ARM

ASSEMBLY PROGRAMMING

1. part

ARM

ASSEMBLY PROGRAMMING

1. part

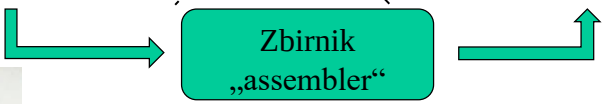
RA LAB 2.1 Basics of ARM microcontroller

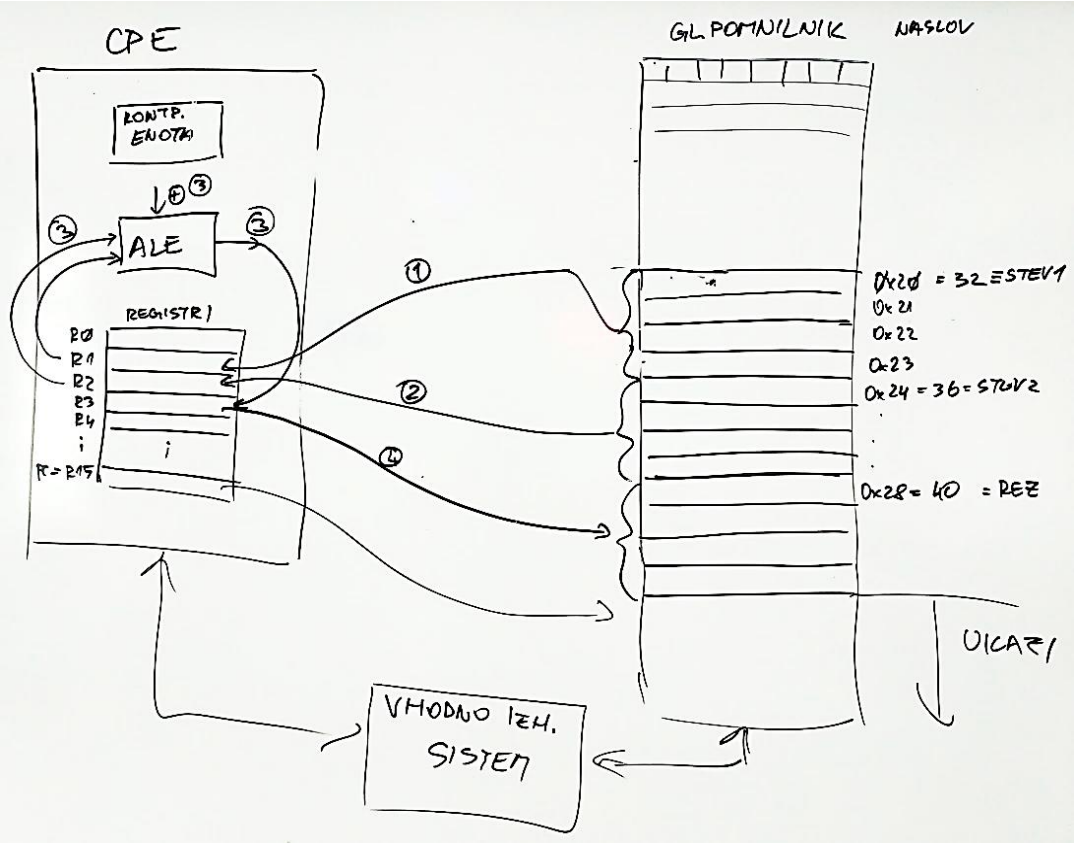
Intro LAB: Assembly programing

Case: Sum of two numbers :

rez := stev1 + stev2

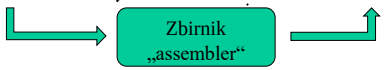
Zbirni jezik	Opis ukaza	Strojni jezik
ldr r1, stev1	R1 ← M[0x20]	0xE51F1014
ldr r2, stev2	R2 ← M[0x24]	0xE51F2014
add r3, r2, r1	R3 ← R1 + R2	0xE0823001
str r3, rez	M[0x28] ← R3	0xE50F3018





Intro LAB: Assembly programing

Zbirni jezik	Opis ukaza	Strojni jezik
ldr r1, stev1	R1 ← M[0x20]	0xE51F1014
ldr r2, stev2	R2 ← M[0x24]	0xE51F2014
add r3, r2, r1	R3 ← R1 + R2	0xE0823001
str r3, rez	M[0x28] ← R3	0xE50F3018

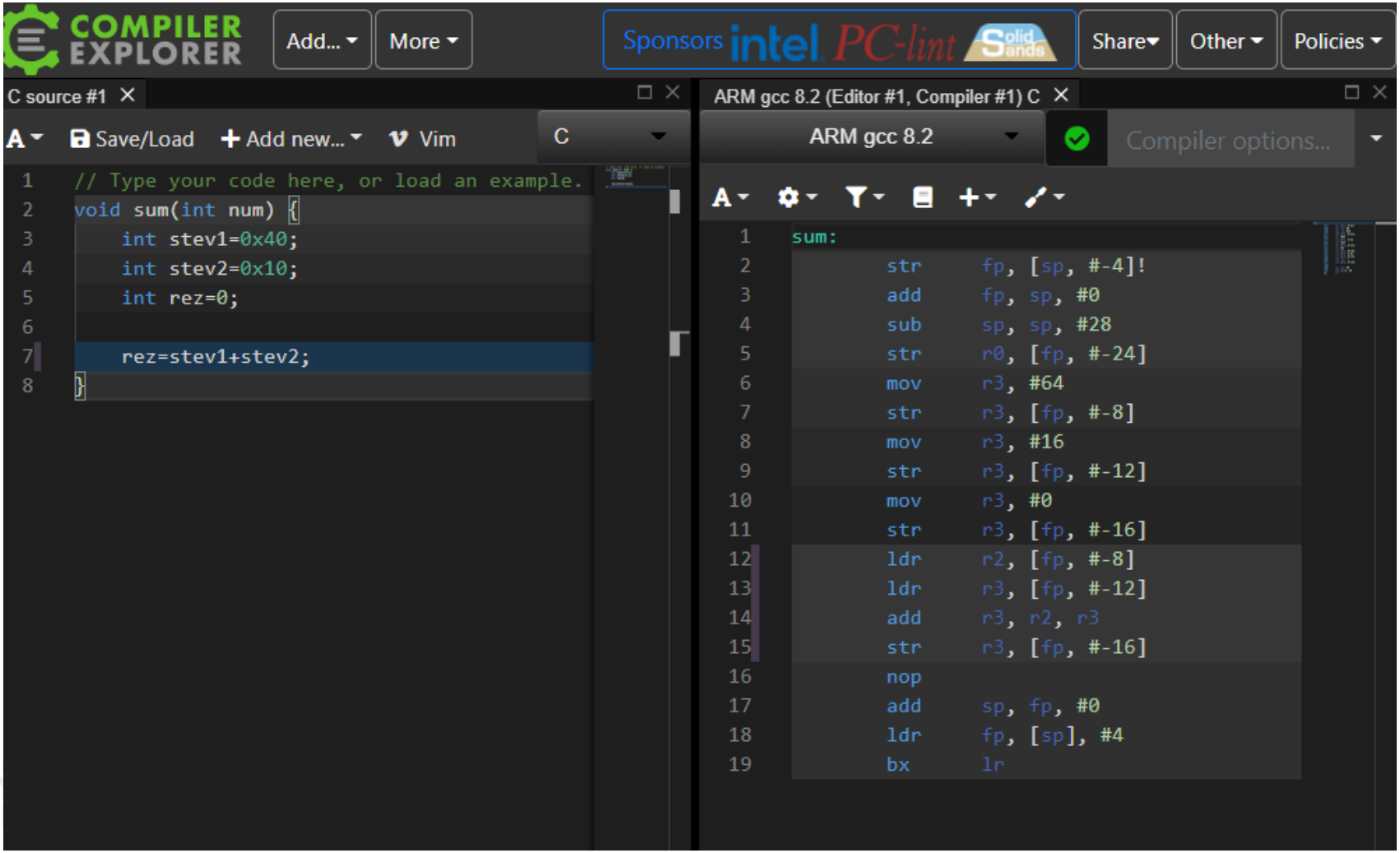


Case: Sum of two numbers :

rez := stev1 + stev2

Example of C-code compiled to ARM Assembler

https://godbolt.org/



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

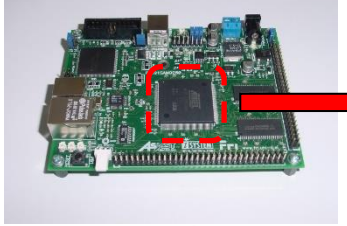
- some instructions take multiple clock cycles to execute (eg. *load/store multiple*) – but they make programmes shorter

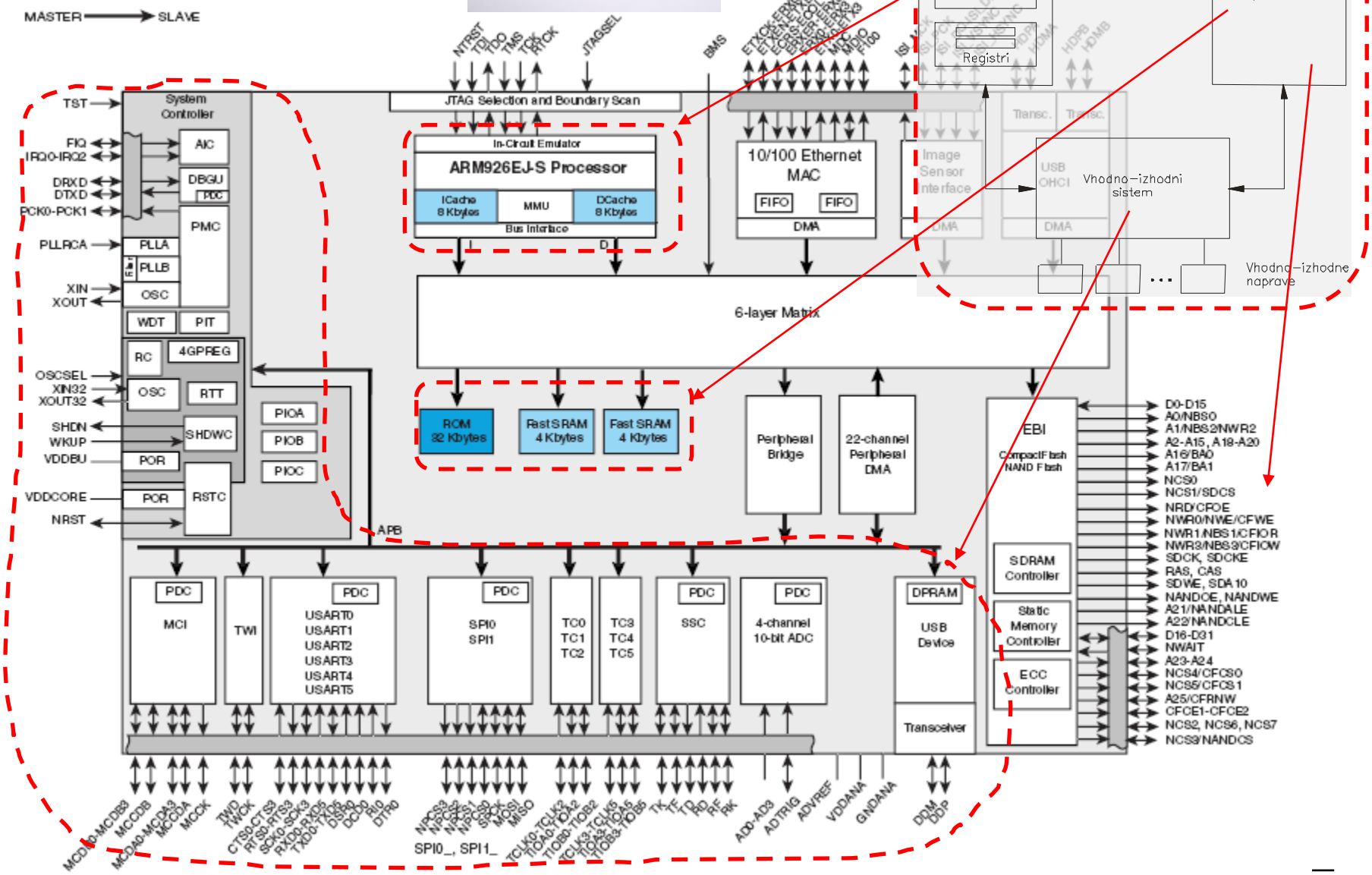
- additional 16-bit instruction set „Thumb“ – shorter programmes

- conditional instruction execution – execute only if condition is true

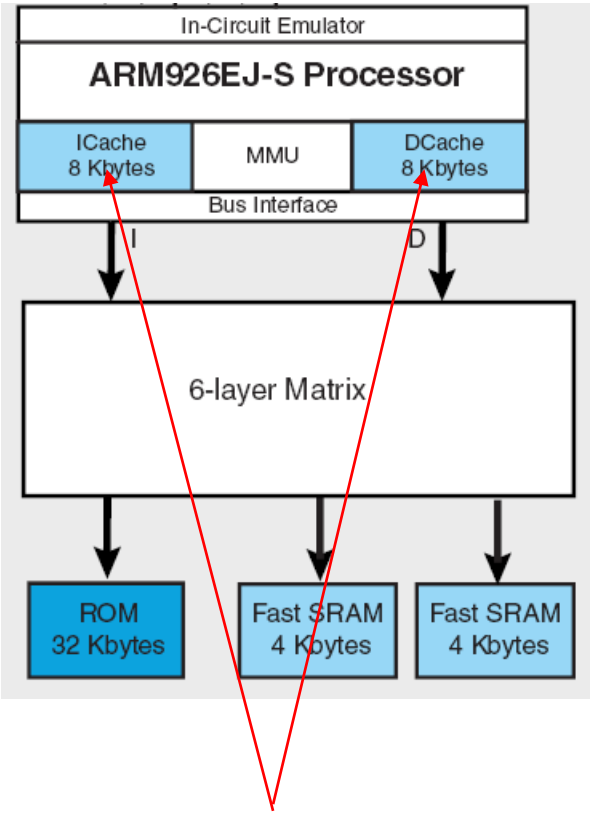
AT91SAM9260

(microcontroller)





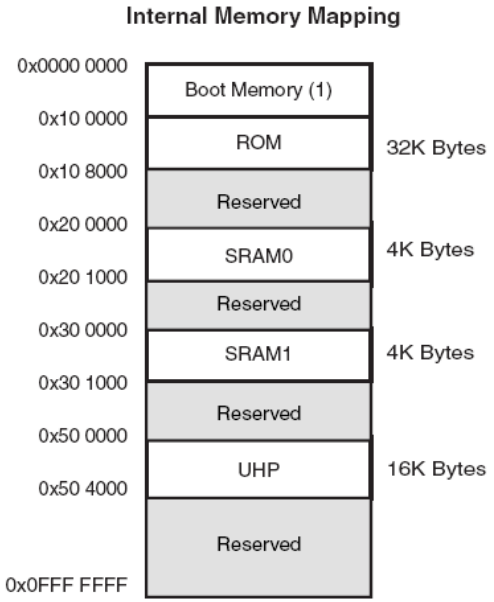
AT91SAM9260



Harvard Architecture

Cache level

Memory map



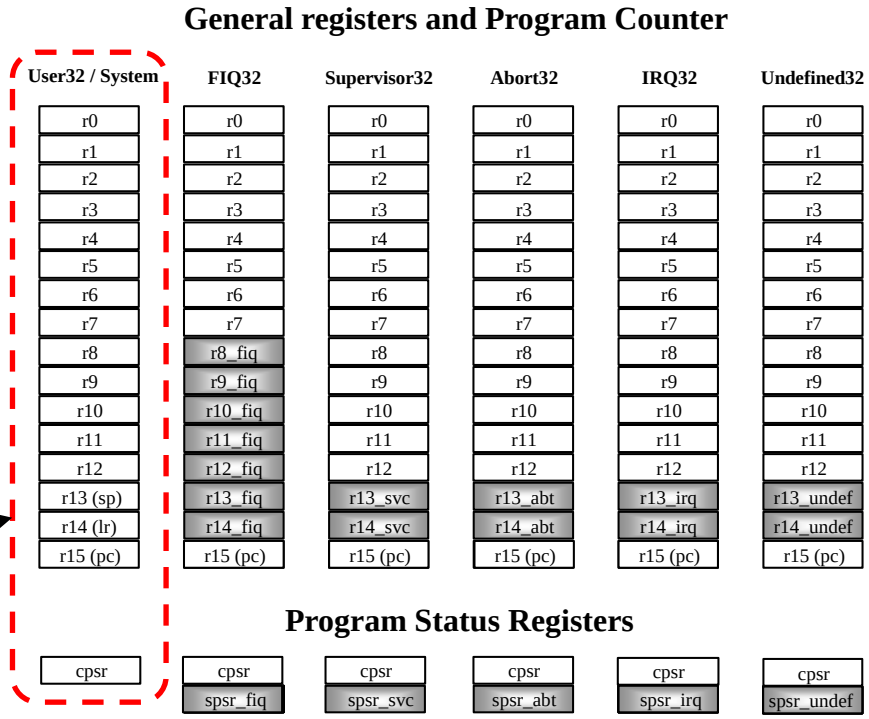
Princeton architeture

Main memory

ARM programming model

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

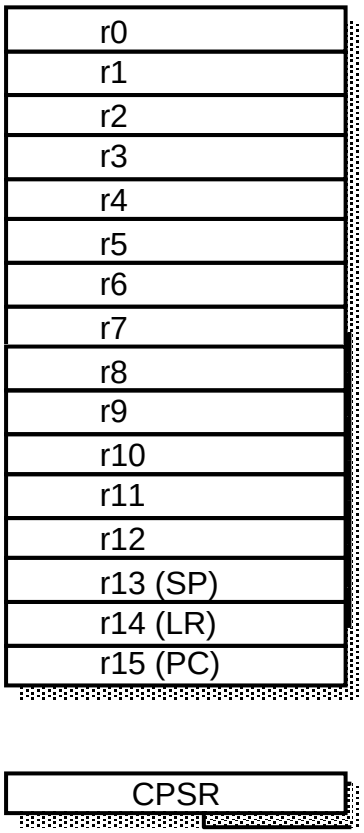
- **CPU supports multiple** *operation modes*, **each has its own set of registers – overall 36 registers for all modes**



- **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

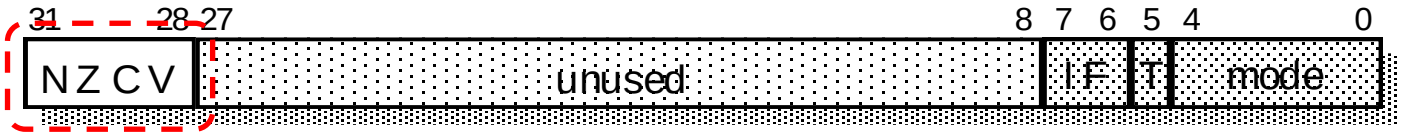
Visible 17 32-bit registers:
r0 – r15 and CPSR

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*)

ARM

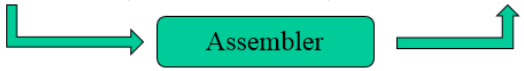
ASSEMBLY PROGRAMMING

1. part

RA LAB 2.2 ARM assembly programming

Assembly programming

Assembly language	Instruction description	Machine language
ldr r1, stev1	R1 ← M[0x20]	0xE51F1014
ldr r2, stev2	R2 ← M[0x24]	0xE51F2014
add r3, r2, r1	R3 ← R1 + R2	0xE0823001
str r3, rez	M[0x28] ← R3	0xE50F3018



- **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 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**

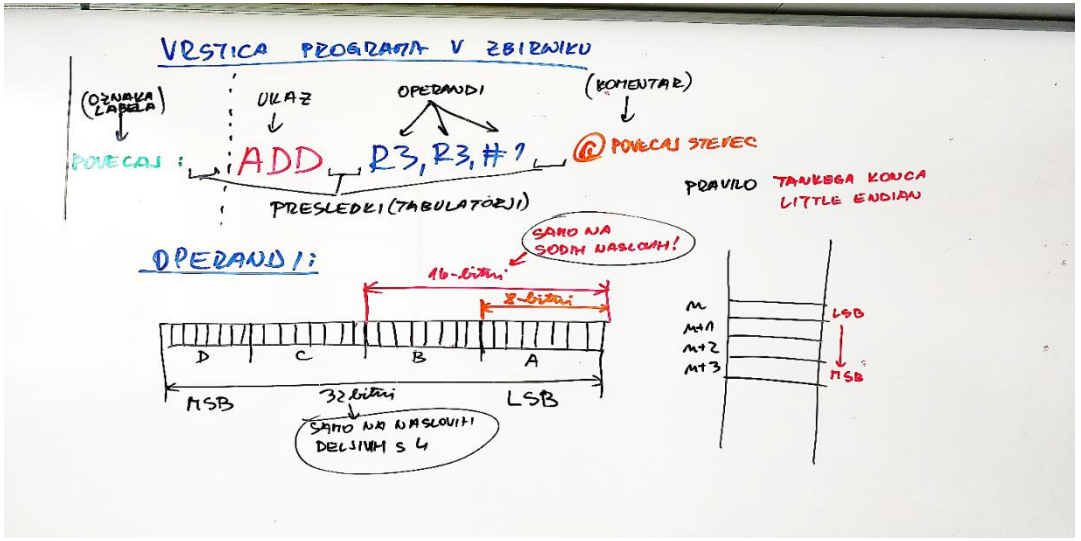
Assembly programming

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

ARMv4T Partial Instruction Set Summary

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

- **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**

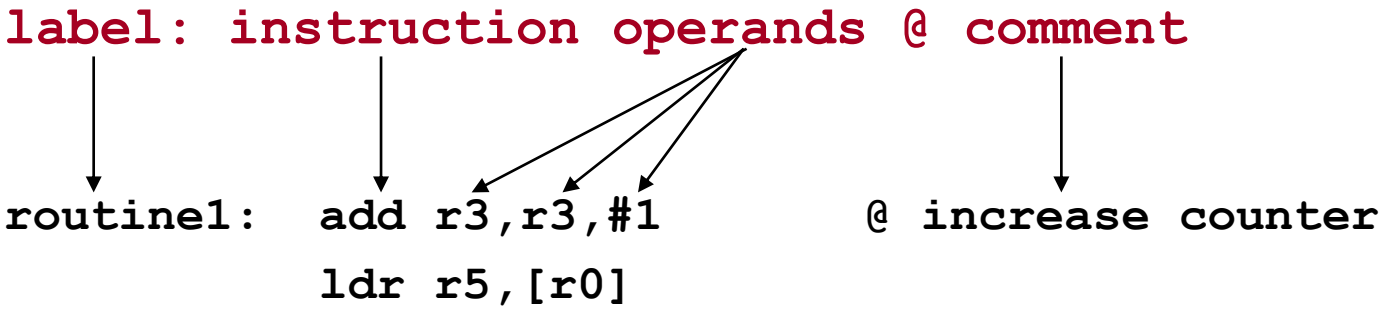
```
add r3,r3,#1
```

- **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:**



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

Operands

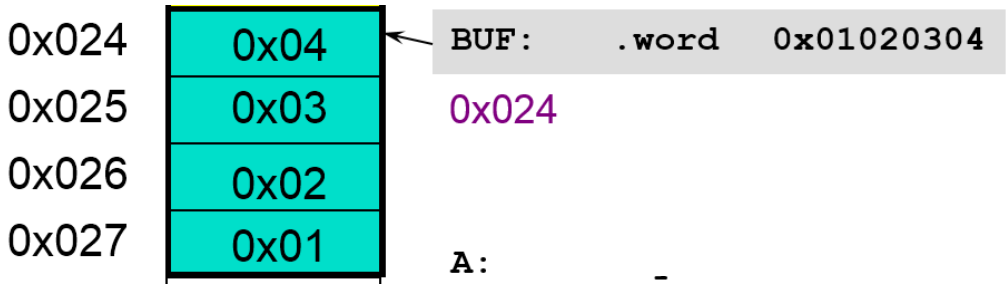
- **can be of 8, 16, 32-bit in length, signed or unsigned in memory**

- **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
```

```
            .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
```

ARM

ASSEMBLY PROGRAMMING

1. part

RA LAB 2.3 Pseudo instructions and directives

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)
- content alignment (16/32bits)
- memory reservation for „variables“
- memory Initialization for „variables“
- end of compilation

```
.text .data
.align
.space
.(h)word, .byte,…
.end
```

Memory segments

Pseudo instrustions 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 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

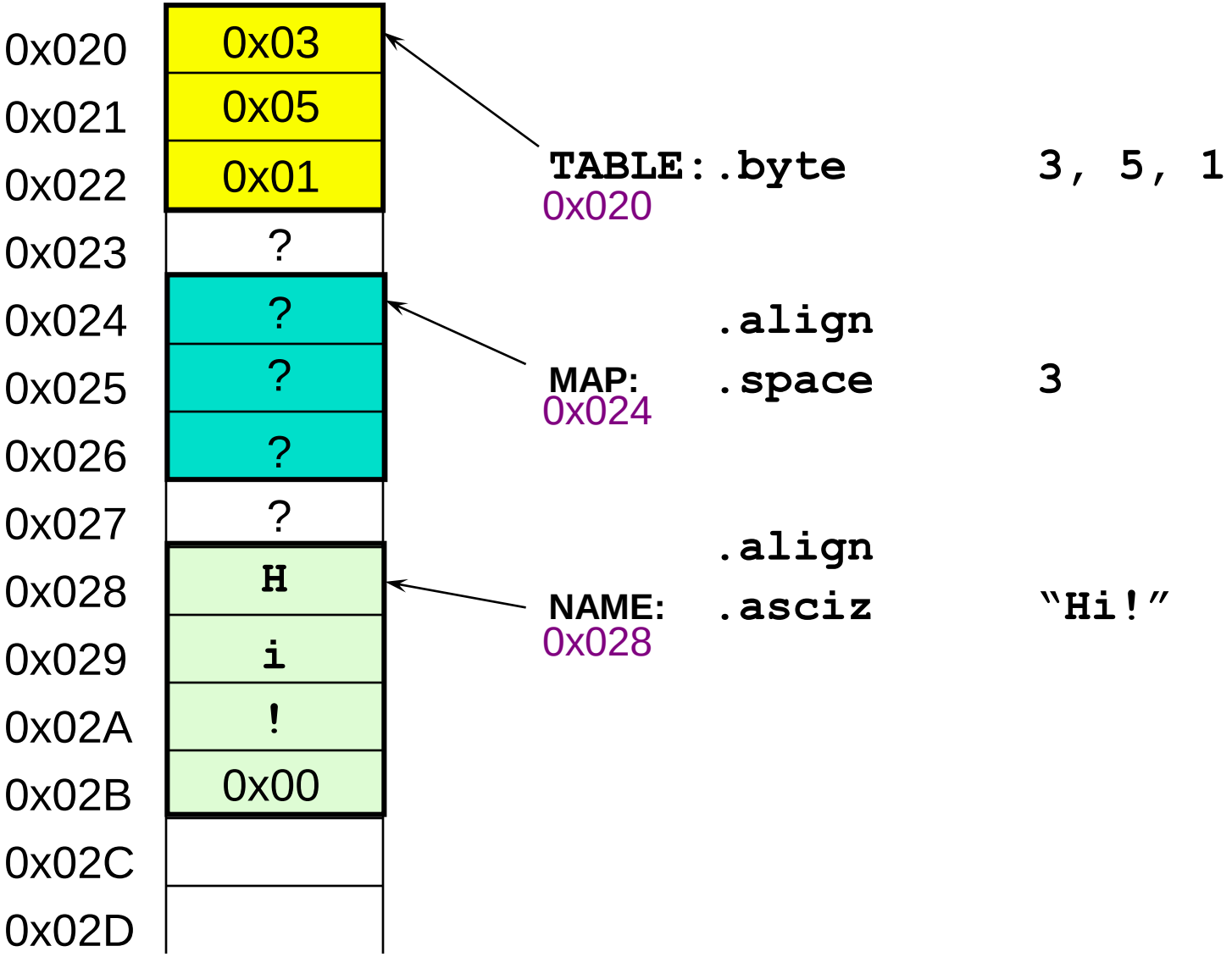
Commonly we want to initalize values.

```
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
```

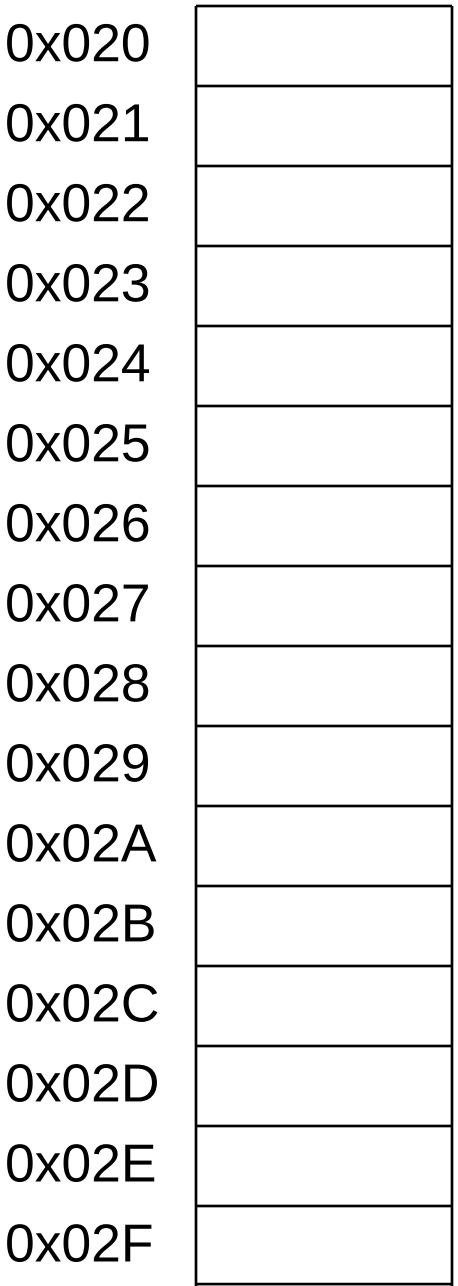
- **„variables“, can be later changed (labels only represent addresses)**
- **We can declare global labels (visible in all files of the project), eg.:**

```
.global str1, str2
```

Summary - pseudo instructions & directives



Summary – compilation of (pseudo) instructions




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

BUF:    .word   0x01020304

A:      .byte   0x15

        .align

_START: mov     r0,#128
```

ASSEMBLER

Location counter

0x020

Labels Table