



Xi'an Jiaotong-Liverpool University

西交利物浦大學

# ARM Assembly – Part 2

Jianjun Chen ([Jianjun.Chen@xjtlu.edu.cn](mailto:Jianjun.Chen@xjtlu.edu.cn))

# Contents

- Accessing the Main Memory
- Flags and the CPSR Register
  - Setting flags
  - Instructions that read flags
    - Branch instruction
    - Arithmetic with the carry bit.

# Accessing the Main Memory

Preparing data in memory

Store and Load

# Declaring Large Data in Memory

- You can use `mov` to assign values to a register.
- The range of immediate values is limited.
- To create larger values, you can use the `OR` instruction to combine small parts:

`example_3_0.s`

```
mov          r0, #0x63000000
orr          r0, r0, #0x00640000
orr          r0, r0, #0x00006500
orr          r0, r0, #0x00000066
```

- The process is tedious though.

# Declaring Large Data in Memory

- Instead of combining immediate values, you can declare words with data or constants or empty words directly in memory and then load it to registers.

Opcode	Meaning	Format
DCD	Declare <u>Word(s)</u> in Memory	<b>name</b> DCD value_1, value_2, ... value_N
EQU	Create a symbol in the symbol table. Can also create constants	<b>name</b> equ expression
FILL	Declare <u>Empty Word(s)</u> in Memory	{ <b>name</b> } FILL N N must be a multiple of 4

- The names of these memory blocks are called **symbols**.
- DCD, EQU and FILL are called **assembler directives**.

# DCD and FILL

- **DCD**: Each piece of integer is stored in a separate word.
- **FILL**: These named words in memory will be filled with zeros.

See “example\_3\_1.s”

The screenshot displays an ARM assembly simulator interface. The assembly code on the left is as follows:

```
1 ; prepare data in the memory
2 integer1 dcd 0x12, 0x13
3 const1 equ 0x14
4 const2 equ 0x14 + 0x15
5 var1 fill 16, 16 bytes = 4 words
6
7 ; starting memory address (first byte)
8 adr r0, integer1
9 adr r1, const1
10 adr r2, var1
11
12 ; store/load with immediate offset
```

The memory view on the right shows the following data:

Symbol	Address	Value
integer1	0x200	0x12
	0x204	0x13
var1	0x208	0x0
	0x20C	0x0
	0x210	0x0
	0x214	0x0

Uninitialized memory is zeroed

# EQU

- Constants can be found in the “Symbols” window:

See “example\_3\_1.s”

The screenshot shows an ARM assembler IDE with the following components:

- Assembly Code:**

```
; prepare data in the memory
integer1 dcd 0x12, 0x13
const1 equ 0x14
const2 equ 0x14 + 0x15
var1 fill 16 ;16 bytes = 4 words

; starting memory address (first byte)
adr r0, integer1
```
- Registers Window:** Shows 'Data Symbol' and 'Value' for 'integer1' (0x200) and 'var1' (0x208).
- Memory Window:** Empty.
- Symbols Window:** Divided into 'Data Symbol' and 'EQU Symbol' sections. The 'EQU Symbol' section is highlighted with a red box, showing:

EQU Symbol	Value
const1	0x14
const2	0x29

# Symbols and Values

- Each symbol is associated with a value.
- If the symbol is created with `DCD` or `FILL`, the value is the memory address allocated.
- If the symbol is created with `EQU`, the value is just a normal constant.

Registers	Memory	Symbols
Data Symbol		Value
integer1		0x200
var1		0x208
EQU Symbol		Value
const1		0x14
const2		0x29

Registers	Memory	Symbols
Enable Byte View		
Enable Reverse Direction		
Symbol	Address	Value
integer1	0x200	0x12
	0x204	0x13
	0x208	0x0
	0x20C	0x0
	0x210	0x0
var1	0x214	0x0



# Getting the Address of a Symbol

- To get the address/value associated with a symbol, use `ADR`:

```
ADR{cond} Rd, label
```

- Example 3.1:

- After running the code on the right.
- Registers' values are changed.

```
adr r0, integer1  
adr r1, const1  
adr r2, var1
```

Registers	Memory	Symbols
Data Symbol		Value
integer1	0x200	
var1	0x208	
EQU Symbol		Value
const1	0x14	
const2	0x29	

Registers	Memory	Symbols
R0	0x200	
R1	0x14	
R2	0x208	

# Loading Data from Memory

- With a memory address in a register, you can load data from that address to a register:

*Ldr* *Rt, [Rn]*

items between {} are optional

**LDR**{type}{cond} Rt, [Rn {, #offset}]

**LDR**{type}{cond} Rt, [Rn, #offset]!

**LDR**{type}{cond} Rt, [Rn], #offset

*pre-index*  
*post-index*

- Items:
  - Rt is the register to load.
  - Rn is the register on which the memory address is based.
  - #offset is the amount of offset in bytes.
  - {type}: can be either B (store a byte) or omitted (store a word).
  - {cond}: explained in “Flags and Conditions”.

# Storing Data to Memory

- STR stores the content of  $R_t$  to the main memory :

**STR**{type}{cond}  $R_t$ , [ $R_n$  {, #offset}]  
**STR**{type}{cond}  $R_t$ , [ $R_n$ , #offset]!  
**STR**{type}{cond}  $R_t$ , [ $R_n$ ], #offset

- Items:
  - $R_t$  is the register holding the data to be stored into the memory.
  - $R_n$  is a register on which the memory address is based.
  - #offset is an immediate value indicating the offset from the memory address of  $R_n$ .
  - {type}: Can be either B (store a byte) or omitted (store a word).
  - {cond}: explained in “Flags and Conditions”.

# Store/Load with Immediate Offset

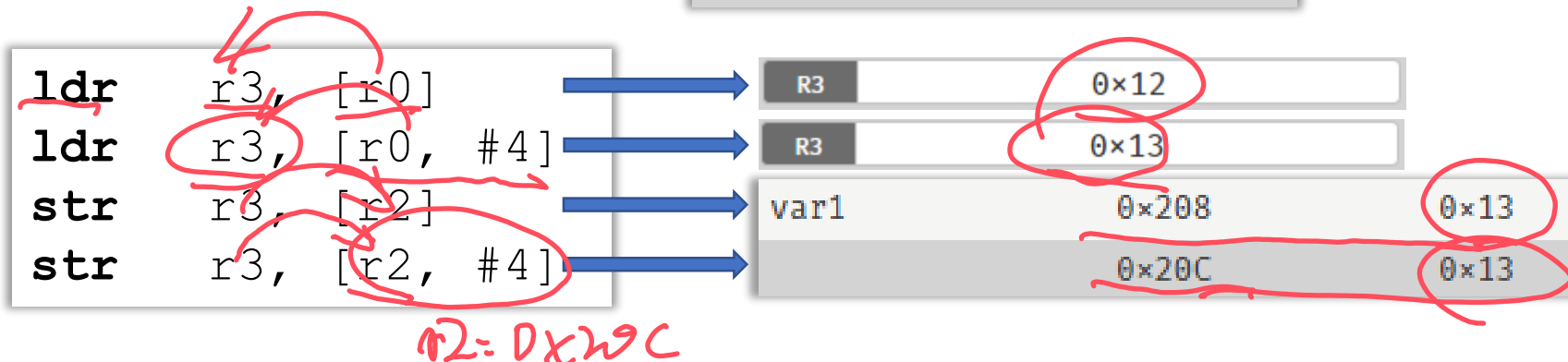
- Syntax:

```
STR{type}{cond} Rt, [Rn {, #offset}]  
LDR{type}{cond} Rt, [Rn {, #offset}]
```

- In example 3.1:

R0	0x200
R1	0x14
R2	0x208

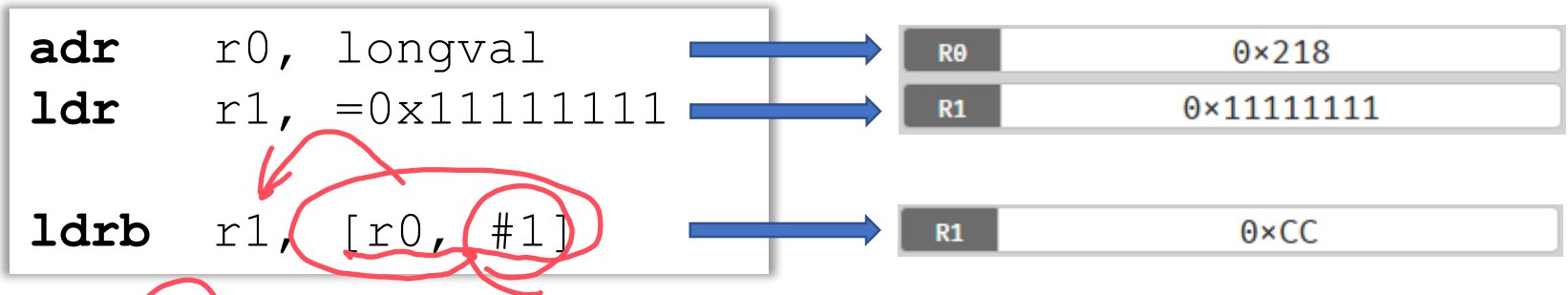
Symbol	Address	Value
integer1	0x200	0x12
	0x204	0x13
var1	0x208	0x0
	0x20C	0x0
	0x210	0x0
	0x214	0x0



# Store/Load with Immediate Offset

- Also in example 3.1. 

longval	0x218	0xAABBCCDD
---------	-------	------------
- **LDRB** clears the whole register, then load the value from memory to register:



- **STRB** writes the least significant byte of a register to the target memory byte:



# LDR and STR: More { type }

- Only LDR/STR and LDRB/STRB are supported by VisUAL.
- Other modes are available on real ARM processors.

Table A4-12 Load/store instructions

Data type	Load	Store	Load unprivileged	Store unprivileged	Load-Exclusive	Store-Exclusive
32-bit word	LDR	STR	LDRT	STRT	LDREX	STREX
16-bit halfword	-	STRH	-	STRHT	-	STREXH
16-bit unsigned halfword	LDRH	-	LDRHT	-	LDREXH	-
16-bit signed halfword	LDRSH	-	LDRSHT	-	-	-
8-bit byte	-	STRB	-	STRBT	-	STREXB
8-bit unsigned byte	LDRB	-	LDRBT	-	LDREXB	-
8-bit signed byte	LDRSB	-	LDRSBT	-	-	-
Two 32-bit words	LDRD	STRD	-	-	-	-
64-bit doubleword	-	-	-	-	LDREXD	STREXD

# LDR and Memory Alignment

- Example 3.2:

Symbol	Address	Value
integer1	0x200	0x61626364
	0x204	0x65666768

- The memory address involved in LDR is aligned to words:  
(must be a multiply of 4)

```
adr    r0, integer1
ldr    r2, [r0]
```

Loading from 0x200, legal address



R2

0x61626364

```
add    r3, r0, #1
ldr    r2, [r3]
```

Loading from 0x201, illegal address



Error on line 13: word address 513 must be divisible by 4

- The offset value must also be a multiply of 4

```
ldr    r2, [r0, #4]
```

Offset is 4, correct usage

```
ldr    r2, [r0, #1]
```

Offset is 1, incorrect usage

# LDRB and Memory Alignment

- Example 3.2:

Symbol	Address	Value
integer1	0x200	0x61626364
	0x204	0x65666768

- The memory address involved in LDRB is aligned to bytes:

```
adr    r0, integer1
ldrb   r2, [r0]
```

Loading from 0x200, legal address



R2

0x64

```
add    r3, r0, #1
ldrb   r2, [r3]
```

Loading from 0x201, also legal



R2

0x63

- The offset value can be any integers:

```
ldrb   r2, [r0, #0]
ldrb   r2, [r0, #1]
ldrb   r2, [r0, #2]
ldrb   r2, [r0, #3]
```

All correct



R2

0x64

R3

0x63

R4

0x62

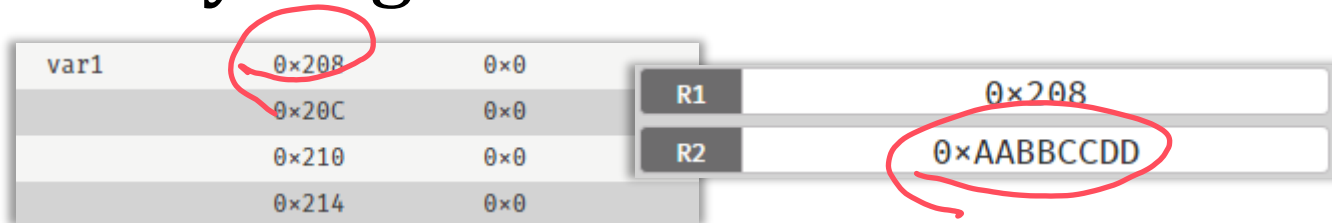
R5

0x61

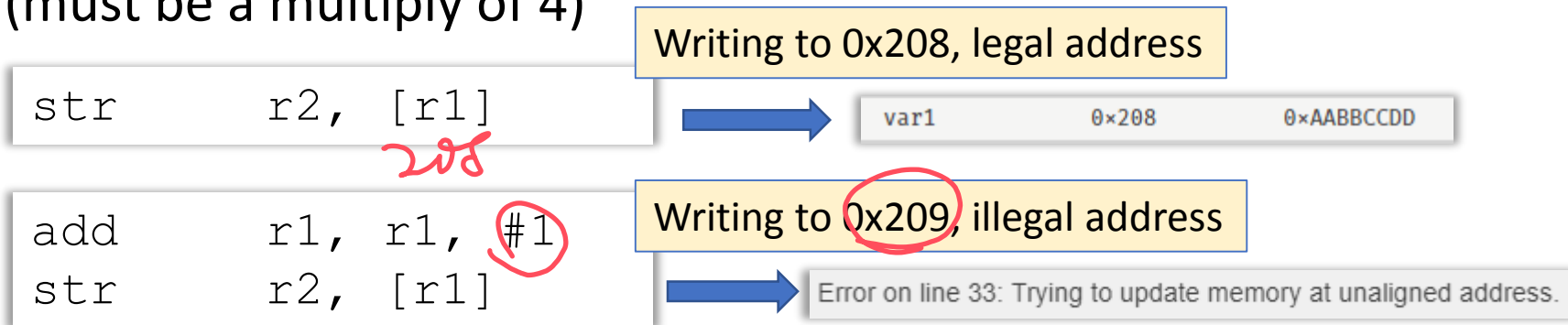


# STR and Memory Alignment

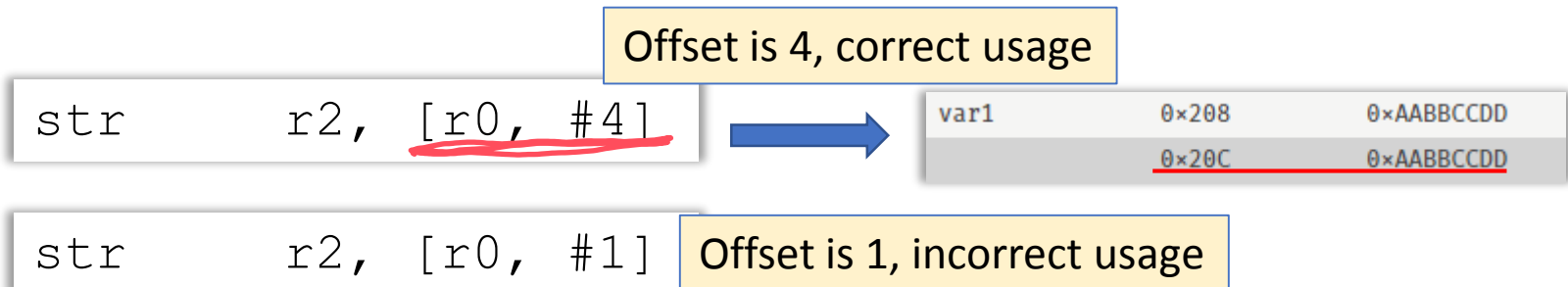
- Example 3.2:



- The memory address involved in STR is aligned to words: (must be a multiply of 4)



- The offset value must also be a multiply of 4



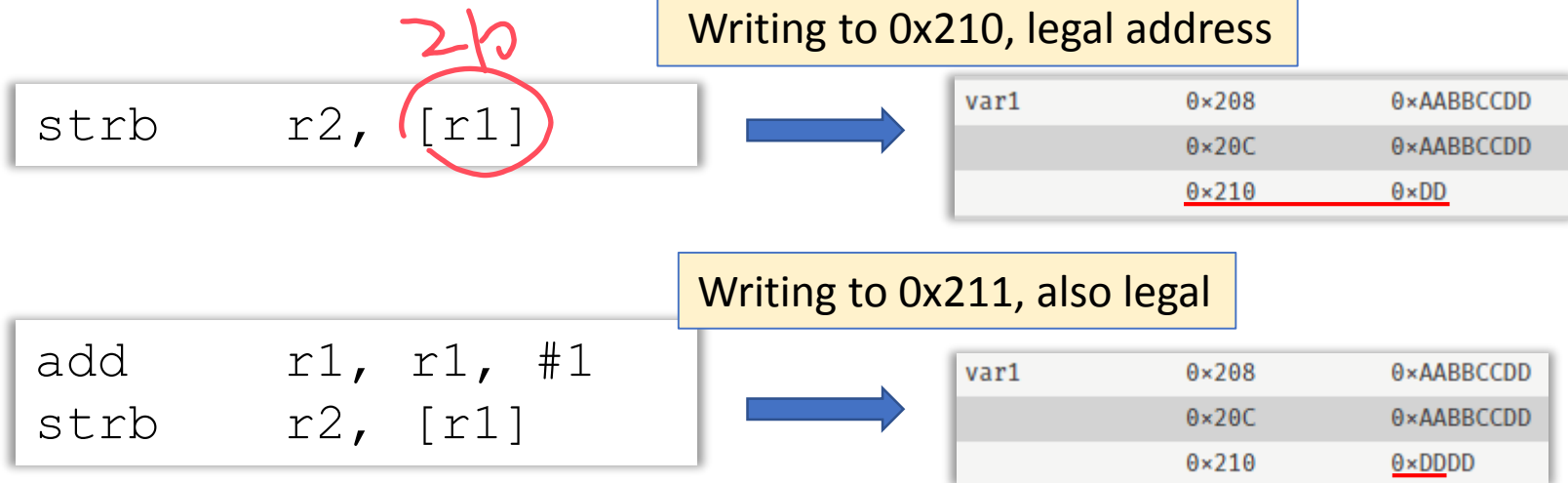
# STRB and Memory Alignment

- Example 3.2:

var1	0x208	0x0
	0x20C	0x0
	0x210	0x0
	0x214	0x0

R1	0x210
R2	0xAABBCCDD

- The memory address involved in STRB is aligned to bytes:



- The offset value can be any integers.
  - Go read example 3.2

# S/L with Pre-Indexed Immediate Offset

- Syntax:

**STR**{type}{cond} Rt, [Rn, #offset]!  
**LDR**{type}{cond} Rt, [Rn, #offset]!

- Mechanism:

- The offset is added to or subtracted from the base register to form the memory address.
- The base register is then updated with this new address, to permit automatic indexing through an array or memory block.

- See example 3.3

- Works like `Array[++x]`.

# S/L with Post-Indexed Immediate Offset

- Syntax:

```
STR{type}{cond} Rt, [Rn], #offset  
LDR{type}{cond} Rt, [Rn], #offset
```

- Mechanism:

- The value of the base register alone is used as the memory address.
- The offset is then added to or subtracted from the base register. and this value is stored back in the base register, to permit automatic indexing through an array or memory block.

- See example 3.3

- Works like `Array[x++]`.

# LDR Pseudo Instruction

- You can use the LDR pseudo instruction to obtain the address of a label or setting a large value:

```
ldr    r3, =integer1
```



```
ldr    r3, integer1
```

```
ldr    r3, =0xAABBCCDD
```



```
mov    r4, #0xDD  
orr    r4, r4, #0xCC00  
orr    r4, r4, #0xBB0000  
orr    r4, r4, #0xAA000000
```

- ARM assembler will automatically convert them into their corresponding “true” instructions when executed.
- In example 3.3.

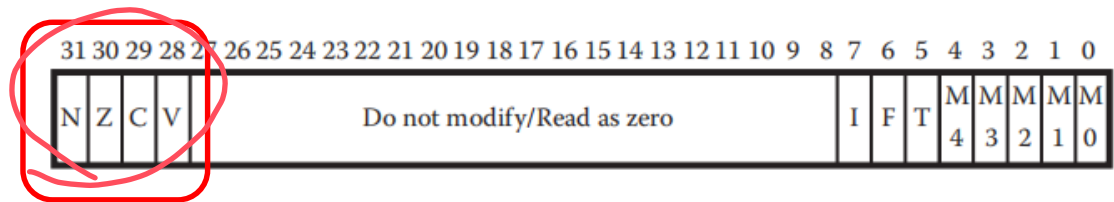
# Other Variants

- There are more variants of STR/LDR supported in real arm CPUs.
- Check the official instruction sets:  
[https://www.keil.com/support/man/docs/armasm/armasm\\_dom1361289906890.htm](https://www.keil.com/support/man/docs/armasm/armasm_dom1361289906890.htm)

# Condition Flags and Related Instructions

Flags, Branch, Condition code, branch

# Flags



- Flags are small pieces of memory (bits) that indicate some status.
  - Stored in CPSR and SPSR registers
- The bits are set according to the most recently executed ALU instruction that has the special “s” suffix.

**MOV**{S}{cond} Rd, Operand2

- Flags are kept until another s-suffix instruction is executed.
- Some other instructions like (CMP) can also set flags.
- These bits can be used for conditional execution of subsequent instructions.



# Flags

- **N**: Negative. The N flag is set by an instruction if the result is negative. In practice, N is set to the two's complement sign bit of the result (bit 31).
- **Z**: Zero. The Z flag is set if the result of the flag-setting instruction is zero.   
*Sign*  
 $0x7 - 0x7 = 0$
- **C**: Carry (or Unsigned Overflow). The C flag is set when an operation results in a carry, or when a subtraction results in no borrow.   
*last borrow*
- **V**: (Signed) Overflow. The V flag works the same as the C flag, but for signed operations.   
*负 + 负 = 正 / 正 + 正 = 负*

# Instructions that **Set** Flags (s suffix)

- Compare (CMP):

```
CMP{cond} Rn, #imm32  
CMP{cond} Rn, Rm
```

```
CMN{cond} Rn, #imm32  
CMN{cond} Rn, Rm
```

- Mechanism:

- Calculates the result of  $(Rn - \#imm32)$  or  $(Rn - Rm)$ .
- If the result is negative, set the N flag to 1.
- If the result is zero, set the Z flag to 1.
- If the result has carry digit, set the C flag to 1.
- If the calculation has overflow, set the V flag to 1.

- A similar instruction is called compare negative (CMN):

- Calculates  $(Rn + \#const)$  or  $(Rn + Rm)$
- Then set NZCV flags.

# Instructions that Set Flags

- See example 3.4:

- $0xffffffff - 0x00000001 \rightarrow$  **N** and **C**

- $= 0xffffffff$

ab 111 ..

- $0xffffffff + 0x00000001 \rightarrow$  **Z** and **C**

- $= 0x0$

z=1

- If treated as unsigned number, the calculation overflows

- $0x00000001 + 0x0000000f \rightarrow$  None set!

- $= 0x10$

正

正

N Z C V

- $0x0000000f + 0x7fffffff \rightarrow$  **N** and **V**

- $= 0x8000000e$  (a negative number)

- $0x7fffffff$  is the largest representable positive number

- If operands are treated as signed number, the result is negative, there's overflow. That's why V is set.

N Z C V

# Signed Overflow & Unsigned Overflow

正 + 正 = 负  
负 + 负 = 正

- For example, `0x7fffffff` is the largest positive two's complement integer that can be represented in 32 bits.
  - So `0x7fffffff + 0x7fffffff = 0xffffffff` triggers a signed overflow.
- But if `0x7fffffff` is treated as an unsigned number, the result does not overflow.
  - Unsigned overflow is indicated by the carry bit set.

# Instructions that **Use** Flags (cond)

- Branching to a target address:

**B**{cond} label

- Items:

- Branch redirects the execution of CPU to a new position indicated by a label.
- A label is a symbol that represents the memory address of an instruction or data.
- {cond}: We have seen it for many times, it will be explained soon!

- Example:

- This is an infinite loop.

```
        mov     r0, #1
        mov     r1, #5
loop:   add     r0, r0, #1
        b       loop
```

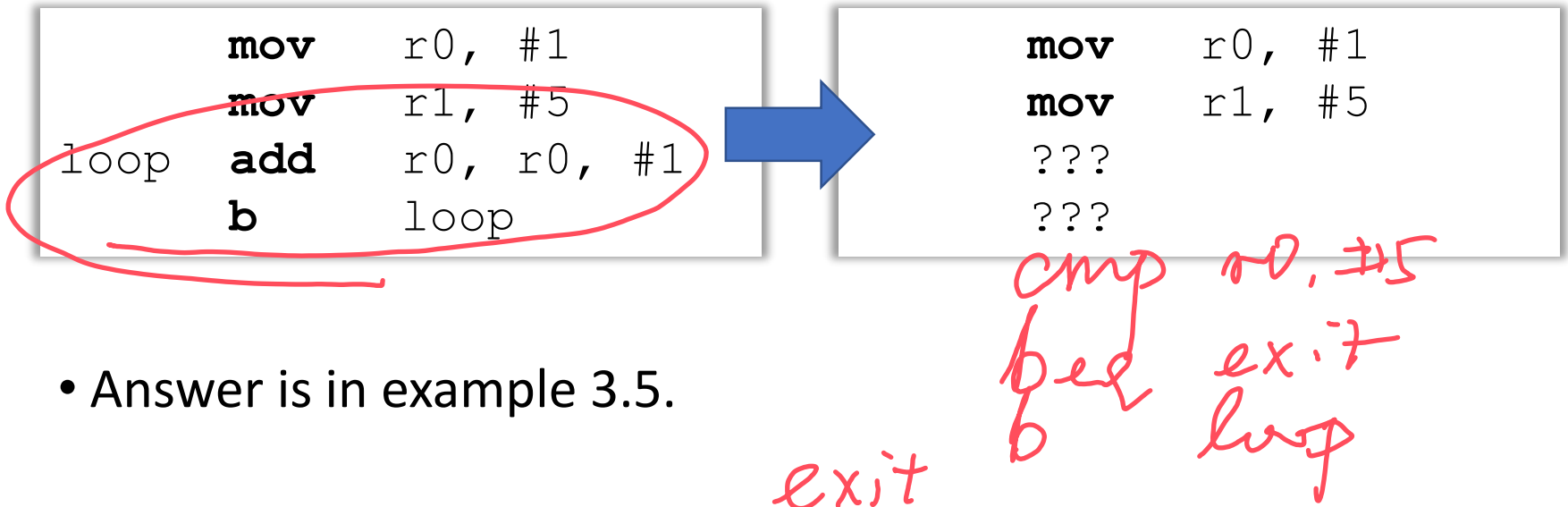
# Condition Codes

st (cond)

Code	Meaning (for CMP or SUBS)	Flags Tested
eq	equal	Z==1
ne	Not equal	Z==0
cs or hs	Unsigned higher or same (or carry set).	C==1
cc or lo	Unsigned lower (or carry clear)	C==0
mi	Negative ( <b>m</b> inus)	N==1
pl	Positive or zero. ( <b>p</b> lus)	N==0
vs	Signed overflow. (V set)	V==1
vc	No signed overflow. (V clear)	V==0
hi	Unsigned higher	(C==1) && (Z==0)
ls	Unsigned lower or same	(C==0)    (Z==1)
ge	Signed greater than or equal	N==V
lt	Signed less than	N!=V
gt	Signed greater than	(Z==0) && (N==V)
le	Signed less than or equal	(Z==1)    (N!=V)
al (or omitted)	Always executed	None tested

# Branch

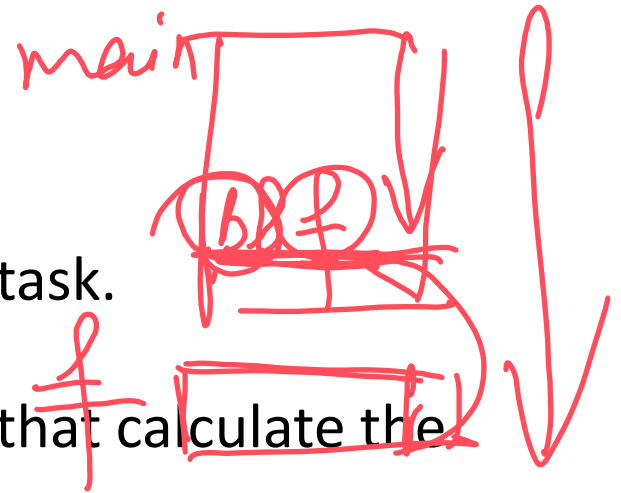
- You can apply anyone of the condition code to the {cond} field of an instruction (that supports {cond}).
- How can we redesign the program so that it finishes when  $r0 = r1$ ?



- Answer is in example 3.5.

# BL & BX

- The previous solution works for a small task.
- Imagine we want to develop a function that calculate the sum of three numbers.
  - These three numbers are stored in `r0`, `r1` and `r2`.
  - The sum should be put into `r5`.
  - We wish to call this function whenever needed.
- With branch (See the second part of example 3.5):
  - We can jump to the sum function with its label.
  - But how can we make sure the program can return to where the sum function is called if there are multiple places call it?
  - We can mark it in a register, then use if/else in the sum function to jump back to the corresponding position. (Elegant solution?)





# BL & BX

- Solution: use “Branch with Link” together with “Branch and Exchange”

**BL**{cond} label

- The BL instruction causes a branch to label, and copies the address of the next instruction into LR (R14, the link register).

**BX**{cond} Rm

- The BX instruction causes a branch to the address contained in Rm
  - And exchanges the instruction set, if required. (This is not supported in VisUAL)
  - [https://www.keil.com/support/man/docs/armasm/armasm\\_dom1361289866466.htm](https://www.keil.com/support/man/docs/armasm/armasm_dom1361289866466.htm)

# BL & BX

- In general, BL allows a program to remember its previous position before jumping. The position is saved in LR.
- BX allows a program to jump to any point indicated by a register, which can be LR.
- Can you fix the problem of the second part of example 3.5 now?


# Solution for the get\_sum()

- Check example 3.6
- However, the VisUAL emulator **does not support bx**.
  - Alternative way? Change the program counter register directly!
  - See slides later.

# Arithmetic with Carry

- The carry (C) flag is set when an operation results in a carry, or when a subtraction results in no borrow.
  - For an addition, including the comparison instruction CMN, C is set to 1 if the addition produced a carry (that is, an unsigned overflow), and to 0 otherwise.
  - For a subtraction, including the comparison instruction CMP, C is set to 0 if the subtraction produced a borrow (that is, an unsigned underflow), and to 1 otherwise.
  - For non-additions/subtractions that incorporate a shift operation, C is set to the last bit shifted out of the value by the shifter.
  - For other non-additions/subtractions, C is normally left unchanged.

lsl/r 5



# Carry Bit: Subtraction

- The carry (C) flag is set (to 1) when an operation results in a carry, **or when a subtraction results in no borrow.**
  - For a subtraction, including the comparison instruction `CMP`, **C is set to 0** if the subtraction produced a borrow (that is, an unsigned underflow), and to 1 otherwise.
  - Produced a borrow: the first number being smaller than the second number.
- Some examples (more can be found in example 3.7):

Operands	C bit	Reason
10 - 11	0	10 is less than 11, produced borrow
10 - 0	1	There's no borrow
0xffffffff - 0x00000001	1	There's no borrow (example 3.4)

# Carry Bit: Shift Operations

- For non-additions/subtractions that incorporate a shift operation, C is set to the last bit shifted out of the value by the shifter. Examples:

examples	C	Reason
<code>mov r0, #0x00000001</code> <code>lsls r0, r0, #1</code>	0	0 at the most significant bit is shifted out.
<code>mov r0, #0x00000001</code> <code>lsrs r0, r0, #1</code>	1	1 at the least significant bit is shifted out.
<code>mov r0, #0x00000001</code> <code>lsrs r0, r0, #2</code>	0	1 at the LSB is first shifted out, then followed by 0.
<code>mov r0, #0x00000001</code> <code>rors r0, r0, #1</code>	1	1 at the LSB is rotated out
Do more experiments by yourself		

# Arithmetic with Carry Bit

- There are several arithmetic instructions that can include the carry bit as a part of input.

items between {}  
are optional

Name	Syntax
Add with Carry	<u>ADC</u> {S}{cond} {Rd}, Rn, Operand2
Subtract with Carry	<u>SBC</u> {S}{cond} {Rd}, Rn, Operand2
Reverse Subtract with Carry	<u>RSC</u> {S}{cond} {Rd}, Rn, Operand2

- Operand2 can be another register or an immediate.
- You can use ADC, SBC or RSC to synthesize multiword arithmetic.

# Arithmetic with Carry Bit

- When the carry bit is 1:

- `ADC r0, r1, #2`  $\rightarrow$   $r0 = r1 + 2 + 1$
- `SBC r0, r1, r2`  $\rightarrow$   $r0 = r1 - r2$
- `RSC r0, r1, r2`  $\rightarrow$   $r0 = r2 - r1$

- When the carry bit is 0:

- `ADC r0, r1, #2`  $\rightarrow$   $r0 = r1 + 2$
- `SBC r0, r1, r2`  $\rightarrow$   $r0 = r1 - r2 - 1$
- `RSC r0, r1, r2`  $\rightarrow$   $r0 = r2 - r1 - 1$



# Multiword Arithmetic Example

- Assume two 64-bit integers are stored in (r0, r1) and (r2, r3), the following two instructions will do a multiword addition:

```
ADDS r4, r0, r2 ; adding the least significant words  
ADC r5, r1, r3 ; adding the most significant words
```

- Assume we want to calculate:

$$0x4000000000 - 0x1800000000 = 0x3E80000000$$

Handwritten calculation:  
40  
- 18  
---  
22

R0	0x40
R1	0x0
R2	0x1
R3	0x80000000

R4	0x80000000
R5	0x3E

*C=0*

```
subs r4, r1, r3  
sbc r5, r0, r2
```

Handwritten calculation:  
40 - 1 - 1 = 3E