MAS (MDP ARM Simulator) Rolecks

Nicholas Caldwell, Paul Fidler, Peter Long, Geoff Martin
Cambridge University Engineering Department
Trumpington St.,
Cambridge CB2 1PZ

29th August 2007

The MDP ARM Simulator (MAS Rolecks) has been developed by the MDP team at Cambridge University as an educational tool to support an introductory course in microprocessors. The package is written in Java and offers a simple IDE that allows the editing of ARM assembler and monitoring of the assembled code under emulation. A simulated hardware interface (inc. LEDs, switches, analogue ip) is also available that allows the user to interact directly with the code. When required, the developed code can be simply loaded onto the MDP Microcontroller which allows the user to work at full speed and with real hardware and interfaces.

The software is based on the Rolecks program developed by Trek Palmer and Tim Richards at the University of Massachusetts

(http://www.cs.umass.edu/t̃rekp/cs201/rolecks.jar)

Contents

1   Introduction

What is being simulated, image of micro/memory map?

Which processors are being covered (26 or 32 bit ARM Proce3ssors)

limitations

Information about hardware (MDP Microcontroller)

2   Getting Started

Where and how to obtain a copy of the java code, web addresses etc.

2.1   Linux/ (MDP EDaL system)

from menu items, from command line

```
java -jar mas-rolecks.jar
```

2.2   Web

Need to look at java start, ...

2.3   MACX

?

2.4   Windows 98/XP/Vista

```
java -jar mas-rolecks.jar
```

3   User Interface

When the application is first started, it defaults to a two window display. The larger primary window has "The Rolecks ARM Simulator" in its titlebar; the smaller window has "Rolecks Hardware Interface" as its title. Both windows may be resized, minimised, or maximised. The Hardware Interface window cannot be closed independently of the main Simulator window; closing the Simulator window closes all Rolecks windows and exits the program.

3.1   Main Window
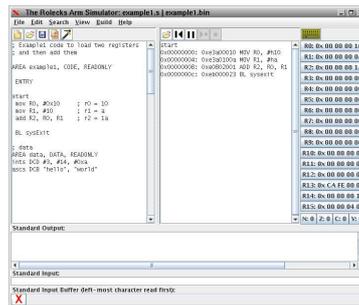
Figure 1: MAS-Rolecks main desktop window

The main window is split into five main areas shown in figure 1. The functionality associated with each area is described in sections 3.1.1 - 3.1.5.

3.1.1   Title bar & Top Menu

The title bar has the name of the programe and the filenames of the source and binary files currently loaded in the editor and binary panes.

The top menu bar has six pull-down menus: File, Edit, Search, View, Build and Help.

The File menu has the following items:

- **New Source** - clears the current editor window. **Warning:** This does not prompt the user to save the current file, so selecting this option will delete any unsaved work.

- **Load Source** - opens a standard file browser window and prompts the user to open a source file, which will be displayed in the editor pane. **N.B.** source files have a suffix .s, e.g. `example1.s`. **Warning:** This does not prompt the user to save the current file (if any), so selecting this option will delete any unsaved work.

- **Save Source** - opens a standard file browser window and prompts the user for a filename to save the text currently in the editor pane.

- **Load Binary** - opens a standard file browser window and prompts the user to open a binary (assembled) file, which will be displayed in the Binary pane. **N.B.** binary files have a suffix .bin, e.g. `example1.bin`.

- **Print Source** - not currently available. Future development.

- **Print Binary** - not currently available. Future development.

- **Quit** - this exits the entire simulator. **Warning:** This does not prompt the user to save the current file, so selecting this option will delete any unsaved work.

The Edit menu has the following items:

- **Undo** - this is a standard Undo facility which can be used to reverse changes to text in the Editor/Source pane (see below) one keystroke at a time.

- **Redo** - this is a standard Redo facility which can restore changes that have been reversed by the Undo command, again one keystroke at a time.

- **Cut** - not currently available from menu.

- **Copy** - not currently available from menu.

- **Paste** - not currently available from menu.

- **Select All** - not currently available from menu.

- **Preferences** - not currently available. Future development.

The Search menu has the following items - all currently unavailable:

- **Find**

- **Find Next**

- **Find Previous**

- **Replace**

- **Goto Line**
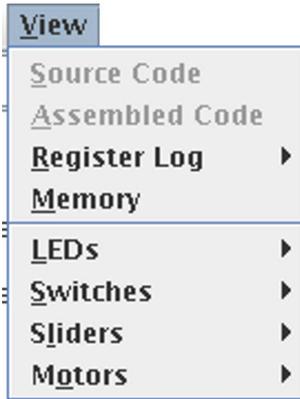
The View menu has the following items:

- **Source Code** - not currently necessary as source code visible in Editor/Source Pane.

- **Assembled Code** - not currently necessary as assembled code appears in Binary Pane.

- **Register Log** - not currently available. Future development.

- **Memory** - this makes the Memory Map window (see below) visible.

- **LEDs** - intended to allow configuration of virtual LED hardware. Not currently available. Future development.

- **Switches** - intended to allow configuration of virtual switch hardware. Not currently available. Future development.

- **Sliders** - intended to allow configuration of virtual slider hardware. Not currently available. Future development.

- **Motors** - - intended to allow configuration of virtual motors. Not currently available. Future development.
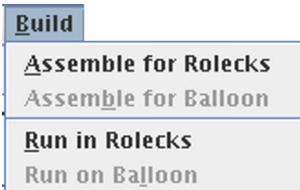
The Build menu has the following items:

- **Assemble for Rolecks** - opens a standard file browser window, enabling the user to select a source file and assemble it into a binary format compatible with this simulator. See below for further details.

- **Assemble for Balloon** - will allow assembly of a Rolecks source file into a binary format capable of being executed on a Balloon board. Not currently available. Future development.

- **Run in Rolecks** - this will execute the current loaded binary file within the Rolecks simulator.

- **Run on Balloon** - this will execute a suitably assembled binary file on a connected balloon board. Not currently available. Future development.

The Help menu has the following items:

- **Online Help** - not currently available. Future development.

- **Code Structure** - not currently available. Future development.

- **Op-codes** - not currently available. Future development.

- **About Rolecks** - displays a simple information box giving version and copyright information for the MDP ARM Simulator.

3.1.2 Editor/Source Pane
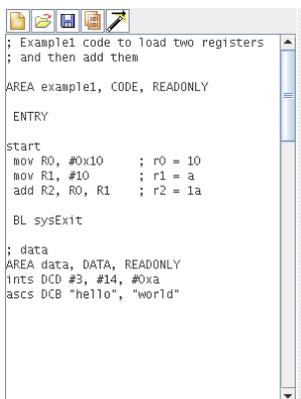
The Editor/Source Pane is the left hand pane on the Simulator window. Its size is adjustable in width with a drag handle on the dividing bar. The pane is a simple editor designed to allow the user to enter and modify programs. Above it is a small menu bar that gives the user access via graphical icons to key load, save and assemble functionality.

Editor Pane Icons

New Source Clears the current editor window. **Warning:** This does not prompt the user to save the current file, so selecting this option will delete any unsaved work.

Open File opens a standard file browser window and prompts the user to open a source file, **N.B.** source files have a suffix .s, e.g. `example1.s`

Save Source opens a standard file browser window and prompts the user for a filename to save the text currently in the editor pane.

Assemble attempts to assemble a source file. By default, it offers the user the currently saved version of the file in the editor window.

Magic (SAL-B) attempts to save the text in the editor pane, under the currently selected filename, assemble the resulting source file and, if successful, load the resultant binary into the binary pane.

Editor Functionality

The editor has basic editing functionality, accessible from control keys and the pulldown Edit menu, in particular:

CTRL-A	Select All
CTRL-C	Copy
CTRL-H	Delete backwards
CTRL-V	Paste
CTRL-X	Delete
DELETE	Delete
↑↓←→	Move cursor

3.1.3 Binary Pane
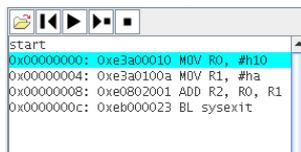
The Binary Pane displays the assembled binary file and supplemental information according to the following format of a notional memory address location (as a hexadecimal number), followed by the actual instruction in its hexadecimal assembled representation, followed by the original source instruction (minus any comments.) Any labels used in the program appear on a separate line preceding their associated instruction for clarity. Neither assembler directives nor any data areas appear in the Binary pane.

Binary Pane Icons

Open Binary File opens a standard file browser window and prompts the user to open a binary file. **N.B.** Binary files have the suffix .bin, e.g. `example1.bin`

Reset terminates the execution of a currently running binary program. It restores the Program Counter to the first line of the program, resets the values of registers R0 to R12 and R14 to zero, and clears the condition code bits.

Run executes the program from start to finish. Execution will only pause if there is input required from the user. During execution, the Run icon will be replaced with a Pause icon so that the program can be temporarily halted.

Pause freezes an executing program. It does **not** corrupt registers or condition code bits. Once a program is paused, the Pause icon is replaced with the Run icon, and the program may be restarted in single step mode or the system reset.

Single Step allows the user to step through the program one instruction at a time each time the button is pressed. The light blue highlight bar indicates the instruction that is just about to be run.

Breakpoint opens the Manage Breakpoints dialog win-

dow that allows the user to set and clear breakpoints. Single-stepping through every line of a program can quickly become tedious; breakpoints allow individual lines to be marked as being of interest. When a program has one or more active breakpoints, the user can click on the Run button (see above) and the program will execute until it reaches the first breakpoint. Instead of immediately executing the breakpoint instruction, it will pause at that instruction allowing the user to single-step at that point and beyond (to observe register value and condition code changes) or to revert to continuous execution until the next breakpoint is encountered. Active breakpoints are highlighted in the Binary Pane with yellow backgrounds. Whenever the instruction that is just about to be run coincides with an active breakpoint, the highlight bar will change to red for that instruction.
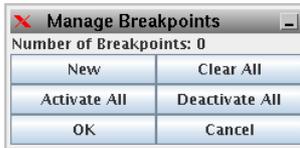
- **New** - this button allows for the setting of a new breakpoint. It raises the Add Breakpoint dialog, which requires the user to enter the hexadecimal address of the instruction where the breakpoint is to be placed. This address can be found by looking at the first column of numbers in the Binary Pane. Click OK to add the breakpoint, Cancel to refrain. Once a breakpoint is added, the Manage Breakpoints dialog will change to list the address of the breakpoint instruction, provide a checkbox (initially ticked) to show if the breakpoint is active, and a Delete button to remove the breakpoint.

- **Clear All** - triggers a Yes/No confirmation dialog that, if the response is "Yes", deletes all current breakpoints.

- **Activate All** - this makes all existing breakpoints active - as indicated by ticks appearing in the corresponding Active checkboxes for each breakpoint. To make a single breakpoint active simply check the appropriate Active checkbox.

- **Deactivate All** - - this makes all existing breakpoints inactive - as indicated by the absence of any tickmarks appearing in the corresponding Active checkboxes for each breakpoint. To make a single breakpoint inactive, simply uncheck the appropriate Active checkbox.

- **OK** - this confirms all changes made and closes the Manage Breakpoints dialog window.

- **Cancel** - this discards all changes made and closes the Manage Breakpoints dialog window.

3.1.4 Register Pane

R0: 0x 00 00 00 10
R1: 0x 00 00 00 0A
R2: 0x 00 00 00 1A
R3: 0x 00 00 00 00
R4: 0x 00 00 00 00
R5: 0x 00 00 00 00
R6: 0x 00 00 00 00
R7: 0x 00 00 00 00
R8: 0x 00 00 00 00
R9: 0x 00 00 00 00
R10: 0x 00 00 00 00
R11: 0x 00 00 00 00
R12: 0x 00 00 00 00
R13: 0x CA FE 00 00
R14: 0x 00 00 00 18
R15: 0x 00 00 04 08

N: 0	Z: 0	C: 0	V: 0

The Register pane on the right of the main window is split into two main areas:

The upper region contains the current value of each of the standard 16 ARM registers. Values can be manually changed by clicking on the register or its value, which raises a dialog window enabling a new value in hexadecimal to be entered.

The lower region displays the condition code bits:

- The **N** bit is the "negative flag" and indicates that a value is negative.

- The **Z** bit is the "zero flag" and is set when an appropriate instruction produces a zero result.

- The **C** bit is the "carry flag" but it can also be used to indicate "borrows" (from subtraction operations) and "extends" (from shift instructions).

- The **V** bit is the "overflow flag" which is set if an instruction produces a result that overflows and hence may go beyond the range of numbers that can be represented in 2's complement signed format.

The values can only currently be set or cleared during program execution, and reset to 0 using the Reset button.

3.1.5 Text Entry Pane
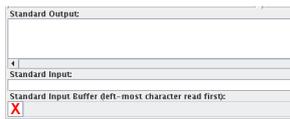
(N.B. This section may be skipped on first reading)

The Text Entry Pane is divided into three sections.

- The upper section, **Standard Output**, is where text directed by assembler system calls is displayed. This text may be output as individual characters or as null-terminated strings. Users cannot change values displayed in this area and it is only cleared when the simulator is closed.

- The middle section, **Standard Input**, allows for keyboard data entry to be entered independent of program execution. Simply type in the characters and press the Enter/Return key to terminate the string, and the characters will appear in the Standard Input Buffer (see below).

- The lower section, **Standard Input Buffer**, stores characters entered via Standard Input for use in providing input to assembler system calls intended to read input. The red **X** button will flush the contents of the Standard Input Buffer.

When a binary file executing in the MDP ARM Simulator makes an assembler system call to obtain character or string input, the simulator first checks the Standard Input Buffer. If the buffer is non-empty, the input will be taken directly from there. If the buffer is empty, then the Simulator will raise a Standard Input dialog window to obtain character input from the user at run-time.

3.2 Memory Map
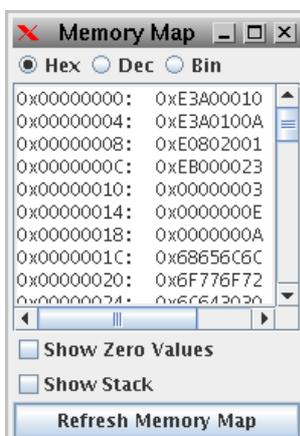
(N.B. This section may be skipped on first reading)

This window, accessible via the **view** → **Memory Map** menu item, provides a simulated view of part of an ARM processor memory map. The main portion of the map is taken up by a two-column list. The leftmost column of numbers are the memory address locations (in hexadecimal). The rightmost column of numbers are the contents held in the associated memory address locations - these values may be instructions, data or values held in the stack.

- **Hex / Dec / Bin** These radio buttons allow the contents of the memory address locations to be displayed in **Hex**adecimal, **Dec**imal, or **Bin**ary representations. The memory address locations remain listed as hexadecimal numbers.

- **Show Zero values** Ticking this checkbox will show all memory locations up to 0x3FC regardless of whether they contain zero or non-zero values. Clearing the checkbox will hide all memory locations that have a value of zero.

- **Show Stack** Ticking this checkbox will show all memory locations that form the temporary memory storage area known as the stack. Clearing the checkbox will hide these.

- **Refresh Memory Map** Pressing this button will refresh the contents of the Memory Map – will require new code to be loaded in to the Binary Pane and (partially) executed before the contents will update.

3.3 Hardware Simulation
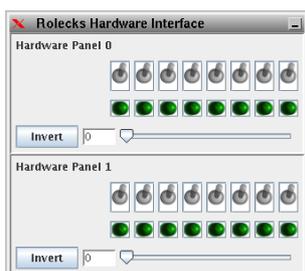
(N.B. This section may be skipped on first reading)

Selecting the display hardware panel icon above the register pane or **View** → **Registers**, opens (or raises in the windows stack) the simulated analogue and digital input/output display. The default setting is for two banks of LED's, Switches and a slider.

The MDP ARM Simulator incorporates a series of assembler system calls to use the virtual hardware for input and output. System calls can read values from the slider and the switch banks, and write values out to the LED bank. For input purposes, a switch has the value 0 if it is pointing up and the value 1 if it is pointing down - simply click on the switch to change its alteration. Each bank of switches can represent an 8-bit binary number - with the most significant bit as the leftmost switch. Each slider can also represent an 8-bit number according to the position of the slider bar - the current value is shown in the box to the left of the slider. To flip between the left-hand and right-hand end of the slider being the most significant, simply click on the corresponding Invert button for that slider. For output purposes, each LED bank also represents an 8-bit binary number. A "lit" (bright green) LED has the value 1, an "unlit" (dark green) LED has the value zero. The leftmost LED is always the most significant bit of the number.

4 Simple Programs

4.1 'Anatomy' of a Program

```
 1   ; Example code to load two registers
 2   ; and then add them
 3
 4   AREA example1, CODE, READONLY
 5
 6   ENTRY
 7
 8   start
 9    mov R0, #0x10                          ; r0 = 10
10    mov R1, #10                           ; r1 = a
11    add R2, R0, R1                        ; r2 = 1a
12
13   BL sysExit                             ; end of program
14
15   END
```

The line numbers (in italics) have been added to this program purely for ease of explanation. There are no line numbers in an ARM assembly language program; including line numbers in the source code of an assembly language program will cause it to fail to assemble.

```
1   ; Example code to load two registers
2   ; and then add them
3
```

The first lines are comments, all text following a ; on a line is considered to be a comment. Although comments are optional, it is good programming practice to include information about the code wherever possible. This is especially important in assembler programming where the format and mnemonics are more difficult to understand and use, especially when the author is trying to maximise speed or minimise memory space at execution time. Note that line 3 is blank; the assembler ignores blank lines when assembling the source code, so they can be safely used to aid the readability of the code.

```
4   AREA example1, CODE, READONLY
5
```

The AREA command is a **directive**. This is an instruction to the assembler, which is not passed on to the actual microprocessor. The AREA directive can be used to specify and name sections of program code or data. Here the AREA directive names this section as "example1" and indicates that it is a code segment using the CODE identifier. It is also declared as READONLY in that the code cannot be altered during its execution. A working ARM assembly program must have at least one CODE area.

---

16   *MDP ARM Simulator*

```
6   ENTRY
7
```

The ENTRY directive indicates the first instruction that should be executed within an application. The complete application can only have one entry point and so only one ENTRY directive should appear in a program.

The `start` label is optional but provides a useful marker to indicate the the beginning of the code that represents the actual program.

```
8   start
```

```
9    mov R0, #0x10   ; r0 = 10
10   mov R1, #10      ; r1 = a
11   add R2, R0, R1   ; r2 = 1a
```

This is an example of sample assembly language code. Assembly language programs may be a few lines (as here) to hundreds of lines of code in length. Note that all the instructions, e.g. `mov`, `add` are indented by 1 white-space character. It is often useful to add a comment at the end of the line to aid readability, see sections 3.1.2 for more information on the functionality available and the precise formats required.

```
13   BL sysExit   ; end of program
14
```

The `BL sysExit` command should be included in all programs. The command **B**ranch **L**inks to a system subroutine `sysExit` that ensures that all the system registers and Program Counter are returned to a suitable state before exiting the program.

```
15   END
```

The END directive indicates the end of the source file. Every ARM assembly program must have an END directive.

4.2 Simple Arithmetic

The program "example1" given in section 4.1 is a simple arithmetic program that loads two registers with immediate values and then adds them together, placing the sum in a third register.

If you are working from a pre-installed version of MAS-Rolecks or have access to the source files, (see www.eng-mdp.cam.ac.uk/microprocessor) use the **File → Load** menu entries to load the program into the edit pane. Alternatively use the functionality of the Editor, see section 3.1.2, to type in the code as shown in example 1.

```
     ..
8    start
9     mov R0, #0x10   ; r0 = 10
10    mov R1, #10      ; r1 = a
11    add R2, R0, R1  ; r2 = 1a
      ..
```

```
9   mov R0, #0x10  ; r0 = 10
```

The instruction `mov` moves a value (in this case, the hexadecimal number 10, decimal value 16) into a register (in this case, register `r0`). `r0` and `#10` are **operands**. `r2` is the **destination** register, #10 is a **source** operand.

```
10  mov R1, #10  ; r1 = a
```

In this line, the instruction `mov` moves the value (decimal 10) into a register, namely register 1.

```
11  add R2, R0, R1  ; r2 = 1a
```

The ADD instruction takes two source operands, in this case register `R0` and register `R3`, adds the values in the registers, i.e. 10 in r0 and 16 in `r1`, and puts the total back into the destination register of `r2`. Hence the value in `R2` after this instruction has been executed should be 26, or 1A in hexadecimal.

4.3   Assembling and Running Code

4.3.1   Writing New Source Files

To create a new source file from scratch in MAS-Rolecks, save any existing work in the Source/Editor pane using the **File → Save Source** command, supplying an existing or new file name as appropriate. Then use the **File → New Source** command or the **New Source** icon to clear the Source/Editor Pane and start typing the program into the Source/Editor Pane. Remember to press Enter/Return after each line of code.

4.3.2   Loading Existing Source Files

To load an existing source file into MAS-Rolecks, simply use the **File → Load Source** command or the **Open File** icon. (**NB:** Both methods will flush any existing source code.) Locate the desired source file (which should have a .s suffix) using the file system browser dialog and click OK. The source file will appear in the Editor/Source pane.

4.3.3   Saving Source Files

To save a source file, use the **File → Save Source** command or the **Save Source** icon. This will raise a file system browser dialog, allowing the file to be saved with its existing name (if any) or with a new name. In either case, the source file should have a .s extension.

4.3.4   One-Click Assembly

To assemble a source file with a single mouse-click, use the Magic Wand icon. This will first save the text in the editor pane, under the currently selected filename if it exists (and

will raise a file system dialog if not). It will then assemble the resulting source file, raising a message window to signal successful or unsuccessful assembly and information on the first error. In the latter case, rectify the errors in the source code, save the file, and try again. If successful, it will finally load the newly created binary file into the binary pane – the new binary file will have the same name as the source but have a .bin extension.

4.3.5   Two-Click Assembly

Use either the Assemble icon or the **Build → Assemble for Rolecks** command to assemble a source file. This will raise a file system browser dialog, which will default to the currently saved version of the file in the Source/Editor pane. Choose the appropriate filename and click OK. A message window will appear, either stating "Source file assembled correctly" if there were no assembly errors or giving details of the first error encountered. In the latter case, rectify the errors in the source code, save the file, and try again.

Once the source file has been successfully assembled, use the **File → Load Binary** command or the Open Binary File icon (in the Binary Pane). This will raise a file system dialog window, which will default to the .bin file generated by the most recent assembly process. Click OK to load the binary file into the Binary Pane.

4.3.6   Running the Program

There are two methods of executing programs in MAS-Rolecks - "run-through" and "single-step".

"Run-through" executes the entire program from start (or currently highlighted code) to finish. The execution will normally only pause if an assembler system call is made for keyboard input and no stored input in the Standard Input Buffer. To perform a "run-through" execution, use either the **Build → Run in Rolecks** command or the Run icon above the Binary Pane. To forcibly suspend a program executing in "run-through" mode, click on the Pause icon (which replaces the Run icon). Execution can then be restarted from the pause point in either "run-through" or "single-step" mode or the system reset.

"Single-step" mode allows the user to step through the execution of the program one instruction at a time each time the "Single step" icon is pressed. The light blue highlight bar will indicate the instruction that is just about to be executed. In "Single step" mode, it is possible to observe the values of

the registers and condition code bits change in response to each succeeding instruction.

The Reset button terminates the execution of a currently running binary program. It restores the Program Counter to the first instruction of the program, resets the values of registers R0 to R12 and R14 to zero, and clears the condition code bits.

4.4 Interaction Simulated Hardware

4.4.1 Switches

```
1   ; Example code to read in switch value and store to register
2   ; repeats until input is 255
3
4   AREA example2, CODE, READONLY
5
6   ENTRY
7
8   start
9     BL MAS_Switch_Read              ; read switch into R0
10    CMP R0, #255                    ; test R0 against 255
11    BNE start                       ; if Z bit clear,
12                                     ; back to start
13  BL sysExit                        ; end of program
14
15  END
```

The program above reads the value of a switch bank (in this case Hardware Panel 1) into a register (RO), and compares that value with 255. If the input value does not equal 255, the program loops back to the start and tries again. If the input value does equal 255, the program terminates. Create, save, assemble and run this program in "single-step" mode to see that this is the case.

```
8   start
9     BL MAS_Switch_Read   ; read switch into R0
```

Line 9 is an assembler system call that reads the value of the switches in Hardware Panel 1, and places that value into register R0. Remember that the switches have value 0 if in the "up" position and value 1 in the "down" position.

```
10    CMP R0, #255   ; test R0 against 255
```

Line 10 uses the CMP (CoMPare) instruction to compare the value of one operand against the value of another. In this case, the comparison is between the value of register R0 and 255. The CMP instruction does not have a destination operand specifying a register where the result of the comparison is to be stored. Instead, the CMP instruction changes the values of the condition code bits as follows assuming two values r1 and r2:

- N = 1 if the most significant bit of (r1 - r2) is 1, i.e. r2 ¿ r1

- Z = 1 if (r1 - r2) = 0, i.e. r1 = r2

- C = 1 if r1 and r2 are both unsigned integers AND (r1 ¡

r2)

- V = 1 if r1 and r2 are both signed integers AND (r1 ¡ r2)

For our example, Z will be set to 1 if R0 has the value 255 and be clear otherwise. Observe the change in the condition code bits after this instruction is executed in "single-step" mode.

```
11    BNE start     ; if Z bit clear,
12                  ; back to start
13    BL sysExit    ; end of program
```

The BNE (Branch Not Equal) instruction at Line 11 inspects the current value of the Z condition code bit. If $Z \neq 1$, then the branch will be taken and the execution of the program will return to the label "start". If Z = 1, then the branch will **not** be taken, and the program will carry on to line 13 and the termination of the program.

4.4.2 Sliders

```
1    ; Example code to read in slider value and store to register
2    ; repeats until input is 128 or higher
3
4    AREA example3, CODE, READONLY
5
6    ENTRY
7
8    start
9      BL MAS_Slider_Read              ; read slider into R0
10     CMP R0, #128                    ; test R0 against 128
11     BLO start                       ; if C bit clear,
12                                     ; back to start
13   BL sysExit                        ; end of program
14
15   END
```

The program above reads the value of a slider (in this case from Hardware Panel 1) into a register (RO), and compares that value with 128. If the input value is less than 128, the program loops back to the start and tries again. If the input value is equal to or greater than 128, the program terminates. Create, save, assemble and run this program in "single-step" mode to see that this is the case.

```
8  start
9    BL MAS_Slider_Read   ; read slider into R0
```

Line 9 is an assembler system call that reads the value of the slider in Hardware Panel 1, and places that value into register R0.

```
10   CMP R0, #128   ; test R0 against 128
```

Line 10 uses the CMP (CoMPare) instruction to compare the value of one operand against the value of another. In this case, the comparison is between the value of register R0 and 128. The CMP instruction will alter the condition code bits as described above. In this example, we are interested in the value of the C bit, which will be cleared (set at 0) if the input (in R0) is less than 128, and set to 1 if the value of R0 is 128 or more. Note that the other condition code bits may also be changed by the CMP instruction.

```
11   BLO start    ; if C bit clear,
12                ; back to start
13   BL sysExit   ; end of program
```

The BLO (Branch if LOwer) instruction at Line 11 inspects the current value of the C condition code bit. If C = 0, then the branch will be taken and the execution of the program will return to the label "start". If C = 1, then the branch will **not** be taken, and the program will carry on to line 13 and the termination of the program.

4.4.3 LEDs

```
1   ; Example code that counts from 255 to 1, displaying each
2   ; value in LEDs program stops when register = 0
3
4   AREA example4, CODE, READONLY
5
6   ENTRY
7
8   start
9     MOV r0, #0xff                  ; set R0 = 255
10  loop BL MAS_LED_ONOFF            ; write R0 to LED bank
11    SUB r0, r0, #0x1               ; set R0 = R0 - 1
12    CMP R0, #0x0                   ; test R0 against 0
13    BHI loop                       ; if C = 1 and Z = 0,
14                                   ; back to loop at line 10
15  BL sysExit                       ; end of program
16
17  END
```

This program counts down from 255 to 1 (inclusive), displaying each integer value as a bit pattern on one of the LED banks.

Line 9 initialises the value of R0 to 255 using a MOVe instruction.

```
8   start
9     MOV r0, #0xff   ; set R0 = 255
```

Line 10 is an assembler system call that takes the current value in register R0 and writes it out

```
10  loop BL MAS_LED_ONOFF   ; write R0 to LED bank
```

to one of the LED banks in the Virtual Hardware Interface, in this case Hardware Panel 1. Note that Line 10 has its own label, "loop".

```
11    SUB r0, r0, #0x1   ; set R0 = R0 - 1
```

Line 11 uses the SUB instruction to decrement the value of R0 by 1. The SUB instruction subtracts 1 (the second source operand) from the value in the first source operand (here R0), and places the new value back into R0

```
12    CMP R0, #0x0   ; test R0 against 0
13    BHI loop       ; if C = 1 and Z = 0,
13                   ; back to loop at line 10
```

Line 12 uses the CMP instruction to compare the values of R0 and 0. In this example, we will be interested in the value of the Z and the C bits. Line 13 is a BHI (Branch if HIgher) instruction. If the value of C is 1 **and** Z is 0 (which occurs if R0 > 0) then the program branches back to the "loop" label at line 10. If not, then the branch is not taken and the program proceeds to line 15 and termination.

4.5   Textual I/O

4.5.1   Text Input

4.5.2   Text Output

4.6   Use of Memory Map

- memory map and what can be seen

4.7   Modification of Registers

- update of registers - watch as happens or inject different values in single-step

5 Simplified ARM Assembler

Internally the ARM uses 32 bit instructions which are generally called from an assembler using a relatively standard Opcode format. The syntax used in MAS Rolecks is similar to those found in other ARM assemblers, e.g. ADS, Kile. Details of individual formats/syntax can be found in Appendicies **??** , however the general construction is shown in the following sections.

5.1 Instruction Syntax

Opcode   Flags   Operand 1
⇓   ⇓   ⇓

ADD EQ S   R0,   R1,   R2, LSL#2
⇑   ⇑   ⇑
Condition   Destination   Operand 2
Code   Register
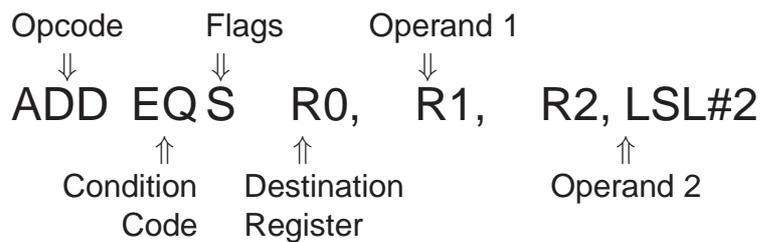
Where

Opcode Typically a 3 letter mnemomic, common codes used below, see Appendix **??** for full details

Mnemonic	Description	Example	Mnemonic	Description	Example
ADC	Add with Carry	ADC R0, R1, #1	MOV	Move	MOV R0, #0x07
ADD	Add	ADD R0, R1, R2	ORR	Logical OR	ORR R0, R1
AND	Logical And	AND R0, R0, #0xFF	STR	Store Register	STR R0, [R1]
B	Branch	B loop	SUB	Subtract	SUB R0, R1, R2
BL	Branch Link	BL mas_leds_onoff	TEQ	Test equivalence	TEQ R1, R2
CMP	Compare	CMP R0, R1			
LDR	Load Register	LDR R0, [R1]			

Condition Code indicates what conditions are inspected prior to actioning the instructions, see Appendix **??** for more details.

Mnemonic	Meaning	Status Flags	Mnemonic	Meaning	Status Flags
EQ	EQual	Z=1	VC	oVerflow Clear	V=1
NE	Not Equal	Z=0	HI	unsigned HIgher	C=1,Z=0
CS	Carry Set	C=1	LS	unsigned Lower or Same	C=0, Z=1
CC	Carry Clear	C=0	GE	signed Greater than or Equal	N=V
HS	Higher or Same	C=1	LT	signed Less Than	N!=V
LO	unsigned LOwer	C=0	GT	signed Greater Than	Z=0, N=V
MI	MInus/negative	N=1	LE	signed Less than or Equal	Z=1,N!=V
PI	PLus/positive	N=0	AL	ALways	-
VS	oVerflow Set	V=1	NE	NEver	-

Status Flag Adding S to the opcode ensures that the status flags are updated when the instruction is actioned.

Destination Register Typically R0 → R12

Operand 1 Normally a register (R0 → R12)

Operand 2 Normally an optional parameter, but typically one of the following formats.

- *Immediate* e.g. #0x23 (35)

- *Register* e.g. R2

- *Resister + barrel Shifter e.g. R2, LSL#2 (4*R2)*

5.2 Instruction Types

There are five general types of instructions/opcodes:-

Data operations (For further information see Appendix **??**)

This group does most of the work. There are sixteen instructions, and they have very similar formats. Examples of instructions from this group are ADD and CMP, which add and compare two numbers respectively. The operands of these instructions are always in registers (or an immediate number stored in the instruction itself), never in memory.

Load and Store (For further information see Appendix **??**)

Group of two instructions: load a register and save a register, with a number of variations which include whether bytes or words are transferred, and how the memory location to be used is obtained.

Multiple Load and Store Group of instructions that allows between one and 16 registers to be moved between the processor and memory. Only word transfers are performed by this group.

Branching Although the PC may be altered using the data operations to cause a change in the program counter, the branch instruction provides a convenient way of reaching any part of the 64M byte address space in a single instruction. It causes a displacement to be added to the current value of the PC. The displacement is stored in the instruction itself.

SWI This one-instruction group is very important. The abbreviation stands for 'SoftWare Interrupt'. It provides the way for user's programs to access the facilities provided by the operating system. All ARM -based computers provide a certain amount of pre-written software to perform such tasks as printing characters on to the screen, performing disc I/O etc. By issuing SWI instructions, the user's program may utilise this operating system software, obviating the need to write the routines for each application.

6 MAS Rolecks Assembler Functions

ADD NOTES ABOUT SWI/BL

The documentation that follows tries to copy the ARM style documentation for SWIs as much as possible. Notable exceptions are the lack of a SWI number. These are function-like calls that are invokes using a BL (branch-link) instruction, possibly with a conditional suffix. Note that these routines differ from C functions in that they may return values in an**Line 8:** `; and store the total back into register 2`

6.1 System operations

MAS-Rolecks implements a number of system level calls similar to other ARM asssemblers, the operation of those commonly used are given below and a the full list is given in appendix **??**.

6.1.1 SYS_Exit

Terminates the current program

On Entry R1 contains the return code for the program

On Exit This routine does not return

6.1.2 SYS_ReadC

Reads a byte from the standard input.

On Entry Register R1 must contain zero. There are no other parameters or values possible.

On Exit R0 contains the byte read from the console.

6.1.3 SYS_WriteC

Writes a character byte, contained in R1, to the standard outputl.

On Entry R1 contains the character.

On Exit None. Register R0 is corrupted.

Comment This differs from ADS SYS_WriteC, where R1 is a pointer to the byte.

6.2 Hardware i/o Routines

In addition to the standard ARM consistan i/o functions MAS also has a number of routines that can be called as subroutines that access simulated or real hardware (if connected or running on a MDP Balloon system) A full list of the available functions is given in appendix **??**. The calls allow access to hardware at various depth levels, but the normal user is likely to use the high level varieties that are listed below and work directly with the simulated hardware describe din section **??**

6.2.1 MAS_LED_OnOff

Sets LED states to either on or of in a banks of 8 LEDs at a time.

On Entry – R0 contains in bits 0 to 7 the new state of the 8 LEDs in the bank indicated by R1

– R1 contains the LED bank number

On Exit – R0 contains 0 if the operation was successful and -1 otherwise

– R1 is preserved

6.2.2 MAS_Slider_Read

Reads the current state of the analogue slide control

On Entry R1 contains the slider number (usually 0)

On Exit R0 contains the slider value in bits 15..0

Comments On exit R0 will contain a 16-bit number, but it should not be assumed that the ADC that took the reading is 16-bit. The lower bits could be 0.

6.2.3 MAS_Switch_Read

Reads the current state of a bank of up to 8 switches

On Entry R1 contains the switch bank number

On Exit R0 contains the switch settings in bits 7..0

Appendices

A ARM Assembler Opcodes

A.1 Data Operations - Arithmetic

A.1.1 ADD $\qquad$ *ADD*

Syntax ADD <cc> <S> Rd, Rn, <Op1>

Description <Rd> = <Rn> + <Op1>
Add the value of <Op1> to register Rn and store the result in Rd.

Example

```
ADD  R0, R0,#1          ;Increment R0
ADD  R1, R1, R1, LSL#2 ;R1*5
ADDS R0, R1, R2          ;R0=R1+R2, check for overflow
```

Condition Codes The **N** and **Z** flags are set on the result of the addition. The **C** and **V** flags are set dependent on whether the the addition generated a carry (unsigned overflow) or a signed overflow.

Instruction Format

Opcode	31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
ADD<cond><S> Rd, Rn, #	cond	0 0	1	0 1 0 0	S	Rn	Rd	rotate	#		
ADD<cond><S> Rd, Rn, Rm OP #	cond	0 0	0	0 1 0 0	S	Rn	Rd	shift#	shift	0	Rm
ADD<cond><S> Rd, Rn, Rm OP Rs	cond	0 0	0	0 1 0 0	S	Rn	Rd	Rs	0 shift	1	Rm

A.1.2 ADC $\qquad$ *ADD with Carry*

Syntax ADC <cc> <S> Rd, Rn, <Op1>

Description <Rd> = <Rn> + <Op1>+ <Carry>

Used in a similar fashion to ADD but with the carry added into the result as well dependant on <op1>

Example $\qquad$;add the 64-bit number in R2, R3
$\qquad$;to that in R0, R1
```
ADDS R0, R0, R2 ;Add the lower words
ADC  R1, R1, R3 ;Add upper words using carry
```

N.B setting the final line to

```
ADCS  R1, R1, R3 ;Add upper words using carry
                 ;Set Flags
```

results in the flags being set if:-
N The 64-bit result was negative
C Unsigned overflow occurred
V signed overflow occurred
Z The top 32 bits are zero

Condition Codes The **N** and **Z** flags are set on the result of the addition. The **C** and **V** flags are set dependent on whether the the addition generated a carry (unsigned overflow) or a signed overflow.

Instruction Format

Opcode	31 30 29 28	27 26 25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
ADC<cond><S> Rd, Rn, #	cond	0 0 1	0 1 0 1	S	Rn	Rd	rotate	#		
ADC<cond><S> Rd, Rn, Rm OP #	cond	0 0 0	0 1 0 1	S	Rn	Rd	shift#	shift	0	Rm
ADC<cond><S> Rd, Rn, Rm OP Rs	cond	0 0 0	0 1 0 1	S	Rn	Rd	Rs	0 shift	1	Rm

A.1.3 **CMP**   *Compare*

Syntax CMP <cc> <S> Rn, <Op1>

Description Sets the status based on the subtraction <Rn> - <Op1>. N.B. the <S> option is assumed.

Examples
```
; Put lower of R0 and R1 in R0
MOV    R0, #0x10
MOV    R1, #0x8
CMP    R0, R1
MOVGT  R0, R1
```

Condition Codes The **N** and **Z** flags are set on the result of the substraction. The **C** and **V** flags are set dependent on whether the substraction generated a carry (unsigned overflow) or a signed overflow.

Note The carry flag [**C**] is inverted as it is cleared by a subtraction that needs a borrow and set by a calculation that does not need a borrow.

Instruction Format

Opcode	31 30 29 28	27 26 25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
CMP<cond><S> Rn, #	cond	0 0 1	1 0 1 0	S	Rn	SBZ	rotate	#		
CMP<cond><S> Rn, Rm OP #	cond	0 0 0	1 0 1 0	S	Rn	SBZ	shift#	shift	0	Rm
CMP<cond><S> Rn, Rm OP Rs	cond	0 0 0	1 0 1 0	S	Rn	SBZ	Rs	0 shift	1	Rm

A.1.4 **CMN**   *Compare Negative*

Syntax CMN <cc> <S> Rn, <Op1>

Description Sets the status based on the subtraction Rn - - <Op1>
= Rn + <Op1>. N.B. the <S> option is assumed.

Examples `CMP    R0, #0x5 ; Compare R0 with -5`

Condition Codes The **N** and **Z** flags are set on the result of the substraction. The **C** and **V** flags are set dependent on whether the substraction generated a carry (unsigned overflow) or a signed overflow.

Note The carry flag [**C**] is inverted as it is cleared by a subtraction that needs a borrow and set by a calculation that does not need a borrow.

Instruction Format

Opcode	31 30 29 28	27 26 25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
CMN<cond><S> Rn, #	cond	0 0 1	1 0 1 1	S	Rn	SBZ	rotate		#		
CMN<cond><S> Rn, Rm OP #	cond	0 0 0	1 0 1 1	S	Rn	SBZ	shift#		shift	0	Rm
CMN<cond><S> Rn, Rm OP Rs	cond	0 0 0	1 0 1 1	S	Rn	SBZ	Rs	0	shift	1	Rm

A.1.5 **SUB** *Subtract*

Syntax SUB <cc> <S> Rd, Rn, <Op1>

Description <Rd> = <Rn> - <Op1>+ <Carry>
Subtract the value of <Op1> from register Rn and store the result in Rd.

Examples

```
SUBS R0, R0,#1          ;Decrement R0, setting flags
                        ;NB can be used to set the
                        ;end of a loop without CMP
SUBS R1, R1, R1, ASR#2 ;R1=R1*(3/4) = R1-(R1/4)
```

Condition Codes The **N** and **Z** flags are set on the result of the substraction. The **C** and **V** flags are set dependent on whether the substraction generated a carry (unsigned overflow) or a signed overflow.

Note The carry flag [**C**] is inverted as it is cleared by a subtraction that needs a borrow and set by a calculation that does not need a borrow. If multiword subtractions are being used the effect is reset if high word subtractions use the SBCS (Subtract with Carry) commands, e.g.

```
                   ;((R1*2^32)+R0) - ((R3*2^32)+R2)
SUBS R0, R0, R2    ;SUB lower words
SBC  R1, R1, R3    ;SUB upper words using borrow
```

Instruction Format

Opcode	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SUB<cond><S> Rd, Rn, #	cond				0	0	1	0	0	1	0	S		Rn				Rd				rotate						#				
SUB<cond><S> Rd, Rn, Rm OP #	cond				0	0	0	0	0	1	0	S		Rn				Rd				shift#				shift		0		Rm		
SUB<cond><S> Rd, Rn, Rm OP Rs	cond				0	0	0	0	0	1	0	S		Rn				Rd				Rs			0	shift		1		Rm		

A.1.6 SBC — Subtract with Carry

Syntax SBC <cc> <S> Rd, Rn, <Op1>

Description <Rd> = <Rn> - <Op1> - NOT(<Carry>)
Used in multiword subtractionsa similar fashion to `ADD` but with the carry added into the result as well dependant on <op1>

Example
```
                        ;((R1*2^32)+R0) - ((R3*2^32)+R2)
SUBS R0, R0, R2    ;SUB lower words
SBC  R1, R1, R3    ;SUB upper words using borrow
```

Condition Codes The **N** and **Z** flags are set on the result of the substraction. The **C** and **V** flags are set dependent on whether the substraction generated a carry (unsigned overflow) or a signed overflow.

Note The carry flag [**C**] is inverted as it is cleared by a subtraction that needs a borrow and set by a calculation that does not need a borrow. If multiword subtractions are being used the effect is reset if high word subtractions use the SBCS (Subtract with Carry) commands.

Instruction Format

Opcode	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SBC<cond><S> Rd, Rn, #	cond				0	0	1	0	1	1	0	S		Rn				Rd				rotate						#				
SBC<cond><S> Rd, Rn, Rm OP #	cond				0	0	0	0	1	1	0	S		Rn				Rd				shift#				shift		0		Rm		
SBC<cond><S> Rd, Rn, Rm OP Rs	cond				0	0	0	0	1	1	0	S		Rn				Rd				Rs			0	shift		1		Rm		

A.1.7 RSB — Reverse Subtract

Syntax RSB <cc> <S> Rd, Rn, <Op1>

Description <Rd> = <Op1> - <Rn>
Subtract the value of register Rn from <Op1> and store the result in Rd, which allows the operands in Op1 to be used in subtractions.

Examples
```
RSB  R0, R0, #1         ;R0 = 1 - R0
RSB  R0, R0, R0, ASL#4 ;R0= 15*R0 = (16*R0)-R0
```

Condition Codes The **N** and **Z** flags are set on the result of the substrac-tion. The **C** and **V** flags are set dependent on whether the substraction generated a carry (unsigned overflow) or a signed overflow.

Note The carry flag [**C**] is inverted as it is cleared by a sub-traction that needs a borrow and set by a calculation that does not need a borrow. If multiword subtractions are being used the effect is reset if high word sub-tractions use the RSCS (Reverse Subtract with Carry) commands.

Instruction Format

Opcode	31 30 29 28	27 26 25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
RSB\<cond\>\<S\> Rd, Rn, #	cond	0 0 1	0 0 1 1	S	Rn	Rd	rotate	#			
RSB\<cond\>\<S\> Rd, Rn, Rm OP #	cond	0 0 0	0 0 1 1	S	Rn	Rd	shift#	shift	0	Rm	
RSB\<cond\>\<S\> Rd, Rn, Rm OP Rs	cond	0 0 0	0 0 1 1	S	Rn	Rd	Rs	0	shift	1	Rm

---

A.1.8 RSC   *Reverse Subtract with Carry*

Syntax RSC \<cc\> \<S\> Rd, Rn, \<Op1\>

Description \<Rd\> = \<Op1\> - \<Rn\> - NOT(\<Carry\>)
Used in multiword subtractionsa similar fashion to RSB but with the carry added into the result as well depen-dant on \<op1\>

Example

```
                              ;((R2*2^32)+16) - ((R1*2^32)+R0)
RSBS R0, R0, #0x10 ;SUB lower words
RSC  R1, R1, R2    ;SUB upper words
```

Condition Codes The **N** and **Z** flags are set on the result of the substrac-tion. The **C** and **V** flags are set dependent on whether the substraction generated a carry (unsigned overflow) or a signed overflow.

Instruction Format

Opcode	31 30 29 28	27 26 25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
RSC\<cond\>\<S\> Rd, Rn, #	cond	0 0 1	0 1 1 1	S	Rn	Rd	rotate	#			
RSC\<cond\>\<S\> Rd, Rn, Rm OP #	cond	0 0 0	0 1 1 1	S	Rn	Rd	shift#	shift	0	Rm	
RSC\<cond\>\<S\> Rd, Rn, Rm OP Rs	cond	0 0 0	0 1 1 1	S	Rn	Rd	Rs	0	shift	1	Rm

---

A.2 Data Operations - Logical

A.2.1 AND   *Logical AND*

Syntax AND \<cc\> \<S\> Rd, Rn, \<Op1\>

Description Performs a full 32 bitwise AND on Rn with the value of
<Op1>, storing the result in Rn.

Rn	<Op1>	Rd
0	0	0
0	1	0
1	0	0
1	1	1

Example

```
ANDS R0, R0, R1    ;Mask R0 with R1
AND  R0, R0, #0xDF ;Set ASCII code to upper case
```

Condition Codes The **N** and **Z** flags are set on the result of the AND
operation. The **C** flag is set but the **V** flag is untouched.

Instruction Format

Opcode	31 30 29 28	27 26 25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
AND<cond><S> Rd, #	cond	0 0 1	0 0 0 0	S	Rn	Rd	rotate	#		
AND<cond><S> Rd, Rm OP #	cond	0 0 0	0 0 0 0	S	Rn	Rd	shift#	shift	0	Rm
AND<cond><S> Rd, Rm OP Rs	cond	0 0 0	0 0 0 0	S	Rn	Rd	Rs	0 shift	1	Rm

A.2.2 **BIC** *Bitwise Clear*

Syntax BIC <cc> <S> Rd, Rn, <Op1>

Description Performs a full 32 bitwise AND on Rn with the value of
NOT<Op1>, storing the result in Rn.

Rn	<Op1>	NOT<Op1>	Rd
0	0	1	0
0	1	0	0
1	0	1	1
1	1	0	0

Example

```
BICS R0, R0, #0xF  ;Clear bits 0-4
```

Condition Codes The **N** and **Z** flags are set on the result of the AND
operation. The **C** flag is set but the **V** flag is untouched.

Instruction Format

Opcode	31 30 29 28	27 26 25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
BIC<cond><S> Rd, #	cond	0 0 1	1 1 1 0	S	Rn	Rd	rotate	#		
BIC<cond><S> Rd, Rm OP #	cond	0 0 0	1 1 1 0	S	Rn	Rd	shift#	shift	0	Rm
BIC<cond><S> Rd, Rm OP Rs	cond	0 0 0	1 1 1 0	S	Rn	Rd	Rs	0 shift	1	Rm

A.2.3 **EOR** — *Logical Exclusive OR*

Syntax EOR <cc> <S> Rd, Rn, <Op1>

Description Performs a full 32 bitwise EOR on Rn with the value of <Op1>, storing the result in Rn.

Rn	<Op1>	Rd
0	0	0
0	1	1
1	0	1
1	1	0

Example

```
EOR R0, R0, #0x55 ;invert bits 0,2,4,6
```

Condition Codes The **N** and **Z** flags are set on the result of the EOR operation. The **C** flag is set but the **V** flag is untouched.

Instruction Format

Opcode	31 30 29 28	27 26 25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
EOR<cond><S> Rd, #	cond	0 0 1	0 0 0 1	S	Rn	Rd	rotate	#		
EOR<cond><S> Rd, Rm OP #	cond	0 0 0	0 0 0 1	S	Rn	Rd	shift#	shift	0	Rm
EOR<cond><S> Rd, Rm OP Rs	cond	0 0 0	0 0 0 1	S	Rn	Rd	Rs	0 shift	1	Rm

A.2.4 **MOV** — *Move Value*

Syntax MOV <cc> <S> Rd, <Op1>

Description Transfers the value of <Op1> to register Rd.

Example

```
MOV  R0, R0, LSL #2 ;R0=R0*4
MOVS R0, R1          ;R0=R1 with flags set
MOV  R1, #0x40       ;R1=64 (40_{16})
```

Condition Codes The **N** and **Z** flags are set on the result of the addition. The **C** and **V** flags are set dependent on whether the the addition generated a carry (unsigned overflow) or a signed overflow.

Instruction Format

Opcode	31 30 29 28	27 26 25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
MOV<cond><S> Rd, #	cond	0 0 1	1 1 0 1	S	SBZ	Rd	rotate	#		
MOV<cond><S> Rd, Rm OP #	cond	0 0 0	1 1 0 1	S	SBZ	Rd	shift#	shift	0	Rm
MOV<cond><S> Rd, Rm OP Rs	cond	0 0 0	1 1 0 1	S	SBZ	Rd	Rs	0 shift	1	Rm

A.2.5 **MVN** *Move Negative*

Syntax MVN <cc> <S> Rd, <Op1>

Description Transfers the logical NOT of Op1 to the desitination register Rd. N.B. -n= NOT (n-1) thus to load a negative number, subtract 1 from the positive number and use MVN.

Examples
```
MVNS R0, R0     ;Invert bits in R0, e.g. form mask
                ;and set flags
MVN  R1, #0     ;Set R1 to -1, e.g. 0xFFFFFFFF
```

Condition Codes The **N** and **Z** flags are set on the result of the operation. The **C** flag is set to the carry output generated by the shifter. **V** flag is not set.

Instruction Format

Opcode	31 30 29 28	27 26 25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
MVN<cond><S> Rd, #	cond	0 0 1	1 1 1 1	S	SBZ	Rd	rotate	#	
MOV<cond><S> Rd, Rm OP #	cond	0 0 0	1 1 1 1	S	SBZ	Rd	shift#	shift 0	Rm
MOV<cond><S> Rd, Rm OP Rs	cond	0 0 0	1 1 1 1	S	SBZ	Rd	Rs	0 shift 1	Rm

A.2.6 **ORR** *Logical OR*

Syntax ORR <cc> <S> Rd, Rn, <Op1>

Description Performs a full 32 bitwise OR on Rn with the value of <Op1>, storing the result in Rn. Effectively the opposite of BIC.

Rn	<Op1>	Rd
0	0	0
0	1	1
1	0	1
1	1	1

Example
```
ORR R0, R0, #0x55 ;Set bits 0,2,4,6
```

Condition Codes The **N** and **Z** flags are set on the result of the EOR operation. The **C** flag is set but the **V** flag is untouched.

Instruction Format

Opcode	31 30 29 28	27 26 25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
ORR<cond><S> Rd, #	cond	0 0 1	1 1 0 0	S	Rn	Rd	rotate	#	
ORR<cond><S> Rd, Rm OP #	cond	0 0 0	1 1 0 0	S	Rn	Rd	shift#	shift 0	Rm
ORR<cond><S> Rd, Rm OP Rs	cond	0 0 0	1 1 0 0	S	Rn	Rd	Rs	0 shift 1	Rm

A.2.7 **TEQ**   *Test Equivalence*

Syntax   TEQ <cc> <S> Rn, <Op1>

Description   Performs a full 32 bitwise EOR on Rn and <Op1> but only only sets the result flags. N.B. the option <S> is assumed.

Example

```
TEQ R1, #0x15 ;Check if R1=21
```

Condition Codes   The **Z** flag is set if the two operands are equal.

Instruction Format

Opcode	31 30 29 28	27 26 25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
TEQ<cond><S> Rn, #	cond	0 0 1	1 0 0 1	S	Rn	SBZ	rotate		#		
TEQ<cond><S> Rn, Rm OP #	cond	0 0 0	1 0 0 1	S	Rn	SBZ	shift#	shift	0		Rm
TEQ<cond><S> Rn, Rm OP Rs	cond	0 0 0	1 0 0 1	S	Rn	SBZ	Rs	0	shift	1	Rm

A.2.8 **TST**   *Test Bits*

Syntax   TST <cc> <S> Rn, <Op1>

Description   Performs a full 32 bitwise AND on Rn and <Op1> but only only sets the result flags. N.B. the option <S> is assumed.

Example

```
TEQ R1, #0x1 ;Check if R1 bit 8 is set
```

Condition Codes   The **Z** flag is set if the two operands are equal.

Instruction Format

Opcode	31 30 29 28	27 26 25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
TST<cond><S> Rn, #	cond	0 0 1	1 0 0 0	S	Rn	SBZ	rotate		#		
TST<cond><S> Rn, Rm OP #	cond	0 0 0	1 0 0 0	S	Rn	SBZ	shift#	shift	0		Rm
TST<cond><S> Rn, Rm OP Rs	cond	0 0 0	1 0 0 0	S	Rn	SBZ	Rs	0	shift	1	Rm

A.3   Load and Store

A.3.1 **LDR**   *Load Register*

Syntax   LDR <cc> Rd, <Op2> <!>

Description   load the register Rd with a word from the memory location calculated by <Op2>. If PC (R15) is used as the Rd the program branches to the address specified in <Op2>.

Examples   Pre-indexing

```
LDR R0, [R1, #1]   ;Load R0 with memory location R1+1
LDR R0, [R1, #4]!  ;Load R0 with memory location R1+1
                        ;Put R1+4 into R1
LDR R0, [R1, #-4, LSR#3];Load R0 with memory location
                              ;R1+(-4*8)
```

Post-indexing

```
LDR R0, [R1], #4   ;Load R0 with memory location R1
                        ;R1 is subsequently set to R1+4
LDR R0, [R1], #-4  ;Load R0 with memory location R1
                        ;Then set R1=R1-4
```

Condition Codes Not changed

Note the result is unpredicatable if:-

– <Op2> is not word aligned

– Rd=<Op2> and the writeback option <!> is selected

Instruction Format

Opcode	31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
LDR<cond> Rd, Rn, #	cond	0 1 0	P	U	0	W	0	Rn	Rd	#
LDR<cond> Rd, Rn, #	cond	0 1 1	P	U	0	W	0	Rn	Rd	shift# / shift / 0 / Rm

A.3.2 **LDRB** *Load Register Byte*

Syntax LDR<cc>B Rd, <Op2> <!>

Description load the register Rd with a byte, extended to 32bits with zeros, from the memory location calculated by <Op2>. If PC (R15) is used as the base register PC relative addressing is possible.

Examples Pre-indexing

```
LDRB R0, [R1, #4]! ;Load R0 with a byte from
                        ;memory location R1+1
                        ;Put R1+4 into R1
```

Post-indexing

```
LDRB R0, [R1], #4  ;Load R0 with a byte from
                        ;memory location R1
                        ;R1 is subsequently set to R1+4
```

Condition Codes Not changed

Note the result is unpredicatable if:-

- Rd=15 (PC)

- Rd=<Op2> and the writeback option <!> is selected

Instruction Format

Opcode	31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
LDR<cond>B Rd, Rn, #	cond	0 1 0	P	U	1	W	0	Rn	Rd	#			
LDR<cond>B Rd, Rn, #	cond	0 1 1	P	U	1	W	0	Rn	Rd	shift#	shift	0	Rm

A.3.3 STR   *Store Register*

Syntax STR <cc> Rd, <Op2> <!>

Description Store the register Rd in the memory location calculated by <Op2>. If PC (R15) is used as the base register allows position independant code using PC-relative addressing.

Examples Pre-indexing

```
STR R0, [R1, #4]!        ;Store R0 in memory location
                         ;R1+4, Put R1+4 into R1
STR R0, [R1, #-4, LSR#3] ;Store R0 in memory
                         ;location R1+(-4*8)
```

Post-indexing

```
STR R0, [R1], #4   ;Store R0 in memory location R1
                   ;R1 is subsequently set to R1+4
STR R0, [R1], #-8  ;Store R0 in  memory location R1
                   ;Then set R1=R1-8
```

Condition Codes Not changed

Note the result is unpredicatable if:-

- <Op2> is not word aligned

- Rd=<Op2> and the writeback option <!> is selected

Instruction Format

Opcode	31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
STR<cond> Rd, Rn, #	cond	0 1 0	P	U	0	W	0	Rn	Rd	#			
STR<cond> Rd, Rn, #	cond	0 1 1	P	U	0	W	0	Rn	Rd	shift#	shift	0	Rm

A.3.4 **STRB** *Store Register Byte*

Syntax STR<cc>B Rd, <Op2> <!>

Description Store the least significant byte in register Rd in the memory location calculated by <Op2>. If PC (R15) is used as the base register PC relative addressing is possible.

Examples Pre-indexing

```
STRB R0, [R1, #1]! ;Store R0 in the
                   ;memory location R1+1
                   ;Put R1+1 into R1
```

Post-indexing

```
STRB R0, [R1], #4  ;Store R0 in the
                   ;memory location R1
                   ;R1 is subsequently set to R1+4
```

Condition Codes Not changed

Note the result is unpredicatable if:-

- Rd=15 (PC)

- Rd=<Op2> and the writeback option <!> is selected

Instruction Format

Opcode	31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
STR<cond>B Rd, Rn, #	cond	1 0 0	P	U	1	W	0	Rn	Rd	#
STR<cond>B Rd, Rn, #	cond	0 1 1	P	U	1	W	0	Rn	Rd	shift# shift 0 Rm

A.4 Branch

A.4.1 **B** *Branch*

Syntax B<cc> <Offset>

Description Simple Store the least significant byte in register Rd in the memory location calculated by <Op2>. If PC (R15) is used as the base register PC relative addressing is possible.

Example Pre-indexing

```
                ;Print Chapital characters
 MOV  R0, #65   ;Load R0 with ASC 'A'
loop
```

```
        BL    syswrite     ;Write character R0 to stdio
        ADD   R0, R0, #1   ;Increment R0
        CMP   R0, #90      ;Compare R0 with ASCII 'Z'
       BNE loop            ;Return to loop if R0<91
```

Post-indexing

```
       STRB R0, [R1], #4   ;Store R0 in the
                           ;memory location R1
                           ;R1 is subsequently set to R1+4
```

Condition Codes Not changed

Note the result is unpredicatable if:-

- Rd=15 (PC)

- Rd=<Op2> and the writeback option <!> is se-
  lected

Instruction Format

Opcode	31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
STR<cond>B Rd, Rn, #	cond	1 0 0	P	U	1	W	0	Rn	Rd	#
STR<cond>B Rd, Rn, #	cond	0 1 1	P	U	1	W	0	Rn	Rd	shift# | shift | 0 | Rm

A.5   Multiple Load and Store

A.6   Software Interrupts

A.7 Condition Codes

Mnemonic	Condition Test	Meaning	Uses
EQ	Z=1	**EQ**ual	Comparison equal or zero result
NE	Z=0	**N**ot **E**qual	Comparison not equal or non-zero result
CS	C=1	**C**arry **S**et	Arithmetic operation gave carry out
CC	C=0	**C**arry **C**lear	Arithmetic operation did not produce a carry
HS	C=1	**H**igher or **S**ame	Unsigned comparison gave higher or same result
LO	C=0	unsigned **LO**wer	Unsigned comparison gave lower result
MI	N=1	**MI**nus	Result is minus or negative
PL	N=0	**PL**us	Result is positive (plus) or zero
VS	V=1	o**V**erflow **S**et	Signed integer operation: overflow occurred
VC	V=0	o**V**erflow **C**lear	Signed integer operation: no overflow occurred
HI	((NOT C) OR z)=0 {C set and Z clear}	Unsigned **Hi**gher	Unsigned comparison gave
LS	((NOT C) OR Z) =1 {C set or Z clear}	unsigned **L**ower or **S**ame	unsigned comparison gave lower or same
GE	(N EOR V) {(N and V) set or (N and V) clear}	signed **G**reater than or **E**qual	Signed integer comparison gave greater than or equal
LT	(N EOR V) =1 {(N set and V clear) or (N clear and V set)}	signed **L**ess **T**han	Signed integer comparison gave less than
GT	(Z OR (N EOR V))=0 {((N and V) set or clear) and Z clear}	signed **G**reater **T**han	Signed integer comparison gave greater than
LE	(Z OR (N EOR V)) =1 {(N set and V clear) or (N clear and V set) or Z set}	**L**ess or **E**qual	Signed integer comparison gave less than or equal
AL	No test	**Al**ways	Always take the branch

B MAS Rolecks Assembler Subroutines

B.1 System Subroutines

The MAS Rolecks system simulates a number of standard system related subroutine calls, similar to those implemented in other assemblers. Within MAS Rolecks they can be called either as SWI's or as subroutines called with a `BL` call. It should be noted that to be run on the MDP Microcontroller hardware only the `BL` method of calling system subroutines is supported.

B.1.1 SYS_Exit

Terminates the current program

On Entry R1 contains the return code for the program

On Exit This routine does not return

B.1.2 SYS_Clock

Returns the number of centiseconds since the execution started.

On Entry Register R1 must contain zero. There are no other parameters.

On Exit R0 contains:

- the number of centiseconds if successful

- -1 if the call is unsuccessful (for example, because of a communications error).

B.1.3 SYS_Close

Closes a file on the host system. The handle must reference a file that was opened with SYS_Open.

On Entry On entry, r1 contains a pointer to a one-word argument block:

word 1 This is a file handle referring to an open file.

On Exit r0 contains:

- 0 if the call is successful

- -1 if the call is not successful.

B.1.4 **SYS_Open**

Opens a file on the host system.

On Entry R1 contains a pointer to the first byte of the string, containing the file name.

On Exit r0 contains:

- – 0 if the call is successful

- – -1 if the call is not successful.

r1 contains

B.1.5 **SYS_Read**

ToDo: As ADS?

B.1.6 **SYS_ReadC**

Reads a byte from the standard input.

On Entry Register R1 must contain zero. There are no other parameters or values possible.

On Exit R0 contains the byte read from the console.

B.1.7 **SYS_WriteC**

Writes a character byte, contained in R1, to the standard outputl.

On Entry R1 contains the character.

On Exit None. Register R0 is corrupted.

Comment This differs from ADS SYS_WriteC, where R1 is a pointer to the byte.

B.1.8 **SYS_Write0**

Writes a null-terminated string to the standard output.

On Entry R1 contains a pointer to the first byte of the string.

On Exit Register R0 is corrupted.

B.1.9 **SYS_Write**

Writes the contents of a buffer to a specified file at the current file position. The file position is specified either:

- explicitly, by a SYS_Seek

- implicitly as one byte beyond the previous SYS_Read or SYS_Write request.

  The file position is at the start of the file when the file is opened, and is lost when the file is closed.

  Perform the file operation as a single action whenever possible. For example, do not split a write of 16KB into four 4KB chunks unless there is no alternative.

On Entry   R1 contains a pointer to a three-word data block:

- word 1 This contains a handle for a file previously opened with SYS_Open.

- word 2 This points to the memory containing the data to be written

- word 3 This contains the number of bytes to be written from the buffer to the file.

On Exit   R0 contains: 0 if the call is successful the number of bytes that are not written, if there is an error.

B.1.10 **SYS_SysCall**

Executes a syscall on the host system

On Entry   R0 contains the syscall number R1-R7 contain the parameters required by the syscall

On Exit   Syscall specific

B.2   High-level Hardware Routines

These routines provide convenient access to the hardware, without exposing implementation details like IC bus addresses. On Rolecks the 'hardware' implementation is simply graphics in a window.

B.2.1 **MAS_LED_OnOff**

Sets LED states to either on or of in a banks of 8 LEDs at a time.

On Entry   – R0 contains in bits 0 to 7 the new state of the 8 LEDs in the bank indicated by R1

– R1 contains the LED bank number

On Exit – R0 contains 0 if the operation was successful and -1 otherwise

– R1 is preserved

B.2.2 **MAS_LED_Flash**

(optional) Sets LEDs to on, off or flashing states in banks of 8 LEDs at a time.

On Entry – R0 contains the LED bank number

– R1 contains in bits 0 to 7 the new state of the 8 LEDs in the bank indicated by R0

On Exit – R0 contains 0 if the operation was successful and -1 otherwise

– R1 is preserved

Remarks This routine is intended for use with LEDs connected to PCA9532 LED dimmers. On implementations with no such devices this routine exits with R0 containing -1.

B.2.3 **MAS_Motor**

(N.B. Note yet implemented) Sets direction and speed of motors, 1-4.

On Entry – R0 contains the motor number 0-4

– R1 contains the 'speed' of the motor. -127 (full speed anti-clockwise, 0 stopped, +127 full speed colckwise)

On Exit – R0 contains 0 if the operation was successful and -1 otherwise

– R1 is preserved

Remarks

B.2.4 **MAS_IR_Read**

Reads the current state of the infra-red sensors

On Entry R1 contains the IR bank number (usually 0)

On Exit R0 contains the state of the IR sensors in bits 0..3

B.2.5 **MAS_Slider_Read**

Reads the current state of the analogue slide control

On Entry R1 contains the slider number (usually 0)

On Exit R0 contains the slider value in bits 15..0

Comments On exit R0 will contain a 16-bit number, but it should not be assumed that the ADC that took the reading is 16-bit. The lower bits could be 0.

B.2.6 **MAS_Switch_Read**

Reads the current state of a bank of up to 8 switches

On Entry R1 contains the switch bank number

On Exit R0 contains the switch settings in bits 7..0

B.3 Mid Level Routines

These routines are convenience wrappers around the the I2C_Control or SMBus_Control routines, and access hardware attached to the MDP Micro-controller.

B.3.1 **PCF8574_Read**

Reads from a PCF8574 (or PCF8574A) device on the default IC bus.

On Entry R0 contains the 7-bit address of the PCF8574 in bits 0 to 6.

On Exit R0 contains:

- – the value read from the device in bits 0 to 7;
- – -1 if an error occurred.

B.3.2 **PCF8574_Write**

Writes to a PCF8574 (or PCF8574A) device on the default IC bus.

On Entry
- – R0 contains the 7-bit address of the PCF8574 in bits 0 to 6.
- – R1 contains 8 bits of data in bits 0 to 7.

On Exit R0 contains: 0 if the call was sucessful; or -1 if an error occurred.

B.3.3 **PCF8591_Configure**

This routine configures a PCF8591 4 channel 8-bit ADC (plus 1 channel 8-bit DAC) device on the default IC Bus. The device can be configured with either differential of ? inputs on between 2 and 4 channels.

On Entry — R0 contains a reason code

— R1 contains a parameter block

On Exit

B.3.4 **PCF8591_ReadADC**

B.3.5 **PCF8591_WriteDAC**

B.3.6 **PIC_ReadADC**

B.3.7 **PIC_WriteMotor**

B.4 ASCII Character Codes

Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex
ˆ@	0	0	SP	32	20	@	64	40	'	96	60
ˆA	1	1	!	33	21	A	65	4 1	a	97	61
ˆB	2	2		34	22	B	66	42	b	98	62
ˆC	3	3	#	35	23	C	6 7	4 3	c	99	63
ˆD	4	4	$	36	24	D	6 8	4 4	d	100	64
ˆE	5	5	%	37	25	E	6 9	4 5	e	101	65
ˆF	6	6	&	38	26	F	7 0	4 6	f	102	66
ˆG	7	7	'	39	27	G	7 1	4 7	g	103	67
ˆH	8	8	(	40	28	H	7 2	4 8	h	104	68
ˆI	9	9	)	41	29	I	7 3	4 9	i	105	69
ˆJ	10	a	*	42	2a	J	74	4a	j	106	6a
ˆK	11	b	+	43	2b	K	75	4b	k	107	6b
ˆL	12	c	,	44	2c	L	76	4c	l	108	6c
ˆM	13	d	-	45	2d	M	77	4d	m	109	6d
ˆN	14	e	.	46	2e	N	78	4e	n	110	6e
ˆO	15	f	/	47	2f	O	79	4f	o	111	6f
ˆP	16	10	0	4 8	3 0	P	80	5 0	p	112	70
ˆQ	17	11	1	4 9	3 1	Q	81	5 1	q	113	71
ˆR	18	12	2	5 0	3 2	R	82	5 2	r	114	72
ˆS	19	13	3	5 1	3 3	S	83	5 3	s	115	73
ˆT	20	14	4	5 2	3 4	T	84	5 4	t	116	74
ˆU	21	15	5	5 3	3 5	U	85	5 5	u	117	75
ˆV	22	16	6	5 4	3 6	V	86	5 6	v	118	76
ˆW	23	17	7	5 5	3 7	W	87	5 7	w	119	77
ˆX	24	17	8	5 6	3 8	X	88	5 8	x	120	78
ˆY	25	19	9	5 7	3 9	Y	89	5 9	y	121	79
ˆZ	26	1a	:	5 8	3 a	Z	90	5 a	z	122	7a
ˆ[	27	1b	;	5 9	3 b	[	9 1	5 b	}	1 23	7b
ˆ	28	1c	<	6 0	3 c	/	9 2	5 c	—	1 24	7c
ˆ]	29	1d	=	6 1	3 d	]	9 3	5 d	{	1 25	7d
ˆˆ	30	1e	>	62	3e	ˆ	94	5 e	~	126	7e
ˆ_	31	1f	?	6 3	3 f	_	9 5	5 f	Del	127	7f