☛Hands-on 1

An Introduction to the ARM Project Manager

Creating a Project using the Development Environment

Start the ARM development environment using the mouse to select *Start->Programs->ARM Developer Studio v1.01/CodeWarrior for ARM Developer Suite*. Create a new project: from *File menu* select *New* and then choose *Project* in the dialog box. In the dialog box, explore the options available and then select *ARM Executable Image*, enter the Project Name as *HelloWorld* and the project location as *C:\ArmCourse\helloworld*. Use the Add Files command from the Project menu to add the file hw.s from the *C:\ArmCourse*\source directory which contains all of the source code files supplied for these hands-on sessions.

The Project Window

The project is represented in the project manager as a project window. Although it is possible to have multiple projects open, all menu commands on the project menu apply to the currently selected project (the foreground project window). Examine the contents of this window, it can be used to view source dependencies for more complex projects.

Building and Executing The Project

Building the project involves running the assembler, compiler and linker as appropriate. The order in which to run these is governed by the project dependencies and is handled automatically by the Project Manager.

Select the project build mode by clicking on Debug, Release or Debugrel in the project window toolbar, and then press the make button to build the project. (Debug mode includes additional symbols to aid debugging and is unoptimised). Once the build has completed the program can be executed using the run button - the ARM Debugger (AXD) is then invoked and “Hello World” is displayed in the Console Window (you may need to select Processor View->Console to see this). Finally, close the debugger.

The Toolbar and On-line Help

The most frequently used commands from the ARM Project Manager menus are available on the toolbar just below the menu-bar. The function of some of these buttons is obvious, such as the cut button showing a pair of scissors or the file-save button showing a floppy disk. To find the meaning of the others, hold the mouse cursor over the button for a few seconds until a message box appears. A more extensive range of commands are available from the menus.

Full details of how to use most of the functions of the Project Manager can be found in the on-line help system which is accessed through the help menu. Context specific help, if available can be obtained by pressing the ‘F1’ function key.

The complete set of manuals for using ADS should be installed on your system. You may wish to refer to these in the later exercises, so locate them now.

☛Hands-on 2

ARM Assembly Language

HexOut - A Simple Assembly Language Routine

Use the *File* menu to open the supplied hex_out file and then examine the code that it contains. HexOut is a useful routine for dumping the contents of a register to the display in hexadecimal (base 16) notation. Build a project around hex_out. When run, it should display the result 1234ABCD.

Take a copy of the program and modify it to output the contents of r1 in binary format. For the value loaded into r1 in the example program you should get: 00010010001101001010101111001101.

TextOut

It is often useful to be able to output a text string without having to set up a separate data area for the text. A call should look like:

```
BL  TextOut
=   “Test string”, &0a, &0d, 0
ALIGN
..                    ;return to here
```

The issue here is that the return from the subroutine must not go directly to the value put in the link register by the call, since this would land the program in the text string. The supplied *text_out.s* file contains the full code for the TextOut routine and a suitable test harness. Create a project for this program, examine the file and understand how it works. Then build and execute the program.

RegDump

Using code from HexOut and TextOut write a program to dump the ARM registers in hexadecimal with formatting such as:

```
r0 = 12345678
r1 = 9ABCDEF0
```

Try to save the registers you need to work with before they are changed, for instance by saving them near the code using PC-relative addressing.

MemDump

If you still have time to spare, use HexOut as the basis of a program to display the contents of an area of memory. You should make use of the load/store multiple instructions for improved performance.

☛Hands-on 3

Compiling ARM C Programs

Hello World

HelloWorld is the classic example used as an introduction to a new programming language or environment. Here it will be used as an introduction to writing C programs within the ARM Project Manager.

Start the ARM development environment and create a new project in the *C:\ArmCourse\* directory called HelloWorld. Do not include any files in the project at this stage.

Create a new file (*File->New*) and then enter the following program code into it.

```
/*Hello World in C*/
#include <stdio.h>
int main(void)
{
    printf(“Hello World\n”);
    return(0)
}
```

Save the file as *C:\ArmCourse\hw.c* and then add it to the project. Finally compile the file using the Build command. The compiler will report one error which you should correct and then rebuild. The program may then be run.

Compiler Options

Compilation options can be set through *Edit->DebugRel Settings*. Create a project for the dhrystone benchmark. Include the dhry_1.c, dhry_2.c and dhry.h files from the source directory in the project. Add the C preprocessor definition MSC_CLOCK using the compiler settings window then explore the other ARM C compiler options and switch on the warnings if they are disabled. Explore the warning options. Compare the effects of the possible compiler optimization settings (Space versus Time) on the size of the object code produced by the compiler, and the execution time of the program (i.e. the performance of the Dhrystone benchmark). Repeat this for both Debug and Release modes of the compiler, and fill in your observations in the table below. (Use a number of iterations of the order of 10k to 100k).

Project Type	**Optimisation**	**Code Size**	**#DhryStones**
Debug	Space		
	Time		
Release	Space		
	Time		

☞Hands-on 4

Debugging ARM Code

Getting Started

The ARM Debugger can be used to debug programs built in debug mode using the CodeWarrior environement. It can be started in two ways:

- Using the *Debug* command from the Project menu of the Codewarrior Project Manager
- Directly from the operating system by choosing *Start->Programs->ARM Developer Suite v1.0.1->AXD Debugger.*

Four windows are displayed when the debugger is loaded:

- ARM Debugger - The parent window for all other debugger windows
- Execution Window - This displays the currently executing program
- Console Window - This allows interaction between yourself and the executing program. Anything printed by the program is displayed in this window and any input required by the program must be entered here.
- RDI LOG - This displays the low-level communication mesages between the Debugger and the target processor. Initially this window shows the startup messages of your target processor, the ARMulator here, but it could also be a PIE card or EmbeddedICE.

Executing The Code

Choose *Interleave Disassembly* from the context sensitive menu that appears when you click the right mouse button over the execution window. This toggles the display between source code only and source code interleaved with the disassembly. When interleaved, the machine instructions appear in lighter grey. You can step through your program using the following commands from the *Execute* or *Context* menu (or use the keyboard shortcuts shown on the menus).

- *Go* - Starts or continues execution of the program, halting at the next breakpoint or watchpoint.
- *Step* - Moves to the next line of the program. If only the source is displayed Step moves to the next line of source. If disassembly is interleaved with source, Step moves to the next machine instruction in the disassembly.
- *Step-In* - Follow a function call.
- *Step Out* - steps from the current function to where it was called from, immediately after the function call.

Breakpoints

A breakpoint halts the program at a specified location. To set a breakpoint, move to the location in the program where you wish to set the breakpoint and click at that position, and then choose *Toggle Breakpoint* from the *Execute* menu.

When you have created a breakpoint it appears as a red marker on the left pane of the Execute window. On the right pane, a red marker appears somewhere on the line. If you set the breakpoint on the wrong statement simply choose *Toggle Breakpoint* again to remove the breakpoint. Note: if you reload the image, the breakpoints will be retained.

Complex breakpoints, where the breakpoint only comes into force after the program has passed the specified point a set number of times, can be set by using the *Watch/Breakpoint* command from the *Execute* menu. This allows you to set a count and/or expression to determine when the breakpoint causes the program to halt.

To view the current breakpoints, choose *Breakpoints* from the *Processor View* menu. A list of breakpoints is displayed showing the filename and the location of the breakpoints within that file. You can edit breakpoints by double clicking on the breakpoint location.

Watchpoints

A watchpoint halts the program when a specified register or variable changes. To set a watchpoint when, for example, a specified local variable changes, choose *Variables* from the *Processor View* menu, highlight the variable value on which you wish to set a watchpoint and choose *Set Watchpoint* from the Execute menu. NOTE: If you set a watchpoint on a local variable, the watchpoint is lost as soon as you leave the function which uses the local variable.

Complex watchpoints which act when a specified variable or expression reaches a given value can be set by using *Watch/Breakpoint* from the *Execute* menu, and then using the *Set or Edit Watchpoint* command.

Viewing Variables, Registers and Backtraces

To view the contents of a variable or register, use the commands on the *Processor View* menu. Choosing the *Backtrace* command from the *Processor View*s menu when the program has halted at a breakpoint or watchpoint will give you a stack trace showing all of the currently active procedures.

Exercises

Basic Exercise

- Load an executable of Dhrystone into the debugger, and view the main function.
- Set a breakpoint at the start of the Proc_1 function.
- Set a watchpoint on the variable Microseconds.
- Choose Go.
- When execution halts in at the breakpoint, step out of the function and then delete the breakpoint.
- When execection halts at the watchpoint towards the end of the main program, examine the variables used in the calculation of Dhystones_Per_Second and step through the calculation.

Advanced Exercises

Use the debugger to debug the string-sort program in the file sorts.c

☛Hands-on 5

System Software: Writing a SWI Handler

The ARM assembly programs that you have written in the earlier exercises have all used Angel semihosted SWIs to perform text output to the console and to terminate the programs execution. The Angel SWIs are not executed by the armulator's model of the ARM processor, but instead are trapped by the armulator to allow it to then call the necessary target routine. In the environment here, that means a call to a windows function to perform the requested function, e.g. outputting characters to the display.

When you create a product using an ARM processor core, you may include a version of Angel in the final product or you may use a more complex RTOS. In either case, the SWI handlers will no longer be emulated, but will be real ARM code, that is executed in supervisor mode.

In this exercise you will learn how to write and install a SWI handler. The SWI handler is to perform the output of a text string declared inline with the code, immediately after the SWI instruction. This can be acheived by modifying the TextOut routine from earlier so that it is entered via a SWI, not a branch-and-link instruction. Note that this exercise requires you to perform the rather unusual act of calling a SWI (to Angel to output a character) from within a SWI handler. Your SWI dispatch code in the handler should of course be rentrant. A skeleton source file to get you started can be found usual source directory as swi.s.

Writing a SWI handler

Recall that the SWI instruction includes the SWI number in its least significant bits. Your handler should trap swi number 0xff and pass on all others to any previously installed SWI handler. Your handler must therofore:

- store the previous SWI vector when you install your handler;
- check the SWI number on entry to your handler, chaining to the old handler if it's not 0xff;
- preserve the user mode registers;
- preserve the supervisor mode stack pointer;
- return to usermode after handling the SWI instruction.

Installing a SWI handler

When a SWI instruction is executed, the processor fetches from address 0x8 (SWI 'vector' address). With the toolkit setup you are given, this address in memory contains a load instruction that will load an address from a vector table into the program counter. This address is the address of the handler to be executed. (The same approach is used here for all of the exception modes, hence the presence of a table of vectors).

To install your handler, you should replace the appropriate value in the vector table with the address of your handler. Your code must also remember the old vector so that a series of SWI handlers can be chained together. To determine the address of the vector table, build your (incomplete) code and examine the memory contents using the debugger.

Testing your SWI handler

A test harness for checking your SWI handler is provided in the switest.s file in the usual source directory.

☛Hands-on 6

Writing Simple Thumb Assembly Programs

Converting HelloWorld to Thumb

Examine the hw.s file used in the first exercise as an introduction to the ARM Project Manager. Most of the instructions used in this file have direct Thumb equivalents, however some do not. The load byte instruction does not support auto-indexing and the supervisor call cannot be conditionally executed. The file hwt.s contains the Thumb code equivalent of hw.s.

The two additional instruction required to compensate for the features absent from the Thumb instruction set are marked with '**T'. The ARM code size is six instructions plus 14 bytes of data making 38 bytes in all. The Thumb code size is eight instructions plus 14 bytes of data (ignoring the preamble required to switch the processor to executing Thumb instructions), making 30 bytes in all.

The key points to bear in mind when writing Thumb code which are illustrated in this example are:

- The CODE32 and CODE16 are assembler directives that instruct the compiler when to produce ARM code and when to produce Thumb code. These directives do not themselves cause any code to be generated.
- The 'BX r0' instruction instructs the processor to switch from executing ARM code to executing Thumb code, provided that r0 has been initialised properly. Note particularly that the bottom bit of r0 is set to cause the processor to execute Thumb instructions at the branch target.
- ADR can only generate word-aligned addresses, and so there is no guarantee that a location following an arbitrary number of (16 bit, half-word) Thumb instructions will be word aligned. Therefore the example program has an explicit ALIGN before the text string.

Building A Thumb Executable

In order to assemble and run Thumb code, an assembler which can generate Thumb code must be invoked and the ARMulator must emulate a 'Thumb-aware' processor core. Create a new project of type *THUMB Executable Image* around the hwt.s file. This automatically sets the compiler/assembler to *TCC/TASM* and the target processor to ARM7TDMI. Changes to the tool configuration can be made on a per-source-file basis by selecting the source file in the project window and then using the *Setting* option from the *Edit* menu. The precedure to assemble and test the Thumb code is the same as for ARM code from here onwards.

Thumb Code Exercises

Convert the hex_out.s and text_out.s programs from exercise 1 to thumb and check that they still function correctly.

Convert the RegDump program that you created in exercise 1 to thumb and check that it still functions correctly.

☞Hands-on 7

Thumb C and Cycle Counting

In order to compile your C code as Thumb, it is not necessary to make any changes to it, providing that you have written it in a portable manner and not hardcoded the size of the instructions in any of your address (pointer) calculations (instead, make good use of the C sizeof() operator). All that is required to obtain a thumb executable image is to use a compiler which can generate Thumb code must be invoked and the ARMUlator must emulate a 'Thumb-aware' processor core. These can be selected as previously described when the project is created, or on a per-file basis.

Using the ARMulator and Debugger to View Cycle Counts

A number of counts are maintained by the ARMulator including the numbers of the different types of processor cycles(non-sequential memory cycles, sequential memory cycles, internal cycles, co-processor cycles and total cycles), and the number of interrupts and the number of instructions executed.

To view the running totals of these values, when the program execution has been stopped, at a breakpoint for example, select *System View->Debugger Internals* to bring up the Debugger Internals window, and then select the *statistics* tab. The values displayed are the totals so-far for the code executed upto this point.
By selecting *Add New Reference Point* from the context sensitive menue (right click over statistics), a new line of statistics will be displayed with all counts zeroed and the statistics accumulated since the windows creation will be displayed at subsequent breaks in the program execution.

Example using Dhrystone

Examine the cycle counts for the three stages of the dhrystone program:

- Initialisation (the main procedure upto the /*Start Timer*/ comment)
- Main Loop (between the /*Start Timer*/ and /*Stop Timer*/ comments in the source code)
- Result Calculation and Display (remainder of the main procedure after the /*Stop Timer*/ comment)

Compare the effects of using ARM versus Thumb code and record your results below. Don't forget to compare the code size and the number of dhrystones per second for the ARM code versus the Thumb code.

Project Code Type	**Program Region**	**Sequential Cycles**	**Non-Sequential Cycles**	**Internal Cycles**	**Co-processor Cycles**	**Instructions Executed**
	Initialisation					
Dhrystone	Main Loop					
ARM	Results					
	Full Program					
	Initialisation					
Dhrystone	Main Loop					
Thumb	Results					
	Full Program					

☞Hands-on 8

System Software: Interrupts and preemptive Schedulers

In this exercise you will write and install an interrupt handler and use this in a simple interrupt driven two-process preemptive scheduler. The ARMulator contains models of the ARM reference peripherals including a timer and an interrupt controller that together will provide a regular source of interrupts.

Scheduler Structure

The basic scheduler consists of three files:

- sched.s - the scheduler code including the process suspend and resume functions
- timer.s - configuration of the timer and interrupt controller reference peripherals
- irq.s - skeleton for you to fill in with an interrupt handler and code to install it

Construct a project for the scheduler, copy these files from the usual source directory to your project directory along with the simpleproc.s files. and include the four files in the project. The simpleproc.s file contains two simple processes - each is a loop that never terminates, keeping a count of how many times the loop has been executed. When you have completed the code as described below, run the program and repeatedly stop and start the execution. You should see that both processes 0 and 1 get a share of the CPU time.

The code in the sched.s file is similar to that presented earlier in the lecture notes, ensure you understand what this code does before proceeding further.

The code in the timer.s file enables the interrupt controller to generate interrupts on the IRQ line when it receives interrupts from the timer. The timer counts down from TIMECNT until it reaches zero, at which point it generates an interrupt. The interrupt can be cleared by writing to the timer clear register, which is mapped into memory at address TIMERBASE+0xC . The timer then resets back to TIMECNT and starts counting down again.

Writing the Interrupt Handler

An interrupt handler is similar in structure to a SWI handler, and where the SWI handler had to check if it could handle the SWI number, the interrupt handler may have to check the source of the interrupt and take a different action for each possible source. In this example there is only one interrupt source, so there is no need for the added complexity of chaining handlers. Your interrupt handler should therefore:

- update a count of how many interrupts have occurred since the active process was scheduled
- clear the source of the interrupt by writing to the timer clear register
- if count>IRQRST then reset the count and call the NXTPROC routine to schedule a new process
- return from interrupt

Remember that your interrupt handler must preserve the contents of the user mode registers and CPSR

Installing the Interrupt Handler

Installation of the interrupt handler is a similar operation to installing a SWI handler. For performance reasons, you may want to consider using a branch rather than a vector !

☛Hands-on 9

System Software: Using SWP to implement a Semaphore

The use of a preemptive scheduler allows multiple processes to be timesliced onto a single processor core. The nature of peripherals is usually such that they should only be allocated to one process at once. The console is a good example If the two processes are both allowed to write to it whenever they desire, then the results are unpredicable. To illustrate this point, remove the simpleproc.s file from the scheduler project created in the earlier exercise and replace it with the swapproc.s file. Rebuild the project and see what happens when you run it. Can you explain whats happening ?

Semaphores

To fix the problem shown above, the resource (the console here) must be allocated to only one process at once, a task usually performed by the operating system, possibly using a semaphore mechanism.

A semaphore is a flag that indicates when the resource is in use. Its key property is that it can be examined and set in one atomic operation. This is essential to avoid the process being swapped out between looking at the flag and changing its value. The ARM SWP instruction performs this function.

Text Output Example

To use a semaphore to ensure that both of the user processes in our simple scheduler system do not interfere with oneanothers use of the console, the textout routine must be modified so that only one process is allowed to enter the critical region (the loop outputting the string) at once. Add this functionality to the swapproc.s file and check that the program functions as expected.

Acheiving similar results using a SWI system call

Add your swi handler code the scheduler by including the swi.s file in the project. Uncomment the BL INITSWI line to call your swi handler installation subroutine. Now replace the BL TextOut lines in the swapproc.s file with SWI 0xff to call your SWI handler instead. Do you get the behaviour you expect to happen when you run the program.

☛ Hands-on 10

Interfacing ‘C’ and Assembly Language

Overview

The ARM toolkit provides a comprehensive environment for developing and debugging applications that run on a variety of ARM processors. Programs can be a mixture of ‘C’ and ARM assembler modules. In order that different modules may coexist, a calling convention must be followed so that parameters and results can be passed between functions in different modules. In this exercise, you are going to explore the standard that ARM has established, APCS, by interfacing an assembly language module to the main ‘C’ program.

APCS

Under the ARM Procedure Call Standard, up to four argument words can be passed to a function in registers. *a1-a4* (*r0-r3*). This mechanism is particularly fast and efficient. If more arguments are needed, then the 5th, 6th, etc., words are passed on the stack (incurring the cost of an *STR* in the calling function and an *LDR* in the called function for each extra parameter. Results are passed either directly, or by a pointer, back in register *a1* (*r0*).

Registers *a1-a4* are free to be overwritten by the called routine, as are registers *ip (r12)* and *lr* (r14) if safe. Registers *fp, sb, sl (r9-r11)* may also be available. Other registers must be preserved (by storing on the stack if they are required by the rotuine)..

APCS allows a choice as to whether or not stack frames are used; if they are, register *fp* (*r11*) is used as a frame pointer and sufficient information is stacked on procedure entry to maintain a full stack backtrace.

Add64

Double-length arithmetic subroutines written in ‘C’ are inefficient because of the inability to specify the use of the carry flag. In this exercise you should write an assembler module to perform a double-length add, returning the state of the carry flag as an integer result (0 or 1). Create a project and add the ‘C’ test harness (*tstadd64.c*) to call the routine, and the template for the assembly language module (*add64.s*). Write the module and verify its correct behaviour. The main point of this exercise is to explore parameter passing, but you could also consider how to use the *ldm/stm* instructions to make the routine as efficient as possible.

The prototype of the add64 function is defined by:

```
typedef struct int64_struct {
  unsigned int lo;
  unsigned int hi;
} int64;

int add64(int64 *dest, int64 *src1, int64 *src2)
```

The algorithm to use is simple:

```
add lower 32-bits of src1 and src2 setting flags
add-with-carry upper 32-bits of src1 and src2
transfer carry flag to result register.
```

An example of the same routine coded in ‘C’ is given in the file *Cadd64.c*

Returning Structures

This is an example of how struct-valued functions are dealt with. The pointer to the location where the struct result is stored is passed in register *a1*, the first argument is passed in register *a2*, the second in register *a3* and so on. Consider the following code:

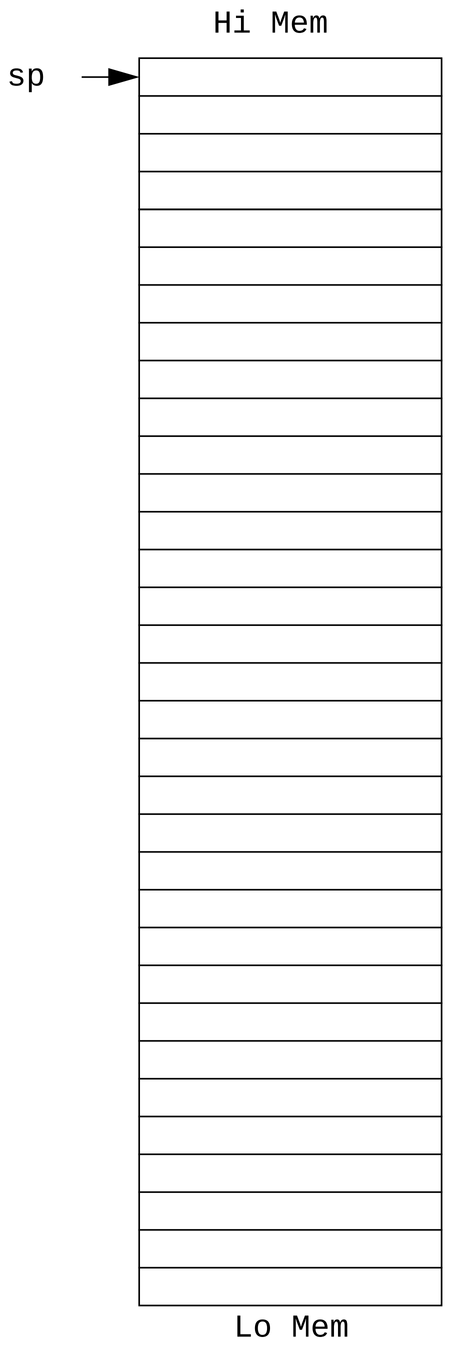
```
typedef struct two_ch_struct
{ char ch1;
  char ch2;
} two_ch;

two_ch max( two_ch a, two_ch b )
{
  return (a.ch1>b.ch1) ? a : b;
}
```

This is available within the file *two_ch.c* (which also contains code to call the function). Create a project around this file, and and the -S flag to the compiler options. Build the project and examine the assembler output in *two_ch.c.s*, ignoring any spurious error message in the project window which may say “two_ch.o is not an object or library file”. The example illustrates parameter and result passing and also how local variables are stored on the stack. Trace through the code (by hand) and fill-in the occupancy of the stack with the various variables and registers using the sheet on the next page. Ensure that you understand the code produced (you will probably need to consult the online manuals here) and then write your own function *my_max* in a separate assembler module that implements:

```
two_ch my_max( two_ch a, two_ch b )
{
  return (a.ch1>b.ch2) ? a : b;
}
```

Write a suitable test harness in ‘C’ and build and test your subroutine.

Stack Usage



☛Hands-on 11

System Software: Memory Protection

The ARM architecture defines the system coprocessor, number 15, as the usual mechanism for configuring a cache and MMU or protection unit if these are present in an ARM system. The ARMulator includes models of these features. In this exercise you will add support for memory protection to the simple scheduler used in the earlier exercises. The lectures have discussed the merits of both the MMU and the Protection unit for providing memory protection, and have shown the complexity of the former. Here we will use the model of the ARM740T which includes a simple protection unit.

Programming ARM 740T Protection Unit

The programming interface for the protection unit was presented in the lecture notes. In summary:

- regions are set up using cp15 register c6 with the region as an argument to the MCR instruction.
- protection permissions are set up using cp15 register c5
- memory protection is enabled by setting bit 0 of cp15 register c1

You should set up two regions:

- region 0 covering the entire address space with privileges set to 0b01 (no user mode access)
- region 1 starting at address range 0x10000 of size 4Kb with privileges set to 0b11 (full access to all)

Use the code skeleton in the prot.s file in the usual source directory to get you started.

Including the protection code in the project

Add the prot.s file to the scheduler project you created earlier, and activate the protection code by uncommenting the BL INITPROT line near the beginning of the file.

Since we have set region 1 starting at address 0x10000 as the only area of memory that user mode code can access, the linker must be instructed to locate the contents of our swpproc.s or simpleproc.s (or any other user code) at this address. The load.txt file in the usual source directory will configure the linker to load the sched.s, prot.s, irq.s, swi.s and timer.s files in the protected memory starting at address 0x8000 and all other files starting at address 0x10000. To use this file to control the linker, select Edit->DebugRel Settings in the Codewarrior Development Environment. Then click on ARM Linker in the linker section of the dialog box. then select the scattered link radio button and enter the path to the load.txt file.

Testing the memory protection

Build the program and execute it under the ARMulator. It should work as before. Remember to configure the ARMulator to use an ARM740T by selecting Options->Configure target in the AXD and then clicking on the middle entry, pressing configure and choosing the processor type.

Once your program runs correctly, add a load instruction to one of the user processes to load from a protected region of memory, e.g. try to load one of the exception vectors. You should get an error message about an unhandled abort exception.

☚Hands-on 12

Code Profiling

The ARM profiler, armprof displays an execution profile of a program from a profile data file generated by the debugger. The profiler displays one of two types of execution profile depending on the content of information present in the profile data:

- If only PC-sampling information is present, the profiler can display only a flat profile giving the percentage time spent in each function itself, excluding time spent in any of its children.
- If function call count information is present, the profiler can display a call graph profile which shows not only the percentage time spent in each function, but also the percentage time accounted for by calls to all children of each function, and the percentage time allocated to calls from different parents.

No special options are required at compile or link time to allow profile data to be generated, other than ensuring that the program image contains symbols , as is the linker default. Profile data generation can be toggled using *Options->Profiling->Toggle Profiling* and the generation of function call count information can be enabled from the *Image Properties* window accessed by right-clicking on the ‘image pane’. After gathering the data it must be written to a file using *Options->Profiling->Write to file*.

The profiler is a command-line tool, run as armprof datafile. Running the profiler without any arguments will display a list of the command line options. The non-sort related options are:

-parent	Displays information about the parents of each function in the profile listing. This gives information about how much time spent in each function servicing calls from each of its parents
-child	Displays information about the children of each function. The profiler displays the amount of time spent by each child performing services on behalf of the parent.
-noparent	Turns off the parent listing
-nochild	Turns off the child listing

Use the profiler and debugger to profile the Dhrystone program and answer the following:

What cumulative % of time was spent in each of main, Proc_1, Proc_6 ?

What percentage of time in Func_2 was spent in its descendents ?

☛Hands-on 13

The ARMulator

The ‘back-end’ of the ARM Toolkit Debugger that you have been using in the previous exercises is a cycle-accurate model of the ARM processors, known as the ARMulator. This exercise will explore its configuration. The ARMulator configuration database, armul.cnf in the ARM toolkit binary directory, contains a set of tag-value pairs that are read by the ARMulator at startup. The debugger console window output displayed at startup shows what configuration the ARMulator is currently using. Examine the armul.cnf file using a texteditor (e.g. Notepad). Semicolons comment out everything to the right of them, upto the end of line. .

Trace Control

Observe the *TraceMemory* entry. This enables the tracer memory model that was used in the cache modelling exercise. Find the section that begins *{ Tracer*. This contains the configuration details for the tracer memory model. Experiment with these settings. Use one of your pieces of ARM assembly code from an earlier exercise to check what is generated in the tracefile. DO NOT use dhrystone for this since it will generate enormous trace files.

Specifying a Memory Map and obtaining Performance Estimates

The default memory model used in the ARMulator is a 4GB flat model. However , if a file called armsd.map exists in the current directory when the ARMulator is started, the contents of this file will be used to configure a memory map. The format of each line of the map file is:

start size name width access read-times write-times

where:

- *start* is the address of the memory region in hexadecimal
- *size* is the size of the memory region in hexadecimal
- *name* is a single word used to identify the region when displaying memory access statistics
- *width* is the width of the data bus in bytes
- *access* describes the type of access allowed for this region, r is for read, w is for write, rw for read-write, - for no access. An asterisk (*) may be appended to this to describe a thumb based system that uses a 32-bit data bus, but which has a 16-bit latch to latch the upper 16-bits of data so that subsequent 16-bit sequential accesses may be fetched directly out of the latch
- *read-times* describes the non-sequential and sequential read times in nanoseconds separated by a /
- *write-times* describes the non-sequential and sequential write times in nanoseconds separated by a /

Example

Create the memory map: 00000000 80000000 RAM 4 RW 135/85 135/85

Then build the dhrystone program and load it into the debugger. Check that the correct memory model was loaded. If not, change the default map file using Options->Configure Target, selecting ARMUL (the middle taget environment) and then configure. You should also set the debugger configuration to use the ARMulator with a clock speed of 20MHz and then restart the debugger. Finally, run the dhrystone program for 100,000 loops. What estimated performance do you get?