sARM User Guide

The sARM is program that implements an experimental CPU simulator. It is called *experimental* because it is not yet complete, and it also incorporates facilities that are not conventionally provided by a CPU simulator.

The principal purpose of the sARM simulator is to allow students to write and test simple assembly language programs in a RISC-style, ARM-like language. In other words, it is a tool to enable students to learn assembly language. Both the simulator and its assembly language are very easy to learn.

sARM includes a range of instruction types (i.e., format) such as RISC, CISC, and stack-based formats. Real computers with such instruction formats do not exist. This simulator allows students to experiment with different architectural paradigms. You can use it to write RISC programs, CISC programs, or even stack-based programs. However, you do not have to use these additional facilities or even know about them in order to learn assembly language.

Unusually, the underlying architecture of sARM does not have an instruction format (i.e., a fixed number of bits). This is because the simulator does not simulate a real machine. Instead, it interprets assembly language instructions; that is, instructions are directly executed from text form without the source first being converted into binary from and then the binary code executed.

The simulator is written in Python version 3.3 that can be executed on most machines (PC, Mac, Linux). Interpreters for Python are freely available. The Python website is at https://www.python.org.

Because the simulator is written in source code, it can be readily modified. The simulator uses text-based output (i.e., it is not graphical windows-based program).

A program may be executed to completion, it may be executed line-by-line in a single step mode, or breakpoints introduced and execution continue to those points.

The assembly language form of an sARM instruction is:

```
Operation operand1,operand2,operand3
```

Each instruction occupies one line. Blank lines are ignored as are lines beginning with a semicolon, ;, because the semicolon introduces a comment field that is ignored.

Note that a space must appear before the operation field because any word beginning in the first column is a label

The simulator is case insensitive, so upper-case, lower-case or any combination is permitted. You could write `aDd r0,R1` and the simulator would be entirely happy.

The simulator has eight registers r0 to r7. It has 16 memory locations and an 8 location stack. Any of these values can easily be changed by modifying the source code.

Sample Fragment of Assembly Language

Here's a fragment of code source code that I used to test the simulator. Although I have set it out in columns. The only requirement is that instructions do not begin in column zero (i.e., the leftmost position) and that any text after a semicolon is ignored and treated as a comment.

The assembler accepts labels beginning in column 0 which can be used as branch target destinations are the following code demonstrates. It is also possible to equate a label to a numeric value and then use that label in instructions instead of the numeric value.

The hash symbol, #, indicates a numeric operand in an instruction (consistent with ARM programming); for example `mov r0,#4` will copy the value 4 into register r0. As in the case of the

ARM, the register order is destination, source1, source 2; for example `add r2,r4,r0` adds registers r4 and r0 and puts the result in r2

This code is ARM-like RISC with register-to-register data processing instructions like `add r1,r1,r0`. There are only two memory access instructions `ldr` (load register from memory) and `str` (store register in memory). Both these are pointer-based (i.e., register indirect) instruction that load or store data using a pointer register that is in square parentheses. For example, `ldr r2,[r1]` means load register r2 with the contents of the memory location whose address is in register r1.

The instructions in the code fragment below are all essentially the same as the ARM's corresponding instructions. The only significant difference is `rnd r2` which returns a random integer in register r2. This instruction makes it easier to test programs by generating random data (otherwise, you would have to set up data structures in memory prior to running a program).

```
; SAMPLE CODE - USED IN TESTING
     mov r0,#0
     nop
xx   add r1,r1,r0
     add r0,r0,#1
     bra xx

; PUT RANDOM NUMBERS IN 4 CONSECUTIVE MEMORY LOCATIOND
     mov  r0,#4        ;4 locations to fill
     clr  r1           ;pointer
Loop rnd  r2           ;put a random value in r2
     str  r2,[r1]      ;store in memory
     add  r1,r1,#1     ;increment pointer
     sub  r0,r0,#1     ;decrement count
     bne  Loop         ;repeat until all done

; SEARCH MEMORY FOR LARGEST VALUE
     mov  r0,#4        ;4 locations to read
     mov  r3,#0        ;dummy biggest
     clr  r1           ;pointer
xxx1 ldr  r2,[r1]      ;read from  memory break
     cmp  r3,r2        ;compare new value  break
     bgt  xxx2         ;skip on old greater than new
     mov  r3,r2        ;record new large value
xxx2 add  r1,r1,#1     ;increment pointer break
     sub  r0,r0,#1     ;decrement count
     bne  xxx1         ;repeat until all done
     stop
```

A program can be terminated by a stop instruction or an end assembler directive. Execution will also stop if an assembly error is detected. In which case you have to re-edit the source file to correct the error.

Using the Simulator

When the simulator is executed, it looks for a source program to load. This source assembly language program provides the source code to be executed and is in text form (I use Microsoft's notepad editor).

Because I found that I was running the same program frequently, in order to save time typing the name of a source text file, I provided a default name for the source file.

When the simulator is first loaded, it asks if you want to run the standard source file. If you type **y** or **yes**, it will look for the built-in file. If you type **n** or **no**, it will expect you to provide the address of the source file.

The simulator then asks whether you wish to execute in line-by-line mode with "**Enter "y" to turn off single step mode**".

If you type **y**, it will run the source program until it is terminated, an error is found, or some other trace mode is selected.

Otherwise, an instruction is executed after each carriage return (i.e., enter). Note that if you have an IN instruction that inputs an integer, you have to type your number followed by two enters: the first enter terminates the input of your integer and the second enter causes the next instruction to be executed.

The following fragments of output demonstrates part of a session with the simulator. Note that the symbol table is first listed followed by the source program after all comments and delimiters have been removed and lower to upper case conversion performed (this is for debugging purposes). The symbol table relates all names (e.g., labels) to their appropriate numerical values.

```
sARM simulator: A simple ARM style simulator
This simlates RISC CISC and stack-based code
sARM is not intended as tool for writing serious assembly programs but as a
means of introducing assembly language
(c) Alan Clements 2014

Do you wish to use the default file? Type 'Yes' or 'No' y
Symbol table
LOOP 2

Line   0     MOV R0 #4
Line   1     CLR R1
Line   2     RND R2
Line   3     STR R2 [R1]
Line   4     ADD R1 R1 #1
Line   5     SUB R0 R0 #1
Line   6     BNE LOOP
Line   7     STOP
Enter "y" to turn off single step mode

PC =  0 Registers   4   0   0   0   0   0   0   0 MOV R0 #4
Z = 0 N = 0 C = 0 V = 0 SP = 15 LR = 0 Memory [0, 0, 0, 0, 0, 0, 0, 0]
Stack = []
Registers in hexadecimal 0x4 0x0 0x0 0x0 0x0 0x0 0x0 0x0

PC =  1 Registers   4   0   0   0   0   0   0   0 CLR R1
Z = 0 N = 0 C = 0 V = 0 SP = 15 LR = 0 Memory [0, 0, 0, 0, 0, 0, 0, 0]
Stack = []
Registers in hexadecimal 0x4 0x0 0x0 0x0 0x0 0x0 0x0 0x0

PC =  2 Registers   4   0  13   0   0   0   0   0 $LOOP RND R2
Z = 0 N = 0 C = 0 V = 0 SP = 15 LR = 0 Memory [0, 0, 0, 0, 0, 0, 0, 0]
Stack = []
Registers in hexadecimal 0x4 0x0 0xd 0x0 0x0 0x0 0x0 0x0
```

Features of sARM

Here we mention a few of sARM's highlights.

Equate

The assembler directive `equ` equates a label to a value; for example `time equ 123` allows you to write `mov r1,#time` instead of `mov r1,#123`.

Input and Output

Two instructions have been provided to permit numeric input and output. These are `in r0` and `out r0`. For example, if an instruction is in r3, then the simulator will wait until you enter a valid decimal number, and then that number will be loaded into register r3.

Operating System Calls

An operating system call in the form of a *trap instruction* has been provided. This instruction is not included (counted) when the total number of instructions is printed, because it is not regarded as being part of a program. The operating system call can be used to provide operating system facilities such as input or output. At the moment, trap is used only for program display and tracing.

Currently, the sARM's trap instruction can take one or two operands. Two legal trap instructions are

`TRAP TRACE ON` (this turns on the trace mode)
`TRAP TRACE OFF` (this turns off the trace mode)

When the trace mode is on, the contents of registers are automatically printed after each instruction is executed. It's like single step except that you don't have to enter a return to execute the next instruction. Some other trap instructions are:

`TRAP SP` (display stack)
`TRAP STATUS` (display status)
`TRAP REG` (display registers)
`TRAP MEM` (display memory)

These are used to display information about the state of the processor.

Breakpoints

A breakpoint is a traditional means of debugging code. Execution continues up to the breakpoint, at which execution stops and the contents of registers and the machine status displayed.

You can add a breakpoint by including the work BREAK at the end of a line with a valid instruction; for example,

```
   ADD r1,r2,r3 break
```

The effect of the break is to cause the contents of the registers etc. to be printed after this instruction has been executed.

Sample Run of sARM

We are now going to look at the output generated by the execution of a small program. Consider the following test code in the source file.

```
; sARM test program
      nop                   ;nop (no operation) does nothing
time  EQU 23
      MOV  r1,#time         ;load r1 with literal 23
      MOV  r2,r1            ;copy r1 to r2
      CLR  r3               ;load r3 with 0 - the same as mov r3,#3
      STR  r2,[r3]  BREAK   ;store 23 in memory location 0 and print
registers
      MOV  r4,#4
      TRAP TRACE ON
      MOV  r5,#6
```

```
ADD  r0,r4,r5
SUB  r0,r0,#1
trap trace off          ;note case doesn't matter!
ADD  r1,r1,#4
add  R1,r1,#2 BREAK
ADD  r1 r2 #2           ;note comma replaced by space
STOP
```

This source file illustrates the structure of code, demonstrates that case does not matter, and shows the use of the trace instruction and the break commend. The output of a session using this code is given below. I have reformatted this slightly to make it easier to read in this document by adding new lines.

```
Do you wish to use the default file? Type 'Yes' or 'No' y

Symbol table
TIME 23

Line   0     NOP
Line   1     MOV R1 #TIME
Line   2     MOV R2 R1
Line   3     CLR R3
Line   4     STR R2 [R3] BREAK
Line   5     MOV R4 #4
Line   6     TRAP TRACE ON
Line   7     MOV R5 #6
Line   8     ADD R0 R4 R5
Line   9     SUB R0 R0 #1
Line  10     TRAP TRACE OFF
Line  11     ADD R1 R1 #4
Line  12     ADD R1 R1 #2 BREAK
Line  13     ADD R1 R2 #2
Line  14     STOP

Enter "y" to turn off single step mode y

Breakpoint at next PC = 5    Opcode  STR R2 [R3]    Instructions executed 5
PC = 4 Registers  [0, 23, 23, 0, 0, 0, 0, 0]
Memory [23, 0, 0, 0, 0, 0, 0, 0] Z= 0 N= 0 SP= 15 Link= 0
Current stack []

PC =   7   Registers [0, 23, 23, 0, 4, 0, 0, 0]
Memory [23, 0, 0, 0, 0, 0, 0]    Opcode  TRAP TRACE ON
Z 0 C 0 N 0 V 0 SP 15 Link 0   Current stack []

PC =   8   Registers [0, 23, 23, 0, 4, 6, 0, 0]
Memory [23, 0, 0, 0, 0, 0, 0]    Opcode  MOV R5 #6
Z 0 C 0 N 0 V 0 SP 15 Link 0   Current stack []

PC =   9   Registers [10, 23, 23, 0, 4, 6, 0, 0]
Memory [23, 0, 0, 0, 0, 0, 0]    Opcode  ADD R0 R4 R5
Z 0 C 0 N 0 V 1 SP 15 Link 0   Current stack []

PC =  10   Registers [9, 23, 23, 0, 4, 6, 0, 0]
Memory [23, 0, 0, 0, 0, 0, 0]    Opcode  SUB R0 R0 #1
Z 0 C 0 N 0 V 1 SP 15 Link 0   Current stack []

Breakpoint at next PC = 13    Opcode  ADD R1 R1 #2
Instructions executed 11
PC = 12 Registers  [9, 29, 23, 0, 4, 6, 0, 0]
```

```
Memory [23, 0, 0, 0, 0, 0, 0, 0] Z= 0 N= 0 SP= 15 Link= 0
Current stack []

STOP execution at PC = 14 Total number of instructions 12
>>>
```

We have entered '**y**' to load the default source program and '**y**' to turn off automatic single stepping. Execution now runs to completion here and terminates at the STOP instruction.

Registers are displayed by the BREAK command in Line 4. Registers are displayed between lines 6 and 10 when trace is first turned on and then turned off. Finally, registers are displayed at line 12 by another BREAK command.

Note that TRAP is a computer instruction that is executed by the program, whereas BREAK is an assembler directive that is not executed (i.e., it has no effect on program execution and the program counter is not modified). A BREAK is simply a command to the simulator to display registers.