UNIVERSITEIT
iYUNIVESITHI
STELLENBOSCH
UNIVERSITY

100
1918 · 2018

forward together · saam vorentoe · masiye phambili

Computer Systems / Rekenaarstelsels 245 - 2020

Lecture 3

Review – Operators, Logic, Loops, Pointers, Arrays and Accessing Memory

Hersiening – Bewerkings, Logika, Lusse, Wysers, Skikkings en Geheue Toegang

Dr Rensu Theart & Dr Lourens Visagie

Lecture Overview

- Operators (C and ASM)
  - C Operators and ASM equivalent
  - Precedence
  - Boolean vs bitwise logic
  - Bitwise logic operators
  - Bit shift operations (multiply, divide)
- Conditional execution
  - Boolean If-then-else, and switch statements
  - Loops
  - `IT` instructions
- Memory and addressing
- Pointers
- Arrays (as pointers)
- Structs
- Endianness

Operators / Operatore

Table eC.3 Operators listed by decreasing precedence

Category	Operator	Description	Example
Unary	`++`	post-increment	`a++; // a = a+1`
	`--`	post-decrement	`x--; // x = x–1`
	`&`	memory address of a variable	`x = &y; // x = the memory // address of y`
	`~`	bitwise NOT	`z = ~a;`
	`!`	Boolean NOT	`!x`
	`-`	negation	`y = -a;`
	`++`	pre-increment	`++a; // a = a+1`
	`--`	pre-decrement	`--x; // x = x–1`
	`(type)`	casts a variable to (type)	`x = (int)c; // cast c to an // int and assign it to x`
	`sizeof()`	size of a variable or type in bytes	`long int y; x = sizeof(y); // x = 4`

Operators / Operatore

Multiplicative	`*`	multiplication	`y = x * 12;`
	`/`	division	`z = 9 / 3; // z = 3`
	`%`	modulo	`z = 5 % 2; // z = 1`
Additive	`+`	addition	`y = a + 2;`
	`–`	subtraction	`y = a - 2;`
Bitwise Shift	`<<`	bitshift left	`z = 5 << 2; // z = 0b00010100`
	`>>`	bitshift right	`x = 9 >> 3; // x = 0b00000001`
Relational	`==`	equals	`y == 2`
	`!=`	not equals	`x != 7`
	`<`	less than	`y < 12`
	`>`	greater than	`val > max`
	`<=`	less than or equal	`z <= 2`
	`>=`	greater than or equal	`y >= 10`

(continued)

Operators / Operatore

Table eC.3 **Operators listed by decreasing precedence—Cont'd**

Category	Operator	Description	Example
Bitwise	&	bitwise AND	y = a & 15;
	^	bitwise XOR	y = 2 ^ 3;
	|	bitwise OR	y = a | b;
Logical	&&	Boolean AND	x && y
	||	Boolean OR	x || y

Ternary	? :	ternary operator	y = x ? a : b; // if x is TRUE,
			// y=a, else y=b

Assignment	=	assignment	x = 22;
	+=	addition and assignment	y += 3; // y = y + 3
	−=	subtraction and assignment	z −= 10; // z = z - 10
	*=	multiplication and assignment	x *= 4; // x = x * 4
	/=	division and assignment	y /= 10; // y = y / 10
	%=	modulo and assignment	x %= 4; // x = x % 4
	>>=	bitwise right-shift and assignment	x >>= 5; // x = x>>5
	<<=	bitwise left-shift and assignment	x <<= 2; // x = x<<2
	&=	bitwise AND and assignment	y &= 15; // y = y & 15
	|=	bitwise OR and assignment	x |= y; // x = x | y
	^=	bitwise XOR and assignment	x ^= y; // x = x ^ y

Operators / Operatore

```c
r1++;

char ch = 'A';
char* r2 = &ch;

if (r3 && r4) {

}




int r5 = r3 & r4;
```

```asm
        ADD     r1, r1, #1

ch      DCB     65  ;'A'
        LDR     r2, =ch

        CMP     r3, #0
        BEQ     ELSE
        CMP     r4, #0
        BEQ     ELSE
        ;do     this if true
ELSE

        AND     r5, r3, r4
```

You must know and apply the equivalent
ASM for each operation in C (and vice versa)

Boolean vs bitwise logic / Boolse vs. per-bis logika

- In the C programming language, what is the difference between

  `c = a & b;` and `c = a && b;`

- A single &, or | implies **bit-wise operation** (the logic operation is applied to every bit of the variable):

  $$01010101_2 \mid 10101010_2 = 11111111_2$$

- A double && or || implies **logic operation**.

  $$01010101_2 \mid\mid 10101010_2 = 00000001_2$$

  - Each operand is either TRUE of FALSE. 0 = FALSE, and anything else is TRUE. Result of logic operation is either 0 or 1.

- Use bit-wise operators if you want to manipulate bits in a variable/register. Use logic operators if you want to test for logic conditions.
  - **Note** that ARM instructions such as `AND`, `ORR`, `EOR` are `BIC` are all <u>bitwise</u>.

- In C kode, gebruik bis-wye bewerkings om bisse in 'n veranderlike of register te manipuleer. Gebruik logiese bewerkings om te toets vir logika

Programming tips - Operators

Precedence

```
if (a && b || c && d)
    do_things();
```

Which will be evaluated first?

→ Add parentheses to make it explicit

If you use function calls in an if-statement, they might never execute

```
bool abool = false;
if (abool && do_stuff())
{…}
```

→ do_stuff() will only get called if abool is true.

Programming tips - Operators

In general, the size and type of storage the compiler will use for the result of an operation depends on the operands. For example:

```
int sum = 13;
int count = 100;
float avg = sum / count;
```

→ avg will have the value 0.0f. (Not 0.13f). This is because the operands are integers. So the result is placed in an integer, and only then converted to float

The right way:

```
float avg = (float) sum / (float) count;
```

→ forces compiler to use float operands

- Use **type casting** to make number and pointer conversions <u>explicit</u>
  - No ambiguity about what compiler might do
  - Improves readability and maintainability of your code
  - Removes compiler warnings

Bitwise logical operators / Per-bis logiese operatore

<table>
<tr><th>Bitwise Logic Operator</th><th>ARM Assembly</th><th>C syntax</th></tr>
<tr><td>NOT</td><td><code>MVN r0, r1</code></td><td><code>r0 = ~r1;</code></td></tr>
<tr><td>AND</td><td><code>AND r0, r1, r2</code></td><td><code>r0 = r1 & r2;</code></td></tr>
<tr><td>OR</td><td><code>ORR r0, r1, r2</code></td><td><code>r0 = r1 | r2;</code></td></tr>
<tr><td>XOR</td><td><code>EOR r0, r1, r2</code></td><td><code>r0 = r1 ^ r2;</code></td></tr>
<tr><td>Bitwise Clear</td><td><code>BIC r0, r1, r2</code></td><td><code>r0 = r1 & (~r5);</code></td></tr>
</table>

Bitwise logical operators / Per-bis logiese operatore

- How can check or manipulate individual bits in a number or register?

  ⇒ By using bitwise logic operators!
- ANDing a bit with 0 produces a 0 at the output while ANDing a bit with 1 produces the original bit.
- This can be used to create a **mask**.

```
 input: 1100 1010
  mask: 0110 0110
result: 0100 0010
```

- Usage example: Mask and match bit pattern. Check if bit 0 is a 1 and bit 3 is a 0:

```
if ((input & 0b00001001) == 0b00000001)
    return 1;
```

Pattern to match	XXXX 0XX1
Mask	0000 1001
Result Bit pattern	0000 0001

Bitwise logical operators / Per-bis logiese operatore

- ORing a bit with 1 produces a 1 at the output and ORing a bit with 0 produces the original bit.
- This can be used to **force** certain bits to 1.
  - For example, 0x12345678 OR 0x000FFFF results in 0x1234FFFF (e.g. the high-order 16 bits are untouched, while the low-order 16 bits are forced to 1s).
- Usage example: Modify only certain bits in a memory mapped peripheral register (i.e. make a LED turn on)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **ODRy**: Port output data (y = 0..15)
These bits can be read and written by software.

```
uint32_t* reg_addr = (uint32_t*) 0x40020c10;  // address of register
uint32_t current_reg_val = *reg_addr;  // dereference pointer
uint32_t new_reg_val = (current_reg_val | 0b1000); // turn on bit 3
*reg_addr = new_reg_val;
```

Bitwise logical operators / Per-bis logiese operatore

- **BIC**ing a bit with 1 resets the bit (sets to 0) at the output while BICing a bit with 0 produces the original bit.
- This can be used to **force** certain bits to 0.
  - For example, 0x12345678 BIC 0x0000FFFF results in 0x12340000 (e.g. the high-order 16 bits are untouched, while the low-order 16 bits are forced to 0s).
- Can be used in similar situations as OR, but to set a bit to 0.

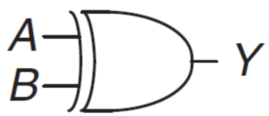
Bitwise logical operators / Per-bis logiese operatore

- How can we '**toggle**' or invert specific bits (set 0 to 1 and 1 to 0)?

⇒ Remember XOR.

- XORing a bit with 1 inverts it and XORing a bit with 0 produces the original bit.

XOR



$Y = A \oplus B$

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

Bitwise Logic Operator	ARM Assembly	C syntax
XOR	`EOR r0, r1, r2`	`r0 = r1 ^ r2;`

```
       Input: 1100 1010
Bits to flip: 0110 0110
      Result: 1010 1100
```

Bit operations / Bis bewerkings

- Bit shifting can also help with changing single bit values, without affecting the others.
- For the initial value $1010\ 1010_2$ (assuming word length of 8 bits)

Type	Result	Equivalent math.	Assembly	C syntax
Shift left	(1) 0101 010<u>0</u>	x 2^n	`LSL r1, r0, #n`	`a = b << n`
Shift right	<u>1</u>101 0101	/ 2^n	`ASR r1, r0, #n`	`a = b >> n`

- To set a bit (make it 1)

  `a = b | (1 << n)`

- To clear a bit (make it 0)

  `a = b & ~(1 << n)`

Set bit in position 6	
1010 1010	`b`
0100 0000	`1 << 6`
1<u>1</u>10 1010	`b | (1 << 6)`

Clear bit in position 5	
1010 1010	`b`
1101 1111	`~(1 << 5)`
10<u>0</u>0 1010	`b & ~(1 << 5)`

Bit operations / Bis bewerkings

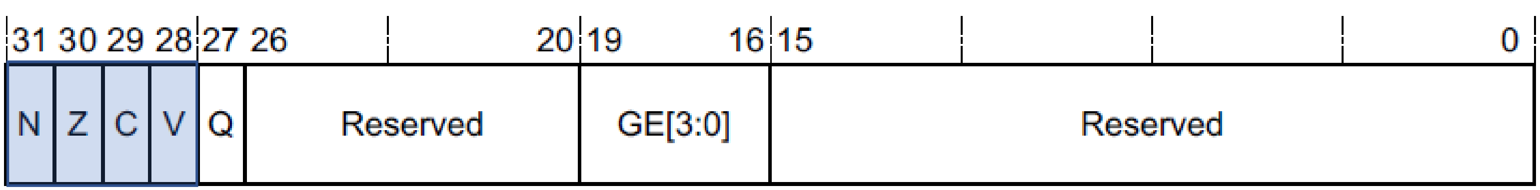
- To **multiply** by any number that is a power of 2, LEFT shift by n
  Math: $a = b * 2^n$
  C-code: `a = b << n;`

- To **divide** by any number that is a power of 2, RIGHT shift by n
  Math: $a = b / 2^n$
  C-code: `a = b >> n;`

- To get the **remainder of division**, when dividing by a power of 2
  Math: $a = b \bmod 2^n$
  C-code: `a = b & ((1 << n) - 1);`

Conditional execution / Voorwaardelike uitvoer

- In this modules we will work with the **ARM Cortex-M4** which uses the **ARMv7E-M** architecture.
  - The reference manual is on SunLearn or HERE
- ARM instructions optionally set **condition flags** based on the result by adding "S" to the instruction mnemonic (e.g. `ADDS R0, R1, R2`)
- Other dedicated instructions such as `CMP`, `CMN`, `TST` and `TEQ` also set the status flags without calculating a result.
  - Subsequent instructions then execute conditionally, depending on the state of those condition flags. (e.g. `ADDEQ R1, R2, R3` execute if Z=0)
- The ARM condition flags, also called status flags, are **negative (N)**, **zero (Z)**, **carry (C)**, and **overflow (V)**.
- These flags are set by the ALU and are held in the top 4 bits of the 32-bit **Application Program Status Register (APSR)**.



Conditional mnemonics

Table A7-1 Condition codes

cond	Mnemonic extension	Meaning, integer arithmetic	Meaning, floating-point arithmetic[a]	Condition flags
0000	EQ	Equal	Equal	Z == 1
0001	NE	Not equal	Not equal, or unordered	Z == 0
0010	CS[b]	Carry set	Greater than, equal, or unordered	C == 1
0011	CC[c]	Carry clear	Less than	C == 0
0100	MI	Minus, negative	Less than	N == 1
0101	PL	Plus, positive or zero	Greater than, equal, or unordered	N == 0
0110	VS	Overflow	Unordered	V == 1
0111	VC	No overflow	Not unordered	V == 0
1000	HI	Unsigned higher	Greater than, or unordered	C == 1 and Z == 0
1001	LS	Unsigned lower or same	Less than or equal	C == 0 or Z == 1
1010	GE	Signed greater than or equal	Greater than or equal	N == V
1011	LT	Signed less than	Less than, or unordered	N != V
1100	GT	Signed greater than	Greater than	Z == 0 and N == V
1101	LE	Signed less than or equal	Less than, equal, or unordered	Z == 1 or N != V
1110	None (AL)[d]	Always (unconditional)	Always (unconditional)	Any

a. Unordered means at least one NaN operand.

b. HS (unsigned higher or same) is a synonym for CS.

c. LO (unsigned lower) is a synonym for CC.

d. AL is an optional mnemonic extension for always, except in IT instructions. See *IT* on page A7-242 for details.

Conditional Execution / Voorwaardelike uitvoer

- The following C-code can be written as assembly code as follows:

C-Code

```c
if(apples == oranges)
{
    f = i + 1;
}
else
{
    f = f - i;
}
```

ARM Assembly Code

```
        ;       R0 = apples, R1 = oranges, R2 = f, R3 = i
        CMP     R0, R1 ; apples == oranges?
        BNE     L1 ; if not equal, skip if block
        ADD     R2, R3, #1 ; if block: f = i + 1
        B       L2 ; skip else block
L1
        SUB     R2, R2, R3 ; else block: f = f - i
L2
```

- **Remember that branching is expensive!** If you use compiler optimization the above assembly code could be reduced using conditional execution as:

```
CMP     R0, R1      ; apples == oranges?
ADDEQ   R2, R3, #1  ; f = i + 1 on equality (i.e., Z = 1)
SUBNE   R2, R2, R3  ; f = f - i on not equal (i.e., Z = 0)
```

Conditional statements – switch/case statements

- Switch statement is the same as nested if-then-else-if statements.

```
switch (button)
{
    case 1:
        amt = 20;
        break;
    case 2:
        amt = 50;
        break;
    case 3:
        amt = 100;
        break;
    default:
        amt = 0;
}
```

```
if (button == 1)
    amt = 20;
else if (button == 2)
    amt = 50;
else if (button == 3)
    amt = 100;
else
    amt = 0;
```

```
        ;           R0 = button, R1 = amt
        CMP         R0, #1 ; is button 1 ?
        MOVEQ       R1, #20 ; amt = 20 if button is 1
        BEQ         DONE ; break
        CMP         R0, #2 ; is button 2 ?
        MOVEQ       R1, #50 ; amt = 50 if button is 2
        BEQ         DONE ; break
        CMP         R0, #3 ; is button 3?
        MOVEQ       R1, #100 ; amt = 100 if button is 3
        BEQ         DONE ; break
        MOV         R1, #0 ; default amt = 0
DONE
```

Loops – while loops

- **while loops** repeatedly execute a block of code until a condition is **not** met.
- In assembly code, once the loop code finishes it branches back to the start of the loop, checks the condition and branches to the code after the loop if the **inverse** condition is met.
- The code below determines the value of x such that $2^x = 128$.

High-Level Code

```
int pow = 1;
int x = 0;
while (pow != 128)
{
    pow = pow * 2;
    x = x + 1;
}
```

ARM Assembly Code

```
        ;       R0 = pow, R1 = x
        MOV     R0, #1 ; pow = 1
        MOV     R1, #0 ; x = 0
WHILE
        CMP     R0, #128 ; pow != 128 ?
        BEQ     DONE ; if pow == 128, exit loop
        LSL     R0, R0, #1 ; pow = pow * 2
        ADD     R1, R1, #1 ; x = x + 1
        B       WHILE ; repeat loop
DONE
```

Loops – for loops (2)

High-Level Code

```
int i;
int sum = 0;
for (i = 0; i < 10; i = i + 1)
{
    sum = sum + i;
}
```

ARM Assembly Code

```
        ;       R0 = i, R1 = sum
        MOV     R1, #0 ; sum = 0
        MOV     R0, #0 ; i = 0 loop initialization
FOR
        CMP     R0, #10 ; i < 10 ? check condition
        BGE     DONE ; if (i >= 10) exit loop
        ADD     R1, R1, R0 ; sum = sum + i loop body
        ADD     R0, R0, #1 ; i = i + 1 loop operation
        B       FOR ; repeat loop
DONE
```

- It functions the same as a while loop, except now we have a variable (R0) that is initialized before the loop, and that is incremented before each loop branch.
- In C the condition is `i < 10`, assembly checks the inverse condition, `i >= 10`, to exit the loop.

Loops – Do/While loop

- What is the difference between

```
while (i > 0) { …; i--}
```

- and

```
do { …; i--} while (i > 0);
```

- Do/while loops will always execute at least once.

High-Level Code

```c
int i = 10;
int sum = 0;
do
{
    sum = sum + i
    i--;
} while (i > 0);
```

ARM Assembly Code

```
        ;           R0 = i, R1 = sum
        MOV         R1, #0 ; sum = 0
        MOV         R0, #10 ; i = 10 loop initialization
DOWHILE
        ADD         R1, R1, R0 ; sum = sum + i loop body
        SUB         R0, R0, #1 ; i = i - 1 loop operation
        CMP         R0, #0 ; i > 0 ? check condition
        BGT         DOWHILE ; repeat loop
DONE
```

- You could also use SUBS to combine SUB and CMP instructions.

Loops / Lusse

- Pay attention to loop indices!

```
int array[100];
for (int i = 0; i <= 100; i++)
{
    array[i] = i;
}
```

- The loop above will iterate 101 times, and in the final iteration it will write a value at array[100];
  But array[100] points to invalid memory!

- When is a variable (incl. loop counters) stored in memory? (and not just a CPU register)

⇒ depends on compiler

Conditional Execution / Voorwaardelike uitvoer

- The ARM Cortex M4 (used in the module) only supports the **Thumb instruction set** (specifically Thumb-2), and therefore instructions cannot execute conditionally.
- For this we use the **If-Then (IT) instruction** for small if-then-else statements, with 4 or less instructions.
  - Note that `IT` is a pseudo-instruction (does not generate any code) and there is no 32-bit equivalent – it only works in the Thumb state.
- Syntax: `IT`{*x*{*y*{*z*}}} *cond*
  - *cond* specifies the condition for the first instruction in the IT block
  - *x* specifies the condition switch for the second instruction in the IT block
  - *y* specifies the condition switch for the third instruction in the IT block
  - *z* specifies the condition switch for the fourth instruction in the IT block
- The structure of the IT instruction is "IF-Then-(Else)" and the syntax is a construct of the two letters T and E:
  - IT refers to If-Then (next instruction is conditional)
  - ITT refers to If-Then-Then (next 2 instructions are conditional)
  - ITE refers to If-Then-Else (next 2 instructions are conditional)
  - ITTE refers to If-Then-Then-Else (next 3 instructions are conditional)
  - ITTEE refers to If-Then-Then-Else-Else (next 4 instructions are conditional)

Conditional Execution / Voorwaardelike uitvoer

<u>Example:</u>

```
ITTE   NE             ; Next 3 instructions are conditional
ANDNE  R0, R0, R1 ; ANDNE does not update condition flags
ADDSNE R2, R2, #1 ; ADDSNE updates condition flags
MOVEQ  R2, R3         ; Conditional move
```

- The number of THEN-execute statements and ELSE-execute statements should match number of 'T's and 'E's.
- Same condition suffix as `IT{`*x*`{`*y*`{`*z*`}}}` instruction should be used for T statements, and opposite for E statements
- You **must** add the conditional mnemonic to the instructions:

```
IT  NE            ; Next instruction is conditional
ADD R0, R0, R1 ; Syntax error: no condition code used in IT block.
```

Memory / Geheue

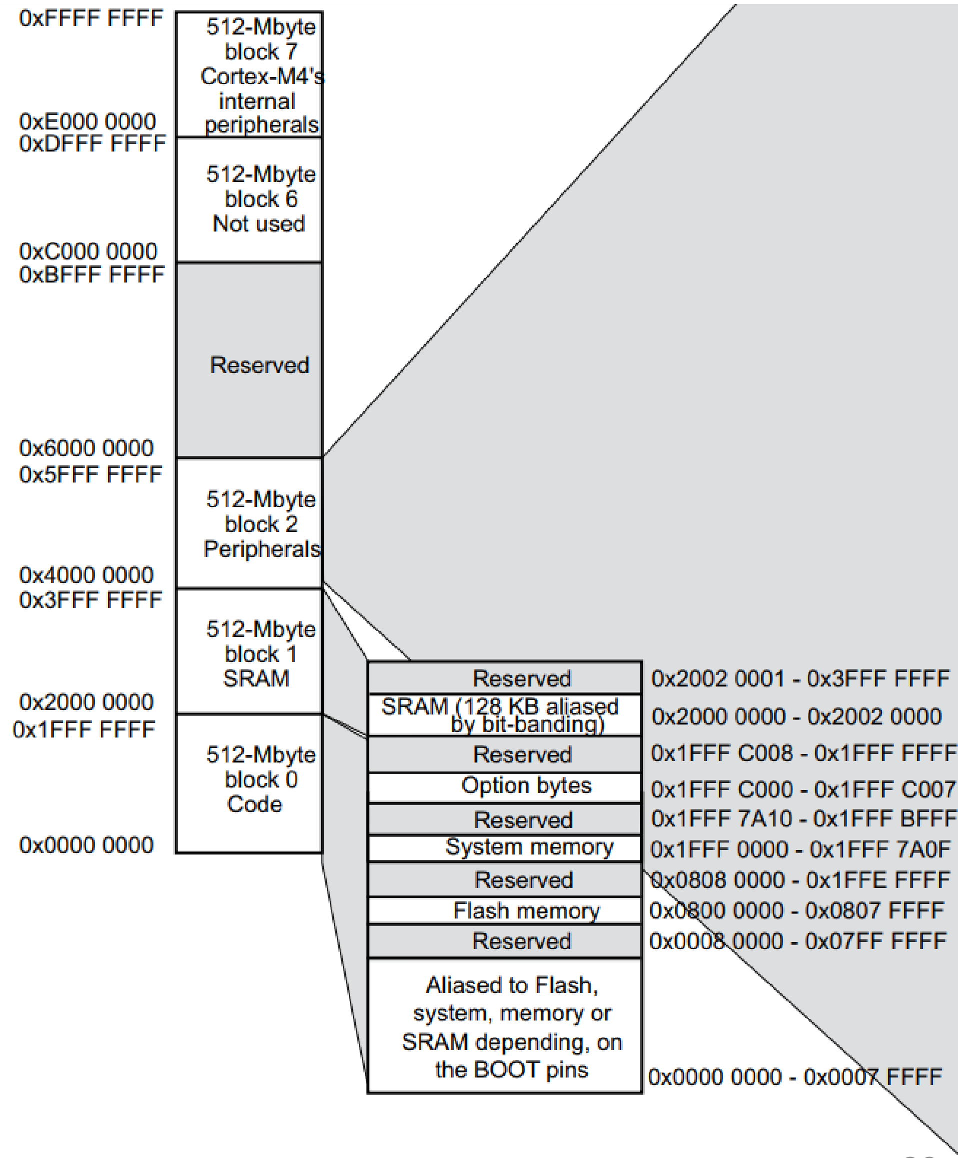
- **Memory** is contiguous storage elements that hold data, each element holding a fixed number of bits and having an address.
- Computer memory is organized in bytes, addressed in bytes, size is measured in bytes
- Different uses of memory (program vs data) and different technologies/implementations.
- Data that your program will update (variables):
  - RAM (Random Access Memory), SRAM (Static RAM)

    ⇒ Volatile - contents not preserved after reset / losing power
- For persistent data such as program code, configuration settings
  - ROM (Read-only memory)
  - EEPROM (Electrically Erasable ROM)
  - Flash memory

    ⇒ Non-volatile – contents preserved after power cycle

Memory / Geheue

- ARM address bus is 32-bits wide.
- Maximum address offset is $2^{32}-1 \approx 4$ GB
- The STM32F411VE (we'll be using for practicals) has **128kB of RAM** and **512kB of Flash** (program) memory
- In a microcontroller, all the internal memory and peripheral registers are mapped to the processor address space.
- Program that will switch on a LED
  - Output port pin mapped to memory address.
  - Write a '1' to the correct memory address, and
  - Remember, most assembly instructions involve moving data around!



Pointers (in C) / Wysers (in C)

Pointer == Address

- A pointer is a variable in C that contains an address.
- A pointer has a type associated with it

```
int* my_ptr;
```

- The type refers to the *type* and *size* of data that is stored at the address.
- Note that the pointer variable will always contain a 32-bit address regardless of what type of data it is pointing to.
- Pointers have two operators associated with it.
- **Pointer Operator 1**. The ampersand (&) operator is used to <u>obtain the address</u> of another variable, i.e.

```
int my_var
int* my_ptr = &my_var;
```

⇒ will cause `my_ptr` to contain the address of `my_var`.

Pointers (in C) / Wysers (in C)

- **Pointer Operation 2**: dereferencing (*) operator. This allows us to access the data at the address contained in the pointer.

```
*my_ptr = *my_ptr + 1;
```

  - The `*my_ptr` on the right-hand side of the above code line will cause the processor to read the data from memory at the address contained in the `my_ptr` variable.
  - It will then increment this value and write the updated value back into the same memory address.

- Use pointers for
  - Accessing (reading from and writing to) specific memory location (i.e. memory-mapped peripheral register)
  - Accessing data stored in arrays
  - Passing data/variables by reference

Pointers (in C) / Wysers (in C)

- Pointer use example 1: Accessing memory at specific location

```
uint32_t* ptr_gpioa_moder = 0x40020000;
uint32_t reg_contents = *ptr_gpioa_moder;  // read from register
*ptr_gpioa_moder =  0; // write to register.
```

- It is also possible to use the '*' operator directly on a numeric constant value, without having to use a pointer variable.

```
uint32_t reg_contents = *(0x40020000);
```

✗

  - this will almost work, but the C-compiler has to know what the size of the data at address 0x4002000 is. So rather use

```
uint32_t reg_contents = *( (uint32_t*) 0x40020000 );
```

✓

  - Have to 'cast' the constant value to a pointer of a specific type, otherwise the C-compiler will not know what size of data to read or write.
  - And then finally, we can make use of a `#define` (compiler directive) to make the code look neater.

```
#define GPIOA_MODER *((uint32_t*)0x40020000)
```

  - notice there are no spaces in `*((uint32_t*)0x40020000)`, and no semi-colon at the end of this line
  - Which will allow you to write

```
GPIOA_MODER = (GPIOA_MODER & 0xfffffff3) + 0x4;
```

Pointers – Assembly equivalent (Load and Store)

- The assembly equivalent of pointers are the load (LDR) and store (STR) instructions.

ARM Assembly Code

```
LDR r12,=0x4002000
LDR r9, [r12]
STR r9, [r12]
```

High-Level Code

```
uint32_t* ptr_gpioa_moder = 0x40020000;
uint32_t r9 = *ptr_gpioa_moder;
*ptr_gpioa_moder = r9;
```

- Note that the first LDR instruction is a pseudo-instruction that uses the literal pool to load a value.
- For pointers that refer to single byte data, use LDR**B** and STR**B**:

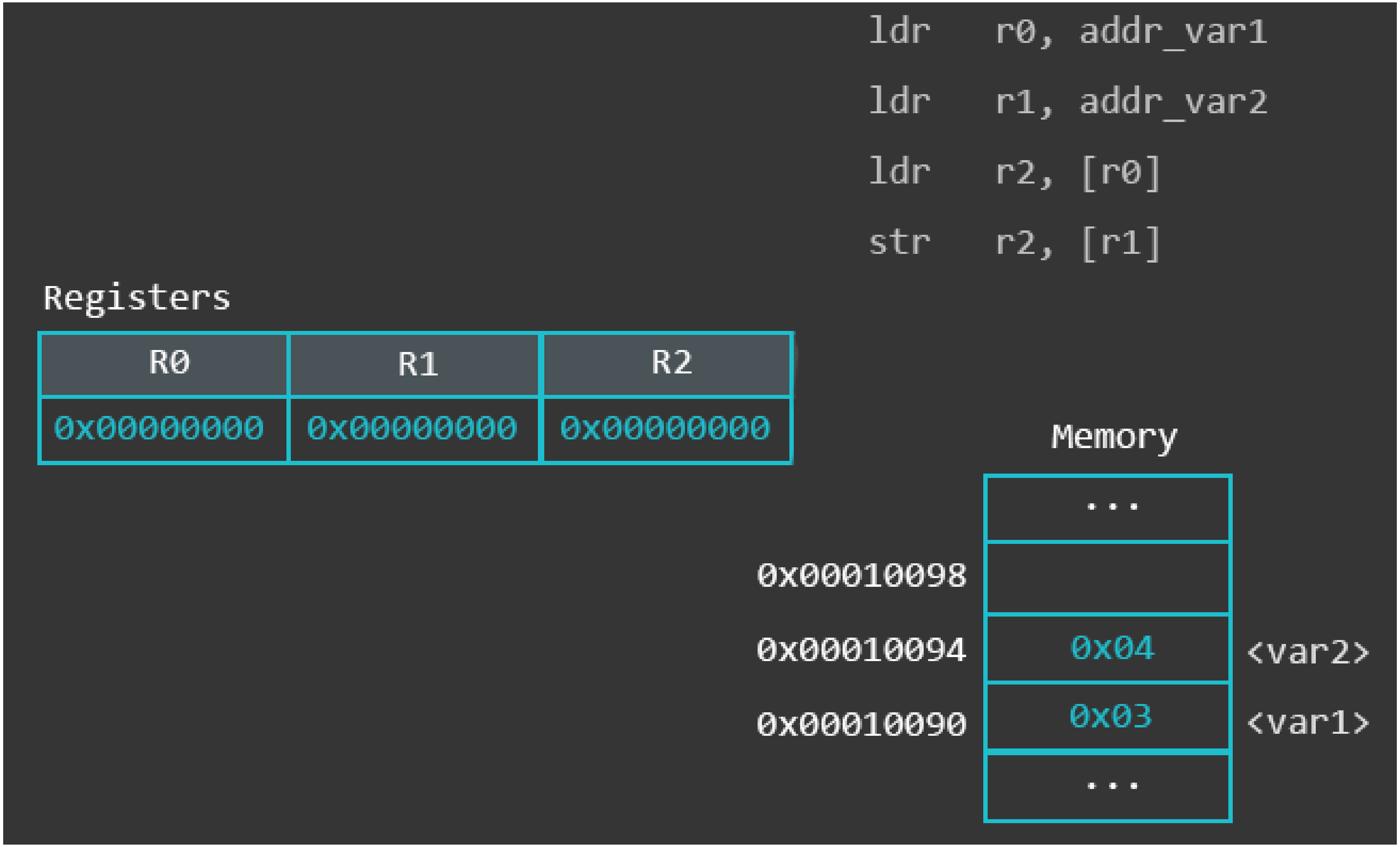
High-Level Code

```
uint8_t* byte_ptr;
*byte_ptr = *byte_ptr + 1;
```

ARM Assembly Code

```
LDRB r9, [r12]
ADD r9, r9, #1
STRB r8, [r12]
```

Practical example / Praktiese voorbeeld



https://azeria-labs.com/memory-instructions-load-and-store-part-4/

More load and store instructions / Ander laai en stoor instruksies

- To load or store data lengths other than words (32-bits) use the following instructions:

TABLE 5.2

Most Often Used Load/Store Instructions

Loads	Stores	Size and Type
LDR	STR	Word (32 bits)
LDRB	STRB	Byte (8 bits)
LDRH	STRH	Halfword (16 bits)
LDRSB		Signed byte
LDRSH		Signed halfword
LDM	STM	Multiple words

- Note the absence of store instructions for signed operands.
- LDRSB will extend the sign of the loaded value to the full 32-bits.
- No need for an equivalent 'signed' store – the interpretation of the data is up to the programmer.

LDR, LDRB and LDRSB

- Memory contents at address in r0

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	0	0	1	0	0	1	1	1	1	0	0	1	0	1	0	0	0	1	0	1	1	1	1	0	0	0	0
8				9				3				C				A				2				F				0			

- LDR loads 32-bits from memory (1 word)
- LDRB loads only 8 bits (1 byte)
  - Therefore after executing: `LDRB r1, [r0]`

Contents of r1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0
0				0				0				0				0				0				F				0			

- $F0_{16}$ is 240_{10}, if the data is to be interpreted as an *unsigned* integer

LDR, LDRB and LDRSB

- What if the data in memory should be interpreted as signed data (in 8-bits)
- $F0_{16} = 11110000_2 = -16_{10}$, if *signed* 8-bit representation is used.

- If you use LDRB, the number in the 32-bit register is still 240, not -16!

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0
0				0				0				0				0				0				F				0			

- LDRSB will fill the rest of the register with '1's if it is a negative number, or in other words the 8-bit number will be **sign extended** to fill all 32 bits.

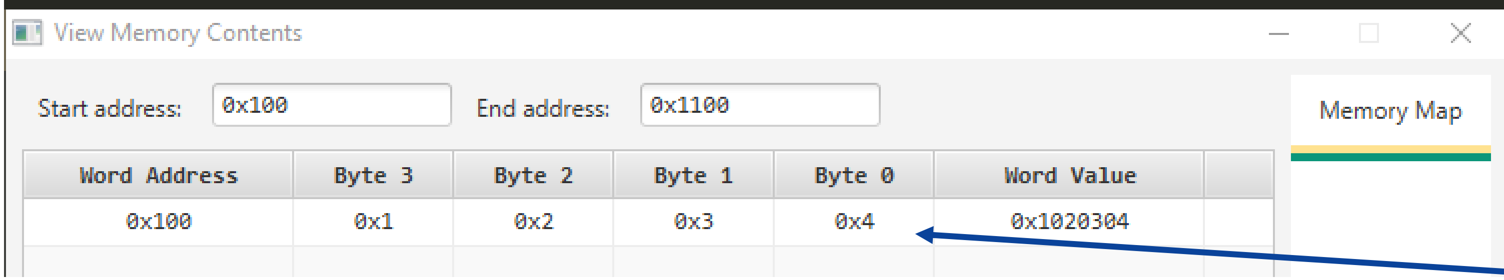
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0
F				F				F				F				F				F				F				0			

- Now, in 32-bit signed representation, r1 contains -16

Endianness

Endianness: the memory address order in which bytes are loaded/stored within a word.

```
            LDR       r1, =0x01020304
somebytes   FILL      4     ; zero-fills the given number of bytes
            LDR       r2, =somebytes
            STR       r1, [r2]
```



Word Address	Byte 3	Byte 2	Byte 1	Byte 0	Word Value
0x100	0x1	0x2	0x3	0x4	0x1020304

Little endian: Least significant byte goes to lowest address.
Big endian: Most significant byte goes to lowest address

Be aware of the endianness of the system you are working with!

Pseudo-instructions / Pseudo-instruksies

```
data DCD 123
     LDR r1, =data
     ADR r2, data
```

```
uint32_t* ptr_data = &data;
```

- In the above code both the LDR and ADR instructions achieve the same result as the equivalent C-code.
- The are known as pseudo instructions since they cannot directly be translated to machine code. They are instead replaced with a different instruction by the assembler.

`LDR r1, =data ⇒ LDR r1, [pc, #offset]`

- Data is stored in the literal pool
- Offset must be in range ±(0–4095)

`ADR r2, data ⇒ ADD r2, pc, #offset`

- ADR replaced by with a single ADD or SUB instruction that loads the address, if it is in range.
- Offset limited by 8-bit immediate and 4-bit rotation.

Pointers (in C) / Wysers (in C)

- Pointer use example 2: Accessing data stored in arrays

```c
int array[5] = {0, 1, 2, 3, 4};
int* testptr = array;
printf("address of array = %p, value of testptr = %p", array, testptr);
```

⇒ `address of array = 6356728, value of testptr = 6356728`

- The variable for the array can be used as a pointer (to the first element in the array)

```c
array[3] = 2;
testptr[3] = 2;   // same thing!

testptr = &array[4];   // testptr has the address of the 4th element of array
testptr = array + sizeof(int) * 4; // same thing!
```

Pointers (in C) / Wysers (in C)

- Pointer use example 3: Passing function arguments by reference

```c
void a_function(int* ptr_val)
{
    *ptr_val = *ptr_val + 1;
}

int main()
{
    int aval = 5;
    a_function(&a_val);
}
```

⇒ `a_val` now contains the value 6

- Functions cannot modify the arguments that are passed to it. But you can pass and address, and let the function modify the data at that address

Pointers (in C) / Wysers (in C)

Pointer arithmetic

- You can add and subtract from pointer variables, to point to a different address

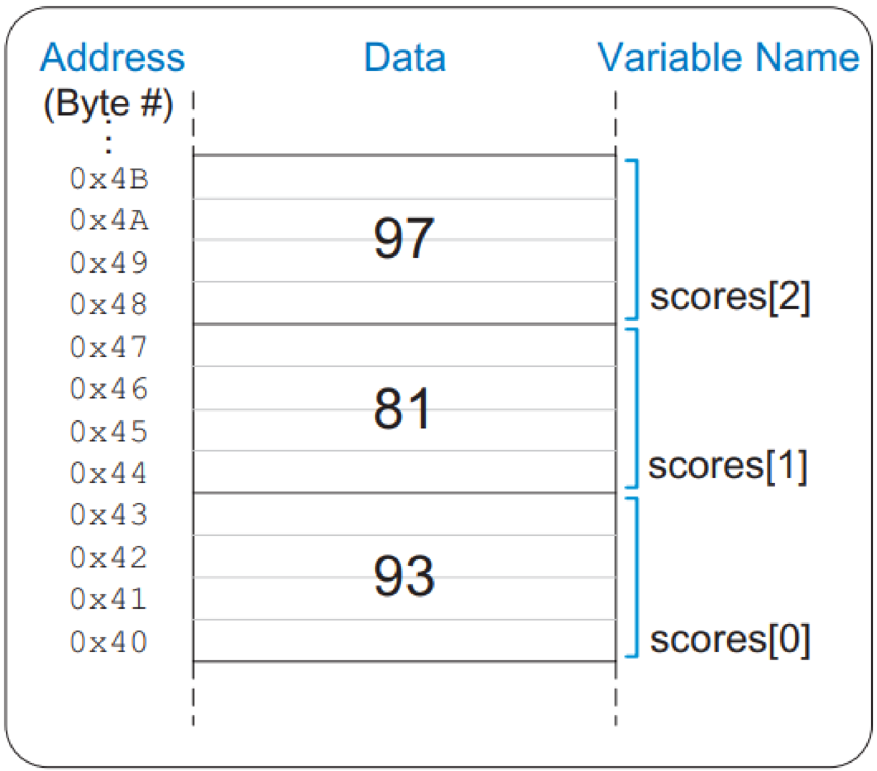
```
int* ptr_i;
*ptr_i = *ptr_i + 1; // modify data that is pointed to. Pointer remains unchanged
ptr_i = ptr_i + 1; // modify the pointer to point to the following int
```

- Adding to the pointer will increase the pointer value (the address) by the size of the data being pointed to (in this case 4 bytes).
- Typically used when accessing arrays.

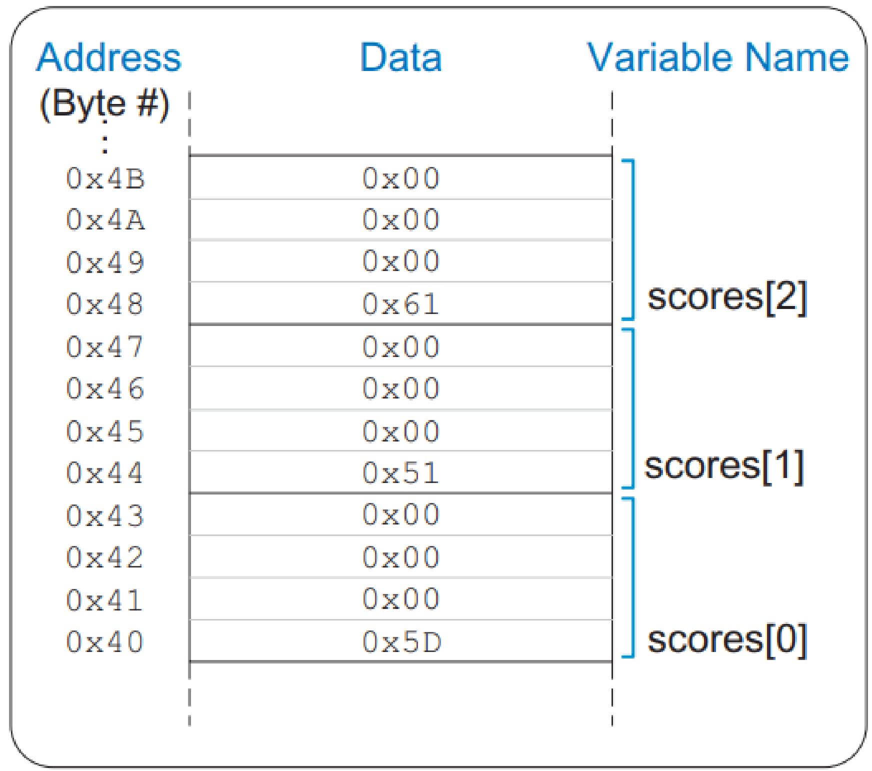
Arrays / Skikkings

- An **array** is a series of same-sized elements, stored contiguously (next-to-each-other) in memory.

```
int scores[3] = {93, 81, 97}; // scores[0]=93; scores[1]=81; scores[2]=97;
```



Memory



Memory

Arrays / Skikkings

- Array variables are essentially pointers

```
scores[0] = 0;
*scores = 0; // same thing

scores[3] = 0;
*(scores + 3) = 0;  // same thing! (why +3, and not +12?)
```

- Array size is declared at compile time

```
// compiler reserves space for 4x ints (=16 bytes total) (and also for the pointer variable)
int scores[4] = {0};
// compiler reserves space for just the pointer variable, not an array
int* ptr_scores;
```

- There is however a slight difference between an array name, which is **not** a variable, and pointers which **are** variables. Therefore:

```
int data[4] = {0};
int* dataPtr = data;

data++;      // this is illegal
dataPtr++;   // this is legal
```

Arrays / Skikkings

- Arrays are always passed by reference (when calling functions)

```
float getMean(int arr[], int len) { … }
int data[4] = {78, 14, 99, 27};
float avg = getMean(data, 4);
```

- All of these function declarations are the same

```
float getMean(int *vals, int len);
float getMean(int vals[], int len);
float getMean(int vals[100], int len);
```

Structs (in C) / Strukture (in C)

```c
struct contact
{
    char name[30];
    int phone;
    float height; // in meters
};
struct contact c1;

strcpy(c1.name, "Ben Bitdiddle");
c1.phone = 7226993;
c1.height = 1.82;
```

```c
typedef struct contact
{
    char name[30];
    int phone;
    float height; // in meters
} Contact;

Contact c1;
```

Structs (in C) / Strukture (in C)

Array of structs:

```
struct contact classlist[200];
classlist[0].phone = 9642025;
```

Pointer to a struct:

```
struct contact *cptr;
cptr = &classlist[42];
cptr->height = 1.9;  // equivalent to: (*cptr).height = 1.9;
```

Use the member access operator -> to dereference a pointer to a structure and access a member of the structure

Load or Store with scaled offset / Laai of stoor met geskaleerde afset

```
LDR r9, [r12, r8, LSL #3]
```

Why would you use such an operation? Think about array access

- The lecture room card readers output a string of 8 characters (your student number) and the microcontroller saves all the scanned cards in a buffer (buffer allows for 200 card scan events to be stored)

```
#define CARDNUM_LEN 8
#define CARDLOG_BUFFER_LEN 200
uint8_t log_buffer[CARDLOG_BUFFER_LEN * CARDNUM_LEN];
```

- The byte offset of `i`th entry in the buffer is:

  `i*CARDNUM_LEN = i * 8`, or `i << 3`

- If the buffer, `log_buffer`, is stored from address x in memory, the absolute memory address of the scan entry `i` is:

  x + `(i << 3)`

- In the example on top, r12 is the offset of `log_buffer` (x) and r8 is `i`.

ARM indexing modes

- In each case, the base register is R1 and the offset is R2.
- The offset can be subtracted by writing –R2.
- The offset may also be an immediate in the range of 0–4095 (12-bits) that can be added (e.g., #20) or subtracted (e.g., #−20).

Table 6.4 ARM indexing modes

Mode	ARM Assembly	Address	Base Register
Offset	`LDR R0, [R1, R2]`	R1 + R2	Unchanged
Pre-index	`LDR R0, [R1, R2]!`	R1 + R2	R1 = R1 + R2
Post-index	`LDR R0, [R1], R2`	R1	R1 = R1 + R2

ARM indexing modes (2)

Table 6.4 **ARM indexing modes**

Mode	ARM Assembly	Address	Base Register
Offset	`LDR R0, [R1, R2]`	R1 + R2	Unchanged
Pre-index	`LDR R0, [R1, R2]!`	R1 + R2	R1 = R1 + R2
Post-index	`LDR R0, [R1], R2`	R1	R1 = R1 + R2

- **Offset** addressing calculates the address as the base register ± the offset; the base register is unchanged.
- **Pre-indexed** addressing calculates the address as the base register ± the offset and updates the base register to this new address.
- **Post-indexed** addressing calculates the address as only the base register only and then, after accessing memory, the base register is updated to the base register ± the offset.

Example / Voorbeeld - strcpy

```
char* strcpy ( char* destination, const char* source );
```

- Copies the C string pointed by source into the array pointed by destination, including the terminating null character (and stopping at that point).
- One possible implementation in C.

```c
// Copy the source string, src, to the destination string, dst
void strcpy(char *dst, const char *src)
{
    int i = 0;
    do
    {
        dst[i] = src[i]; // copy characters one byte at a time
    } while (src[i++]); // until the null terminator is found
}
```

Example / Voorbeeld - strcpy

- Possible implementation in ARM assembly language (without using a function)

```
destptr    EQU       0x0120 ;define constant - similar to #define in C
srcstr     DCB       1,2,3,4,5,6,7,8,9,0
ENTRY ; mark the first instruction
Main       ADR       r1, srcstr ; pointer to the first string
           LDR       r0, =destptr ; pointer to the second string
strcopy
           LDRB      r2, [r1], #1 ; load byte, update address
           STRB      r2, [r0], #1 ; store byte, update address
           CMP       r2, #0 ; check for zero terminator
           BNE       strcopy ; keep going if not
```

View Memory Contents

Start address: 0x100   End address: 0x1100

Word Address	Byte 3	Byte 2	Byte 1	Byte 0	Word Value
0x100	0x4	0x3	0x2	0x1	0x4030201
0x104	0x8	0x7	0x6	0x5	0x8070605
0x108	0x0	0x0	0x0	0x9	0x9
0x120	0x4	0x3	0x2	0x1	0x4030201
0x124	0x8	0x7	0x6	0x5	0x8070605
0x128	0x0	0x0	0x0	0x9	0x9