

*ARM*

*ASSEMBLY PROGRAMMING*

*2. part*

# 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
```

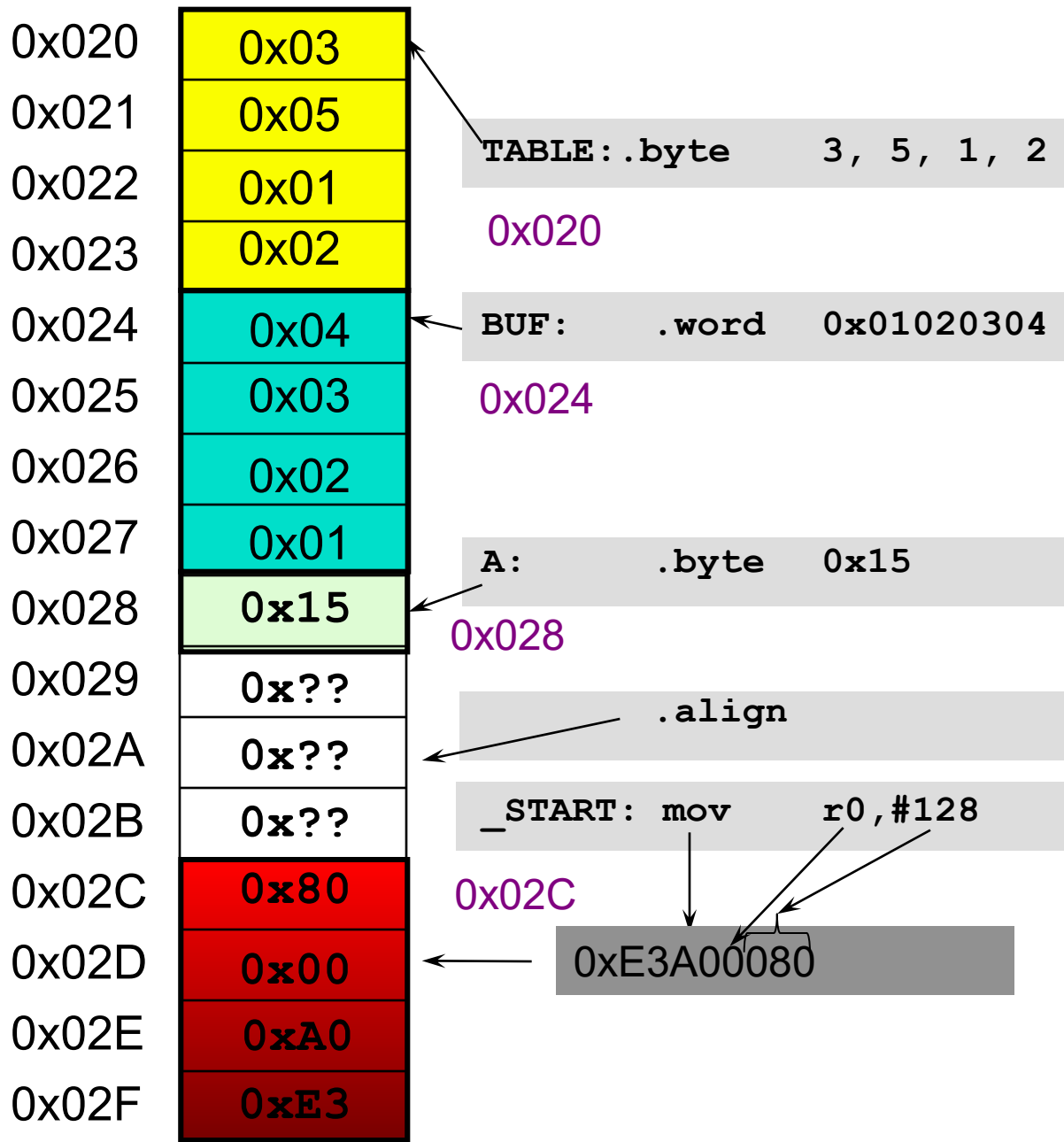
ASSEMBLER

Location counter

0x20

Labels Table


# Summary – compilation of (pseudo) instructions



**ASSEMBLER**

Location counter

0x030

Labels Table

TABLE	0x020
BUF	0x024
A	0x028
_START	0x02C

# Load/store instructions – addressing modes

Assembly 1

```
adr r0, stev1  
ldr r1, [r0]  
adr r0, stev2  
ldr r2, [r0]  
add r3, r2, r1  
adr r0, rez  
str r3, [r0]
```

## 1. Indirect register addressing – base addressing with no offset

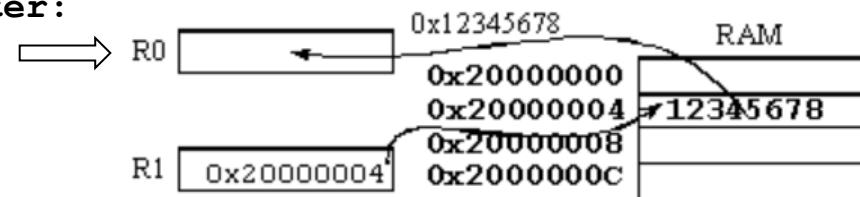
Access to operand in two steps :

a) Variable's address is loaded into a base register with:

```
adr r1, stev1
```

b) Then we use load/store instruction to access memory location on address stored in base register:

```
ldr r0, [r1]    @ r0 <- mem32[r1]  
str r5, [r0]    @ mem32[r0] <- r5
```



Remark:

**adr is pseudo („not real“) instruction.** Assembler replaces it with ALU instruction, that calculates address with R15 (PC) and constant.

Primer:

**adr r0, stev1** assembler replaces with eg. **sub r0, pc, #2c**  
(ALU instructions that puts real address into r0)

# Load/store instructions – addressing modes

## Examples for base indirect addressing mode without offset :

### 32-bit operands

```
adr r1, VAR1           @ r1 <- addr of var. VAR1
ldr r0, [r1]           @ r0 <- mem32[r1]
str r0, [r1]           @ mem32[r1] <- r0
```

### 16-bit operands

```
adr r1, VAR2           @ r1 <- addr of var. VAR2
ldr(s)h r0, [r1]       @ r0 <- mem16[r1]
strh r0, [r1]          @ mem16[r1] <- r0[b0..b15]
```

### 8-bit operands

```
adr r1, VAR3           @ r1 <- addr of var. VAR3
ldr(s)b r0, [r1]       @ r0 <- mem8[r1]
strb r0, [r1]          @ mem8[r1] <- r0[b0..b7]
```

**s** .. operand is signed number !!!

# Load/store instructions – addressing modes

## 2. Indirect register addressing – base addressing with immediate offset (preindex with immediate offset):

```
ldr r0,[r1, #n12]      @ r0<-mem32[r1+n12]
str r0,[r1, #n12]     @ mem32[r1+n12]<-r0
strb r0,[r1, #n12]    @ mem8[r1+n12]<-r0[b0..b7]
```

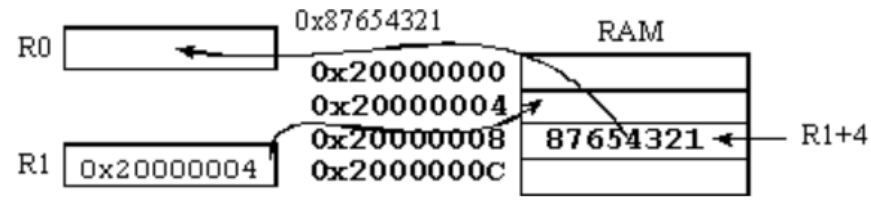
```
ldr(s)b r0,[r1, #n8]   @ r0<-mem8[r1+n8]
ldr(s)h r0,[r1, #n8]  @ r0<-mem16[r1+n8]
strh r0,[r1, #n8]     @ mem16[r1+n8]<-r0[b0..b15]
```

n12 – 12-bit signed offset

n8 – 8-bit signed offset

### Examples:

```
ldr r0, [r1, #4]      @ r0 <- mem32[r1 + 4]
ldr r5, [r0, #-20]   @ r5 <- mem32[r0 - 20]
                    @ r0 must contain proper address!!!
strb r7, [r2,#10]    @ mem8[r2 + 10] <- r7[b0..b7]
                    @ r2 must contain proper address!!!
```

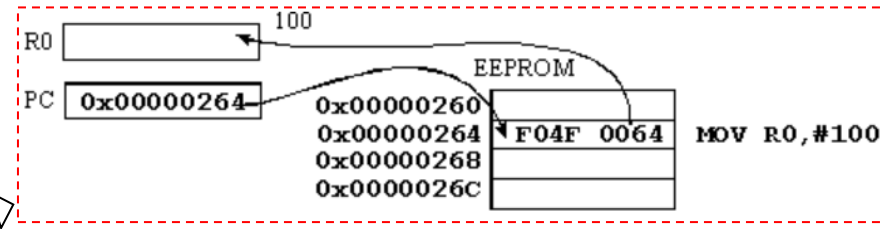


Final address is a sum of values : **base register + signed offset**

# Arithmetic-logic instructions

## 3. Immediate addressing

```
mov r0, #100          @ r0 <- 100
add r2, r7, #0x20     @ r2 <- r7 + 32
sub r5, r5, #1        @ r5 <- r5 - 1
```



$$\text{Immediate operand} = (0..255) * 2^{2*(0..12)}$$

32-bit immediate operand is a 8-bit number, that can be shifted for even number of places inside 32 bit content. A **value of immediate operand cannot be arbitrary!** It's determined by assembler – if it's not possible then we get **warning from assembler.**

Immediate operand is a part of instruction, therefore it must be known before compilation from assembly to machine language. Immediate operands are **constants.**

# Arithmetic-logic instructions (immediate addressing)

## Examples

### Valid immediate operands:

```
mov r1,#255          @ r1 <- 0b00000000000000000000000001111111
add r2,r2,#1024      @ r2 <- r2 + 0b00000000000000000000000100000000
sub r1,r0,#110592    @ r1 <- r0 - 0b00000000000000000110110000000000
```

### Invalid immediate operands :

```
mov r1, #257    @ r1 <- 0b00000000000000000000000100000001
add r7, r3, #65535 r7 <- r3 + 0b00000000000000001111111111111111
```

Immediate operand is **unsigned 8-bit number**, that can be shifted for  $2*n$  bits on left (where  $n$  can be between 0 and 12).



# Arithmetic-logic instructions

## 4. Direct register addressing

- Purpose: for calculation with registers content and movement of the content between registers.

```
add r2, r7, r12
sub r4, r5, r1
mov r1, r4
```

# Arithmetic-logic instructions, list

## • Arithmetic instructions:

```
add r0, r1, r2    @ r0 <- r1 + r2
adc r0, r1, r2    @ r0 <- r1 + r2 + C      (add with C)
sub r0, r1, r2    @ r0 <- r1 - r2
sbc r0, r1, r2    @ r0 <- r1 - r2 + C - 1  (-not(C)=- (1-C)= C-1
rsb r0, r1, r2    @ r0 <- r2 - r1          (reverse subtract)
rsc r0, r1, r2    @ r0 <- r2 - r1 + C - 1  (rev. sub -not(C) )
```

## • Logic instructions:

```
and r0, r1, r2    @ r0 <- r1 AND r2
orr r0, r1, r2    @ r0 <- r1 OR r2
eor r0, r1, r2    @ r0 <- r1 XOR r2
bic r0, r1, r2    @ r0 <- r1 AND NOT r2
```

## • Move register's content:

```
mov r0, r2        @ r0 <- r2
mvn r0, r2        @ r0 <- NOT r2
```

## • Comparisons:

```
cmp r1, r2        @ set CPSR flags on r1 - r2
cmn r1, r2        @ set CPSR flags on r1 + r2
tst r1, r2        @ set CPSR flags on r1 AND r2
teq r1, r2        @ set CPSR flags on r1 XOR r2      (equivalence test)
```