1 COMP15111 Lab 1 – An Introduction to KMD

Duration: 1 lab session

1.1 Aims

To introduce you to the use of KMD. (KMD – also known as Komodo – provides a comprehensive environment for developing and debugging programs that run on a variety of ARM processors, but in this course-unit you will only be expected to use a small subset of the facilities.)

1.2 Learning Outcomes

On successful completion of this exercise a student will:
– be able to use KMD to load, assemble and run an ARM program
– be able to use KMD to modify and debug a simple ARM program
– have seen simple examples of the binary representation of ARM programs and data

1.3 Summary

You will use KMD to assemble and run a simple ARM program provided for you. You will use its debugging facilities to watch what happens when the program runs and to modify the program. If you finish this with time to spare, you can attempt part 2, where you work with a slightly more complex program used in the lectures.

Each lab exercise has the usual deadline at the end of your scheduled lab session. If you attend the lab you will, if you need it, get an automatic 7-day extension (up to the usual starting time of the lab). This is so that you can get help in the lab and then have some time to make use of that help to complete the exercise. This will be the case for all your labs for this course-unit. Remember that you must use “submit” in the usual way to show that you completed your work by the (extended) deadline. If you decide to complete the exercise after our labs shut in the evening, it is your responsibility to make sure that you can “submit” remotely.

1.4 Description

A source-level-debugging system such as KMD allows the user to
– choose particular source files (files which contain the program)
– assemble or compile a source file to ARM machine code (object code)
– run the program under controlled conditions – step through instructions one at a time or a fixed number at a time, or run until some event occurs, such as a particular instruction being executed.
– examine areas of memory and display the contents of memory locations in various formats, including as instructions (disassembling).
– examine and change processor resources, such as the contents of memory and registers.

KMD is an ARM development system, originally written in the School by a postgraduate student, C. Brej, and is freely available for students to download. It is still being maintained, so please let us know if there are any problems.

1.4.1 Getting started

The KMD system runs on Linux, so if the computer you are using is currently running Microsoft Windows use <ctrl><alt><delete> to reboot into Linux.

For this first exercise only, you must make a set of directories for COMP15111 in your home space:

```
> mkdir COMP15111
> cd COMP15111
> mkdir ex1 ex2 ex3 ex4 ex5
```

At the start of this exercise, and every subsequent COMP15111 lab exercise, you will need to copy the starting files to your lab directory. Thus, for this exercise, you will need to:

```
> cd ex1
> cp /opt/info/courses/COMP15111/Lab/ex1/* .
```

So freaking important

and then run the KMD debugger:

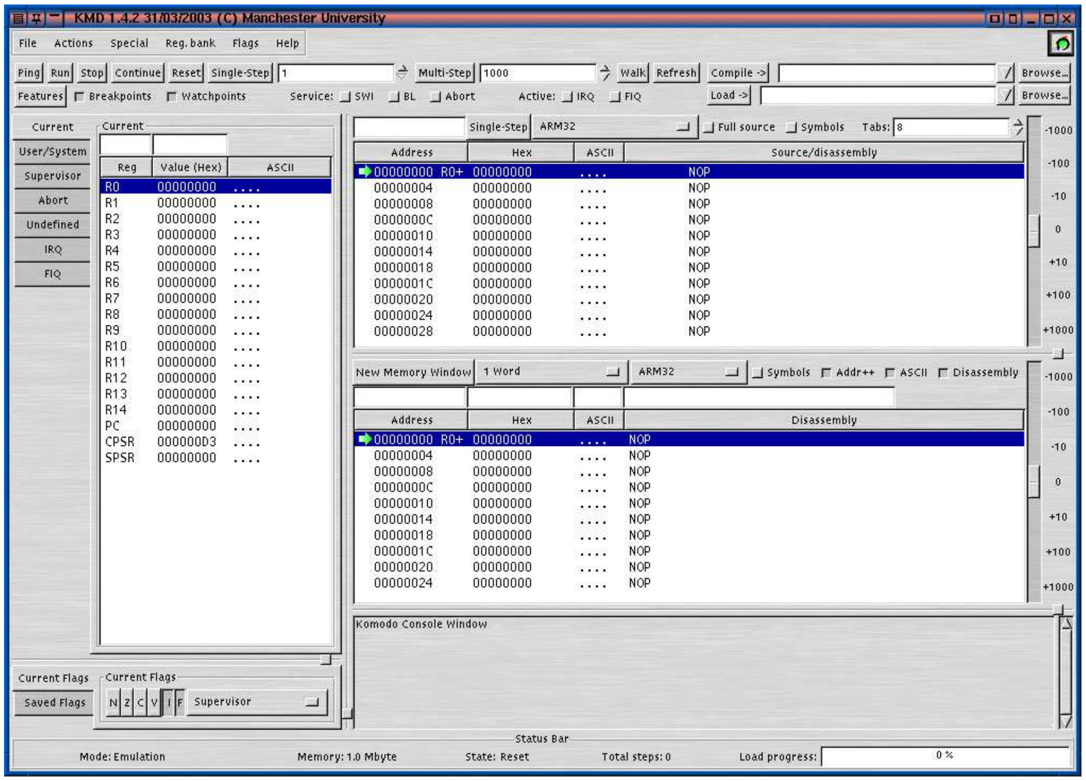
```
> start_komodo 15111&
```

Note: If you did not set correctly the path during the introductory labs you will need to use the full path to run komodo:

```
> /opt/teaching/bin/start_komodo 15111&
```

1.4.2 KMD overview

The KMD debugger should now be running and you should see something similar to that shown below. On the right there are three main panes: the Source Pane, the Memory Pane and the Log Pane. Identify them.


KMD 1.4.2 31/03/2003 (C) Manchester University
File Actions Special Reg. bank Flags Help
Ping Run Stop Continue Reset Single-Step Multi-Step Walk Refresh Compile -> Browse...
Features Breakpoints Watchpoints Service: SWI BL Abort Active: IRQ FIQ Load -> Browse...
Current User/System Supervisor Abort Undefined IRQ FIQ
Single-Step ARM32 Full source Symbols Tabs: 8
-1000 -100 -10 0 +10 +100 +1000
New Memory Window 1 Word ARM32 Symbols Addr++ ASCII Disassembly
Komodo Console Window
Current Flags Saved Flags N Z C V I F Supervisor
Status Bar

```
Current
Reg   Value (Hex)  ASCII
R0    00000000     ....
R1    00000000     ....
R2    00000000     ....
R3    00000000     ....
R4    00000000     ....
R5    00000000     ....
R6    00000000     ....
R7    00000000     ....
R8    00000000     ....
R9    00000000     ....
R10   00000000     ....
R11   00000000     ....
R12   00000000     ....
R13   00000000     ....
R14   00000000     ....
PC    00000000     ....
CPSR  000000D3     ....
SPSR  00000000     ....
```

```
Address            Hex        ASCII   Source/disassembly
00000000  R0+      00000000   ....    NOP
00000004           00000000   ....    NOP
00000008           00000000   ....    NOP
0000000C           00000000   ....    NOP
00000010           00000000   ....    NOP
00000014           00000000   ....    NOP
00000018           00000000   ....    NOP
0000001C           00000000   ....    NOP
00000020           00000000   ....    NOP
00000024           00000000   ....    NOP
00000028           00000000   ....    NOP
```

```
Address            Hex        ASCII   Disassembly
00000000  R0+      00000000   ....    NOP
00000004           00000000   ....    NOP
00000008           00000000   ....    NOP
0000000C           00000000   ....    NOP
00000010           00000000   ....    NOP
00000014           00000000   ....    NOP
00000018           00000000   ....    NOP
0000001C           00000000   ....    NOP
00000020           00000000   ....    NOP
00000024           00000000   ....    NOP
```

```
Mode: Emulation    Memory: 1.0 Mbyte    State: Reset    Total steps: 0    Load progress: 0 %
```

The **source pane** normally displays the source code corresponding to the program that is currently loaded. At the left are memory addresses followed by the contents of each location. Since no program has been loaded, this pane will be 'empty' (grey) – later it will also show your source code. The figure above shows the situation after a program has been loaded.

The **memory pane** shows the contents of a sequence of memory locations in a variety of formats: the hexadecimal values of the locations, the ASCII (character) equivalent representations of those values and also the instruction corresponding to the value stored at a memory location. A button allows byte, half word (2-byte) or full word (4-byte) representations. Don't worry if you are puzzled by these terms, all will be explained in coming lectures.

The **log pane** shows various progress, warning or error messages produced by the system, for example errors in the source code found whilst compiling.

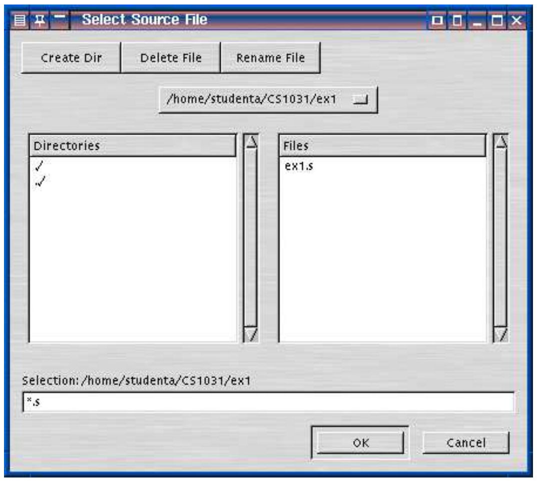
To the left of the previous panes, is a **register pane** displaying the internal ARM registers and below it the status flags (N,C,Z,V). Identify them. Selecting a register or a flag allows its contents to be changed. (Unlike the real ARM hardware, the value of `PC`/`R15` displayed by KMD always points to the instruction about to be executed – we hope that this is more useful.)

To the top of the window are various menus and control buttons that can be used to e.g. load, assemble and run programs. We will look at these in more detail below as we need to use them.

Resize the window to a convenient size. Each of the panes may be resized separately. Examine the features available. The meaning of most buttons is available via tooltips – hold the mouse cursor over the button for a few seconds until a message box appears.

Compiling (Assembling) a file:
For the first exercise, a program has already been created for you. First we need to select the program and then we need to compile it. Click on the top "Browse..." button on the same row as the "Compile ->" button. A dialogue box will appear (see picture below).

The central drop-down button shows the currently selected directory (folder) – "/home/studenta/COMP15111/ex1" in the picture. A click and hold of the mouse on the button reveals the path to the current directory and allows a selection of higher level directories to be made. The left-hand pane represents directories in the current directory; the right-hand pane shows files in the currently selected directory. Double clicking selects a directory or file. "./" denotes the current directory and "../" denotes the parent directory.
Find the directory "ex1" in your own "COMP15111" directory (not in "/opt/info/courses") and choose the file "part1.s" – assembly language source files have the suffix "**.s**". The filename will be entered automatically into the box in the main KMD window next to the "compile ->" button.
The selected program must now be compiled/assembled. Pressing the "compile ->" button will start the compilation process with output messages appearing in the log pane. Any errors in the compilation process will be marked in red giving the line number of the offending instruction and the instruction itself.



Loading a compiled/assembled file:
If the compilation succeeds, the name of the file containing the resulting machine code (in this case "part1.kmd") will automatically appear in the main KMD window next to the "load ->" button. Click the "load ->" button to load this program.

If you wanted to load a different file, you would first click on the lower "Browse..." button, on the same row as the "Load ->" button, and another dialogue box would appear, which you could use to select the file. As a short cut, previously selected files are remembered and may be accessed by holding the mouse down over the "disclosure" triangle next to the filename boxes.

After the machine code file is loaded, the compiled code is displayed in the memory pane and the corresponding source code is displayed in the source pane. The source pane is not editable – if you wish to make changes to the file you must do so in a text editor (e.g. nedit). What happens when you click on the "symbols" button in the memory pane or the source pane?

You will notice that some of the lines in the Memory and Source panes are coloured – **green** (and a green arrow) is used to indicate the next instruction to be obeyed, that the PC is pointing to, and a **blue** bar indicates the currently selected line – initially the first line, or the last line that you clicked on. Try selecting different lines and seeing how the colours change – note also how information appears in previously blank slots at the top of the pane.

Running the code:
Note the current values of registers `R0`, `R1`, `R2`, `R3`, `R4` and `PC`. To run the program press the "Reset" button and then the "Run" button (or the F5 key). Not much appears to have happened, however some of the register values should have changed. To see why, we will step through the instructions one at a time.

Stepping through the code:
First we need to be able to rerun the program. Press the "Load" button and then the "Reset" button to get the program ready to run again. Clicking on the "Single-Step" button (or the F7 key) causes a single instruction to be executed. Step through the program examining the effect of each instruction on the relevant registers: in particular note the effect on the `PC`. (You should stop when you reach the `NOP` instructions – the `SVC` instruction stops a "run", but it won't prevent you stepping on further.)

TASK: Answer these questions by editing your copy of the file "answers":

Q1) After obeying the `LDR R0` instruction, the value of `PC` is ... and of `R0` is ...
Q2) After obeying the `LDR R1` instruction, the value of `PC` is ... and of `R1` is ...
Q3) After obeying the `LDR R2` instruction, the value of `PC` is ... and of `R2` is ...

The "Multi-Step" button allows several instructions to be executed before pausing. For example, change the value in the entry box to the left-hand side of the "Multi-Step" button from 1 to 2. Reload and reset the program and step through using the "Multi-Step" button.

Try typing `PC` or `PC-8` (followed by pressing the ↩ key – also known as enter or return) in the box just at the top of the memory pane, between "Address" and "New Memory Window", (or in the similar box at the top of the source pane) and stepping through the program again – what happens now?

The "Walk" button has the same effect as the "Multi-Step" button except the display pauses for a fixed number of milliseconds between each multi-step. By default the pause duration is 1000 ms (or 1 second). Reload and reset the program and then click on "Walk" observing the behaviour.

Setting breakpoints:
Single stepping through the code is rather tedious, especially for real programs. More useful is the ability to run the program continuously until a particular line in the program is encountered. To do this, a breakpoint is inserted at that line. If no breakpoint is found, the program

will run until completion. To set a breakpoint, double-click the line at which you wish to set the breakpoint – try the `ADD` instruction. You will see the line turn **orange**. Having set the breakpoint, run the program by clicking on the "Run" button: the program should halt at the breakpoint. To remove a breakpoint, double-click the instruction again. The orange colour will disappear. More than one breakpoint may be set; to inspect the list of set breakpoints click on the "Special" menu and select the "Simple Breakpoints" menu item. A breakpoint can be temporarily deactivated (becoming grey) and reactivated (becoming orange again) in the window that the "Simple Breakpoints" button spawns by clicking on the "activate" button, or removed altogether by the "delete" button.

TASK: Answer these questions by editing your copy of "answers": (I am asking about the 'working' registers, not `PC`/`R15`):

Q4) The first time the `ADD` instruction is obeyed, the value of register ... changes from ... to ...
Q5) The first time the `SUB` instruction is obeyed, the value of register ... changes from ... to ...

As a result of running the program from the beginning until it stops at the `SVC` instruction:

Q6) the `BNE` instruction is obeyed ... times but only performs the branch ... times

Only 2 registers (other than the `PC` register) change value:

Q7) register ... counts down from ... to ...
Q8) register ... is modified to be the ... (hint: arithmetic operation) of memory locations ... and ...

Modifying registers:
Click on a register name in the register pane: the name and value of the register will be entered into the entry boxes allowing the user to modify the contents of the register (remember to press ←֓ to complete the modification). With the program halted at the `ADD` instruction, modify the contents of registers `R0` or `R1` and then continue single stepping through the program.

Memory:
The contents of memory locations can be examined (and modified). In the memory/disassembling pane, the memory is visible as plain data and as ARM instructions. The instructions are a **d**isassembly of the memory contents; this is what the ARM would do *if* it executed that location. There is less information here than in your source code because the various *labels* have been replaced by simple addresses – that's all the machine needs. In later exercises you may notice some apparent differences between the source and the disassembly; this occurs when pseudo-operations are substituted.

Change the contents of the memory locations for "jack" and "jill" and run the program again. (To do this, click on the line to select it – it should turn **blue**, and be copied into the boxes just at the top of the pane – edit the second of these boxes and press return.) The disassembly pane contents should have changed and the corresponding line in the source pane will turn **red**, to indicate that it no longer matches – do you understand what has happened? (You may need to move the blue bar, by clicking elsewhere, to be able to see this.)

1.4.3 Part 2

You are now going to make a small change to the program. Load "part1.s" into your favourite editor (e.g. nedit) and insert a new instruction `STR R0, tom` just before the `SVC` instruction. Now save the file as "part2.s" and use KMD to compile, load and run it as before.

TASK: Answer these questions by editing your copy of the file "answers":

Q9) The memory location for "tom" turns red because ...
Q10) If you reset and run the program again **without** reloading it then what happens to the value of "tom" and why: ...

1.5 Completion, Feedback and Marking Process

There is 1 mark for your answer for each of the 10 questions above.

As soon as you have completed the exercise, you need to run `submit` (making sure you are in the correct directory – COMP15111/ex1 for this exercise). `submit` records the time that you completed the exercise and archives your work.

After this, use `labprint` to generate a marking sheet (i.e. make sure you are in the correct directory – COMP15111/ex1 for this exercise, type the command `labprint` in a shell window and collect your marking sheet from a printer) and then ask a demonstrator to mark your work (you may need to add your name and seat number to a list of those queueing to be marked). (You cannot get different parts marked on different days.)

IMPORTANT – University rules say that all work must be marked within 15 days, so you must get your work marked during your next scheduled lab (unless have a good excuse e.g. that you were ill). If a lab does not get marked by this deadline, it may get removed from Arcade and, hence, will not count towards the final mark. This will be the case for all of your labs for this course-unit.