Introduction to ARM thumb

Joe Lemieux - September 24, 2003

Many complex functions that are performed in a single, albeit slow, instruction in a CISC processor may require multiple instructions in a RISC. To reduce the memory costs of these extra instructions, consider a processor with Thumb.

Many of today's most popular 32-bit microcontrollers use RISC technology. Unlike CISC processors, RISC engines generally execute each instruction in a single clock cycle, which typically results in faster execution than on a CISC processor with the same clock speed.

Increased performance, however, comes at a price: a RISC processor typically needs more memory than a CISC does to store the same program. Many of the complex functions performed in a single, albeit slow, instruction in a CISC processor may require two, three, or more simpler instructions in a RISC.

Except in the most speed-critical of embedded devices, the cost of memory is much more critical than the execution speed of the processor. To reduce memory requirements and, thereby, cost, Advanced RISC Machines (ARM) created the Thumb instruction set as an option for their RISC processor cores. The most well-known chip that includes the Thumb instruction set is the ARM7TDMI. The "T" in the core's full name specifies Thumb.

Size matters

The Thumb instruction set consists of 16-bit instructions that act as a compact shorthand for a subset of the 32-bit instructions of the standard ARM. Every Thumb instruction could instead be executed via the equivalent 32-bit ARM instruction. However, not all ARM instructions are available in the Thumb subset; for example, there's no way to access status or coprocessor registers. Also, some functions that can be accomplished in a single ARM instruction can only be simulated with a sequence of Thumb instructions.

At this point, you may ask why have two instruction sets in the same CPU? But really the ARM contains only one instruction set: the 32-bit set. When it's operating in the *Thumb state*, the processor simply expands the smaller shorthand instructions fetched from memory into their 32-bit equivalents.

The difference between two equivalent instructions lies in how the instructions are fetched and interpreted prior to execution, not in how they function. Since the expansion from 16-bit to 32-bit instruction is accomplished via dedicated hardware within the chip, it doesn't slow execution even a bit. But the narrower 16-bit instructions do offer memory advantages.

The Thumb instruction set provides most of the functionality required in a typical application. Arithmetic and logical operations, load/store data movements, and conditional and unconditional branches are supported. Based upon the available instruction set, any code written in C could be executed successfully in Thumb state. However, device drivers and exception handlers must often be written at least partly in ARM state.

Register sets

When operating in the 16-bit Thumb state, the application encounters a slightly different set of registers. Figure 1 compares the programmer's model in that state to the same model in the 32-bit *ARM state*.

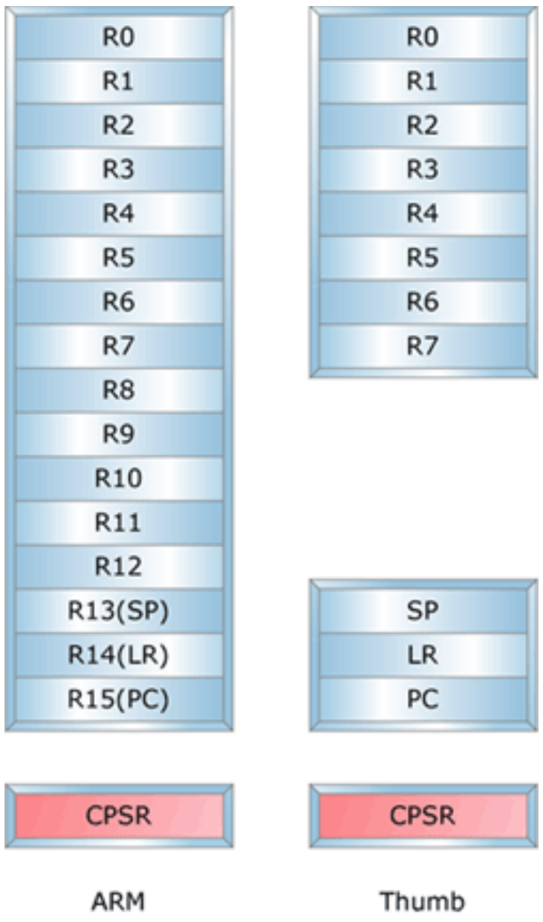

Figure 1 ARM vs. Thumb programmer's models

In the ARM state, 17 registers are visible in user mode. One additional register—a saved copy of Current Program Status Register (**CPSR**) that's called **SPSR** (Saved Program Status Register)—is for exception mode only.

Notice that the 12 registers accessible in Thumb state are exactly the same physical 32-bit registers accessible in ARM state. Thus data can be passed between software running in the ARM state and software running in the Thumb state via registers R0 through R7. This is done frequently in actual applications.

The biggest register difference involves the **SP** register. The Thumb state has unique stack mnemonics (**PUSH, POP**) that don't exist in the ARM state. These instructions assume the existence of a stack pointer, for which **R13** is used. They translate into load and store instructions in the ARM state.

The **CPSR** register holds the processor mode (user or exception flag), interrupt mask bits, condition codes, and Thumb status bit. The Thumb status bit (**T**) indicates the processor's current state: 0 for ARM state (default) or 1 for Thumb. Although other bits in the CPSR may be modified in software, it's dangerous to write to **T** directly; the results of an improper state change are unpredictable.

Thumbs up

There are several ways to enter or leave the Thumb state properly. The usual method is via the Branch and Exchange (**BX**) instruction. See also Branch, Link, and Exchange (**BLX**) if you're using an ARM with version 5 architecture. During the branch, the CPU examines the least significant bit (LSb) of the destination address to determine the new state. Since all ARM instructions will align themselves on either a 32- or 16-bit boundary, the LSB of the address is not used in the branch directly. However, if the LSB is 1 when branching from ARM state, the processor switches to Thumb state before it begins executing from the new address; if 0 when branching from Thumb state, back to ARM state it goes.

Listing 1: How to change into Thumb state, then back

```
  mov  R0,#5         ;Argument to function is in R0
  add  R1,PC,#1    ;Load address of SUB_BRANCH, Set for THUMB by
                   adding 1

  BX   R1          ;R1 contains address of SUB_BRANCH+1
;Assembler-specific instruction to switch to Thumb
SUB_BRANCH:
  BL   thumb_sub   ;Must be in a space of +/- 4 MB


  add  R1,#7       ;Point to SUB_RETURN with bit 0 clear


  BX   R1


;Assembler-specific instruction to switch to ARM
SUB_RETURN:
```

Listing 1 shows one example (not the only one) of using the **BX** instruction to go from ARM to Thumb state and back. This example first switches to Thumb state, then calls a subroutine that was written in Thumb code. Upon return from the subroutine, the system again switches back to ARM state; though this assumes that **R1** is preserved by the subroutine. The **PC** always contains the address of the instruction that is being executed plus 8 (which happens to be **SUB_BRANCH**). The Thumb **BL** instruction actually resolves into two instructions, so 8 bytes are used between **SUB_BRANCH** and **SUB_RETURN**.

When an exception occurs, the processor automatically begins executing in ARM state at the address of the exception vector. So another way to change state is to place your 32-bit code in an exception handler. If the CPU is running in Thumb state when that exception occurs, you can count on it being in ARM state within the handler. If desired, you can have the exception handler put the CPU into Thumb state via a branch.

The final way to change the state is via a return from exception. When returning from the processor's exception mode, the saved value of **T** in the **SPSR** register is used to restore the state. This bit can be used, for example, by an operating system to manually restart a task in the Thumb state—if that's how it was running previously.

Put your thumb out

The biggest reason to look for an ARM processor with the Thumb instruction set is if you need to reduce code density. In addition to reducing the total amount of memory required, you may also be able to narrow the data bus to just 16 bits. With the narrower bus, it will take two bus cycles to fetch a single 32-bit instruction; but you'll only pay that penalty in the parts of your code that can't be implemented with the Thumb instructions. And you'll still have the benefits of a powerful 32-bit RISC processor. A nifty trick indeed.

Joe Lemieux is manager of Controls and Electronics at Ricardo, Inc. He has developed embedded systems for the automotive and medical industries for over 20 years and is the author of *Programming in the OSEK/VDX Environment* (CMP Books). He can be reached at joe@osekbook.com.