Data in Program Memory/Data Memory

- Constant Data can be placed in Program Memory
  - Table used for conversion
  - Strings used in GUI
- Access is very similar, and not very intuitive…
- Access to data memory is simpler…
- You will need to do all of the above in Lab3!

Storing fixed data in flash memory

DATA1: .DB 28 ;DECIMAL(1C in hex)

DATA2: .DB 0b00110101 ;BINARY (35 in hex)

DATA3: .DB 0x39 ;HEX

DATA4: .DB 'Y' ;single ASCII char

DATA4: .DB ‘1’, ‘2’, ‘3’ ;ASCII chars

DATA6: .DB "Hello ALI" ;9 ASCII chars (string)

…combination:

.DATA7: .DB ’H’, ‘E’, ‘J’, "Magnus”, 15

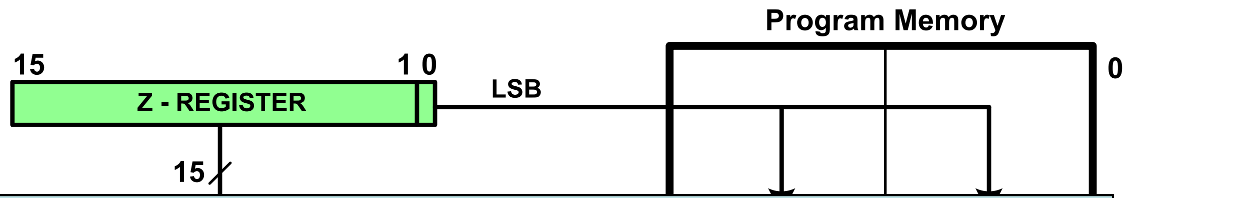
General-Purpose Registers in AVR – Some are also “special purpose”!

Address		
0x00	r0	
0x01	r1	
	• • •	
0x1A	r26	x register low byte
0x1B	r27	x register high byte
0x1C	r28	y register low byte
0x1D	r29	y register high byte
0x1E	r30	z register low byte
0x1F	r31	z register high byte

Storing fixed data in flash memory

- LPM Rd, Z

  LPM R15, Z



Low	High	Address
0000 0000 0000 0000	0000 0000 0000 0001	0000 0000 0000 0000
0000 0000 0000 0010	0000 0000 0000 0011	0000 0000 0000 0001
0000 0000 0000 0100	0000 0000 0000 0101	0000 0000 0000 0010
0000 0000 0000 0110	0000 0000 0000 0111	0000 0000 0000 0011
0000 0000 0000 1000	0000 0000 0000 1001	0000 0000 0000 0100
0000 0000 0000 1010	0000 0000 0000 1011	0000 0000 0000 0101
⋮	⋮	
1111 1111 1111 1100	1111 1111 1111 1101	0111 1111 1111 1110
1111 1111 1111 1110	1111 1111 1111 1111	0111 1111 1111 1111

Low	High	Address
$0000	$0001	$0000
$0002	$0003	$0001
$0004	$0005	$0002
$0006	$0007	$0003
$0008	$0009	$0004
$000A	$000B	$0005
$FFFC	$FFFD	$7FFE
$FFFE	$FFFF	$7FFF

This is important and the basis for storing constant strings!

Program memory (aka Flash) as 2-dim array

Word-addr	High byte	Low byte	Byte-addr
0000	‘B’	‘A’	0000
0001	‘D’	‘C’	0002
0002	‘F’	‘E’	0004
0003	‘H’	‘G’	0006
0004	‘J’	‘I’	0008
0005	‘L'	‘K’	000A

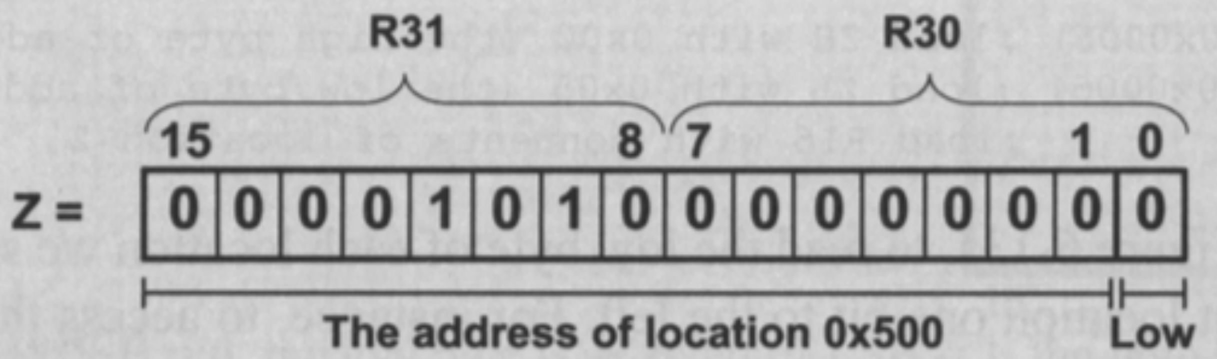
‘K’	‘I’	‘G’	‘E’	‘C’	‘A’	0
‘L’	‘J’	‘H’	‘F’	‘D’	‘B’	1
5	4	3	2	1	0	

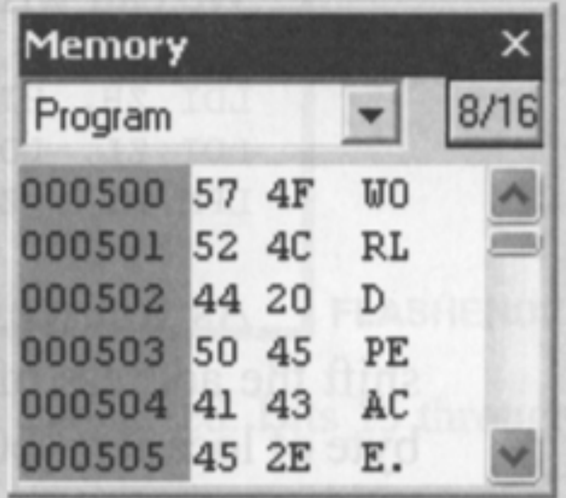
Mem

Memory content can be seen as a 2-dimensional array referenced through: “Mem”

String in PM – ”memory dump”

```
        .ORG   $500   ;data is burned into program space starting at $500
MYDATA:  .DB "WORLD PEACE."
```





Solution:

In the above program, ROM locations $500–$505 have the following contents.

```
$500  (Low byte)  =  ('W')        $500  (High byte)  =  ('O')
$501  (Low byte)  =  ('R')        $501  (High byte)  =  ('L')
$502  (Low byte)  =  ('D')        $502  (High byte)  =  (' ')
$503  (Low byte)  =  ('P')        $503  (High byte)  =  ('E')
$504  (Low byte)  =  ('A')        $504  (High byte)  =  ('C')
$505  (Low byte)  =  ('E')        $505  (High byte)  =  ('.')
```

Examples

Some Assembler directives

	Example
+	LDI R20,5+3 ;LDI R20,8
-	LDI R30,9-3 ;LDI R30,6
*	LDI R25,5*7 ;LDI R25,35
/	LDI R19,8/2 ;LDI R19,4

	Example
&	LDI R20,0x50&0x10 ;LDI R20,0x10
|	LDI R25,0x50|0x1 ;LDI R25,0x51
^	LDI R23,0x50^0x10 ;LDI R23,0x40

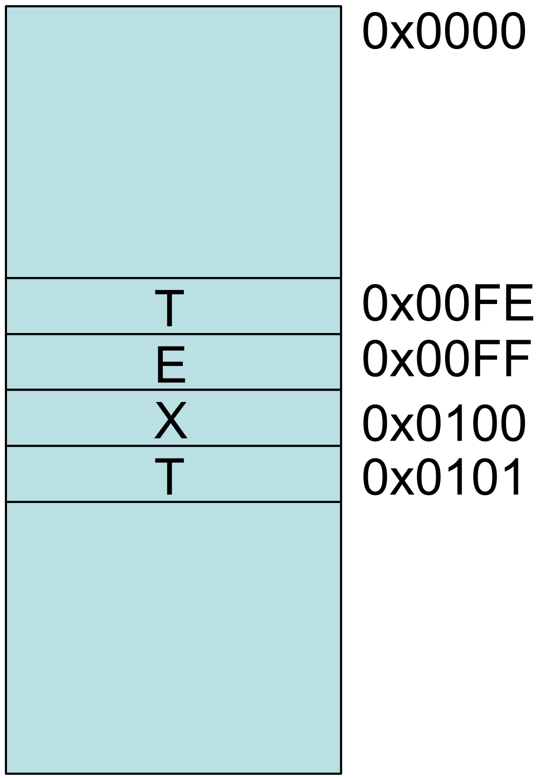
	Example
<<	LDI R16, 0x10<<1 ;LDI R16,0x20
>>	LDI R16, 0x8 >>2 ;LDI R16,0x2

Mapping table example:

Write a program that gets an X-value from Port B and sends X^2 to Port C. Assume that PB0-PB3 contains a number 0-9. Use a look-up table instead of the MUL instruction.

```
.ORG 300
XSQR_TABLE: .DB 0, 1, 4, 9, 16, 25, 36, 49, 64, 81 ;0-9 sqr
<init section removed..>
Calc_sqr:
        LDI     ZL, LOW(XSQR_TABLE<<1)
        LDI     ZH, HIGH(XSQR_TABLE<<1)
        IN      R16, PINB
        ANDI    R16, 0x0F       ; Mask off high bits)
        ADD     ZL, R16         ; offset in table
        LPM     R16, Z          ; get SQR-value
        OUT     PORT C, R16
```

Access across ”byte border”…



- Data may be placed across a ”byte border”. When adjusting for an offset, the high byte may need to be updated

- Code example:

```
LDI     ZL, LOW(map_table <<1)
LDI     ZH, HIGH(map_table <<1)
ADD     ZL, INDEX          ; index = offset
LDI     INDEX, 0x00        ; Add 0 to catch…
ADC     ZH, INDEX          ; …carry, if present
LPM     R16, Z             ; get data at offset
```

Atmel example

```
LDI ZH, high(Table_1<<1) ;Initialize Z pointer
LDI ZL, low(Table_1<<1)

      LPM R16, Z                    ; Load constant from
                                    ; program memory
                                    ; pointed to by Z
                                    ; (R31:R30)
...
Table_1:
.DW 0x5876                          ; 0x76 is addressed
                                    ; when ZLSB = 0
                                    ; 0x58 is addressed
                                    ; when ZLSB = 1
...
```

Example

- Analyze the following program; then rewrite it using Z+

```
.ORG    PM_START                        ;store into PM starting at “PM_START
        LDI     R20,0xFF
        OUT     DDRB,R20                ;make PB an output
        LDI     ZL, LOW(MYDATA<<1)      ;ZL = 0  look-up table low-byte addr
        LDI     ZH,HIGH(MYDATA<<1)      ;ZH = 0A look-up table high-byte addr
        LPM     R20,Z
        OUT     PORTB,R20
        INC     ZL
        LPM     R20,Z
        OUT     PORTB,R20
        INC     ZL
        LPM     R20,Z           ;load R20 with 'A' char pointed to by Z
        OUT     PORTB,R20       ;send it to Port B
HERE:   RJMP    HERE            ;stay here forever
;data is stored into PM space starting at $500
        .ORG    $500
MYDATA: .DB     "USA"
```

```
LPM     R20,Z+
OUT     PORTB,R20

LPM     R20,Z+
OUT     PORTB,R20
```

Lower-Case to Upper-Case (1/3)

```
.EQU   size          = 5    ; nmbr of chars in “hello”
.DEF   counter       = R17
; ------------------------------------
.DSEG                       ; RAM
.ORG   0x100                ; Set starting
                            ; address of data
                            ; segment to 0x100
Cap_string: .BYTE 5         ; space for 5 bytes
; ------------------------------------
.CSEG                       ; FLASH
Low_string: .DB "hello”
```

Lower-Case to Upper-Case (2/3)

```
; Register Z points to Flash
LDI ZL, LOW(Low_string<<1)          ; Get low byte of
                                    ; address of "h"
LDI ZH, HIGH(Low_string<<1)         ; Get high byte of
                                    ; address of "h"
; ------------------------------------------
; Register Y points to RAM
LDI YH, HIGH(Cap_string)
LDI YL, LOW(Cap_string)
; ------------------------------------------
CLR counter                         ; counter=0
```

Lower-Case to Upper-Case (3/3)

```
main:
      LPM  R20, z+             ; Load a letter from
                               ; flash memory
      SUBI R20, 32             ; Convert it to a
                               ; capital letter
      ST   y+,R20              ; Store the capital
                               ; letter in SRAM
      INC  counter
      CPI  counter, size
      BRLT main                ; counter < size?
loop: NOP                      ; done, "stop"
      RJMP loop
```

Example – copy string from program memory to RAM (1/2)

“Assume that ROM space starting at $500 contains the message “The Promise of World Peace”. Write a program to bring it into CPU one byte at a time and place the bytes in RAM locations starting at $140.”

Example – copy string from program memory to RAM (2/2)

```
.CSEG
.ORG  PM_START                              ;place in PM starting at "PM_START"
          LDI     ZL, LOW(MYDATA<<1)        ;R30 = 00 low-byte addr
          LDI     ZH, HIGH(MYDATA<<1)       ;R31 = 0A, high-byte addr
          LDI     XL, LOW(0x140)            ;R26 = 40, low-byte RAM address
          LDI     XH, HIGH(0x140)           ;R27 = 1, high-byte RAM address

AGAIN:    LPM     R16, Z+                   ;read the table, then increment Z
          CPI     R16,0                     ;compare R16 with 0
          BREQ    END                       ;exit if end of string
          ST      X+, R16                   ;store R16 in RAM and inc X
          RJMP    AGAIN
END:      RJMP    END

.ORG      0x500                             ;data starting at 0x500 (word-count)
MYDATA: .DB "The Promise of World Peace",0
```

X,Y,Z registers as pointers to Data Memory

The 16-bit registers X, Y, and Z are widely used as pointers. We can use them with the LD instruction to read the value of a location pointed to by these registers. For example, the following instruction reads the value of the location pointed to by the X pointer.

```
LD     R24, X         ;load into R24 from location pointed to by X
```

For instance, the following program loads the contents of location 0x130 into R18:

```
LDI    XL, 0x30       ;load R26 (the low byte of X) with 0x30
LDI    XH, 0x01       ;load R27 (the high byte of X) with 0x1
LD     R18, X         ;copy the contents of location 0x130 to R18
```

The above program loads 0x130 into the X register; this is done by loading 0x30 into R26 (the low byte of X) and 0x1 into R27 (the high byte of X). Then it loads R18 with the contents of the location to which X points. See Figure 6-7.

The ST instruction can be used to write a value to a location to which any of the X, Y, and Z registers points. For example, the following program stores the contents of R23 into location 0x139F:

```
LDI    ZL, 0x9F       ;load 0x9F into the low byte of Z
LDI    ZH, 0x13       ;load 0x13 into the high byte of Z (Z=0x139F)
ST     X, R23         ;store the contents of location 0x139F in R23
```

Example 6-5

Write a program to copy the value $55 into memory locations $140 through $144 using
(a) direct addressing mode,
(b) register indirect addressing mode without a loop, and
(c) a loop.

Solution:

(a)
```
LDI   R17,0x55          ;load R17 with value 0x55
STS   0x140,R17         ;copy R17 to memory location 0x140
STS   0x141,R17         ;copy R17 to memory location 0x141
STS   0x142,R17         ;copy R17 to memory location 0x142
STS   0x143,R17         ;copy R17 to memory location 0x143
STS   0x144,R17         ;copy R17 to memory location 0x144
```

(b)
```
LDI   R16,0x55    ;load R16 with value 0x55
LDI   YL,0x40     ;load R28 with value 0x40 (low byte of addr.)
LDI   YH,0x1      ;load R29 with value 0x1 (high byte of addr.)
ST    Y,R16       ;copy R16 to memory location 0x140
INC   YL          ;increment the low byte of Y
ST    Y,R16       ;copy R16 to memory location 0x141
INC   YL          ;increment the pointer
ST    Y,R16       ;copy R16 to memory location 0x142
INC   YL          ;increment the pointer
ST    Y,R16       ;copy R16 to memory location 0x143
INC   YL          ;increment the pointer
ST    Y,R16       ;copy R16 to memory location 0x144
```

Example (1/2) – not optimised

Write a program to copy the value $55 into memory locations $140 to $144

```
      LDI  R19,0x5          ;R19 = 5 (R19 for counter)
      LDI  R16,0x55         ;load R16 with value 0x55 (value to be copied)
      LDI  YL,0x40          LDI  YL,LOW(0x140)
      LDI  YH,0x1           LDI  YH,HIGH(0x140)
  L1: ST   Y,R16            ;copy R16 to memory location 0x140
      INC  YL               ;increment the low byte of Y
      DEC  R19              ;decrement the counter
      BRNE L1               ;loop until counter = zero
```

Example (2/2) – result after optimisation…

- Write a program to copy the value $55 into memory locations $140 to $444

```
     LDI R19,0x5          ;R19 = 5 (R19 for counter)
     LDI R16,0x55         ;load R16 with value 0x55 (value to be copied)
     LDI YL,LOW($140)     ;load the low byte of Y with value 0x40
     LDI YH,HIGH($140)    ;load the high byte of Y with value 0x1
L1:  ST  Y+,R16           ;copy R16 to memory location Y
     DEC R19              ;decrement the counter
     BRNE  L1             ;loop until counter = zero
```

IF-statements in Assembler (C-complier…)

```
      if (PINA==0x51) {
98:   89 b3          IN    R24, 0x19     ; 25
9a:   81 35          CPI   R24, 0x51     ; 81
9c:   19 f4          BRNE .+6            ; 0xa4
        PORTB=0xFF;
9e:   8f ef          LDI   R24, 0xFF     ; 255
a0:   88 bb          OUT   0x18, R24     ; 24
a2:   01 c0          RJMP .+2            ; 0xa6
      }
      else {
        PORTB=0x00;
a4:   18 ba          OUT   0x18, R1      ; 24
      }l
```

AVR Addressing Modes

- In assembler, the addressing modes are as important as the different instructions that are available
- Since the Atmega 32U4 uses the Harvard architecture, we need to be careful with the addressing modes, regarding:
  – Are we addressing the program memory?
  – Are we addressing the data memory?

Single Register Addressing Mode

- Single Register Addressing Mode
  - INC Rd

```
INC R19
```

  - DEC Rd

```
DEC R23
```

;R23 = R23 – 1

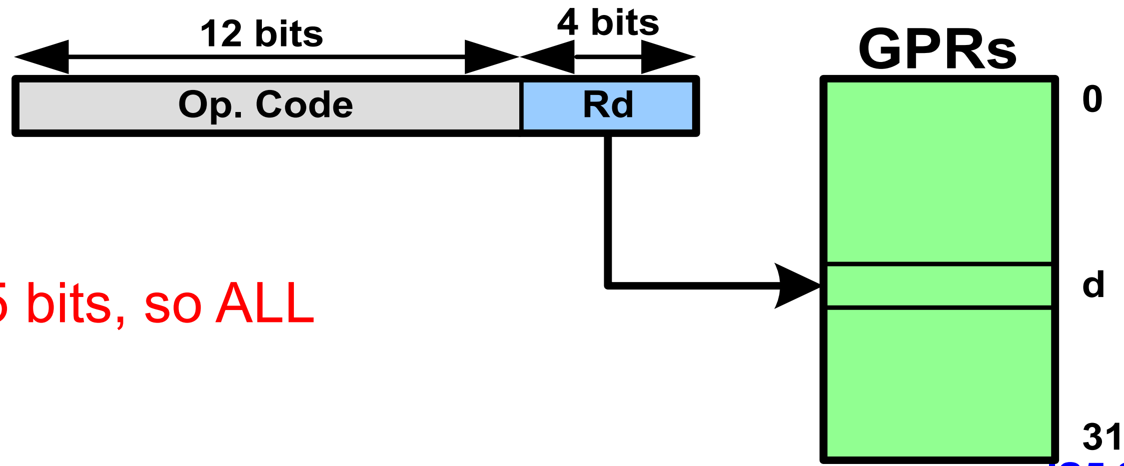
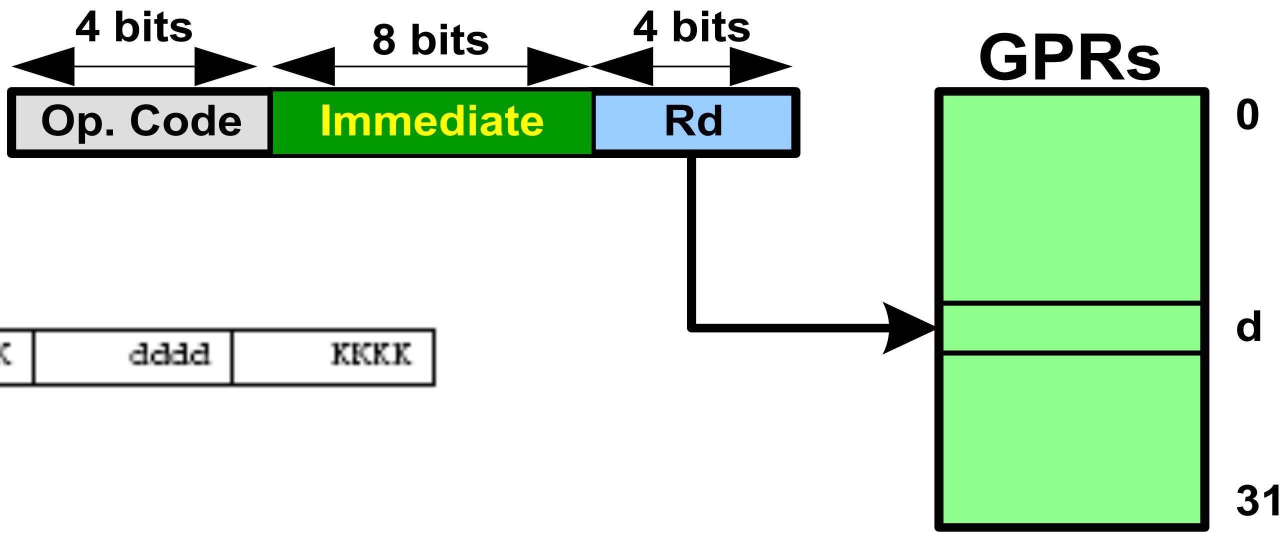


Figure is Not correct! Rd is 5 bits, so ALL registers can be used.

Immediate Addressing Mode (Single register with immediate)



- LDI Rd,K

  `LDI R19,25`

1110	KKKK	dddd	KKKK

- SUBI Rd,K

  `SUBI R23,5`   ;R23 = R23 – 5

0101	KKKK	dddd	KKKK

- ANDI Rd,K

  `ANDI R21,0x15`

0111	KKKK	dddd	KKKK

Rd is only 4 bits, so only 16 registers can be used. This is the reason for the limitation to R16-R31 that we have seen…

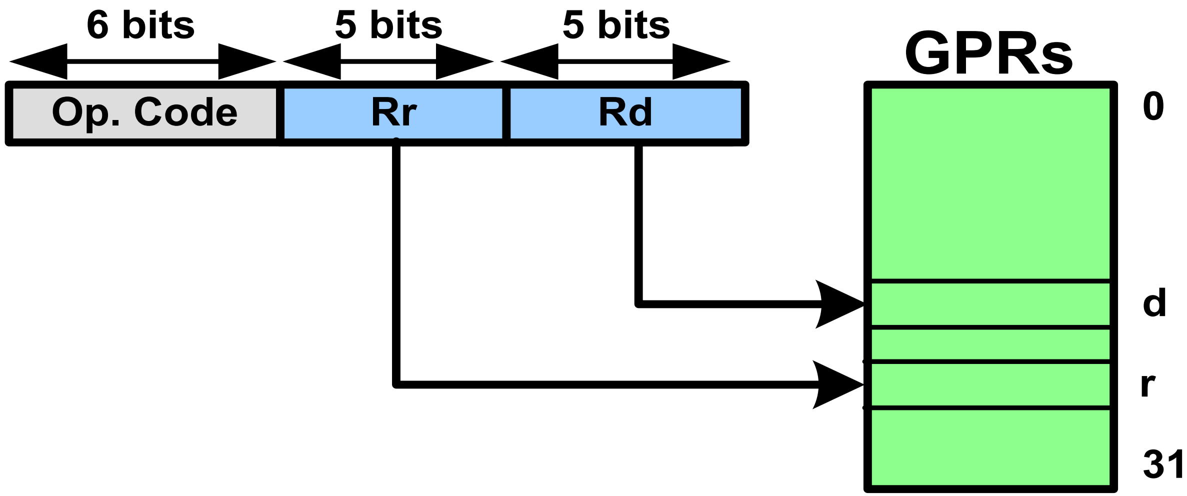
Two-register addressing mode

- ADD Rd,Rr

  `ADD R26,R23`

- SUB Rd,Rr

  `LDI R20,R10`



I/O direct addressing mode

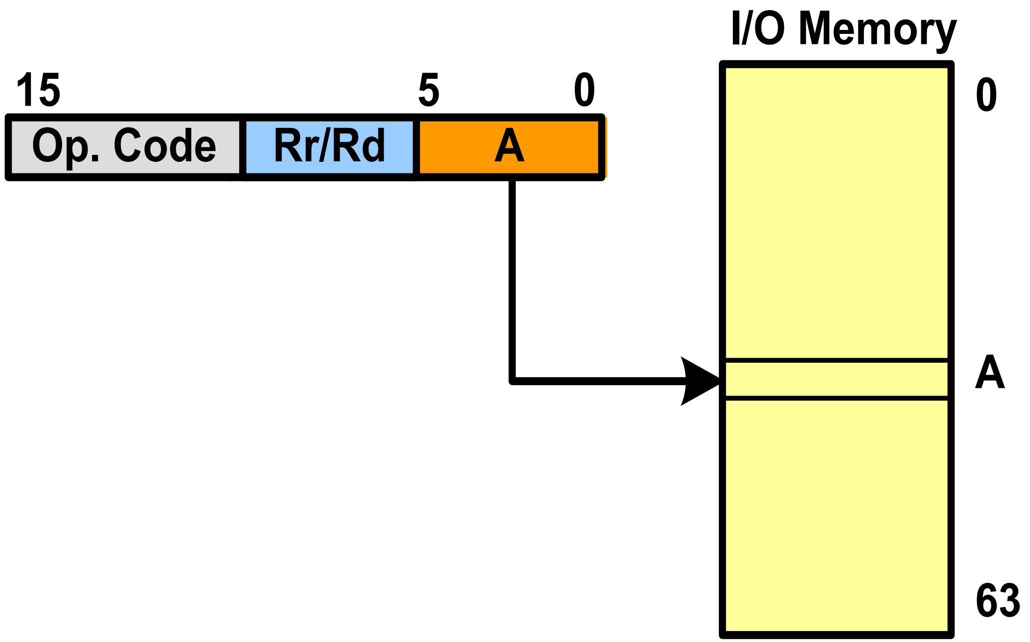
- OUT address, Rs

  `OUT PORTD,R16`

- IN Rd, address

  `IN R19,PINC`

1011	1AAr	rrrr	AAAA



Direct addressing mode

- LDS Rd,address

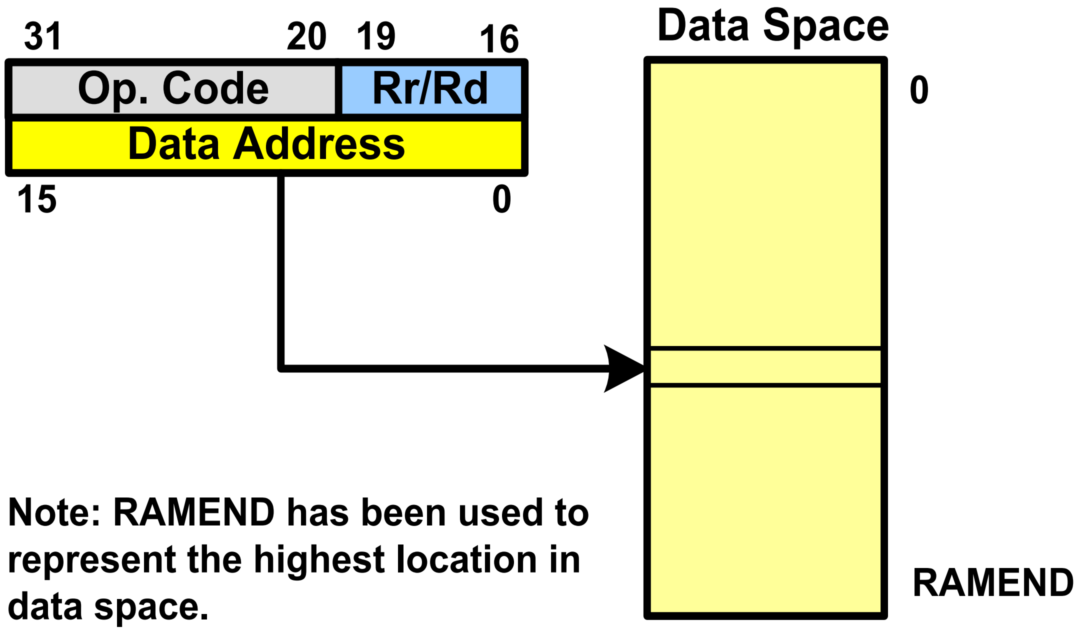
  `LDS R19,0x313`

1001	000d	dddd	0000
kkkk	kkkk	kkkk	kkkk

- STS address,Rs

  `STS 0x95,R19`

1001	001d	dddd	0000
kkkk	kkkk	kkkk	kkkk

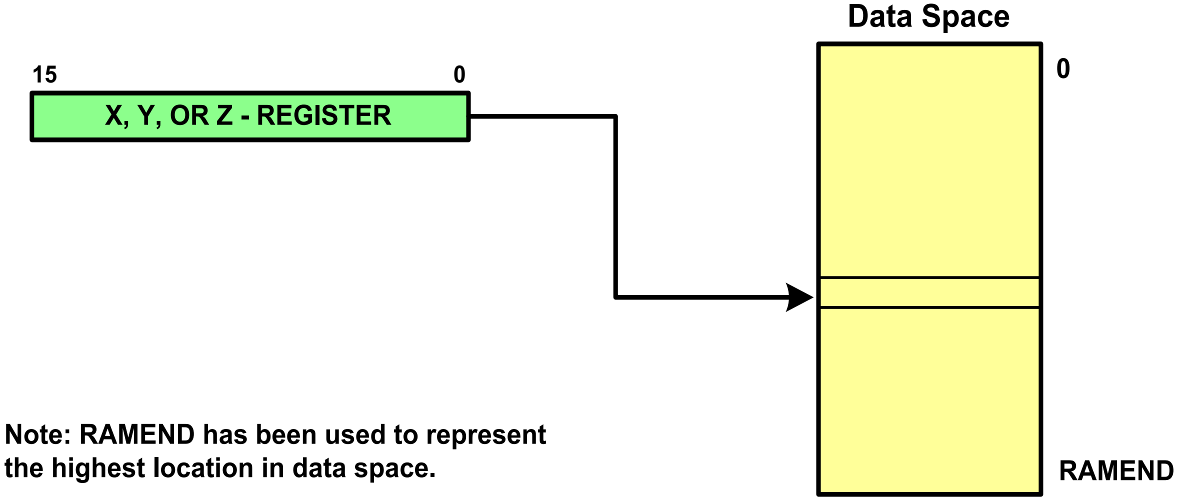


Note: RAMEND has been used to represent the highest location in data space.

Register indirect addressing mode

- LD Rd,X

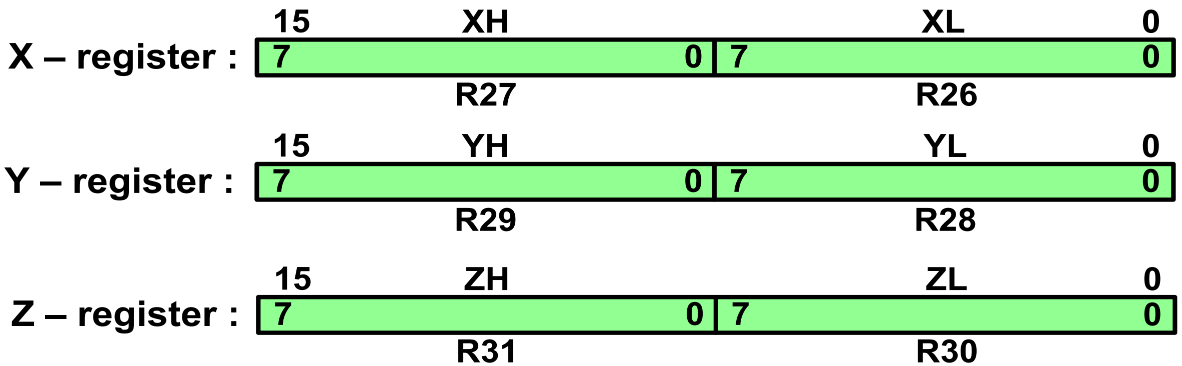
```
LD R24,X
LD R19,Y
LD R20,Z
```



Note: RAMEND has been used to represent the highest location in data space.

- ST X,Rs

```
ST X,R18
ST Y,R20
```

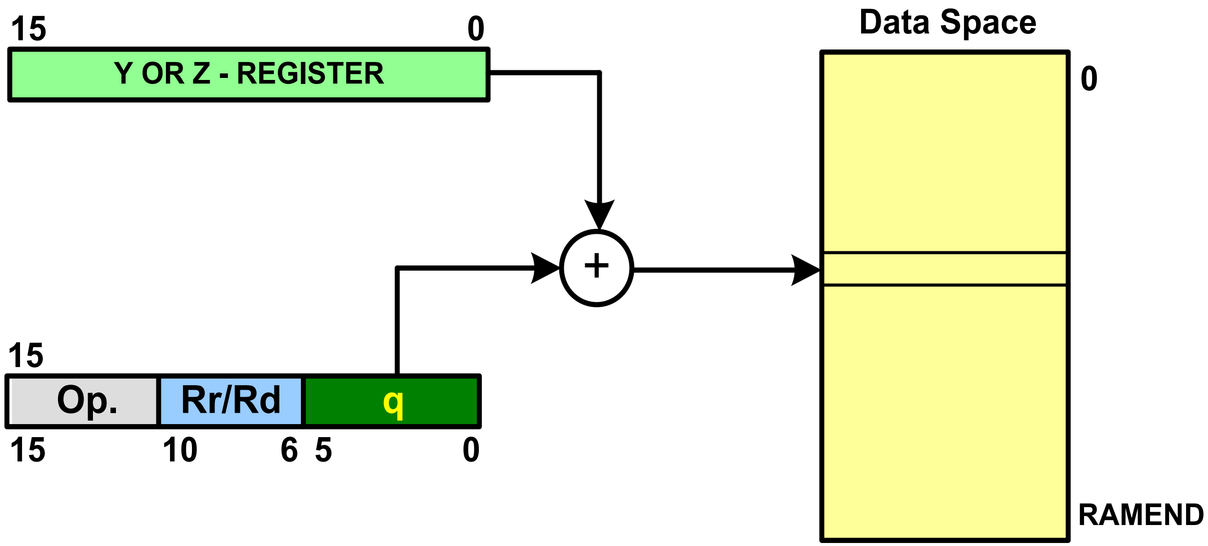


Register indirect with displacement

- STD Z+q,Rs ;store Rs into location Z+q

  `STD Z+5,R20` ;store R20 in location Z+5
- LDD Rd, Z+q ;load from Z+q into Rd

  `LDD R20, Z+8` ;load from Z+8 into R20



Auto-increment and Auto decrement

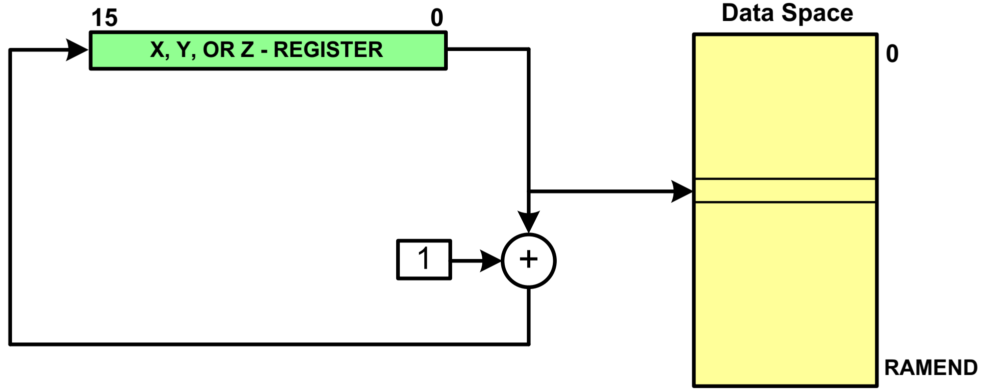
- Register indirect addressing with Post-increment

  LD Rd, X+

  ```
  LD R20,X+
  ```

  ST X+, Rs

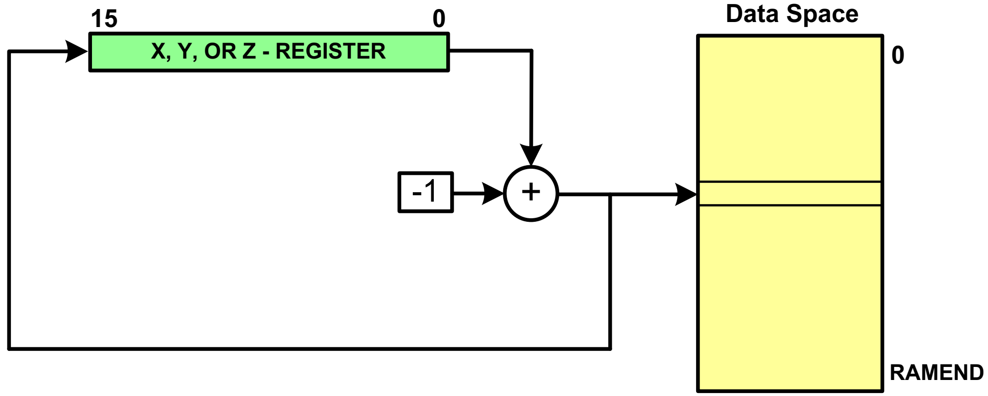
  ```
  ST X+,R8
  ```



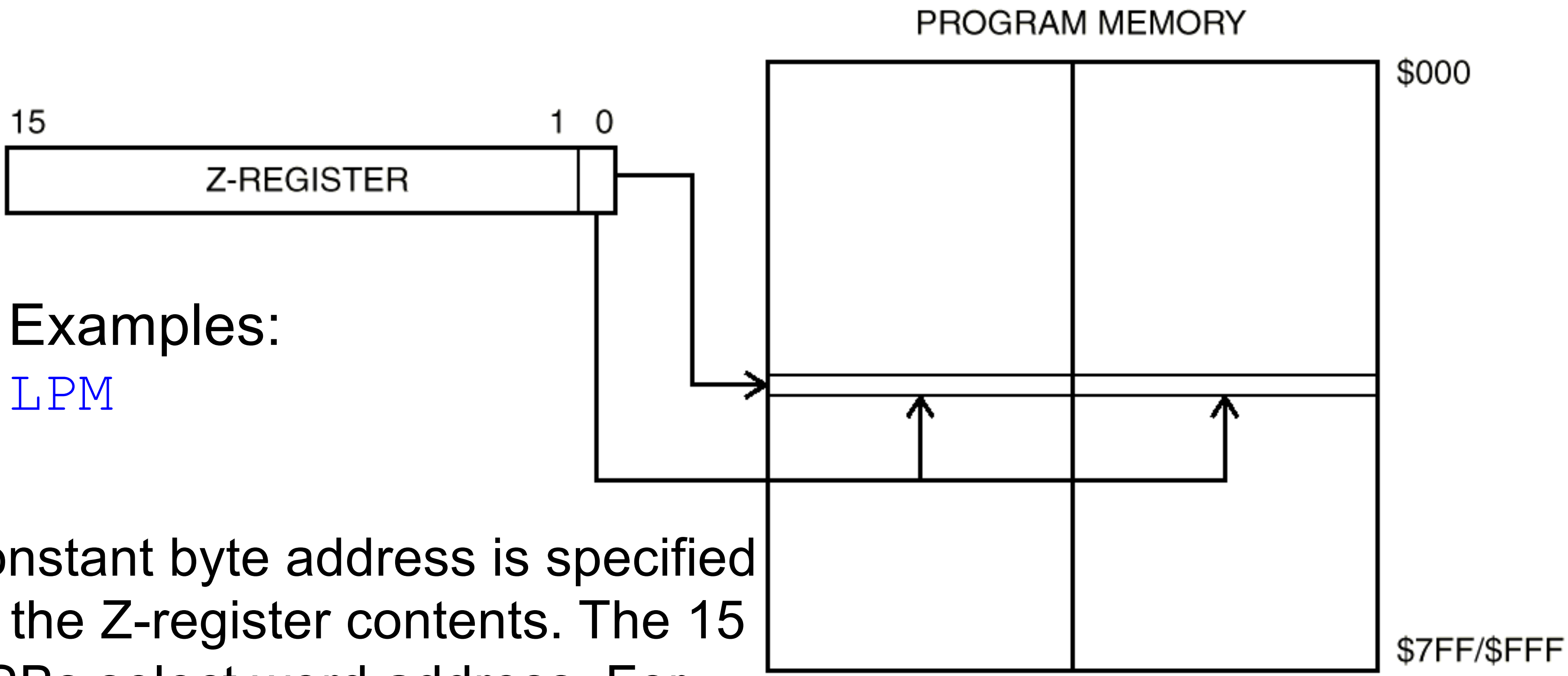
- Register indirect addressing with Pre-decrement

  LD Rd, -X

  ```
  LD R19,-X
  ST -X,R31
  ```



Program Memory Constant Addressing using the LPM Instruction

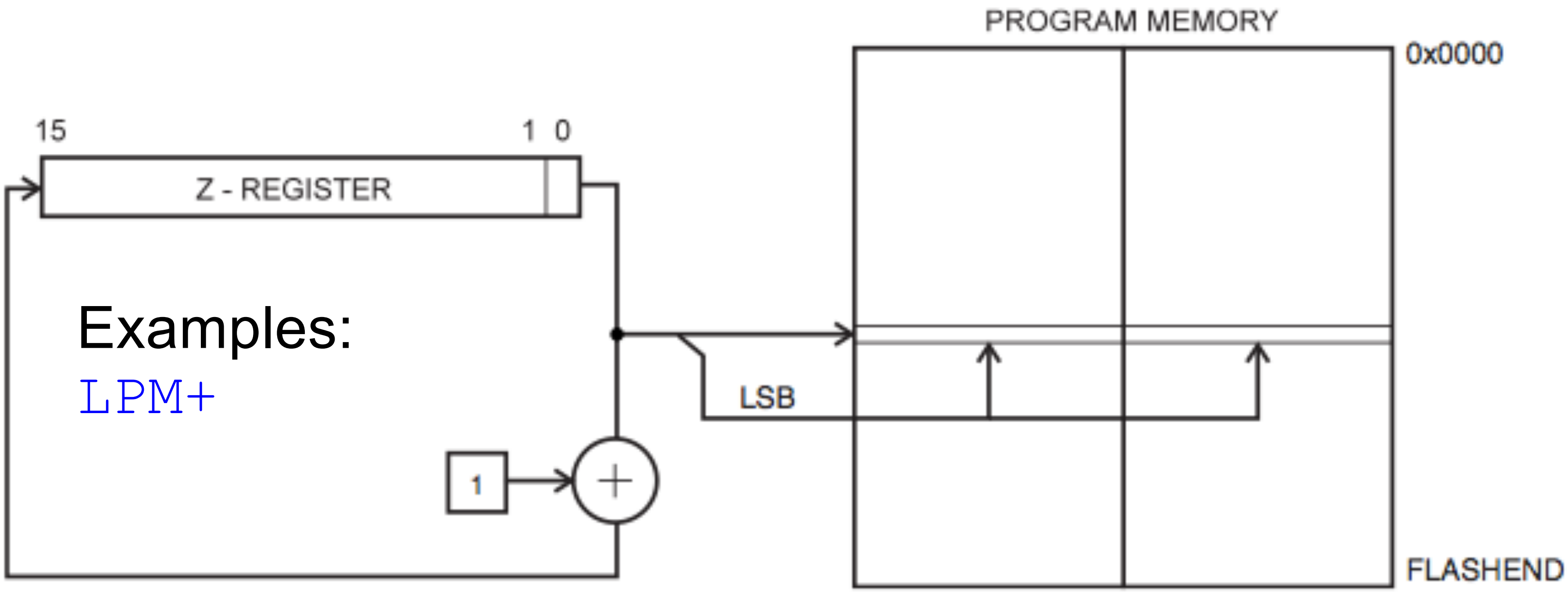


Examples:

`LPM`

Constant byte address is specified by the Z-register contents. The 15 MSBs select word address. For LPM, the LSB selects low byte if cleared (LSB = 0) or high byte if set (LSB = 1).

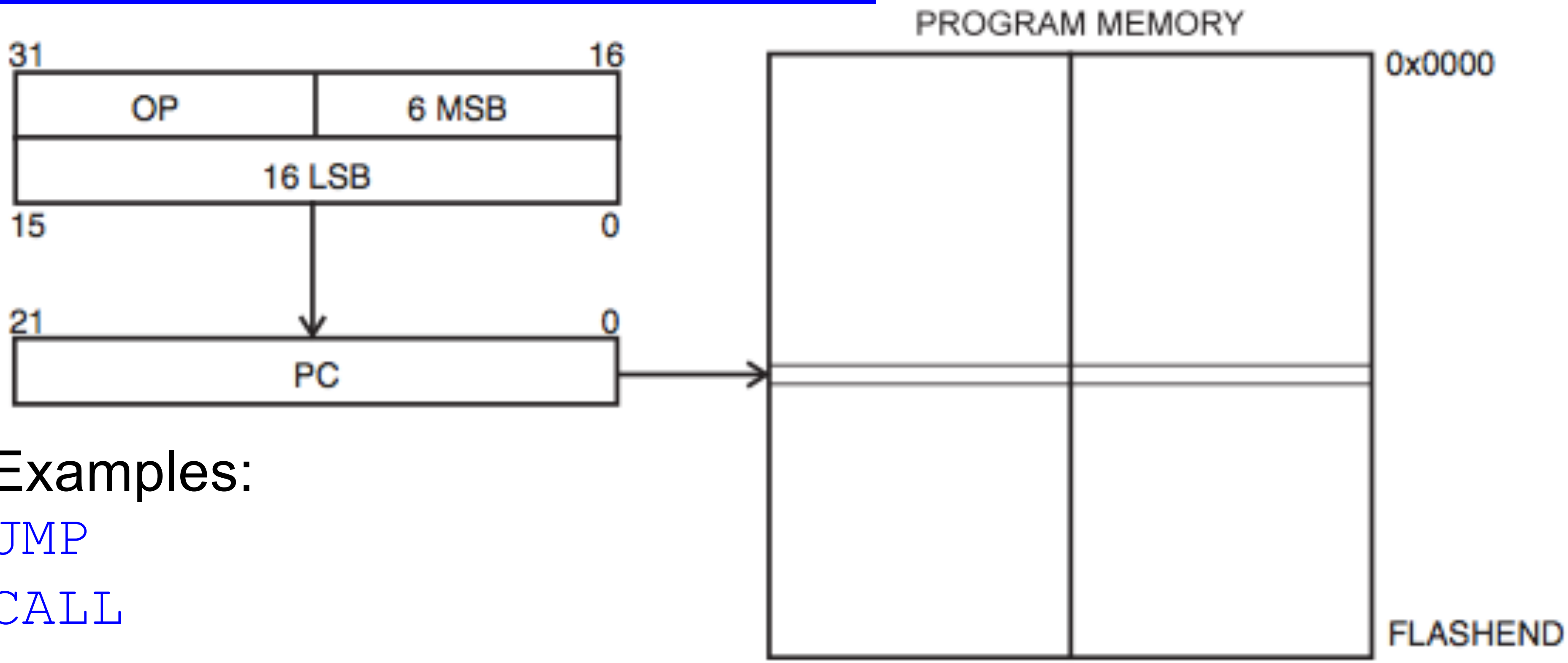
Program Memory with Post-increment using the LPM Z+ Instruction



Examples:
LPM+

Constant byte address is specified by the Z-register contents. The 15 MSBs select word address. The LSB selects low byte if cleared (LSB = 0) or high byte if set (LSB = 1).

Direct Program Addressing, JMP and CALL



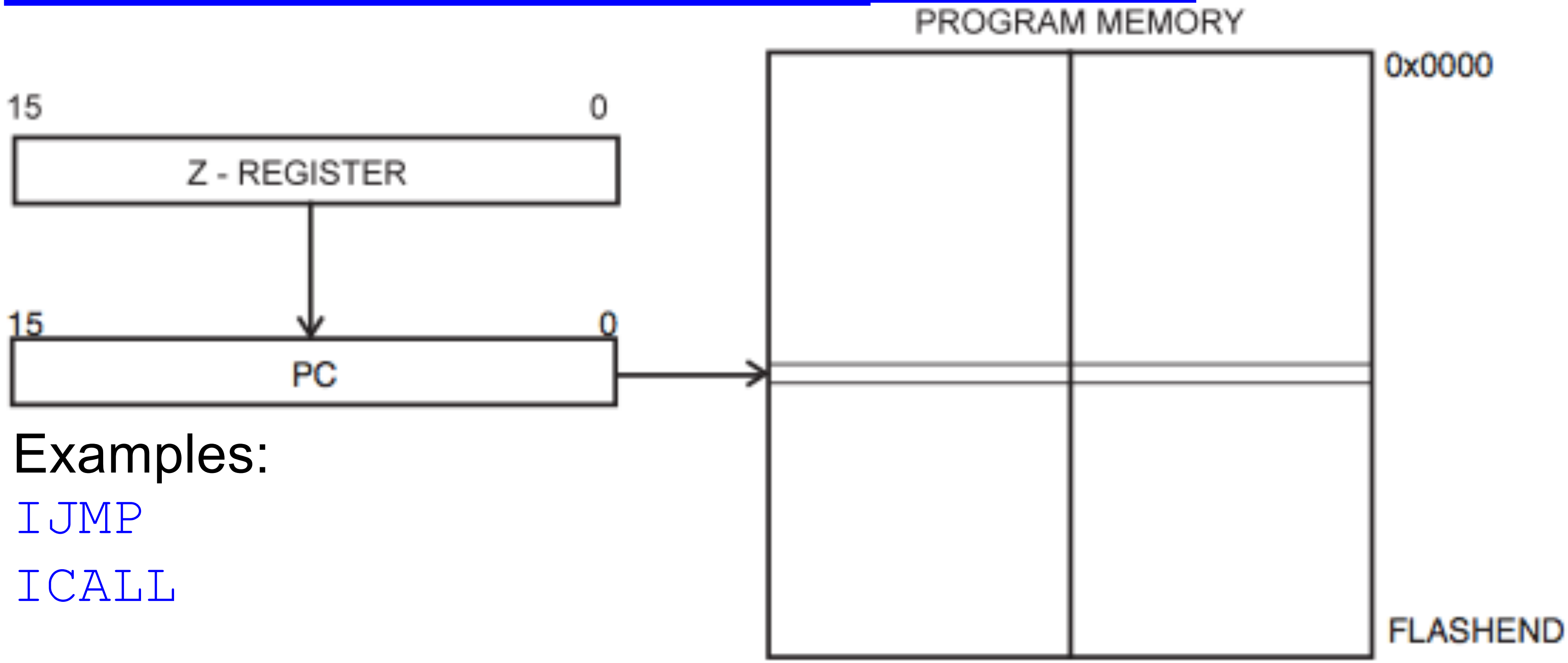
Examples:

`JMP`

`CALL`

Program execution continues at the address immediate in the instruction word.

Indirect Program Addressing, IJMP and ICALL



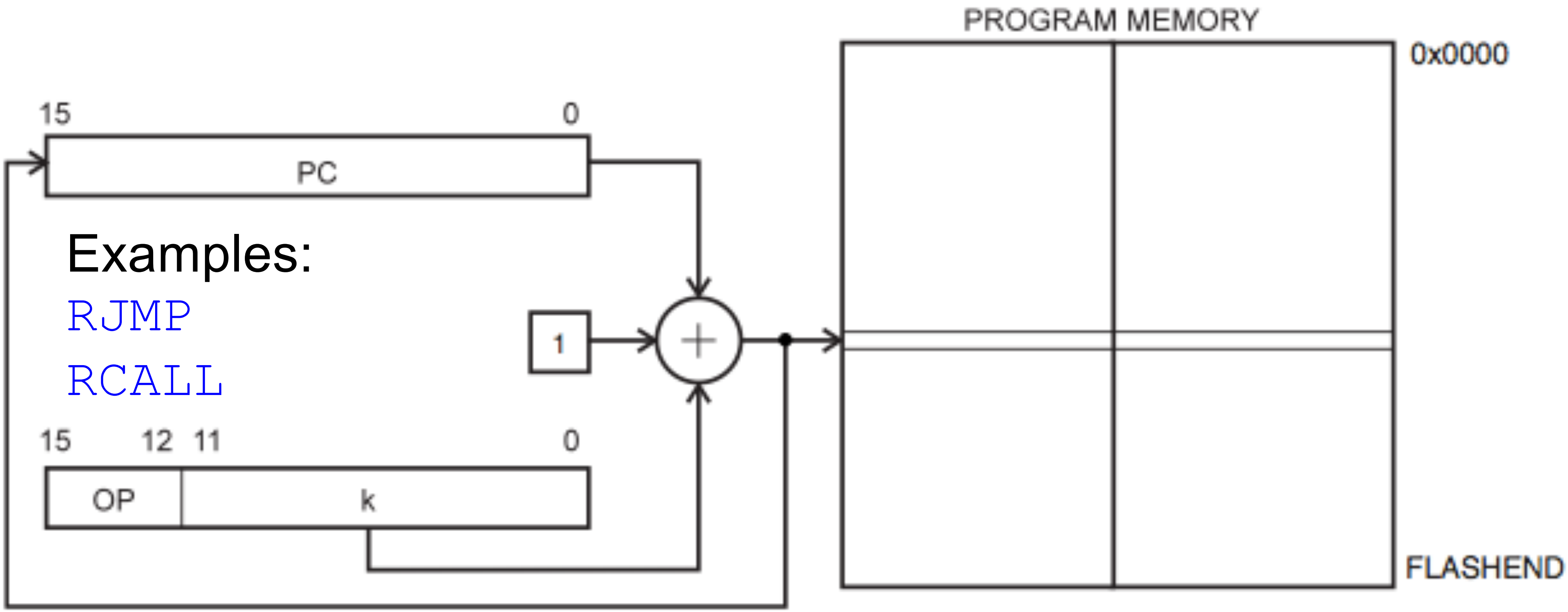
Examples:

`IJMP`

`ICALL`

Program execution continues at address contained by the Z-register (i.e., the PC is loaded with the contents of the Z- register).

Relative Program Addressing, RJMP and RCALL



Examples:

RJMP

RCALL

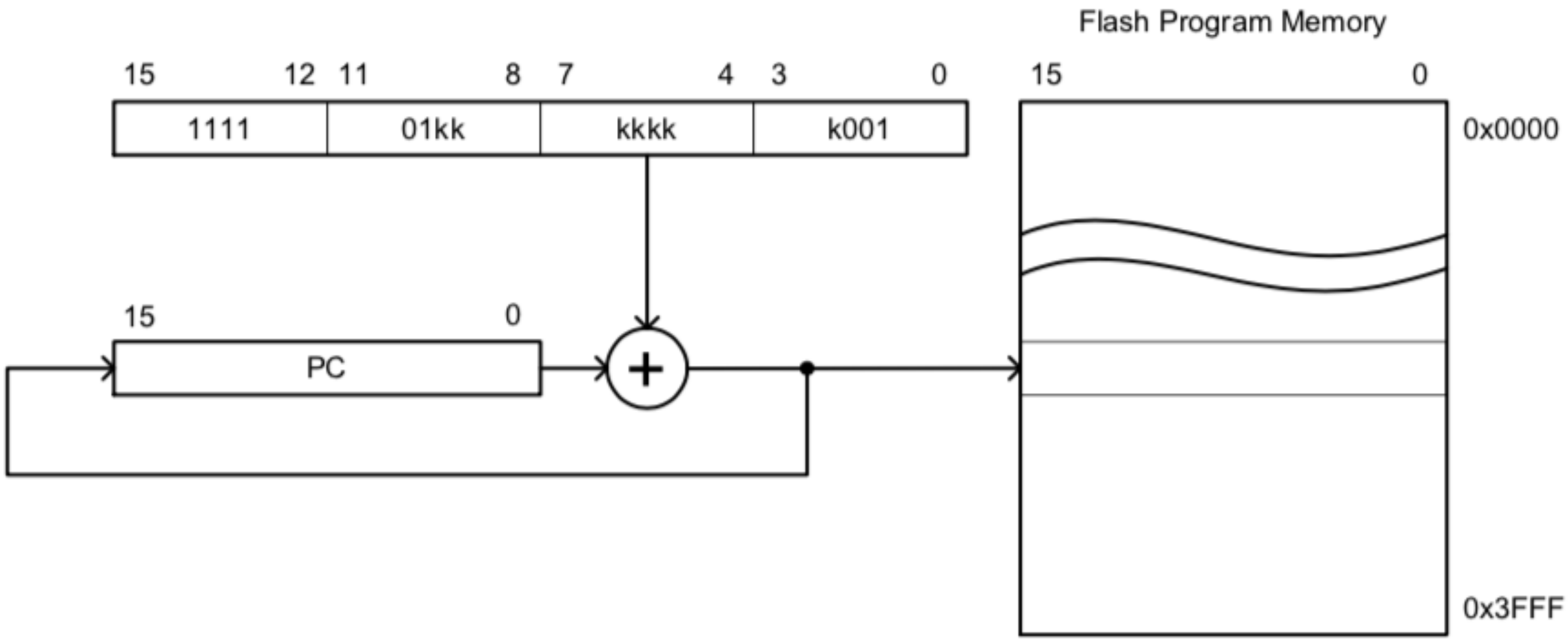
Program execution continues at address PC + k + 1. The relative address k is from -2048 to 2047.

Relative addressing

- RJMP k – RCALL k	[-2048...+2047] words
- Conditional Branches	[-64...+63] words
- skip instructions	+1 instruction

Relative – (BREQ as example)

If a relative branch is taken (test condition is true) a 7-bit signed offset is added to the PC (k in figure below). The result corresponding to the target address.



A CONDITIONAL CONTROL TRANSFER (BRANCH) SEQUENCE

A conditional control transfer (branch) sequence is typically comprised of 2 instr:

1. The first instruction performs some arithmetic or logic operation using the ALU of the processor.
    1. Examples of this first type of instruction includes: cp, cpc, cpi, tst. These ALU operations result in SREG flag bits 5 to 0 being set or cleared (i.e., H, S, V, N, Z, C).
        - WARNING: The Atmel "Instruction Set Summary" pages provided as part of each quiz and exam incorrectly classifies compare instructions (cp, cpc, cpi) as "Branch Instructions." They should be listed under "Arithmetic and Logical Instructions." To highlight this inconsistency on Atmel's part, the tst instruction is correctly listed under "Arithmetic and Logical Instructions."
    2. To allow for multiple branch conditions to be tested, these instructions typically do not modify any of our 32 general purpose registers. *For compare instructions, this is accomplished by a subtraction without a destination operand.*
2. The second instruction is a conditional branch instruction testing one or more SREG flag bits.

Sending data serially under program control (1/2) (ÖK)

- *Write a program to transfer the value 0x41 serially one bit at a time via pin PB1. Put one high bit at the start and end of the data and send LSB first*

```
SETUP:
        SBI DDRB, 1             ; B1 is output
        LDI R20, 0x41           ; Data to be transferred
        CLC                     ; C=0 to start with
        LDI R16, 8              ; 8 bits to transfer
        SBI PORTB, 1            ; B1 = 1 (start bit – see above)
AGAIN:
```

Sending data serially under program control (2/2) (ÖK)

```
AGAIN:
        ROR R20           ; Next bit to C
        BRCS ONE          ; C = 1?
        CBI PORTB, 1      ; no
        JMP NEXT
ONE:    SBI PORTB, 1      ; yes
NEXT:
        DEC R16           ; bit counter
        BRNE AGAIN        ; all bits done?
        SBI PORTB, 1      ; yes, set stop bit
        HERE:  JMP HERE
```

Receive Data Serially (ÖK)

- *Write a program to receive a byte of data serially through pin PC7 and save it in R20 register. The byte comes in LSB first.*

- *Note! We need to synchronize with sender somehow…*

```
SETUP:     CBI DDRC, 7      ; C7 = input
           LDI R16, 8       ; bit counter
           LDI R20, 0       ; data
AGAIN:
           SBIC PINC, 7
           SEC
           SBIS PINC, 7
           CLC
           ROR R20
           DEC R16
           BRNE AGAIN       ; done?

HERE:      JMP HERE
```

Random number generator (ÖK)

- Linear Congruential Method (Lehmer, 1948)

  $I_{n+1} = \mathrm{mod}_m(a{*}I_n + c)$

  Choose ”appropriate” constants:

  m = 255 (8 bit registers, that overflows…)

  a = 5, c= 1

  I = variable ”Random”

  Code comments:

  ; Random number will be between 0 and 255

  ; Random ”seed” = content of register ”Random” at start

  ; ”Count” to be random, between 60 and 188

Random number generator – code example (ÖK)

```
; Random number between 0 and 255
MOV     Temp, Random        ; multiply by 5…
ADD     Random, Temp
ADD     Random, Temp
ADD     Random, Temp
ADD     Random, Temp
INC     Random              ; … and add 1
; Counter to be 60 to 188
MOV     Count, Random
LSR     Count               ; divide by 2, and …
SUBI    Count, -60          ; …add 60 (trick!)
```