Application Note 93

Benchmarking with ARMulator

ARM

Application Note 93
Benchmarking with ARMulator

Copyright © 2002 ARM Limited. All rights reserved.

Release information

The following changes have been made to this Application Note.

Change history

Date	Issue	Change
Mar-2002	A	First release

Proprietary notice

ARM, the ARM Powered logo, Thumb and StrongARM are registered trademarks of ARM Limited.

The ARM logo, AMBA, Angel, ARMulator, EmbeddedICE, ModelGen, Multi-ICE, ARM7TDMI, ARM9TDMI, TDMI and STRONG are trademarks of ARM Limited.

All other products, or services, mentioned herein may be trademarks of their respective owners

Confidentiality status

This document is Open Access. This document has no restriction on distribution.

Feedback on this Application Note

If you have any comments on this Application Note, please send email to errata@arm.com giving:

- the document title

- the document number

- the page number(s) to which your comments refer

- an explanation of your comments.

General suggestions for additions and improvements are also welcome.

ARM web address

http://www.arm.com

Table of Contents

1 Introduction

When developing performance-critical embedded applications it is useful to evaluate software performance prior to implementation on hardware. This can allow estimation of clock frequency and memory subsystem requirements while highlighting areas of code which need optimizing.

By using the ARMulator (ARM instruction simulator) supplied with ADS (ARM Developer Suite), it is possible to perform accurate benchmarking and gathering of code execution statistics.

Note   *ARMulator consists of C based models of ARM cores and as such cannot be guaranteed to completely reproduce the behavior of real hardware. If 100% accuracy is required, an HDL model should be used.*

This document addresses:

- The process involved in benchmarking with the ARMulator via the AXD debugger supplied with ADS version 1.2.

- The meaning and purpose of the ARMulator generated statistics.

- Special considerations when benchmarking cached cores.

The Dhrystone example benchmarking application will be used throughout as a test program for benchmarking. It is recognised that Dhrystone provides an incomplete picture of system performance, but it is useful to highlight the benchmarking features of the ARMulator.

Note   *The analysis of the benchmarking results in this document reflects the behavior of the specified core. Although much of this information is general, other cores may demonstrate different behavior.*

Note   *This document refers to ADS version 1.2 unless otherwise stated. Earlier versions of ADS, and the older Software Development Toolkit (SDT) provide broadly similar functionality.*

2   The Dhrystone benchmark

2.1   Introduction

The MIPS figures which ARM (and most of the industry) quotes are "Dhrystone VAX MIPs". The idea behind this measure is to compare the performance of a machine (in our case, an ARM system) against the performance of a reference machine. The industry adopted the VAX 11/780 as the reference 1 MIP machine.

The benchmark is calculated by measuring the number of Dhrystones per second for the system, and then dividing that figure by the number of Dhrystones per second achieved by the reference machine.

So "80 MIPS" means "80 Dhrystone VAX MIPS", which means 80 times faster than a VAX 11/780.

The reason for comparing against a reference machine is that it avoids the need to argue about differences in instruction sets. RISC processors tend to have lots of simple instructions. CISC machines like x86 and VAX tend to have more complex instructions. If you just counted the number of instructions per second of a machine directly, then machines with simple instructions would get higher instructions-per-second results, even though it would not be telling you whether it gets the job done any faster. By comparing how fast a machine gets a given piece of work done against how fast other machines get that piece of work done, the question of the different instruction sets is avoided.

There are two different versions of the Dhrystone benchmark commonly quoted:

*   Dhrystone 1.1

*   Dhrystone 2.1

ARM quotes Dhrystone 2.1 figures. The VAX 11/780 achieves 1757 Dhrystones per second.

The maximum performance of the ARM7 family is 0.9 Dhrystone VAX MIPS per MHz.

The maximum performance of the ARM9 family is 1.1 Dhrystone VAX MIPS per MHz.

These figures assume ARM code running from 32-bit wide, zero wait-state memory. If there are wait-states, or (for cores with caches) the caches are disabled, then the performance figures will be lower.

To estimate how many ARM instructions are executed per second then simply divide the frequency by the average CPI (Cycles Per Instruction) for the core.

The average CPI for the ARM7 family is about 1.9 cycles per instruction.

The average CPI for the ARM9 family is about 1.5 cycles per instruction.

2.2   Building Dhrystone

The Dhrystone application is located in the `examples\dhry` subdirectory of the ARM Developer Suite installation directory. It is recommended that a working copy of the Dhrystone directory is taken and used for benchmarking.

By default, the compilers supplied with ADS generate highly optimized code. User supplied compiler options can control the balance between code size, execution speed and debug view. In these examples our image will be built for maximum execution speed.

Compile the Dhrystone files, without linking:

```
armcc -c -Otime -W -DMSC_CLOCK dhry_1.c dhry_2.c
```

The `-Otime` switch results in code optimized for speed, rather than space (`-Ospace` is the default). The `-DMSC_CLOCK` switch results in the C library function `clock()` being used for timing measurements.

For a full description of compiler optimizations and other command line options see Chapters 2 and 3 of the ADS version 1.2 Compiler, and Libraries Guide.

The compiler produces a number of warnings that you can either ignore, or suppress using the `-W` option. The warnings are generated because the Dhrystone application is coded in Kernighan and Ritchie style C, rather than ANSI C.

For benchmarking comparisons, it is advised not to use the `-g` switch as this will default to the lowest level of compiler optimization. If you do use the `-g` switch you should also use the `-O2` switch to override the default and set the maximum optimization level. Similarly, if you are using the CodeWarrior project file, ensure you use the 'Release' variant (maximum optimization, no debug) to create the Dhrystone benchmark.

Perform the link, as follows:

```
armlink dhry_1.o dhry_2.o -o dhry.axf
```

Further details are available in the file `readme.txt`, which can be found in the `examples\dhry` directory.

Alternatively, you may wish to use the dhryansi example, which is the same benchmark written in ANSI C. This example is included with ADS and is located in the `examples\dhryansi` subdirectory of ADS installation.

2.2.1 Using CodeWarrior:

Load the Dhrystone project file `dhry.mcp` into CodeWarrior. Change the project settings to produce a release build by choosing "Release" from the project window. To build Dhrystone, click the "Make" button or press F7.

3   Performance Benchmarking

The basis for improving performance is to minimize the number of machine cycles required to perform a task.

3.1   Measuring performance

In AXD the debugger internal variable $statistics contains bus and core related statistics. This can be displayed by selecting System Views -> Debugger Internals or by typing `print $statistics` at the Command Line Interface prompt:

`$statistics`

> Can be used to output any statistics that the ARMulator has been keeping. It is a read-only variable.

In the ADW debugger (supplied with ADS version 1.1 and earlier only), the following variable is also defined:

`$statistics_inc`

> This shows the number of cycles of each type since the previous time `$statistics` or `$statistics_inc` was displayed.

Similar functionality is provided in AXD by the creation of a new reference point. This creates an additional set of counters starting from zero.

3.2   Cycle counting example: Dhrystone using the ARM7TDMI

In this example, the number of instructions executed by the main loop of the Dhrystone application and the number of cycles consumed are determined. A suitable place to break within the loop is the entry point of the function `Proc_6`, which is called once per iteration.

The compiler may choose to inline functions for improved code performance. The criteria for this decision will change as compiler options are changed.

3.2.1   Procedure

1   If you have not already done so, build the Dhrystone project as described in section 2.

2   Start AXD: If you are running Codewarrior then choose Project->Debug from the menu to start AXD and load the Dhrystone project. If you are working from the command line use `axd dhry.axf`.

3   Within AXD select Options -> Configure Target… select ARMUL as the target and click on the Configure button. Select the ARM7TDMI as the processor variant and ensure that the check box for Floating Point Emulation is cleared, then click OK. Choose OK in the configuration dialog. Click Yes when asked to reload the last image

4   Select Processor Views -> Low Level Symbols and locate `Proc_6` in the Low Level Symbols window. Right-click on it and select Locate Disassembly. Place a breakpoint on this line in the Disassembly window.

5   Click on the Go button (or press F5) to begin execution, the program will run to `main`. Click on Go again, the program will run, when prompted, request at least two runs through Dhrystone. The program will then run to the breakpoint at `Proc_6` and stop.

6   Select System Views -> Debugger Internals and click on the Statistics tab in the Debugger Internals window. Right-click in the Statistics pane and select Add New Reference Point. Enter a suitable name when prompted and click on OK.

7   Click on the Go button. When the breakpoint at Proc_6 is reached again, the contents of the reference point are updated to reflect the number of instructions and cycles consumed for one iteration of the loop.

3.2.2   Results

The results obtained from following the above procedure for one iteration of the loop are shown in the table below:

Instructions	S-cycles	N-cycles	I-cycles	C-cycles	Total
308	349	156	53	0	558

Note   *You may obtain slightly different figures, depending on the version of ADS used and the processor for which ARMulator is configured.*

3.3   Statistics for uncached cores

AXD displays a number of statistics for each core, these can be split into two categories based on the two memory access architectures; Von Neuman and Harvard. These statistics are explained below.

Refer to section 5.3 for Cached core statistics

3.3.1   Von Neuman cores e.g. ARM7TDMI:

Von Neuman cores use a single bus for both data and instruction accesses so the cycle types refer to the both types of memory access.

S-cycles   Sequential cycles. The CPU requests transfer to or from the same address, or from an address that is a word or halfword after the preceding address.

Memory Access Control signals: SEQ=1, nMREQ=0

N-cycles   Non-sequential cycles. The CPU requests transfer to or from an address that is unrelated to the address used in the preceding cycle.

Memory Access Control signals: SEQ=0, nMREQ=0

I-cycles   Internal cycles. The CPU does not require a transfer because it is performing an internal function (or running from cache).

Memory Access Control signals: SEQ=1, nMREQ=1

C-cycles   Coprocessor cycles.

Memory Access Control signals: SEQ=0, nMREQ=1

Total   The sum of the S-Cycles, N-Cycles, I-Cycles and C-Cycles.

Certain cores will generate an IS cycle statistic. This is a special I-cycle followed by sequential cycle. The timing of this cycle depends on the memory controller implementation. It can start speculatively decoding the address during the I-cycle, so it is ready to issue a fast S-cycle if one occurs. Hence so called 'merged I-S' cycles need to be treated specially by the simulation.

3.3.2 Harvard cores e.g. ARM9TDMI, ARM9E-S

Harvard cores have a separate data and instruction bus, thus allowing simultaneous data accesses and instruction fetches.

Harvard cores are not normally used in their 'raw' state due to the difficulties in designing Harvard memory systems, typically a cached variant will be used. Such cores are usually Harvard at the cache level and have a Von Neuman memory interface.

The ARMulator's default memory model for raw Harvard cores simulates dual ported RAM allowing simultaneous instruction and data accesses. The following cycle types will be generated.

Core Cycles	Total Number of ticks of core clock, this includes pipeline stalls due to interlocks and instructions that take more than 1 cycle.
ID-Cycles	Instruction bus active and data bus active.
I-Cycles	Instruction bus active, data bus idle.
Idle Cycles	True idle cycles, instruction bus idle and data bus idle.
D-Cycles	Instruction bus idle, data bus active.
Total	Total number of cycles on memory bus.

Benchmarking raw Harvard cores within ARMulator can be useful as an indication of the theoretical maximum performance that would be obtained for a cached variant if 100% cache efficiency could be achieved.

3.4 Interpreting the statistics

The statistics total generated reflects the total number of memory bus cycles that have occurred while code was executing. Provided the frequency of the memory bus is known, the execution time can thus be calculated.

From earlier we saw that a single iteration of the Dhrystone loop took 558 bus cycles to complete on an ARM7TDMI. If we assume a specific bus frequency, for example 10MHz, we can calculate the execution time for a single loop iteration:

Total iteration time is : `558 x 1 / 10,000,000 = 55.8uS`

By running Dhrystone to completion at an emulated clock speed of 10MHz, we can see confirmation of this result below:

```
Microseconds for one run through Dhrystone:    55.8
Dhrystones per Second:                      17937.2
```

The next chapter looks at real time simulations in more detail.

Note *A deliberately low clock speed has been chosen in this example as this reduces the number of Dhrystone iterations required and hence the time required for ARMulator to run Dhrystone to completion.*

Note *Knowing the results Dhrystone will return at a given clock frequency is very useful because it allows us to quickly measure (**or confirm**) the performance of a given piece of hardware. We now know that Dhrystone will complete a loop iteration in approximately 50uS at 10MHz (assuming 32bit memory and zero wait states).*

Thus Dhrystone can be used as a quick test of system configuration, for example, cache configuration, actual clock speeds etc.. Such tests are particularly useful for cached cores since Dhrystone is very small and will typically execute entirely from core cache memory

See the later section 5.3 Interpreting Cached Core Statistics.

4 Real-time simulation

The ARMulator also provides facilities for real-time simulation.

When a clock speed has been specified, the cycle counts recorded by the model can be used to calculate execution time. Also when memory characteristics are known, for any given clock speed, the appropriate waitstate cycles can be inserted.

To carry out such a simulation, you must specify:

- The clock speed of the processor.

- The type and speed of the memory attached to the processor

Refer to section 4.4 – Map files, for more information and examples.

4.1 ARMulator performance

The actual MIPs performance of ARMulator models is dependent on the performance of the host computer. As a rough guide, simpler models such as the ARM7TDMI, can achieve approximately 1 MIPS per 100MHz of PC performance. More complex models will execute at a lower speed.

4.2 Reading the simulated time

When it performs a simulation, the ARMulator keeps track of the total time elapsed. This value may be read by the simulated program or by the debugger.

4.2.1 Reading the simulated time from assembler

To read the simulated clock from an assembly language program use the semihosting SYS_CLOCK SWI.

4.2.2 Reading the simulated time from C

From C, use the standard C library function clock(). The default implementation of this function returns the number of elapsed centiseconds.

4.2.3 Reading the simulated time from the debugger

The internal variable $sys_clock records the number of centiseconds since the simulation started. To display this value, select System Views -> Debugger Internals in AXD.

4.3 ARMulator clock frequency

For the ARMulator, an unspecified clock frequency is of no consequence as ARMulator does not need a clock frequency to be able to 'execute' instructions and count cycles (for $statistics). However, your application program may sometimes need to access a clock, for example, if it contains calls to the standard C function clock()or the semihosting SYS_CLOCK SWI, so ARMulator must always be able to give clock information. It does so in the following way:

- if a clock speed has been specified, ARMulator uses that frequency value for its timing

Note   *If the system clock is set to Real-time, then $sys_clock will return actual time using the host computer's real-time clock rather than simulated execution time. This will benchmark the performance of the host computer!*

To specify a clock frequency from AXD, select Options->Configure Target…->Configure and enter a clock speed in the 'Speed:' box. Note that entering a speed without specifying units assumes Hz, for example 50 assumes 50Hz rather than 50MHz. Speeds given in kHz and GHz are also acceptable.

4.4   Map files

The default for the ARMulator is to model a system with 4GB of zero wait state 32bit memory.

Real systems are unlikely to have such an ideal memory system! Hence an alternative memory model called mapfile can be used.The mapfile memory model reads a memory description file called a map file which describes the type and speed of memory in a simulated system.

A map file defines a number of regions of attached memory, and for each region:

- the address range to which that region is mapped

- the bus width in bytes

- the access time for the memory region

ARMulator accepts a map file of any name. The file must have the extension `.map` or `.txt` for the browse facility to recognize it; however, any extension may be used if you are entering the path and filename explicitly in the map file text entry field. To specify a map file to use, choose Options->Configure Target from the menu, select ARMUL then click on the Configure button. Browse to, or type in the path and filename of the memory map file and click on OK.

To calculate the number of wait states for each possible type of memory access, the ARMulator uses the values supplied in the map file and the clock frequency specified to the model.

Note   *For cached cores, the clock frequency specified is the core clock frequency. The bus clock frequency is calculated by dividing the specified core clock frequency by the ARMulator constant - MCCFG. The derived bus clock frequency is used to calculate wait states in cached cores.*

4.4.1   Format of a map file

The format of each line is:

```
start size name width access{*} read-times write-times
```

where:

start   the start address of the memory region in hexadecimal, for example 0x80000.

size   the size of the memory region in hexadecimal, for example, 0x4000.

name   a single word that you can use to identify the memory region when memory access statistics are displayed. You can use any name. To ease readability of the memory access statistics, give a descriptive name such as `SRAM`, `ROM`, or `FLASH etc.`

width   ithe width of the data bus in bytes (that is, 1 for an 8-bit bus, 2 for a 16-bit bus, or 4 for a 32-bit bus).

access   describes the type of accesses that can be performed on this region of memory:

`r`   for read-only.

`w`   for write-only.

`rw`   for read-write.

-                 for no access. Any access will generate an abort

An asterisk (*) can be appended to the access type to describe a Thumb-based system that uses a 32-bit data bus to memory. This models a system that has a 16-bit latch to latch the upper 16 bits of data, so that a subsequent 16-bit sequential access can be fetched directly out of the latch. However, this technique is not recommended and is unnecessary for most memory systems.

read-times   describes the nonsequential and sequential read times in nanoseconds. These must be entered as the nonsequential read access time followed by a slash (/), followed by the sequential read

access time. Omitting the slash and using only one figure indicates that the nonsequential and sequential access times are the same

write-times   describes the nonsequential and sequential write times. The format is the same as that given for read times.

Example 1

```
0 80000000 RAM 4 rw 135/85 135/85
```

This describes a system with a single continuous section of RAM from 0 to 0x7FFFFFFF with a 32-bit data bus, read-write access, nonsequential access time of 135ns, and sequential access time of 85ns.

Example 2

This example describes a typical embedded system with 32KB of on-chip memory, 16-bit ROM and 32KB of external DRAM:

```
00000000 8000 SRAM  4 rw   1/1      1/1
00008000 8000 ROM   2 r  100/100 100/100
00010000 8000 DRAM  2 rw 150/100 150/100
7FFF8000 8000 DRAM2 2 rw 150/100 150/100
```

There are four regions of memory:

- A fast region from 0 to 0x7FFF with a 32-bit data bus. This is labelled SRAM.

- A slower region from 0x8000 to 0xFFFF with a 16-bit data bus. This is labelled ROM and contains the image code. It is marked as read-only.

- A region of RAM from 0x10000 to 0x17FFF that is used for image data.

- Another region of RAM from 0x7FFF8000 to 0x7FFFFFFF.

In the final hardware, the two distinct regions of the external DRAM are combined. This does not make any difference to the accuracy of the simulation.

To represent fast (no wait state) memory, the SRAM region is given access times of 1ns. In effect, this means that each access takes 1 clock cycle, because ARMulator rounds this up to the nearest clock cycle. However, specifying it as 1ns allows the same map file to be used for a number of simulations with differing clock speeds.

Note   *To ensure accurate simulations, make sure that all areas of memory likely to be accessed by the image you are simulating are described in the memory map.*

To ensure that you have described all areas of memory that the image accesses, you can define a single memory region that covers the entire address range as the last line of the map file. For example, you could add the following line to the above description:

```
00000000 80000000 Dummy 4 - 1/1 1/1
```

You can then detect if any reads or writes are occurring outside the regions of memory you expect using the "print $memory_statistics" command. Data Abort exceptions will also be generated for the previously undefined addresses covered by this range.

Note   *A dummy memory region must be the last entry in a map file as the entries in the file are processed sequentially.*

4.5   Reading the memory statistics

To read the memory statistics using AXD enter the command di (short form of dgbinternals) and pressing any key until $memstats are displayed.

See below example statistics for the DRAM region only.

```
$memstats[2]
        .name               DRAM
        .start              0x00010000
        .limit              0x00008000
        .width              0x01
        .access             0x03
        .Nread_ns           0x00000096
        .Nwrite_ns          0x00000096
        .Sread_ns           0x00000064
        .Swrite_ns          0x00000064
        .Nreads             0x00004674
        .Nwrites            0x00002E7F
        .Sreads             0x0000005B
        .Swrites            0x000000F0
        .ns                 0x0087279C
        .s                  0x00000000
```

4.6   Real-time simulation example: Dhrystone

To work through this example you must create a map file as below. Call it `test.map`.

```
00000000    80000000    RAM    4    RW    135/85    135/85
```

This describes a system that has:

- A section of memory starting at address 0x0

- 0x80000000 bytes in length

- labelled as RAM

- a 32-bit (4-byte) bus

- read and write access

- read access times of 135ns nonsequential and 85ns sequential

- write access times of 135ns nonsequential and 85ns sequential

Follow the instructions in section 4.4 to select this map file.

The association is now set up and you can run the program. If you are running CodeWarrior, ensure the Dhrystone project file is loaded and choose Project->Debug from the menu. Otherwise, launch AXD and press the "Load Image" button and choose the `dhry.axf` file.

Follow the note in section 4.3 to set the emulated clock speed to 20MHz.

Click the Go button (or press F5) to begin execution.

When requested for the number of Dhrystones, enter 50000.

When the application completes, record the number of Dhrystones per second reported. This is your performance figure.

4.6.1   Results

The results along with the reported memory statistics are shown below:

```
Microseconds for one run through Dhrystone:    84.2
Dhrystones per Second:                      11876.5


$memstats[0]
        .name           RAM
        .start          0x00000000
        .limit          0x80000000
        .Nreads         0x008370FD
        .Nwrites        0x00204627
        .Sreads         0x014BE20B
        .Swrites        0x00126BA5
        .ns             0x35F349D8
        .s              0x00000003
```

Note   *You may obtain slightly different figures, depending on the version of ADS in use and the processor for which the ARMulator is configured.*

4.6.2 Reducing the time required for simulation

You may be able to significantly reduce the actual time taken for a simulation by dividing the specified clock speed by a factor of ten or a hundred and multiplying the memory access times by the same factor. Take the time reported by the `clock()` function (or by the semhosting SWI operation `SYS_CLOCK`) and divide by the same factor.

This works because the simulated time is recorded internally in microseconds, but `SYS_CLOCK` only returns centiseconds. Dividing the clock speed shifts digits from the nanosecond count into the centisecond count, allowing the same level of accuracy but taking much less time to simulate.

Note   *To reduce the actual execution time, you would also have to reduce the number of interations!*

5 Benchmarking Cached Cores

5.1 Introduction

Modern processor cores can typically process instructions and data far faster than external memory systems can deliver them.

Caches and Tightly Coupled Memories (TCMs) are 2 different approaches with the aim of enhancing system performance when the external memory is slow and/or narrow compared to the core.

Caches and TCMs are small fast memories local to the core.

A cache is a compromise that takes advantage of the fact that the majority of subsequent accesses will also be from the cache. A whole line of memory locations is cached when a miss occurs. Performance gains rely on the 'locality of reference' principle which determines that most accesses will occur within a small address distance from the current instruction.

Tightly Coupled memory will be of benefit only if system code and/or data is located (copied) to the TCM. TCMs, when enabled, form part of the system memory map.

They can provide a number of common benefits:

- Increase system performance

- Reduce system power consumption by reducing the number of external memory accesses

- Increase available external bus bandwidth

Note *Caches and TCMs are generally only of benefit if external memory is slow or narrow. If fast memory is available then an uncached processor will probably be a better choice.*

Note *Cached Harvard architecture cores (i.e. cores with separate instruction and data buses) use a Harvard architecture at the cache and TCM level, with unified access to external memory. Therefore the benefit of the Harvard architecture will only be seen when the caches (or TCMs) are being accessed.*

Caches hold copies of external memory locations, normally these will be recently accessed locations. Once in the cache, these copies will automatically be used in preference to external memory.

For caches to be of benefit, these cached memory locations must be used again – in a real system this is very common, for example:

- Instruction loops

- Frequently referenced data

Cache operation is completely transparent, however some core initialization will be required to specify what external memory ranges should be cached.

A cached core usually operates in two clock domains - the slow clock (that of the bus and external memory) and the fast clock (that of the core when it is operating from cache). The fast clock frequency is that which is specified to ARMulator in the Configure Target… dialog.

The actual clock to the core will sourced from a combination of the two clocks – this can be considered as the pipeline clock. This clock can only advance when there are instructions to be executed, therefore for some core operations, for example synchronising from the core clock to the bus clock, the pipeline clock will not advance.

Fast core clock

Slow bus clock

Pipeline clock

ARMulator includes models of ARM's cached cores, to allow easy benchmarking and comparisons between cores.

For cores with multiple clock domains, for example cached cores and cores with TCMs, ARMulator will generate statistics for both domains, where:

- F Clock cycles - are the fast clock 'pulses' only of the core pipeline clock.

- Total cycles - are the total number of Core cycles - are clock ticks to the core itself i.e. the core pipeline clock

- bus cycles

Note   *Within ARMulator the ratio of Fast core clock to Bus clock is held in the configuration variable MCCFG as described in the following section. This means that only a synchronous relationship can be simulated.*

5.1.1   Non deterministic behavior

Consider the instruction: `LDR r0, [r1]`

For an uncached processor (such as the ARM7TDMI) operating from perfect memory, the number of cycles to execute a particular instruction is predictable. However, for a cached core this is not so, and there can be many factors affecting the time an instruction takes to execute. For example:

- Is the instruction cached?

- Is the address contained in r1 cached?

- Is the write buffer draining?

- If the processor has an MMU, does the instruction fetch cause a TLB miss to occur? Does the data access cause a TLB miss to occur?

- If a cache eviction occurs, did the old cache line contain dirty data?

The situation is made more complex because ARM's cached cores also support streaming - whereby during cache line fills, information is made available to the core at the same time as it is written into the cache. Whether a given instruction or data access can benefit from streaming will depend on factors unrelated to the actual instruction or data access.

Advanced cores such as the ARM1020E support additional features, for example: non blocking caches, and hit under miss support.

From the above information it is clear that for cached cores individual instructions or code fragments cannot be usefully benchmarked in isolation.

5.2   ARMulator cache model

Unlike real silicon, ARMulator models of cached processors have their caches enabled by default - in order to simplify benchmarking. For cached cores with Memory Protection Units (MPU), or Memory Management Units (MMU), the Pagetable module sets up an initial configuration with the lower 128MB of memory being marked as cacheable.

The default cache configuration may be changed by editing the Pagetables section in the ARMulator configuration file `peripherals.ami`.

To control whether to include the pagetable model, find the Pagetables tag in the ARMulator configuration file, `default.ami`, and alter it as appropriate:

```
{Pagetables=Default_Pagetables
}

or

{Pagetables=No_Pagetables
}
```

Within ADS 1.2 using AXD, you can also control selection of the pagetable module as follows. Select Options->Configure Target from the menu, then select the target ARMUL then click on the Configure button. From this dialogue you can choose the MMU/PU Initialization, select either DEFAULT_PAGETABLES or NO_PAGETABLES from the drop down menu.

For cores with cache or TCM, the bus memory clock is derived from the core clock speed. The ARMulator constant MCCFG is used to calculate the bus memory clock. This constant may be changed by editing the value in the ARMuator configuration file, `default.ami`.

The ARMulator startup banner will display the relationship of the clocks, for example:

```
ARMulator ADS1.2 [Build 805]

ARM940T, 4KB I-cache, 4KB D-cache, 200.00MHz FCLK, (Physical memory, BIU), Little
endian, Semihosting, Debug Comms Channel, 66.7MHz, 4GB, Mapfile, Timer,

Profiler, Tube, Millisecond [66666.7 cycles_per_millisecond], Pagetables, IntCtrl,

Tracer, RDI Codesequences
```

```
ARM RDI 1.5.1 -> ASYNC RDI Protocol Converter ADS v1.2 [Build number 805]. Copyright (c) ARM
Limited 2001
```

In this instance, the core clock is 200MHz and the bus memory clock is 66.7MHz. From the ratio of the core clock to the memory bus clock we can deduce that MCCFG must be 3. This is the default value.

5.3   Statistics for cached cores

As mentioned earlier, ARMulator models of cores with multiple clock domains generate statistics related to both domains. Modern ARM cores have native interfaces to an ARM open bus standard called AMBA (Advanced Micro controller Bus Architecture). Such core models generate bus statistic information related to AMBA bus cycle types. AMBA comes in two 'flavours', the older ASB (AMBA System Bus) and AHB (AMBA High-performance Bus).

5.3.1   AMBA ASB statistics – for example ARM720T, ARM 940T

ARM's implementation of cores with AMBA ASB interfaces do not generate N-cycles. So when ARMulating cores with AMBA ASB interfaces, you will not see any N-cycles in `$statistics` or `$memstats`, even if your code contains branch instructions. The only cycle counts shown by the ARMulator for these cores are the two AMBA cycle types:

A cycle   Address only cycle. An address is published (speculatively), but no data is transferred.

S cycle   Sequential cycle. Data is transferred from the current address

Total   Total number of cycles on the AMBA ASB memory bus.

A non-sequential access is performed with an A-cycle followed by an S-cycle ('merged I-S' cycle). Please refer to the Technical Reference Manual for the AMBA interface description of cycle types for each core.

Note   *In ADS version 1.1 and earlier, A-cycles are shown in `$statistics` under the heading 'I-Cycle' to correspond with the ARM7TDMI cycle labelling. Under ADS version 1.2, appropriate AMBA ASB names are used.*

5.3.2

5.3.3 AMBA AHB statistics – for example ARM946E-S, ARM926E-S

4 types of transfer are possible on the AHB and these are indicated on the HTRANS signals.

Seq Continuing with a burst. The address is equal to the previous address plus the data size.

Non-Seq The start of a burst or single access. The address is unrelated to the address of the previous access.

Idle The bus master does not want to use the bus. Slaves should respond with a zero wait state OKAY response on HRESP.

Busy The bus master is in the middle of a burst, but cannot proceed to the next sequential access. Slaves must respond with a zero wait state OKAY response on HRESP.

Total Total number of cycles on the AMBA AHB memory bus.

Note *In ADS version 1.1 and earlier, Busy cycles are shown in `$statistics` under the heading 'C-Cycle' to correspond with the ARM7TDMI cycle labelling. Under ADS version 1.2, appropriate AMBA ASB names are used.*

5.4 Cache initialization

If the default cache initialization model is not used, cached models may be initialized in the same way as real silicon.

Example code to perform cache initialization for various ARM cores is supplied as standard with ADS version 1.2.

5.5 Why does uncached performance appear to be so poor?

An important point to note is that for small sequential code examples, where the cache is empty or disabled, any cached processor will perform worse than one with no cache. Cached processors will only show performance benefits compared to uncached processors with code that contains loops and/or with memory that requires wait states.

For example, the ARM940T will show the following behavior for an instruction fetch when the cache is not enabled. All the following steps are required in the worst case.

1 cycle cache miss BCLK if fastbus mode, FCLK otherwise

1 internal cycle BCLK if fastbus mode, FCLK otherwise, used for some internal decoding

Synchronization none in fastbus, max 1/2 BCLK in synchronous and 1 max BCLK in asynchronous

Write buffer drain number of BCLK cycles is dependent on AMBA interface and is system specific

1 cycle address only This takes longer than 1 cycle but is factored into either the synchronisation period or write buffer drain

1 cycle word fetch BCLK cycle to perform the word fetch

So a single word fetch when the cache is disabled will typically cost 4 Internal cycles (depending upon clock mode) followed by an S cycle. This penalty will also be seen for the first word fetch of a cache line fill.

5.6 Cached core additional statistics

In addition to the standard core and bus statistic information, the ADS ARMulator can display additional statistics relating to the cache, translation look-aside buffer (TLB), and write buffer etc. operations.

To enable verbose statistics for all models, find the Counters tag in the ARMulator configuration file `default.ami`. This is set to false by default, change this line as below:

```
Counters=True
```

Under older versions of ADS, this line is not present, however adding the line will enable verbose statistics. Add this line directly after the line setting MCCFG.

Below are example additional statistics that are available when using cached cores. These may be accessed in the usual way via the "Debugger Internals" Statistics tab. Alternatively, choose System Views->Command Line Interface, and enter the following:

```
Debug >print $statistics
$statistics             structure
      .Instructions             unsigned      0x000000000007E19B
      .Core_Cycles              unsigned      0x00000000000C6249
      .Instr TLB_Hits           unsigned      0x00000000000A8577
      .Instr TLB_Misses         unsigned      0x0000000000000001
      .Instr Cache_Hits         unsigned      0x00000000000A83C0
      .Instr Cache_Misses       unsigned      0x0000000000000192
      .Instr Cache_Fills        unsigned      0x0000000000000192
      .Instr Cache_Stalls       unsigned      0x0000000000000025
      .Data TLB_Hits            unsigned      0x0000000000024CA0
      .Data TLB_Misses          unsigned      0x0000000000000002
      .Data Cache_Read_Hits     unsigned      0x000000000001DA74
      .Data Cache_Read_Misses   unsigned      0x0000000000000067
      .Data Cache_Write_Hits    unsigned      0x00000000000140BD
      .Data Cache_Write_Misses  unsigned      0x0000000000002029
      .Data Cache_Fills         unsigned      0x0000000000000067
      .Data Cache_Stalls        unsigned      0x0000000000000006
      .WB_Stalls                unsigned      0x0000000000000A3C
      .Number of Core Clocks    unsigned      0x00000000000CACA6
      .S_Cycles                 unsigned      0x00000000000025C4
      .N_Cycles                 unsigned      0x0000000000000000
      .A_Cycles                 unsigned      0x00000000000413C9
      .C_Cycles                 unsigned      0x0000000000000000
      .Total                    unsigned      0x000000000004398D
```

This represents a number of runs through Dhrystone on an ARM920T core.

The standard statistics S-Cycles, N-Cycles, A-Cycles, C-Cycles and Total are all available.

- Instructions - indicates the number of instructions executed.

- Core-Cycles – number of clock ticks to the core (i.e. pipeline clock)

- Total – Number of bus cycles.

The following statistics are split into instruction and data events:

- TLB_Hits - Translation Lookaside Buffer (TLB) hits

- TLB_Misses - Translation Lookaside Buffer (TLB) mises

- Cache_Hits - Hits on this particular cache (will only increment if a hit was possible ie the memory address had a cacheable attribute)

- Cache_Misses - Misses on this particular cache (where a cache line fill will be instigated)

5.7   Estimating cache efficiency

The calculation:

```
Cache Efficiency = Core-Cycles / Total Bus Cycles
```

may be used to assess efficiency.

If all memory accesses hit the cache, then the pipeline clock will consist entirely of fast clock pulses. So the maximum value that can be returned by this calculation is equal to MCCFG i.e. the core:bus clock ratio. So as results tend to MCCFG, cache efficiency tends towards 100%.

A result of around one would indicate that the cached core is giving similar performance to an uncached core connected to the same external memory. Similarly, results less than 1 indicate performance worse than an uncached core.

If such low results are obtained some possible considerations would be:

- Ensure the core is correctly initialized

    -ensure appropriate memory regions are marked as being cacheable

    -ensure the appropriate clocks are being applied

- Consider reworking the design

    - perhaps use TCM

    - use a more appropriate ARM core

A calculation of percentage cache efficiency could be obtained as follows:

```
Cache Efficiency % = 100 x (Core-Cycles / (Total Bus Cycles x MCCFG))
```

5.8 Interpreting cached core statistics

Earlier we examined the statistics generated for a single iteration of the Dhrystone loop on an uncached core. Below we will look again at this loop first using a raw uncached Harvard core – the ARM9TDMI and subsequently using a cached variant – the ARM940T.

Note *A deliberately low clock speed has been chosen in this example as this reduces the number of Dhrystone iterations required and hence the time required for ARMulator to run Dhrystone to completion.*

5.8.1 ARM9TDMI, Harvard arm9 dual-ported, 10.0MHz

Below are the statistics generated for a single iteration of the Dhrystone loop. Each loop will generate the same statistics.

Instructions	Core Cycles	ID-cycles	I-cycles	Idle-cycles	D-cycles	Total
306	446	71	305	12	58	446

As before we can calculate the execution time for a loop iteration:

Total iteration time is : `446 x 1 / 10,000,000 = 44.6uS`

We can see confirmation of this calculation by the results returned when Dhrystone completes (at an emulated clock speed of 10MHz)

```
Microseconds for one run through Dhrystone:    44.6

Dhrystones per Second:                        22421.5
```

In this case we can see that Core Cycles = Total Bus Cycles. This is as expected as the system we are modelling can return data to the core without delay.

The results we have achieved here indicate the maximum we could expect for a cached variant of the ARM9TDMI (operating at the same core clock speed).

5.8.2 Cache On: ARM940T, 4kB I-cache, 4kB D-cache, 10.00MHz core clock, (Physical memory, 3.3MHz)

In this example although the core is clocked at the same speed, the physical memory is 3 times slower.

Here we can see the results for the first iteration of the loop and also the results for a later iteration.

	Instructions	Core Cycles	S-cycles	N-cycles	I-cycles	C-cycles	Total
Iteration 1	306	446	377	0	345	0	722
Iteration n	306	446	7	0	142	0	149

Performing our calculations as before, we can calculate the execution times for Iteration 1, and Iteration n.

Note *The bus clock frequency is now 1/3 of the original, so each bus cycle will take 3 times longer.*

Iteration 1 time is : `673 x 1 / 3,333,333 = 216.6uS`

Iteration n time is : `149 x 1 / 3,333,333 =  44.7us`

The results returned for many iterations of Dhrystone (at an emulated core clock speed of 10MHz) are below.

```
        Microseconds for one run through Dhrystone:   44.8
Dhrystones per Second:                    22321.0
```

For the first iteration of the loop, the loop instructions and data would not be held in the cache memory, hence the pipeline clock would contain few fast clock pulses and see many stalls.

Note   *The location of the breakpoint for the 'worst case' interation is important. In this example a breakpoint was placed on the first instruction of the Dhrystone loop rather than Proc_6. This was to ensure that a minimal amount of the loop would be held in the cache.*

For the nth iteration, the small Dhrystone loop had been executed many times and was held in cache memory. The small discrepancy between calculated loop time and the Dhrystone result can be explained due to the coarse resolution of the slow bus cycles used for the calculation.

Note   *For each loop the instructions and Core Cycles are constant as the instructions, and cycles required for their execution are not related to the state of the cache.*

Note   *We can calculate our cache efficiency for the Dhrystone benchmark for a large numer of iterations:*

```
Total Core Cycles: 27074407

Total Bus Cycles :  9034428

Cache Efficiency  :  2.9979  (MCCFG=3)

Cache Efficiency % : 100 x (27084407/(9034428 x MCCFG) = 99.93%
```

We can see that by adding a cache to the ARM9TDMI we have virtually identical performance from slow memory to that of the ARM9TDMI operating from perfect memory… at least for running Dhrystone!

5.8.3   Cache off: ARM940T, 4kB I-cache, 4kB D-cache, 10.00MHz core clock, (Physical memory, 3.3MHz)

Here we can see the same evaluation as above with the cache disabled by disabling the pageable module i.e. `pagetables=no_pagetables`

Instructions	Core Cycles	S-cycles	N-cycles	I-cycles	C-cycles	Total
306	446	505	0	1864	0	2369

Iteration time is : `2369 x 1 / 3,333,333 = 710.7uS`

The results returned for many iterations of Dhrystone (at an emulated core clock speed of 10MHz) are below.

```
Microseconds for one run through Dhrystone:  710.6

Dhrystones per Second:                    1407.3
```

Note   *For each loop the instructions and Core Cycles are constant as the instructions, and cycles required for execution are not related to the state of the cache.*

Note   *Although the cache was disabled the efficiency calculation for the Dhrystone benchmark shows interesting results:*

```
Total Core Cycles:  22508588

Total Bus Cycles : 119583853
```
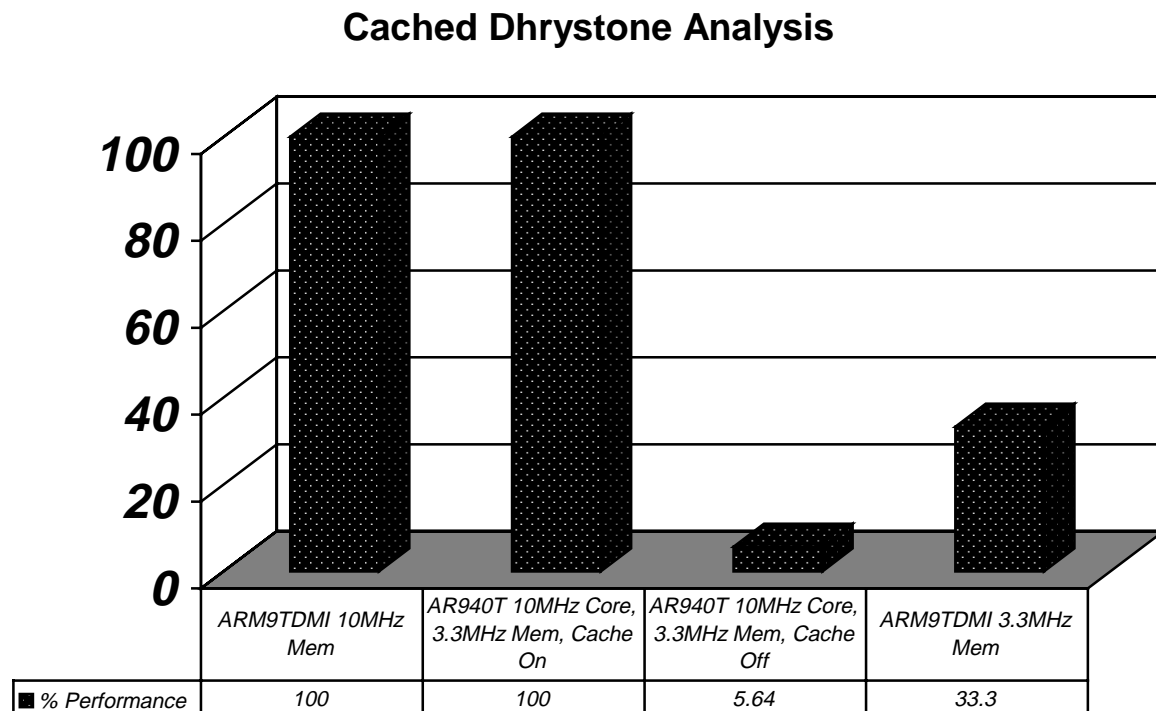
```
Efficiency    :  0.188

Efficiency % : 100 x (22508588/(119583853 x MCCFG) = 5.64%
```

Thus we can see that a cached core with its cache disabled demonstrates performance much worse than for an uncached core operating from similar memory.

5.8.4 Summary of cached core performance

The graph below summarises the findings above.

Cached Dhrystone Analysis

	ARM9TDMI 10MHz Mem	AR940T 10MHz Core, 3.3MHz Mem, Cache On	AR940T 10MHz Core, 3.3MHz Mem, Cache Off	ARM9TDMI 3.3MHz Mem
■ % Performance	100	100	5.64	33.3

5.9 Tightly coupled memories

Tightly coupled memories (TCMs) are an alternative approach to caches. When TCMs are enabled they will occupy a specific location in the system memory map.

Unlike caches, system software must be specifically written to take advantage of TCMs. A typical system might copy time critical code to the TCMs, e.g. interrupt handlers, similarly frequently changing data would also be referenced from the TCMs – for example stack access could be located there.

There are two main benefits offered by TCMs:

- Cached cores generally exhibit non-deterministic performance that may be problematic in certain systems, however performance from TCMs can be accurately predicted.

- For a given size, TCMs require roughly half the silicon area.

5.9.1 Tightly coupled memories and the ARM966E-S

Some cores, such as the ARM966E-S have TCMs. In the ARM966E-S, the I-TCM and D-TCM regions are mapped to the first and second 64Mb address ranges respectively. The TCM memory is aliased multiple times within the range.

To enable the TCMs in the ARM966E-S model, find the following entry in `peripherals.ami`:

```
IRAM=No
DRAM=No
```

And change these settings to `Yes`.

When TCMs are enabled in this manner, an image may be loaded directly into TCM memory from within the debugger – this scheme simplifies benchmarking as it removes the need for relocating code and data.

Note   *The default location of the ARMulator stack is at 0x0800000 i.e. the top of the data TCM.*

Note   *Code for ARM cores will contain some literal constants within code sections, thus the data interface must have access to Instruction TCM. However penalty cycles will occur for these data accesses, therefore be sure that read write data is not located in the Instruction TCM address space in error.*

Alternatively, the TCMs may be enabled using the following code sequence and code and data section copied as desired.
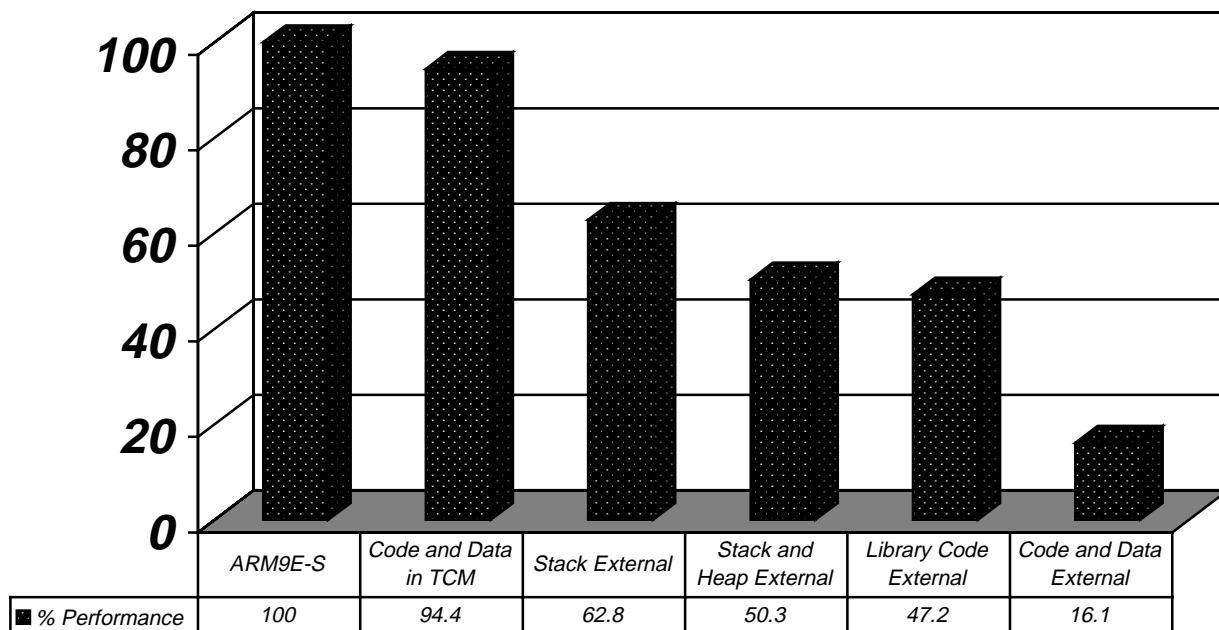
```
MRC    p15, 0, r0, c1, c0, 0   ; read CP15 register 1 into r0

ORR    r0, r0, #(0x1 <<12)     ; enable I-TCM

ORR    r0, r0, #(0x1 <<2)      ; enable D-TCM

MCR    p15, 0, r0, c1, c0, 0   ; write cp15 register 1
```

5.9.2 Drystone Analysis using TCM on ARM966E-S

The ARM966E-S contains a Harvard ARM9E-S core and TCM memory. Below is a graphical summary of the Dhrystone benchmark performed on the ARM966E-S compared with the theoretical maximum achieveable using an ARM9E-S raw Harvard core modelled with dual ported RAM.

For this example, external memory was clocked at 1/3 of the core speed i.e. MCCFG=3.

Dhrystone Analysis using TCM on ARM966E-S

	ARM9E-S	Code and Data in TCM	Stack External	Stack and Heap External	Library Code External	Code and Data External
■ % Performance	100	94.4	62.8	50.3	47.2	16.1

Note   *Even locating all code and data in TCM memory, ideal performance could not quite be duplicated. This is due to penalty cycles seen for certain accesses from TCM memory, for example data accesses to Instruction TCM will incur a cycle penalty.*

In a real system TCM memory will be a finite resource, thus care will be needed to identify the key data and code sections that will most benefit system performance by location within TCM.

6 References

ADS version 1.2 Debug Target Guide:

- Chapter 2: ARMulator Basics

- Chapter 4: ARMulator Reference

7   Appendix

7.1   ARMulator known problems

7.1.1   ARM920T model, behavior in ADS version 1.1

The ARM920T model under ADS version 1.1 defaults to operating in Fastbus mode, whereby the core clock is sourced from the bus clock. Therefore the default core clock frequency described to this model will be the bus clock.

To operate with two clock domains the model must be switched to Synchronous mode This can be achieved by setting bit 30 in coprocessor 15 register 1, either from within the debugger or by incorporating the following code sequence.

```
MRC p15, 0, r0, c1, c0, 0
BIC r0, r0, #0xc0000000
ORR r0, r0, #0x40000000
MCR p15, 0, r0, c1, c0, 0
```

Note   *If the core model is placed in Synchronous mode, the clock frequency described to the model is still be used to for the bus clock. The core clock is derived by multiplying this bus clock by the ARMulator constant MCCFG.*

Therefore if an ARM920T is modelled with a 100MHz clock, the bus clock will be 100MHz. Then if the model is changed into synchronous mode, the core clock will be 100MHz x MCCFG.

This behavior is corrected in ADS 1.2.

7.1.2   Enabling the stack tracking model in ADS version 1.1

ARMulator supports a stackuse model within ADS version 1.1 but errors in the configuration files will prevent this model operating.

Some configuration files will need modifying - these are all located in the bin directory - this can be found at the location: `<ADS_Installation_Directory>/bin`.

Locate and open the file called - default.ami (it is good practice to save a backup copy of this file before any modification is performed)

Add the following entry:

```
{ StackUse=Default_StackUse
}
```

a good location would be following the Tracer entry. Note to disable stack tracking, the entry can be modified to:

```
{ StackUse=No_StackUse
}
```

Now save this file.

Next locate and open the file called - peripheral.ami This should contain an entry as below towards the end of the file:

```
{ Default_StackUse=Stackuse
StackBase=0x80000000
StackLimit=0x70000000
}
```

This entry needs to be modified to match the stackbase and stacklimit entries in the semihost settings of the this file, otherwise stack tracking will not work. Provided you have not already modified the semihost entries then the change as below will enable stack tracking to work.

```
{ Default_StackUse=Stackuse
StackBase=0x08000000
StackLimit=0x07000000
}
```

Again - save the file.

When ARMulator is restarted you should see the Stack Tracker model appear in the ARMulator startup text. Stack information can now be viewed in the debugger internals -> Statistics view

Note   *This behavior is corrected in ADS 1.2*

Note   *Using the stack tracking feature of ARMulator slows execution speed considerably.*