Instruction set of AVR 8-bit processors

Dr Asimakis Leros
ICSD Dept., Univ. of the Aegean

AVR 8 bit instruction set

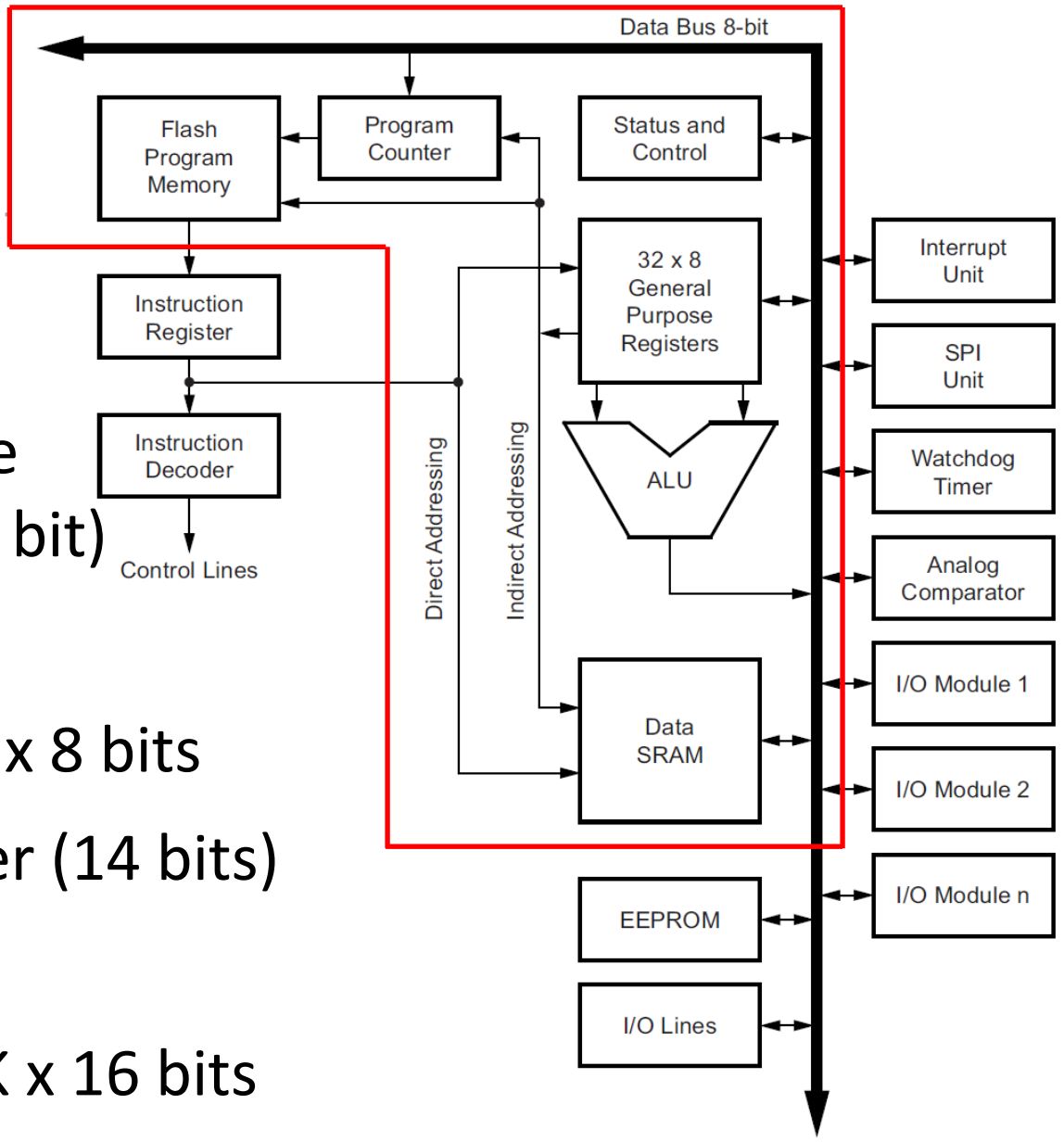
- Instruction format
- Data transfer instructions (move, load, store)
- Arithmetic operations
- Logical operations
- Program flow control (branch, jump)
- Implementation of if-then-else

AVR 8 bit instruction set

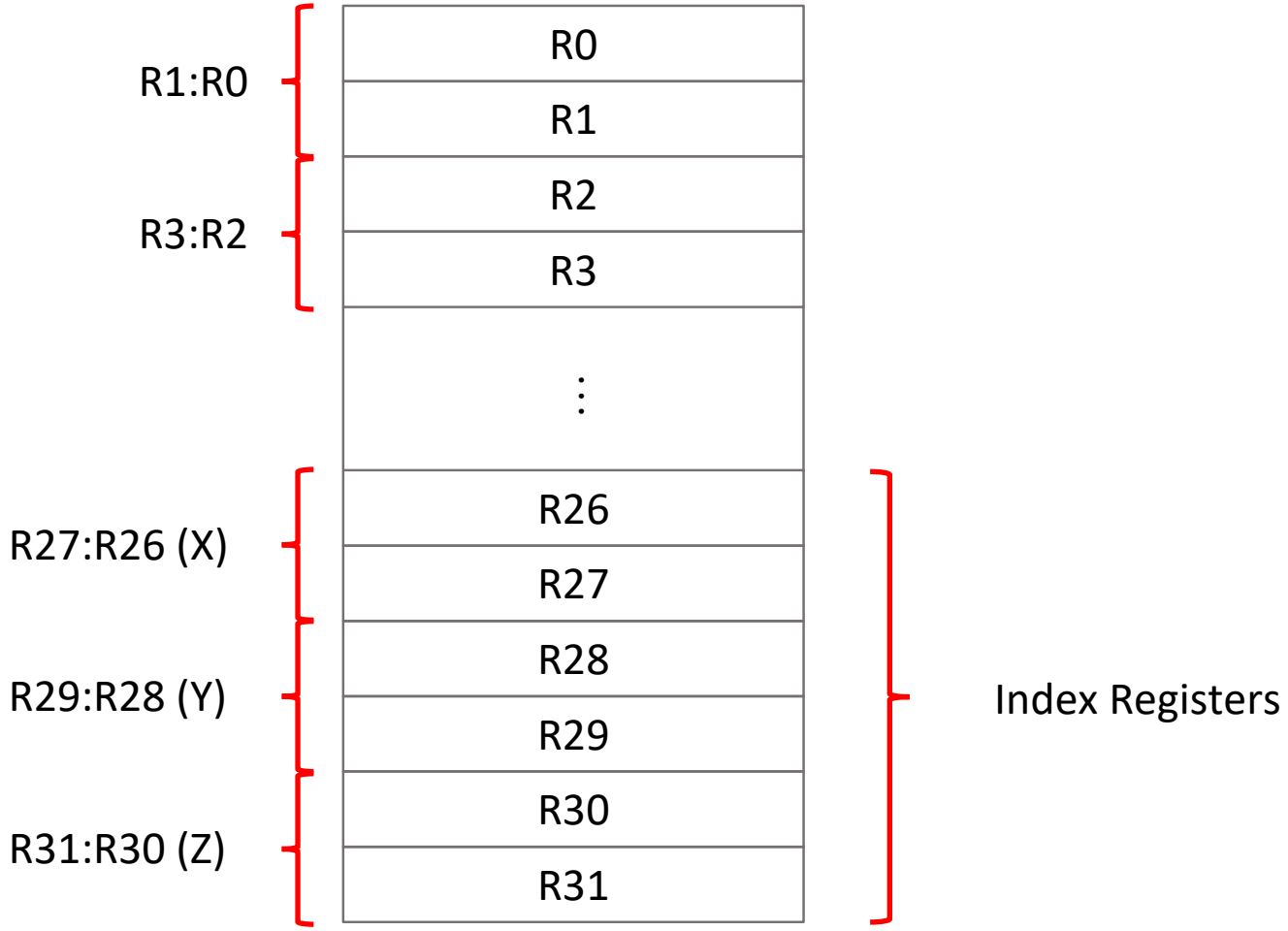
- Implementation of for & while loops
- Arrays & pointers
- Procedure call
- Stack management
- Input-output instructions

AVR CPU core

- ALU
- General purpose registers (32 x 8 bit)
- SREG
- SRAM (data) 2K x 8 bits
- Program Counter (14 bits)
- Stack Pointer
- Flash (code) 16K x 16 bits

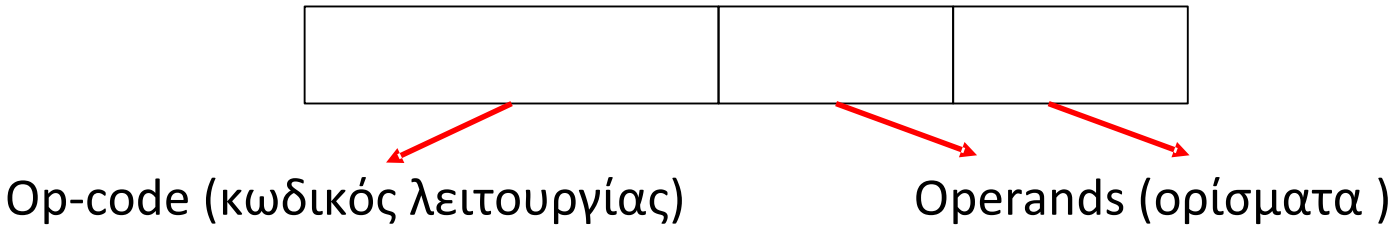


General purpose registers



Instruction format

- One word (0, 1 ή 2 operands)



- Two words (1 ή 2 operands)

Instruction format

- Orthogonal instruction set
  - Similar instructions differ in one or a few bits
  - Operands are located in fixed positions within the instruction

(this results in simpler and efficient instruction decoding)

- RISC
  - Most intructions have only registers as operands
  - Only load and store instructions access the main memory

Data transfer instructions

- Between registers:

  MOV Rd, Rr ; Rd ← Rr

0 0 1 0 1 1 r d d d d d r r r r

- Assign a value to a register (load immediate):

  LDI Rd, K ; Rd ← K, where 16 ≤ d ≤ 31, 0 ≤ K ≤ 255

1 1 1 0 K K K K d d d d K K K K

Direct memory addressing
Αμεση διευθυνσιοδότηση μνήμης

- Direct data transfer between register and memory

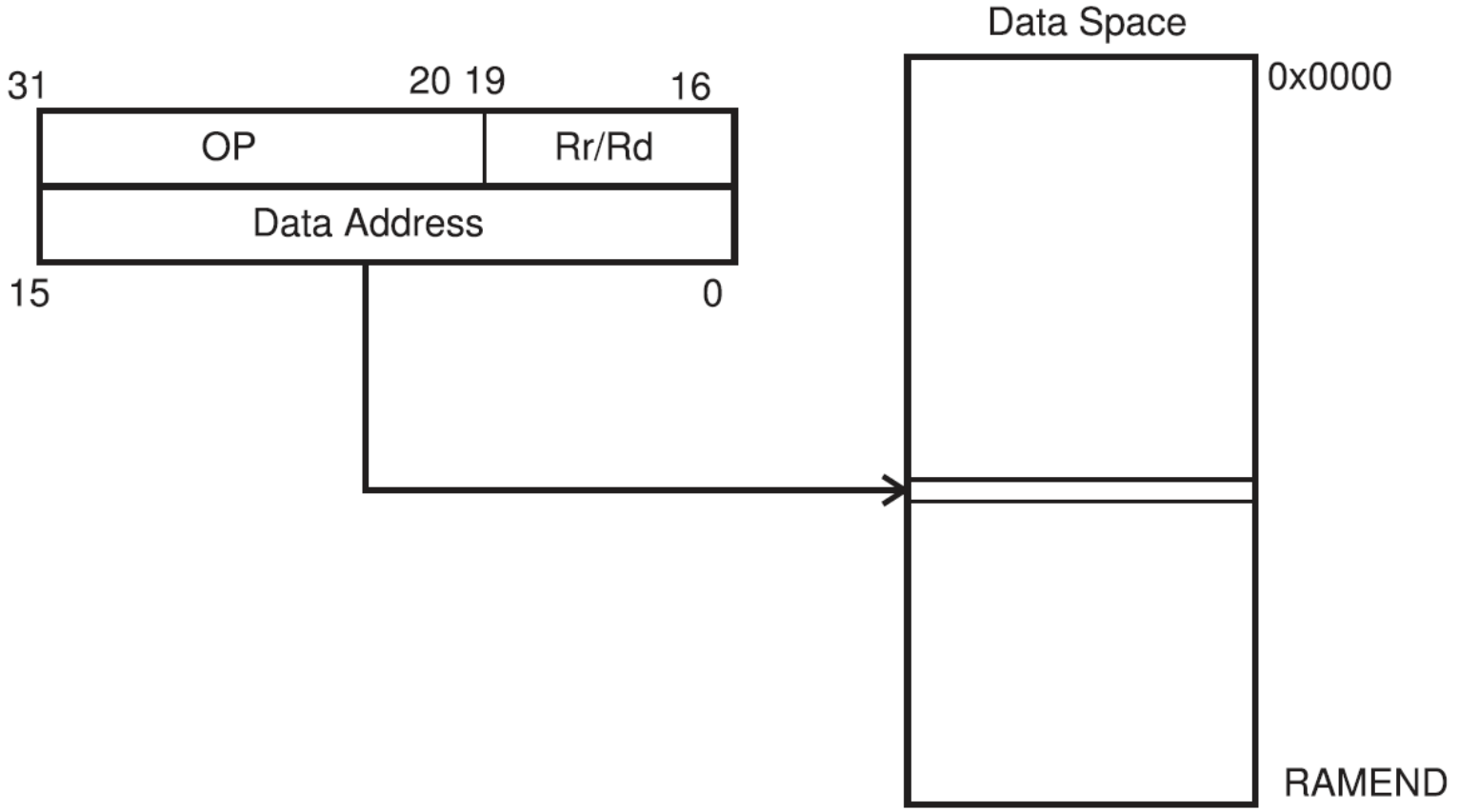
LDS Rd, k ; Rd ← (k), where 0 ≤ r ≤ 31, 0 ≤ k ≤ 65535

1 0 0 1 0 0 0 d d d d 0 0 0 0
k k k k k k k k k k k k k k k k

STS k, Rr ; (k) ← Rr

1 0 0 1 0 0 1 r r r r r 0 0 0 0
k k k k k k k k k k k k k k k k

Direct memory addressing



Indirect memory addressing using index

- From memory to register

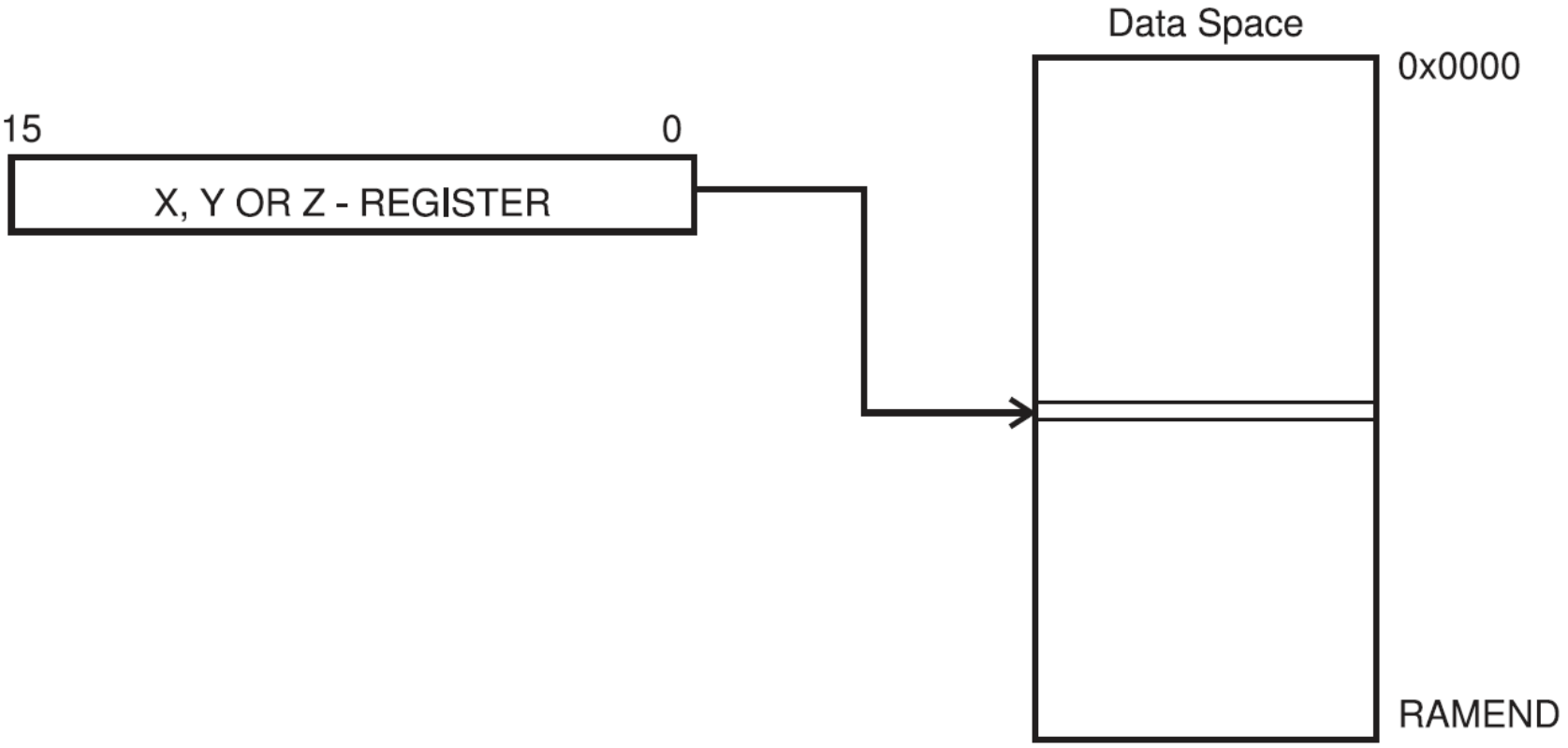
LD Rd, X	; Rd ← (X)	1 0 0 1 0 0 0 d d d d 1 1 0 0
LD Rd, X+	; Rd ← (X), X ← X + 1	… 1 1 0 1
LD Rd, -X	; X ← X – 1, Rd ← (X)	… 1 1 1 0

- From register to memory

ST X, Rr	; (X) ← Rr κλπ	1 0 0 1 0 0 1 r r r r 1 1 0 0

- The X, Y, Z 16-bit register pairs are used as indices (δείκτες).

Indirect memory addressing using index



Arithmetic instructions - addition

- Addition of registers

  ADD Rd, Rr ; Rd ← Rd + Rr

  ADC Rd, Rr ; Rd ← Rd + Rr + C

- Addition of a constant to a register

ADIW Rd+1:Rd,K ; Rd+1:Rd ← Rd+1:Rd + K

; d ∈ {24,26,28,30}, 0 ≤ K ≤ 63

1 0 0 1 0 1 1 0 K K d d K K K K

Example: addition of 16 bit integers

- Add the value in the register pair r3:r2 to the pair r1:r0. The result is found in r1:r0.

```
add  r0, r2    ; add LS bytes
adc  r1, r3    ; add MS bytes + carry
```

Subtraction

- Works like addition:

  SUB Rd, Rr ; Rd ← Rd - Rr

  SBC Rd, Rr ; Rd ← Rd - Rr - C

- There is also a version to subtract an 8-bit value from a register (subtract immediate):

  SUBI Rd, K ; Rd ← Rd - K

  SBCI Rd, K ; Rd ← Rd - K - C

Comparison

- Functions like the subtraction operation.
  The difference is that the result is not stored:

```
CP  Rd, Rr     ; Rd - Rr
CPC  Rd, Rr    ; Rd - Rr – C
CPI  Rd, K     ; Rd - K
```

- Why subtract and not keep the result? This instruction is used to assign values to flags (Z, C, S, N, V) to be used by a subsequent branch instruction.

More arithmetic instructions

- Negation – 2's complement (signed value):

  NEG Rd ; Rd ← 0x00 – Rd

- Increment or decrement by 1:

  INC Rd ; Rd ← Rd + 1

  DEC Rd ; Rd ← Rd – 1

Logical operations

AND Rd, Rr ; $Rd \leftarrow Rd \wedge Rr$

OR Rd, Rr ; $Rd \leftarrow Rd \vee Rr$

EOR Rd, Rr ; $Rd \leftarrow Rd \oplus Rr$

- AND/OR with a constant: used to set or reset specific bits

ANDI Rd, K ; $Rd \leftarrow Rd \wedge K$

ORI Rd, K ; $Rd \leftarrow Rd \vee K$

; $16 \leq d \leq 31, 0 \leq K \leq 255$

Logical operations

- One's complement (reversal of all bits):

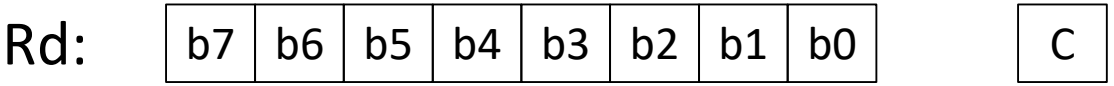
  COM Rd ; Rd $\leftarrow$ 0xFF – Rd

- Test: no result is stored. Used to check if Rd is 0, Positive or negative

  TST Rd ; Rd $\leftarrow$ Rd $\wedge$ Rd

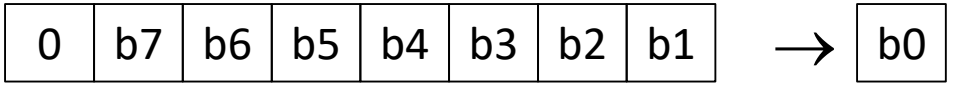
- The result can be found in the Z and N flags.

Shift (ολίσθηση) comes in two flavors



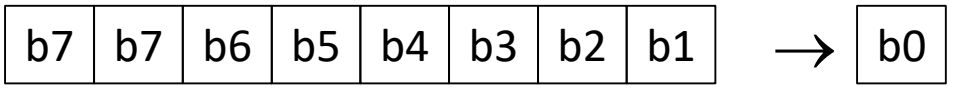
- Logical shift right is equivalent to division of an unsigned int by 2 (half bit goes to carry flag).

LSR Rd

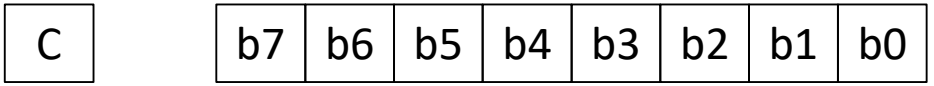


- Arithetic shift right is equivalent to division of a signed int by 2. The sign bit is not affected by the shift, half bit goes to carry flag.

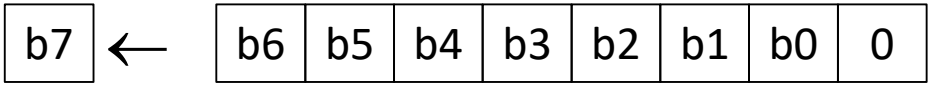
ASR Rd



Shift (cont.)

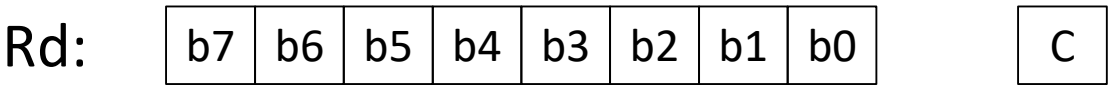


- Left shift is equivalent to multiplication by 2.
  Only for unsigned numbers.



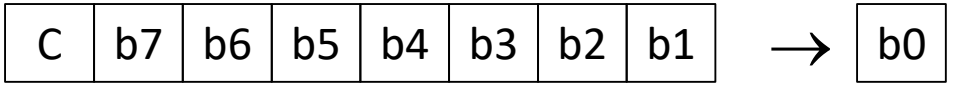
- The instruction LSL Rd is compiled to ADD Rd, Rd.

Rotation (περιστροφή)



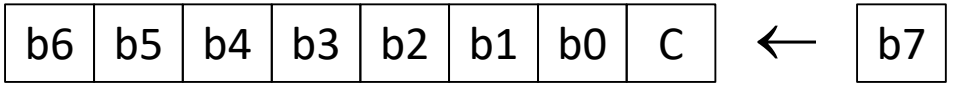
- Rotate right through carry:

ROR Rd →



- Rotate left through carry:

ROL Rd ←



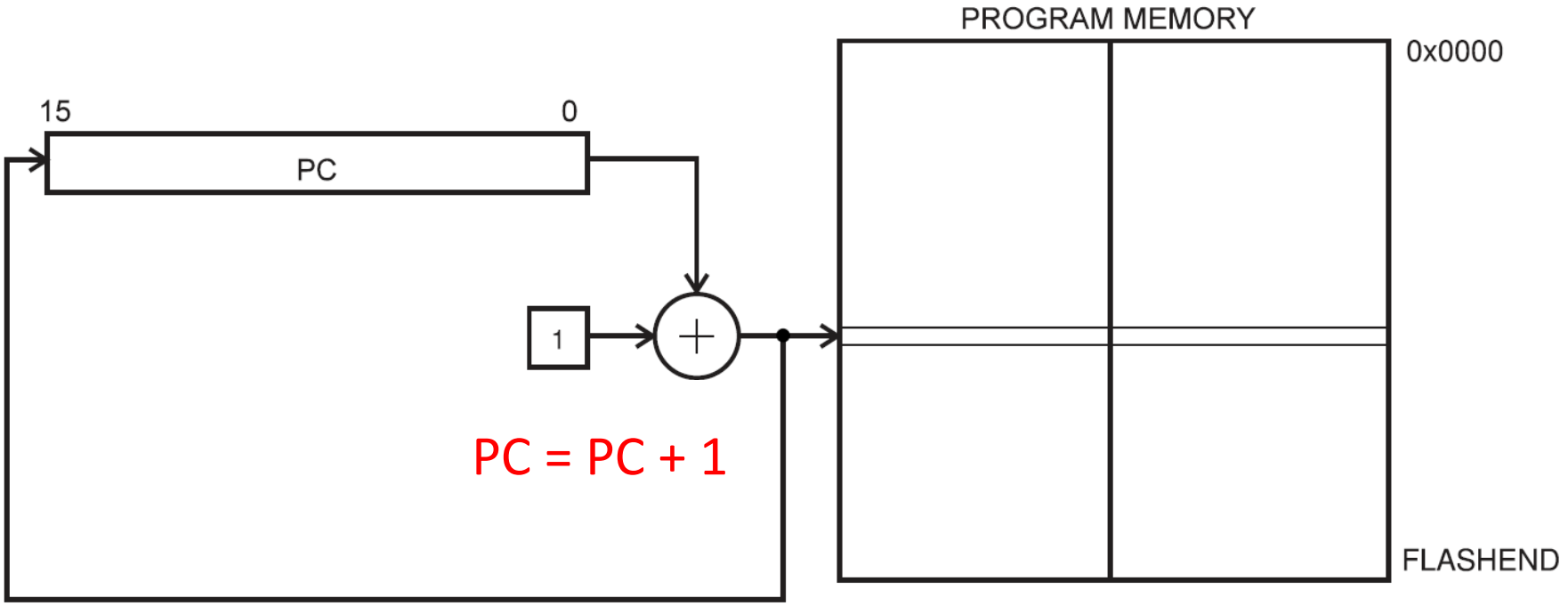
- ROL Rd is compiled as ADC Rd, Rd.
- These instructions are used to implement shift operation on long integers (2 or more bytes).

Program flow control

There are three basic ways to transfer program control to another instruction:

- Unconditional transfer (διακλάδωση χωρίς συνθήκη)
  JUMP instructions, equivalent to goto
- Conditional transfer (διακλάδωση με συνθήκη)
  BRANCH instructions, equivalent to if <condition> goto
- Subroutine/procedure CALL and RETURN

Normal program flow

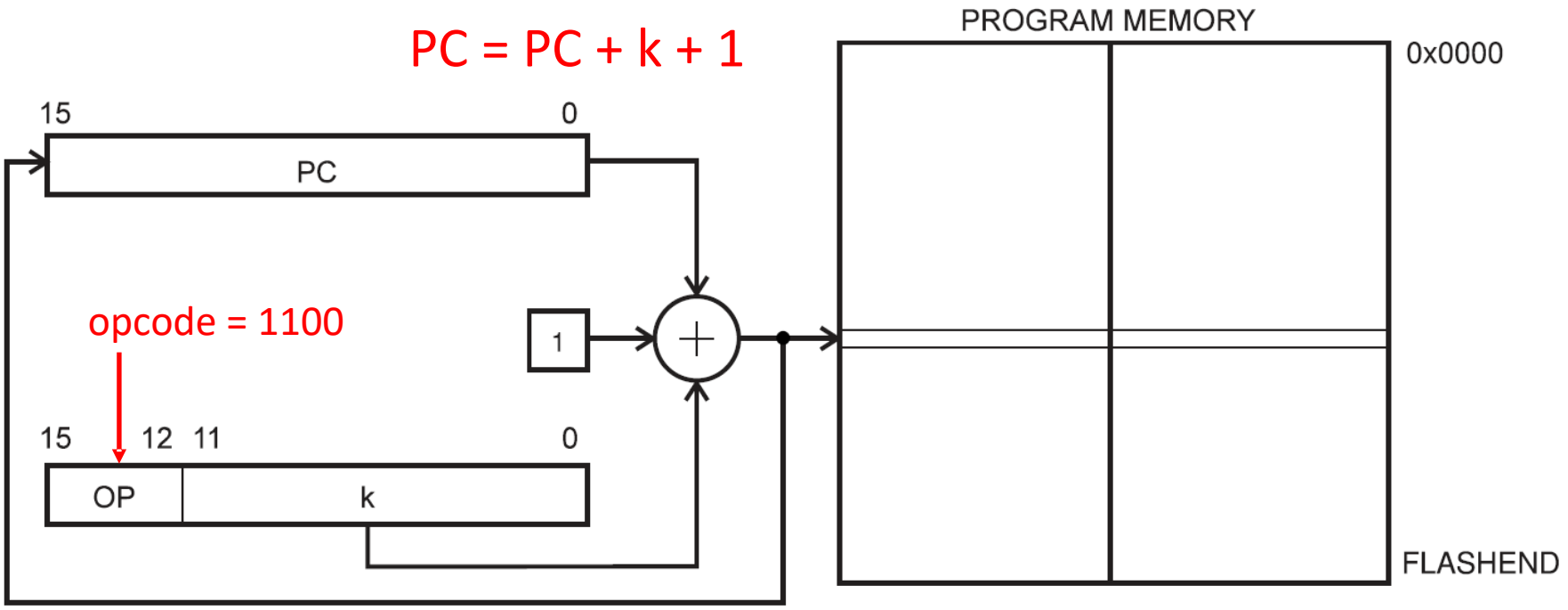


Instructions are executed sequentially
(Note that in the ATMega328 the PC uses 14 bits)

Unconditional transfer of control (1)

Near/short or relative jump (κοντινό ή σχετικό):

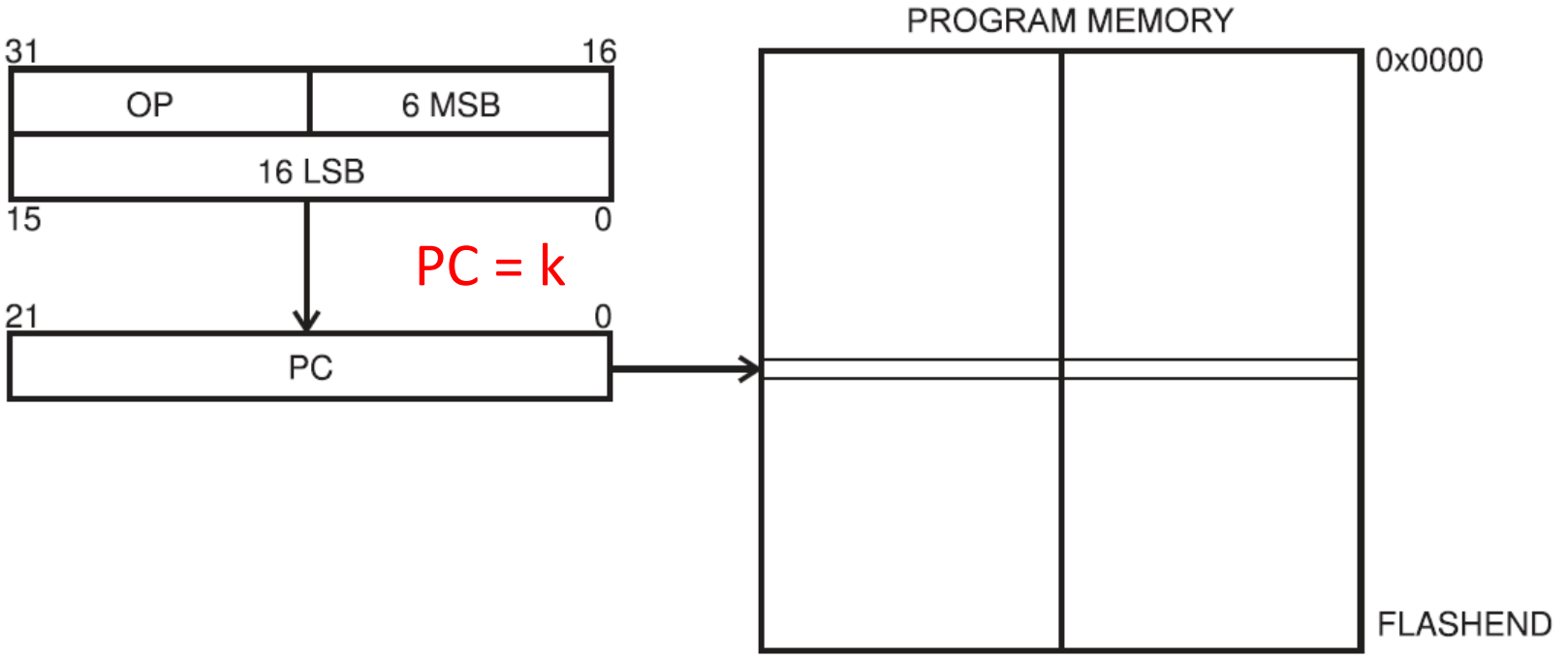
RJMP k ; −2048 ≤ k ≤ +2047



Unconditional transfer of control (2)

Far/long or direct jump (μακρινό ή άμεσο):

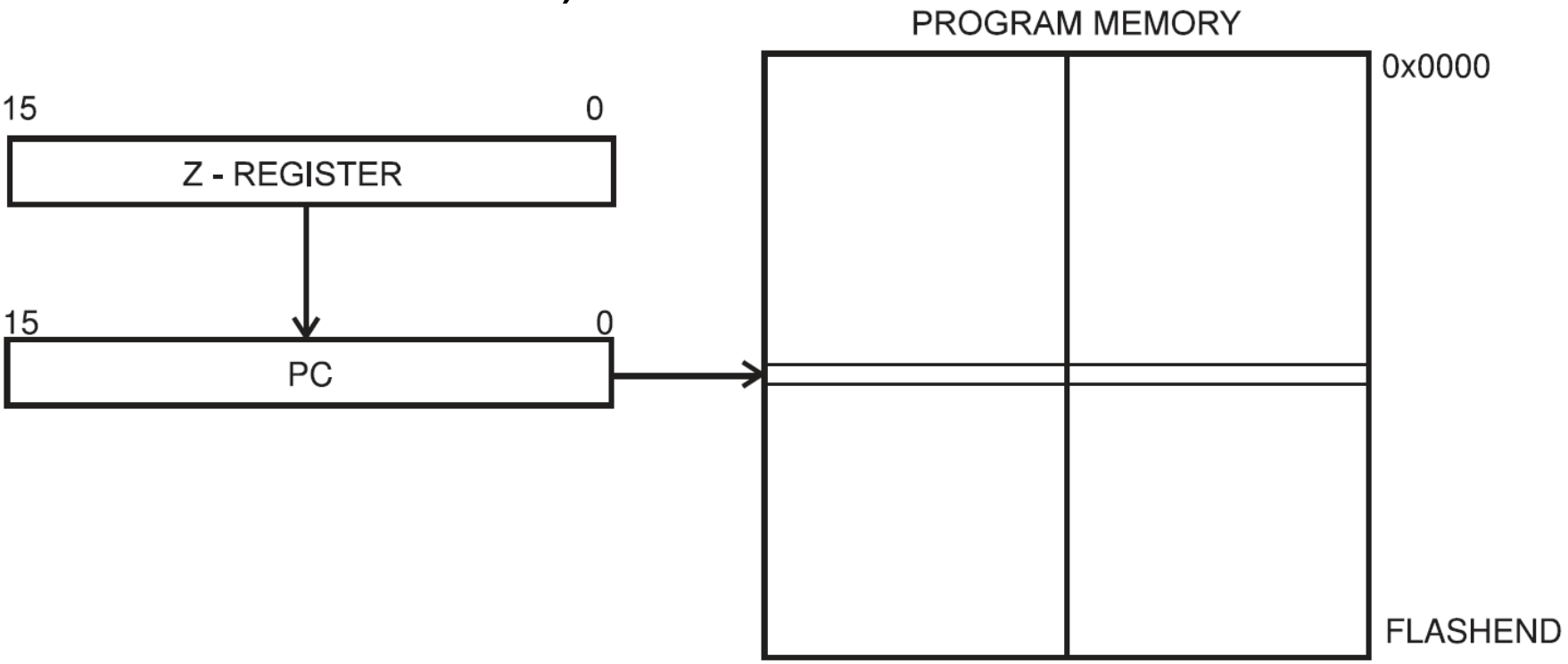
JMP k ; 0 ≤ k ≤ 4M



Unconditional transfer of control (3)

Indirect jump (έμμεσο):

IJMP ; PC ← Z



Conditional transfer of control (1)

- Branch if bit set:

```
BRBS  n, k     ;  -64 ≤ k ≤ +63
               ;  B = SREG[n], 0 ≤ n ≤ 7
```

- Function:

```
if SREG[n] == 1
        PC ← PC + k + 1
else
        PC ← PC + 1
```

Status register (καταχωρητής κατάστασης)

I	T	H	S	V	N	Z	C

(0) C Carry (result > 255 or < 0)

(1) Z Result is zero

(2) N Negative (b7 of result)

(3) V Signed overflow (result > 127 or < -128)

(4) S True sign of result (b8), S = N $\oplus$ V

(5) H Half-carry (from b3 to b4 of result)

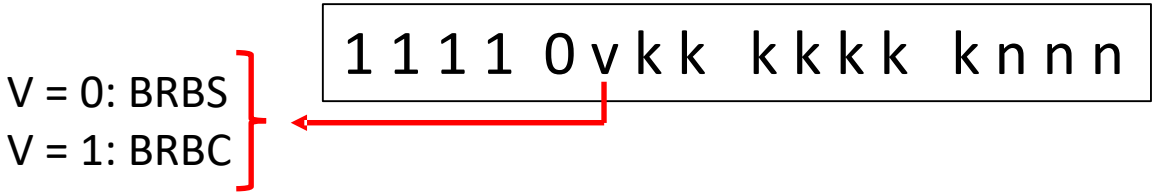
(6) T Temporary storage

(7) I Interrupts enabled

Conditional branching (2)

- Branch if bit clear:

```
BRBC  n, k     ;  -64 ≤ k ≤ +63
               ;  B = SREG[n], 0 ≤ n ≤ 7
```



- Simpler sytax:
  - BRCS k or BRCC k uses the Carry Flag
  - BRZS k or BRZC k uses the Zero Flag

Unsigned compare and branch (1)

- Comparison of unsigned integers and branching: We use the carry flag

```
        CP  Rd, Rr     ;  Rd – Rr (the result is not stored)
        BRCC  greq     ;  if C = 0 ie. Rd >= Rr goto greq
less:   (…)            ;  else (Rd < Rr) continue here
        (…)            ;
greq:   (…)            ;
```

Unsigned compare and branch (2)

- We can reverse the branching logic:

```
        CP  Rd, Rr     ;  Rd – Rr, not stored
        BRCS  less     ;  if C = 1 ie. Rd < Rr goto less
greq:   (…)            ;  else (Rd >= Rr) continue here
less:   (…)            ;
```

- In order to test for > or <= we reverse the operands:

```
        CP  Rr, Rd     ;
        BRCS  less     ;  if Rd > Rr goto less
```

Signed compare and branch (1)

- Comparison of signed integers and branching:
  We use the S flag (true sign of the result)

```
        CP  Rd, Rr      ;  Rd – Rr
        BRGE  greq ;  if S = 0 ie. Rd >= Rr goto greq
less:   (…)             ;  else (Rd < Rr) continue here
        (…)             ;
greq:   (…)             ;
```

Signed compare and branch (2)

- Equivalent branching structure:

```
        CP  Rd, Rr     ;  Rd – Rr
        BRLT  less     ;  if S = 1 ie. Rd < Rr goto less
greq:   (…)            ;  else (Rd < Rr) continue here
        (…)            ;
less:   (…)            ;
```

Testing for equality

- For both signed / unsigned we use the zero flag

```
        CP  Rd, Rr     ;
        BREQ  equal ;  if Z = 1 ie. Rd = Rr goto equal
neq:                   ;  case not equal
        (…)
equal:                 ;  case equal
```

- BRNE (branch if not equal) has a similar syntax

Other branching structures (1)

- Test for a carry or overflow after an arithmetic operation (add, subtract):

```
        ADD  Rd, Rr ;  add signed int
        BRVC  cont   ;  if no overflow (V = 0) continue
        CALL  overflow_error
cont:  (...)
```

- For signed integers, the overflow flag is relevant.
  For unsigned integers, the carry flag is relevant.

Other branching structures (2)

- Test for a positive, negative or zero value:

```
        TST  Rd        ; Rd ∧ Rd
        BREQ  zero     ; Z = 1 ie. Rd = 0
        BRPL pos       ; N = 0 ie. Rd >= 0
neg:    (…)            ; else Rd < 0
pos:    (…)
zero:   (…)
```

Implementation of loops

Two types of loops are possible:

- for or while loops
  - initialization
  - test terminating condition
  - loop body
- while-do loops
  - initialization
  - loop body
  - test terminating condition

for loops

```
                for(i = 0; i < 4; i++)

                LDI  r16, 0    ; i = 0
next:           CPI  r16, 4    ; if i >= 4...
                BRCC  exit     ; ...exit loop
                ; (loop body)
                INC  r16       ; i++
                RJMP  next     ; ...goto next
exit:           ; (next instruction)
```

do-while loop

```
            LDI  r16, 0     ; i = 0
next:       ; (loop body)
            INC  r16        ; i++
            CPI  r16, 4     ; if i < 4...
            BRCS  next      ; ...repeat loop
exit:       ; (next instruction)
```

Arrays and pointers

- We can access an array in two ways:
  - using an index, e.g. LD R16, array[N]
    (this method is not supported in AVR CPUs)
  - using a pointer to the memory address array[N]
- The index registers can be used for this purpose:

r27:r26	(X)
r29:r28	(Y)
r31:r30	(Z)

Example: adding the elements of an array

```
        EOR  r16, r16          ; i = 0
        EOR  r18, r18          ; sum = 0
        LDI  r27, high(array)  ; initialize X
        LDI  r26, low(array)
next:   LD  r17, X+            ; r17 = array[i]
        ADD  r18, r17
        INC  r16               ; i++
        CPI  r16, 10           ; if i < 10...
        BRCS  next             ; ...goto next
```

Subroutine call

We may call a subroutine in three ways:

- CALL (direct or long addressing),
- ICALL (indirect addressing),
- RCALL (relative or short addressing).

All three instructions have similar operation.

RCALL (relative or short call)

- Syntax:

  RCALL label

The assembler translates the label to a constant value k of 12 bits ($-2048 \leq k \leq +2047$) that express the distance from the current memory location to the destination.

1 1 0 1	k k k k	k k k k	k k k k

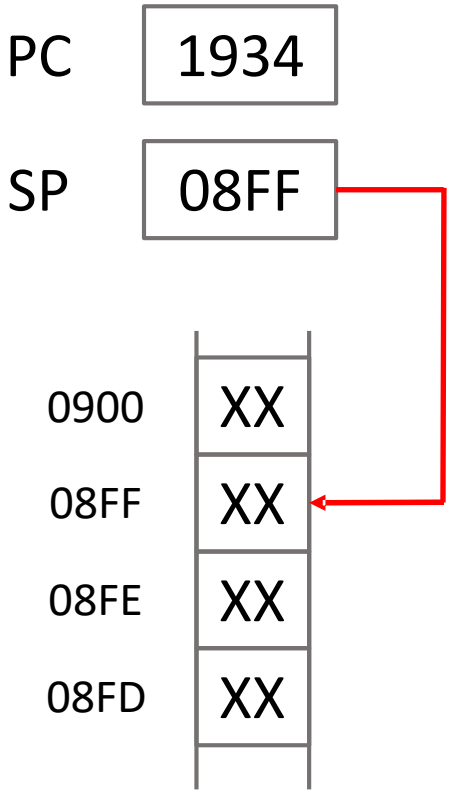
RCALL (cont.)

Operation:

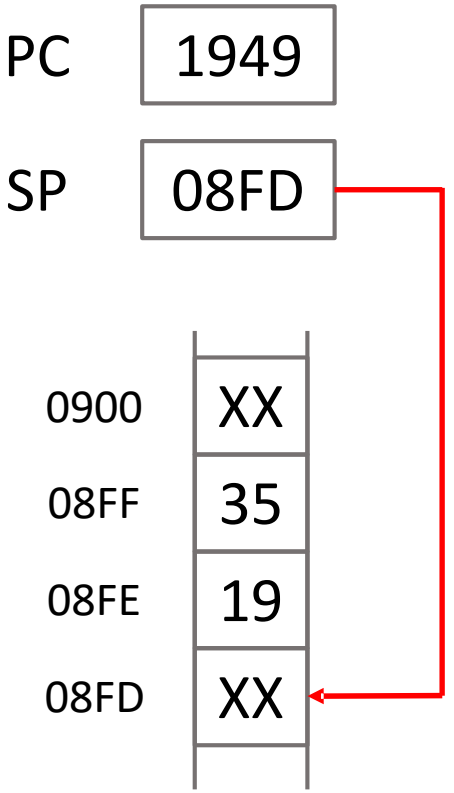
- The return address, PC + 1, is stored on the stack.
  This address occupies two bytes:
  PCL $\rightarrow$ (SP), PCH $\rightarrow$ (SP – 1).
- The stack pointer (δείκτης της στοίβας) decrements by 2
  in order to point at the first available position of the stack.
- The program counter (απαριθμητής προγράμματος)
  is incremented by k + 1 in order to point at the
  first instruction in the subroutine.

RCALL: example

Initial values:



After executing RCALL 14:



(instead of 1935)

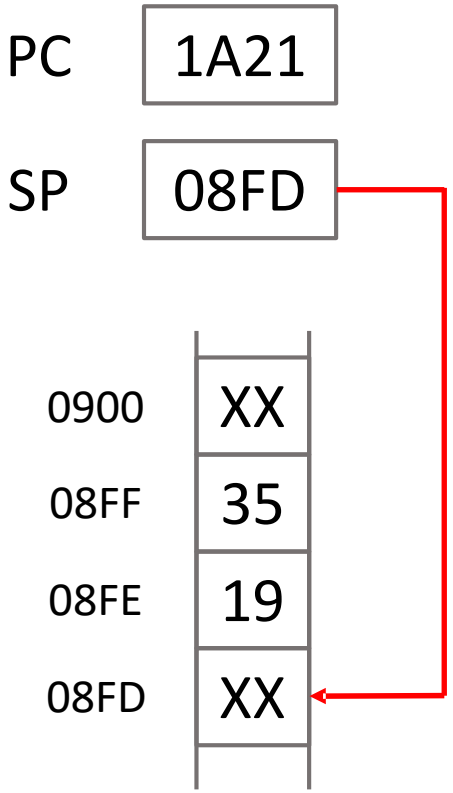
Return from a subroutine

The instruction to return from a subroutine is RET (just the op-code, no operands):

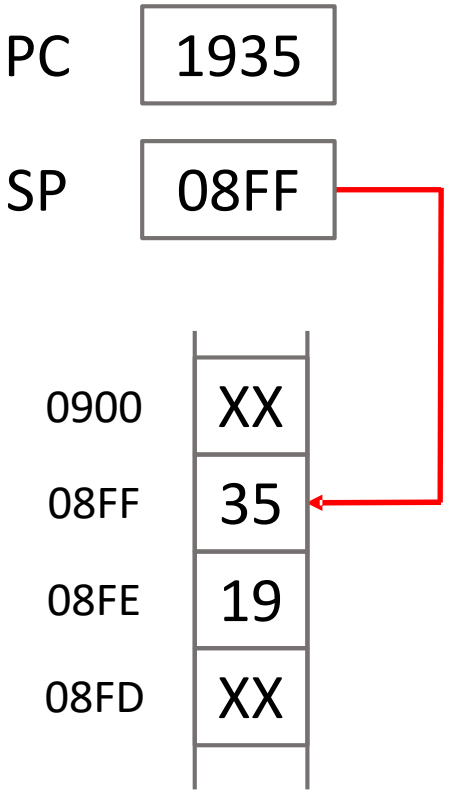
- The return address is extracted from the stack and transferred to the program counter. The PC does not increment by 1, as it usually does.
- The stack pointer increments by 2 and points again to the first available p0osition in the stack.

RET: example

Before RET:



After executing RET:



Parameter passing

We can pass arguments to and from the subroutine in three ways:

- using specific memory locations (for global variables)
- using specific registers (fastest),
- using the stack (the most general way).

Parameter passing using registers Example (main)

```
LDI r16, low(RAMEND)     ; set up the stack
OUT SPL, r16
LDI r16, high(RAMEND)
OUT SPH, r16
LDS r16, A               ; Input arguments
LDS r17, B
RCALL calc
STS C, r16               ; Output argument
```

Parameter passing using registers Example (subroutine)

```
calc:
        ; calculate C = 4A – 2B
        ADD r16, r16        ; 2*A
        ADD r16, r16        ; 4*A
        ADD r17, r17        ; 2*B
        SUB  r16, r17       ; 4*A - 2*B
RET
```

Parameter passing using the stack

PUSH Rr

- The value of Rr is placed in the first available location of the stack: Rr $\rightarrow$ (SP)
- The stack pointer is decremented by 1.

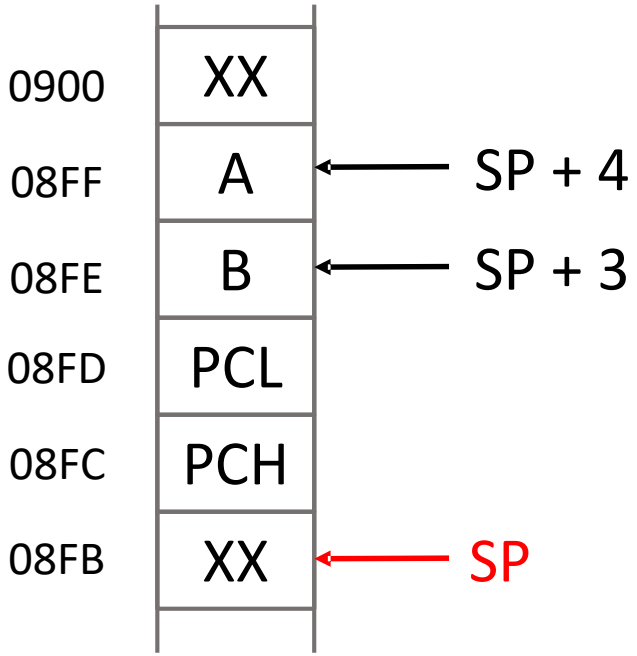
POP Rd

- The value of the register is extracted from the stack: Rd $\leftarrow$ (SP + 1).
- SP is incremented by 1.

Example (main)

```
LDS r16, A
PUSH r16      ; Place A in stack
LDS r16, B
PUSH r16      ; Place B in stack
RCALL calc
POP r16       ; Get to C position in stack
POP r16       ; Extract C from stack
STS C, r16
```

The stack after executing CALL



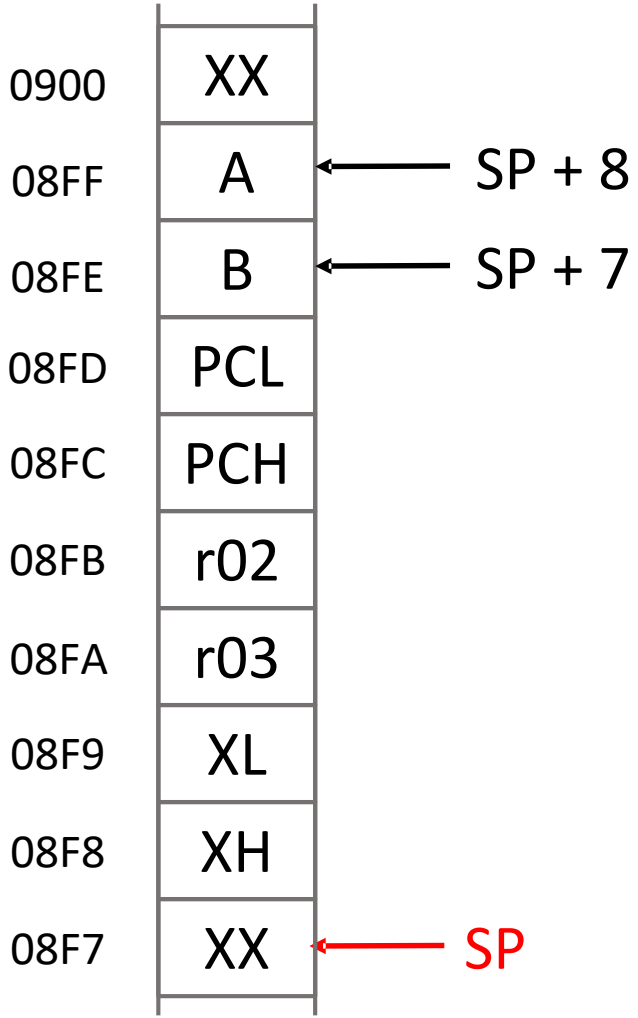
Example (subroutine)

```
calc:   IN XH, SPH
        IN XL, SPL      ; X ← SP
        ADIW X, 3       ; X points to 2nd argument
        LD r02, X+      ; get 2nd argument
        LD r03, X       ; get 1st argument
        ADD r03, r03            ; carry out calculations
        ADD r03, r03
        ADD r02, r02
        SUB r03, r02            ; finish calculations
        ST X, r03               ; place result in stack
        RET
```

Example (subroutine) – additions

```
calc:   PUSH r02      ; Save in the stack all registers
        PUSH r03      ; used by the routine
        PUSH XL
        PUSH XH
        IN XH, SPH
        IN XL, SPL    ; X ← SP
        ADIW X, 7     ; X points to 2nd argument
        LD r02, X+    ; get 2nd argument
        LD r03, X     ; get 1st argument
        ADD r03, r03          ; carry out calculations
```

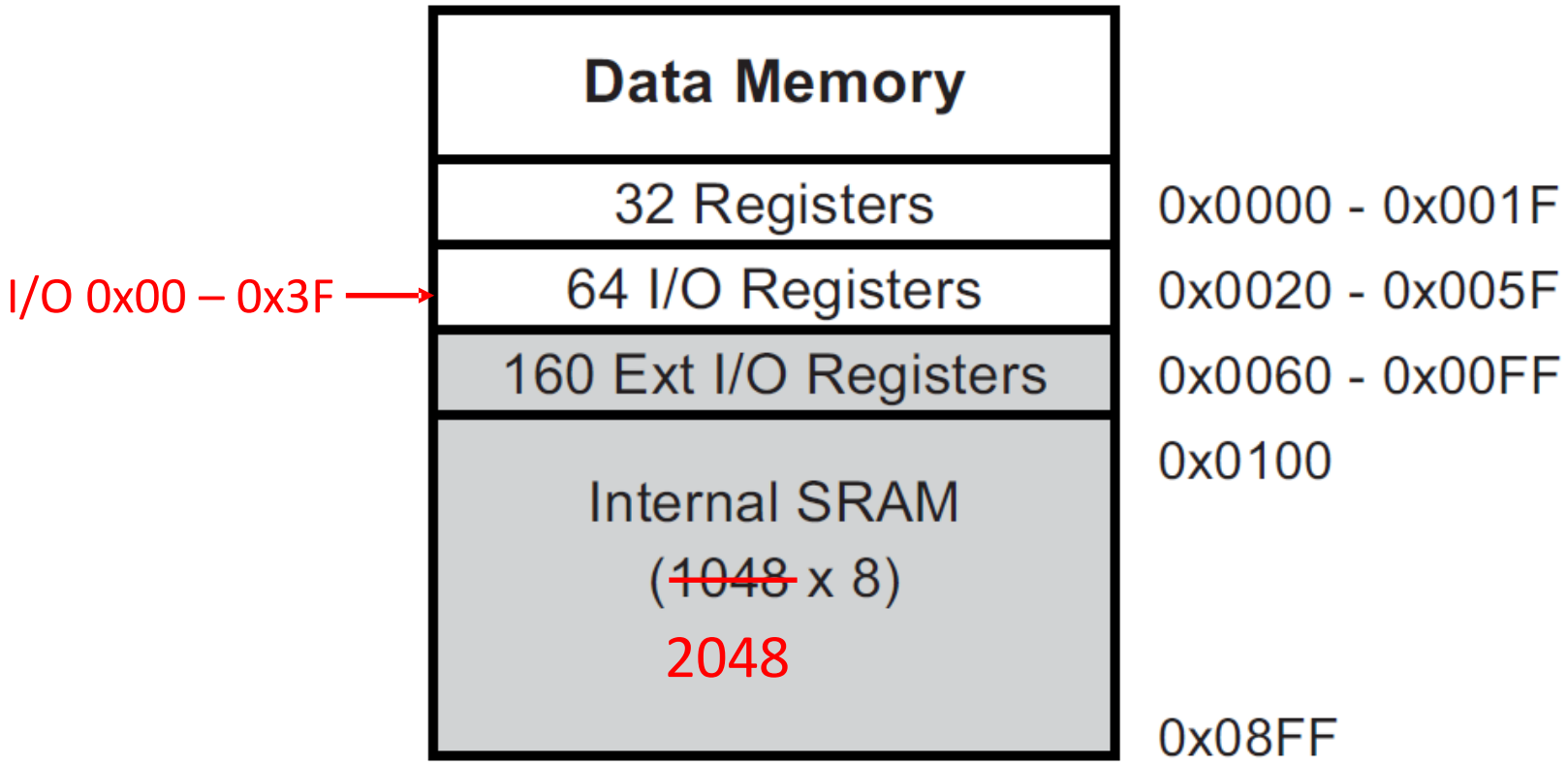
The stack viewed from within the routine



Example (subroutine) – additions

```
calc:   ADD r03, r03
        ADD r02, r02
        SUB r03, r02        ; finish calculations
        ST X, r03           ; place result in stack
        POP XH              ; replace registers
        POP XL
        POP r03
        POP r02
        RET
```

Memory map of the ATMega328



Input-output instructions

- Data transfer from a special purpose register (I/O) to a general purpose register:

```
IN  Rd, A       ; Rd ← I/O(A)
                ; 0 ≤ d ≤ 31, 0 ≤ A ≤ 63 (3Fh)
```

- Data transfer from a GPR to an I/O register:

```
OUT  A, Rr      ; I/O(A) ← Rr
```

- Instead of IN, OUT we may use LD, ST instructions. But LD and ST are slower (2 words, 2 clock cycles).