

Computer organization and architecture

Lesson 3

Simple instructions

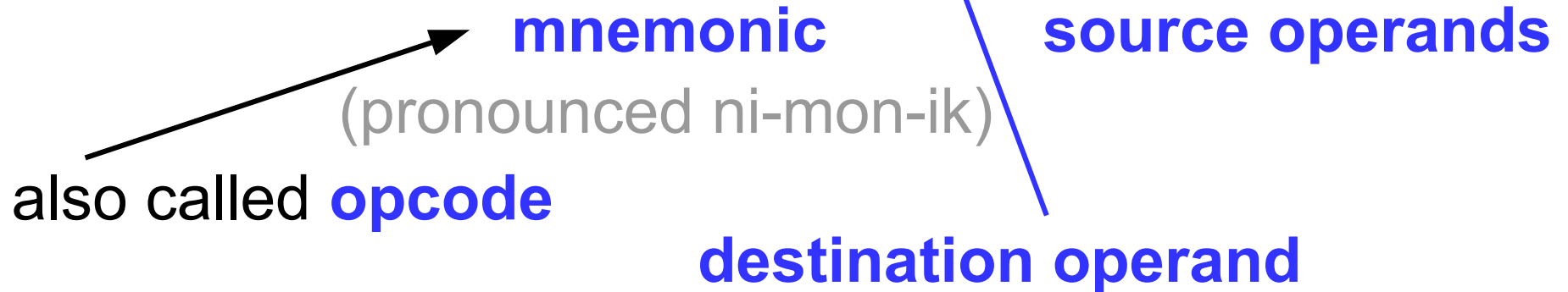
Addition

High-level

$a = b + c;$

ARM assembly

ADD a, b, c



The **mnemonic** indicates what operation to perform

Subtraction

High-level

$a = b - c;$

ARM assembly

SUB a, b, c

More complex high-level code translates into multiple ARM instructions

High-level

```
/* multiple-line  
comment */
```

```
a = b + c - d;    // single-line comment
```

ARM assembly

```
ADD t, b, c      ; t = b + c  
SUB a, t, d      ; a = t - d
```

In ARM assembly language: only single-line comments

An instruction operates on operands.

```
ADD a, b, c
```

Operands can be stored in registers or memory, or they may be **constants stored in the instruction itself.**

Operands stored as constants or in registers are accessed quickly, but they hold only a small amount of data.

Additional data must be accessed from memory, which is large but slow.

ARM register names are preceded by the letter R

```
; R0 = a, R1 = b, R2 = c  
ADD R0, R1, R2 ; a = b + c
```

This instruction adds the 32-bit values contained in R1 and R2 and writes the 32-bit result to R0

The following ARM assembly code uses a register, R4, to store the intermediate calculation:

```
; R0 = a, R1 = b, R2 = c, R3 = d; R4 = t  
ADD R4, R1, R2 ; t = b + c  
SUB R0, R4, R3 ; a = t - d
```

Example

Translate the following high-level code into ARM assembly language.

```
a = b - c;  
f = (g + h) - (i + j);
```

Solution:

```
; R0 = a, R1 = b, R2 = c, R3 = f,  
; R4 = g, R5 = h, R6 = i, R7 = j  
SUB R0, R1, R2      ; a = b - c  
ADD R8, R4, R5      ; R8 = g + h  
ADD R9, R6, R7      ; R9 = i + j  
SUB R3, R8, R9      ; f = (g + h) - (i + j)
```

Constants (Immediates)

In addition to register operations, ARM instructions can use **constant** or **immediate** operands.

These constants are called **immediates**, because their values are immediately available from the instruction and do not require a register or memory access.

High-level

```
a = a + 4;  
b = a - 12;
```

ARM assembly

```
; R7 = a, R8 = b  
ADD R7, R7, #4  
SUB R8, R7, #0xC
```

Hexadecimal constants in ARM assembly start with 0x

Single character constants: `MOV R5, #'A'`

Logical Instructions

MVN – MoVe and Not

ORR – OR

EOR – XOR

BIC – bit clear


Source registers

R1	0100 0110	1010 0001	1111 0001	1011 0111
R2	1111 1111	1111 1111	0000 0000	0000 0000

Assembly code

AND  R3, R1, R2

ORR R4, R1, R2

EOR  R5, R1, R2

BIC R6, R1, R2

MVN R7, R2

Result

R3	0100 0110	1010 0001	0000 0000	0000 0000
R4	1111 1111	1111 1111	1111 0001	1011 0111
R5	1011 1001	0101 1110	1111 0001	1011 0111
R6	0000 0000	0000 0000	1111 0001	1011 0111
R7	0000 0000	0000 0000	1111 1111	1111 1111

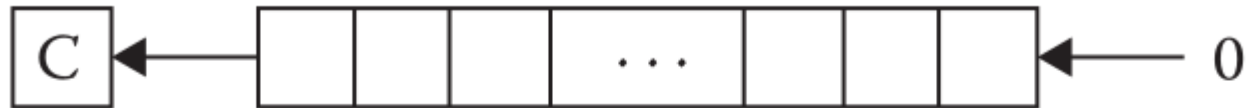
The 1st source is always a register

The 2nd source is either an immediate or another register

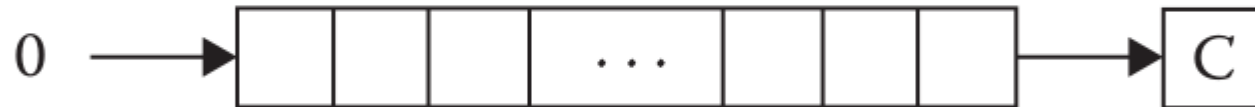
BIC clears the bits that are asserted in R2

Shift Instructions

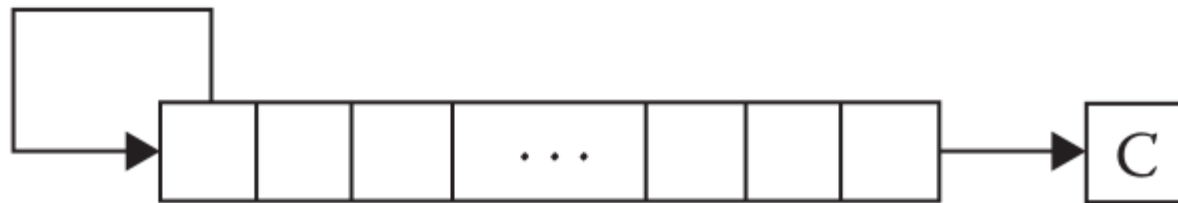
LSL Logical shift left by n bits Multiplication by 2^n



LSR Logical shift right by n bits Unsigned division by 2^n

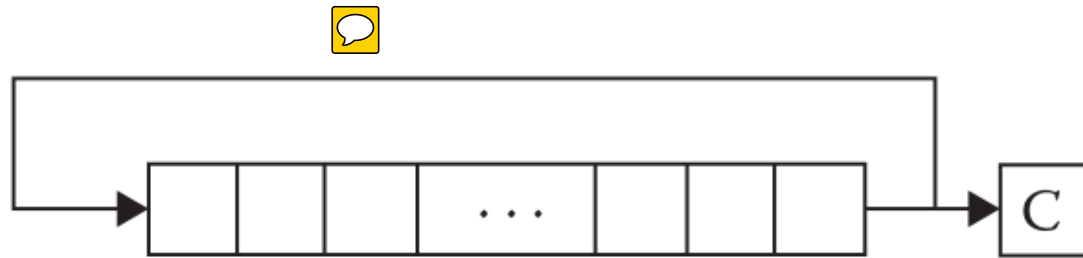


ASR Arithmetic shift right by n bits Signed division by 2^n

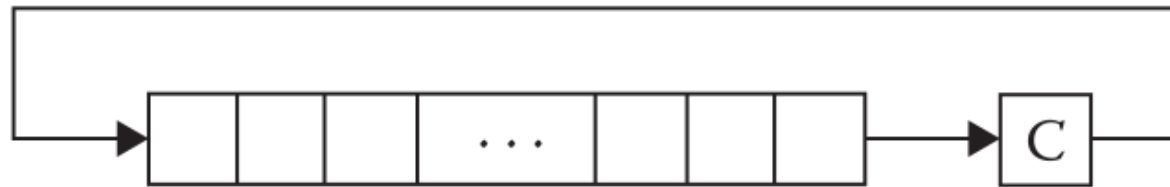


ASR – the sign bit shifts into the most significant bits

ROR Rotate right by n bits 32-bit rotate



RRX Rotate right extended by one bit 33-bit rotate. 33rd bit is Carry flag



The amount by which to shift can be an immediate or a register.

RRX provides the value of the contents of a register shifted right one bit. The old carry flag is shifted into bit[31].

There is no ROL instruction because left rotation can be performed with a ROR by a complementary amount.

Source register

R5	1111 1111	0001 1100	0001 0000	1110 0111
----	-----------	-----------	-----------	-----------

Assembly Code

Result

LSL R0, R5, #7	R0	1000 1110	0000 1000	0111 0011	1000 0000
LSR R1, R5, #17	R1	0000 0000	0000 0000	0111 1111	1000 1110
ASR R2, R5, #3	R2	1111 1111	1110 0011	1000 0010	0001 1100
ROR R3, R5, #21	R3	1110 0000	1000 0111	0011 1111	1111 1000

Source registers

R8	0000 1000	0001 1100	0001 0110	1110 0111
R6	0000 0000	0000 0000	0000 0000	0001 0100

Assembly code

Result

LSL R4, R8, R6	R4	0110 1110	0111 0000	0000 0000	0000 0000
ROR R5, R8, R6	R5	1100 0001	0110 1110	0111 0000	1000 0001

There is no ASL, or an arithmetic shift left

You would never need such an instruction, since arithmetic shifts need to preserve the sign bit, and shifting signed data to the left will do so as long as the number doesn't overflow.

Example: -1 is `0xFFFFFFFF`

shifting it left results in `0xFFFFFFFFE`,

which is -2 and correct

A 32-bit number such as `0x8000ABCD` will overflow if shifted left, resulting in `0x0001579A`, which is now a positive number.

When the final argument of basic arithmetic instruction is a register, we can optionally add a shift distance:



```
ADD R0, R0, R1, LSL #1
```

- adds a left-shifted version of R1 before adding it to R0
- R1 itself remains unchanged

The shift distance can be an immediate between 1 and 32, or it can be based on a register value:

```
MOV R0, R1, ASR R2
```

Initializing values using immediates

MOV is a useful way to initialize register values

High-level

```
i = 0;  
x = 4080;
```

ARM assembly

```
; R4 = i, R5 = x  
MOV R4, #0  
MOV R5, #0xFF0
```

The instruction `MOV R5, #0xFF1` generates an **error**

Exercise: Explain why

You can overcome this using

```
MOV R5, #0xFF0  
ADD R5, #0x1
```

Another method:

```
MOV R5, #0xFF0  
ORR R5, R5, #0x001
```

Multiply Instructions

`MUL R1, R2, R3`

- multiplies the values in R2 and R3 and places the least significant bits of the product in R1

the most significant 32 bits of the product are discarded

UMULL – unsigned multiply long

SMULL – signed multiply long

- multiply two 32-bit numbers and produce a 64-bit product

Example:

`UMULL R1, R2, R3, R4`

- performs an unsigned multiply of R3 and R4

The least significant 32 bits of the product is placed in R1

The most significant 32 bits are placed in R2

Exercise 3.1

Explain why the instruction `MOV R5, #0xFF1` generates an error.

Hint: observe the machine instructions for MOV

Try to move different values to a register, and see if it generates error.

Can you move `#0xFF` to a register?

What about `#0xFF000000`, `#0xF000000F`, `#0xF000F000`, `#4096`, `#4097` ?

Why some of them can be moved into a register, while others cannot?

Exercise 3.2

Swap the contents of two registers without using any other registers.

Place the value 0xF631024C into R0.

Place the value 0x17539ABD into R1.

When two values A and B are to be exchanged, the following algorithm could be used:

$$A = A \oplus B$$

$$B = A \oplus B$$

$$A = A \oplus B$$

The result of your program should be that the value 0xF631024C is in R1, and the value 0x17539ABD is in R0.

Exercise 3.3

The NOR instruction is not part of the ARM instruction set, because the same functionality can be implemented using existing instructions.

Write a short assembly code snippet that has the following functionality:

$R2 = R0 \text{ NOR } R1$

Use as few instructions as possible.

Put hexadecimal A into R0 and C into R1.

Which value will be in R2 as a result of your program?

Exercise 3.4

Do the same for the NAND instruction, which is also not part of the ARM instruction set

Exercise 3.5

```
MOV R0, #3
MOV R1, #4
MOV R2, #1
ADD R0, R0, R1, LSL #1
ADD R0, R1, ASR R2
```

What will be the result of this code?

Which value will be in R0 at the end of the program?

Which value will be in R1, and in R2?

Answer first without a computer, and then check your answer using Keil.

Exercise 3.6

Practice multiply instructions

a) Multiply #0xFF000000 by 2 using MUL, UMULL, SMULL

Can it be done using one of the following instructions?

MUL R2, R0, #2

or

UMULL R2, R3, R0, #2

Explain, why

b) Observe and explain the difference between

UMULL R2, R3, R0, R1

and

SMULL R2, R3, R0, R1

Exercise 3.7

Write shift routines that allow you to logically shift left or right 64-bit values that are stored in two registers.

Put the least significant 32 bits of the source in R0.

Put the most significant 32 bits of the source in R1.

Take the following 64-bit value for testing:

0xABCDEF0112345678

```
LDR R0, =0x12345678 ; LS32B
```

```
LDR R1, =0xABCDEF01 ; MS32B
```

Shift it 8 bits left to get 0xCDEF011234567800

and then 8 bits right to get 0x00ABCDEF01123456