

Developing a digital cellular phone using a 32-bit Microcontroller

By Oliver Gunasekara B.Eng (Hons) AMIEE
Business Manager Wireless, ARM

Abstract

This article looks at the problem faced by a design engineer when developing a digital cellular phone. A solution is provided in using a 32-bit RISC microcontroller. The article then explains how this solution can lower the system cost and power consumption of a traditional system based on an 8-bit CISC.

Constraints faced by designers

Cellular phones have now become a consumer item. This has led to far greater demand on manufacturers to reduce their costs. Each new design must offer better features and be cheaper to manufacture. Power consumption is also a primary target to reduce.

A current wireless handset contains an RF analog side, which consists of a Low Noise Amplifier (LNA) followed by a modem. The baseband (digital) side normally consists of a DSP, μ C and a mixed signal ASIC. This approach has many disadvantages, since the devices are discrete. The handset ends up with many ICs which in turn increase the cost. The size and footprint of the terminal are increased as is the power consumption. The terminal is also harder to manufacture and less reliable.

Current handsets typically have around 12 hours standby time. This figure is small due to the relatively discrete nature of current generation handsets. Future generations need to offer many more hours of battery life.

Cellular protocol stacks are not standing still. The new range of PCS services have added many new features to cellular systems. These additional features are data services, SMS (short message service) and other enhancements. On top of the cellular protocol stacks you also have the requirement to add more features to handsets. These are in the form of more advanced user interfaces. Some handsets will also require dual mode operation, where the phone can operate with different cellular protocol and possibly different frequencies. Some designers are considering adding PDA (Personal Digital Assistant) function to cellular phones (like the 9000 Communicator from Nokia). All these enhancements will require additional microcontroller performance and additional memory to hold the firmware.

How can these constraints be resolved?

32-bit RISC microcontrollers

One of the major components of any cellular phone is the microcontroller. Typically the microcontroller was a small 8- or 16-bit CISC like the Z80 or H8\300. This component was discrete and typically used to do all the user interface and control functions, such as keyboard monitoring. Designers are beginning to see the advantages of integrating the microcontroller in the baseband ASIC. This is possible if an embedded microprocessor core is available.

Currently, most digital ASICs being used in wireless applications are built with a 0.6 μ m feature size. In the next few years, 0.35 μ m based technology will allow for much higher levels of integration, which will in turn reduce the IC count and hence reduce cost and power consumption.

A 32-bit RISC controller can make a good general purpose microcontroller. It has excellent performance with most instruction executed in a single cycle. This provides ample performance for current generation products and gives scalable performance for future needs. A modern 32-bit RISC architecture can provide software compatibility between a range of products. For example: the ARM family of microprocessors from Advanced RISC Machines has products that work from DC to 200 MHz.

A modern microcontroller family like the ARM is also very simple to implement. The microprocessor is available as a core for integration into ASICs, and the core is designed to be as small as possible. This, coupled with the added advantage of on-chip debug support, results in a controller that is a good fit for embedded applications like cellular phones.

Unfortunately, there are two problems with traditional 32-bit RISC microprocessors. The first problem has to do with code size. Traditional 32-bit RISC processors, like PowerPC and MIPS, have a fixed 32-bit instruction set. Thus each instruction occupies four bytes in memory. When you compare this with a typical 8-bit CISC, your result is twice the amount of ROM required in the system to hold the firmware. This means that the system cost of a cellular phone has gone up significantly as additional memory (flash) is required. In consumer-type volume applications, like cellular phones, you can not afford this increase in cost.

The second disadvantage is running code out of narrow width memory. As the microcontroller becomes integrated into a baseband ASIC, you run into the problem of running out of pins. This, with the cost advantages of using 8-bit wide memory, results in narrow data buses. This in turn means that to fetch a 32-bit instruction from an 8-bit wide memory requires three wait states and so reduces

your performance by a factor of three. A 16-bit wide external flash device would still require one wait cycle to fetch a 32-bit instruction. The net result is that performance from your 32-bit microcontroller is significantly reduced.

The ARM7TDMI microcontroller

The ARM7TDMI is a microprocessor core designed by Advanced RISC Machines in Cambridge, England. The processor uses a three stage pipeline which allows it to execute an instruction every cycle. This allows the ARM processor to deliver much higher performance, at the same clock speeds, than other CISC microprocessors.

The ARM7TDMI is a very simple RISC processor. The core is fully 32-bit including 32-bit ALU, barrel shifter, data and address bus. Although the 4GB of address range is very rarely used in wireless applications, it does have the advantage of simplifying the decode logic by using the upper address lines as chip selects.

The main advantage of the ARM7TDMI is the fact that it has two instruction sets. The ARM7TDMI implements both the traditional 32-bit wide ARM instruction set and the new Thumb instruction set which is only 16-bit wide. The Thumb instruction set was added to remove the limitations of code density and performance from narrow memory. Effectively, the traditional 32-bit ARM instruction set was compressed into the Thumb 16-bit instruction set. Thumb instructions are then decompressed at execution time to produce a traditional 32-bit wide ARM instruction, which is then executed on the core as normal. As the ARM decode phase is relatively simple, it is possible to do the Thumb decompression on the fly without taking any additional cycles.

Example of ARM and Thumb code

Here is a simple C routine that demonstrates the differences between Thumb and ARM code. A cellular handset would typically contain around 512k bytes of microcontroller code, so the reduction in code size shown below would be significantly larger. This simple 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):

```
                CODE32                ;Directive for 32-bit (ARM)
                ;instructions
iabs    CMP      r0,#0                ;Compare r0 to zero
```

```

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

```

The Thumb assembly version is:

```

      CODE16      ;Directive for 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

```

The code sizes for both versions are shown in the figure 7.

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

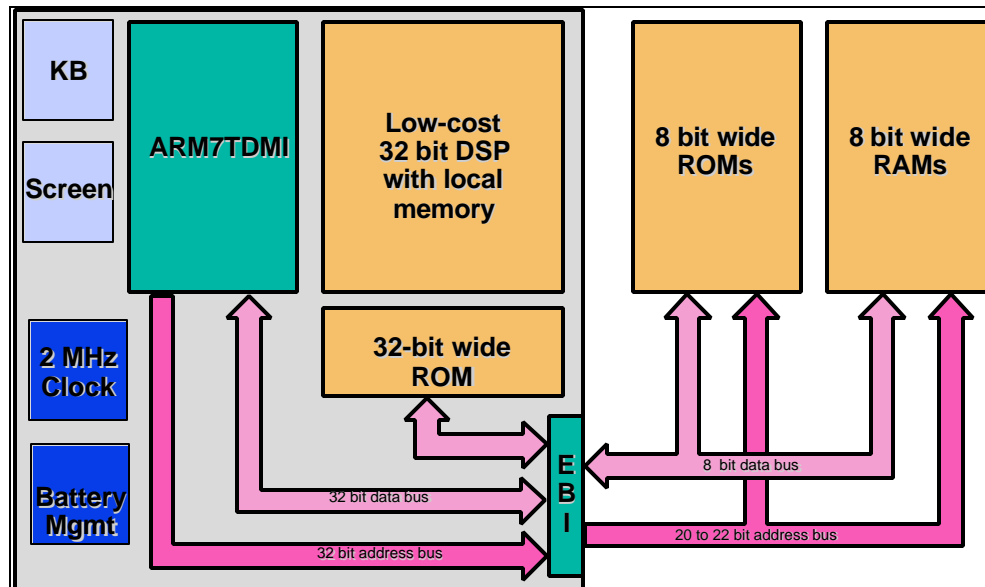
Code size for ARM and Thumb Example

Even though the Thumb assembly uses one more instruction, it still works out 33% smaller in code size. The Thumb code will also execute more quickly than ARM code on a narrow memory system, since less fetches are required to build up an instruction.

As the ARM7TDMI has both the ARM and Thumb instruction sets, the user is free to trade off performance against memory consumption. This can be done on a function by function basis. The result is a lower system cost by reducing the amount of memory required and allowing more features than existing 8 and 16-bit CISCs.

The ARM7TDMI also includes JTAG on-chip debug support. This allows a debugger to completely download code to a target board and control execution without requiring resources of the target board. No RAM, ROM, interrupts or serial ports are required. The EmbeddedICE macrocell allows hardware breakpoints and watchpoints, providing the ability to breakpoint on ROM locations.

Example Digital cellular phone



The diagram above shows an example baseband ASIC for a cellular handset. Integrated in the ASIC are the microcontroller, low cost DSP, memory and other peripherals and glue logic. As the ARM7TDMI core is only 4.9mm^2 , it only occupies a tiny percentage of the total ASIC area. 8-bit wide external memory is used to hold the Thumb instructions. Using the ARM provides a major advantage because the system power consumption can be significantly reduced when the core is clocked at a lower frequency. You also have the benefit of requiring less external memory, which allows additional features to be added to the firmware.

Comparisons with other architectures

In the development of the Thumb architecture, it was decided to work very closely with customers who were developing wireless applications. The fundamental goal of the Thumb architecture extensions was to reduce the total system cost of an end application by improving code density.

The Thumb ANSI 'C' compiler was developed early in the project life and allowed ARM to compile code for the Thumb instruction set. ARM was provided, via its customers, with a large amount of wireless 'C' source code. This was then fed into the Thumb compiler to optimize the use of the 16-bit to encode a Thumb instruction. The procedure was repeated a number of times until the optimal mix of instructions was achieved. As a result ARM has been able to evaluate real GSM, DECT and D-AMPS code from the leading wireless players. Three main components are important when benchmarking code:

Code Density

The code density is used to report how much memory is required for a given piece of high level 'C' code. This is a function of the target architecture (how wide the instruction set is, etc.) and the amount of optimization that the 'C' compiler carries out. The smaller the size the better, as this will either reduce the system cost (less memory is required) or allow more features into the same size of memory.

Performance

The performance of a microprocessor specifies at what clock speed the processors need to run to allow it to execute a given algorithm. The slower the clock rate the less power consumed and the easier it is to design with. Most cellular systems have fixed frequencies that are available to drive the microcontroller; GSM900 has 13MHz. A 32-bit RISC controller will spend most of its time in an idle mode and hence save power.

Power Consumption

The lower the power consumption of the microprocessor, the longer the batteries will last in a portable wireless product. This also allows smaller, cheaper batteries. The ARM7TDMI consumes only 1.85mW per MHz.

Exact benchmark results are client confidential, but summary information follows.

Customer 'X' code consisted of GSM protocol layer three, User Interface. This code was written in 86 'C' source files and used 150 header files. In total there were 100,699 source lines. This customer was originally using a Hitachi H8/500H and wished to compare the code size and performance of the ARM7TDMI in both ARM state and Thumb state. The code density of the ARM code over the Hitachi code was larger, but the Thumb code was 10% less. This 10% would allow for more features in the code for the same size of memory as the H8/500. The ARM7TDMI, with an 8-bit wide memory, was over three times faster than both a H8/500 and an Intel 80C188EB running at the same clock rate. With an 16-bit wide memory system the ARM7TDMI was over six times faster.

Customer 'Z' code consisted of D-AMPS protocol layers 3 and 2. (User interface and Network Management) The code contained 89 'C' source files and 32 header files. In total the code contained 150,286 source lines. Customer 'Z' wished to compare the code density of the ARM7TDMI against a Z80. The ARM7TDMI in Thumb state was 31% more compact than Z80.

The final benchmark carried out was for DECT (Digital Enhanced Cordless Telephone) cordless code. The ARM7TDMI in Thumb state requires 15% less memory than the Mitsubishi 7702 and 25% less than the 68k. On performance of a DECT algorithm, the ARM7TDMI is twice as fast as a 68k and 3 times as fast as the Mitsubishi 7702. This extra performance could be used to either reduce the clock rate of the system, and hence save power, or do more processing on

the ARM and remove the need for a low-end DSP. For the DECT example, you could clock the ARM7TDMI at 3.8MHz and still get the same performance as the 7702 at 12MHz. This would offer a significant power savings. If you wish to use the performance then one of the applications in DECT is G.721 ADPCM. In software the ARM7TDMI can do 8K samples encode and decode at 10MHz.

Summary

The complexity of designing a digital cellular phone is increasing. Performance requirements are going up, power budgets are declining, and system costs must be reduced. A 32-bit RISC, deeply embedded microcontroller is the primary route to meet these demands.

The ARM is an ideal microprocessor to use. It was the first commercially available RISC processor (1985) and has always had the design goals of keeping the part very small and simple. With the introduction of the Thumb instruction set you have the ability to trade off performance with memory requirements. Power consumption is also reduced as algorithms execute much quicker on a 32-bit RISC architecture.

ARM's business model is such that the company has 15 semiconductor partners who each manufacture and sell ARM based products. ARM itself has no manufacturing capabilities but uses its semiconductor partners listed below. Each of these 15 partners is free to make ARM based ASICs, and standard parts for internal and external use. A customer is thus able to obtain second source agreements from multiple fabs.