

## **ADuCM320 Code Execution Speed**

By Eckart Hartmann

### **INTRODUCTION**

The [ADuCM320](#) contains an ARM® Cortex®-M3 processor and integrated flash and RAM for code and data. To increase the execution speed of the central processing unit (CPU), the [ADuCM320](#) also includes a cache with various modes of operation. This application note presents the modes that are of interest for user applications. This application note does not cover the specific features of the ARM Cortex-M3. Consult the documentation provided by ARM for these specific features.

The basic CPU clock speed is 80 MHz. In normal mode, the [ADuCM320](#) executes from the 64-bit wide flash, fetching 64 bits of instruction code per four clock cycles. If a 64-bit fetch contains four 16-bit instructions, the CPU can execute these four instructions in four clock cycles, resulting in an effective speed of 80 million CPU instructions per second (raw MIPS). If, however, a 64-bit fetch contains two 32-bit instructions, the CPU can only execute these two instructions in four clock cycles because it must wait for the next fetch to complete before it can continue. In this case, the effective execution speed is 40 MIPS. For a mixture of 16-bit and 32-bit instructions, the effective speed is somewhere between these numbers, and for typical code, it is around 60 MIPS.

Multiply and divide instructions and memory access instructions take more than one CPU clock cycle to execute and, in this case, the 64-bit look ahead buffer, which is included in the [ADuCM320](#), has time to fill and speed up subsequent instructions. The details of this scenario are very complex, depending on the exact instruction sequences and on the 16-bit alignments. It is advised to ignore these details and to accept them as an extra safety margin. Interaction with data RAM in this mode is very efficient because the RAM is on a different bus and can work in parallel with the flash.

To increase execution speed, a cache is provided and is enabled by default. After the code has executed and is in the cache, it runs at 80 MIPS, regardless of the instruction size. As new code is executed, old code is overwritten and the code that is no longer in the cache runs again at the slower speed. For typical code, however, an effective speed of between 70 MIPS and 80 MIPS is achieved.

In some systems, higher execution speed can be achieved by placing critical code in the RAM and executing from there, because RAM is typically much faster than flash. However, on the [ADuCM320](#), executing from the RAM is actually slower than executing from the flash, even without cache.

For the fastest execution, an L1 cache mode is provided. In L1 cache mode, half of the existing RAM is placed in a mode where it is connected to the CPU instruction bus so that code can execute from it at full speed (80 MIPS). A minor disadvantage is that L1 cache mode reduces the RAM available for data. Note that, when executing from L1 cache, C language data must not be in the L1 area of the RAM to avoid contention between the buses. For both the code in flash and the code in L1 cache, use the non L1 area of the RAM for C language variables and data. Use the spare capacity of the L1 area of the RAM for bulk or seldom used data. The details of using the spare capacity of the L1 area must be analyzed on a case by case basis.

This application note describes the implementation of the methods described in the preceding paragraphs. Note, however, that the methods described are those that make use of typical compiler tools to handle most of the complications. Other methods are possible, but make it difficult for the code generated by the compiler tools to determine where functions and variables are located. Therefore, such methods are not recommended.

### **CACHED MODE**

Cached mode is the default mode and is used for most code. The basic settings of the compiler tools assume this mode, and no special setup is required. The performance details are as described in the Example Project section. Note, however, that the improved performance is only applicable for the particular code in the cache at the time of execution.

### **UNCACHED MODE**

If it is important to have consistent (though slower) execution speed, disable the cache mode.

The instruction cache is switched off by the following code:

```
pADI_FEE_CACHEKEY = 0xf123f456;  
pADI_FEE_CACHESSETUP = 0x0;
```

The instruction cache is switched on by the following code:

```
pADI_FEE_CACHEKEY = 0xf123f456;  
pADI_FEE_CACHESSETUP = 0x2;
```

It is not advised to repeatedly switch cached mode on and off, unless very careful control of timing is required by the application.

## L1 CACHE MODE

L1 cache mode is not easy to set up and needs careful attention to detail. First, the programmer must identify which functions are time critical and must place these functions in separate files from the files for the other code. Only the essential functions are included, because any code placed in RAM reduces the RAM available for variables and data. After the code is written, set up the tools.

### Procedure for Keil $\mu$ Vision4

To set up the Keil  $\mu$ Vision4 tools, follow these steps:

1. Click **Project/Options for Target "Target 1"** to open the window shown in Figure 1. Select **IRAM1** and set its start and size to 0x20004000 and 0x4000, respectively. This is the range for non L1 RAM. Select **IRAM2** and set its start and size to 0x10000000 and 0x4000, respectively. This is the range for the L1 RAM. Leave **IROM1** unchanged to hold the bulk of the code in flash.

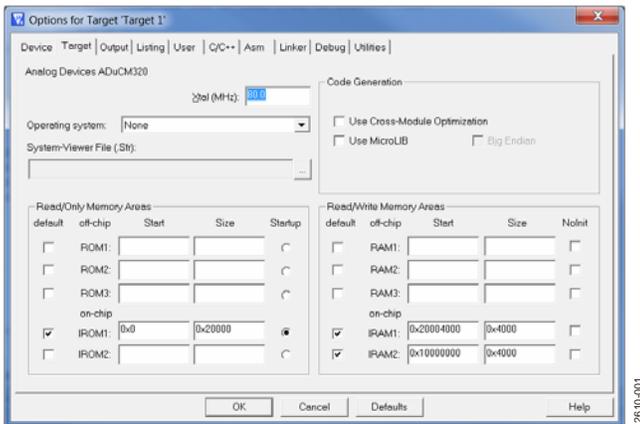


Figure 1. Selecting Memory Ranges

2. Place the code into IRAM2. In the **Project** tab, right click the file to be placed in L1 cache. In the window that opens, click the top item (**Options for File 'RamCache.s'**). In the resulting window shown in Figure 2, select **IRAM2 [0x10000000-0x10003FFF]** for **Code / Const**. For **Zero Initialized Data**, select **IRAM1 [0x20004000-0x20007FFF]** or leave as <default> to allow the linker to choose the range. In the same window, choose the range for **Other Data** or leave as <default> to allow the linker to choose the range. For each additional file destined for L1 cache, repeat this procedure to define the **Code / Const**, **Zero Initialized Data**, and **Other Data**, but make sure none of the chosen ranges overlap with each other.

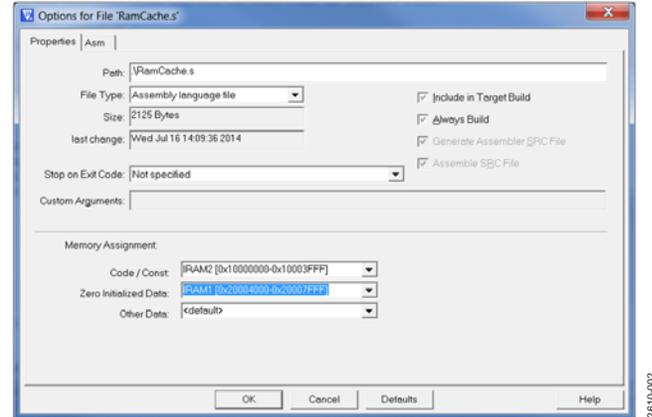


Figure 2. Assigning Memory Ranges

3. Put the code into L1 cache. In Step 2, the code was placed into IRAM2. The compiler first places the code in flash and then, during startup, before reaching the main function, main(), there is hidden code that copies the code to the L1 cache area. Unfortunately, at this stage, the L1 cache is not enabled and writing to 0x10000000 causes an exception, which is not acceptable. Therefore, the start-up file must be modified at the beginning of the start-up code to select the L1 cache mode. Modify the start-up file by adding the indented code shown.

```
Reset_Handler    PROC
EXPORT Reset_Handler [WEAK]
IMPORT __main
    movw    r1, #0x8104
    movt    r1, #0x4002
    movw    r0, #0x0000
    movt    r0, #0x5129
    str    r0, [r1]
LDR R0, = __main
BX        R0
ENDP
```

After following Step 1 through Step 3, the critical code is in L1 cache for fastest execution and all code uses the correct entry points for all functions; the location for all variables and other data and execution is as intended.

Be careful not to combine incompatible start-up code and memory map settings because this can cause the device to be locked out from debugging. If lock out does occur, use the application MDIOWSD.exe to erase the offending code and download other nonoffending code.

**Example Project**

An example project is provided with the QuickStart development system at C:\ADuCM320V1.0\code\ADuCM320\examples\Cache\Cache.uvproj. The main code of this example switches from cached mode to uncached mode and then to L1 cache mode. Each mode runs an identical short piece of test code twice with a narrow marker at the start of each test code and a wider marker at the end of each test code. The oscilloscope screenshot in Figure 3 shows how the execution times vary for the short tests running in the different modes.

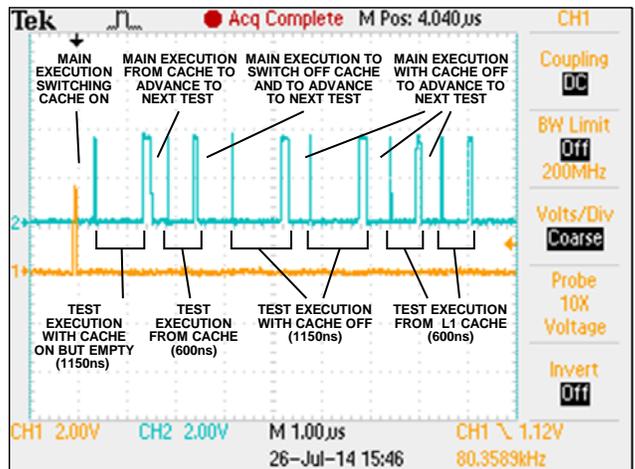


Figure 3. Example Oscilloscope Screenshot