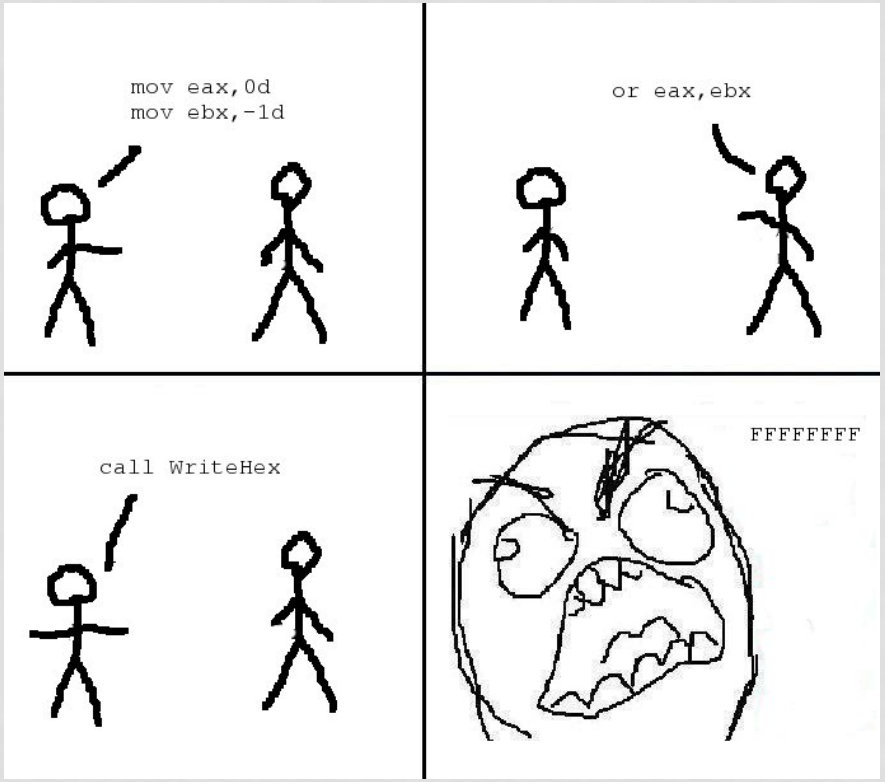


ASSEMBLY & C

WEEK 1-2

AGENDA

week	onderwerp	week	week
1	de structuur van AVR-assembly AVR instructies AVR registers en I/O ATmega memory map Atmel Studio AVR expressies en directives AVR addressing modes	3	de structuur van C-programma's ATMEL studio en AVR libc typen, constanten en operatoren AVR register access in C control statements functies & stackframe visibility scope arrays & strings struct & enum
2	flow of control spring instructies, control structuren Arduino UNO AVR studio stack & subroutines interrupts timer/counters switch bounce	4	interrupts in C TM1638 led&key UART PWM & ADC using a TTC-scheduler state diagram

AGENDA

- **AVR expressions**
- AVR directives
- AVR addressing modes

EXPRESSIES

- AVR instructies hebben 0, 1, of 2 operanden :
  - NOP, RETI (0)
  - RJMP, INC, PUSH (1)
  - ADD, CPI, MOV (2)
- operanden zijn òf registers òf expressies
  - dit geeft de mogelijkheid om expressies i.p.v. getallen te gebruiken
- evaluatie door assembler (***niet*** CPU)
  - zie AVR Assembler User Guide (doc1022.pdf op BB)

VOORBEELDEN

```
; load r26 with lower byte of address represented by
; label + 0xfff0
ldi r26, low(label + 0xfff0)
; load r17 with 1 shifted left 5 times
ldi r17, 1<<5
; load PORTB with bitwise inverse of 5
ldi r18, ~5
out PORTB, r18
; clear or set a single bit
cbr r17, (1 << 3) ; clear bit 3 in r17
sbr r17, (1 << 3) ; set bit 3 in r17
```

EXPRESSIES

- constanten gedefinieerd met .EQU
- integers
  - 255 (decimaal), 0xff of $ff (hex) en 0b11111111 (binair) representeren allemaal dezelfde waarde
- karakters
  - .db 'a','b','c','d','e'
- strings
  - .db "abcde"

EXPRESSIES

- functies
  - voorbeeld : HIGH en LOW
- operatoren
  - de bekende Java en C operatoren zijn beschikbaar
  - rekenkundige operatoren: +,-,*,/
  - relationele operatoren: >, >=,<,<=,==,!=
  - logische operatoren: &&, | |, !
  - bitwise operatoren: >>, <<, &, |, ~, ^

BITWISE OPERATOREN

Symbol	Operator
&	bitwise AND
|	bitwise inclusive OR
^	bitwise exclusive OR
<<	left shift
>>	right shift
~	one's complement (unary)

wiki "bitwise operations in C"

AGENDA

- AVR expressies
- **AVR directives**
- AVR addressing modes

DIRECTIVES

- zie de AVR Assembler User Guide
- directives zijn geen instructies voor CPU maar voor assembler:
  - het geheugensegment kiezen : .cseg, .dseg
  - de plaats in het geheugensegment te bepalen : .org
  - symbolen te definiëren : .equ, .set, .def, .undef
  - include files opnemen in source file : .include
  - reserveren van geheugen :
    - in het code segment : .db, .dw, .dd, .dq
    - in het data segment : .byte

4.5 Assembler directives

The Assembler supports a number of directives. The directives are not translated directly into opcodes. Instead, they are used to adjust the location of the program in memory, define macros, initialize memory and so on. An overview of the directives is given in the following table.

Summary of directives:

Directive	Description
BYTE	Reserve byte to a variable
CSEG	Code Segment
DB	Define constant byte(s)
DEF	Define a symbolic name on a register
DEVICE	Define which device to assemble for
DSEG	Data Segment
DW	Define constant word(s)
ENDMACRO	End macro
EQU	Set a symbol equal to an expression
ESEG	EEPROM Segment
EXIT	Exit from file
INCLUDE	Read source from another file
LIST	Turn listfile generation on
LISTMAC	Turn macro expansion on
MACRO	Begin macro
NOLIST	Turn listfile generation off
ORG	Set program origin
SET	Set a symbol to an expression

Note: *All directives must be preceded by a period.*

uit AVR Assembler User Guide

SYMBOLEN DEFINIËREN

- .def : geef een naam aan een register
  - .def temp=r16
  - .def counter=r17

- .equ : geef een naam aan een expressie

```
.equ io_offset = 0x23
.equ porta = io_offset + 2
```

GEHEUGEN BEPALEN

- segment directives bepalen actieve geheugen
  - .cseg – code segment (flash)
    - = is default segment
  - .dseg – data segment (SRAM)
- .org x betekent : zet de volgende instructie op adres x
  - default origin voor code segment = $0000
  - default origin voor data segment = $0060

GEHEUGEN ALLOCEREN

- code segment : constanten definiëren
  - .DB, .DW.
- data segment : ruimte reserveren
  - .BYTE

VOORBEELD

```
.include "m328Pdef.inc"
; default is cseg at address 0
.def counter=r16        ;r16 will hold counter value
.def temp=r17           ;r17 is a temporary register

ldi temp,0xff           ;configure PORTB as output
out DDRB,temp
ldi counter,0x00        ;init counter

loop:
   out PORTB,counter    ;put counter value on PORTB
   inc counter          ;increment counter
   rjmp loop            ;repeat forever
```

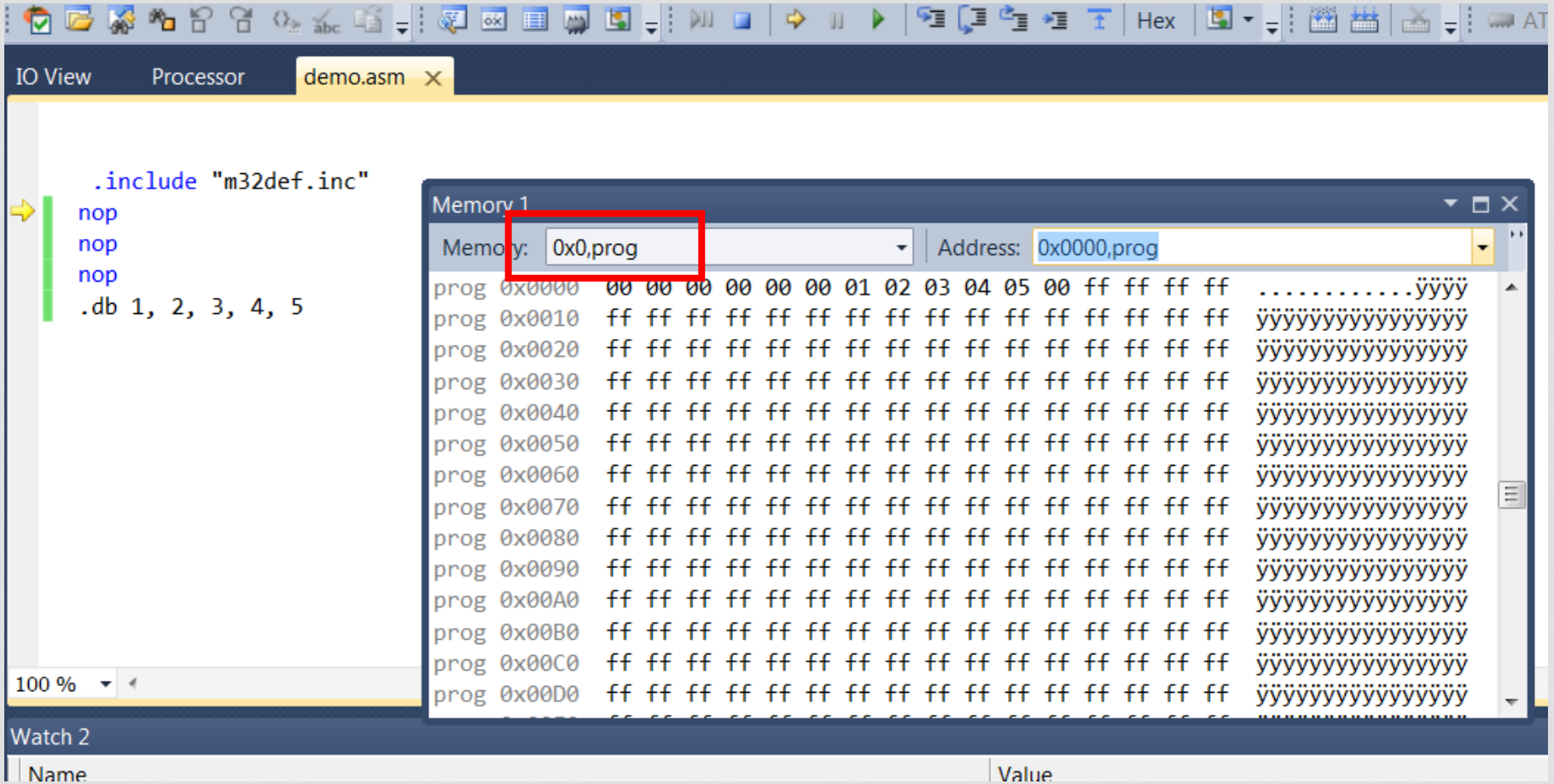
VOORBEELD

```
.dseg
.org 0x37              ; ga naar adres 0x37
a: .byte 4             ; reserveer voor var a 4 bytes
table: .byte 10        ; reserveer voor var table 10 bytes


.cseg
.org 0x100               ; ga naar adres 0x100
s: .db "pietje puk"      ; definieer string s="pietje puk"
str: .db 'a', "abc"      ; definieer string str="aabc"
c: .dw 9001              ; definieer constante c=9001
```

ARRAYS

- arrays zonder initialisatie kunnen in SRAM
- Arrays met initialisatie
  - Kan in SRAM maar is dan vluchtig
  - In flash is dit de enige mogelijkheid

```
.dseg
arr1: .byte size*elem_size

.cseg
arr2: .db 1,2,3,4,5
```



opcode NOP = 0x0000

AGENDA

- AVR expressions
- AVR directives
- **AVR addressing modes**

ADRESSERING MODES

- waar zijn de operanden ?
  - gegeven in de instructie zelf
  - in een CPU register
  - in een I/O poort (register buiten CPU)
  - in het geheugen (data of code segment)
- Java : simple & reference types
- c : base types & pointer types

IMMEDIATE ADDRESSING

- immediate (bijv. 0x80)

```
ldi r1, 0x80   ; r1 = 0x80
```

DIRECT ADDRESSING

- 3 vormen van direct addressing :
  - register
  - I/O
  - (data)memory

REGISTER & I/O DIRECT

- register direct (bijv. r16)

```
inc r16         ; r16++
add r1, r2      ; r1 = r1 + r2
mov r1, r2      ; r1 = r2
```

- I/O direct (bijv. portb)

```
in r16, pinb     ; r16 = portb
out portb, r16   ; portb = r16
```

REGISTER DIRECT

- meest gebruikte mode door compilers
  - operaties met CPU registers zijn snel
- compiler : optimalisatie door meest gebruikte variabelen in een register te plaatsen
  - bijvoorbeeld een index variabele in een lus

```
for (i=0; i<100; i++) {
   ...
}
```

MEMORY DIRECT

- adres opgeven, bijv. 0x0060

	Operation:	
(i)	Rd ← (k)	
	Syntax:	**Operands:**
(i)	LDS Rd,k	$0 \le d \le 31, 0 \le k \le 65535$

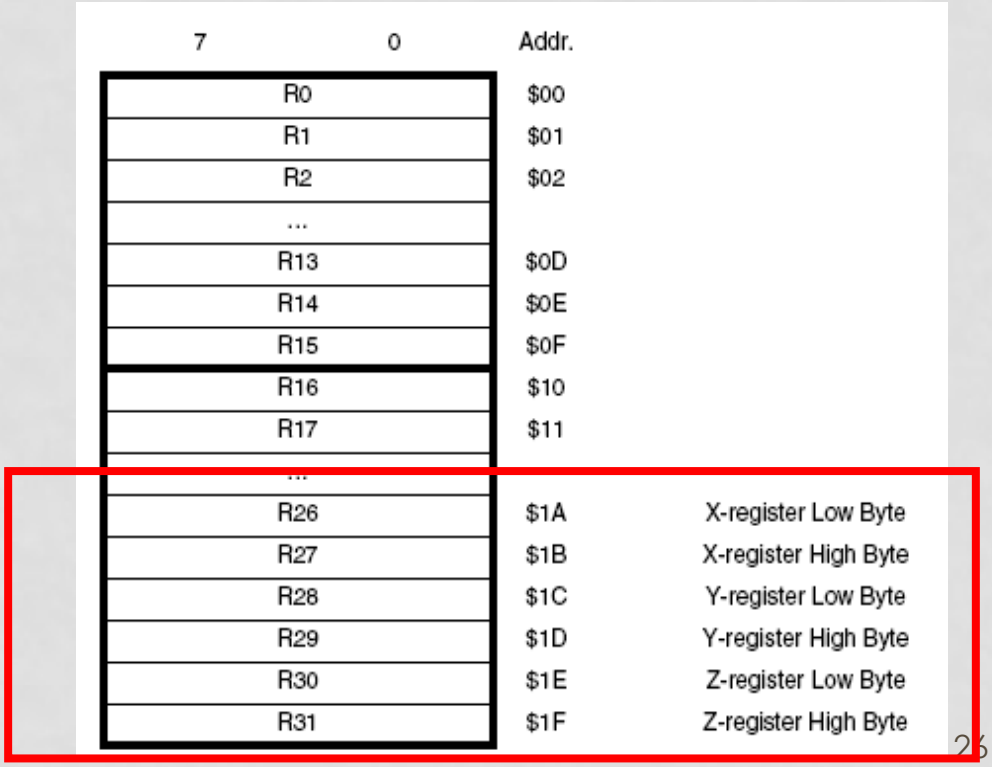
lds, sts : load/store direct from/to data space

	Operation:	
(i)	(k) ← Rr	
	Syntax:	**Operands:**
(i)	STS k,Rr	$0 \le r \le 31, 0 \le k \le 65535$

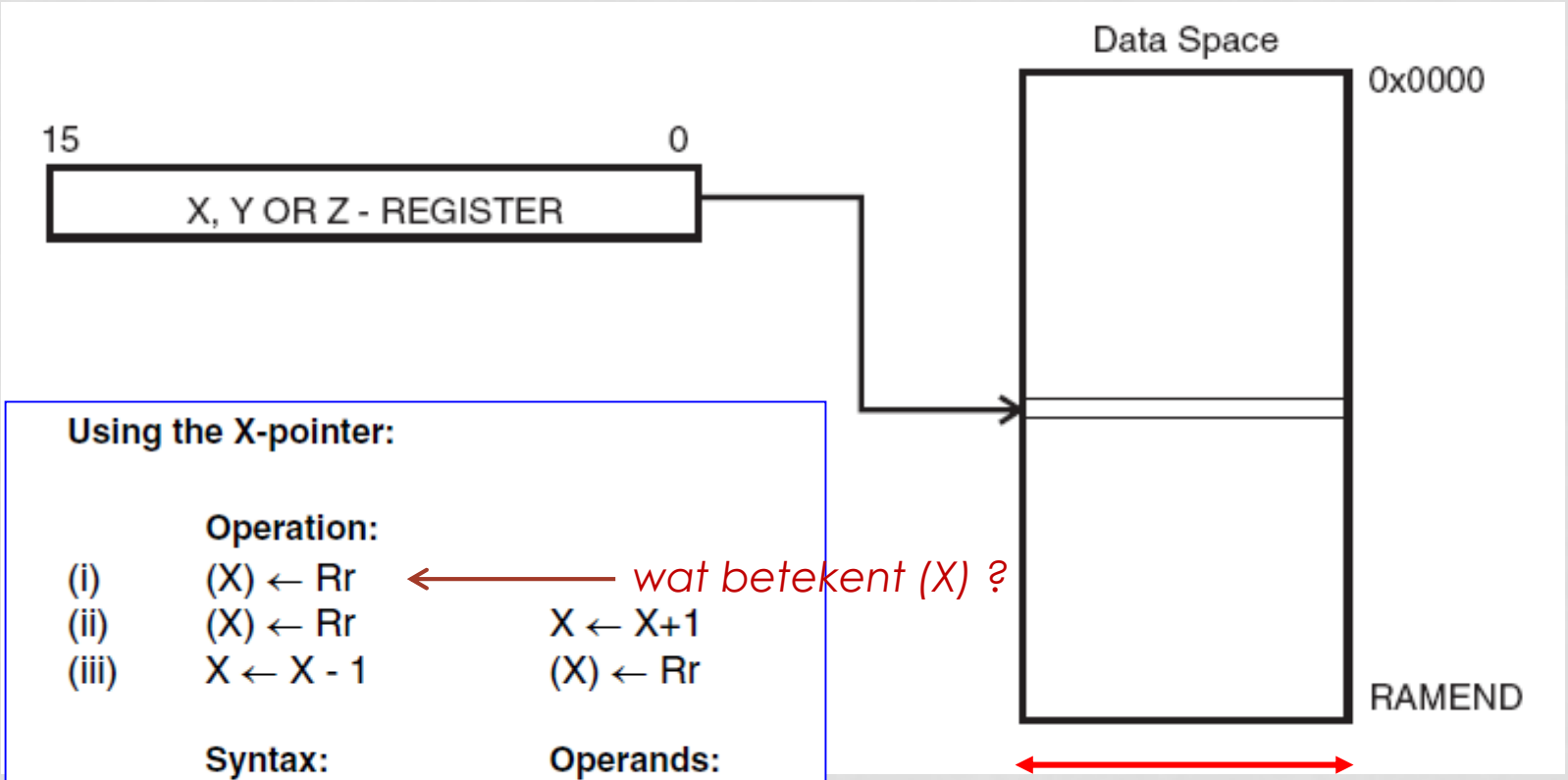
```
ldi r16, 5          ; r16=5
sts 0x0100, r16     ; (0x0100) = r16
lds r1, 0x0100      ; r1 = (0x0100)
add r1, r16         ; r1 = 5+5
sts 0x0100, r1      ; (0x0100) = r1
```

INDIRECT ADDRESSING

- indirecte adressering met adres pointers
- R26..R31 : pointer registers X,Y,Z
  - X : R26:R27
  - Y : R28:R29
  - Z : R30:R31



INDIRECT ADDRESSING (DATA MEMORY)



Using the X-pointer:

	Operation:	
(i)	(X) ← Rr	
(ii)	(X) ← Rr	X ← X+1
(iii)	X ← X - 1	(X) ← Rr

wat betekent (X) ?

	Syntax:	Operands:
(i)	ST X, Rr	$0 \le r \le 31$
(ii)	ST X+, Rr	$0 \le r \le 31$
(iii)	ST -X, Rr	$0 \le r \le 31$

INDIRECT ADDRESSING (DATA MEMORY)

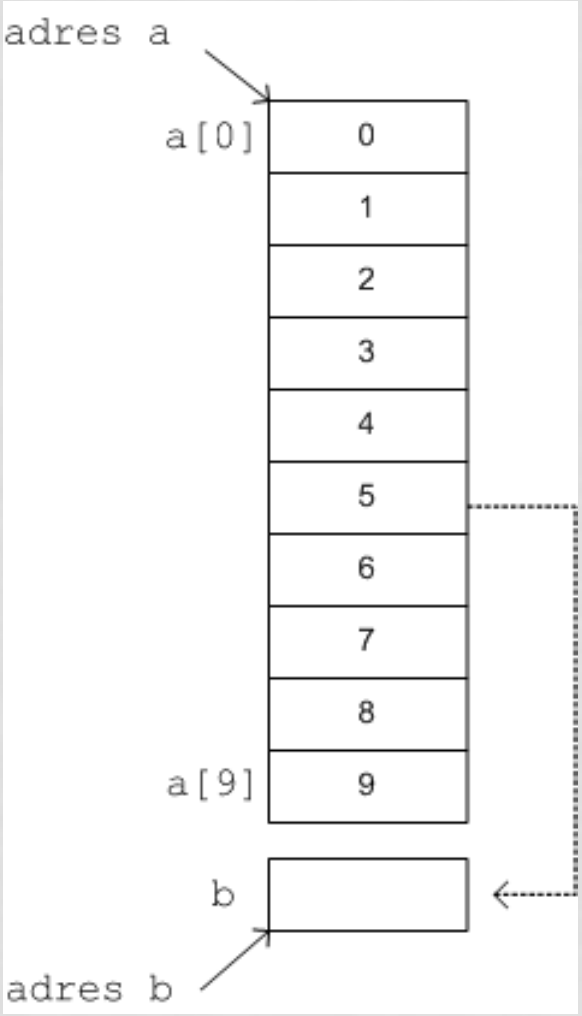
```
.def i = r17
ldi i, 1  ; i = 1
; setup pointer x to 0x0100
ldi xl, 0x00
ldi xh, 0x01
st x+, i  ; (0x0100) = 1; x++
inc i     ; i++
st x+, i  ; (0x0101) = 2; x++
```

ld, st : load/store indirect from/to data space

INDIRECT ADDRESSING (DATA MEMORY)

C- of (ongeveer) Java-code :

```
char b, a[10];
for (i=0; i<10; i++)
   a[i] = i;
b = a[5];
```



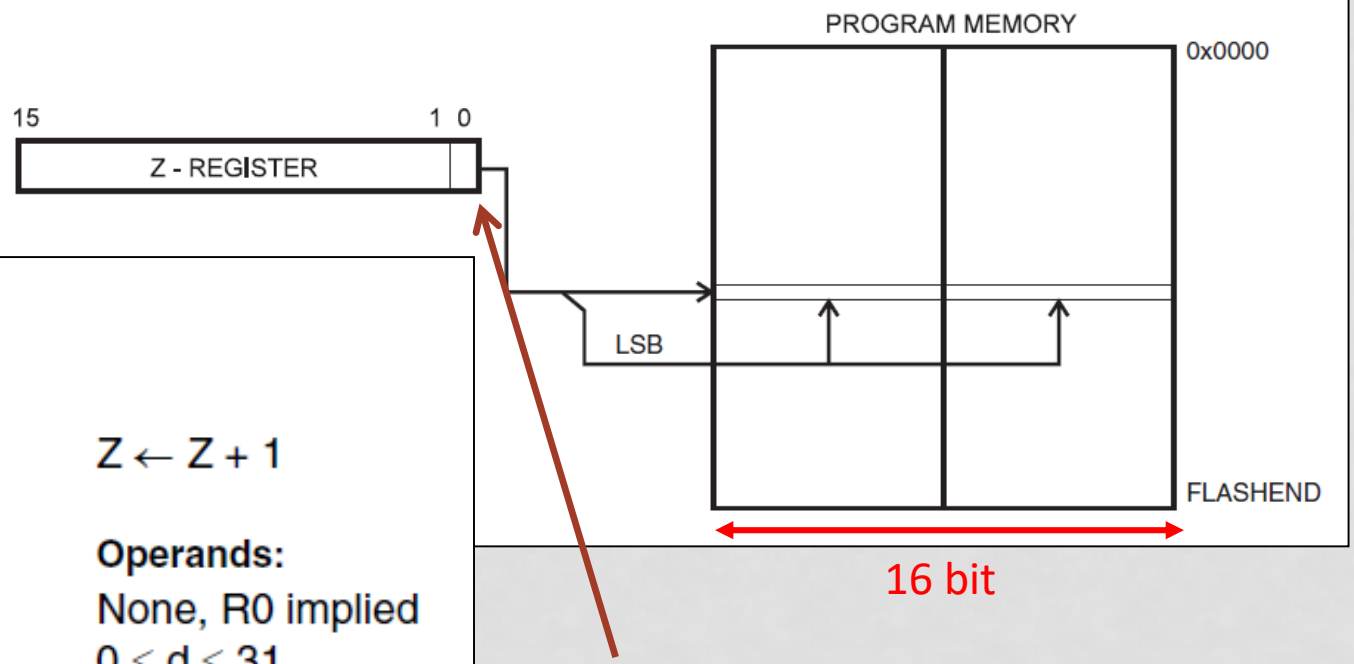
INDIRECT ADDRESSING (DATA MEMORY)

```
char b, a[10];
for (i=0; i<10; i++)
       a[i] = i;
b = a[5];
```

```
.dseg
.org 0x0100
a: .byte 10 ;array op adres a
b: .byte 1  ;char op adres b

.cseg
;pointer x wijst naar adres a
ldi xh, high(a)
ldi xl, low(a)

;hier moet nog de for-lus komen

; code voor b = a[5]
; pointer x = x + 5
ldi r16, 5
add xl, r16 ;x wijst nu naar a[5]
brcc PC+2   ;xh verhogen ? check carry bit
inc xh

ld r16, x  ;r16=(x) oftewel r16=a[5]
sts b, r16 ;(b)=r16
```

INDIRECT ADDRESSING (PROGRAM MEMORY)

Figure 9. Program Memory Constant Addressing



	Operation:	
(i)	R0 ← (Z)	
(ii)	Rd ← (Z)	
(iii)	Rd ← (Z)	Z ← Z + 1

	Syntax:	Operands:
(i)	LPM	None, R0 implied
(ii)	LPM Rd, Z	0 ≤ d ≤ 31
(iii)	LPM Rd, Z+	0 ≤ d ≤ 31

-program memory bestaat uit 16-bit woorden
-registers r0..r31 zijn 8 bit
-voor het adres hebben we max. 14 bits nodig
lsb van Z-pointer selecteert higher of lower byte
(0 = lower byte)
=> daarom adres ***vermenigvuldigen met 2 (= 0 inschuiven)***
=> Z++ wijst naar volgende ***byte***

INDIRECT ADDRESSING (PROGRAM MEMORY)

```
;init Z pointer, table = startadres
ldi ZH, high(2*table)
ldi ZL, low(2*table)

lpm r16, Z ;r16 = (Z) (lower byte)
```

Figure 9. Program Memory Constant Addressing