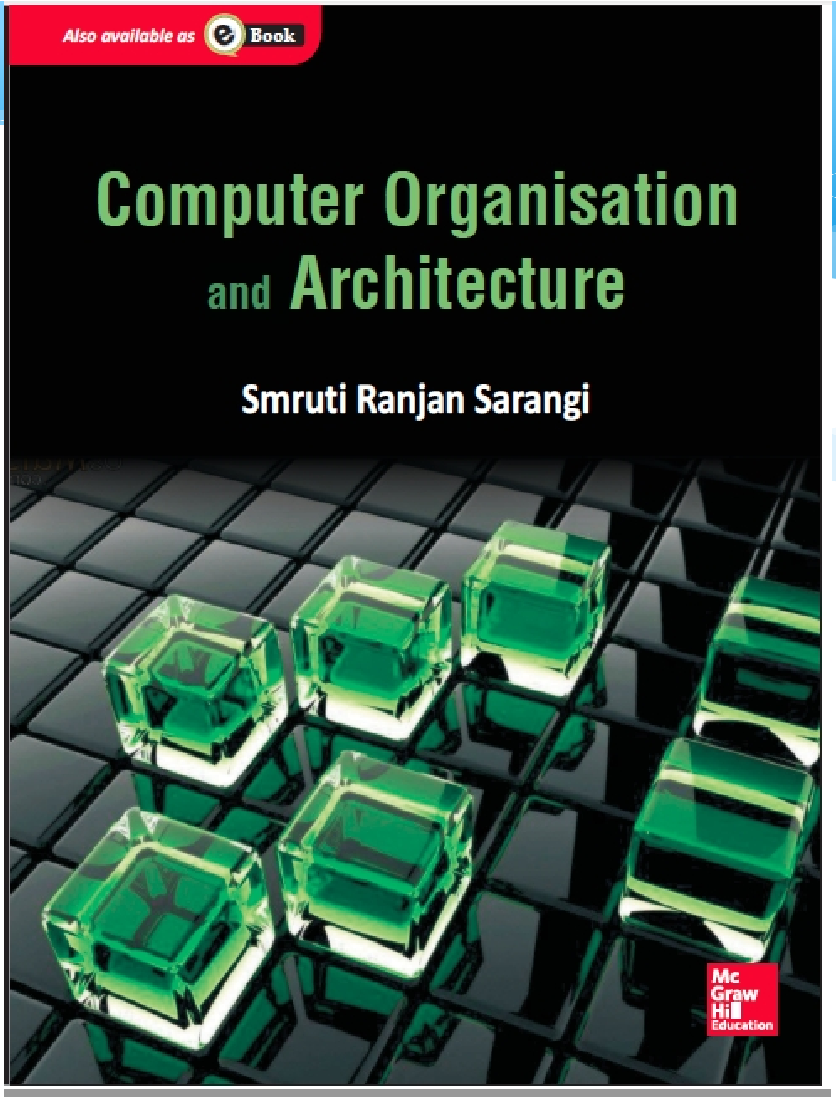
PowerPoint Slides

ARM Assembly Language (Modified Slides)

Computer Organisation and Architecture

Smruti Ranjan Sarangi,
IIT Delhi



Also available as e Book

Computer Organisation and Architecture

Smruti Ranjan Sarangi

Mc Graw Hill Education

ARM Assembly Language

- One of the most popular RISC instruction sets in use today
- Used by licensees of ARM Limited, UK
  - ARM processors
  - Some processors by Samsung, Qualcomm, and Apple
- Highly versatile instruction set
  - **Floating-point** and vector (multiple operations per instruction) extensions

Register Organisation

General registers and Program Counter

User32 / System	FIQ32	Supervisor32	Abort32	IRQ32	Undefined32
r0	r0	r0	r0	r0	r0
r1	r1	r1	r1	r1	r1
r2	r2	r2	r2	r2	r2
r3	r3	r3	r3	r3	r3
r4	r4	r4	r4	r4	r4
r5	r5	r5	r5	r5	r5
r6	r6	r6	r6	r6	r6
r7	r7	r7	r7	r7	r7
r8	r8_fiq	r8	r8	r8	r8
r9	r9_fiq	r9	r9	r9	r9
r10	r10_fiq	r10	r10	r10	r10
r11	r11_fiq	r11	r11	r11	r11
r12	r12_fiq	r12	r12	r12	r12
r13 (sp)	r13_fiq	r13_svc	r13_abt	r13_irq	r13_undef
r14 (lr)	r14_fiq	r14_svc	r14_abt	r14_irq	r14_undef
r15 (pc)	r15 (pc)	r15 (pc)	r15 (pc)	r15 (pc)	r15 (pc)

Program Status Registers

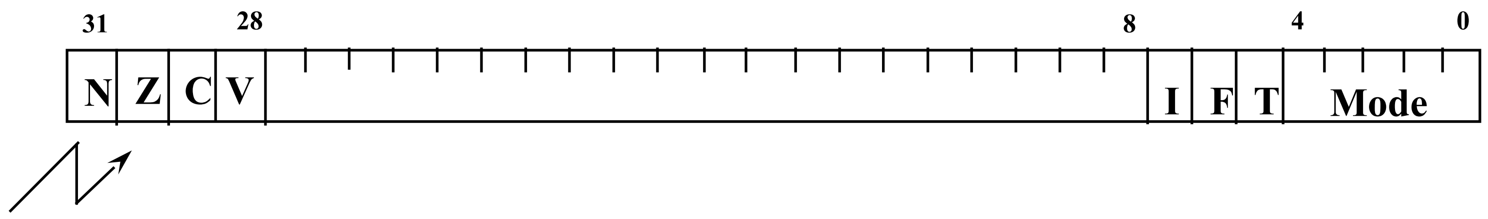
cpsr	cpsr	cpsr	cpsr	cpsr	cpsr
	spsr_fiq	spsr_svc	spsr_abt	spsr_irq	spsr_undef

ARM Machine Model

- 16 registers – r0 … r15
  - The PC is explicitly visible
- Memory (Von Neumann Architecture)

Register	Abbrv.	Name
$r11$	fp	frame pointer
$r12$	ip	intra-procedure-call scratch register
$r13$	sp	stack pointer
$r14$	lr	link register
$r15$	pc	program counter

The Program Status Registers (CPSR and SPSRs)



Copies of the ALU status flags (latched if the instruction has the "S" bit set).

* **Condition Code Flags**

N = **N**egative result from ALU flag.
Z = **Z**ero result from ALU flag.
C = ALU operation **C**arried out
V = ALU operation o**V**erflowed

* **Mode Bits**

M[4:0] define the processor mode.

* **Interrupt Disable bits.**

I = 1, disables the IRQ.
F = 1, disables the FIQ.

* **T Bit** **(Architecture v4T only)**

T = 0, Processor in ARM state
T = 1, Processor in Thumb state

Outline

- Basic Instructions
- Advanced Instructions
- Branch Instructions
- Memory Instructions

Data Transfer Instructions

Semantics	Example	Explanation
mov *reg*, (*reg*/*imm*)	mov r1, r2	r1 $\leftarrow$ r2
	mov r1, #3	r1 $\leftarrow$ 3
mvn *reg*, (*reg*/*imm*)	mvn r1, r2	r1 $\leftarrow \sim$ r2
	mvn r1, #3	r1 $\leftarrow \sim$ 3

- mov and mvn (move not)

Arithmetic Instructions

Semantics	Example	Explanation
add *reg*, *reg*, (*reg*/*imm*)	add r1, r2, r3	r1 ← r2 + r3
sub *reg*, *reg*, (*reg*/*imm*)	sub r1, r2, r3	r1 ← r2 - r3
rsb *reg*, *reg*, (*reg*/*imm*)	rsb r1, r2, r3	r1 ← r3 - r2

- add, sub, rsb (reverse subtract)

Example

Write an ARM assembly program to compute: 4+5 - 19. Save the result in r1.

Answer: *Simple yet suboptimal solution.*

```
mov r1, #4
mov r2, #5
add r3, r1, r2
mov r4, #19
sub r1, r3, r4
```

Optimal solution.

```
mov r1, #4
add r1, r1, #5
sub r1, r1, #19
```

Logical Instructions

Semantics	Example	Explanation
and *reg*, *reg*, (*reg*/*imm*)	and r1, r2, r3	r1 ← r2 AND r3
eor *reg*, *reg*, (*reg*/*imm*)	eor r1, r2, r3	r1 ← r2 XOR r3
orr *reg*, *reg*, (*reg*/*imm*)	orr r1, r2, r3	r1 ← r2 OR r3
bic *reg*, *reg*, (*reg*/*imm*)	bic r1, r2, r3	r1 ← r2 AND (∼ r3)

- and, eor (exclusive or), orr (or), bic(bit clear)

Example

Write an ARM assembly program to compute: $\overline{A \vee B}$, *where A and B are 1 bit Boolean values. Assume that* $A = 0$ *and* $B = 1$. *Save the result in* $r0$.

Answer:

```
mov r0, #0x0
orr r0, r0, #0x1
mvn r0, r0
```

Multiplication Instruction

Semantics	Example	Explanation
mul *reg*, *reg*, (*reg*/*imm*)	mul r1, r2, r3	$r1 \leftarrow r2 \times r3$
mla *reg*, *reg*, *reg*, *reg*	mla r1, r2, r3, r4	$r1 \leftarrow r2 \times r3 + r4$
smull *reg*, *reg*, *reg*, *reg*	smull r0, r1, r2, r3	$\underbrace{r1\ r0}_{64} \leftarrow r2 \times_{signed} r3$
umull reg, reg, reg, reg	umull r0, r1, r2, r3	$\underbrace{r1\ r0}_{64} \leftarrow r2 \times_{unsigned} r3$

- smull and umull instructions can hold a 64 bit operand

Example

Compute $12^3 + 1$, *and save the result in r3.*

Answer:

```
/* load test values */
mov r0, #12
mov r1, #1

/* perform the logical computation */
mul r4, r0, r0 @ 12*12
mla r3, r4, r0, r1 @ 12*12*12 + 1
```

Outline

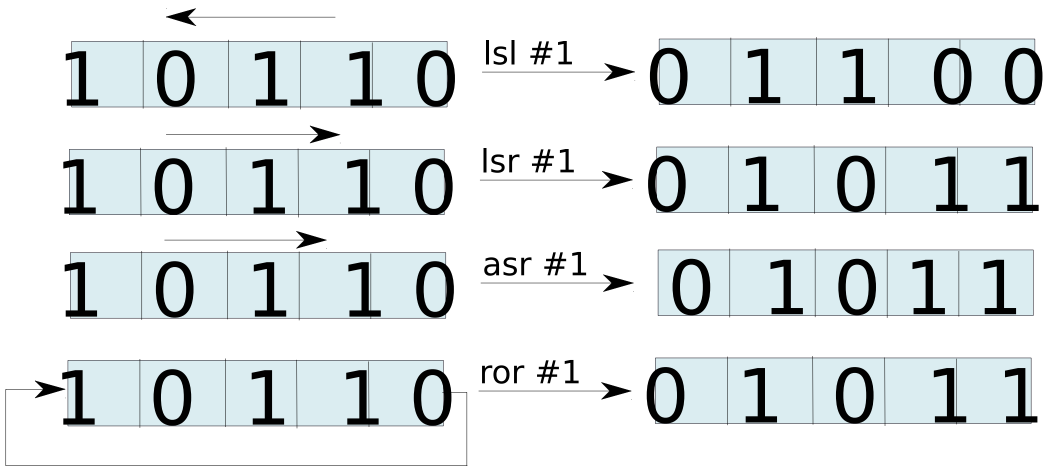
- Basic Instructions
- Advanced Instructions
- Branch Instructions
- Memory Instructions

Shifter Operands

Generic format

reg1 , lsl | lsr | asr | ror #shift_amt | reg2

Examples



Examples of Shifter Operands

Write ARM assembly code to compute: r1 = r2 / 4.

Answer:

```
mov r1, r2, asr #2
```

Write ARM assembly code to compute: r1 = r2 + r3 × 4.

Answer:

```
add r1, r2, r3, lsl #2
```

Compare Instructions

Semantics	Example	Explanation
cmp *reg*, (*reg/imm*)	cmp r1, r2	Set flags after computing (r1 - r2)
cmn *reg*, (*reg/imm*)	cmn r1, r2	Set flags after computing (r1 + r2)
tst *reg*, (*reg/imm*)	tst r1, r2	Set flags after computing (r1 AND r2)
teq *reg*, (*reg/imm*)	teq r1, r2	Set flags after computing (r1 XOR r2)

- Sets the flags of the CPSR register
  - CPSR (Current Program Status Register)
  - N (negative) , Z (zero), C (carry), F (overflow)
  - If we need to borrow a bit in a subtraction, we set C to 0, otherwise we set it to 1.

Instructions with the 's' suffix

- Compare instructions are not the only instructions that set the flags.
- We can add an s suffix to regular ALU instructions to set the flags.
  - An instruction with the 's' suffix sets the flags in the CPSR register.
  - adds (add and set the flags)
  - subs (subtract and set the flags)

Instructions that use the Flags

Semantics	Example	Explanation
adc *reg, reg, reg*	adc r1, r2, r3	r1 = r2 + r3 + Carry Flag
sbc *reg, reg, reg*	sbc r1, r2, r3	r1 = r2 - r3 - NOT(Carry Flag)
rsc *reg, reg, reg*	rsc r1, r2, r3	r1 = r3 - r2 - NOT(Carry Flag))

- add and subtract instructions that use the value of the carry flag

64 bit addition using 32 bit registers

Add two long values stored in r2,r1 and r4,r3.

Answer:

```
adds r5, r1, r3
adc r6, r2, r4
```

The (adds) instruction adds the values in r1 and r3. adc(add with carry) adds r2, r4, and the value of the carry flag. This is exactly the same as normal addition.

Outline

- Basic Instructions
- Advanced Instructions
- Branch Instructions
- Memory Instructions

Simple Branch Instructions

Semantics	Example	Explanation
b *label*	b .foo	Jump unconditionally to label .foo
beq *label*	beq .foo	Branch to .foo if the last flag setting instruction has resulted in an equality and (Z flag is 1)
bne *label*	bne .foo	Branch to .foo if the last flag setting instruction has resulted in an inequality and (Z flag is 0)

- b (unconditional branch)
- b<code> (conditional branch)

Branch Conditions

Number	Suffix	Meaning	Flag State
0	eq	equal	Z = 1
1	ne	notequal	Z = 0
2	cs/hs	carry set/ unsigned higher or equal	C = 1
3	cc/lo	carry clear/ unsigned lower	C = 0
4	mi	negative/ minus	N = 1
5	pl	positive or zero/ plus	N = 0
6	vs	overflow	V = 1
7	vc	no overflow	V = 0
8	hi	unsigned higher	$(C = 1) \wedge (Z = 0)$
9	ls	unsigned lower or equal	$(C = 0) \vee (Z = 1)$
10	ge	signed greater than or equal	N = 0
11	lt	signed less than	N = 1
12	gt	signed greater than	$(Z = 0) \wedge (N = 0)$
13	le	signed less than or equal	$(Z = 1) \vee (N = 1)$
14	al	always	
15	–	reserved	

Example

Write an ARM assembly program to compute the factorial of a positive number (> 1) stored in r0. Save the result in r1.

Answer:

ARM assembly

```
    mov r1, #1 /* prod = 1 */
    mov r3, #1 /* idx = 1 */
.loop:
    mul r1, r3, r1 /* prod = prod * idx */
    cmp r3, r0 /* compare idx, with the input (num) */
    add r3, r3, #1 /* idx ++ */
    bne .loop /* loop condition */
```

Branch and Link Instruction

Semantics	Example	Explanation
bl *label*	bl .foo	(1) Jump unconditionally to the function at .foo (2) Save the next PC (PC + 4) in the *lr* register

- We use the bl instruction for a function call

Example

Example of an assembly program with a function call.

C

```
int foo() {
    return 2;
}
void main() {
        int x = 3;
        int y = x + foo();
}
```

ARM assembly

```
foo:
    mov r0, #2
    mov pc, lr

main:
    mov r1, #3 /* x = 3 */
    bl foo /* invoke foo */
    /* y = x + foo() */
    add r2, r0, r1
```

The *bx* Instruction

Semantics	Example	Explanation
bx *reg*	bx r2	(1) Jump unconditionally to the address contained in register, r2

- This is the preferred method to return from a function.
- Instead of : mov pc, lr
  Use : bx lr

Example

```
foo:
   mov r0, #2
   bx lr

main:
   mov r1, #3 /* x = 3 */
   bl foo /* invoke foo */
   /* y = x + foo() */
   add r2, r0, r1
```

Conditional Variants of Normal Instructions

- Normal Instruction + <condition>
  - **Examples** : addeq, subne, addmi, subpl
- Also known as predicated instructions
- If the condition is true
  - Execute instruction **normally**
- Otherwise
  - Do not execute at all

*Write a program in ARM assembly to count the number of 1s in a 32 bit number stored in r*1*. Save the result in r*4.

Answer:

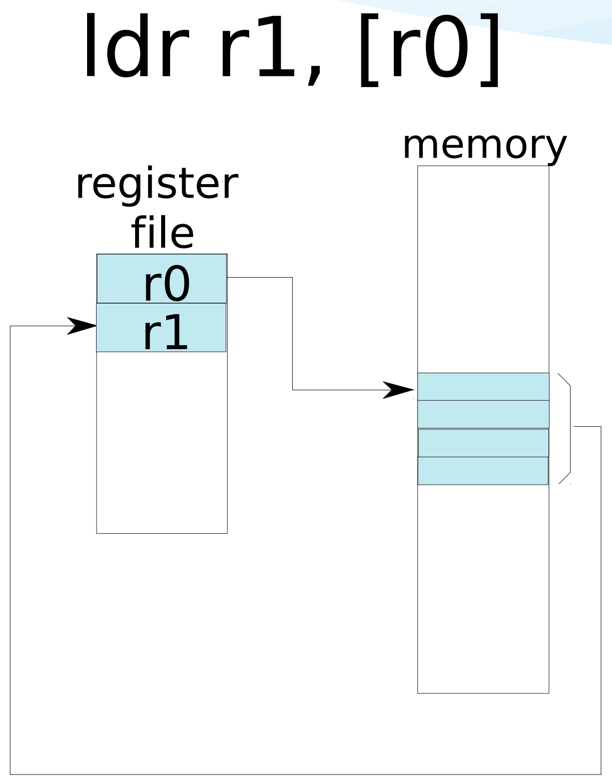
```
mov r2, #1 /* idx = 1 */
mov r4, #0 /* count = 0 */
/* start the iterations */
.loop:
    /* extract the LSB and compare */
    and r3, r1, #1
    cmp r3, #1

    /* increment the counter */
    addeq r4, r4, #1

    /* prepare for the next iteration */
    mov r1, r1, lsr #1
    add r2, r2, #1

    /* loop condition */
    cmp r2, #32
    ble .loop
```

Outline

- Basic Instructions
- Advanced Instructions
- Branch Instructions
- Memory Instructions
- Instruction Encoding

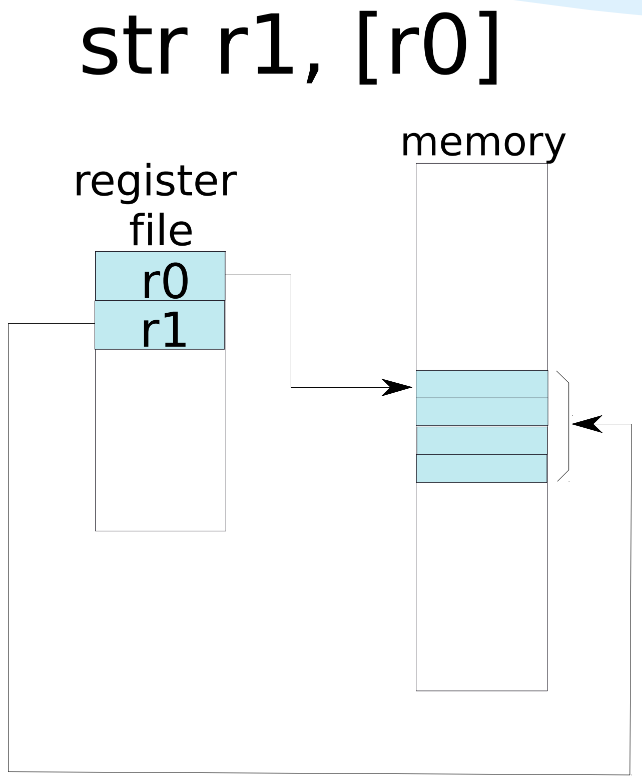
Basic Load Instruction

- ldr r1, [r0]



Basic Store Instruction

- str   r1, [r0]



Memory Instructions with an Offset

- ldr r1, [r0, #4]
  - r1 ← mem[r0 + 4]
- ldr r1, [r0, r2]
  - r1 ← mem[r0 + r2]

Table of Load/Store Instructions

Semantics	Example	Explanation	Addressing Mode
ldr *reg*, [*reg*]	ldr r1, [r0]	$r1 \leftarrow [r0]$	register-indirect
ldr *reg*, [*reg*, *imm*]	ldr r1, [r0, #4]	$r1 \leftarrow [r0 + 4]$	base-offset
ldr *reg*, [*reg*, *reg*]	ldr r1, [r0, r2]	$r1 \leftarrow [r0 + r2]$	base-index
ldr *reg*, [*reg*, *reg*, shift *imm*]	ldr r1, [r0, r2, lsl #2]	$r1 \leftarrow [r0 + r2 << 2]$	base-scaled-index
str *reg*, [*reg*]	str r1, [r0]	$[r0] \leftarrow r1$	register-indirect
str *reg*, [*reg*, *imm*]	str r1, [r0, #4]	$[r0 + 4] \leftarrow r1$	base-offset
str *reg*, [*reg*, *reg*]	str r1, [r0, r2]	$[r0 + r2] \leftarrow r1$	base-index
str *reg*, [*reg*, *reg*, shift *imm*]	str r1, [r0, r2, lsl #2]	$[r0 + r2 << 2] \leftarrow r1$	base-scaled-index

- Note the base-scaled-index addressing mode

Example with Arrays

C

```c
void addNumbers(int a[100]) {
    int idx;
    int sum = 0;
    for (idx = 0; idx < 100; idx++){
        sum = sum + a[idx];
    }
}
```

Answer:

ARM assembly

```
/* base address of array a in r0 */
mov r1, #0 /* sum = 0 */
mov r2, #0 /* idx = 0 */

.loop:
    ldr r3, [r0, r2, lsl #2]
    add r2, r2, #1   /* idx ++ */
    add r1, r1, r3   /* sum += a[idx] */
    cmp r2, #100     /* loop condition */
    bne .loop
```

Advanced Memory Instructions

- Consider an array access again
  - ldr r3, [r0, r2, lsl #2]   /* access array */
  - add r2, r2, #1   /* increment index */
- Can we fuse both into one instruction
  - ldr r3, [r0], r2, lsl #2
  - Equivalent to :
    - r3 = [r0]
    - r0 = r0 + r2 << 2

Post-indexed addressing mode

Pre-Indexed Addressing Mode

- Consider
  - ldr r0, [r1, #4]!
- This is equivalent to:
  - r0 ← mem [r1 + 4]
  - r1 ← r1 + 4

Similar to i++ and ++i in Java/C/C++

Example with Arrays

C

```
void addNumbers(int a[100]) {
    int idx;
    int sum = 0;
    for (idx = 0; idx < 100; idx++){
        sum = sum + a[idx];
    }
}
```

Answer:

ARM assembly

```
/* base address of array a in r0 */
mov r1, #0          /* sum = 0 */
add r4, r0, #400    /* set r4 to address of a[100] */

.loop:
    ldr r3, [r0], #4
    add r1, r1, r3  /* sum += a[idx] */
    cmp r0, r4      /* loop condition */
    bne .loop
```

Memory Instructions in Functions

Instruction	Semantics
ldmfd sp!, {list of registers }	Pop the stack and assign values to registers in ascending order. Update *sp*.
stmfd sp!, {list of registers }	Push the registers on the stack in descending order. Update *sp*.

- stmfd → spill a set of registers
- ldmfd → restore a set of registers

Example

*Write a function in C and implement it in ARM assembly to compute x^n, where x and n are natural numbers. Assume that x is passed through r*0*, n through r*1*, and the return value is passed back to the original program via r*0*. **Answer:***

ARM assembly

```
power:
    cmp r1, #0         /* compare n with 0 */
    moveq r0, #1           /* return 1 */
    bxeq  lr           /* return */

    stmfd sp!, {r4, lr}  /* save r4 and lr */
    mov r4, r0          /* save x in r4 */
    sub r1, r1, #1         /* n = n - 1 */
    bl power            /* recursively call power */
    mul r0, r4, r0         /* power(x,n) = x * power(x,n-1) */
    ldmfd sp!, {r4, pc}   /* restore r4 and return */
```

THE END