

ARM Assembly Language & Programming

ARM7

Clicker question

Have you installed ARMSim#?

- A. Yes
- B. No

<http://armsim.cs.uvic.ca/>

B1.5

GNU Assembler Quick Reference

This section summarizes the more useful commands and expressions available with the GNU assembler, *gas*, when you target this assembler for ARM. Each assembly line has the format

```
{<label>:} [<instruction or directive>] @ comment
```

Unlike the ARM assembler, you needn't indent instructions and directives. Labels are recognized by the following colon rather than their position at the start of the line. The following example shows a simple assembly file defining a function *add* that returns the sum of the two input arguments:

```
.section      .text, "x"

.global      add                @ give the symbol add external linkage

add:
    ADD      r0, r0, r1 @ add input arguments
    MOV      pc, lr     @ return from subroutine
```

GNU Assembler Directives

Here is an alphabetical list of the more common *gas* directives.

Assembler directives

`.align n`

`.ascii "<string>"`

`.asciiz "<string>"`

`.byte <byte1> {, <byte2>, ...}`

`.data {<addr>}`

`.global <symbol>`

`.text {<addr>}`

`.word <word1> {, <word2>, ...}`

The ARMSim# User Guide

Table 4. SWI I/O operations (0x00 - 0xFF)

Opcode	Description and Action	Inputs	Outputs	EQU
swi 0x00	Display Character on Stdout	r0: the character		SWI_PrChr
swi 0x02	Display String on Stdout	r0: address of a null terminated ASCII string	(see also 0x69 below)	
swi 0x11	Halt Execution			SWI_Exit
swi 0x12	Allocate Block of Memory on Heap	r0: block size in bytes	r0:address of block	SWI_MeAlloc
swi 0x13	Deallocate All Heap Blocks			SWI_DAlloc
swi 0x66	Open File (mode values in r1 are: 0 for input, 1 for output, 2 for appending)	r0: file name, i.e. address of a null terminated ASCII string containing the name r1: mode	r0:file handle If the file does not open, a result of -1 is returned	SWI_Open
swi 0x68	Close File	r0: file handle		SWI_Close
swi 0x69	Write String to a File or to Stdout	r0: file handle or Stdout r1: address of a null terminated ASCII string		SWI_PrStr

Table 4. SWI I/O operations (0x00 - 0xFF)

Opcode	Description and Action	Inputs	Outputs	EQU
swi 0x6a	Read String from a File	r0: file handle r1: destination address r2: max bytes to store	r0: number of bytes stored	SWI_RdStr
swi 0x6b	Write Integer to a File	r0: file handle r1: integer		SWI_PrInt
swi 0x6c	Read Integer from a File	r0: file handle	r0: the integer	SWI_RdInt
swi 0x6d	Get the current time (ticks)		r0: the number of ticks (milliseconds)	SWI_Timer

.data

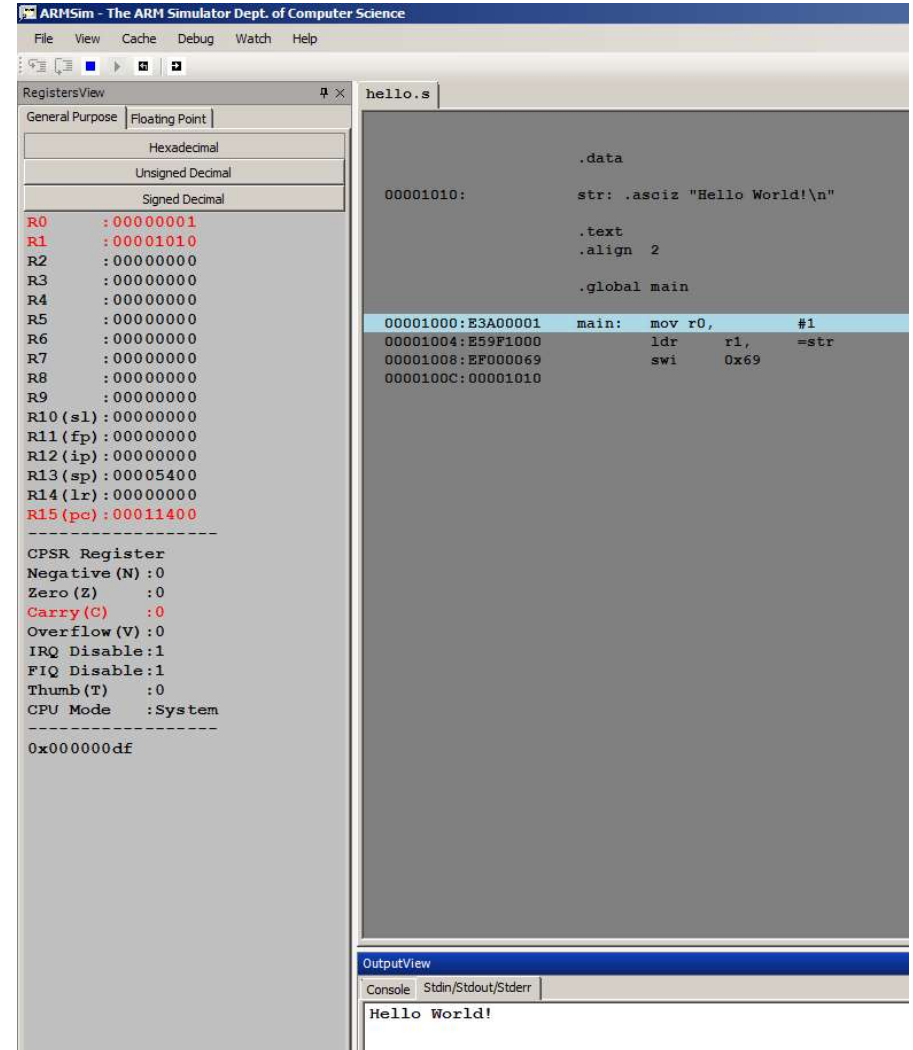
str: .asciz "Hello World!\n"

.text

.align 2

.global main

```
main: mov  r0,    #1
      ldr   r1,    =str
      swi   0x69
```



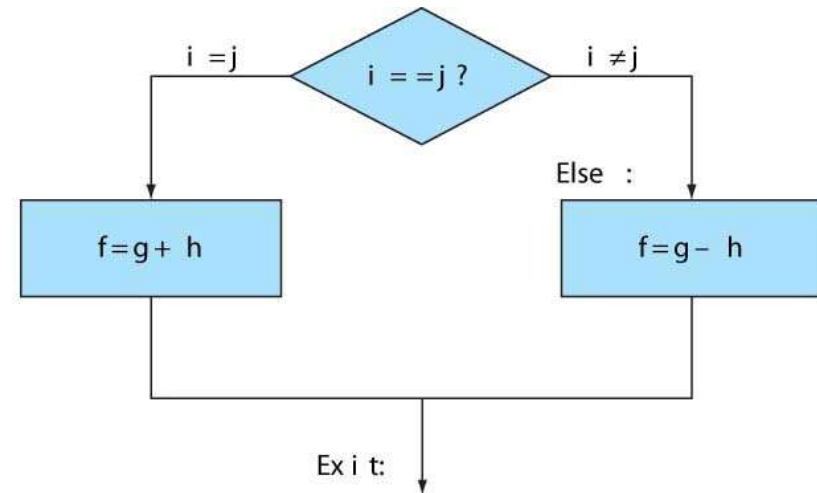
ARM Programming

Branch (cmp, beq, bne)

“If-then-else”

Example:

```
if(i==j) f = g + h;  
else    f = g - h;
```



assume f, g, h, i, and j are in r0, r1, r2, r3, and r4 respectively

CMP r3, r4

BNE Else

ADD r0, r1, r2

B Exit

Else: **sub** r0, r1, r2

Exit:

r3 - r4

A more compact and efficient version:

CMP r3, r4

ADDEQ r0, r1, r2

SUBNE r0, r1, r2

@ f = g + h (skipped if i != j)

@ f = g - h (skipped if i = j)

Signed versus unsigned comparison

Suppose

r0 = 1111 1111 1111 1111 1111 1111 1111 1111_{two}
r1 = 0000 0000 0000 0000 0000 0000 0000 0001_{two}

If the following instructions are executed

0x0000 1000: CMP r0, r1

0x0000 1004: BLO L1

0x0000 1008: BLT L2

Where will be the PC at?

A: L1

B: L2

C: 0x0000 100C

D: 0x0000 1010

Value	Meaning	Value	Meaning
0	EQ (Equal)	8	HI (unsigned Higher)
1	NE (Not Equal)	9	LS (unsigned Lower or Same)
2	HS (unsigned Higher or Same)	10	GE (signed Greater than or Equal)
3	LO (unsigned Lower)	11	LT (signed Less Than)
4	MI (Minus, <0)	12	GT (signed Greater Than)
5	PL - (Plus, >=0)	13	LE (signed Less Than or Equal)
6	VS (oVerflow Set, overflow)	14	AL (Always)
7	VC (oVerflow Clear, no overflow)	15	NV (reserved)

r0 = 1111 1111 1111 1111 1111 1111 1111 1111
r1 = 0000 0000 0000 0000 0000 0000 0000 0001

So, the result of $r0 - r1$ is computed as $r0 + (-r1) = r0 + (\sim r1 + 1)$

```

  1111 1111 1111 1111 1111 1111 1111 1111   (r0)
+) 1111 1111 1111 1111 1111 1111 1111 1111   (~r1 + 1)
-----
11111 1111 1111 1111 1111 1111 1111 1110
```

Based on this result,

N = 1 (the result is negative, treated as two's complement)

Z = 0 (the result is not zero)

C = 1 (there is carry out of the left-most bit)

V = 0 (there is no overflow)

Therefore, the instruction "BLO" is not taken because of suffix "LO" indicates unsigned lower, which is only taken when the carry bit is clear. – because if r0 is unsigned lower than r1, $r0 - r1$ will never have a carry.

Instead instruction "BLT" is taken when $N \neq V$, which is the case as shown above.

Condition Mnemoics

<i>cond</i>	Mnemonic	Name	CondEx
0000	EQ	Equal	Z
0001	NE	Not equal	\bar{Z}
0010	CS / HS	Carry set / Unsigned higher or same	C
0011	CC / LO	Carry clear / Unsigned lower	\bar{C}
0100	MI	Minus / Negative	N
0101	PL	Plus / Positive of zero	\bar{N}
0110	VS	Overflow / Overflow set	V
0111	VC	No overflow / Overflow clear	\bar{V}
1000	HI	Unsigned higher	$\bar{Z}C$
1001	LS	Unsigned lower or same	$Z OR \bar{C}$
1010	GE	Signed greater than or equal	$\overline{N \oplus V}$
1011	LT	Signed less than	$N \oplus V$
1100	GT	Signed greater than	$\bar{Z}(\bar{N \oplus V})$
1101	LE	Signed less than or equal	$Z OR (N \oplus V)$
1110	AL (or none)	Always / unconditional	ignored

CMP Compare two 32-bit integers

- | | | |
|--------------|----------------------|---------|
| 1. CMP<cond> | Rn, #<rotated_immed> | ARMv1 |
| 2. CMP<cond> | Rn, Rm [, <shift>] | ARMv1 |
| 3. CMP | Ln, #<immed8> | THUMBv1 |
| 4. CMP | Rn, Rm | THUMBv1 |

Action

- | | | | |
|---------|-------|----------------------|------------------------|
| 1. cpsr | flags | set on the result of | (Rn - <rotated_immed>) |
| 2. cpsr | flags | set on the result of | (Rn - <shifted_Rm>) |
| 3. cpsr | flags | set on the result of | (Ln - <immed8>) |
| 4. cpsr | flags | set on the result of | (Rn - Rm) |

Notes

- In the *cpsr*: $N = \langle \text{Negative} \rangle$, $Z = \langle \text{Zero} \rangle$, $C = \langle \text{NoUnsigned-Overflow} \rangle$, $V = \langle \text{SignedOverflow} \rangle$. The carry flag is set this way because the subtract $x - y$ is implemented as the add $x + \sim y + 1$. The carry flag is one if $x + \sim y + 1$ overflows. This happens when $x \geq y$ (equivalently when $x - \hat{A}y$ doesn't overflow).
- If *Rn* or *Rm* is *pc*, then the value used is the address of the instruction plus eight bytes for ARM instructions, or plus four bytes for Thumb instructions.

Example

```
CMP    r0, r1, LSR#2    ; compare r0 with (r1/4)
BHS    label            ; if (r0 >= (r1/4)) goto label;
```

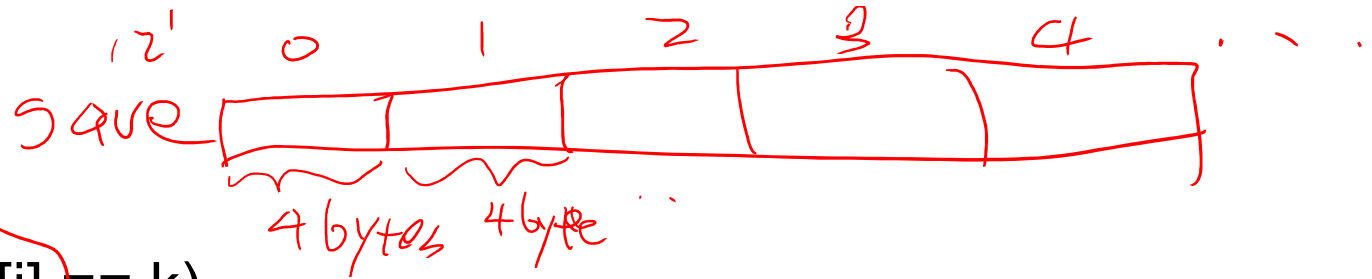
Q: In instruction CMP r0, r1, if the values of r0 have a one as their left most bit, should we assume these values are signed integers or unsigned?

A: Yes, you assume they are signed integers. The instruction "CMP r0, r1" subtracts value in r0 by value in r1. What matters is the subtraction result, though not saved, based on which the high 4 flag bits NZCV of the current program status register (CPSR) are set accordingly. It is these four bits that determine the behavior of the next conditional instruction. Although $r0 - r1$ is carried out as subtraction of two signed integers, careful analysis of the NZCV values allows us to know what would be the answer for "BLO r0, r1" if r0 and r1 are treated as unsigned.

Loop

Example:

```
while (save[i] == k)
    i += 1;
```



assume i is in $r3$, k is in $r5$, and the base of the array is in $r6$.

```
Loop:  ADD    r12, r6, r3, LSL #2
        LDR    r0, [r12, #0]
        CMP    r0, r5
        BNE    Exit
        ADD    r3, r3, #1
        B      Loop
```

Exit:

Procedure call and stack

motivation: abstraction and code reusability.

C code:

```
while(b != 0) {  
    sum = sum + a;  
    b = b - 1;  
}
```

e.g., we like to reuse the code
for multiplication $a * b$

ARM Assembly code:

@ c = a x b, assume a is in r0, b is in r1

sub	r7, r7, r7	@ initialize
Loop:cmp	r1, #0	
beq	Done	@if b == 0
add	r7, r7, r0	@sum += a
sub	r1, r1, #1	@a = a-1
b	Loop	
Done:		

How do we make use of that piece of code (let's give it a name MUL)?

E.g., $a * b + c$

solution 1: ("**cut-and-paste**": embed the code of MUL to where it is needed)
(a valid approach under certain circumstances, inline function in C++)

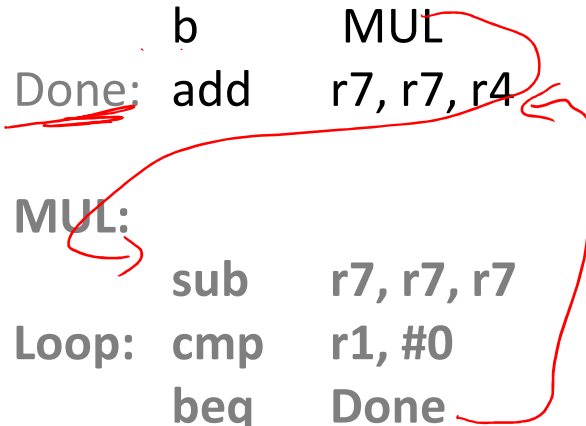
@ assume a, b, c, are in r0, r1, r7

	mov	r2, r7	@ move c out of r7, as it is to be used by MUL
	sub	r7, r7, r7	@initialize
Loop:	cmp	r1, #0	
	beq	Done	@ if b == 0
	add	r7, r7, r0	@ sum += a
	sub	r1, r1, #1	@b = b-1
	b	Loop	
Done:	add	r7, r7, r2	@add c to a*b

Solution 2: ("call-by-name": Jump to the MUL code and jump back when it is done)

@ assume a, b, c are in r2, r3, r4

	mov	r0, r2	@ move a to r0
	move	r1, r3	@ move b to r1
	b	MUL	@ call MUL, product a*b will be put in r7
<u>Done:</u>	add	r7, r7, r4	@ add c to a*b
MUL:			
	sub	r7, r7, r7	@ initialize
Loop:	cmp	r1, #0	
	beq	Done	@ if b == 0
	add	r7, r7, r0	@ sum += a
	sub	r1, r1, #1	@ b = b-1
	b	Loop	



We want "MUL" to return to the next instruction where it is called. In this example, the return address "Done" is hard coded -- so it can only return to one place.

Can we reuse the routine "MUL" multiple times?

We don't want to write the same piece of code wherever it is used; we may not know how many times it is to be used when we write the code, e.g., multiply integer a by itself for b times where b is an integer to be read from the keyboard.

So we want to write the MUL code once and reuse it.

E.g., $a*a + b*b$

For example, let's compute $a*a + b*b$, assume a, b are in $r2$ and $r3$

	mov r0, r2	@ copy a to r0
	mov r1, r2	@ copy a to r1
	b MUL	@ call MUL, product $a*a$ will be in r7
Done:	mov r4, r7	@ move $a*a$ to r4
	mov r0, r3	@ copy b to r0
	mov r1, r3	@ copy b to r1
	b MUL	@ call MUL, product $b*b$ will be in r7
Done?:	add r7, r7, r4	@ $a*a + b*b$ is put to r7
MUL:	sub r7, r7, r7	@ initialize
Loop:	cmp r1, #0	@ if $a == 0$
	beq Done	@ sum += a
	add r7, r7, r0	@ $b = b-1$
	sub r1, r1, #1	
	b Loop	

A hard-coded return address
 "Done" is not working well -- it
 can only return to one place.

Solution:

1. save the return address to a register before calling the subroutine
2. jump to the saved return address when the subroutine is done.

Hardware support:

bl ProcedureAddress

@save the return address (PC+4) into a designated
@register r14 and then jump to the address of the callee.

mov pc, r14

@ reset PC to the return address stored in register r14.

@ For example, let's compute $a*a + b*b$

@ assume a and b are in r3 and r4

mov	r0, r3	@ copy a to r0
mov	r1, r4	@ copy b to r1
bl	MUL	@ call MUL, address of next instruction is put in r14
mov	r5, r7	@ move $a*a$ to r5
mov	r0, r4	@ copy b to r0
mov	r1, r4	@ copy b to r1
bl	MUL	@ call MUL, address of next instruction is put in r14
add	r7, r7, r5	@ $a*a + b*b$ is put to r7

MUL:

	sub	r7, r7, r7	@ initialize
Loop:	cmp	r1, #0	
	beq	Done	@ if $a == 0$
	add	r7, r7, r0	@ $sum += a$
	sub	r1, r1, #1	@ $b = b - 1$
	b	Loop	
Done:	mov	pc, r14	@ this is always the last executed instruction for a procedure

Six steps

1. (caller) place parameters in a location where the procedure (callee) can access them (r0, r1, r2, r3)
 2. transfer control to the procedure. (**BL**)
-

} caller

3. acquire the storage resources needed for the procedure.
4. perform the desired task
5. place the result value in a place where the caller can access it.
6. return control to the point next to where the program is called.
(**MOV pc, r14**)

} callee

ARM conventions:

- r0 - r3, r12: registers for storing arguments or scratch registers to used by the callee (not preserved).
- r4-r11: registers that need to be preserved, if used by callee.
- lr: register storing the return address (r14)
- sp: stack pointer (r13)

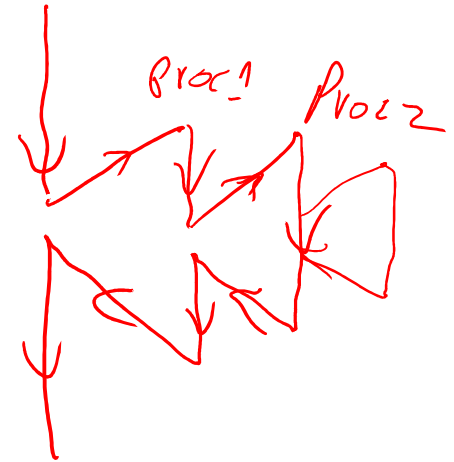
Clicker question:

What is a leaf-procedure?

- A. It is a routine you do during the fall to clean up the falling leaves.
- B. It is a procedure that does not call any other procedures
- C. It is a procedure that is not called by any other procedures
- D. It is a procedure that calls other procedures but itself
- E. It is a procedure that calls only itself.

What if a procedure calls another procedure?

	PC →	Addr:			
...		...	bl	proc1	@ r14 = *
proc1(a);		.			
...		*			
...		.			
...		.			
.		.			
proc1(A) {		.	proc1:		
...		.			
...		.			
proc2(A);		.	bl	proc2	@r14 = * *
...		**)			
}		.	mov	pc, r14	@r14 has wrong return address
		.			
proc2(B) {		.	proc2:		
...		.			
...		.			
...		.			
}		.	mov	pc, r14	

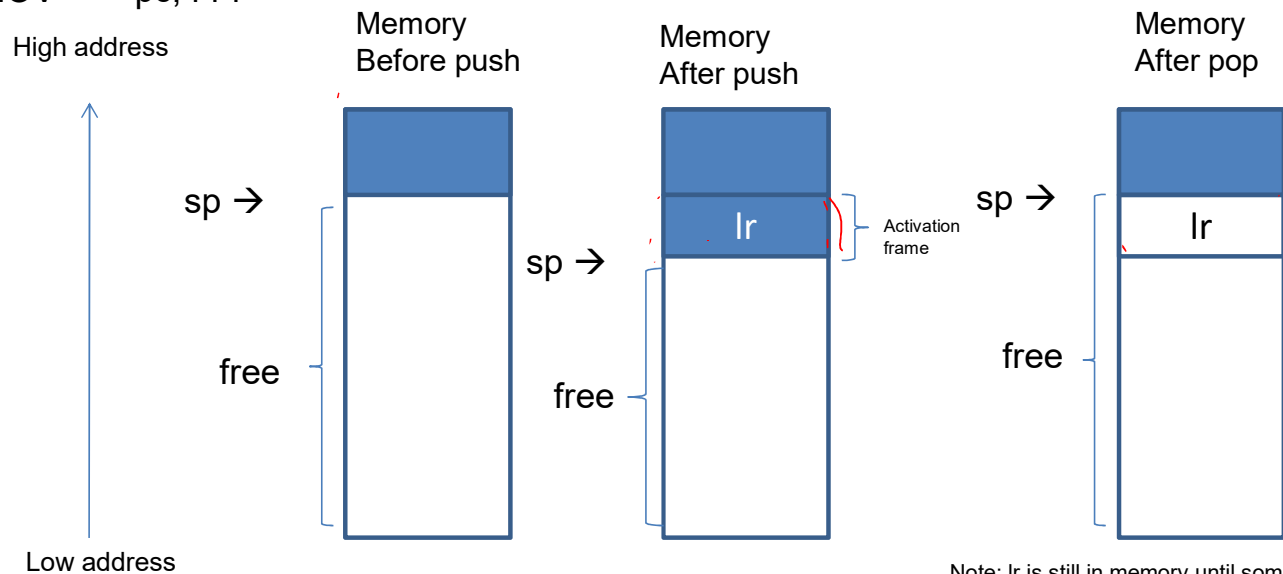


Follow the PC and monitor the value change in r14

Solution: Stack (LIFO)

Acquire storage space in main memory, the acquired space is called activation frame

```
SUB    sp, sp, #4    @ push
STR    r14, [sp]     @ push
...
...
...
LDR    r14, [sp]     @ pop
ADD    sp, sp, #4    @ pop
MOV    pc, r14
```



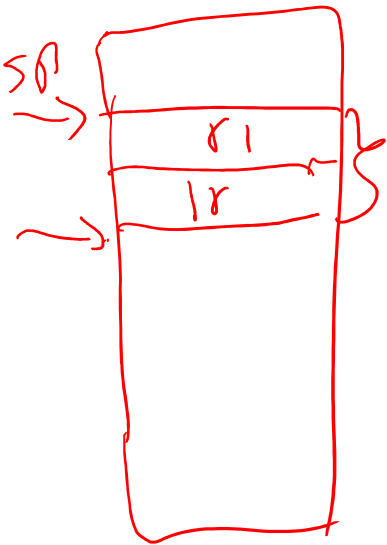
```
Main() {  
    int n = 4;  
    int res = fact(n);  
    printf(res);  
}  
  
int fact (int n) {  
    if(n < 1) return (1);  
    else return (n * fact (n-1) );  
}
```

fact(4)
= 4*fact(3)
= 4*3*fact(2)
= 4*3*2*fact(1)
= 4*3*2*1*fact(0)
= 4*3*2*1*1

```

int fact (int n) {
    if(n < 1) return (1);
    else
        return (n * fact (n-1) );
}

```



fact:

```

SUB sp, sp, #8
STR lr, [sp, #0]
STR r1, [sp, #4]

```

@move stack pointer downwards to accommodate 2 items
 @store the return address
 @store the argument n

```

CMP r1, #1
BGE Else

```

@check if n < 1
 @ branch to Else if n >= 1

```

MOV r1, #1
ADD sp, sp, #8

```

@ make a value of 1, and save it to register \$r1
 @ POP 2 items off the stack; lr and r1 haven't been overwritten,
 @ so need not to restore

```

MOV pc, lr

```

@ return

Else: SUB r1, r1, #1

@ n = (n-1)

```

BL fact

```

@ call fact with argument (n-1); lr and r1 are overwritten

```

MOV r2, r1

```

```

LDR r1, [sp, #4]

```

@ restore r1

```

LDR lr, [sp, #0]

```

@ restore r14

```

ADD sp, sp, #8

```

@ move stack pointer upwards

```

MUL r1, r1, r2

```

@ multiply fact(n-1) in r2 with n in r1 and write the result to r1

```

MOV pc, lr

```

@ return to the caller

A Programmer's Perspective

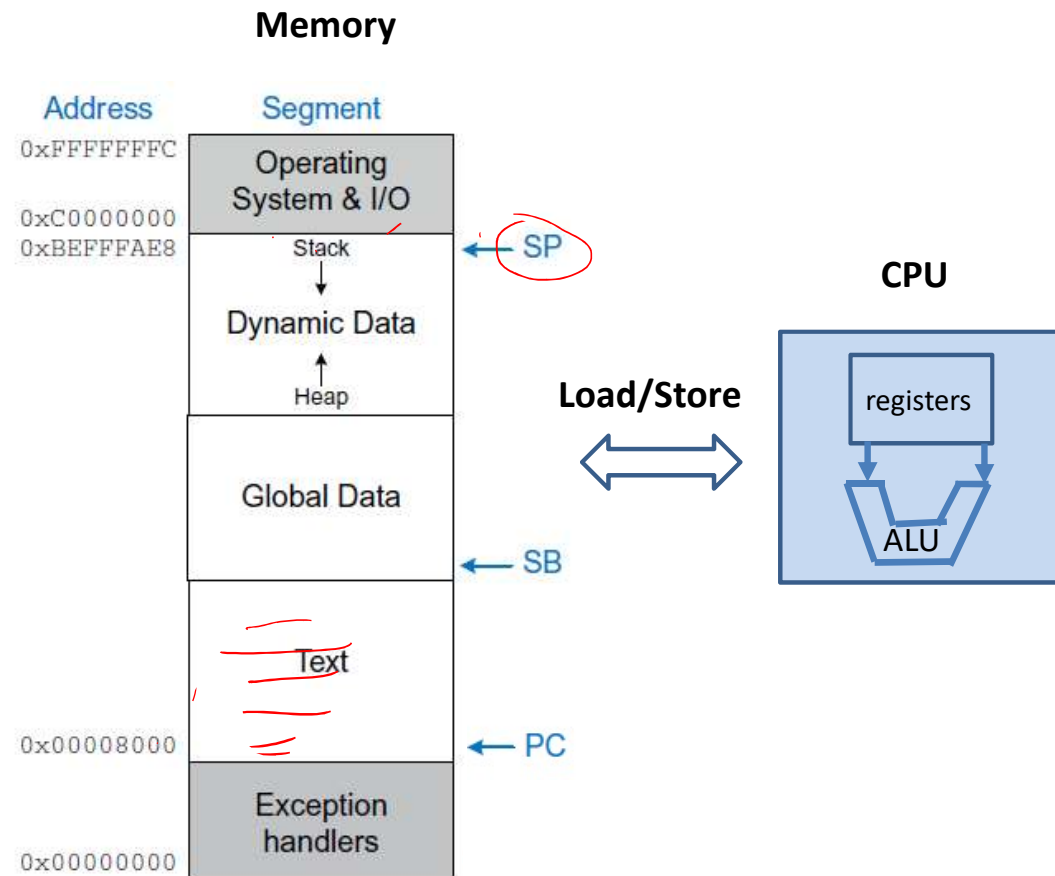


Figure 6.30 Example ARM memory map

Address:

0x 0000 1000 fact: **SUB** sp, sp, #8
 0x 0000 1004 **STR** lr, [sp, #0]
 0x 0000 1008 **STR** r1, [sp, #4]

0x 0000 100C **CMP** r1, #1
 0x 0000 1010 **BGE** Else
 0x 0000 1014 **MOV** r1, #1
 0x 0000 1018 **ADD** sp, sp, #8
 0x 0000 101C **MOV** pc, lr

0x 0000 1020 Else: **SUB** r1, r1, #1
 0x 0000 1024 **BL** fact
 0x 0000 1028 **MOV** r2, r1
 0x 0000 102C **LDR** r1, [sp, #4]
 0x 0000 1030 **LDR** lr, [sp, #0]
 0x 0000 1034 **ADD** sp, sp, #8

0x 0000 1038 **MUL** r1, r1, r2
 0x 0000 103C **MOV** pc, lr

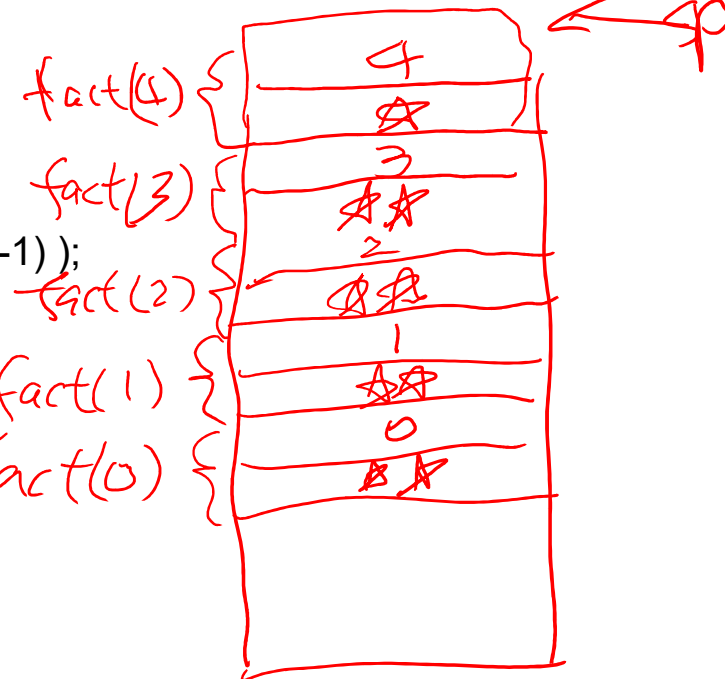
Main() {
 int n = 4;
 int res = fact(n);
 printf(res);
 }

int fact (int n) {
 if(n < 1) return (1);
 else return (n * fact (n-1));
 }

fact(4)
 = 4*fact(3)
 = 4*3*fact(2)
 = 4*3*2*fact(1)
 = 4*3*2*1*fact(0)
 = 4*3*2*1*1

r14 ← ★
 r1 ← 24

r2 ← 6



Q: The instruction `MOV r1, #1` will overwrite what's already in `r1`.
Is that a problem?

A. Yes

B. No

```

gcd(a, b) {
  if(a==b) return a;
  else if (a>b) return gcd(a-b, b);
  else return gcd(a, b-a);
}

```

gcd:

```

sub    sp, sp, #4
str    lr, [sp, #0]
cmp    r0, r1
beq    return

```

@ push
@push

```

elseif: ble    Else
          sub    r0, r0, r1
          bl     gcd
else: sub    r1, r1, r0
          bl     gcd

```

@ recursive call

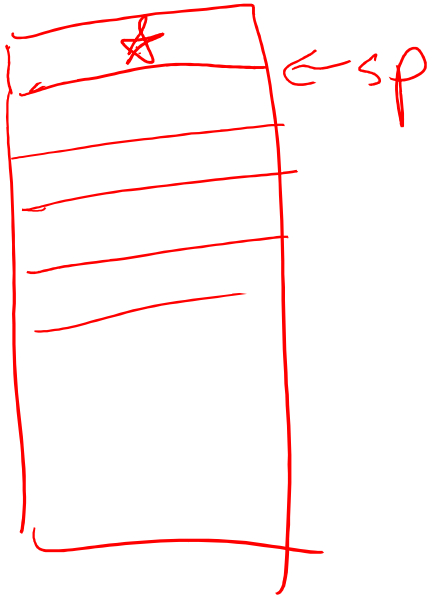
@ recursive call

```

return: ldr    lr, [sp, #0]
          add    sp, sp, #4
          move   r7, r0
          mov    pc, lr

```

@ pop
@ pop



Source code

```
while (a!=b) {  
    if (a > b) {  
        a = a - b;  
    } else {  
        b = b - a;  
    }  
}  
return a;
```



Assembler1 (using branch instructions)

```
while: cmp    r0, r1  
       beq    return  
       blt    else  
       sub    r0, r0, r1  
else:  sub    r1, r1, r0  
       b      while  
return:
```

b while

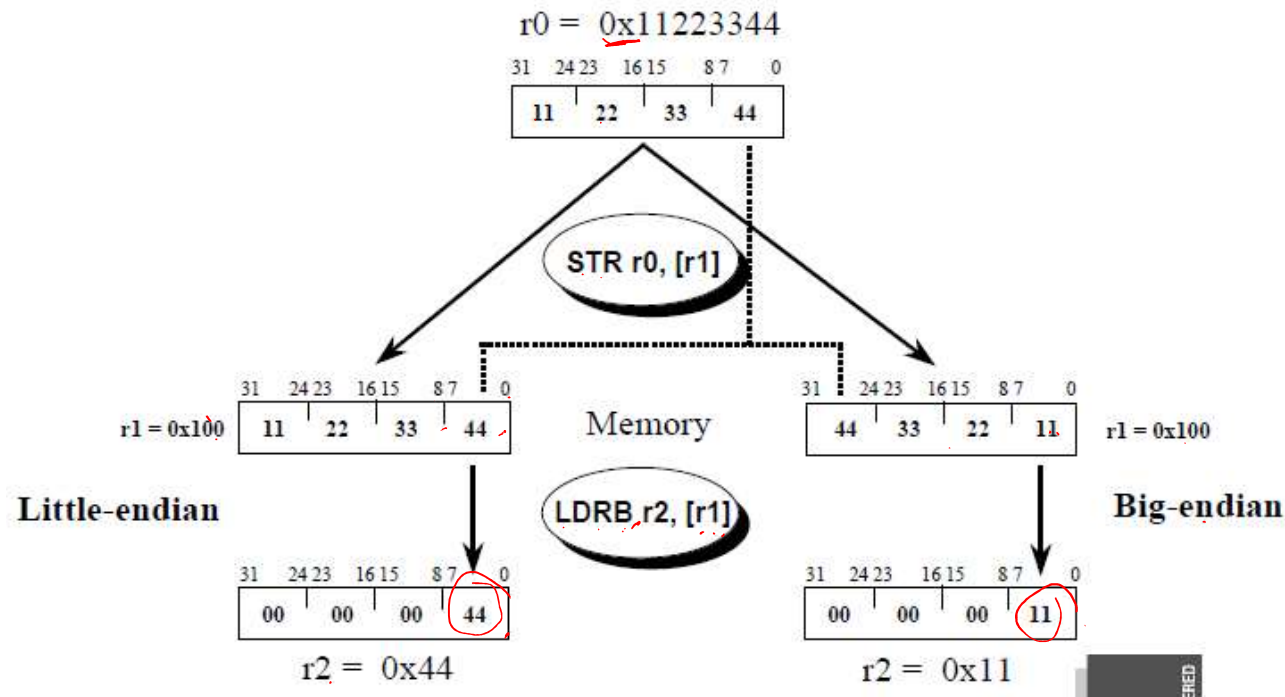
Assembler2 (using ARM conditional instructions)

```
while: cmp    r0, r1  
       subgt  r0, r0, r1  
       sublt  r1, r1, r0  
       bne    while  
return:
```

a b

Little-endian byte ordering – least-significant byte of word or half-word stored at lower address in memory

Big-endian byte ordering – most-significant byte of word or half-word stored at lower address in memory



- The ARM architecture supports both little-endian and big-endian access to memory. The ARM-Sim# supports only the little-endian format (the same as the Intel architecture which hosts the ARMSim#).
- This is only relevant when data is stored by words and is then accessed by bytes.

Etymology [edit]

In 1726, [Jonathan Swift](#) described in his satirical novel *Gulliver's Travels* tensions in Lilliput and Blefuscu: whereas royal edict in Lilliput requires cracking open one's soft-boiled egg at the small end, inhabitants of the rival kingdom of Blefuscu crack theirs at the big end (giving them the moniker *Big-endians*).^{[2][3]} The terms *little-endian* and *endianness* have a similar intent.^[4]



Wikisource has original text related to this article:

[Gulliver's Travels \(Part I, Chapter IV\)](#)

[Danny Cohen](#)'s "On Holy Wars and a Plea for Peace" published in 1980^[3] ends with: "Swift's point is that the difference between breaking the egg at the little-end and breaking it at the big-end is trivial. Therefore, he suggests, that everyone does it in his own preferred way. We agree that the difference between sending eggs with the little- or the big-end first is trivial, but we insist that everyone must do it in the same way, to avoid anarchy. Since the difference is trivial we may choose either way, but a decision must be made."

This trivial difference was the reason for a hundred-years war between the fictional kingdoms. It is widely assumed that Swift was either alluding to the historic [War of the Roses](#) or – more likely – parodying through oversimplification the religious discord in England, Ireland and Scotland brought about by the conflicts between the [Roman Catholics](#) (Big Endians) on the one side and the [Anglicans](#) and [Presbyterians](#) (Little Endians) on the other.

Credit: <http://en.wikipedia.org/wiki/Endianness>

Handling large immediate values, label addresses, words, and bytes, ...

.text

@mov r0, #345

@ see this number cannot be used as immediate value

pseudo instruction

ldr r0, =0x12345678

@ the way to load a large number to register

@ see where the number is and pc-relative addressing

ldr r1, =myByte

@ the way to load address of a label to register

ldr r2, [r1]

@ see the order of these 4 bytes in memory and in register

str r0, [r1]

@ see the 4 bytes in a word are stored in memory (little endian)

ldrb r4, [r1]

@ see which byte in 0x12345678 is loaded back

.data

myByte: .byte 1, 2, 3, 4

literal pool