

ARM

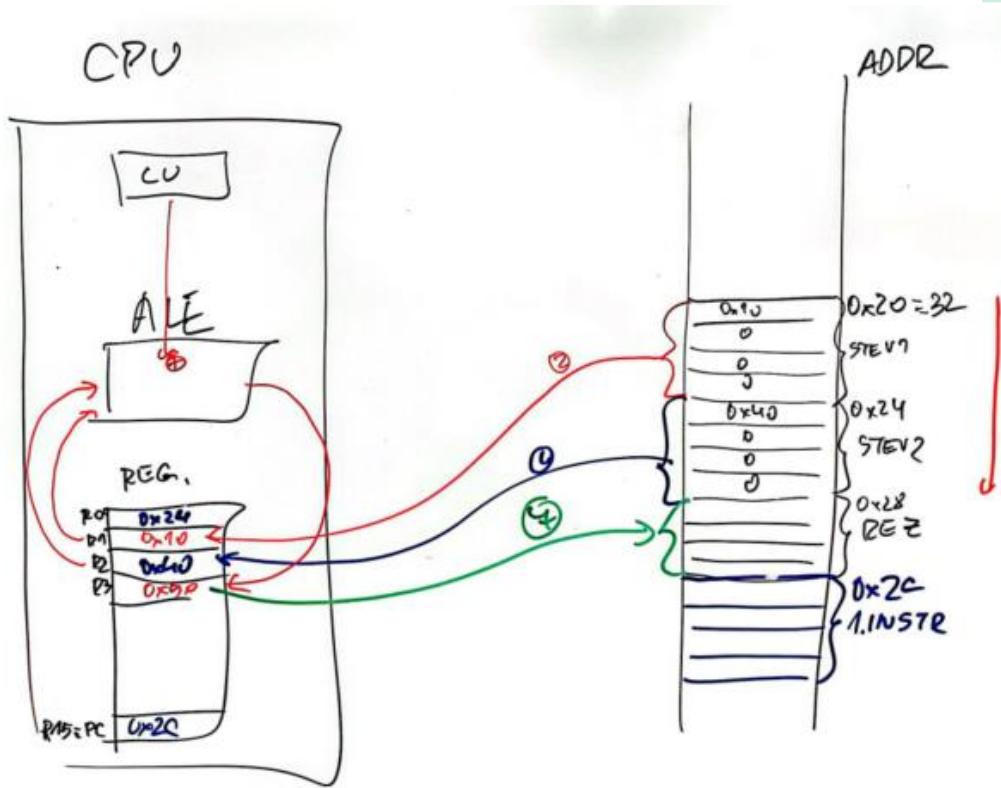
ASSEMBLY PROGRAMMING

*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 $\leftarrow$ Addr. of stev1	0xE24F0014
ldr r1, [r0]	R1 $\leftarrow$ M[R0]	0xE5901000
adr r0, stev2	R0 $\leftarrow$ Addr. of stev2	0xE24F0018
ldr r2, [r0]	R2 $\leftarrow$ M[R0]	0xE5902000
add r3, r2, r1	R3 $\leftarrow$ R1 + R2	0xE0823001
adr r0, rez	R0 $\leftarrow$ Addr. of rez	0xE24F0020
str r3, [r0]	M[R0] $\leftarrow$ R3	0xE5803000



MS Teams RA Team  
English videos

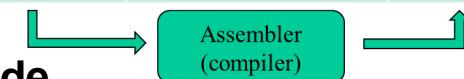
The screenshot shows the 'English videos' section of the MS Teams channel. It lists four video files:

- RA LAB ENG 01.1 Basics on Von Neumann... - nedelja ob 7:15 PM, Rozman, Robert
- RA LAB ENG 01.2 Main Memory.mp4 - nedelja ob 7:15 PM, Rozman, Robert
- RA LAB ENG 01.3 Numeral\_Systems.mp4 - nedelja ob 7:15 PM, Rozman, Robert
- RA LAB ENG 01.4 Big\_and\_Little\_Endian\_Rul... - nedelja ob 7:14 PM, Rozman, Robert

# Intro LAB: Assembly programing

**Case: Sum of two numbers in Python:**  
**rez := stev1 + stev2**

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



**Example of Python code partially compiled to byte-code**

The screenshot shows the Compiler Explorer interface. On the left, the Python source code is displayed:

```
1 def sum():
2     STEV1=0x40
3     STEV2=0x10
4     REZ = STEV1 + STEV2
5     return REZ
```

On the right, the generated assembly and byte-code are shown:

```
1 0 0 RESUME 0
2
3 1 2 LOAD_CONST 0 (<code object sum at 0x561e1ddfc3e0,
4 4 MAKE_FUNCTION 0
5 6 STORE_NAME 0 (sum)
6 8 LOAD_CONST 1 (None)
7 10 RETURN_VALUE
8
9 Disassembly of <code object sum at 0x561e1ddfc3e0, file "example.py", line 1>:
10 1 0 RESUME 0
11
12 2 2 LOAD_CONST 1 (64)
13 4 STORE_FAST 0 (STEV1)
14
15 3 6 LOAD_CONST 2 (16)
16 8 STORE_FAST 1 (STEV2)
17
18 4 10 LOAD_FAST 0 (STEV1)
19 12 LOAD_FAST 1 (STEV2)
20 14 BINARY_OP 0 (+)
21 18 STORE_FAST 2 (REZ)
22
23 5 20 LOAD_FAST 2 (REZ)
24 22 RETURN_VALUE
```

<https://godbolt.org/>

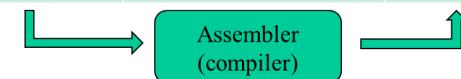
```
def sum():
    STEV1=0x40
    STEV2=0x10
    REZ = STEV1 + STEV2
    return REZ
```

Case Sum

# Intro LAB: Assembly programing

**Case: Sum of two numbers in C:**  
**rez := stev1 + stev2**

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



**Example of C-code compiled to ARM Assembler**

The screenshot shows the Compiler Explorer interface. On the left, the C source code is displayed:

```

1 /* Type your code here, or load an example. */
2 int sum(int sum) {
3     int stev1=0x40;
4     int stev2=0x10;
5     int rez=0;
6
7     rez = stev1+stev2;
8
9     return rez;
10}
11
  
```

On the right, the ARM assembly output is shown:

```

1 sum:
2     str    fp, [sp, #-4]!
3     add    fp, sp, #0
4     sub    sp, sp, #28
5     str    r0, [fp, #-24]
6     mov    r3, #64
7     str    r3, [fp, #-8]
8     mov    r3, #16
9     str    r3, [fp, #-12]
10    mov   r3, #0
11    str    r3, [fp, #-16]
12    ldr    r2, [fp, #-8]
13    ldr    r3, [fp, #-12]
14    add    r3, r2, r3
15    str    r3, [fp, #-16]
16    ldr    r3, [fp, #-16]
17    mov    r0, r3
18    add    sp, fp, #0
19    ldr    fp, [sp], #4
20    bx    lr
  
```

<https://godbolt.org/>

```

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

    rez = stev1+stev2;

    return rez;
}
  
```

**Case Sum**

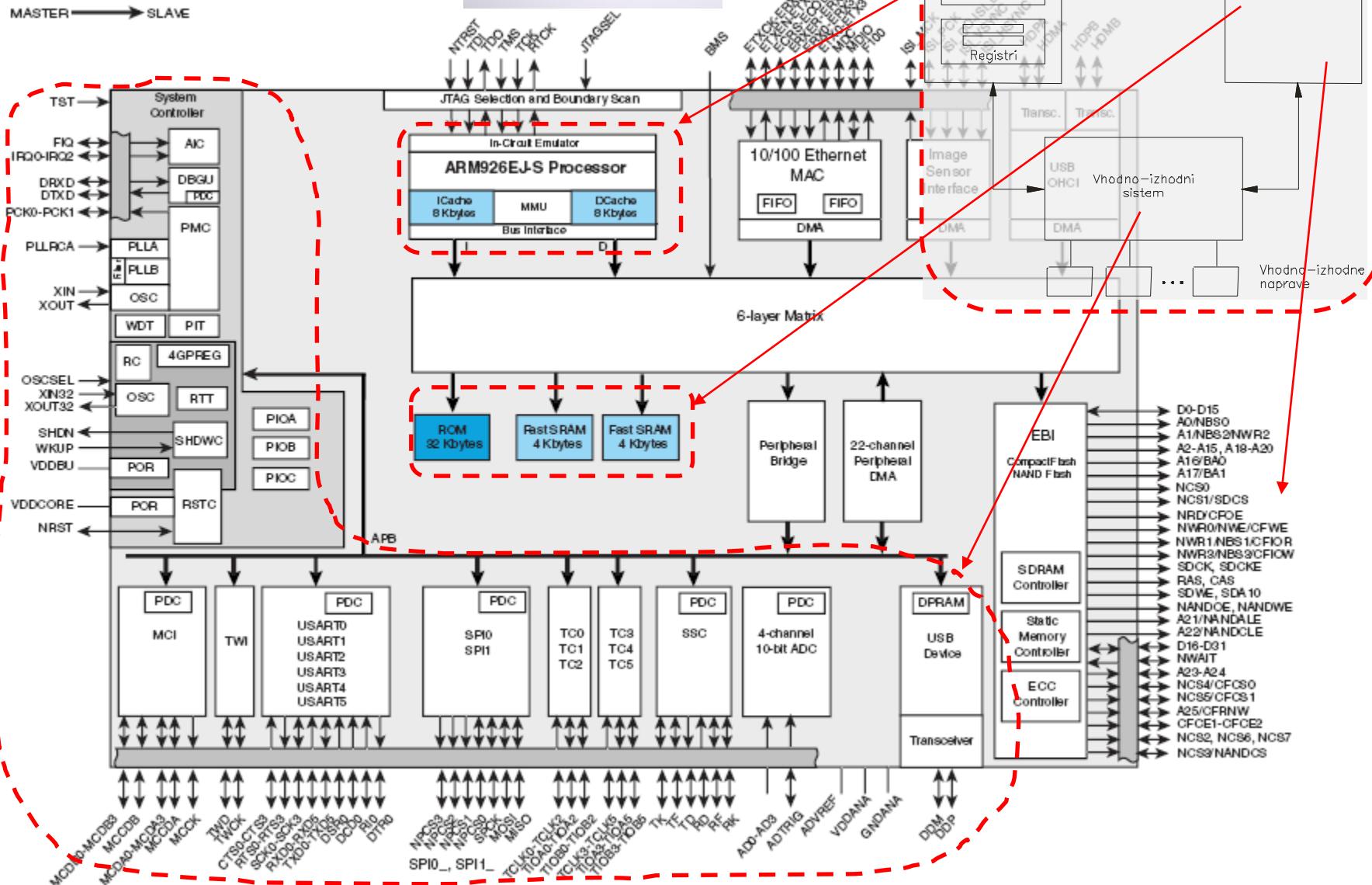
# ARM (Advanced RISC Machine) = RISC?

## 32-bit ISA (Instruction Set Architecture) :

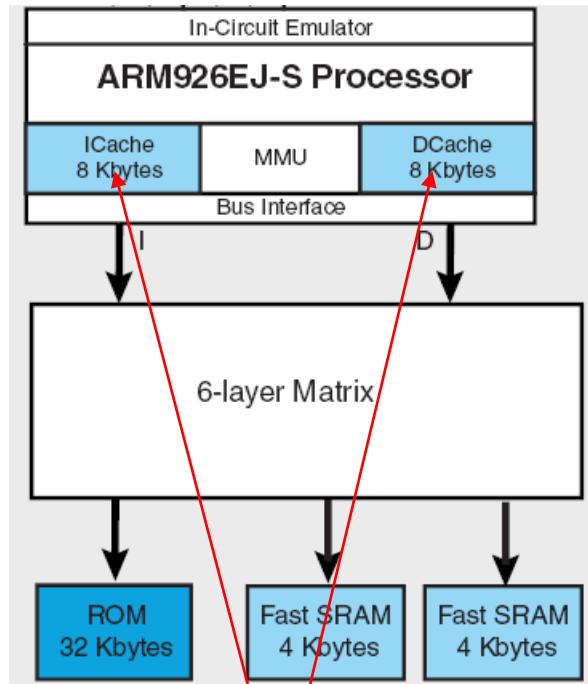
- + load/store architecture
- + pipeline
- + reduced instruction set, all instructions are 32-bit
- + orthogonal registers – all 32-bit
  
- many addressing modes
- many instruction formats
  
- some instructions take **multiple clock cycles** to execute (eg. *load/store multiple*) – but they make programmes shorter
- additional 16-bit instruction set „Thumb“ – shorter programmes
- conditional instruction execution – execute only if condition is true

# AT91SAM9260

(microcontroller)



# AT91SAM9260



Harvard Architecture

On Cache level

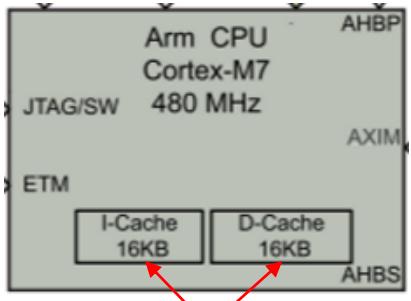
Memory map

Internal Memory Mapping	
0x0000 0000	Boot Memory (1)
0x10 0000	ROM
0x10 8000	Reserved
0x20 0000	SRAM0
0x20 1000	Reserved
0x30 0000	SRAM1
0x30 1000	Reserved
0x50 0000	UHP
0x50 4000	Reserved
0x0FFF FFFF	

Princeton architecture

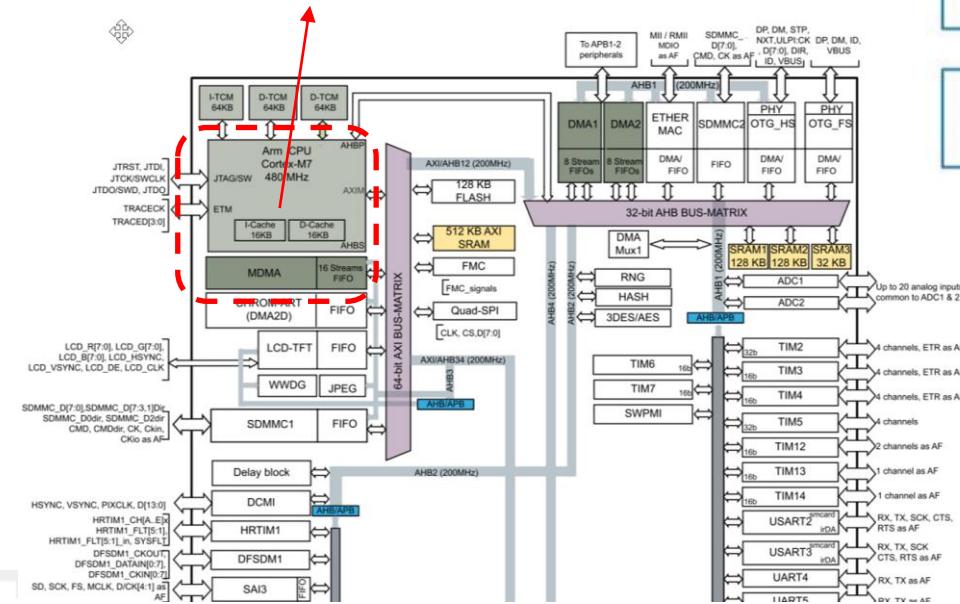
Main memory

# STM32H750XB



# Harvard Architecture

## On Cache level



## Memory map

Vendor-specific memory	511MB	0xFFFFFFFF
Private peripheral bus	1.0MB	0xE0100000 0xE00FFFFF
External device	1.0GB	0xE0000000 0xDFFFFFFF
External RAM	1.0GB	0xA0000000 0x9FFFFFFF
Peripheral	0.5GB	0x60000000 0x5FFFFFFF
SRAM	0.5GB	0x40000000 0x3FFFFFFF
Code	0.5GB	0x20000000 0x1FFFFFFF
		0x00000000

Princeton architecture

## Main memory

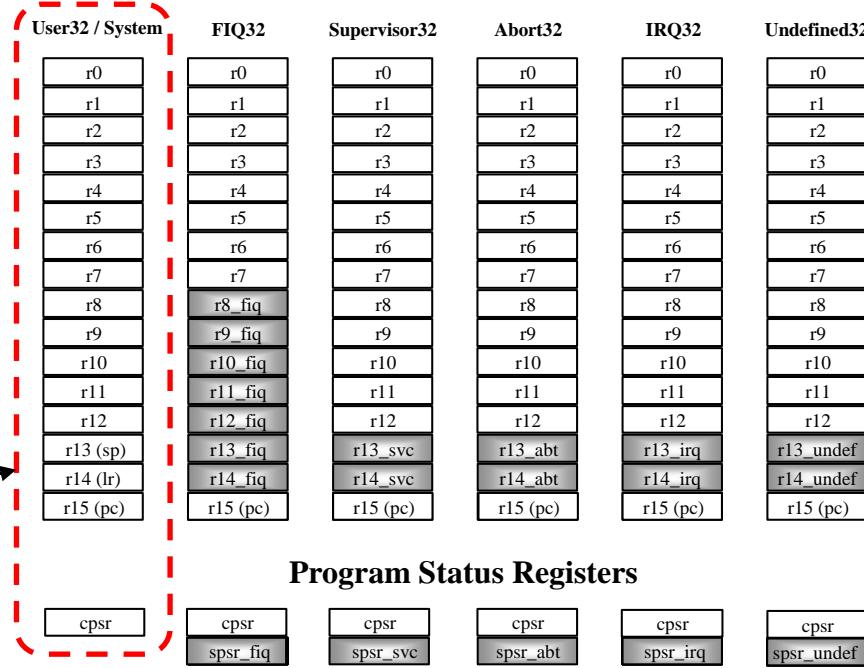
## MEMORY

1

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

# ARM programming model

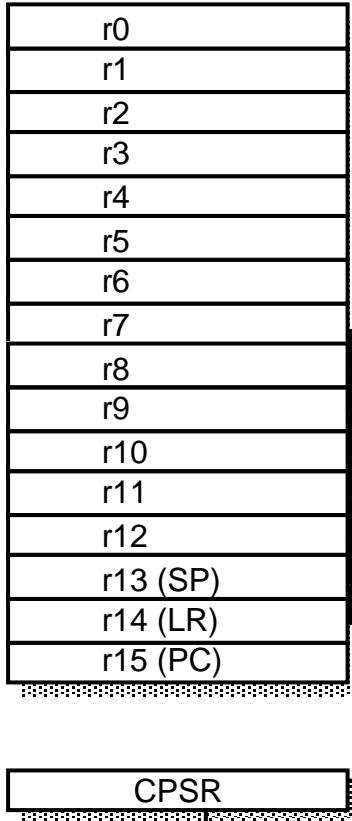
- **Consists of :**
  - 16 general purpose registers
  - Status register CPSR (Current Program Status Register)
- **CPU supports multiple operation modes, each has its own set of registers – overall 36 registers for all modes**
- **Only few are visible in certain processor's operation mode**
- **Operation 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*)

# 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:**
  - Instructions (mnemonics),
  - registers
  - addresses
  - constants

Assembly language	Instruction description	Machine language
adr r0, stev1	$R_0 \leftarrow \text{Addr. of stev1}'$	0xE24F0014
ldr r1, [r0]	$R_1 \leftarrow M[R_0]$	0xE5901000
adr r0, stev2	$R_0 \leftarrow \text{Addr. of stev2}$	0xE24F0018
ldr r2, [r0]	$R_2 \leftarrow M[R_0]$	0xE5902000
add r3, r2, r1	$R_3 \leftarrow R_1 + R_2$	0xE0823001
adr r0, rez	$R_0 \leftarrow \text{Addr. of rez}$	0xE24F0020
str r3, [r0]	$M[R_0] \leftarrow R_3$	0xE5803000

The diagram shows a central teal rounded rectangle labeled "Assembler (compiler)". To its left is a teal L-shaped arrow pointing towards it from the left. To its right is a teal T-shaped arrow pointing upwards from below. Dashed arrows connect the assembly code in the table to the corresponding machine language values, illustrating the mapping process.

- **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 corresponding 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**

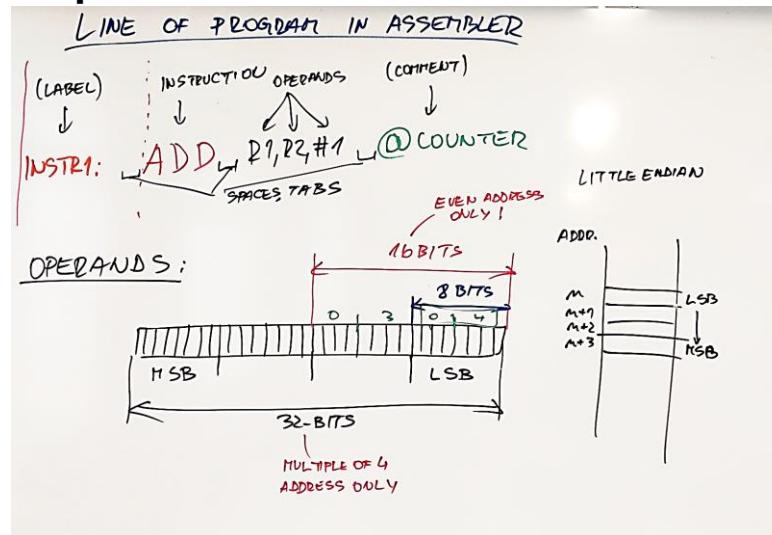
# Assembly programming

## ARMv4T Partial Instruction Set Summary

- List of instructions
  - On Moodle platform

Operation	Syntax
Move	<code>mov[cond]{s} Rd, shift_op</code> <code>mvn[cond]{s} Rd, shift_op</code> <code>mrs[cond] Rd, cpsr</code> <code>mrs[cond] Rd, spsr</code> <code>msr[cond] cpsr_fields, Rm</code> <code>msr[cond] spsr_fields, Rm</code> <code>msr[cond] cpsr_fields, #imm8r</code> <code>msr[cond] spsr_fields, #imm8r</code>
Arithmetic	<code>add[cond]{s} Rd, Rn, shift_op</code> <code>adc[cond]{s} Rd, Rn, shift_op</code> <code>sub[cond]{s} Rd, Rn, shift_op</code> <code>sbc[cond]{s} Rd, Rn, shift_op</code> <code>rsb[cond]{s} Rd, Rn, shift_op</code> <code>rsc[cond]{s} Rd, Rn, shift_op</code> <code>mul[cond]{s} Rd, Rm, Rs</code> <code>mla[cond]{s} Rd, Rm, Rs, Rn</code> <code>umull[cond]{s} RdLo, RdHi, Rm, Rs</code> <code>umlal[cond]{s} RdLo, RdHi, Rm, Rs</code> <code>smull[cond]{s} RdLo, RdHi, Rm, Rs</code> <code>smal[cond]{s} RdLo, RdHi, Rm, Rs</code>

- 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
              ldr r5,[r0]
```

@ increase counter

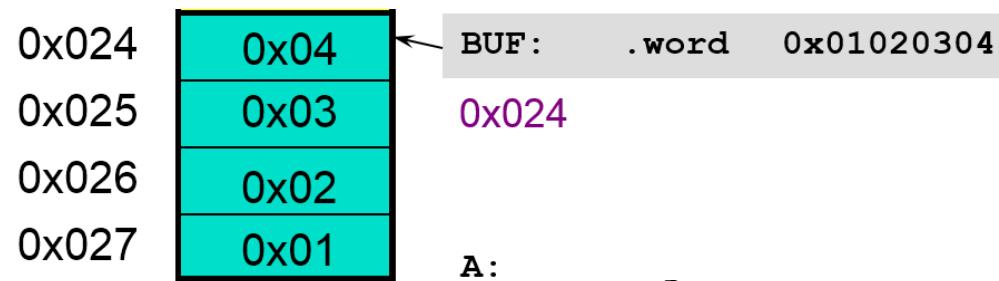
- Columns are separated by tabulators, spacings are also allowed

# 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                       $\Rightarrow$             0x000000FF

- Rule for longer operands : „Little Endian“



# Labels

Labels are meant as a symbolic name of:

- Memory locations or
- Program lines

Labels 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, stvl` 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)      | <code>.text .data</code>          |
| • memory address for compilation        | <code>.org</code>                 |
| • content alignment (16/32bits)         | <code>.align</code>               |
| • memory reservation for „variables“    | <code>.space</code>               |
| • memory Initialization for „variables“ | <code>.(h)word, .byte, ...</code> |
| • end of compilation                    | <code>.end</code>                 |

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

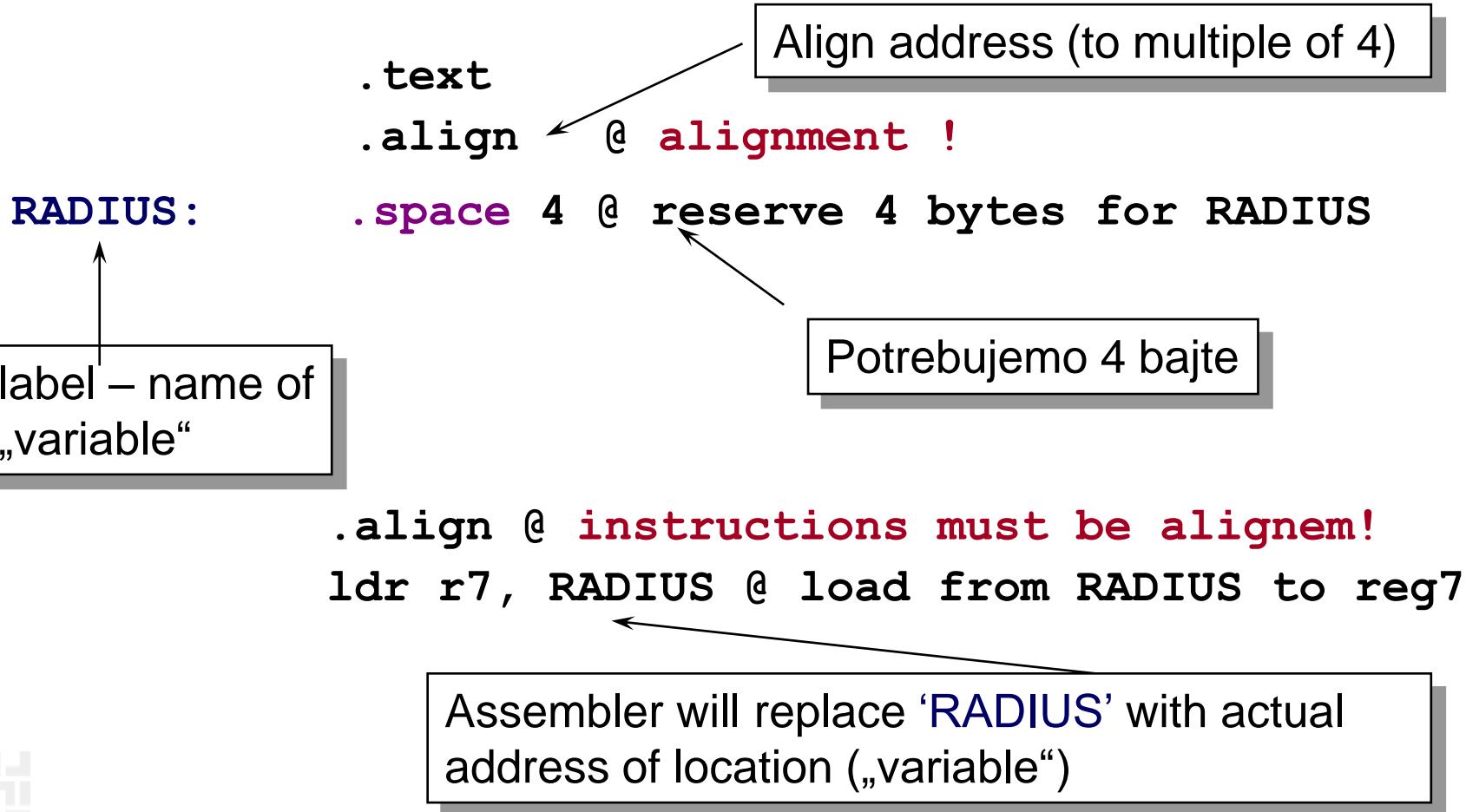
```
.text
.org 0x20
@spremenljivke
```

```
.align
.global _start
_start:
@program
```

```
end: b end
```

# Memory reservation for „variables“

We have to reserve corresponding space for „variables“.



# 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** corresponds to address of the first byte in a row of 40B.
- label **BUFFER2** corresponds to address of the first byte in a row of 10B.  
**It's value is 40 more than BUFFER**
- label **BUFFER3** corresponds 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 initialize 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

0x20	0x03			
0x21	0x05			
0x22	0x01			
0x23	?			
0x24	?			
0x25	?			
0x26	?			
0x27	?			
0x28	H			
0x29	i			
0x2A	!			
0x2B	0x00			
0x2C				
0x2D				

Diagram illustrating memory layout and pseudo-instructions:

- TABLE:** .byte 0x00 3, 5, 1
- MAP:** .align .space 3
- NAME:** .align .asciz "Hi!"

The memory layout shows the following data:

- Address 0x20: Value 0x03
- Address 0x21: Value 0x05
- Address 0x22: Value 0x01
- Address 0x23: Value ? (uninitialized)
- Address 0x24: Value ? (uninitialized)
- Address 0x25: Value ? (uninitialized)
- Address 0x26: Value ? (uninitialized)
- Address 0x27: Value ? (uninitialized)
- Address 0x28: Value H
- Address 0x29: Value i
- Address 0x2A: Value !
- Address 0x2B: Value 0x00 (string terminator)
- Address 0x2C: Value (empty)
- Address 0x2D: Value (empty)

# 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