

ARM

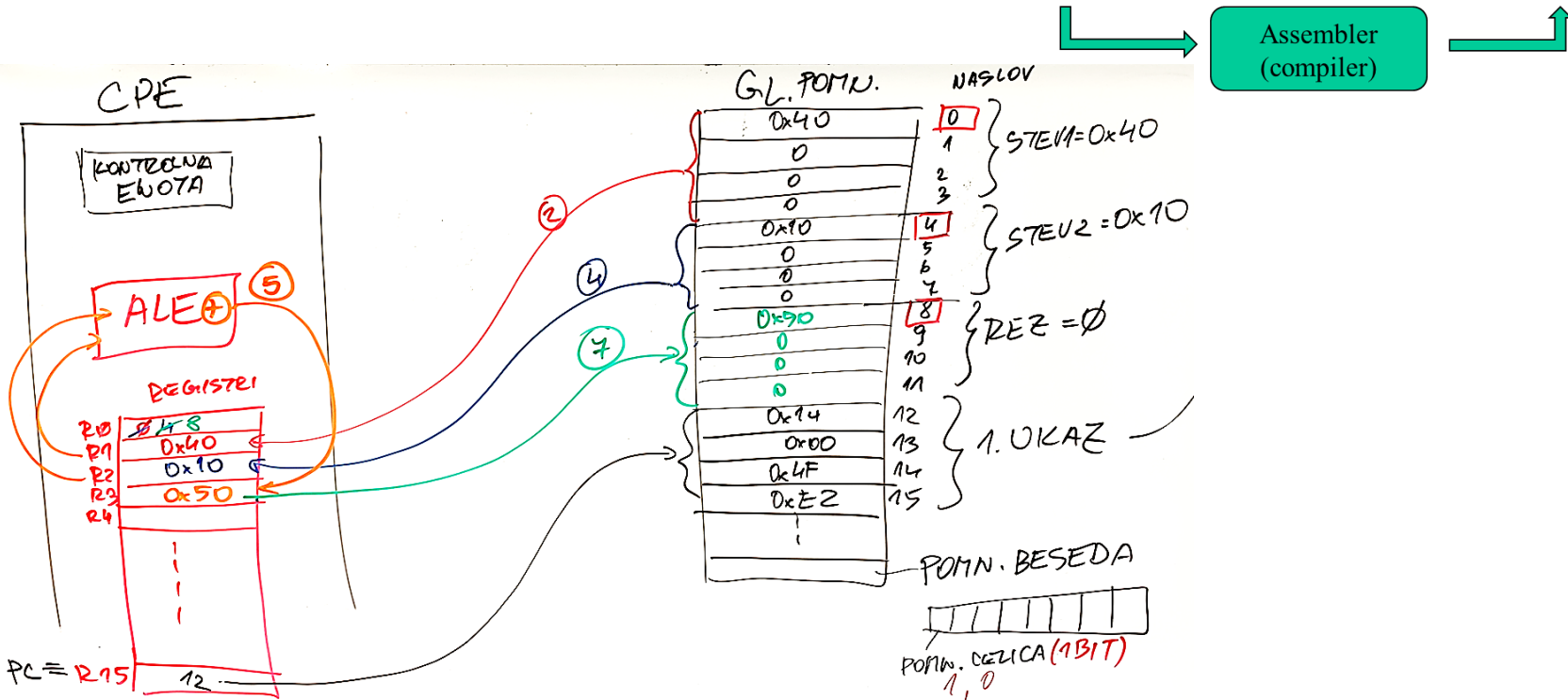
ASSEMBLY PROGRAMMING

1. part

Intro LAB: Assembly programming

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

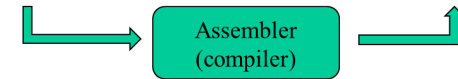


Intro LAB: Assembly programming

Case: Sum of two numbers :
rez := stev1 + stev2

Example of C-code compiled to ARM Assembler

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



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 void sum(int num) {
3     int stev1=0x40;
4     int stev2=0x10;
5     int rez=0;
6
7     rez=stev1+stev2;
8 }
```

On the right, the ARM assembly output for the same code 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    nop
17    add    sp, fp, #0
18    ldr    fp, [sp], #4
19    bx    lr
```

<https://godbolt.org/>

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

    rez = stev1+stev2;

    return rez;
}
```

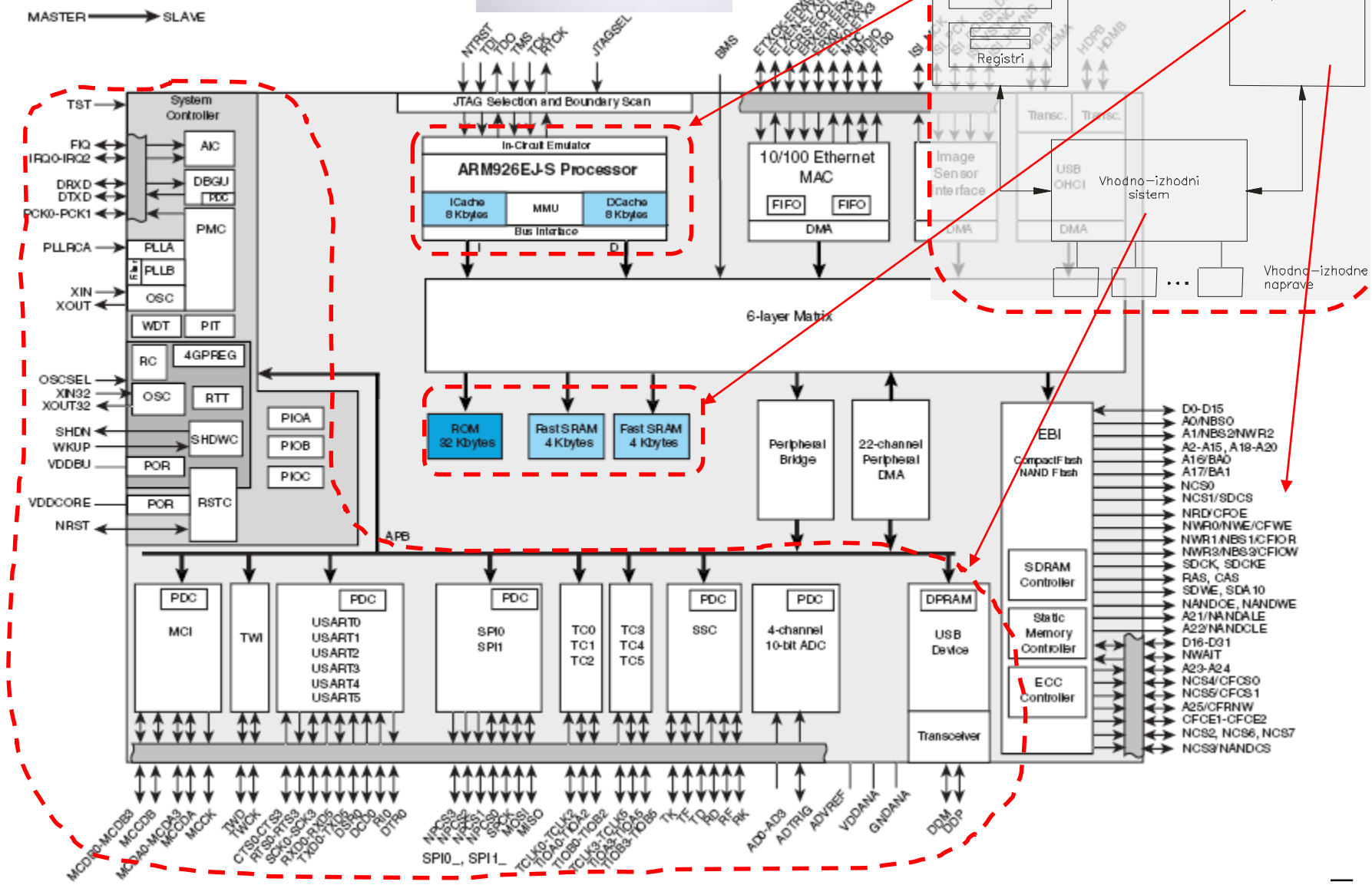
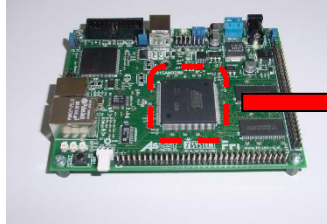
Case Sum

ARM (Advanced RISC Machine) = RISC?

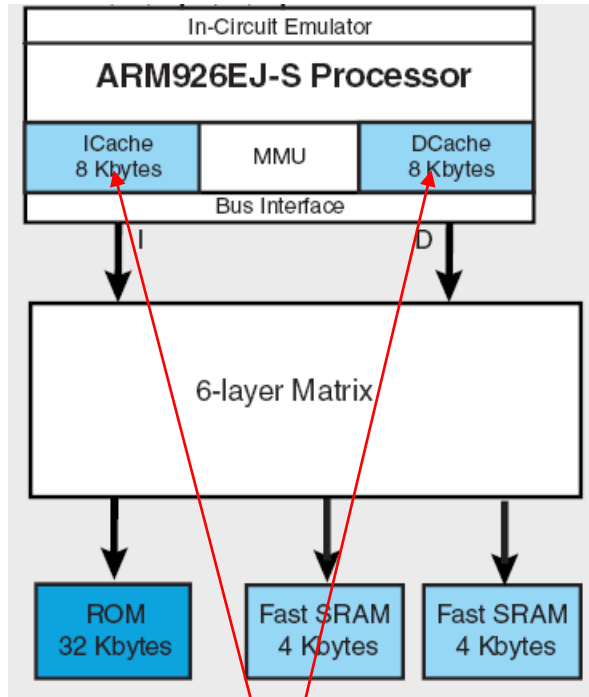
- + load/store architecture
 - + pipeline
 - + reduced instruction set, all instructions are 32-bit
 - + orthogonal registers – all 32-bit
-
- many addressing modes
 - many instruction formats
-
- some instructions take multiple clock cycles to execute (eg. *load/store multiple*) – but they make programmes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

AT91SAM9260

(microcontroller)



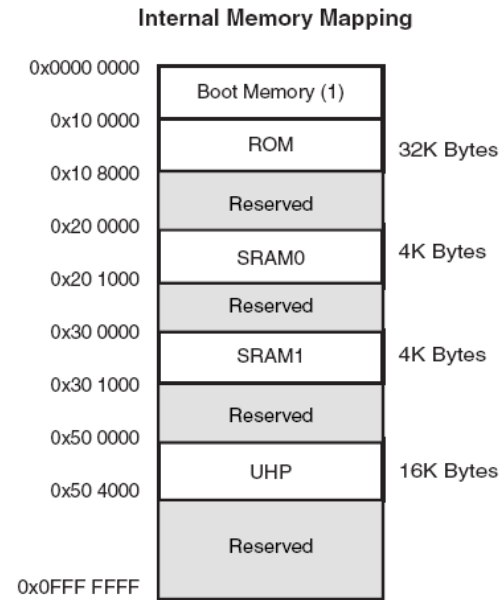
AT91SAM9260



Harvard Architecture

On Cache level

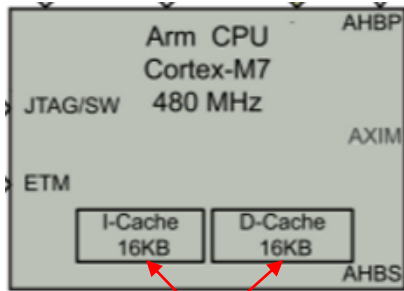
Memory map



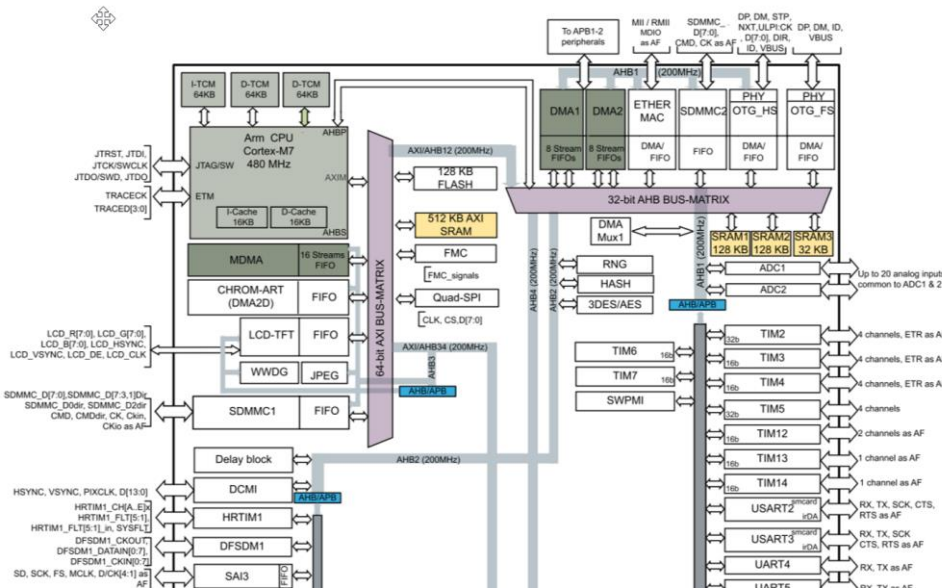
Princeton architecture

Main memory

STM32H750XB



Harvard Architecture
On Cache level



Memory map

Figure 8. Processor memory map

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

MSv39642V1

Princeton architecture

Main memory

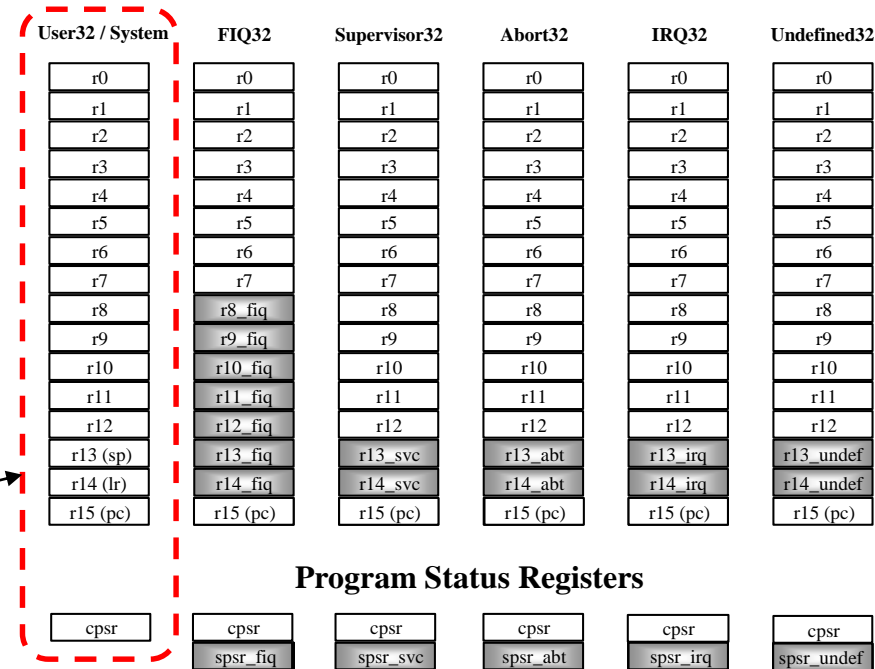
MEMORY

```

{
  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

r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 (SP)
r14 (LR)
r15 (PC)

Visible 17 32-bit registers:

r0 – r15 and CPSR

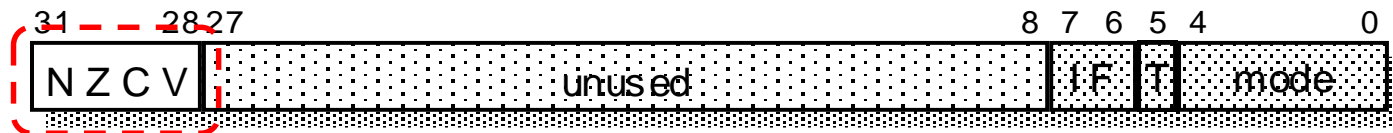
Visible registers:

- 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*)

CPSR

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:

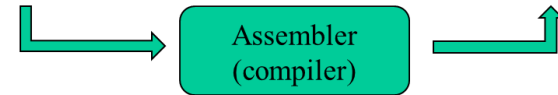
N = 1: bit 31 of result is 1 (Negative),	N = 0: bit 31 of result is 0	(<i>Negative</i>)
Z = 1: result is 0,	Z = 0: result is not 0 (non-zero)	(<i>Zero</i>)
C = 1: carry,	C = 0: no carry	(<i>Carry</i>)
V = 1: overflow,	V = 0: no overflow	(<i>oVerflow</i>)

Assembly programming

- **Assembly language:**

- Instructions (mnemonics),
- registers
- addresses
- constants

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



- **You don't have to:**

- Know machine instructions and their composition
- Calculate with addresses

Assembly language compiler (assembler) :

- Compiles symbolic names 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**

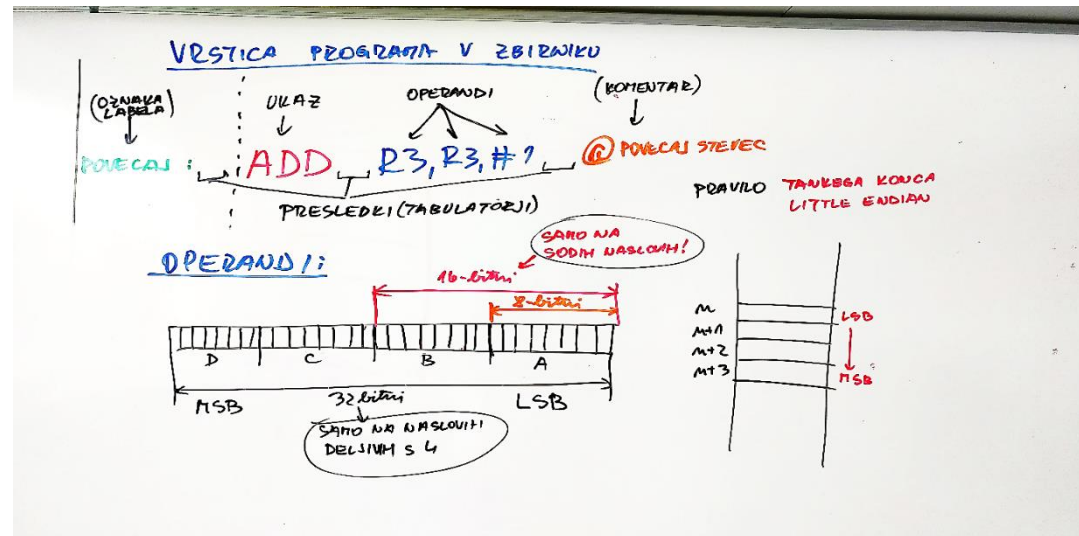
Assembly programming

ARMv4T Partial Instruction Set Summary

- List of instructions
 - On Moodle platform

Operation		Syntax
Move	Move	<code>mov{cond}{s} Rd, shift_op</code>
	with NOT	<code>mvn{cond}{s} Rd, shift_op</code>
	CPSR to register	<code>mrs{cond} Rd, cpsr</code>
	SPSR to register	<code>mrs{cond} Rd, spsr</code>
	register to CPSR	<code>msr{cond} cpsr_fields, Rm</code>
	register to SPSR	<code>msr{cond} spsr_fields, Rm</code>
	immediate to CPSR	<code>msr{cond} cpsr_fields, #imm8r</code>
	immediate to SPSR	<code>msr{cond} spsr_fields, #imm8r</code>
Arithmetic	Add	<code>add{cond}{s} Rd, Rn, shift_op</code>
	with carry	<code>adc{cond}{s} Rd, Rn, shift_op</code>
	Subtract	<code>sub{cond}{s} Rd, Rn, shift_op</code>
	with carry	<code>sbc{cond}{s} Rd, Rn, shift_op</code>
	reverse subtract	<code>rsb{cond}{s} Rd, Rn, shift_op</code>
	reverse subtract with carry	<code>rsc{cond}{s} Rd, Rn, shift_op</code>
	Multiply	<code>mul{cond}{s} Rd, Rm, Rs</code>
	with accumulate	<code>mla{cond}{s} Rd, Rm, Rs, Rn</code>
	unsigned long	<code>umull{cond}{s} RdLo, RdHi, Rm, Rs</code>
	unsigned long with accumulate	<code>umlal{cond}{s} RdLo, RdHi, Rm, Rs</code>
	signed long	<code>smull{cond}{s} RdLo, RdHi, Rm, Rs</code>
	signed long with accumulate	<code>smlal{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  $\implies$  0xE0823001=0b1110...0001
```

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

```
R1 + R2  $\implies$  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]
```

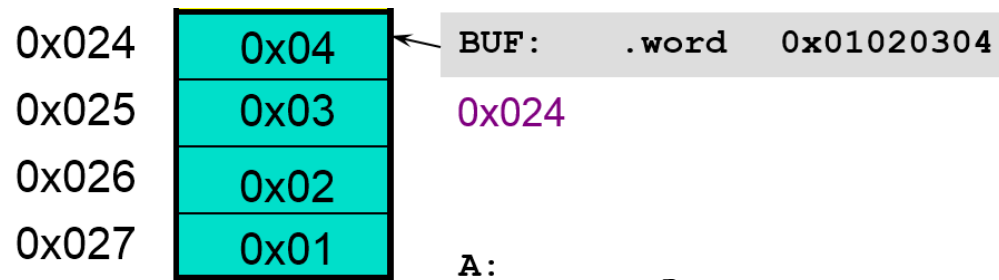
- Columns are separated by tabulators, spacings are also allowed

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 \Longrightarrow 0x000000FF

- Rule for longer operands : „Little Endian“



Labels

Labels are meant as a symbolic name of:

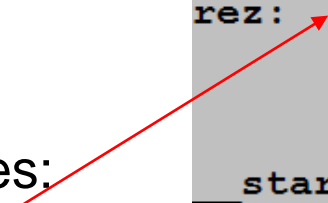
- Memory locations or
- Program lines

Labels are commonly used in two cases:

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

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

```
                .text
stev1:          .word  64
stev2:          .word  0x10
rez:           .space 4
                .align
                .global __start
__start:
                ldr   r1, stev1
                ldr   r2, stev2
                add  r3, r2, r1
                str  r3, rez
__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 are used for:

- have a dot in front of them „.“
- memory segments (starting point) **.text .data**
- 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

Pseudo instructions 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

`.text`

Memory reservation for „variables“

We have to reserve corresponding space for „variables“.

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

Align address (to multiple of 4)

RADIUS:

label – name of
„variable“

Potrebujemo 4 bajte

```
.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** 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

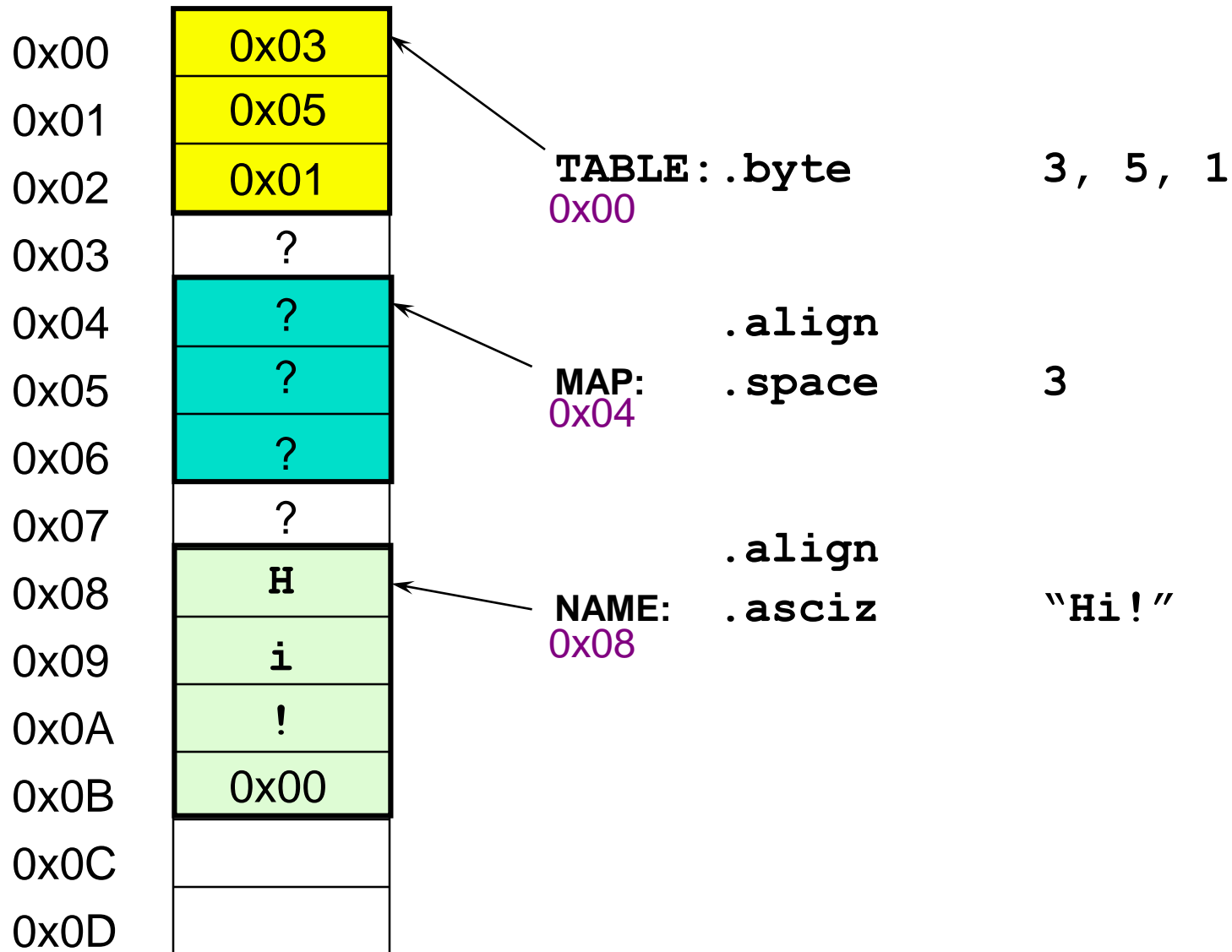
Commonly we want to initialize values.

```
niz1:  .asciz      "Dober dan"
niz2:  .ascii
      .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

0x00	
0x01	
0x02	
0x03	
0x04	
0x05	
0x06	
0x07	
0x08	
0x09	
0x0A	
0x0B	
0x0C	
0x0D	
0x0E	
0x0F	

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

BUF:    .word   0x01020304

A:      .byte   0x15

        .align

_START: mov     r0, #128
```

ASSEMBLER

Location counter

0x00

Labels Table
