decode and execution is maintained.

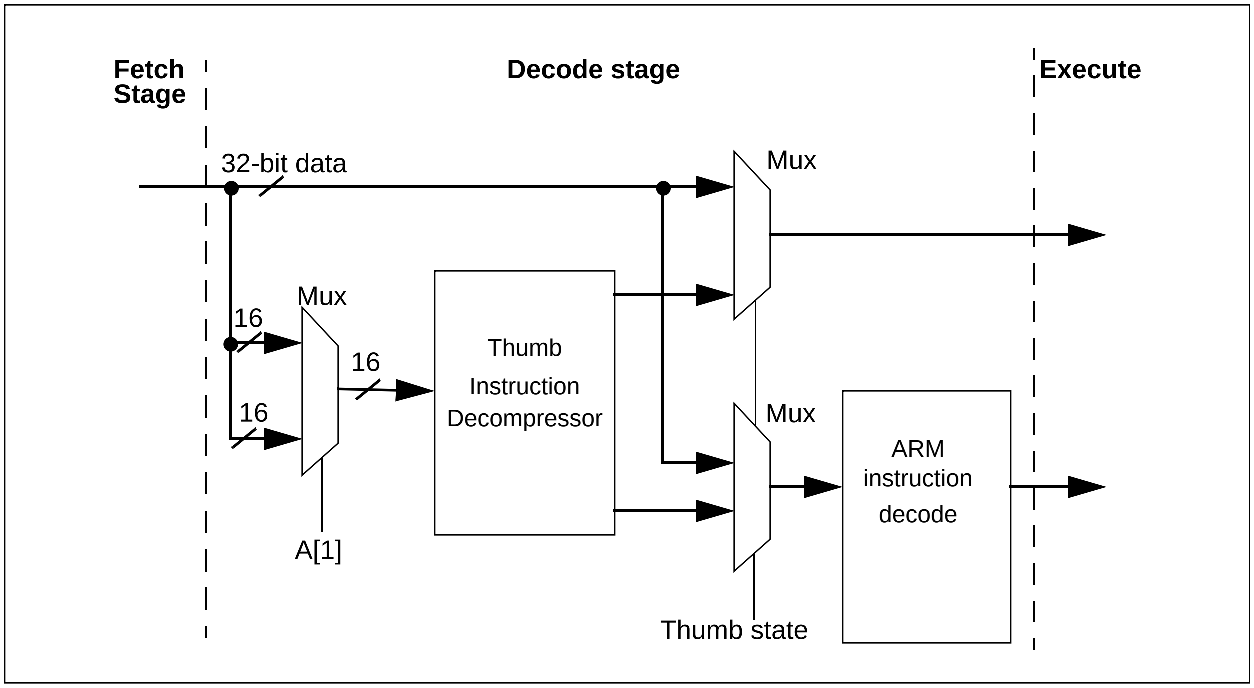


***Figure 10: Thumb decoding and decompression***

ARM instructions arriving from the Fetch stage of the pipeline pass through the ARM decoder and activate major and minor op-code bit control signals. Major op-code bits describe the type of instruction to execute while minor bits specify instruction detail such as the registers or operand specified.

In Thumb state, multiplexers direct Thumb instructions through the Thumb decompression logic. This effectively explodes the Thumb instruction into its equivalent ARM instruction. The execution of that ARM instruction then happens as normal. This may be more easily understood with an example:

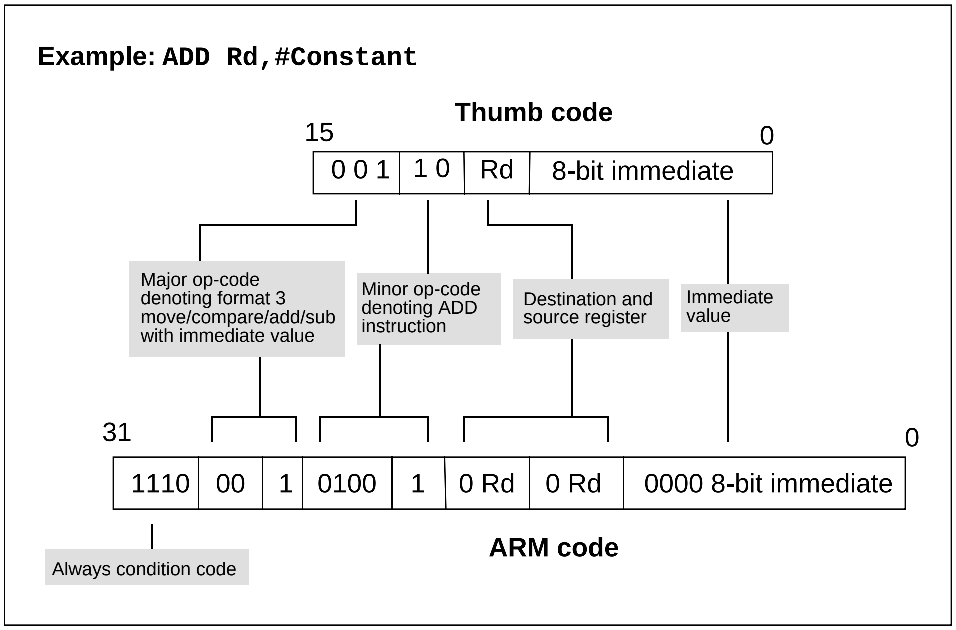

Example: ADD Rd,#Constant
Thumb code
15
0
0 0 1
1 0
Rd
8-bit immediate
Major op-code denoting format 3 move/compare/add/sub with immediate value
Minor op-code denoting ADD instruction
Destination and source register
Immediate value
31
0
1110
00
1
0100
1
0 Rd
0 Rd
0000 8-bit immediate
Always condition code
ARM code

***Figure 11: Translation of Thumb ADD to ARM ADD instruction***

*Elegant solution*

In this case, the major op-code of the Thumb instruction is placed in the ARM instruction and the minor op-code is translated via a look-up table.

The ARM instruction inherits the Always condition code driven from the major op-code.

The major op-code then selects the operand routing from the Thumb op-code to the ARM op-code. The register specifiers are expanded with zero extension from the Thumb op-code (3 bits) to 4 bits since this Thumb instruction only accesses ARM registers R0-R7. The constant value is also zero extended, specifying an unrotated 8-bit constant in the ARM op-code.

## (ii) Software aspects

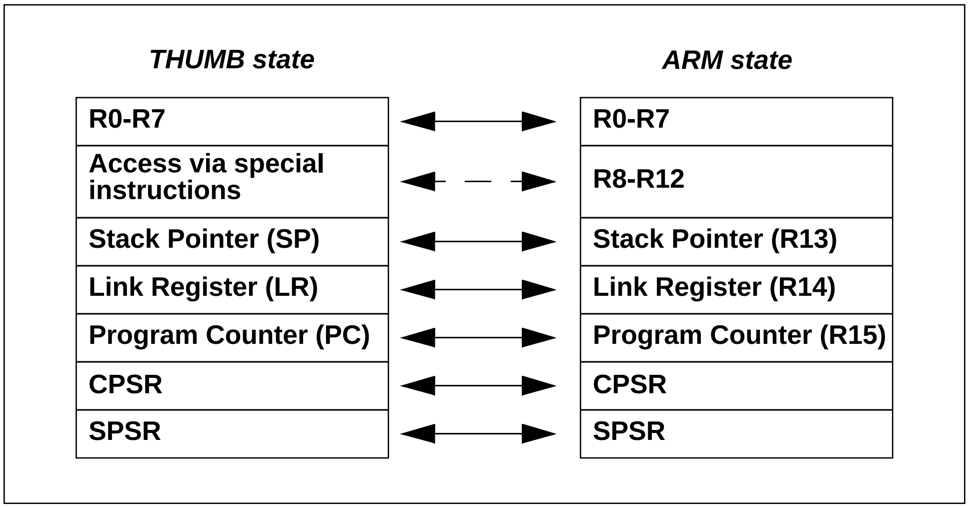
*36 instruction formats*

The Thumb instruction set contains the 16-bit equivalents of 36 instruction formats taken from the standard 32-bit ARM instruction set. Instructions chosen were those which on average did not benefit from the full width of standard 32-bit ARM instructions op-codes, or those which customer experience had shown were used most often and were therefore the most important, or those which the compiler needed in order to get the best possible code density.

*Performance/code-size trade-off*

The selection process involved shortening the fields of all the most common instructions until they fitted into the 16-bits available. The resulting instruction set was then iterated, giving more op-code bits to the most common instructions (for example, subroutine calls which account for 1/16 of the instruction set) and stealing bits from the less critical instructions until the best performance/code-size trade-off was achieved. The instruction set was future-proofed by leaving space in the op-codes for new instructions that currently trap to software handlers; a new release of the ARM compiler would be the only requirement to support this.

In order to accommodate 16-bit op-codes, some limitations on the richness of the Thumb instruction set were incurred. Most obvious is the reduced number of registers available when executing Thumb code. In place of the 15 32-bit General Purpose Registers (GPR) plus PC of the ARM, the programmer has access to 8 GPRs, the Stack Pointer, Link Register and the PC. In the Thumb instruction set, the 8 GPRs (R0-R7) are called the Lo register set. The other ARM registers (R8-R15) are known as the Hi set. The programmer has limited access to the Hi set in Thumb via moves, compares and ADDs giving him some local fast temporary storage. Moving between ARM and Thumb code does not affect the contents of the GPRs.

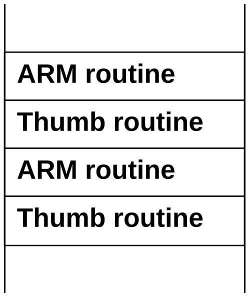


| THUMB state | ARM state |
|---|---|
| R0-R7 | R0-R7 |
| Access via special instructions | R8-R12 |
| Stack Pointer (SP) | Stack Pointer (R13) |
| Link Register (LR) | Link Register (R14) |
| Program Counter (PC) | Program Counter (R15) |
| CPSR | CPSR |
| SPSR | SPSR |

***Figure 12: Mapping of Thumb state registers onto ARM state registers***

*Thumb/ARM state bit*

Transfer between instruction sets is achieved using the BX instruction which toggles a Thumb/ARM state bit in the Program Status Register (PSR) of the Thumb-aware core. This means that routines of Thumb and ARM code can live together in the same memory space:



Rather than taking up a bit in the op-code to distinguish between formats, use of a status bit leaves room in the 16-bit op-code for a richer instruction set.

New mnemonics include ASR, LSL, LSR, ROR, LDRH, LDSB, LDSH, PUSH, POP, and STRH. These are not new instructions, but effectively pointers to full 32-bit ARM instructions, since virtually every Thumb instruction has an equivalent ARM instruction.

*Large memory-size capability*

The demand for memory in embedded applications continues to grow. This means that for an embedded solution to be durable, it must be able to handle large memory sizes. In the ARM state with its 4Gbytes of address space, this is not a problem. Nor is it an issue in Thumb state, since Thumb-aware cores are still full 32-bit processors with a 32-bit address space. 16-bit op-codes could limit the maximum displacement to 64KBytes. However the Thumb instruction set provides both a long branch instruction for up to 4MBytes and full 4GByte branches in 2 instructions.

## 12.0.1 Thumb code in action

### Simple C routine

Here is a simple C routine that demonstrates the differences between Thumb and ARM code. This routine returns the absolute value of the C integer passed to it as a parameter. The C code is:

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

The equivalent ARM assembly version is (excluding preamble):

```
iabs   CMP   r0,#0      ;Compare r0 to zero
       RSBLT r0,r0,#0   ;If r0<0 (less than=LT) then do
                        ;r0= 0-r0
       MOV   pc,lr      ;Move Link Register to PC (Return)
```

The Thumb assembly version is:

```
       CODE16           ;Directive specifying 16-bit (Thumb)
                        ;instructions
iabs   CMP      r0,#0   ;Compare r0 to zero
       BGE      return  ;Jump to Return if greater or
                        ;equal to zero
       NEG      r0,r0   ;If not, negate r0
return MOV      pc,lr   ;Move Link register to PC (Return)
```

Comparing code sizes:

| Code | Instructions | Size (Bytes) | Normalised |
|---|---|---|---|
| ARM | 3 | 12 | 1.0 |
| Thumb | 4 | 8 | 0.67 |

***Table 1: Code size comparison***

*Smaller assembled Thumb code size*

In this case, the Thumb code is 33% more dense than the ARM code for exactly the same function. Notice in the example that more instructions are needed to perform the task in Thumb code than in the ARM equivalent. However, as Thumb instructions are only half the length of ARM instructions, the assembled Thumb code size is still smaller.

### Hand-coded example

As a hand-coded example, consider a binary to hexadecimal converter:

**ARM code**:

```
      MOV   r1,r0         ;r0 holds value to convert
                          ;Load r1 with value
      MOV   r2,#8         ;Load r2 with decimal 8
Loop: MOV   r1,r1,ROR #28 ;Rotate r1 right by 28 and
                          ;put result in r1
      AND   r0,r1,#15     ;AND r1 with decimal 15
      CMP   r0,#10        ;Compare r0 with decimal 10 and
      ADDLT r0,r0,#'0'    ;if it’s less than 10 then
                          ;ADD ASCII 0 to r0
      ADDGE r0,r0,#'A'    ;otherwise (greater or equal)
                          ;ADD ASCII A to r0
      SWI 0               ;routine to write char to screen
      SUBS  r2,r2,#1      ;Subtract 1 from r2
      BGT   Loop          ;and loop back to 1 if r2 is
                          ;still greater than zero
      MOV   pc,lr         ;Load PC with Link reg (Return)
```

**Thumb code**:

```
      MOV   r1,r0         ;Value to convert in r0
                          ;Load r1 with value
      MOV   r2,#8         ;Put 8 in r2
Loop1 LSR   r0,r1,#28     ;Do logical shift right on r1
                          ;by 28 places and place in r0
      LSL   r1,r1,#4      ;Do logical shift left on r1
                          ;by 4 places
      CMP   r0,#10        ;Compare r0 with 10 and
      BLT   Loop2         ;if less than 10, branch to Loop2
      ADD   r0,#’A’-’0’-10;ADD ASCII A-0-10 (7) to r0
Loop2 ADD   r0,#'0'       ;ADD ASCII 0 (48) to r0
      SWI 0               ;routine to write char to screen
      SUB   r2,#1         ;subtract 1 from r2
      BNE   Loop1         ;if unfinished loop1
      MOV   pc,lr         ;else,load PC with Link register
                          ;(Return)
```

When code sizes are compared:

| Code | Instructions | Size (Bytes) | Normalised |
|---|---|---|---|
| ARM | 11 | 44 | 1.0 |
| Thumb | 12 | 24 | 0.55 |

***Table 2: Code size comparison***

This time, the Thumb code comes out at 45% more dense than the ARM equivalent, again for exactly the same algorithm.

| Mnemonic | 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,#lmm]` | `LDR Rd,[PC,#lmm]` |
| LDRB | Load byte | `LDRB Rd,[Rb,Ro]` | `LDRB Rd,[Rb,Ro]` |
| LDRH | Load halfword | `LDRH Rd,[Rb,#lmm]` | `LDRH Rd,[Rb,#lmm]` |
| LSL | Logical Shift Left | `LSL Rd,Rs,#Offset5` | `MOVS Rd,Rs,LSL#Offset5` |
| LDRSB | Load sign-extended byte | `LDRSB Rd,[Rb,Ro]` | `LDRSB Rd,[Rb,Ro]` |
| LDRSH | Load sign-extended 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` |

***Table 3: The Thumb instruction set***

## (iii) Software development route

Since Thumb-aware cores are able to execute ARM instructions, all of the existing ARM software base continues to run on Thumb cores, though in order to support the new Thumb instruction set fully, ARM & Mitel Semiconductor have significantly enhanced the tools available to the programmer for software development. These enhancements to the Mitel Semiconductor ARM software Development toolkit include the seamless interaction between Thumb and ARM states.

*Enhanced Mitel Semiconductor ARM software toolkit*

The Mitel Semiconductor ARM toolkit contains the following primary components:

- Full Windows GUI
- Project management features
- C compiler
- assembler
- linker
- librarian
- simulator
- UDB™ debugger

*New features*

For Thumb, a new Thumb C compiler and assembler are included, together with hooks to make the other tools “Thumb-Aware”.

**The Thumb C compiler**

The Thumb C compiler (tcc) compiles ANSI C to 16-bit Thumb instructions.
Its optimisations include:

- in-lining
- constant folding
- common subexpression elimination
- expression lifting
- live range splitting for dynamic register allocation
- tail calling
- cross-jump elimination
- table-driven peepholing
- switches for speed or code-size optimisation.

The Thumb Compiler may be used in conjunction with the standard ARM C Compiler allowing code written for Thumb to call ARM code and vice-versa.

*New Software Floating-point library*

Both ARM and Thumb compilers support software floating point through ARM’s new Software Floating-point library which runs up to twice as fast as the original (version 1.4) floating point emulation, with improved code density.

### Thumb Assembler

*ARM or Thumb code*

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

### Linker

*Mixing ARM and Thumb routines*

The ARM Linker has been enhanced to support both ARM and Thumb object types. ARM and Thumb routines can be freely mixed in an application, allowing the designer to trade off code-size against performance. Objects can be linked across the fragmented memory maps common to many embedded applications.

### Debug support

*Full C source or assembler-level debugging*

Debugging support is provided by UDB™, a full windowing debugger on Microsoft Windows platforms, which has been extended to support Thumb-aware cores. These tools provide full C source or assembler-level debugging. UDB can either debug code running on an instruction-accurate simulator (ARMulator) or on target hardware. The associated monitor program, UMON™, can readily be ported to target systems, as the target-dependent routines are supplied as source code.

For further information on the Mitel Semiconductor ARM Software Toolkit please contact your local Mitel Semiconductor Sales office or Distributor.

### ARM instruction simulator

*Benchmark and develop code*

ARM’s instruction-accurate processor simulator, 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 ARMs new C profiling tool, designers can choose optimal memory configurations that incorporate the three critical factors of speed, space and memory cost.

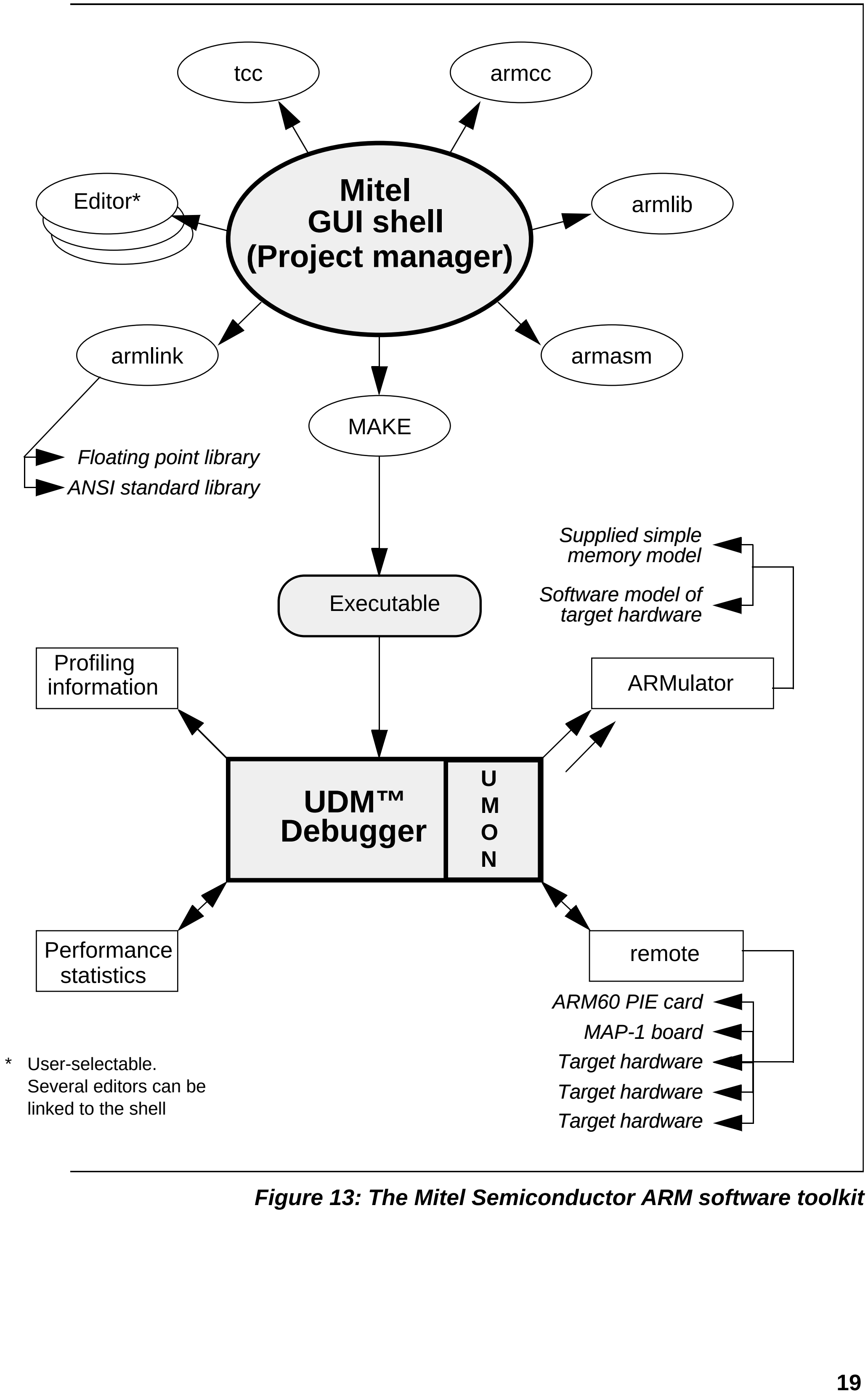


*Figure 13: The Mitel Semiconductor ARM software toolkit*

# Thumb Benchmarks

In order to build a complete picture of the performance of the ARM7TDMI Thumb-aware core against alternative solutions, ARM has put together a set of benchmarks that test two critical aspects of the Thumb concept:

- code size
- performance

## Code-size benchmarking

Through patented code compression techniques, the Thumb concept brings 32-bit performance to 16-bit systems at 16-bit system cost. The code-size benchmarking that follows measures how effective this solution is in bringing the designer the minimum code-size possible for an 8/16 bit system.

The approach taken was to commission Micrologic Solutions to generate results for Espresso, Xlisp and Eqntott. These are routines taken from the SPECint benchmarking suite. Numbers were derived for four popular competition processors by using third-party tool offerings. The data for the ARM7TDMI Thumb-aware core was generated using ARM's Thumb C compiler, described on page 17.

| **Processor** | **Eqntott** | **Xlisp** | **Espresso** |
|---|---|---|---|
| ARM7TDMI | 10608 | 26388 | 72596 |
| ARM7 core | 16768 | 40768 | 109932 |
| Intel 386 | 17640 | 28097 | 125686 |
| Intel 8088 | 19106 | 29401 | 137194 |
| Moto 68020 | 20542 | 46746 | 131854 |
| Sparc2 | 22256 | 44648 | 142752 |

***Table 4: User code size in bytes for three benchmarks***

*Source: Micrologic Solutions*

In order to ensure a fair comparison, ARM also took code size numbers that are publicly available for competing solutions and added data for the ARM7TDMI Thumb-aware core. Again with code size in bytes, these numbers are for Dhrystone 1.1, as this data is freely available.

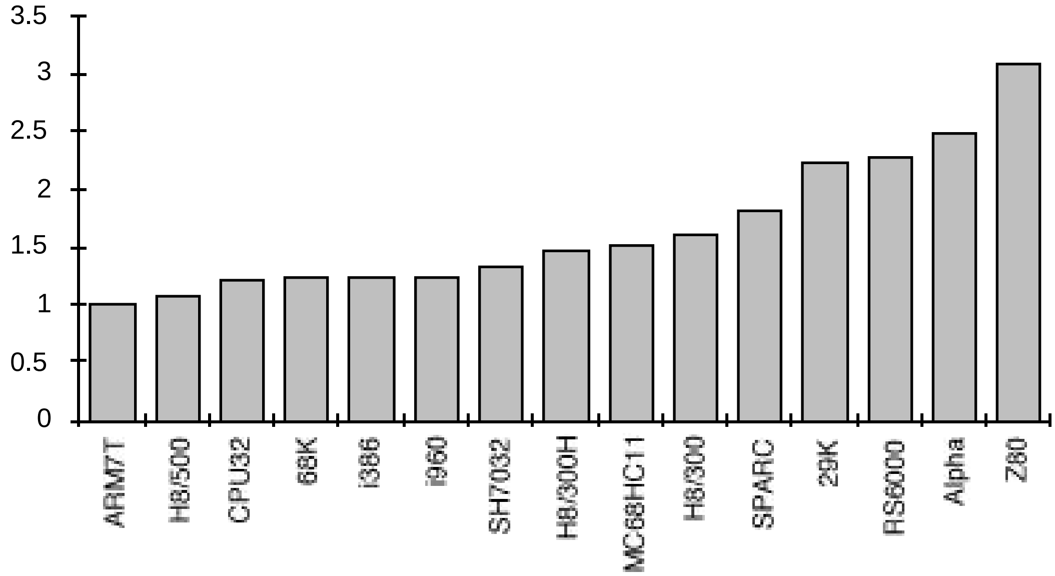


*Figure 14:*

| Processor | Size (Bytes) | Normalised |
|---|---|---|
| ARM7TDMI | 1032 | 1.00 |
| H8/500 | 1097 | 1.06 |
| CPU32 | 1254 | 1.22 |
| 68000 | 1268 | 1.23 |
| i386 | 1280 | 1.24 |
| i960 | 1280 | 1.24 |
| SH7032 | 1384 | 1.34 |
| H8/300H | 1530 | 1.48 |
| MC68HC11 | 1558 | 1.51 |
| 29000 | 2296 | 2.22 |
| Z80 | 3201 | 3.10 |

***Table 5: Normalised Dhrystone code size for large memory model***

*Source: Microprocessor Forum 1993 and vendor data*

ARM 32-bit RISC code has always been acknowledged as being exceptionally dense. In fact, as the results show, the density of native ARM code comes close to the traditional 16-bit processors, leaving other 32-bit RISC cores far behind. This is due to novel features in the ARM instruction set such as conditional execution for every instruction and register write-back options.

*Industry-leading code density*

These results show that with 30% better code density than competitors' controllers, the Thumb concept has built on ARM's already efficient code to bring industry-leading code density. This means that designers who were previously considering 8 and 16-bit controllers in order to save system code memory can now migrate to the power of ARM's 32-bit cores and get a reduction in the size of their system code. This will either enable them to eliminate a memory IC, or to use the freed memory space for new software features.

## Performance benchmarks

Methods such as performing two16-bit fetches and providing only 16-bit instructions on a 32-bit core are simple approaches to tackling the code-density problems associated with 32-bit RISC cores. However, both of these solutions lead to large losses in performance.

*Superior performance*

The purpose of these benchmarks is to demonstrate that even with Thumb core's excellent code density, its performance in 16-bit systems is superior to both standard ARM cores and common solutions from the competition.

Again, results are compared for Dhrystone 1.1, as these numbers are freely available for competing cores. The data for the ARM7TDMI core was generated using the Thumb-aware instruction simulator (ARMulator) described on page 18. This simulator provides a clock cycle count from which Dhrystone values were calculated.

In order to ensure a fair comparison, the comparison was done in 2 stages:

1. For processors connected to 16-bit wide memory systems.
2. For processors connected to 32-bit wide memories.

ARM cores are excellent solutions for portable embedded applications because of their extremely low power consumption. Therefore, values for both Dhrystone 1.1 MIPS and MIPS/Watt are shown. Processors marked with a star have an on-chip cache.

*Dhrystone MIPS*

This benchmark is defined as the Dhrystone performance (Dhrystones/second) of the processor being evaluated, divided by the performance of a VAX 11/780, namely 1757 Dhrystones/sec.

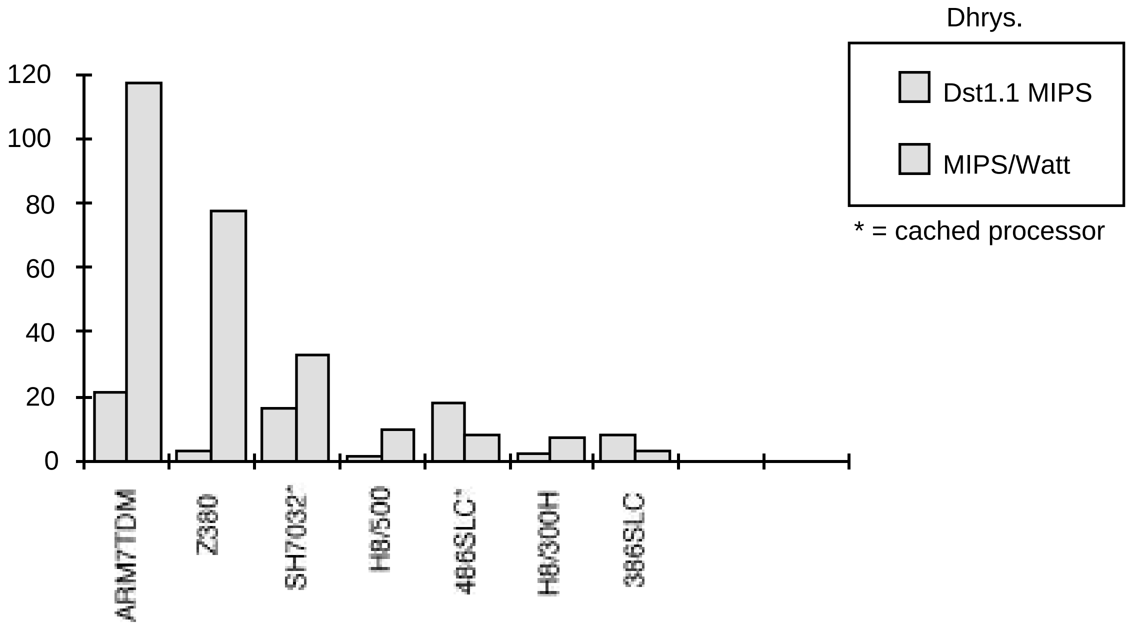


***Figure 15: Dhrystone 1.1 MIPS and MIPS/Watt at 5 Volts for processors in 16-bit systems***

| Processor | System | Power (W) | Dhrys1.1 MIPS | MIPS/Watt |
|---|---|---|---|---|
| ARM7TDMI | 33MHz 5V | 0.181 | 21.2 | 117 |
| Z380 | 18 MHz | 0.04 | 3.1 | 78 |
| SH7032(*) | 20MHz 5V | 0.5 | 16.4 | 33 |
| H8/500 | 10MHz 5V | 0.1 | 1 | 10 |
| 486SLC (*) | 33MHz 5V | 2.25 | 18 | 8 |
| H8/300H | 16MHz 5V | 0.25 | 1.9 | 8 |
| 386SLC | 25 MHz 5V | 2.5 | 8 | 3 |

***Table 6: Processors at 5 volts in 16-bit memory systems***

*Source: Microprocessor Forum 1993 and vendor data*

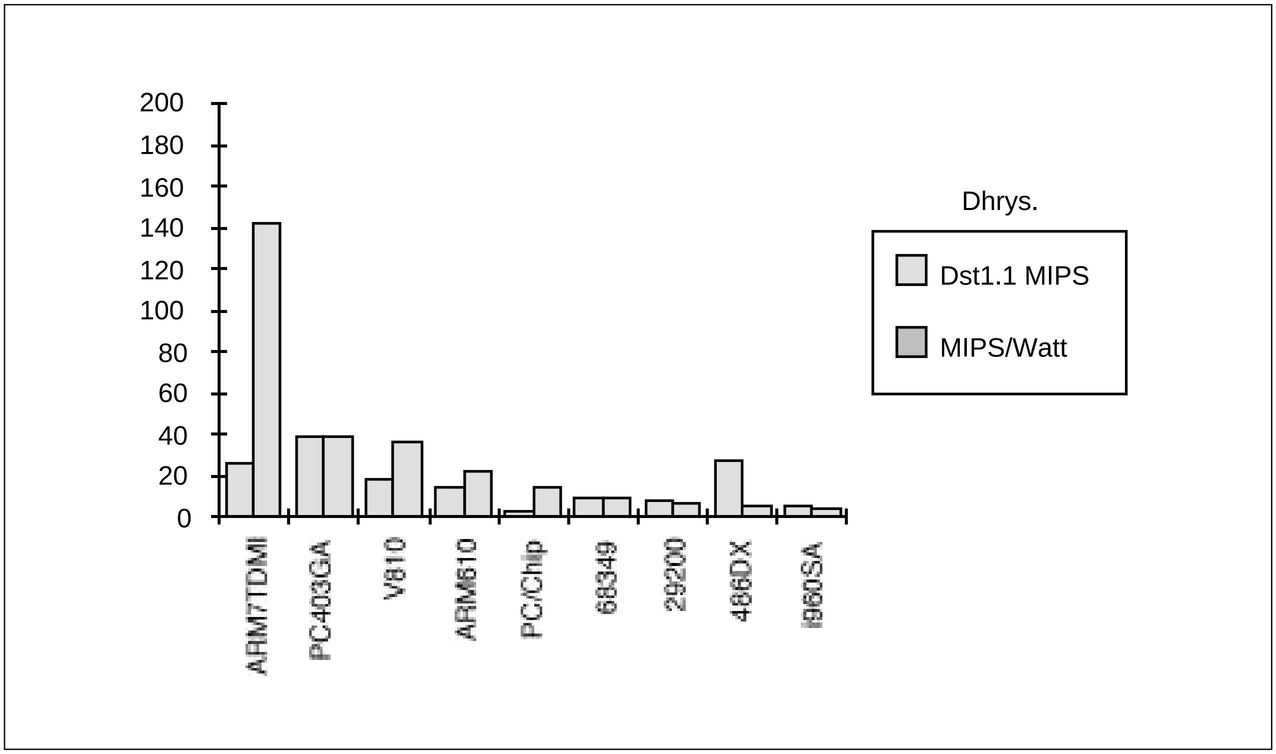


***Figure 16: Drystone 1.1 MIPS and MIPS/Watt at 5 Volts for processors in 32-bit systems***

| Processor | System | Power (W) | Dhrys 1.1 MIPS | MIPS/Watt |
|---|---|---|---|---|
| ARM7TDMI | 33MHz 5V | 0.181 | 25.8 | 143 |
| PC403GA | 40MHz 5V | 1 | 39 | 39 |
| V810 | 25 MHz 5V | 0.5 | 18 | 36 |
| ARM610 | 25MHz 5V | 0.625 | 14 | 22 |
| PC/Chip | 14.3MHz 5V | 0.216 | 3 | 14 |
| 68349 | 25MHz 5V | 0.96 | 9 | 9 |
| 29200 | 16MHz 5V | 1.1 | 8 | 7 |
| 486DX | 33MHz 5V | 4.5 | 27 | 6 |
| i960SA | 16MHz 5V | 1.25 | 5 | 4 |

***Table 7: Processors at 5 volts in 32-bit memory systems***
*Source: Microprocessor Forum 1993 and vendor data*

Performance and power consumption numbers have been simulated for the ARM7TDMI running Dhrystones 1.1/2.1 at 20 MHz and 3.3V:

| Benchmark | Power (W) | DS MIPS | MIPS/Watt |
|---|---|---|---|
| Dstone 2.1 | 0.036 | 11.6 | 322 |

***Table 8: ARM 7TDM, at 3.3Volts in a 16-bit wide memory system.***

| Benchmark | Power (W) | DS MIPS | MIPS/Watt |
|---|---|---|---|
| Dstone 1.1 | 0.036 | 15.6 | 433 |
| Dstone 2.1 | 0.036 | 14.0 | 389 |

***Table 9: ARM 7TDMI at 3.3Volts in a 32-bit wide memory system***

It is important to remember that the ARM7TDMI is capable of executing both ARM and Thumb instructions. Therefore, in a 32-bit wide memory system, it will deliver as many MIPS as the ARM7 if it runs in ARM state 100% of the time.

*Exceptional code density and performance*

The above results demonstrate clearly that the Thumb concept not only delivers exceptional code density, but excellent performance as well. Notice that even though Dhrystone 1.1 is completely cacheable in 2KB, ARM7TDMI easily out performs the competition cached processors on Dhrystone 1.1 MIPS.

*Leading Dhrystone MIPS performance*

The extremely low power consumption of the ARM7 and ARM7TDMI family make them ideal choices for portable applications. For applications where power consumption is not so important, ARM solutions still provide leading Dhrystone MIPS performance.

The ARM7TDMI outperforms the ARM7 by 150% in a 16-bit system since it does not have to do two fetches per instruction; in a 32-bit system, a Thumb-aware core would retain all the performance of an ARM7 simply by operating in ARM state 100% of the time.

HEADQUARTERS OPERATIONS
**MITEL SEMICONDUCTOR**
Cheney Manor, Swindon,
Wiltshire, United Kingdom. SN2 2QW
Tel: (01793) 518000
Fax: (01793) 518411

**MITEL SEMICONDUCTOR**
P.O. Box 660017,
1500 Green Hills Road,
Scotts Valley, California 95067-0017,
United States of America.
Tel (408) 438 2900
Fax: (408) 438 5576

CUSTOMER SERVICE CENTRES
• **FRANCE & BENELUX** Les Ulis Cedex Tel: (1) 69 18 90 00 Fax: (1) 64 46 06 07
• **GERMANY** Munich Tel: (089) 3609 06-0 Fax: (089) 3609 06-55
• **ITALY** Milan Tel: (02) 6607151 Fax: (02)66040993
• **JAPAN** Tokyo Tel: (03) 5276-5501 Fax: (03) 5276-5510
• **KOREA** Seoul Tel: (2) 5668141 Fax: (2) 5697933
• **NORTH AMERICA** Scotts Valley, USA Tel: (408) 438 2900 Fax: (408) 438 7023
• **SOUTH EAST ASIA** Singapore Tel: 3827708 Fax: 3828872
• **SWEDEN** Stockholm Tel: (8)702 97 70 Fax: (8)640 47 36
• **TAIWAN, ROC** Taipei Tel: (2)5461260 Fax: (2)7190260
• **UK, EIRE, DENMARK, FINLAND & NORWAY**
Swindon Tel: (01793) 726666 Fax: (01793) 518582
These are supported by Agents and Distributors in major countries worldwide.

This publication is issued to provide information only which (unless agreed by the Company in writing) may not be used, applied or reproduced for any purpose nor form part of any order or contract nor to be regarded as a representation relating to the products or services concerned. No warranty or guarantee express or implied is made regarding the capability, performance or suitability of any product or service. The Company reserves the right to alter without prior notice the specification, design or price of any product or service. Information concerning possible methods of use is provided as a guide only and does not constitute any guarantee that such methods of use will be satisfactory in a specific piece of equipment. It is the user's responsibility to fully determine the performance and suitability of any equipment using such information and to ensure that any publication or data used is up to date and has not been superseded. These products are not suitable for use in any medical products whose failure to perform may result in significant injury or death to the user. All products and materials are sold and services provided subject to the Company's conditions of sale, which are available on request.
All brand names and product names used in this publication are trademarks, registered trademarks or tradenames of their respective owners.