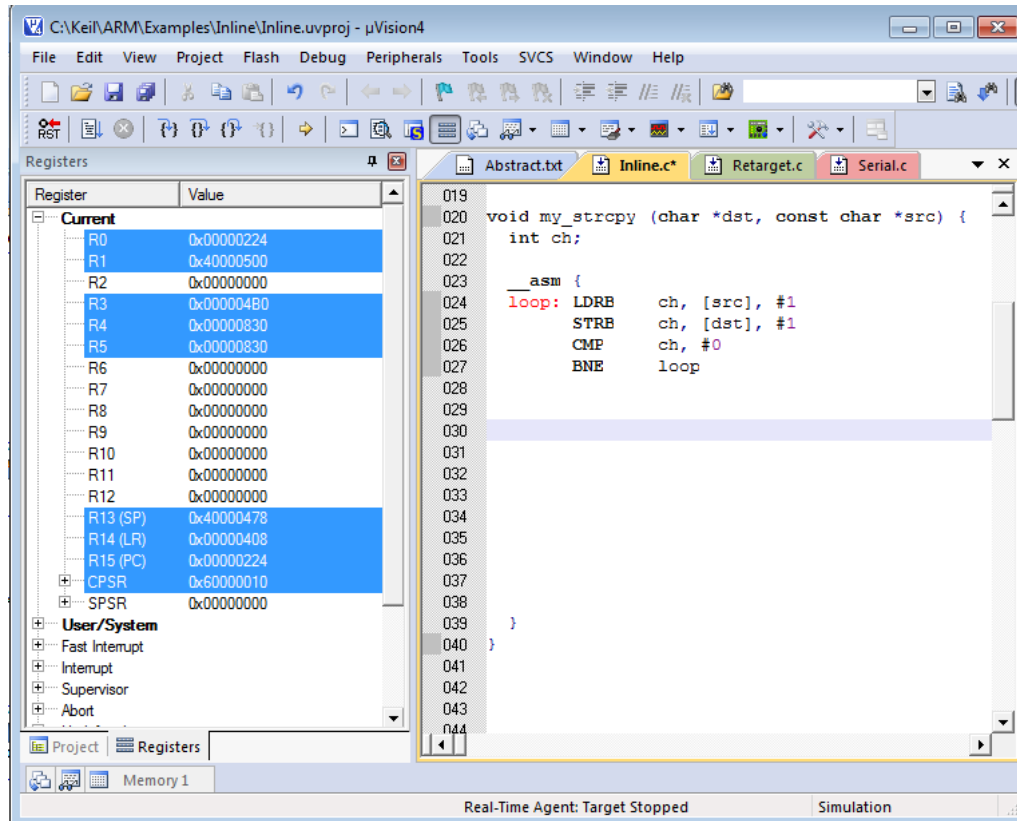
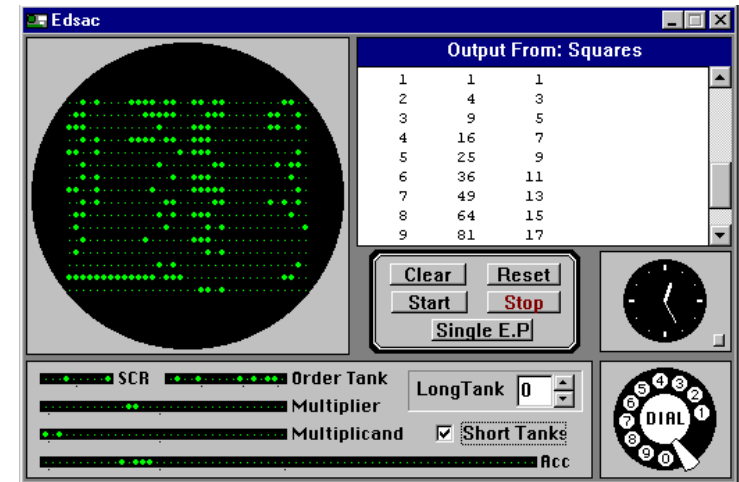


# PART 2 ARM Assembly Language



**Keil ARM simulator & debugger**



**EDSAC simulator** (Written by Martin Campbell-Kelly, Univ Warwick) – reproduces original EDSAC control panel.

**EDSAC was the first computer to be programmed in an “assembly language”. The assembler was 41 instructions long!**

# Part 2 - Contents

---

## ❖ Lecture 5 - ARM data processing

- ✦ ARM data processing instructions in detail
- ✦ ARM status flags and tests
- ✦ floating point representations: IEEE-754

## ❖ Lecture 6 – ARM Memory access

- ✦ Addressing modes for LDR/STR
- ✦ Assembler pseudo-instructions

## ❖ Lecture 7 – Execution conditions, branches, & shifts

- ✦ Comparison & test instructions
- ✦ Signed and unsigned comparison
- ✦ ARM tips & tricks
- ✦ Shifts & rotates
- ✦ multiply trick

## ❖ Lecture 8 – Subroutines, return addresses, and stacks

- ✦ Why use subroutines?
- ✦ Why use stacks?
- ✦ Implementing stacks on ARM
  - ✧ ARM Multiple register transfer instructions

## ❖ Lecture 9 - Miscellaneous

- ✦ Hardware timing
- ✦ multiply instructions

# Lecture 5 – Data Processing: ARM implementation

---

“and then the different branches of Arithmetic: Ambition, Distraction, Uglification, and Derision” The Mock Turtle - Lewis Carroll

- ❖ Arithmetic is the most complex data processing operation at an assembly language level. ARM implements 32 bit addition and subtraction. Longer calculations must make appropriate use of carries.
- ❖ We will look at:
  - ✦ ARM data processing arithmetic & logical instructions
  - ✦ Use of immediate operands in data processing instructions
  - ✦ Simple examples

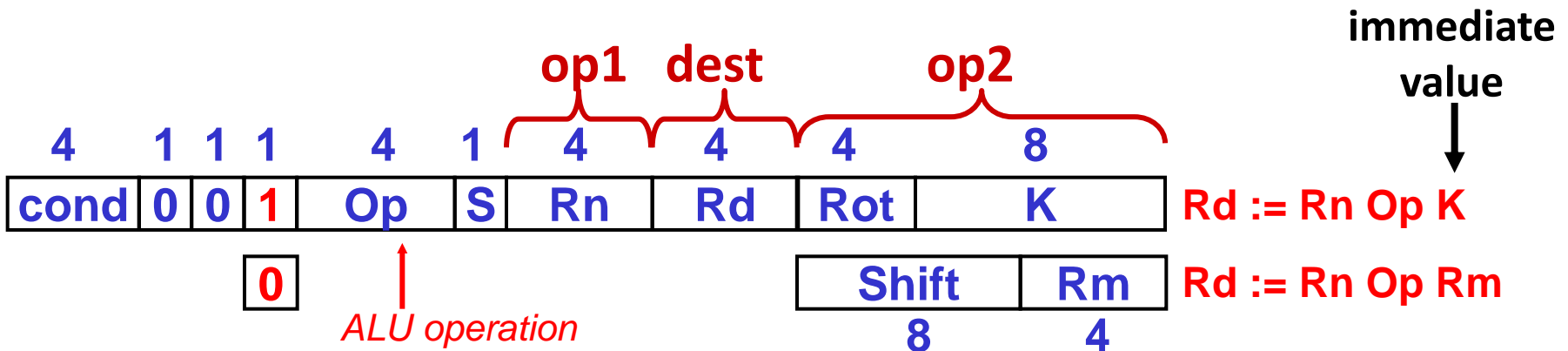
# What are data processing instructions?

---

**Rd := Rn op Rm**

- ❖ ARM data processing instructions are 3 operand.
  - ✦ d, n, m = 0-15 (any of registers R0-R15)
- ❖ Performs specified ALU operation on two operands (usually two registers).
- ❖ Write results into a destination register
- ❖ This is register-to-register - can't use memory locations
- ❖ Data processing instructions can also have side-effect of modifying condition codes
  - ✦ Condition codes preserve status info for later use

# Data processing machine word format



S bit = 1 => status bits are written

S bit = 0 => status bits unchanged

**dest := op1 op op2**

The first operand, op1, is always register Rn

The second operand, op2, is either a constant K or register Rm

This lecture: assume Shift=0, Rot=0, for unshifted Rm or immediate constant K

# ARM data processing instructions

Op	Assembly	Operation	Pseudocode
0000	AND Rd,Rn,op2	Bitwise logical AND	$Rd := Rn \text{ AND } op2$
0001	EOR Rd,Rn,op2	Bitwise logical XOR	$Rd := Rn \text{ XOR } op2$
0010	SUB Rd, Rn, op2	Subtract	$Rd := Rn - op2$
0011	RSB Rd, Rn, op2	Reverse subtract	$Rd := op2 - Rn$
0100	ADD Rd,Rn,op2	Add	$Rd := Rn + op2$
0101	ADC Rd,Rn,op2	Add with carry	$Rd := Rn + op2 + C$
0110	SBC Rd, Rn, op2	Subtract with carry	$Rd := Rn - op2 + C - 1$
0111	RSC Rd, Rn, op2	Reverse sub with carry	$Rd := op2 - Rn + C - 1$
1100	ORR Rd,Rn,op2	Bitwise logical OR	$Rd := Rn \text{ OR } op2$
1101	MOV Rd, op2	Move	$Rd := op2$
1110	BIC Rd,Rn,op2	Bitwise clear	$Rd := Rn \text{ AND NOT } op2$
1111	MVN Rd,op2	Bitwise move negated	$Rd := \text{NOT } op2$

- ❖ Here are the move and arithmetic data processing instructions.
- ❖ The operations with **Carry** allow multi-word addition and subtraction
- ❖ MOV, MVN do not use Rn, (Rn should be set to 0 in instruction word)

# Example - instruction in 32 bits

---

cond	0	0	0	Op	S	Rn	Rd	Shift	Rm
------	---	---	---	----	---	----	----	-------	----

$Rd := Rn \text{ Op } Rm$

1110	0	0	0	0100	0	0001	0000	0000,0000	0010
------	---	---	---	------	---	------	------	-----------	------

$R0 := R1 + R2$

**n**

**d**

**m**

- ❖ Op = 0100 (ADD)
- ❖ Cond = 1110 (always)
- ❖ Rd = 0 R0
- ❖ Rn = 1 R1
- ❖ Rm = Op2 = 2 R2
- ❖ S = 0 (don't write condition codes)

**ADD R0, R1, R2**

- ❖ Is translated by **assembler** into this instruction word

# Data Processing Instructions

❖ Rules that apply to ARM **data processing** instructions:

- ✦ All operands are 32 bits, come either from registers or are specified as constants (called literals) in the instruction itself
- ✦ The result is also 32 bits (arithmetic result is truncated) and is placed in a register

❖ Example:

**SUB** r0, r1, r2 ; r0 := r1 - r2

- ★ Arithmetic works for both unsigned and 2's complement signed
  - ✖ Remember from Part 1 the truncated result bits are the same!
- ★ Note that source registers are unchanged (unless dest = source)

- ◆ Result register can be the same as an input operand register:

- ✦ **ADD r0, r0, r0 ; doubles the value in r0!**



# Data Processing Instructions (1) - Arithmetic with CARRY

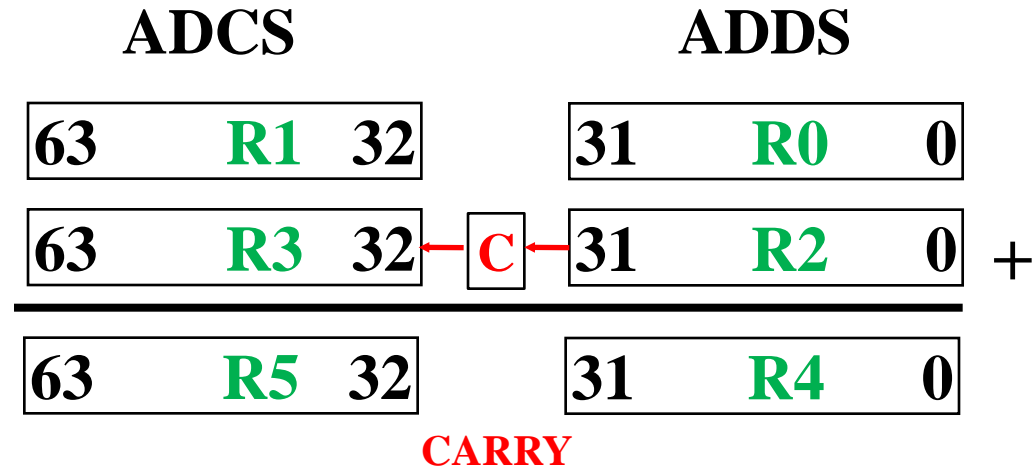
<b>ADD</b>	<b>r0, r1, r2</b>	<b>; r0 := r1 + r2</b>
<b>SUB</b>	<b>r0, r1, r2</b>	<b>; r0 := r1 - r2</b>
<b>ADC</b>	<b>r0, r1, r2</b>	<b>; r0 := r1 + r2 + C</b>
<b>SBC</b>	<b>r0, r1, r2</b>	<b>; r0 := r1 - r2 + (C - 1)</b>

- ❖ r0,r1,r2 is an example, any registers r0-r15 can be used
- ❖ Operands & result may be interpreted as unsigned or 2's complement signed integers.
- ❖ 'C' is the carry (C) status bit in the CPSR

	<b>C = 0</b>	<b>C = 1</b>
ADC	normal	+1 carry in
SBC	-1 borrow in	normal

# Multi-word addition

- ❖ "Long addition" uses multiple 32 bit words with carry condition code providing link from LSW to MSW
- ❖ In this example 2 pairs of registers (64 bit numbers) are added to make a 64 bit result in a third pair of registers
- ❖ Why are R0,R2 added with ADDS & R1,R3 added with ADCS?
- ❖ What will C be at end?



**R5:R4 := R1:R0 + R3:R2**

**ADDS R4, R2, R0**  
**ADCS R5, R3, R1**

# Example – 96 bit addition

Y: 

95	r5	64	63	r4	32	31	r3	0
----	----	----	----	----	----	----	----	---

X: 

95	r2	64	63	r1	32	31	r0	0
----	----	----	----	----	----	----	----	---

- ❖ Let's add two 96-bit numbers X and Y, storing the result back in X. This reduces the number of registers needed.
- ❖ We need **three registers** to hold each number - registers are 32 bit
  - ✦ Store X as r2:r1:r0, Y as r5:r4:r3
  - ✦ Notation:  $r2:r1:r0 \equiv r2 = X(65:64), r1 = X(63:32), r0 = X(31:0)$
- ❖ Then:  
**ADDS r0, r0, r3** ;  $r0 := r0 + r3$  (write C)  
**ADCS r1, r1, r4** ;  $r1 := r1 + r4 + C$  (write C)  
**ADCS r2, r2, r5** ;  $r2 := r2 + r5 + C$  (write C)
- ❖ "S" at the end of an instruction means you want to write the C, V, N, and Z status bits. In this case only the C flag is used.
- ❖ Similarly, if we wanted to subtract the two numbers (X-Y):  
**SUBS r0, r0, r3** ; without carry  
**SBCS r1, r1, r4** ; with carry  
**SBCS r2, r2, r5** ; with carry

# Data Processing Instructions (2) - Register Moves

---

- ❖ Here are ARM's register move operations:

MOV	r0, r2	; r0 := r2
MVN	r0, r2	; r0 := NOT r2

- ❖ Special case of data processing where one register is not used, but other options (constant op2 etc – see later) still apply.
- ❖ MVN stands for 'move negated' – **bitwise** NOT
- ❖ **This is not** two's complement negate - no addition of 1!

r2:	0101 0011 1010 1111 1101 1010 0110 1011
r0:	1010 1100 0101 0000 0010 0101 1001 0100

# Data Processing Instructions (3) - Logical Register operations

Op	Assembly	Operation	Pseudocode
0000	AND Rd,Rn,op2	Bitwise logical AND	$Rd := Rn \text{ AND } op2$
0001	EOR Rd,Rn,op2	Bitwise logical XOR	$Rd := Rn \text{ XOR } op2$
1100	ORR Rd,Rn,op2	Bitwise logical OR	$Rd := Rn \text{ OR } op2$
1110	BIC Rd,Rn,op2	Bitwise clear	$Rd := Rn \text{ AND NOT } op2$

assembler	R0=0000 0000 0000 0000 1111 1111 0000 0000 R1=0000 0000 0000 0000 1010 1010 1010 1010	Boolean op
AND R2, R1, R0	R2=0000 0000 0000 0000 1010 1010 0000 0000	and
EOR R2, R1, R0	R2=0000 0000 0000 0000 0101 0101 1010 1010	xor
ORR R2, R1, R0	R2=0000 0000 0000 0000 1111 1111 1010 1010	or
BIC R2, R1, R0	R2=0000 0000 0000 0000 0101 0101 0000 0000	bit clear

**"Bitwise logical" each bit of result is calculated from op applied to the same bit in the two operands**

# How to change condition codes

---

- ❖ Data processing instructions with **S** suffix will write condition codes
  - ✦ arithmetic writes NZCV
  - ✦ all others write NZ
- ❖ Instructions without **S** **preserve** condition codes
- ❖ **SUBS R10, R10, #1** ; decrements R10 & tests its value

Assembler	Condition codes written
ADDS, SUBS, ADCS, SBCS, RSBS, RSCS	NZCV
ANDS, MOVS, all others	NZ

```
MOV R0, #7 ; "loop counter"
LOOP      ; execute loop 7 times
....; do something
SUBS R0, R0, #1 ; sets Z
BNE LOOP ; branches if Z=0
```

# Operand 2 "literal value" option

---

- ❖ Data processing instructions have 3 operand format:

**Rd := Rn op op2**

- ❖ Destination (Rd) and first operand (Rn) - always a register
- ❖ Second operand (Op2) can be
  - ✦ A register Rm
  - ✦ An immediate (literal) value K in range 0-255

**# indicates  
literal**

<b>ADD R5, R2, #200</b>	<b>; Op2 = 200 is decimal literal value</b>
<b>ADD R5, R2, R3</b>	<b>; Op2 = R3</b>
<b>MOV R5, #-5</b>	<b>; Op2 = -5, NB note no Rn</b>
<b>MOV R5, R3</b>	<b>; Op2 = R3, NB note no Rn</b>

# Negative literal values

---

- ❖ Since literal op2 is an **unsigned** value it cannot be used directly to set a register to a negative number
- ❖ However usually this does not matter, because a different op-code can be used, giving equivalent function with a **positive** literal op2:
  - ★ **ADD r0, r1, #-11**            =>    **SUB r0, r1, #11**
  - ★ **SUB r0, r1, #-100**           =>    **ADD r0, r1, #100**
- ❖ **The assembler will do this conversion automatically**
  - ★ See next slides



# Examples



1110 00**1** 0010 0 1111 0011 0000 01100100  
 always SUB R15 R3 #100

↑  
 do not write  
 status bits

R3 := R15 - 100

ADD r3, r15, #-100

SUB r3, r15, #100

*The "ADD with negative constant" is turned into equivalent SUB automatically by assembler*

# Examples



1110 00**1** 0110 1 0100 0001 0000 00000011`  
 always SBC R4 R1 #3

↑  
 write status  
 bits N,Z,C,V

**R1 := R4 -4+C**

**ADCS r1, r4, #-4**

**SBCS r1, r4, #3**

*The ADCS is turned into  
 equivalent SBC  
 automatically*

**Why #3 not #4?**

- ❖ ADC R0, R1, #n;       $R0 := R1 + n + C$       ; C is carry
- ❖ SBC R0, R1, #n;       $R0 := R1 - n + (C - 1)$       ; (C-1) is borrow

- ❖ SBC R0, R1, R2;       $R0 := R1 - R2$       ; if C = 1
- ❖ SBC R0, R1, R2;       $R0 := R1 - R2 - 1$       ; if C = 0

correct  
notes!

The difference in polarity of C due to borrow means literal must be changed when transforming from ADC to SBC or vice versa.

This issue **only exists** for ADC, SBC, RSC, since these use C as input

Assembler	Translates to
ADC Ra, Rb, #-n	SBC Ra, Rb, #(n-1)
SBC Ra, Rb, #-n	ADC Ra, Rb, #(n-1)

# Examples



1110 00**1** 1111 1 0000 0001 0000 00000000

**MVN** ↑ **not used R1**

**K=#0**

write status  
bits N,Z

why are C,V not changed?

**R1 := -1**

**MOVS r1, #-1**

**MVNS r1, #0**

**NB:**

**NOT( n-1) = -n**

*Note that MVN negates bits (-1-x), not two's complement negation (0-x)*

*S = 1 => N,Z status bits are written*

*C,V status bits are only written on arithmetic operation*

# A real example: 64-bit checksum

- ❖ A **checksum** is often calculated to check that data has not been corrupted.

$$C = \sum_i d_i$$

- ❖ In this example 200 bytes of data is stored in memory in a buffer with start address initially in **R2**. Each 8 contiguous bytes (2 words) are interpreted as a 64 bit number  $d_i$ .
- ❖ We will use register offset addressing in a loop to add each  $d_i$  number into a 64 bit sum C held in **R3:R4**.
- ❖ Overflow from the 64 bit sum will be ignored.

Memory			
<b>R2</b>	31	0	$d_0$
<b>R2+4</b>	63	32	
<b>R2+8</b>	31	0	$d_1$
<b>R2+12</b>	63	32	
<b>R2+16</b>	31	0	$d_2$
<b>R2+20</b>	63	32	
<b>R2+24</b>	31	0	$d_3$
<b>R2+28</b>	63	32	
<b>R2+192</b>	31	0	$d_{24}$
<b>R2+196</b>	63	32	

## CHECKSUM64

```
MOV    r3, #0           ; bits 31:0 of sum
MOV    r4, #0           ; bits 63:32 of sum
ADD    r6, r2, #200      ; address of first word not added
```

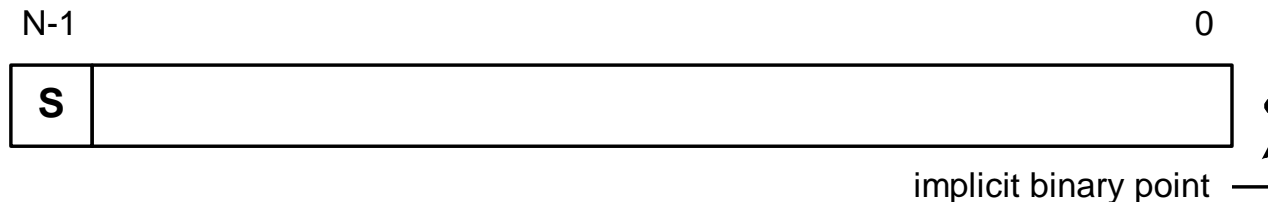
## LOOP

```
LDR    r0, [r2]          ; load 31:0 of current 64 bit word
ADD    r2, r2, #4        ; move r2 to 63:32 of current word
LDR    r1, [r2]          ; load it
ADD    r2, r2, #4        ; move r2 to 31:0 of next 64 bit word
ADDS   r3, r3, r0        ; 31:0 of 64 bit addition, set C
ADCS   r4, r4, r1        ; add bits 63:32, with C
CMP    r2, r6            ; see if next 64 bits should be added
BNE    LOOP              ; if not finished add next 64 bits
```

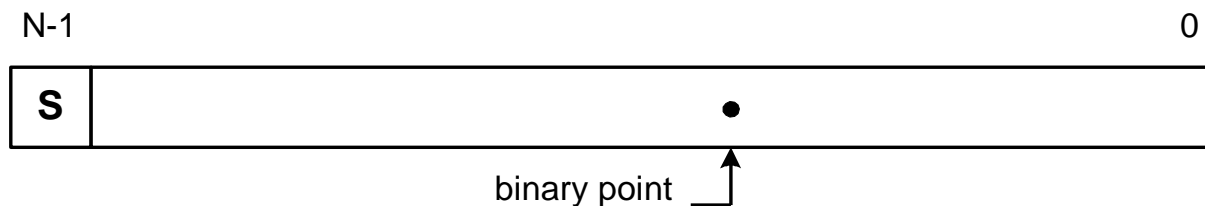
- ❖ **r2 -> current word**
- ❖ **r3,r4 -> 64 bit sum**
- ❖ **r6 -> value of r2 at which we will stop (word after 50 words)**
- ❖ **Auto-increment memory load (discussed later) would make the code much more efficient.**
- ❖ **Note that 64 bit result will overflow because MSW C is discarded**
- ❖ **Why is R6 calculated at start, not inside loop?**

# Arithmetic on real numbers

- ❖ So far, we have concentrated on **integer** representations – signed or unsigned.
- ❖ There is an implicit binary point to the right:



- ❖ In general, the binary point can be in the middle of the word (or off the end!). This is **FIXED POINT** representation of fractional numbers




- ❖ Fixed point arithmetic requires no extra hardware – the binary point is in the mind of the programmer, like signed/unsigned.

# Idea of floating point representation

---

- ❖ Although fixed point representation can cope with numbers with fractions, the range of values that can be represented is still limited.
- ❖ Alternative: use the equivalent of “scientific notation”, but in binary:

$$\text{number} = \textcolor{red}{s} \times \textcolor{blue}{m} \times \textcolor{blue}{2^e}$$



- ❖ For example:

**10.5 in binary:**  $1010.1_{(2)}$   
**Move binary point 3 places to left:**  $1.0101_{(2)} \times 2^3$

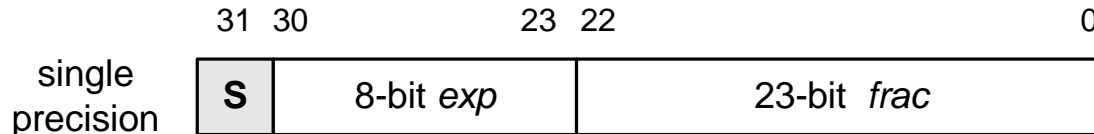
$$10.5 = 1.3125 \times 8$$

- ❖ Thus by choosing the correct exponent any number can be represented as a fixed point binary number multiplied by an exponent
- ❖ Equivalently, the binary point is "floating"



# IEEE-754 standard floating point

- ❖ 32-bit single precision floating point:



$$x = -1^s \times 2^{exp-127} \times 1.frac$$

$$5.9 \times 10^{-39} < |x| < 3.4 \times 10^{38}$$

**Why not  
exponent  
= 128?**

- ❖ MSB *s* is sign-bit: 1 => negative

- ❖ Exponent = *exp* - 127

Note this gives exponent in range [-127,127], and **special case** *exp*=255

- ❖ The MSB of the mantissa is ALWAYS '1', therefore it is not stored

✦ mantissa = 1 + *frac*\*2<sup>-23</sup> (mantissa = 1.*frac*):

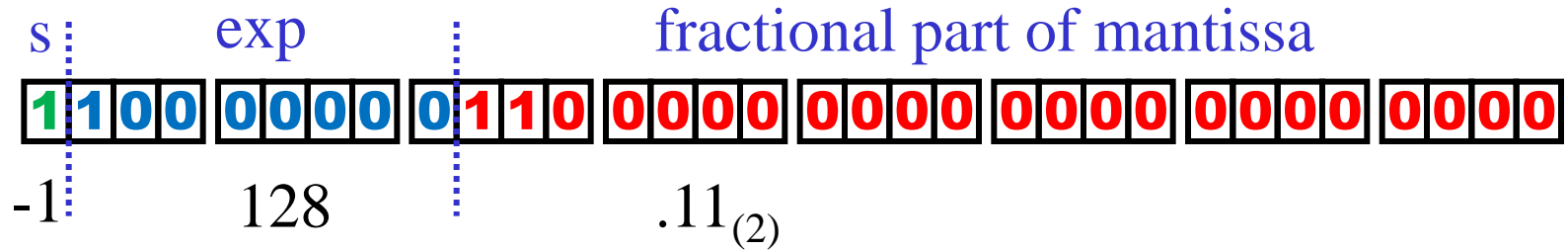
- ❖ Special cases which break this rule:

✦ *exp* field = 0, *frac* field = 0 => number is +/- 0

✦ *exp* field = 255, *frac* field = 0 => +/- ∞

✦ *exp* field = 255, *frac* field ≠ 0 => NaN (invalid number)

# IEEE-754 → Decimal



❖ The number above,  $C0600000_{(16)}$ , must have negative sign,

❖ (subtract 127) Exponent =  $exp - 127 = 1$ ,

❖ (add 1.) mantissa =  $1 + 0.11_{(2)} = 1.11_{(2)}$

$$- 2^1 \times 1.11_{(2)} = -11.1_{(2)} = -3.5$$

Note leading  
1.0 is always  
added to frac

# Decimal → IEEE 754

❖  $17.5_{(10)} = 35_{(10)} * 2^{-1} = 10001.1_{(2)} = 2^4 * 1.00011_{(2)} \rightarrow$

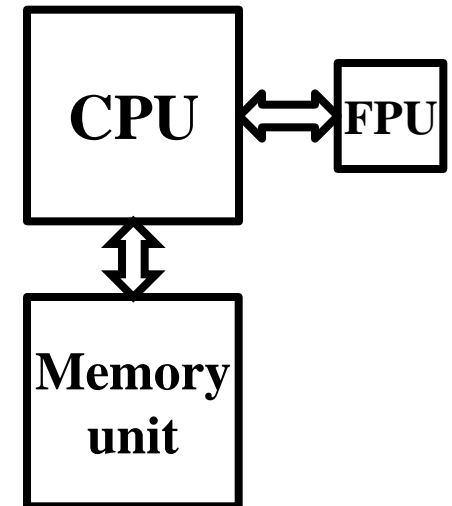
(add 127)  $exp = 4 + 127 = 131 = 10000011_{(2)}$

(extract bits to right of binary point)  $frac =$   
 $000110000000000000000000_{(2)}$

$s = 0$  (positive)

$0100\ 0001\ 1000\ 1100\ 0000\ 0000\ 0000\ 0000$

4      1      8      C      0      0      0      0



- ❖ Floating point is typically handled by Floating Point coprocessor (FPU) separate from but connected to the CPU. ARM architecture has FPUs, see latest ARM datasheets for more details.

- ❖ We will not consider FPU instructions in this course.

# Lecture 6 - Data Transfer Instructions (Load/Store)

---

"Computer programmers don't byte, they nibble a bit" - Unknown

- ❖ This lecture will examine in detail the ARM LOAD/STORE instructions
  - ✦ Multiple register load/store instructions will be dealt with separately, when we are discussing stacks.
- ❖ The ARM architecture has
  - ✦ some clever tricks which mean that memory locations close to the PC can easily be accessed.
  - ✦ special support for sequential data access

# Why use LOAD/STORE?

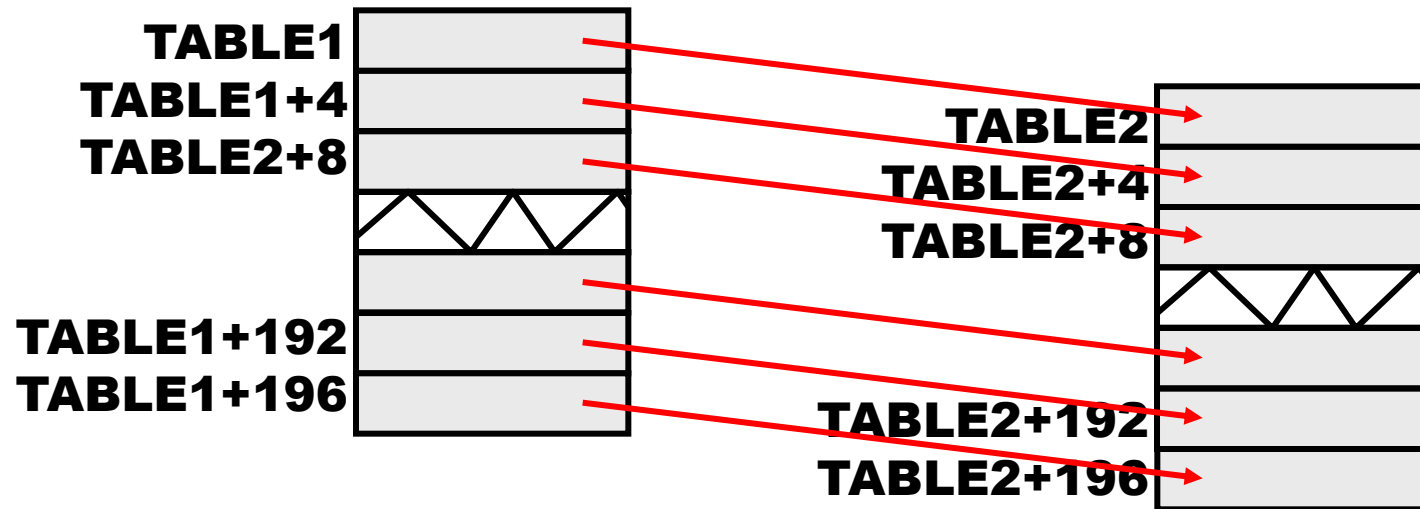
- ❖ 1. Calculations can be done quickly using registers, with no use of memory.
- ❖ 2. Calculations on memory locations require LOAD of input data to registers followed by STORE of results back to memory
- ❖ 3. Since number of registers is limited all practical programs need to use memory locations as well as registers
  - ✦ Exactly how intermediate results are transferred between registers and memory locations (register allocation) is interesting and an important research field in Computer Science
  - ✦ When programming in a high level language you don't care. The compiler will manage data, storing it in memory or registers as necessary. Clever optimising compilers will look at the whole program and optimise register allocation for most efficient execution
  - ✦ Not part of this course, but those interested could start by reading:

[http://en.wikipedia.org/wiki/Register\\_allocation](http://en.wikipedia.org/wiki/Register_allocation)

# Example of LOAD/STORE usage

## ❖ Block memory copy.

- ★ A block of memory at address TABLE1 is copied to address TABLE2.
  - ✖ Both TABLE1 & TABLE2 are word-aligned (address divisible by 4)
  - ✖ The copy operation can be implemented by moving words
- ★ The size of this block is  $200_{(10)}$  bytes, or  $50_{(10)}$  words.



# Block copy solutions

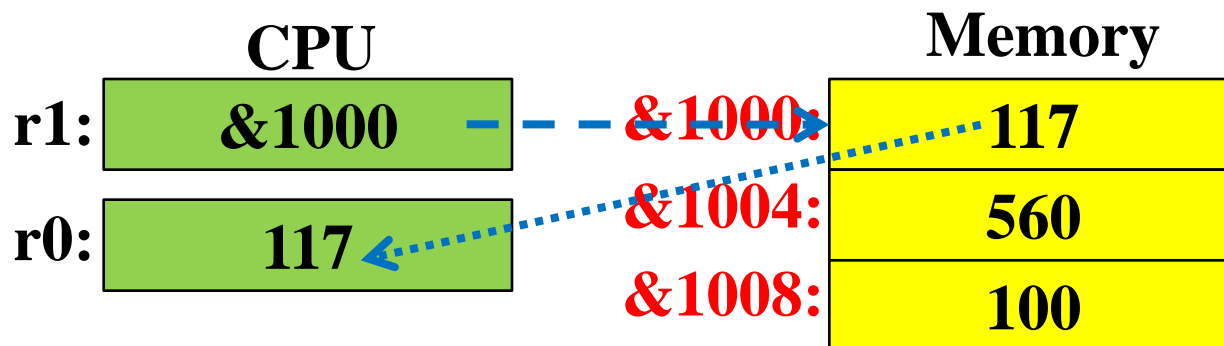
---

- ❖ Block copy illustrates how to use different addressing modes, so let's see how to do this with ARM LDR & STR **single register data transfer** instructions.
- ❖ ARM has many ways to implement block copy - we will come back to this topic later in the course with **load/store multiple register** instructions.

# Data Transfer Instructions – single register load/store instructions

- ❖ Basic operation
- ❖ Use a value in one register (called the **base** register) as a memory **address** and either load the **data** value from that address into a destination register or store the register value to memory:  

LDR	r0, [r1]	; r0 := mem <sub>32</sub> [r1] (r0 dest)
STR	r0, [r1]	; mem <sub>32</sub> [r1] := r0 (r0 source)
- ❖ This is called **register addressing (AKA indexed)**
- ❖ Here r1 is used as a **memory pointer (AKA index register)**
- ❖ **LDR r0, [r1]** ; this is a word transfer, r1 must be a word address (divisible by 4)





# Data Transfer Instructions –

## Set up the address pointer with ADR

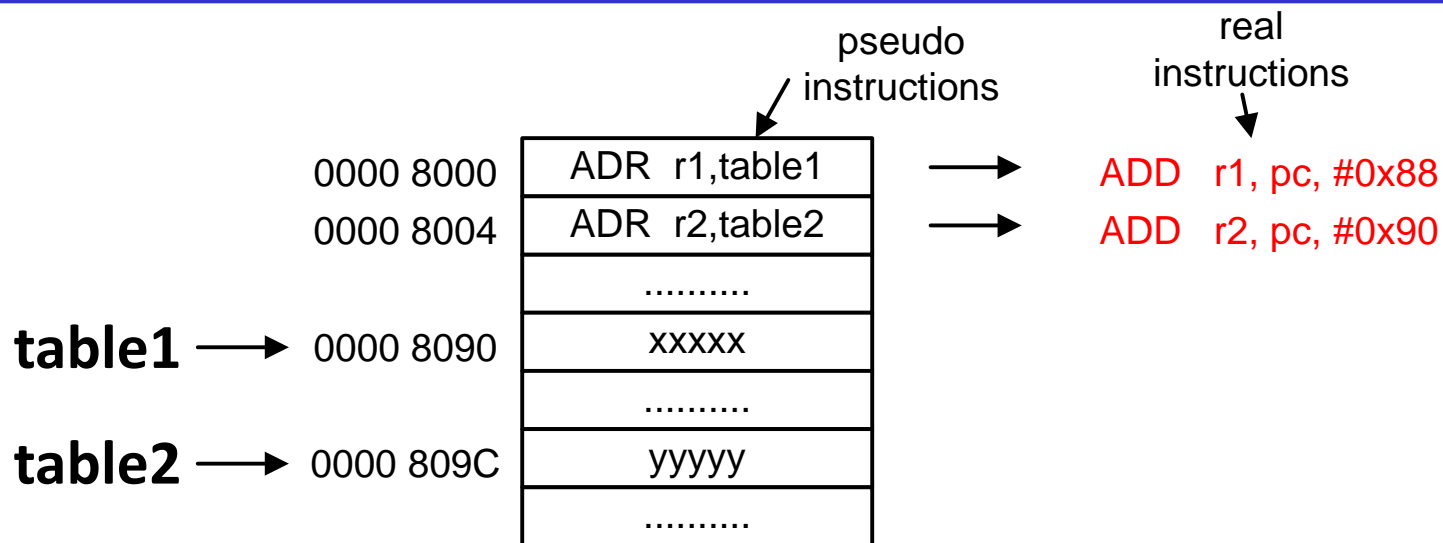
- ❖ Need to initialize address in r1 in the first place.  
How?
- ❖ ADR is a **pseudo instruction** - looks like normal instruction, but it is actually an assembler **directive**.
  - ✦ ADR sets a *register* to the *address* of an assembler label.
  - ✦ ADR translates to a machine instruction which moves a constant value into a register, see next slide.

Value of r1 = address  
of TABLE1

**Symbols** (TABLE1) in  
assembler represent  
**numeric addresses**

❖	copy	ADR	r1, TABLE1	; r1 points to TABLE1
		ADR	r2, TABLE2	; r2 points to TABLE2
		LDR	r0, [r1]	; load first <i>word</i> ....
		STR	r0, [r2]	; and store it in TABLE2
		.....		
	TABLE1	.....		; <source of data>
		.....		
	TABLE2	.....		; <destination of data>

# Data Transfer Instructions – ADR instruction



- ❖ TABLE1 address is 32-bit, difficult to put a 32-bit address value in a register in the first place (op2 literals are 8 bit)
- ❖ Solution: Program Counter PC (**r15**) is close to TABLE1
- ❖ **ADR r1, TABLE1** is translated into a data processing instruction that adds or subtracts a constant to PC (**r15**), and puts the result in r1
- ❖ This constant is known as a **PC-relative offset**, and it is calculated as:  
 $\text{addr\_of\_TABLE1} - (\text{<address of instruction>} + 8)$ 
  - ✦ (+8 is because of hardware pipelining in ARM7 CPU, see Part 3)

# Data Transfer Instructions – Moving multiple data items

- ❖ Extend the copy program further to copy NEXT word:

```
copy      ADR    r1, TABLE1      ; r1 points to TABLE1
          ADR    r2, TABLE2      ; r2 points to TABLE2
          LDR    r0, [r1]          ; load first value ....
          STR    r0, [r2]          ; and store it in TABLE2
          ADD    r1, r1, #4         ; step r1 onto next word
          ADD    r2, r2, #4         ; step r2 onto next word
          LDR    r0, [r1]          ; load second value ...
          STR    r0, [r2]          ; and store it
          .....

```

- ❖ Simplify this code with **immediate offset addressing mode**

**LDR r0, [r1, #4] ; r0 := mem<sub>32</sub> [r1 + 4]**

base  
address

offset

effective  
address

# Data Transfer Instructions – immediate offset

copy	ADR	r1, TABLE1	; r1 points to TABLE1
	ADR	r2, TABLE2	; r2 points to TABLE2
	LDR	r0, [r1]	; load first value ....
	STR	r0, [r2]	; and store it in TABLE2
	LDR	r0, [r1, #4]	; load second value ...
	STR	r0, [r2, #4]	; and store it
	.....		

- ❖ Immediate offset addressing does not change the base register (r1 & r2 here).
- ❖ To copy 50 words this way we need 100 instructions.

# Data Transfer Instructions – immediate-offset with auto-indexing

- ❖ Immediate offset addressing does not change the base register (r1 & r2 here).
- ❖ Sometimes, it is useful to modify the base register to point to the new address. This is achieved by adding a '!', and is immediate-offset addressing with **auto-indexing**:

```
LDR    r0, [r1, #4]!    ; r0 := mem32 [r1 + 4]
                        ; r1 := r1 + 4
```

- ❖ The '!' indicates that the instruction should update the base register after the data transfer. Here is how we can rewrite COPY as a loop

```
COPY      ADR    r1, TABLE1 - 4    ; r1 points to TABLE1
          ADR    r2, TABLE2 - 4    ; r2 points to TABLE2
          ADR    r3, TABLE1_END    ; last word of TABLE1
COPY1     LDR    r0, [r1,#4]!        ; load first value ....
          STR    r0, [r2,#4]!        ; and store it in TABLE2
          CMP    r1, r3
          BNE    COPY1              ;NB correct notes!
```

- ❖ Note that r1,r2 have to start 1 word before the tables, because the word loaded is always 4 more than the register.

# Data Transfer Instructions – post-indexed addressing

- ❖ Another useful form of the instruction is:

```
LDR    r0, [r1], #4    ; r0 := mem32 [r1]
                        ; r1 := r1 + 4
```

- ❖ This is called **post-indexed addressing** - the base address is used without an offset as the transfer address, after which it is *always modified*.
- ❖ Using this, we can tidy up the COPY loop:

```
copy    ADR    r1, TABLE1        ; r1 points to TABLE1
        ADR    r2, TABLE2        ; r2 points to TABLE2
        MOV    r3, AFTER_TABLE1   ; r3 points to first word after TABLE1
loop    LDR    r0, [r1], #4        ; get TABLE1 1st word ....
        STR    r0, [r2], #4        ; copy it to TABLE2
                                           ; .... r1, r2 are updated afterwards
        CMP    r2, r3             ;
        BNE    loop               ; loop if not finished
```

# Data Transfer Instructions – register-offset addressing

- ❖ Sometimes it is useful to have a base register and a **register offset**:  
**LDR    r0, [r1,r2]        ; r0 := mem<sub>32</sub> [r1+r2]**
- ❖ This is called register-offset addressing - the offset register is added to the base register to make the address.
- ❖ Using this, we can use fixed base registers and a single offset register which also counts the loop iterations:

<b>copy</b>	<b>ADR    r1, TABLE1</b>	<b>; r1 points to TABLE1</b>
	<b>ADR    r2, TABLE2</b>	<b>; r2 points to TABLE2</b>
	<b>MOV    r3,#0</b>	
<b>loop</b>	<b>LDR    r0, [r1,r3]</b>	<b>; get TABLE1 1st word ....</b>
	<b>STR    r0, [r2,r3]</b>	<b>; copy it to TABLE2</b>
	<b>ADD    r3,r3,#4</b>	<b>; move to next word</b>
	<b>CMP    r3, #200</b>	<b>; if more, go back to loop</b>
	<b>BNE loop</b>	<b>; if r3 ≠ 200</b>
	<b>.....</b>	
<b>TABLE1</b>	<b>.....</b>	<b>; &lt; source of data &gt;</b>

# ARM addressing modes


- ❖ The first operand is always the data transfer register
  - ✦ this register is data source for STR
  - ✦ this register is data destination for LDR
  - ✦ If LDRB this register is written all zeros except bottom 8 bits is byte
  - ✦ If STRB bottom 8 bits of register are source
- ❖ The memory address is whatever is inside the square brackets
- ❖ "!" means update the base register (the **first** register inside the square brackets)
- ❖ Note the register post-indexed exception, no "!" needed to write back base register

Initial values: r0 = 500, r1 = 1000, r2 = 2000			
r0 will be destination of LOAD r2 will never be changed			
Assembler		memory address	final value r1
LDR	r0, [r1]	1000	1000
LDR	r0, [r1, # 12]	1012	1000
LDR	r0, [r1, #-12]!	988	988
LDR	r0, [r1], # 4	1000	1004
LDR	r0, [r1], r2	1000	3000
LDR	r0, [r1, r2]	3000	1000
LDR	r0, [r1, r2]!	3000	3000



# ARM equivalent of direct addressing

- ❖ Sometimes it is not necessary to load a base register (eg with ADR). The code below accesses TABLE1 & TABLE2 by *using PC as the base register* with offset computed by assembler as for ADR.
- ❖ The assembly **LDR r0, TABLE1** below is translated automatically into a load using PC as base with the correct offset, for example:
  - ✦ LDR r0, [r15, #88]
- ❖ Because value of R15 is known this is effectively **direct addressing**, in limited range close to PC
  - ✦ It does not use a normal base register so can't be used for auto-increment modes etc which would change PC

 **LDR r0, [r15,#88]**

8000	LDR	r0, TABLE1	; load using PC as base
	STR	r0, TABLE2	; store using PC as base
	.....		
			; will only work if TABLE1, TABLE2
8090	TABLE1		; are within 4096 bytes of PC at
			; LDR, STR instructions

# Benefits of PC = r15: pseudo-instructions

---

- ❖ We see here two benefits of allowing PC to be a general purpose register (R15)
- ❖ Adding a constant number to PC can often be used to load a register with a useful constant memory address
  - ✦ **ADR R0, TABLE** → **ADD R0, R15, #offset**
- ❖ Using PC offset addressing allows a fixed memory location to be accessed **without first loading the address**
  - ✦ **LDR R0, TABLE** → **LDR R0, [R15, offset]**
- ❖ These **pseudo-instructions**, the transformations, and the offset calculations, are implemented by assembler
- ❖ Don't calculate these offsets yourself! They depend on precisely when PC is incremented internally in hardware.

# Data Transfer Instruction Assembly

- ❖ Size of data can be reduced to 8-bit byte for any LDR/STR instruction:

<b>LDRB</b>	<b>r0, [r1]</b>	<b>; r0 := mem<sub>8</sub> [r1]</b>
<b>STRB</b>	<b>r0,[r1]</b>	<b>; mem<sub>8</sub>[r1] := r0</b>

- ❖ In practice, most loops which access data sequentially can be simplified by using immediate-offset or post-indexed addressing, as appropriate, with **auto-indexing**.
- ❖ Summary of addressing modes (replace LDR by STR for STORE):

<b>LDRB</b>	<b>r0, [r1]</b>	<b>; register addressing</b>
<b>LDRB</b>	<b>r0, [r1, # offset]</b>	<b>; immediate offset addressing</b>
<b>LDRB</b>	<b>r0, [r1, # offset]!</b>	<b>; immediate offset auto-indexing</b>
<b>LDRB</b>	<b>r0, [r1], r2</b>	<b>; register offset post-indexed</b>
<b>LDRB</b>	<b>r0, [r1], # offset</b>	<b>; post-indexed</b>
<b>LDRB</b>	<b>r0, [r1, r2]</b>	<b>; register-offset addressing</b>
<b>LDRB</b>	<b>r0, address_label</b>	<b>; PC relative addressing</b>
<b>ADR</b>	<b>r0, address_label</b>	<b>; load PC relative address</b>

pseudo  
instructions

# Lecture 7 - ARM Tips & Tricks

---

"When I hear somebody sigh, 'Life is hard', I am always tempted to ask, 'Compared to what?'" Sydney J Harris

- ❖ The ARM has a unique and clever way of implementing *conditional* branches.
  - ✦ Instead of having special instructions, **all instructions** are given an execution condition which determines whether they are executed, or ignored. If an instruction is ignored it has no effect on registers.
  - ✦ Condition is top 4 bits of instruction word
  - ✦ The “**always true**” condition is used with most instructions to make execution **unconditional**
- ❖ A single branch instruction thus provides conditional and unconditional branches.

# Conditional Branches

- ❖ **Conditional** branch instructions can be used to control loops:

loop	MOV	r0, #10	; initialize loop counter r0
	.....		; start of body of loop
	.....		
	SUB	r0, r0, #1	; decrement loop counter
	CMP	r0, #0	; is it zero yet?
	BNE	loop	; branch if r0 ≠ 0

- ❖ Here the CMP instruction is a SUBTRACTION, which gives no results EXCEPT possibly changing condition codes in CPSR.
- ❖ After CMP, if  $r0 = 0$ , then Z bit is set (= '1'), else Z bit is reset (= '0')
  - ✦ Z controls the following **BNE** conditional branch instruction

# Remember "S" suffix on data processing instructions

- ❖ Explicit comparisons are not needed after a SUBS or ADDS:

	<b>MOV</b>	<b>r0, #10</b>	<b>; initialize loop counter r0</b>
<b>loop</b>	<b>.....</b>		<b>; start of body of loop</b>
	<b>.....</b>		
	<b>SUBS</b>	<b>r0, r0, #1</b>	<b>; decrement loop counter AND set flags</b>
	<b>BNE</b>	<b>loop</b>	<b>; branch if r0 ≠ 0</b>

- ❖ SUBS instruction is the same as SUB except that the former updates the NZCV flags in the CPSR.
- ❖ Compare this optimisation, not needing CMP, with the previous slide!
- ❖ All data processing instructions can have S:

**EORS R0,R1,R2**

**ANDS R0,R3,#0**

**ADCS R0, R1, R2**

- ❖ **CMP** is identical to **SUBS** but with no destination register.

# Tricks with "S"

❖ Branch with condition after data processing instruction

```
MOV R0, #10
LOOP ADD R1, R1, R2
     SUBS R0, R0, #1 ; set codes
     BNE LOOP ; branch on codes
```

❖ Branch with condition after comparison

```
MOV R0, #10
LOOP ADD R1, R1, R2
     SUB R0, R0, #1
     CMP R0, #0 ; set codes
     BNE LOOP ; branch on codes
```

; Note S instruction need not be immediately before BNE instruction

```
MOV R0, #10
LOOP SUBS R0, R0, #1 ; set codes
     ADD R1, R1, R2 ; all other instructions preserve codes
     BNE LOOP ; branch on codes
```

# An If-Then-Else example

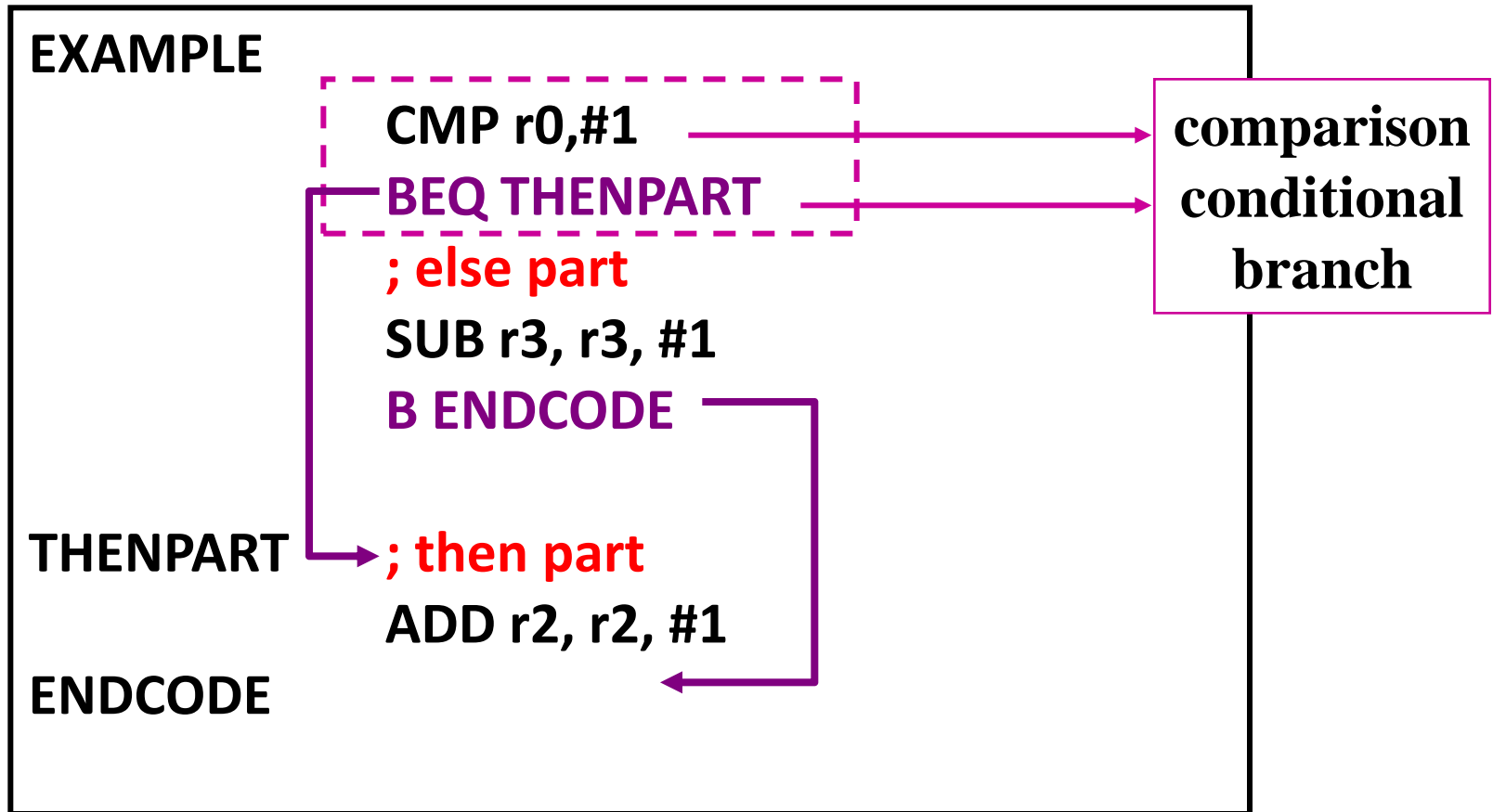
---

- ❖ Consider the pseudo-code:
  - ✦ **If (a = 1) then c := c+1 else d := d-1**
- ❖ Needs to be implemented using conditional branches, or, as we will see, conditional execution.
- ❖ First step is to assign registers to variables. We assume: a=r0, c=r2, d=r3, and then the problem becomes:
  - ✦ **if (r0 = 1) then r2 := r2+1 else r3 := r3-1**
- ❖ To translate this pseudocode we need to use **branches** and **conditional execution**



# ITE – with conditional branches

✦ if ( $r0 = 1$ ) then  $r2 := r2+1$  else  $r3 := r3-1$

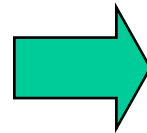


# Conditional Execution in ITE

- ❖ We have seen that IF-THEN-ELSE constructions in pseudo-code turn into multiple branches in assembly.
- ❖ If the THEN and ELSE statements are short, branches can be avoided by using conditional execution.
- ❖ Note: NE condition is opposite of EQ condition

```
CMP r0, #1
BEQ THENPART
SUB r3, r3, #1      ; else part
B ENDCODE           ; go to end

THENPART
ADD r2, r2, #1      ; then part
ENDCODE             ; finished
```



```
CMP r0, #1
SUBNE r3, r3, #1 ; else part
ADDEQ r2, r2, #1 ; then part
; finished
```

# Conditional Execution - more

- ❖ Here is another very clever use of this unique feature in ARM instruction set. ALL instructions can be qualified by the condition codes, including CMP!

```
; if ( (a=b) and (c=d)) then e := e + 1
    CMP    r0, r1          ; r0 has a, r1 has b
    CMPEQ  r2, r3          ; r2 has c, r3 has d
    ADDEQ  r4, r4, #1      ; e := e+1
```

- ❖ Note how if the first comparison finds unequal operands, the second and third instructions are both skipped.
- ❖ Also the logical 'and' in the if clause is implemented by making the second comparison conditional on the first.
- ❖ Conditional execution is normally only efficient if the conditional sequence is three instructions or fewer. If the conditional sequence is longer, use branches.

# Moving bits – shifts etc

---

“The best teachers have shown me that things have to be done bit by bit. Nothing that means anything happens quickly – we only think it does”, Joseph Bruchac

- ❖ Individual bits can have separate meanings in assembly programs
  - ✦ Hardware registers where every bit is a separate flag
  - ✦ Hardware registers where bit fields have specific meaning
- ❖ Two types of operation help manipulating bits
  - ✦ Shifts & rotates
  - ✦ 32 bit **bitwise** logical data processing instructions:
  - ✦ AND, ORR, EOR, BCI (see lecture 5).

# Register Shifts

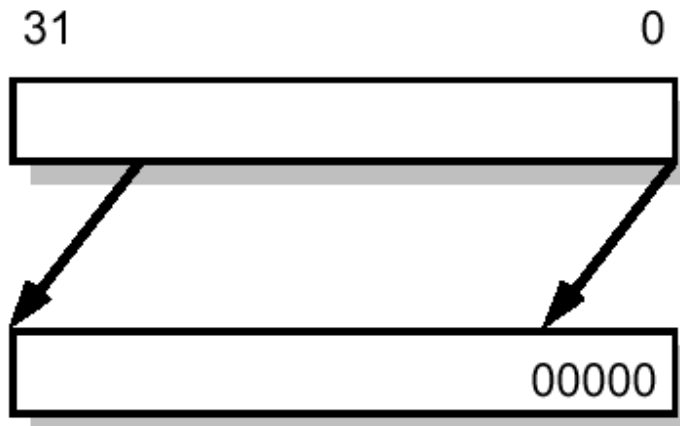
---

ADD r0, r1, r2, lsl #3  
MOV r0, r1, lsr #11 } op2 shifted

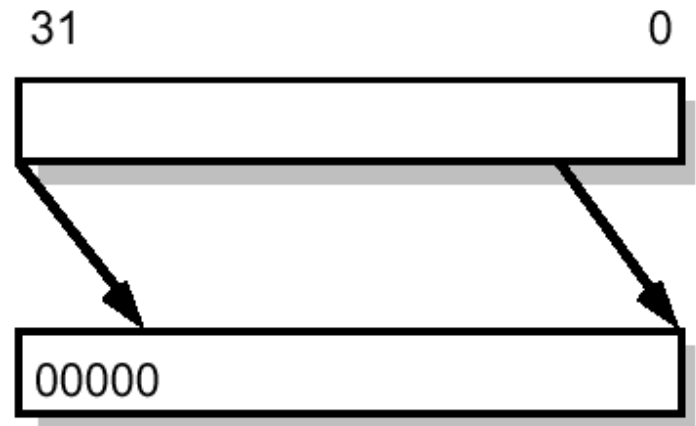
- ❖ The key to manipulating **bit fields** – contiguous groups of bits – is the use of data shifts.
- ❖ ARM has a large collection of shifts available for the 2<sup>nd</sup> register operand of a data processing instruction.
  - ✦ shifts can be combined with arithmetic or bitwise logical operations in one instruction.
- ❖ Rd := Rn op (Rm shift by n) ; shift = lsl, asr, asl, ror, rrx
  - ✦  $0 \leq n \leq 31$
  - ✦ RRX is special case only possible by 1 bit (n=1).
- ❖ **NOTE Rm is not changed by shift – shifted value is used as operand**

# ARM shift operations - LSL and LSR

- ❖ Here are all the six possible ARM shift operations you can use:



LSL #5



LSR #5

- ❖ LSL: logical shift left by 0 to 31 places; fill the vacated bits at the least significant end of the word with zeros.
  - ✦  $x \text{ LSL } n = x * 2^n$  if no overflow
- ❖ LSR: logical shift right by 0 to 31 places; fill the vacated bits at the most significant end of the word with zeros.
  - ✦  $x \text{ LSR } n = x / 2^n$  if  $x$  is positive (integer division)

# ARM Trick: Multiplying by a (small) constant using LSL

- ❖ Multiplying by  $2^N$  is easy using a **left shift**. Other constants can be derived from this by using ADD or RSB as in the table below.
  - ★ 2,3,4,5,7,8,9, etc are all possible in this way
- ❖ Where possible it is better than using a MUL instruction because it is faster, and uses immediate values directly from instruction words (don't need to set register to n)

$r0 := 2^N r1$	MOV r0, r1 lsl #n	
$r0 := (2^N + 1)r1$	ADD r0, r1, r1 lsl #n	
$r0 := (2^N - 1)r1$	RSB r0, r1, r1 lsl #n	Note RSB not SUB

ADD	r0, r0, r0, LSL #2	; r0' := 5 x r0
RSB	r0, r0, r0, LSL #3	; r0'' := 7 x r0'

**What does this multiply by?**

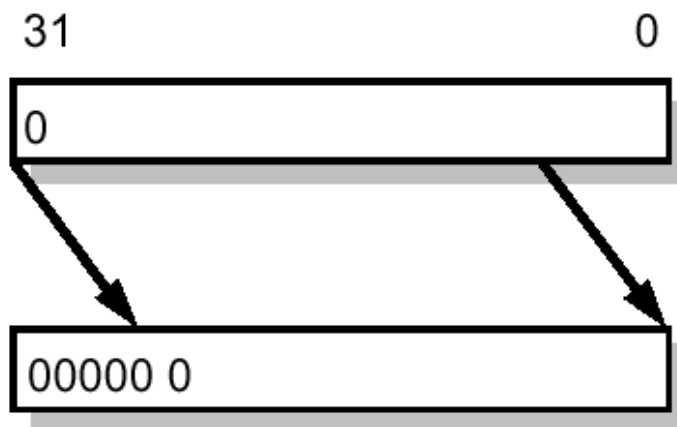
# ARM shift ops - ASL and ASR

x	x asr 1
3	1
2	1
1	0
0	0
-1	-1
-2	-1
-3	-2
-4	-2

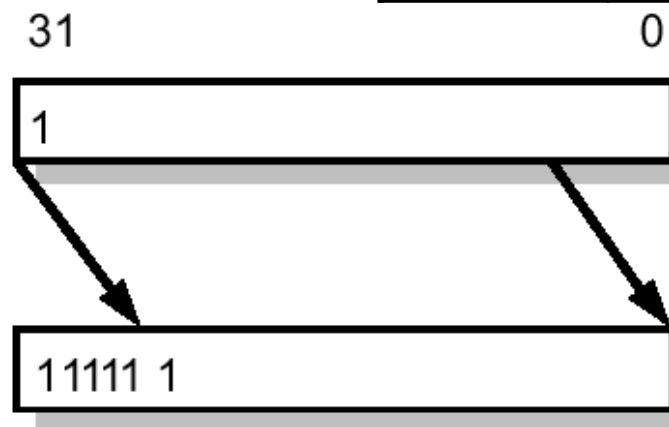
- ❖ ASL: **arithmetic shift left**; this is the same as LSL
- ❖ ASR: **arithmetic shift right** by 0 to 31 places; fill the vacated bits at the most significant end of the word with zeros if the source operand was positive, and with ones if it is negative. That is, sign extend while shifting right.

✦  $x \text{ ASR } n = x / 2^n \ (x > 0)$

✦  $-x \text{ ASR } n = -(x+1) / 2 = -x/2^n \text{ (rounding negatively)}$



ASR #5, positive operand

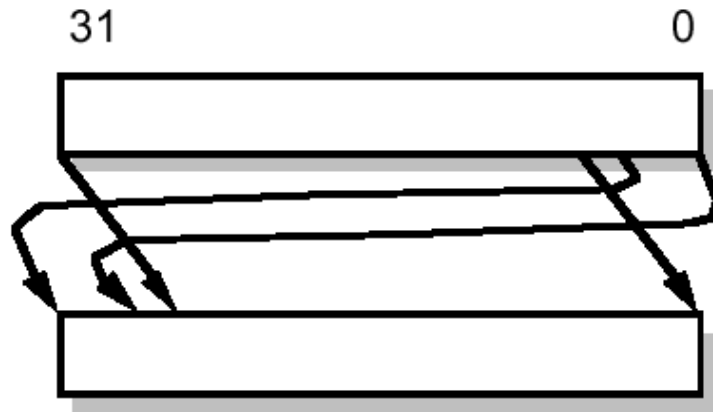


ASR #5, negative operand

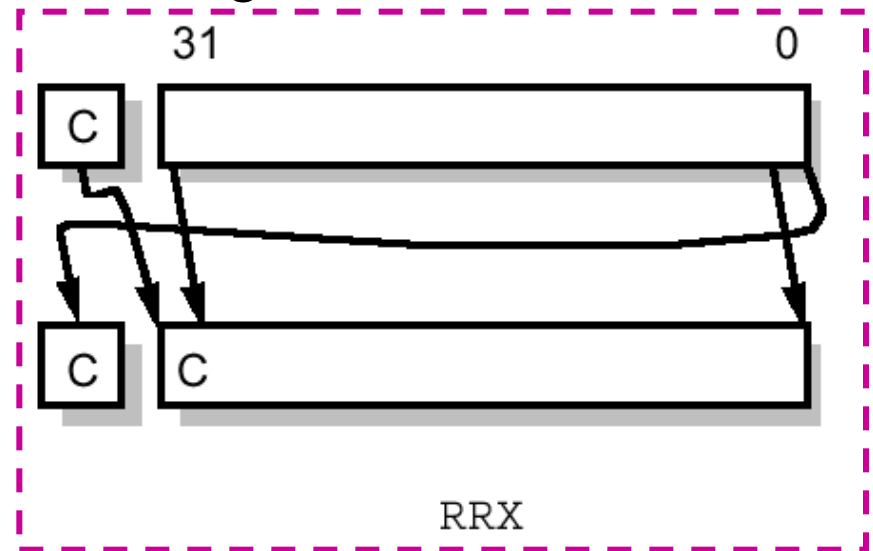


# ARM rotate operations - ROR and RRX

- ❖ ROR: rotate right by 0 to 31 places; the bits which fall off the least significant end are used to fill the vacated bits at the most significant end of the word. ( $\text{ROL } n = \text{ROR } 32-n$ )
- ❖ RRX: rotate right extended by 1 place; the vacated bit (bit 31) is filled with the old value of the C flag and the operand is shifted one place to the right. This is effectively a 33 bit rotate using the register and the C flag.



ROR #5



RRX

# Shift by number of bits equal to value of a register

**ADD r0, r1, r2, lsl r3** ; shift r2 by value of register r3 and add!  
**MOV r0, r1, asr r10** ; shift r1 by value of register r10

❖ The number (n previously) of **bits to shift** is now the **value** of another register.

★ Note that one extra operand is needed!

★ These "register-valued" shifts take two cycles to execute

❖ **MOV r0, r1, lsl r3**

★ If  $r3 = 4$  &  $r1 = 11$  this will set

★  $r0 := 11 * 2^4$

❖ This allows variable shifts, eg, to select bit  $n$  from a 32 bit register

# Data Transfer Instructions – scaled register-offset addressing

```
LDR    r0, [r1,r2, lsl #n]      ; r0 := mem32 [r1+(r2 * 2n)]
```

- ❖ The second (index) register can have an optional shift – useful in this case so that it can count words (bytes\*4) directly
- ❖ In principle any of the shift modes: **lsl**, **asl**, **asr**, **rrx** can be used
  - ★ **lsl #n** used here multiplies by a scale factor of 2<sup>N</sup>

copy	ADR	r1, TABLE1	; r1 points to TABLE1
	ADR	r2, TABLE2	; r2 points to TABLE2
	MOV	r3,#0	
loop	LDR	r0, [r1, r3, lsl #2]	; get TABLE1 1st word ....
	STR	r0, [r2, r3, lsl #2]	; copy it to TABLE2
	ADD	r3,r3,#1	; move to next word
	CMP	r3, #50	; if more, go back to loop
	BNE	loop	; if r3 ≠ 50
	.....		
TABLE1	.....		; < source of data >

# Reference - ARM condition code field

Opcode [31:28]	Mnemonic extension	Interpretation	Status flag state for execution
0000	EQ	Equal / equals zero	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set / unsigned higher or same	C set
0011	CC/LO	Carry clear / unsigned lower	C clear
0100	MI	Minus / negative	N set
0101	PL	Plus / positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N equals V
1011	LT	Signed less than	N is not equal to V
1100	GT	Signed greater than	Z clear and N equals V
1101	LE	Signed less than or equal	Z set or N is not equal to V
1110	AL	Always	any
1111	NV	Never (do not use!)	none

**BMI LABEL ; Branch to LABEL on MI condition**

# Reference - Inequality Conditions Summarised

---

- ❖ ARM has the full set of **signed** and **unsigned** inequality conditions. They can be confusing.
- ❖ After a CMP or SUBS, if x,y are the two operands, the 8 possible inequalities are shown in the table below
  - ✦ It is important to choose the correct condition if the test is to work for all inputs, even though for positive numbers signed and unsigned comparisons are identical.

Test	Signed		Unsigned	
$x > y$	GreaTer	<b>GT</b>	HIgher	<b>HI</b>
$x \geq y$	Greater or Equal	<b>GE</b>	Higher or Same	<b>HS (= CS)</b>
$x \leq y$	Less or Equal	<b>LE</b>	Lower or Same	<b>LS</b>
$x < y$	Less Than	<b>LT</b>	LOWer	<b>LO (= CC)</b>

# Reference - ARM mnemonics with condition codes

---

- ❖ The two letter **condition code** is appended to the **3 letter** instruction op-code to make instruction execution conditional: MOVEQ, ADDPL, BCC, LDRMI, etc.
  - ✦ Always condition “AL” is omitted for (normal) unconditional execution
- ❖ Op-code suffixes (S for data processing instructions, B for LDR/STR) go **after** the condition code:
  - ✦ ADDPLS, STRNEB
  - ✦ SBCCSS

# Lecture 8 – Subroutines & Stacks

---

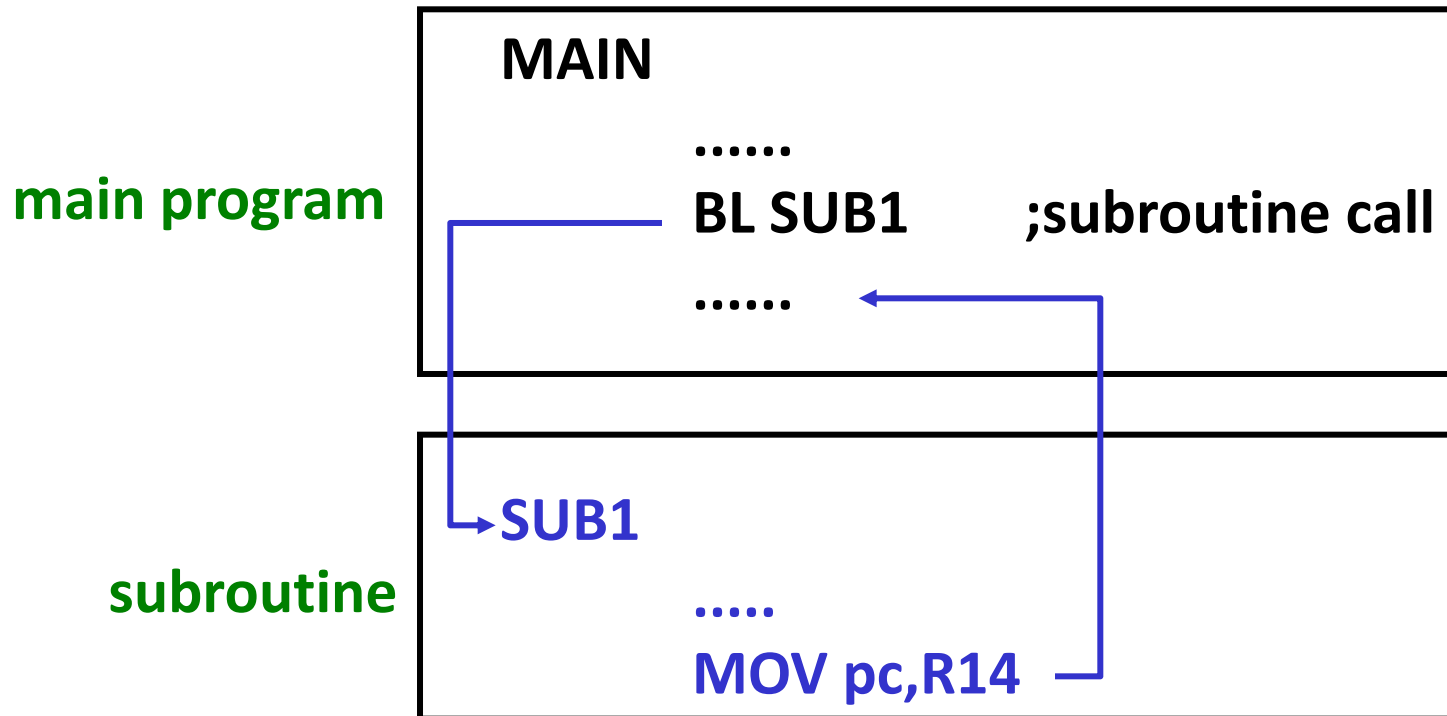
*"Television is like the American toaster, you push the button and the same thing pops up everytime" Alfred Hitchcock*

- ❖ The **subroutine** is a key element in assembly language programs, allowing code reuse
  - ✦ It is also the way that High Level Language procedures and functions are implemented
- ❖ Storage of data on a **stack** is an essential element of all modern computer programs and typically is done on subroutine entry & exit
- ❖ ARM has instructions to support subroutines and stacks
- ❖ This lecture will consider
  - ✦ Use of return addresses by subroutines
    - ✖ Branch & link instruction
  - ✦ Storing data on stacks in the ARM ISA
    - ✖ Load & Store Multiple Registers instructions

# Subroutines

---

- ❖ Subroutines allow you to modularize your code so that they are more reusable.
- ❖ The general structure of a subroutine in a program is:





# Branch & Link instruction

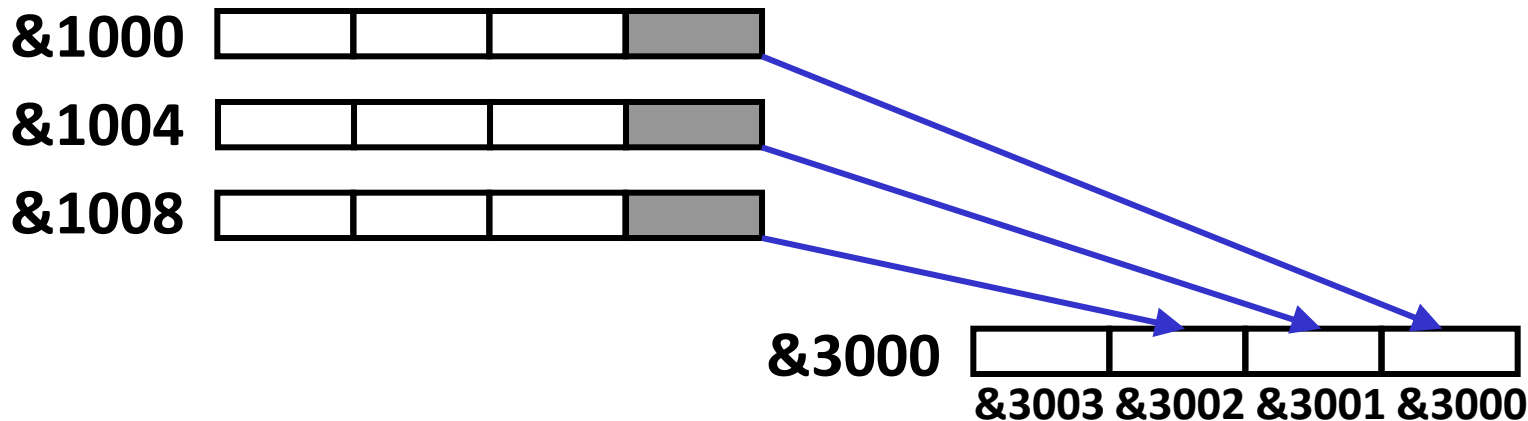
---

- ❖ **BL** `subroutine_name` (Branch-and-Link) is the instruction to jump to subroutine. It performs the following operations:
  - ✦ 1) It saves the **PC** value (which points to the next instruction) in **r14**. This is the return address.
  - ✦ 2) It loads **PC** with the address of the subroutine. This performs a branch.
- ❖ BL always uses r14 to store the return address. r14 is called the **link register** (can be referred to as **lr** or **r14** in assembly code).
- ❖ Return from subroutine is simple: - just put r14 back into PC (r15).

# Example

---

- ❖ Essential documentation for subroutines must describe
  - ✦ Inputs
  - ✦ Outputs (if any)
  - ✦ What subroutine does (other than compute outputs)
  - ✦ Which registers it changes
- ❖ EXAMPLE: Subroutine to move n bytes (spaced one per word) into n contiguous bytes at a different position in memory



## **PACK\_BYTES**

**; Input: src=r0, dest=r1, n=r2**

**; loads LS bytes in words [R0],[R0+4], ..., [R0+4(n-1)]**

**; into contiguous bytes [R1],[R1+1],.....[R1+n-1]**

**; Changes r2,r3**

**LOOP SUBS R2, R2,#1**

**; n := n-1**

**LDRB R3, [R0,R2, lsl #2]**

**; load first byte [R0+4(n-1)]**

**STRB R3, [R1,R2]**

**; store it [R1+n-1]**

**BNE LOOP**

**MOV pc, r14**

**; return to caller**

## **MAIN**

**ADR R0, TAB1 ; set up subroutine inputs**

**ADR R1, TAB2**

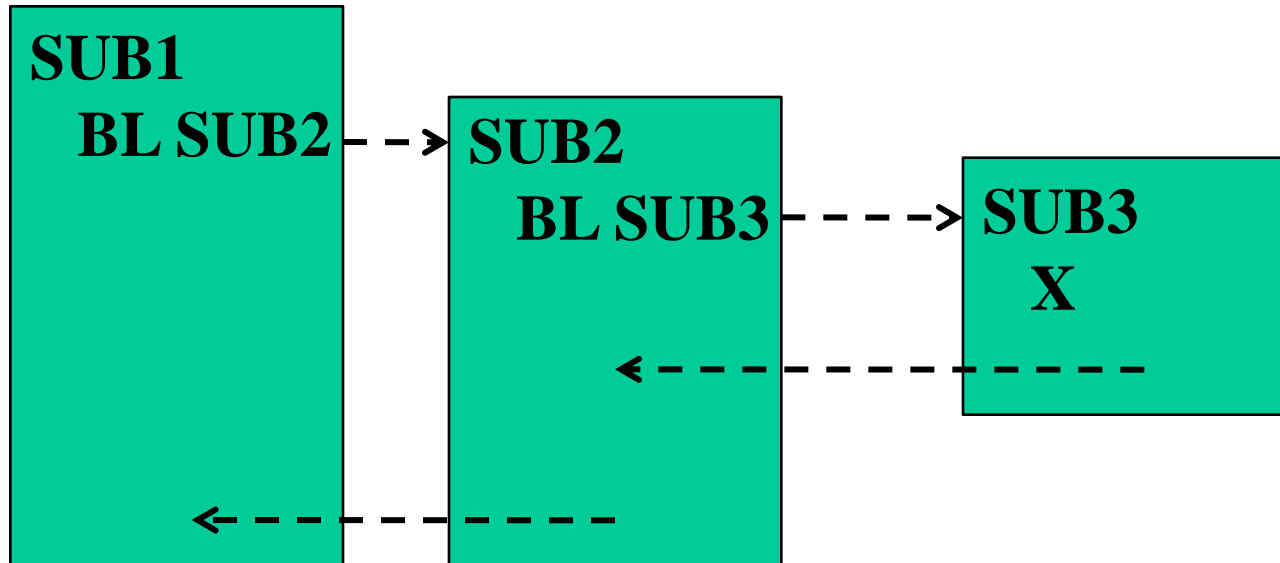
**MOV R2, #100**

**BL PACK\_BYTES ; call the subroutine**

**; instruction here executed after subroutine return**

# Nested Subroutines

---

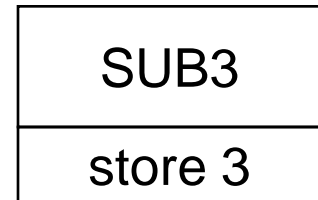
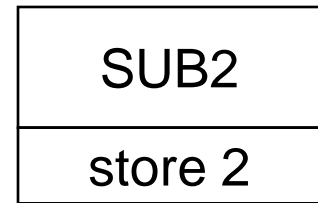
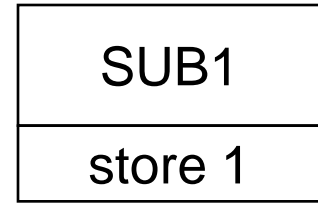


When executing at "X" the **nested** subroutines  
SUB1, SUB2, SUB3 are all active

# Nested Subroutines

---

- ❖ Since the return address is held in register r14, you should not call a further subroutine without first saving r14.
- ❖ How do you achieve this goal?
  - ✦ Could use separate storage for each subroutine
  - ✦ Problem: storage needed scales with number of subroutines.
  - ✦ Typically may have 1000s of subroutines, means 1000s of separate storage locations
- ❖ The number of subroutines active at any time (**nested**) is much smaller than the total number, typically less than 10.
- ❖ This motivates use of a **stack** – an area of memory which is shared for storage by subroutines.
- ❖ Can store **all registers changed by subroutine** on stack, not just R14

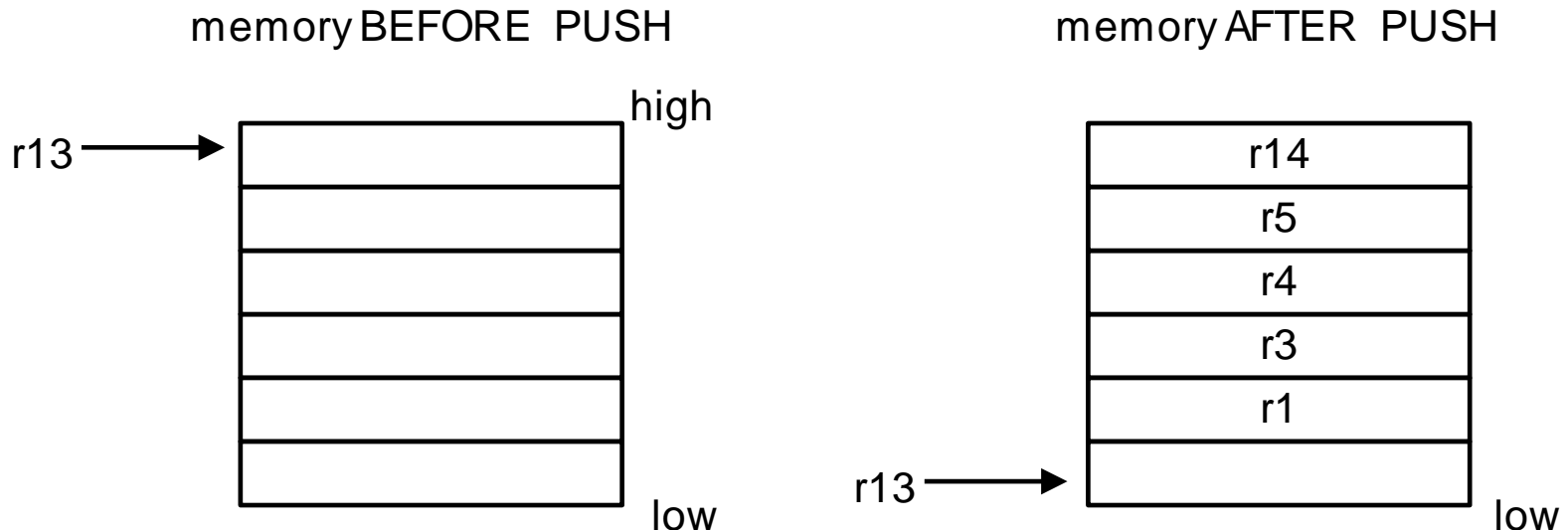


# The idea of a STACK

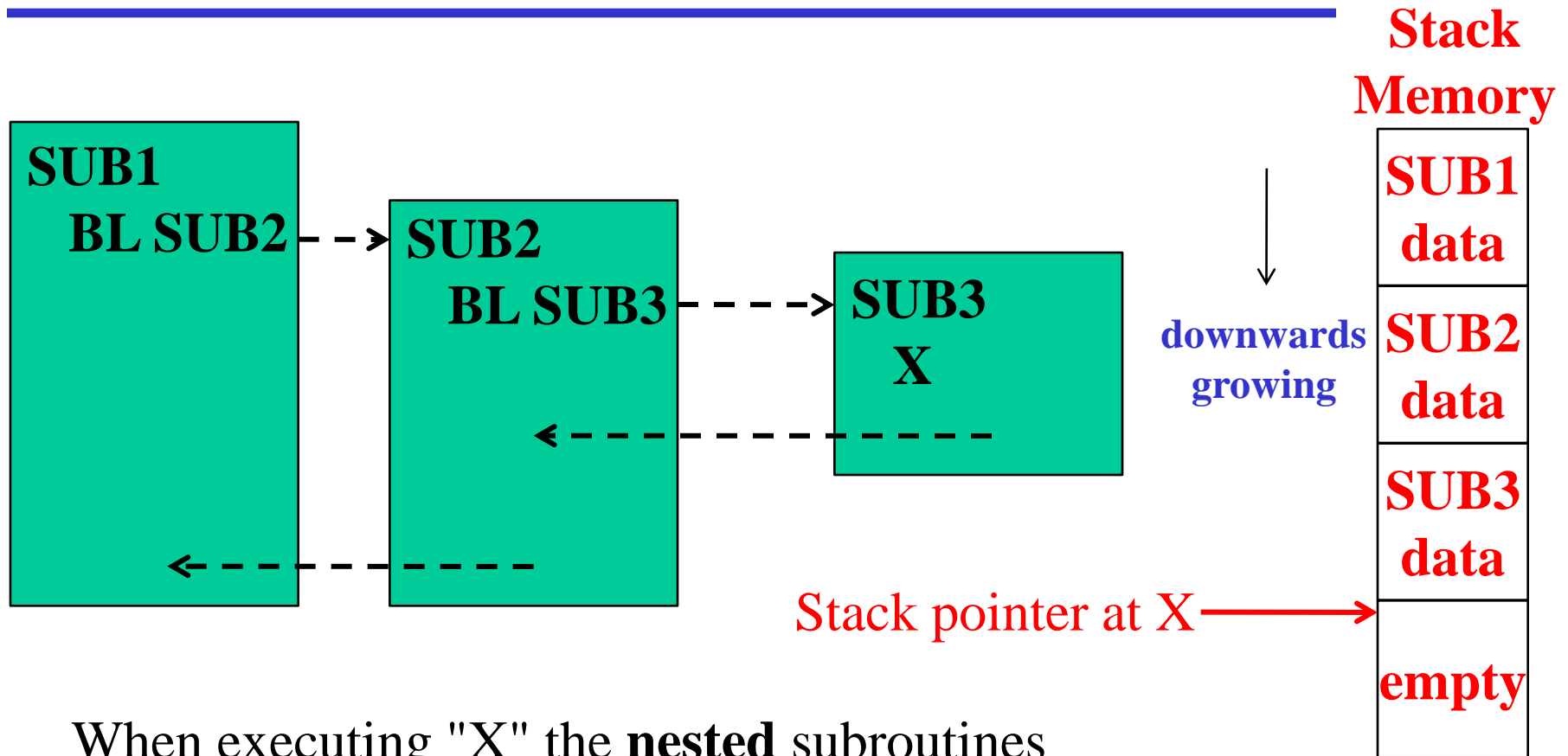
- ❖ A stack is a portion of main memory used to store data temporarily, so that the memory can be shared between different items at different times.
- ❖ A PUSH operation stores a number of registers onto the stack memory.

**PUSH {r1, r3-r5, r14}**

**r13 is called the  
stack pointer SP**



# Nested Subroutines using stack



When executing "X" the **nested** subroutines  
SUB1, SUB2, SUB3 are all active

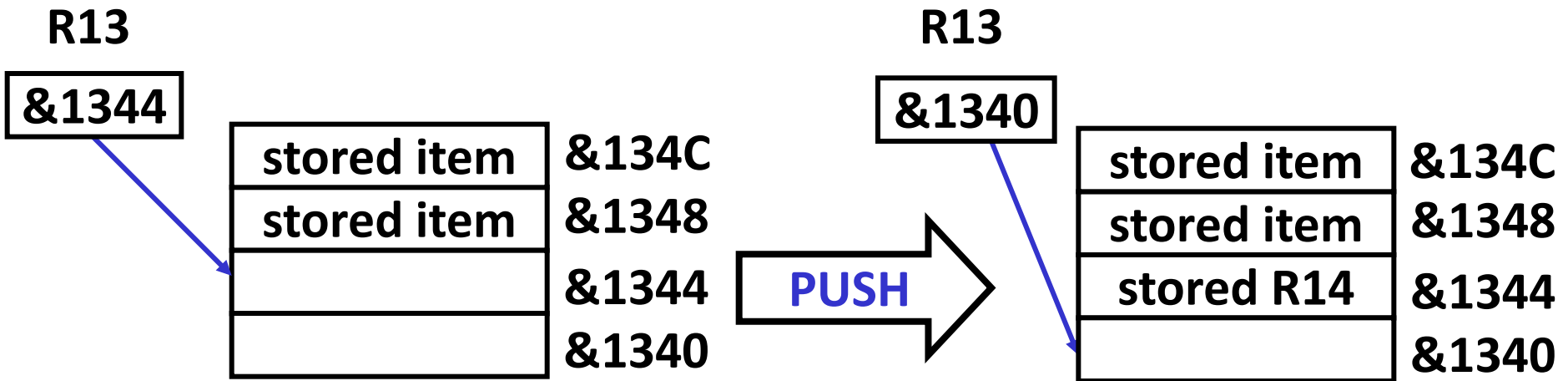
# PUSH R14 onto stack: method 1

---

**mem32[R13] := R14**  
**R13 := R13-4**

**STR R14, [R13], #-4**

**Would need one LDR instruction for each item...**





# PUSHing onto a Stack: multiple registers

---

- ❖ Note the following properties of this ARM PUSH operation:
  - ✦ **r13** is used as the address pointer. We call this **STACK POINTER (SP)**. We could have used any other registers (except r15) as SP, but it is good practice to use **r13** unless there is a good reason not to do so.
  - ✦ This stack grows **down** through **decreasing** memory address, and
  - ✦ The base registers points to the first **empty** location of the stack. To store values in memory, the SP is **decremented after** it is used.
- ❖ ARM has a single instruction which transfers multiple registers to a stack and implements PUSH this way:

**STMED r13!, {r1, r3-r5, r14}**

**; Push r1, r3-r5, r14 onto stack  
; Stack grows down in mem  
; r13 points to next empty loc.**

# STMED vs STR

---

- ❖ These two instructions look different but do same thing with one register
- ❖ STMED can be used with any number of registers
- ❖ STMED is conventionally used for stacks even when only a single transfer is needed.

**STMED R13!, {R14}**

**stack pointer first, then list of one or more data registers, offset is calculated and added after operation**

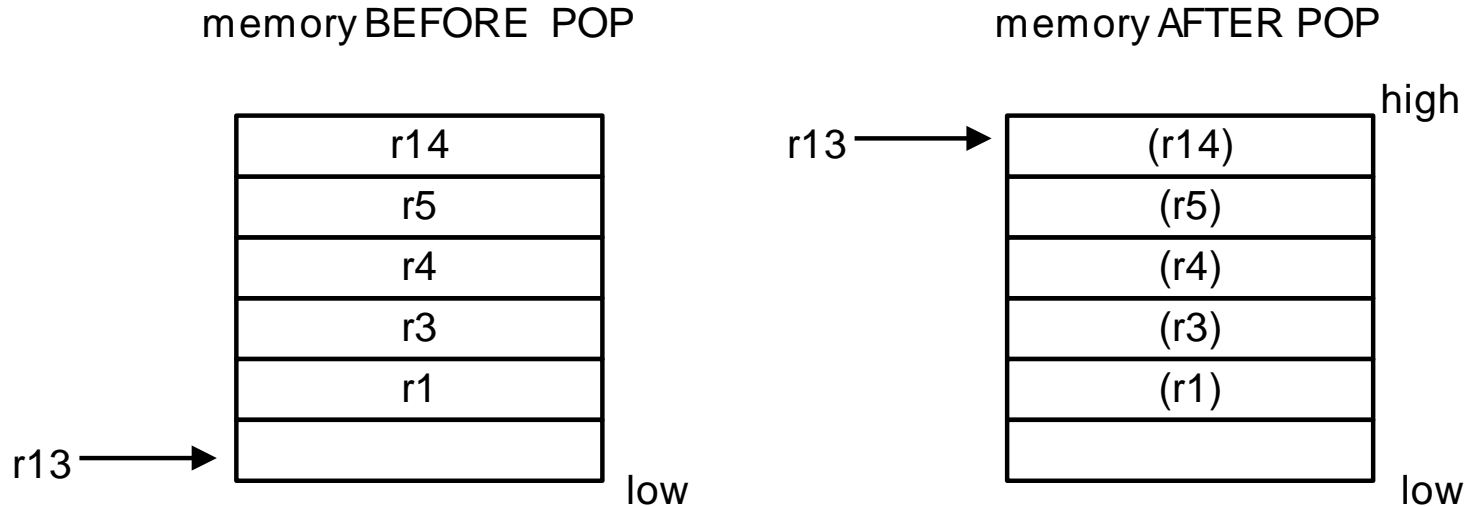
**STR R14, [R13], #-4**

**data register first, then stack pointer, offset is explicitly written and added to SP after operation**

# POP operation

- ❖ The complementary operation of PUSH is the **POP** operation.

**POP {r1, r3-r5, r14}**



- ❖ This is equivalent to the ARM instruction:

**LDMED r13!, {r1, r3-r5, r14}**

**; Pop r1, r3-r5, r14 from stack**

# Multiple Stack Operations

---

❖ A stack operates as a Last In First Out memory:

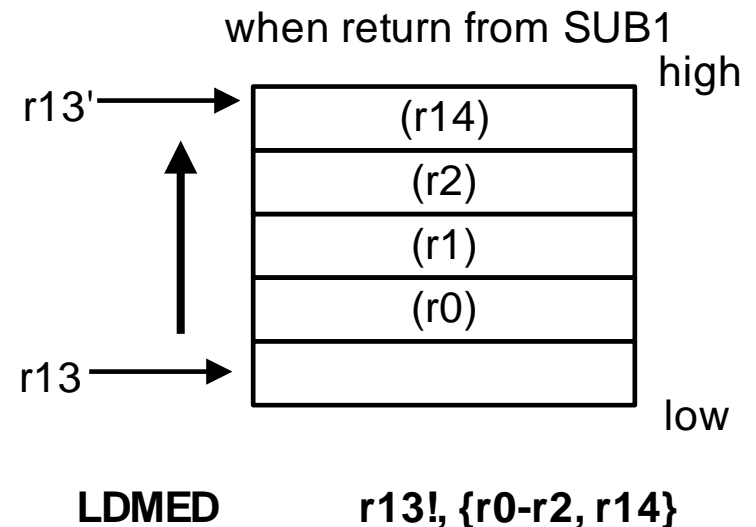
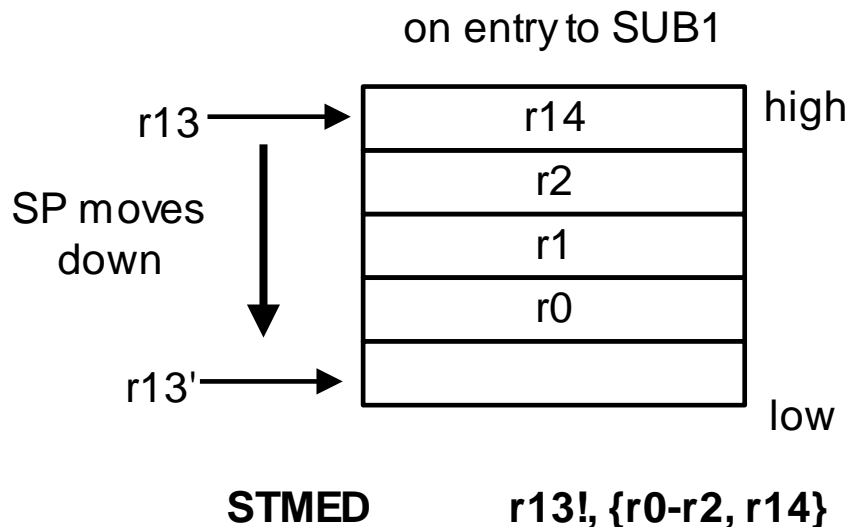
- ★ PUSH A                      A stored
- ★ PUSH B                      B, A stored
- ★ PUSH C                      C,B,A stored
- ★ POP (returns C)      B,A stored
- ★ POP (returns B)      A stored
- ★ POP (returns A)      empty

**Stack implements a  
Last-In-First-Out  
(LIFO) memory**

❖ Nested subroutines will each PUSH and then POP their registers at the same level (all PUSHes & POPs from subroutine calls will balance) so this will work.

# Preserve things inside subroutine with STACK

```
BL      SUB1
.....
SUB1    STMED r13!, {r0-r2, r14}      ; push work & link registers
....
BL      SUB2                          ; jump to a nested subroutine
...
LDMED r13!, {r0-r2, r14}             ; pop work & link registers
MOV     pc, r14                      ; return to calling program
```



**; Input: r0**

**; Output: r1=1 if odd parity (xor of all 32 bits), otherwise 0**

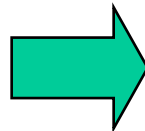
**; preserves value of r2 on stack**

	<b>STMED</b>	<b>r13!, {r2}</b>	<b>; save registers, why not r1?</b>
	<b>MOV</b>	<b>r2, #31</b>	
	<b>MOV</b>	<b>r1, #0</b>	
<b>LOOP</b>	<b>EOR</b>	<b>r1, r0, r1, ror #1</b>	
	<b>SUBS</b>	<b>r2, r2, #1</b>	
	<b>BPL</b>	<b>LOOP</b>	<b>; loop 32 times</b>
	<b>AND</b>	<b>r1, r1, #1</b>	
	<b>LDMED</b>	<b>r13!, {r2}</b>	<b>; restore registers</b>
	<b>MOV</b>	<b>pc, r14</b>	<b>; return to caller</b>

# Optimising subroutine entry/exit

- ❖ The usual case is for a subroutine which calls other subroutines, and so which saves and restores registers **including R14**, the return address.
- ❖ In this case the subroutine exit can be optimised by restoring r14 directly to the PC, r15.
  - ★ Note that it is important NOT to include both r14 & r15 in the LDMED register list - which would be one too many POPs!

```
STMED  r13!, {r0,r1,r2,r14}  
.....  
LDMED  r13!,{r0,r1,r2, r14}  
MOV     pc, r14 ; return to caller
```



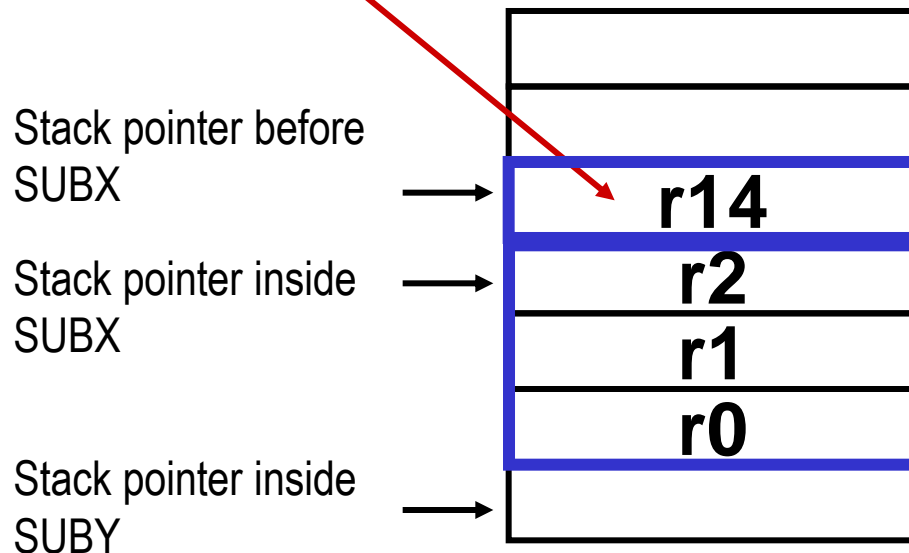
```
STMED  r13!, {r0,r1,r2,r14}  
.....  
LDMED  r13!,{r0,r1,r2,pc} ; return to caller
```

# Effect on stack of subroutine nesting

- ❖ SUBX (1) calls SUBY(2)
- ❖ The arrangement of storage on the stack when inside SUBY is as follows

**SUBX caller  
return address**

**Stack (downwards growing)**



```
SUBX  STMED r13!, {R14}  
      BL SUBY
```

```
.....  
      LD MED r13!, {pc}
```

```
SUBY  STMED r13!, {r0,r1,r2}
```

```
.....  
      LD MED r13!, {r0,r1,r2}  
      MOV pc, r14
```

**Base of stack is highest location**

Rest of stack

Stack frame (1) SUBX

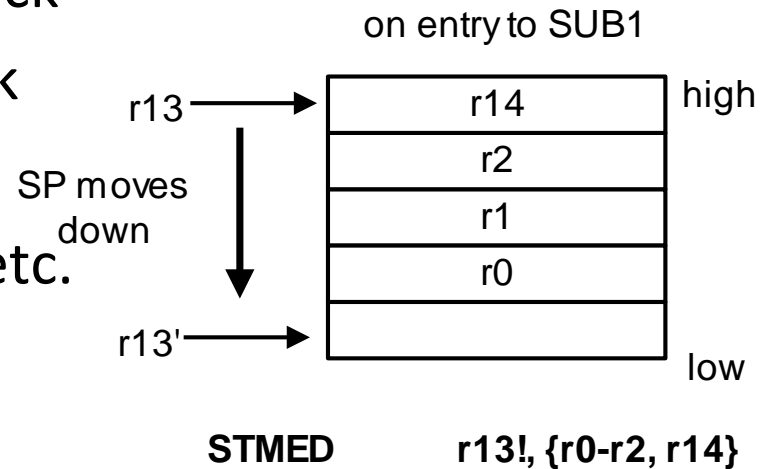
Stack frame (2) SUBY

**Top of stack is SP+4 (lowest location)**



# ARM PUSH instructions

- ❖ STMED implements **Descending stack**, with SP pointing to **Empty** location
- ❖ Stacks can be Ascending or Descending
- ❖ SP can point to Full location (last item PUSHED) or Empty location (first space available to PUSH next item)
- ❖ **STMED** - Empty location, Descending stack
- ❖ **STMEA** - Empty location, Ascending stack
- ❖ **STMFD** - Full location, Descending stack
- ❖ **STMFA** - Full location, Ascending stack
- ❖ **LDMED (pop)** matches **STMED (push)** etc.



# Other uses of LDM/STM

---

- ❖ LDM,STM can work with any register being SP, not just R13
- ❖ Can move block of memory by setting up SP1, SP2, POP from SP1, PUSH to SP2
- ❖ Faster than loop with LDR/STR
- ❖ The 4 types of stack POP & PUSH have different mnemonics (for convenience) when used for general data movement like this.
- ❖ It does not matter which mnemonic you use:  
LDMED & LDMIB are the **same instruction**

# Example of using Load/Store Multiple

- ❖ Here is an example to move 8 words from a source memory location to a destination memory location:-

ADR	r0, src_addr	; initialize src addr
ADR	r1, dest_addr	; initialize dest addr
LDMIA	r0!, {r2-r9}	; fetch 8 words from mem
		; ... r0 := r0+32
STMIA	r1!, {r2-r9}	; copy 8 words to mem, r1 := r1+32

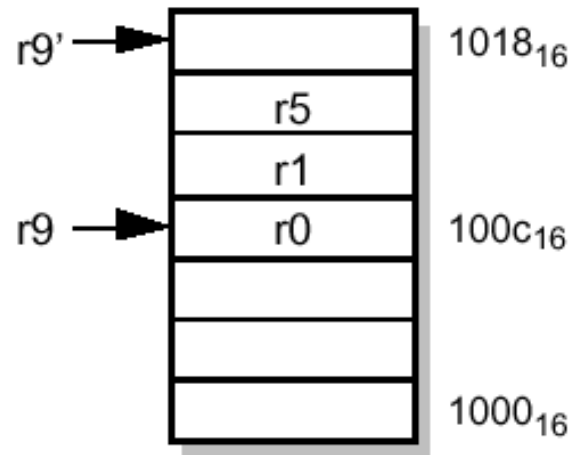
- ❖ When using LDMIA and STMIA instructions, you:-
  - ✦ **INCREMENT** the address in memory to load/store your data
  - ✦ the increment of the address occurs **AFTER** the address is used.
- ❖ In fact, one could use 4 different form of load/store:
  - ✦ Increment - **After** LDMIA and STMIA
  - ✦ Increment - **Before** LDMIB and STMIB (see next slide)
  - ✦ Decrement - **After** LDMDA and STMDA
  - ✦ Decrement - **Before** LDMDB and STMDB

# Alternative names for LDM instructions!

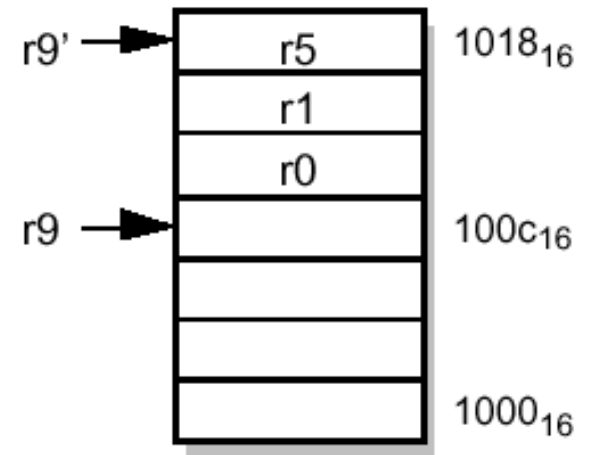
---

Name	Stack	Other
pre-increment load	LDMED	LDMIB
post-increment load	LDMFD	LDMIA
pre-decrement load	LDMEA	LDMDB
post-decrement load	LDMFA	LDMDA
pre-increment store	STMFA	STMIB
post-increment store	STMEA	STMIA
pre-decrement store	STMFD	STMDB
post-decrement store	STMED	STMDA

# The four variations of the STM instruction

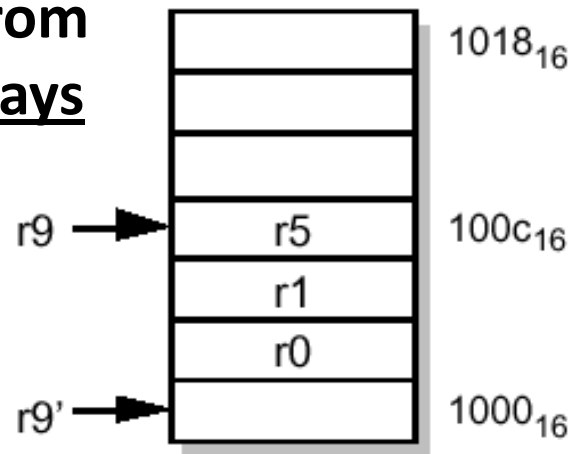


`STMIA r9!, {r0,r1,r5}`

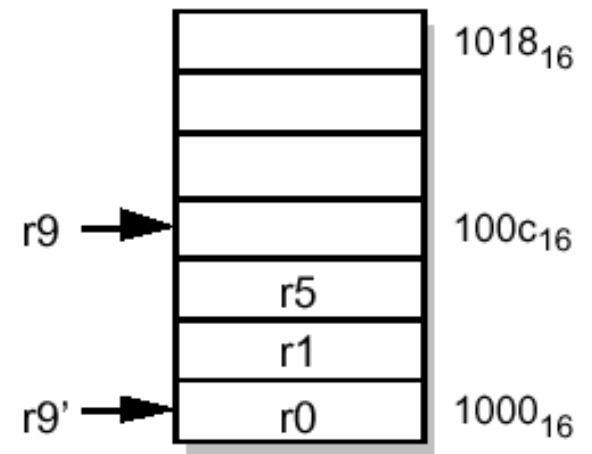


`STMIB r9!, {r0,r1,r5}`

Higher register numbers  
stored or loaded to/from  
higher addresses, always



`STMDA r9!, {r0,r1,r5}`



`STMDB r9!, {r0,r1,r5}`

# Optional update of base address register with Load/Store Multiple Instructions

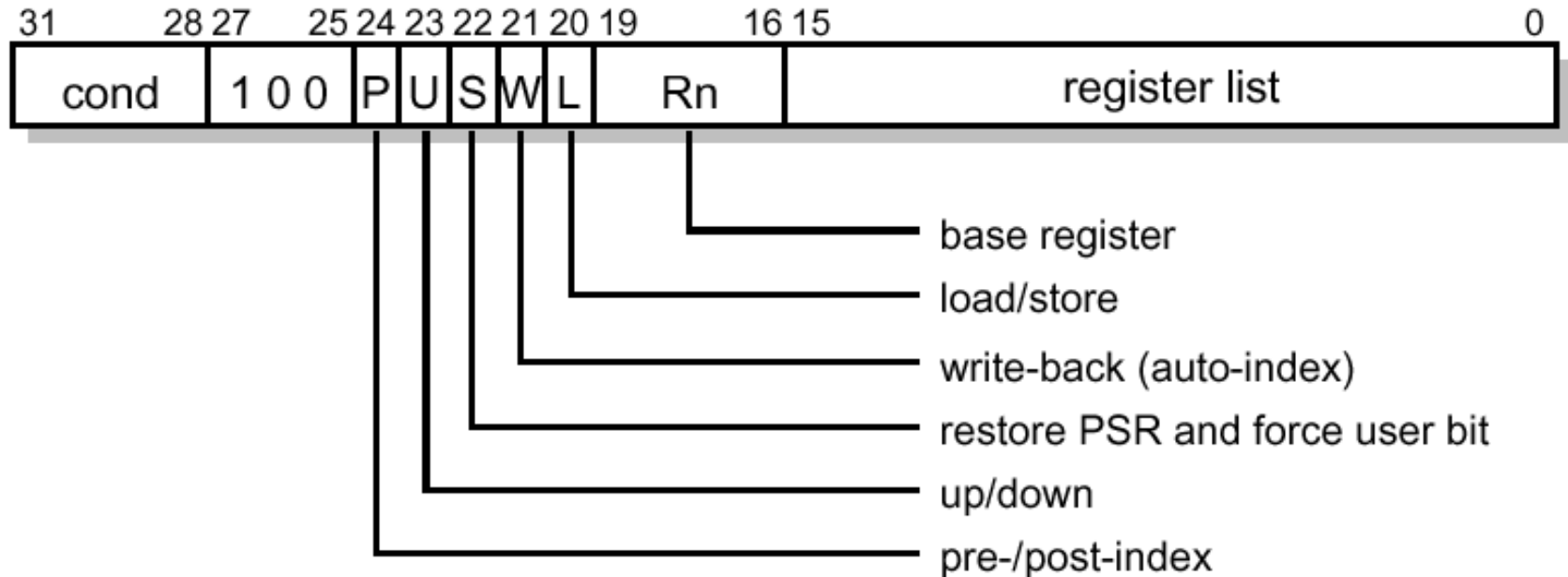
- ❖ So far the base address register, r1 below, has always been written back (also called auto-indexing). You can choose NOT update this pointer register removing the "!". All variants of LDM/STM have **optional** base register update.

```
LDMIA      r1, {r2-r9}    ; r2 := mem32[r1]
                               ; .....
                               ; r9 := mem32[r1+28]
```

```
LDMIA      r1!, {r2-r9}   ; r2 := mem32[r1]
                               ; .....
                               ; r9 := mem32[r1+28]
                               ; r1 := r1 + 32 (8 registers)
```

"!" indicates r1  
is changed

# Multiple register transfer instructions



- ❖ Register list has one bit per register
- ❖ bit 0 = 1 => load/store r0; bit 1 = 1 => load/store r1; etc
- ❖ **STMIA**      **r13!, {r0-r2, r14}**      **E8AD 4007**

# Lecture 9: Miscellaneous

---

- ❖ Multiplication
- ❖ Overview of machine instructions
- ❖ Machine instruction timing



# ARM Multiply instructions

---

- ❖ The original ARM 1 architecture did not have multiply instructions
  - ✦ 32X32->32 bit (least significant 32 bits of result kept) was added for **ARM 3** and above
  - ✦ 32X32->64 multiplication was added for **ARM7DM** and above.
- ❖ The multiplications were “shoe-horned” into the data processing instructions, using bit combinations specifying shifts that were previously **unused** and **illegal**.

# Multiply in detail

- ❖ MUL, MLA were the original (32 bit LSW result) instructions

- ★ Why does it not matter whether they are signed or unsigned?

- ❖ Later architectures added 64 bit results

**Register operands only**  
**No constants, no shifts**

**NB d & m must be different for MUL, MLA**

## ARM3 and above

MUL	rd, rm, rs	multiply (32 bit)	$Rd := (Rm * Rs)[31:0]$
MLA	rd, rm, rs, rn	multiply-acc (32 bit)	$Rd := (Rm * Rs)[31:0] + Rn$
UMULL	rl, rh, rm, rs	unsigned multiply	$(Rh:RI) := Rm * Rs$
UMLAL	rl, rh, rm, rs	unsigned multiply-acc	$(Rh:RI) := (Rh:RI) + Rm * Rs$
SMULL	rl, rh, rm, rs	signed multiply	$(Rh:RI) := Rm * Rs$
SMLAL	rl, rh, rm, rs	signed multiply-acc	$(Rh:RI) := (Rh:RI) + Rm * Rs$

## ARM7DM core and above (64 bit multiply result)

# Example of using ARM Multiplier

❖ This calculates a 64 bit scalar product of two signed vectors, each 20 words long:

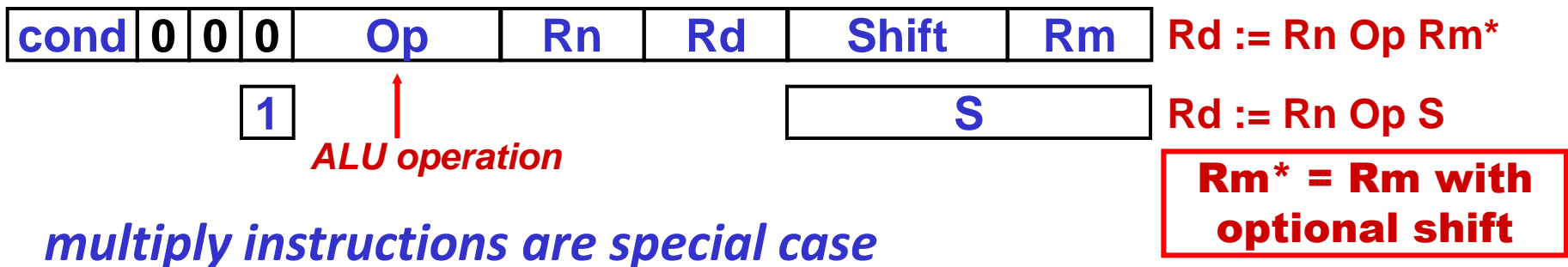
$$z = \sum_{j=0}^{19} x_j * y_j$$

- ❖ r8 and r9 point to the two vectors  $x_j$  and  $y_j$
- ❖ r11 is the loop counter
- ❖ r7:r6 stores the result

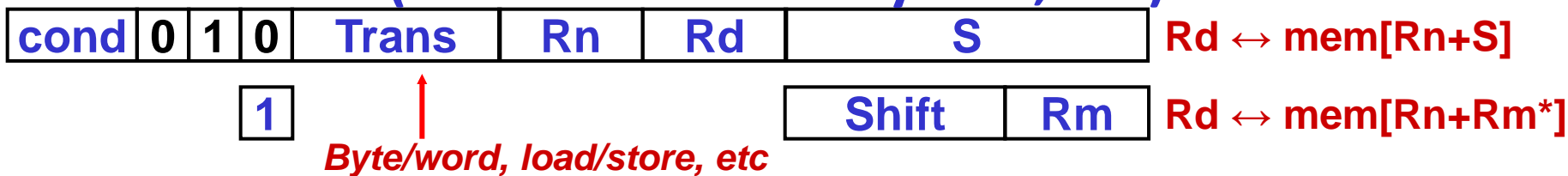
	MOV	r11, #20	; initialize loop counter
	MOV	r7, #0	; initialize 32 bit total
	MOV	r6, #0	
LOOP	LDR	r0, [r8], #4	; get x component
	LDR	r1, [r9], #4	; .... and y component
	SMLAL	r6, r7, r0, r1	; accumulate product
	SUBS	r11, r11, #1	; decrement loop counter
	BNE	LOOP	; loop 20 times

# ARM Machine Instruction Overview (1)

## Data processing (ADD,SUB,CMP,MOV)



## Data transfer (to or from memory LDR,STR)



## Multiple register transfer



# Overview (2)

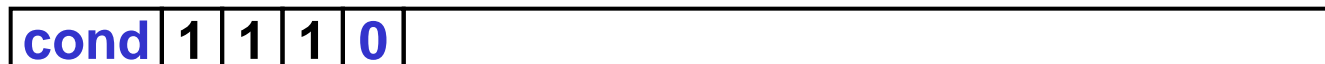
---

## Branch B, BL, BNE, BMI...



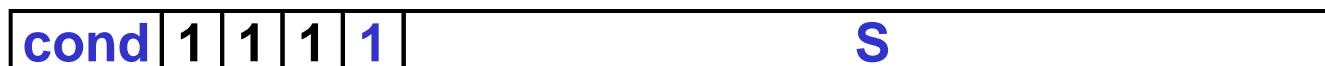
0 L = 0 => Branch, B ...

1 L = 1 => Branch and link (R14 := PC), BL ...



coprocessor  
interface

## Software Interrupt (SWI)



Simulate hardware  
interrupt: S is  
passed to handler

# ARM Instruction Timing

*Exact instruction timing is very complex and depends in general on memory cycle times which are system dependent. The table below gives an approximate guide.*

Instruction	Typical execution time (cycles)
Any instruction, <b>with condition false</b>	1
data processing (all except register-valued shifts)	1 (+3 if PC is dest)
data processing (register-valued shifts): <b>MOV R1, R2, lsl R3</b>	2 (+3 if PC is dest)
<b>LDR,LDRB, STR, STRB</b>	4
<b>LDM</b> (n registers)	n+3 (+3 more if PC is loaded)
<b>STM</b> (n registers)	n+3
<b>B, BL</b>	4
Multiply	7-14

# Instruction Timing Notes

---

- ❖ Most instructions take 1 cycle - RISC
- ❖ Memory reference takes longer (4 cycles typically)
- ❖ Branch takes longer (4 cycles)
  - ✦ Writing to PC => branch
- ❖ ALL instructions take 1 cycle if not executed (condition false)
- ❖ "register-valued shift" is special case 2 cycles
  - ✦ Make sure you know what a register-valued shift is (slide 2.58)!
- ❖ Multiply takes a lot longer though exact timing depends on data and also on ARM core - later cores have more efficient hardware multiply
- ❖ Instruction timing is hardware-dependent. Not part of Instruction Set Architecture