ARM

ASSEMBLY PROGRAMMING

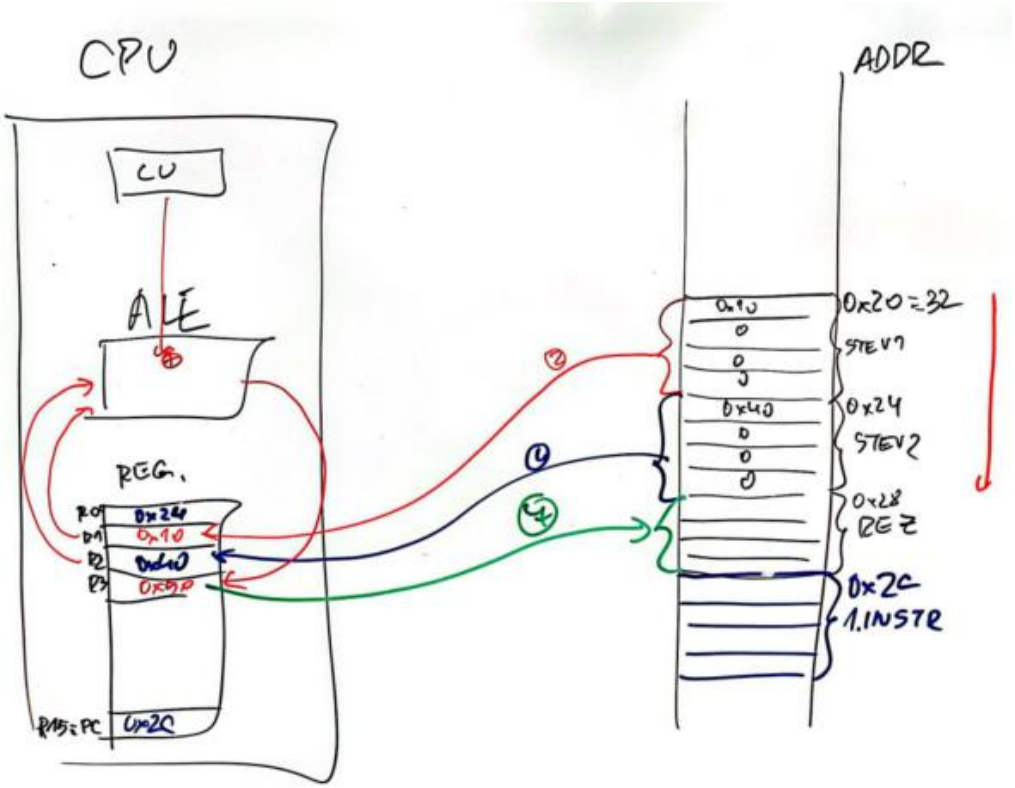
1. part

Intro LAB: Assembly programing

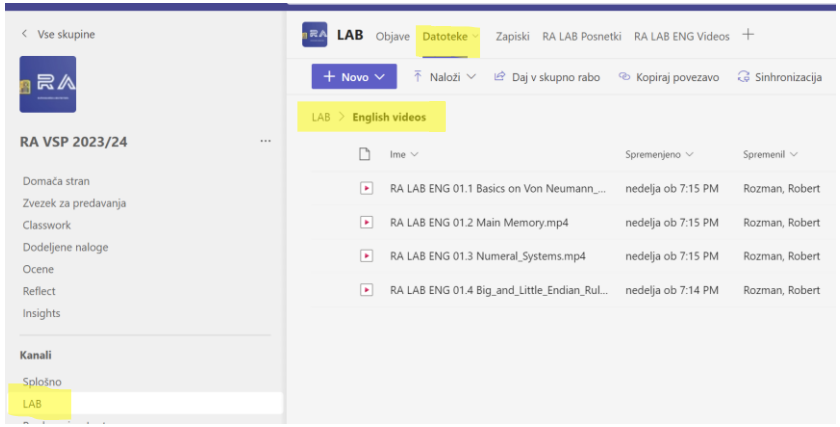
An example of adding two numbers:
rez: = stev1 + stev2

Assembly language	Instruction description	Machine language
adr r0, stev1	R0 ← Addr. of stev1	0xE24F0014
ldr r1, [r0]	R1 ← M[R0]	0xE5901000
adr r0, stev2	R0 ← Addr. of stev2	0xE24F0018
ldr r2, [r0]	R2 ← M[R0]	0xE5902000
add r3, r2, r1	R3 ← R1 + R2	0xE0823001
adr r0, rez	R0 ← Addr. of rez	0xE24F0020
str r3, [r0]	M[R0] ← R3	0xE5803000

Assembler (compiler)



MS Teams RA Team English videos



Intro LAB: Assembly programing

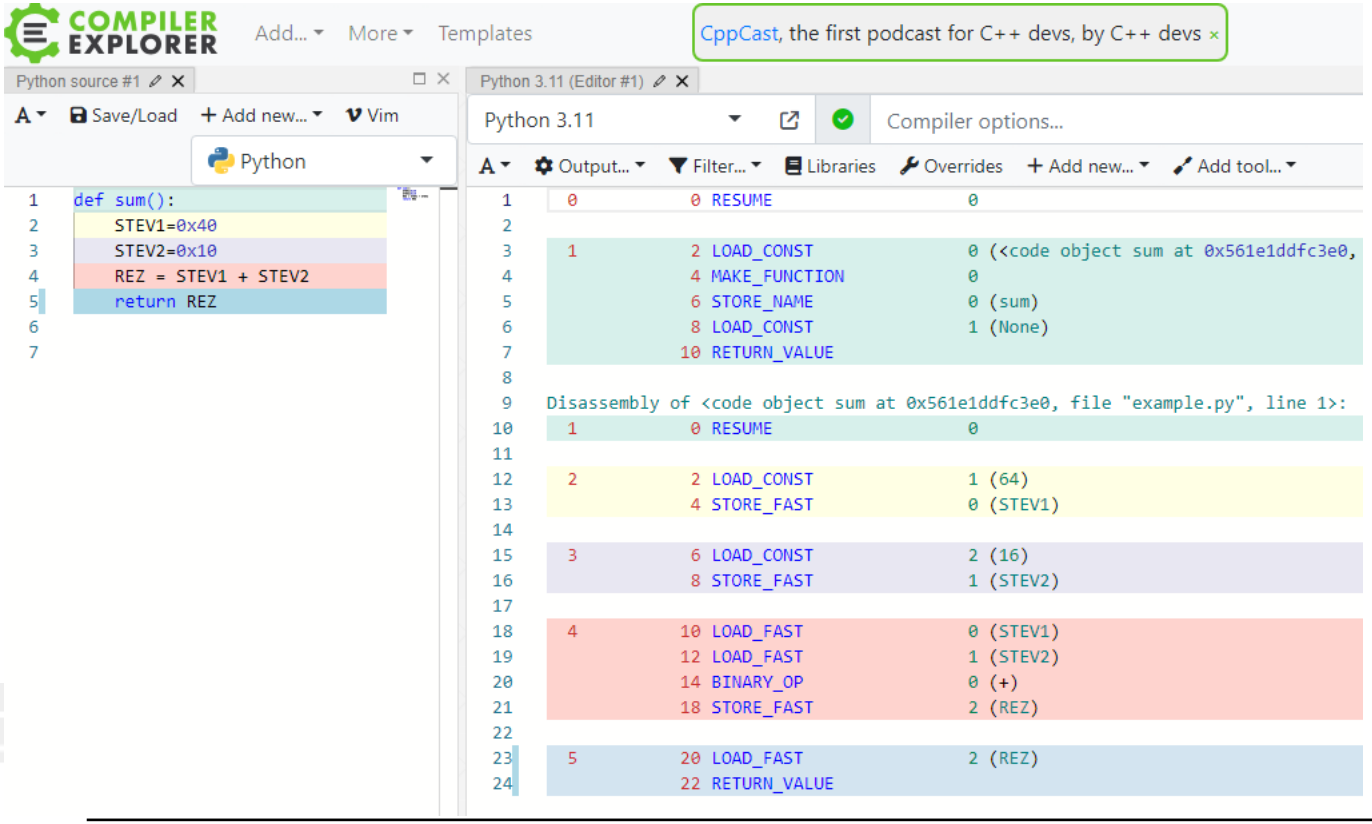
Assembly language	Instruction description	Machine language
adr r0, stev1	R0 ← Addr. of stev1	0xE24F0014
ldr r1, [r0]	R1 ← M[R0]	0xE5901000
adr r0, stev2	R0 ← Addr. of stev2	0xE24F0018
ldr r2, [r0]	R2 ← M[R0]	0xE5902000
add r3, r2, r1	R3 ← R1 + R2	0xE0823001
adr r0, rez	R0 ← Addr. of rez	0xE24F0020
str r3, [r0]	M[R0] ← R3	0xE5803000

Assembler (compiler)

Case: Sum of two numbers in Python:

rez := stev1 + stev2

Example of Python code partially compiled to byte-code



https://godbolt.org/

```
def sum():
    STEV1=0x40
    STEV2=0x10
    REZ = STEV1 + STEV2
    return REZ
```

Case Sum

Intro LAB: Assembly programing

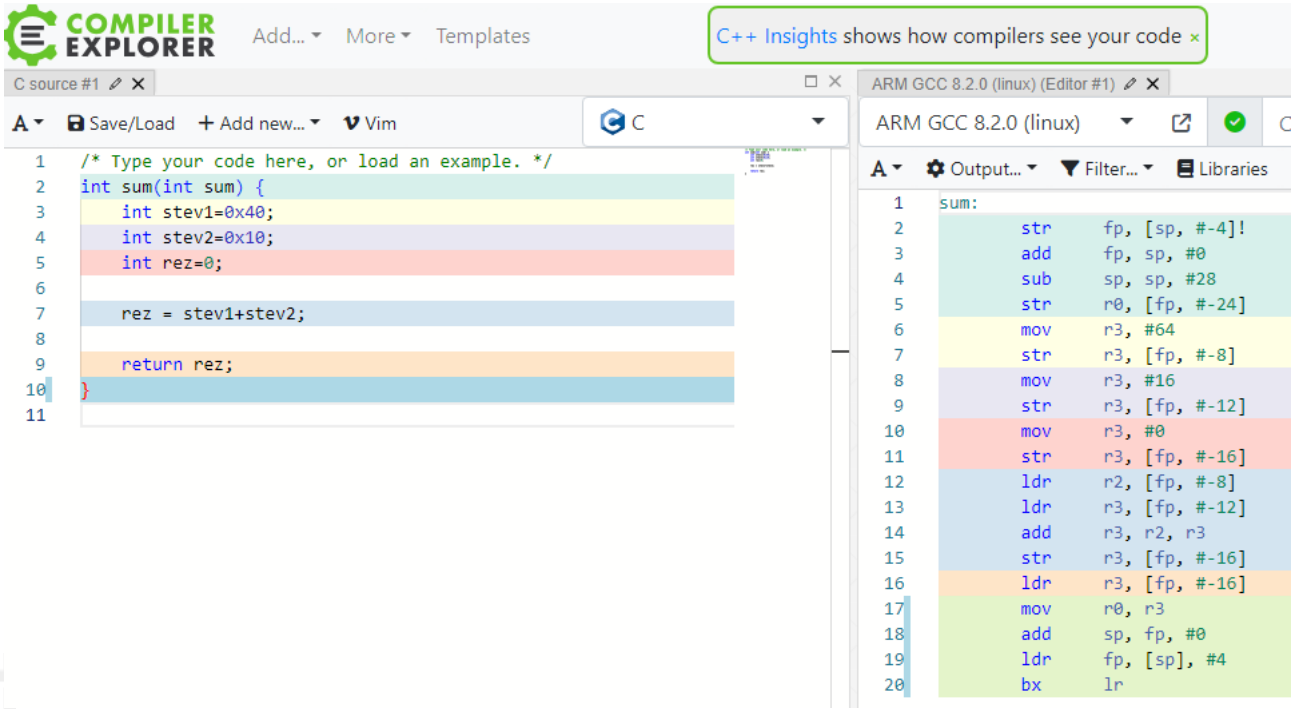
Assembly language	Instruction description	Machine language
adr r0, stev1	R0 ← Addr. of stev1	0xE24F0014
ldr r1, [r0]	R1 ← M[R0]	0xE5901000
adr r0, stev2	R0 ← Addr. of stev2	0xE24F0018
ldr r2, [r0]	R2 ← M[R0]	0xE5902000
add r3, r2, r1	R3 ← R1 + R2	0xE0823001
adr r0, rez	R0 ← Addr. of rez	0xE24F0020
str r3, [r0]	M[R0] ← R3	0xE5803000

Assembler (compiler)

Case: Sum of two numbers in C:

rez := stev1 + stev2

Example of C-code compiled to ARM Assembler



https://godbolt.org/

```
int sum(int sum) {
    int stev1=0x40;
    int stev2=0x10;
    int rez=0;

    rez = stev1+stev2;

    return rez;
}
```

Case Sum

ARM (Advanced RISC Machine) = RISC?

32-bit ISA (Instruction Set Architecture) :

+ load/store architecture

+ pipeline

+ reduced instruction set, all instructions are 32-bit

+ orthogonal registers – all 32-bit

- many addressing modes

- many instruction formats

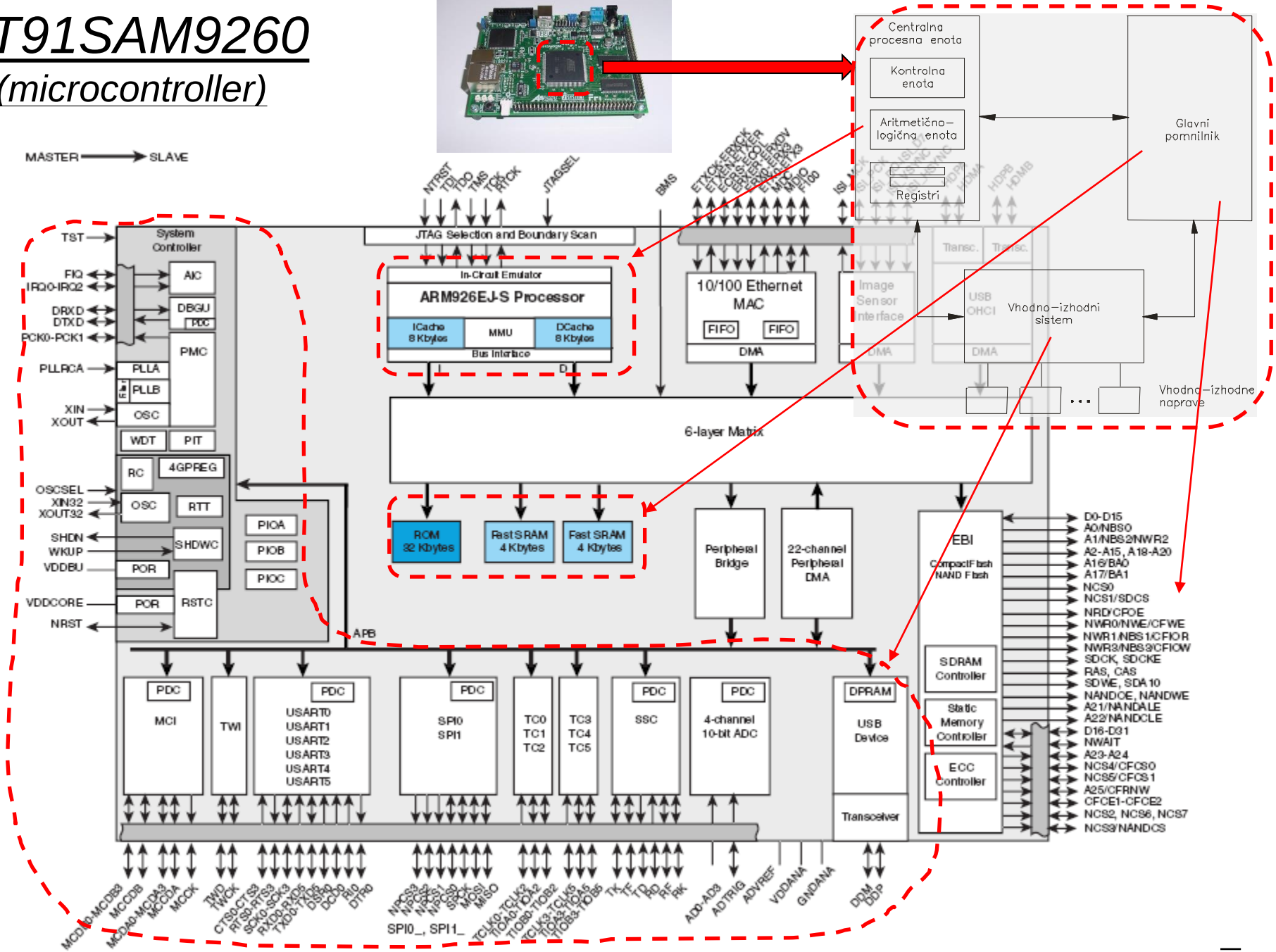
- some instructions take multiple clock cycles to execute (eg. *load/store multiple*) – but they make programmes shorter

- additional 16-bit instruction set „Thumb“ – shorter programmes

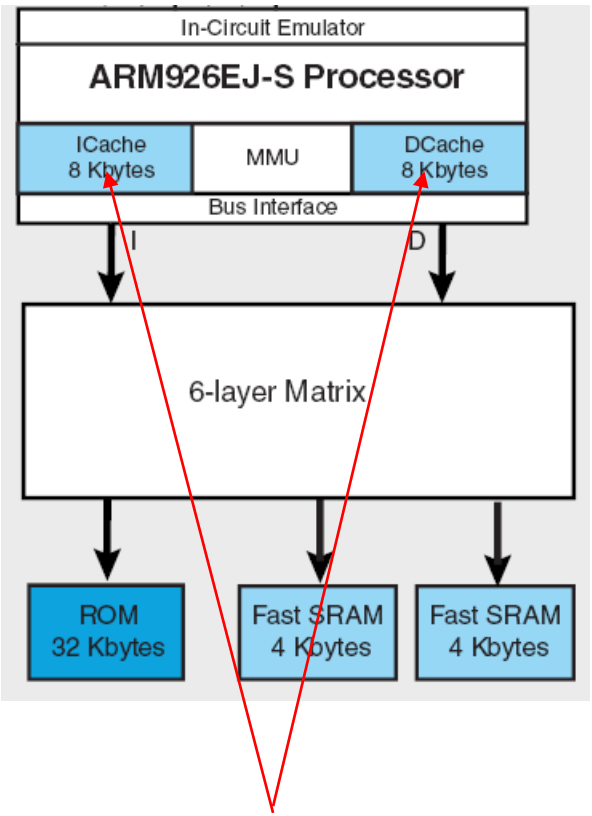
- conditional instruction execution – execute only if condition is true

AT91SAM9260

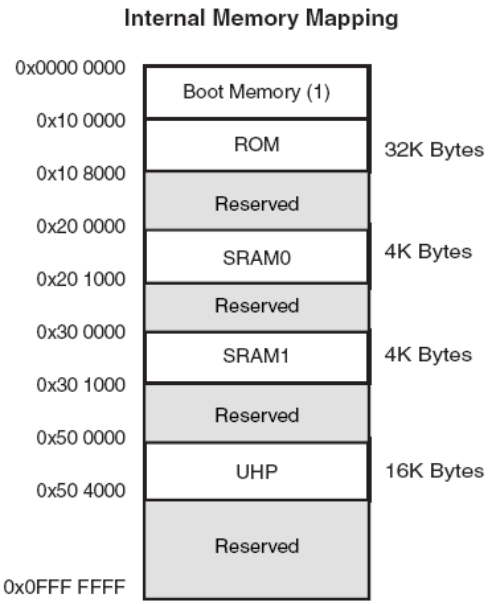
(microcontroller)



AT91SAM9260



Memory map



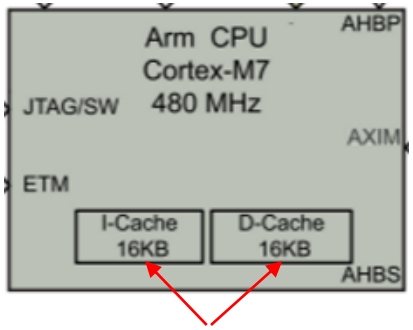
Harvard Architecture

On Cache level

Princeton architeture

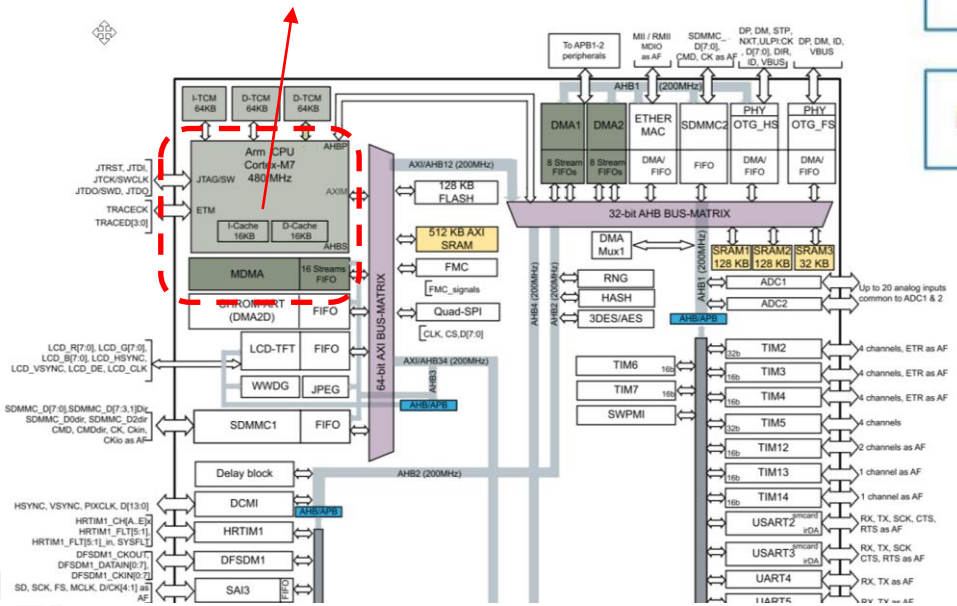
Main memory

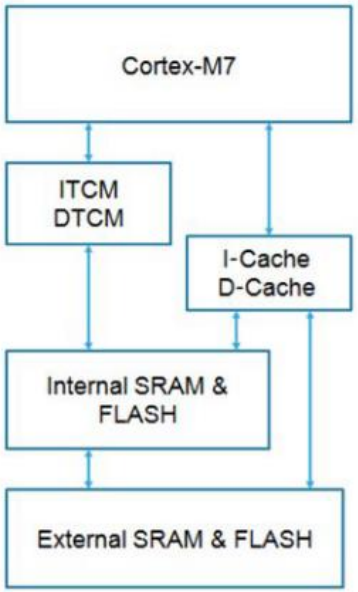
STM32H750XB



Harvard Architecture

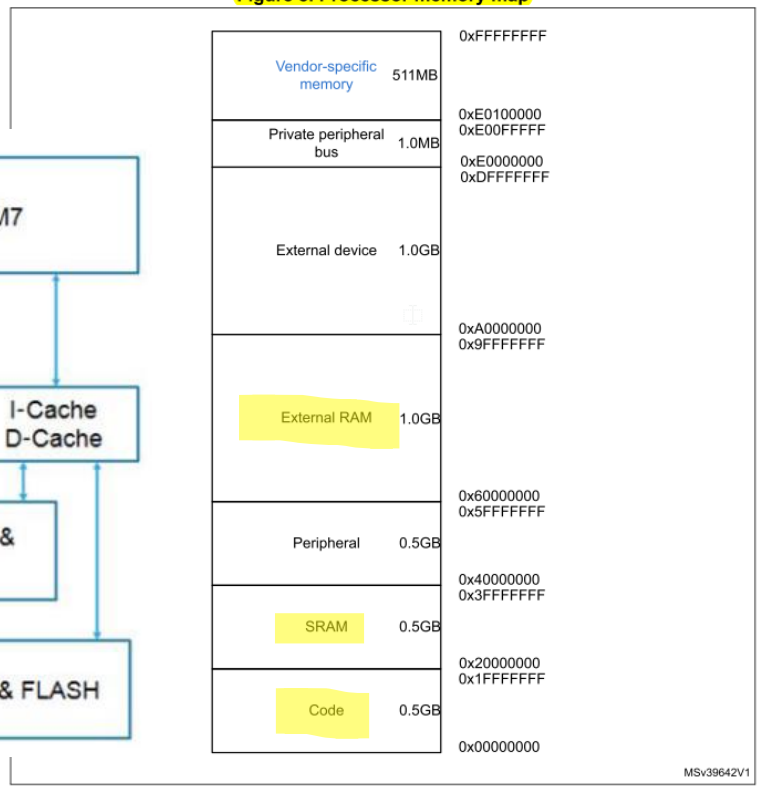
On Cache level





Memory map

Figure 8. Processor memory map



Princeton architeture

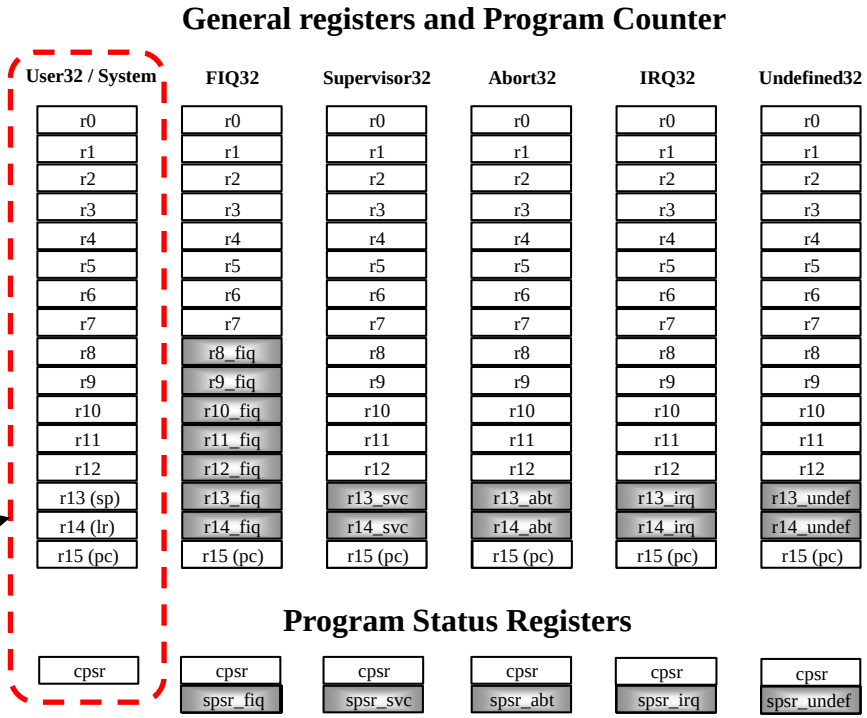
Main memory

```
MEMORY
{
 FLASH (rx)    : ORIGIN = 0x08000000, LENGTH = 128K
 DTCMRAM (xrw) : ORIGIN = 0x20000000, LENGTH = 128K
 RAM_D1 (xrw)  : ORIGIN = 0x24000000, LENGTH = 512K
 RAM_D2 (xrw)  : ORIGIN = 0x30000000, LENGTH = 288K
 RAM_D3 (xrw)  : ORIGIN = 0x38000000, LENGTH = 64K
 ITCMRAM (xrw) : ORIGIN = 0x00000000, LENGTH = 64K
}
```

ARM programming model

- **Consists of :**
  - 16 general purpose registers
  - Status register CPSR (Current Program Status Register)

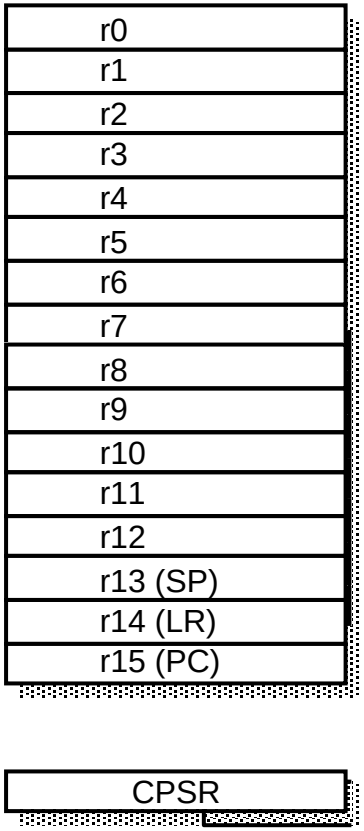
- **CPU supports multiple** *operation modes*, **each has its own set of registers – overall 36 registers for all modes**



- **Only few are visible in certain processor‘s operation mode**

- **Operation modes can be divided into two groups:**
  - Privileged (Read/Write access to CPSR)
  - Non-Privileged or User Mode (Read access to CPSR)

Programming model – user mode



User mode:

- Only non-privileged mode
- For execution of user programmes

Visible 17 32-bit registers:

r0 – r15 and CPSR

Visible registers:

- r0-r12: general purpose (orthogonal) registers
- r13(sp): *Stack Pointer*
- r14(lr): *Link Register*
- r15(pc): *Program Counter*

- CPSR: status register *(Current Program Status Register)*

More details later…

Status register – CPSR

CPSR - Current Program Status Register

- flags (N,Z,V,C)
- interrupt mask bits (I, F)
- bit T determines instruction set:
  T=0 : ARM architecture, 32-bit ARM instruction set
  T=1: Thumb architecture, 16-bit Thumb instruction set
- lowest 5 bits determine processor mode
- in user mode only read access to CPSR; instructions are allowed only to change state of flags.

Flags can be changed according to result of ALU operation:

N = 1: bit 31 of result is 1 (Negative),	**N** = 0: bit 31 of result is 0	(*Negative*)
Z = 1: result is 0,	**Z** = 0: result is not 0 (non-zero)	(*Zero*)
C = 1: carry,	**C** = 0: no carry	(*Carry*)
V = 1: overflow,	**V** = 0: no overflow	(*oVerflow*)

Assembly programming

Assembly language	Instruction description	Machine language
adr r0, stev1	R0 ← Addr. of stev1	0xE24F0014
ldr r1, [r0]	R1 ← M[R0]	0xE5901000
adr r0, stev2	R0 ← Addr. of stev2	0xE24F0018
ldr r2, [r0]	R2 ← M[R0]	0xE5902000
add r3, r2, r1	R3 ← R1 + R3	0xE0823001
adr r0, rez	R0 ← Addr. of rez	0xE24F0020
str r3, [r0]	M[R0] ← R3	0xE5803000

Assembler (compiler)

- **Assembly language:**
  - Instructions (mnemonics),
  - registers
  - addresses
  - constants

- **You don‘t have to:**
  - Know machine instructions and their composition
  - Calculate with addresses

Assembly language compiler (*assembler*) :

- Compiles symbolic names (mnemonics) for instructions into coresponding machine instructions,
- Calculates addresses for symbolic labels and
- Creates memory image of whole program (data and code)

- **Program in machine language is not transferable:**
  - Executes only on same processor and system

- **Assembler (*assembly language*) is „low-level“ programming language**

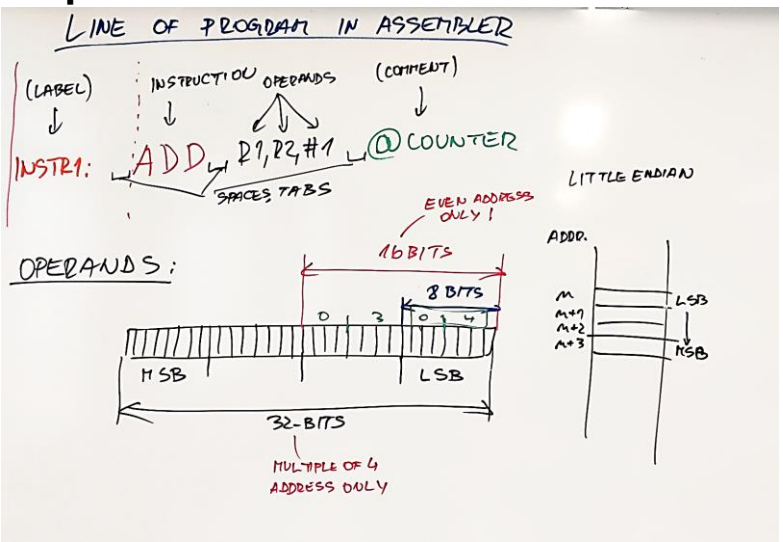
Assembly programming

ARMv4T Partial Instruction Set Summary

- **List of instructions**
  - On Moodle platform

Operation		Syntax
Move	Move	**mov**{*cond*}{**s**} R*d*, *shift_op*
	with NOT	**mvn**{*cond*}{**s**} R*d*, *shift_op*
	CPSR to register	**mrs**{*cond*} R*d*, **cpsr**
	SPSR to register	**mrs**{*cond*} R*d*, **spsr**
	register to CPSR	**msr**{*cond*} **cpsr**_*fields*, R*m*
	register to SPSR	**msr**{*cond*} **spsr**_*fields*, R*m*
	immediate to CPSR	**msr**{*cond*} **cpsr**_*fields*, #*imm8r*
	immediate to SPSR	**msr**{*cond*} **spsr**_*fields*, #*imm8r*
Arithmetic	Add	**add**{*cond*}{**s**} R*d*, R*n*, *shift_op*
	with carry	**adc**{*cond*}{**s**} R*d*, R*n*, *shift_op*
	Subtract	**sub**{*cond*}{**s**} R*d*, R*n*, *shift_op*
	with carry	**sbc**{*cond*}{**s**} R*d*, R*n*, *shift_op*
	reverse subtract	**rsb**{*cond*}{**s**} R*d*, R*n*, *shift_op*
	reverse subtract with carry	**rsc**{*cond*}{**s**} R*d*, R*n*, *shift_op*
	Multiply	**mul**{*cond*}{**s**} R*d*, R*m*, R*s*
	with accumulate	**mla**{*cond*}{**s**} R*d*, R*m*, R*s*, R*n*
	unsigned long	**umull**{*cond*}{**s**} R*dLo*, R*dHi*, R*m*, R*s*
	unsigned long with accumulate	**umlal**{*cond*}{**s**} R*dLo*, R*dHi*, R*m*, R*s*
	signed long	**smull**{*cond*}{**s**} R*dLo*, R*dHi*, R*m*, R*s*
	signed long with accumulate	**smlal**{*cond*}{**s**} R*dLo*, R*dHi*, R*m*, R*s*

- **Hand-written sheet of A4 – example of table notes**



Instructions

- **All instructions are 32-bit**

```
add r3, r2, r1 ⟹ 0xE0823001=0b1110…0001
```

- **Results are 32 bits (except multiplication)**

```
R1 + R2        ⟹ R3
```

- **Arithmetic-Logic instructions have 3 operands**

```
add r3,r3,#1
```

- **Load/store architecture (computing model)**

```
ldr r1, stev1       @ read in register
ldr r2, stev2       @ read in register
add r3, r2, r1      @ sum to register
str r3, res         @ write from register
```

Assembly programming

- **Each line usually represents one instruction in machine language**

- **Line consists of 4 columns:**

```
label: instruction operands @ comment

routine1:  add r3,r3,#1        @ increase counter
           ldr r5,[r0]
```

- **Columns are separated by tabulators, spacings are also allowed**

Operands

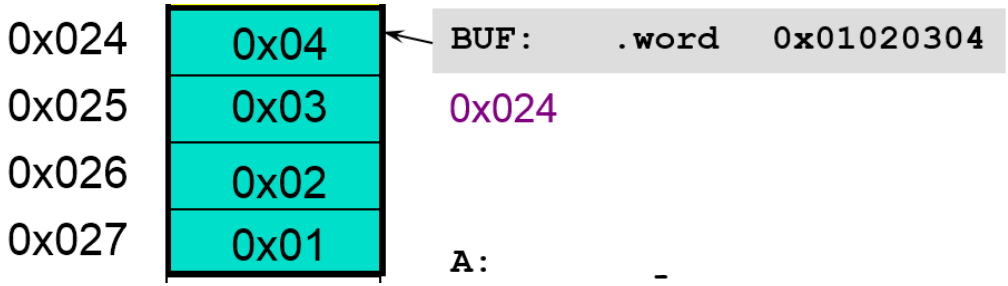
- **can be of 8, 16, 32-bit in length, signed or unsigned in memory**

- **obligatory alignment (16 or 32 bit instructions and variables)**
  - 16-bit operands on even addresses
  - 32-bit operands on multiple of 4 addresses

- **CPU executes operations in 32 bits (operands are expanded)**

```
0xFF          ⟹    0x000000FF
```

- **Rule for longer operands :„Little Endian“**



Labels

Labels are meant as a symbolic name of:

- Memory locations or
- Program lines

Labesl are commonly used in two cases:

- naming **memory locations – „variables“**

```
STEV1:      .word   0x12345678
STEV2:      .byte   1,2,3,4
REZ:        .space  4
```

```
1          .text
2          .org 0x20
3
4  STEV1:  .word   0x10
5  STEV2:  .word   0x40
6  REZ:    .word   0
7
8          .align
9          .global _start
10 _start:
11
12             adr   r0,STEV1
13             ldr   r1,[r0]
14
15             adr   r0,STEV2
16             ldr   r2,[r0]
17
18             add   r3,r1,r2
19
20             adr   r0,REZ
21             str   r3,[r0]
22
23 end:        b     end
```

- Naming of **program lines** that are **branch (jump) targets**

```
            mov r4,#10
LOOP:       subs r4, r4, #1
            ...
            bne LOOP
```

Pseudo instructions and directives – instructions for assembler (compiler)

Pseudo-instructions:

- CPU doesn‘t know them, they are for assembler
- Are translated by compiler to real instructions

```
Example:
adr r0, stev1  compiler replaces with e.g. sub r0, pc, #2c
               (ALU instruction that puts real address into r0)
```

Directives (denoted by a dot in front of them) are used for:

- memory segments (starting point) `.text .data`
- memory address for compilation `.org`
- content alignment (16/32bits) `.align`
- memory reservation for „variables“ `.space`
- memory Initialization for „variables“ `.(h)word, .byte,…`
- end of compilation `.end`

Both are not present in final memory image !

Memory segments

Directives for definition of memory segments are:

```
.data
.text
```

With those we can determine segments in memory with data and instructions.

In our case, we will use the same segment for data and instructions and use only

```
.text
```

and start address 0x20

```
.org 0x20
```

```
.text
.org 0x20
@spremenljivke

.align
.global _start
_start:
@program

end: b end
```

Memory reservation for „variables“

We have to reserve coresponding space for „variables“.

```
                .text
                .align   @ alignment !
RADIUS:         .space 4 @ reserve 4 bytes for RADIUS
```

Align address (to multiple of 4)

Potrebujemo 4 bajte

label – name of „variable“

```
                .align @ instructions must be alignem!
                ldr r7, RADIUS @ load from RADIUS to reg7
```

Assembler will replace ‘RADIUS’ with actual address of location („variable“)

Reservation of segment in memory

Labels allow better memory management:

– we give names (labels) to memory segments and don‘t use addresses (clarity of program)

```
BUFFER:        .space 40     @reserve 40 bytes
BUFFER2:       .space 10     @reserve 10 bytes
BUFFER3:       .space 20     @reserve 20 bytes

;alignment? If you‘re accessing bytes-no problem,
otherwise alignment has to be obeyed (.align)
```

- **label BUFFER coresponds to address of the first byte in a row of 40B.**

- **label BUFFER2 coresponds to address of the first byte in a row of 10B. It‘s value is 40 more than BUFFER**

- **label BUFFER3 coresponds to address of the first byte in a row of 20B. It‘s value is 10 more than BUFFER2**

Reservation with the initialization of values

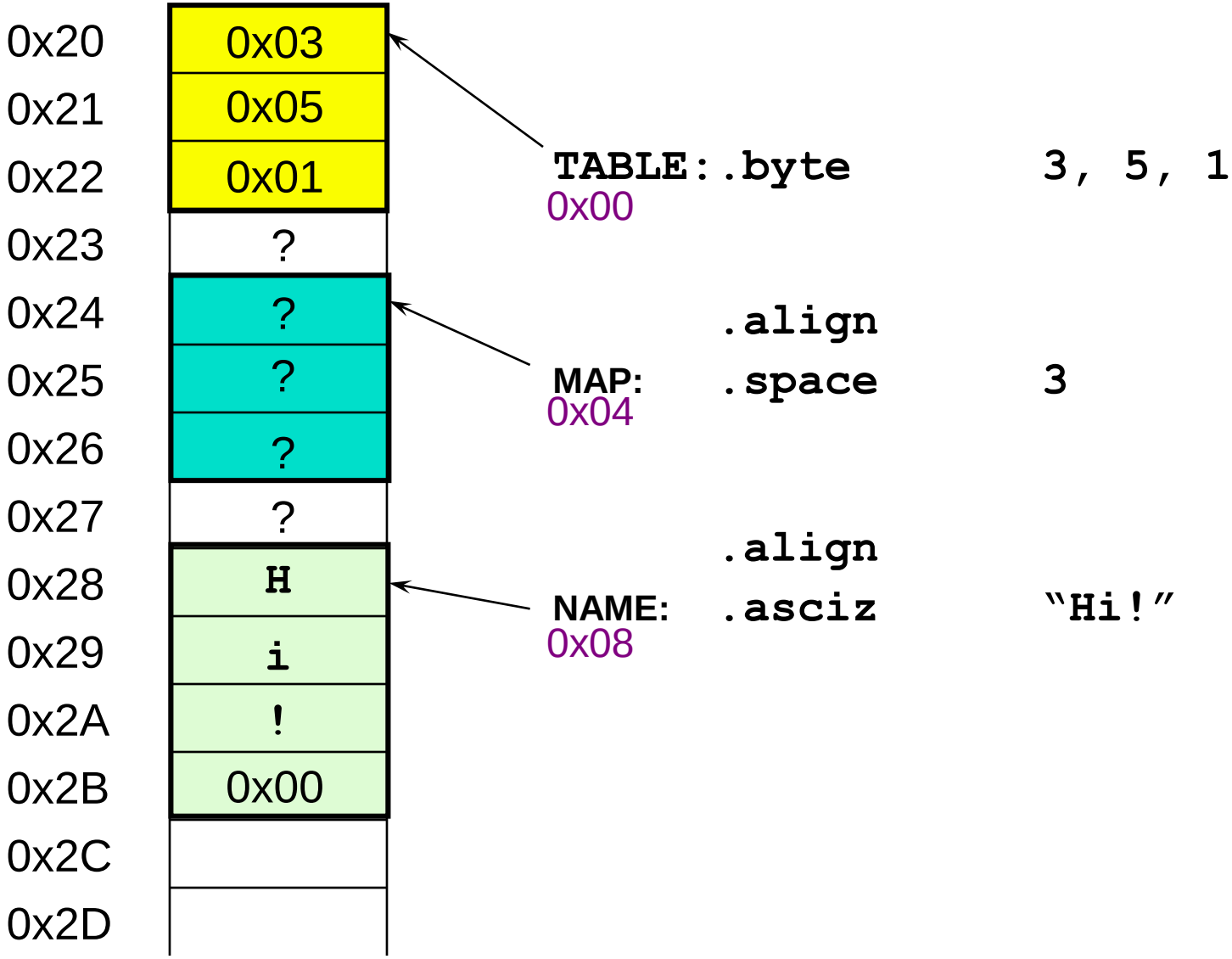
Commonly we want to initalize values.

```
niz1:   .asciz          "Dober dan"
niz2:   .ascii          "Lep dan"
        .align
stev1:  .word           512,1,65537,123456789
stev2:  .hword          1,512,65534
stev3:  .hword          0x7fe
stev4:  .byte           1, 2, 3
        .align
naslov: .word           niz1
```

- **„variables“, can be later changed (labels only represent addresses)**
- **We can declare global labels (visible in all files of the project), eg.:**

```
.global str1, str2
```

Summary-pseudo instructions & directives



Summary – compilation of (pseudo) instructions

0x20	
0x21	
0x22	
0x23	
0x24	
0x25	
0x26	
0x27	
0x28	
0x29	
0x2A	
0x2B	
0x2C	
0x2D	
0x2E	
0x2F	

```
TABLE:.byte     3, 5, 1, 2

BUF:    .word   0x01020304

A:      .byte   0x15

        .align

_START: mov     r0,#128
```

ASSEMBLER

Location counter

0x20

Labels Table

https://cpulator.01xz.net/?sys=arm&loadasm=share/sOjDspT.s