Atmel's AVR 8-bit Microcontroller

Prof. Ben Lee

Oregon State University

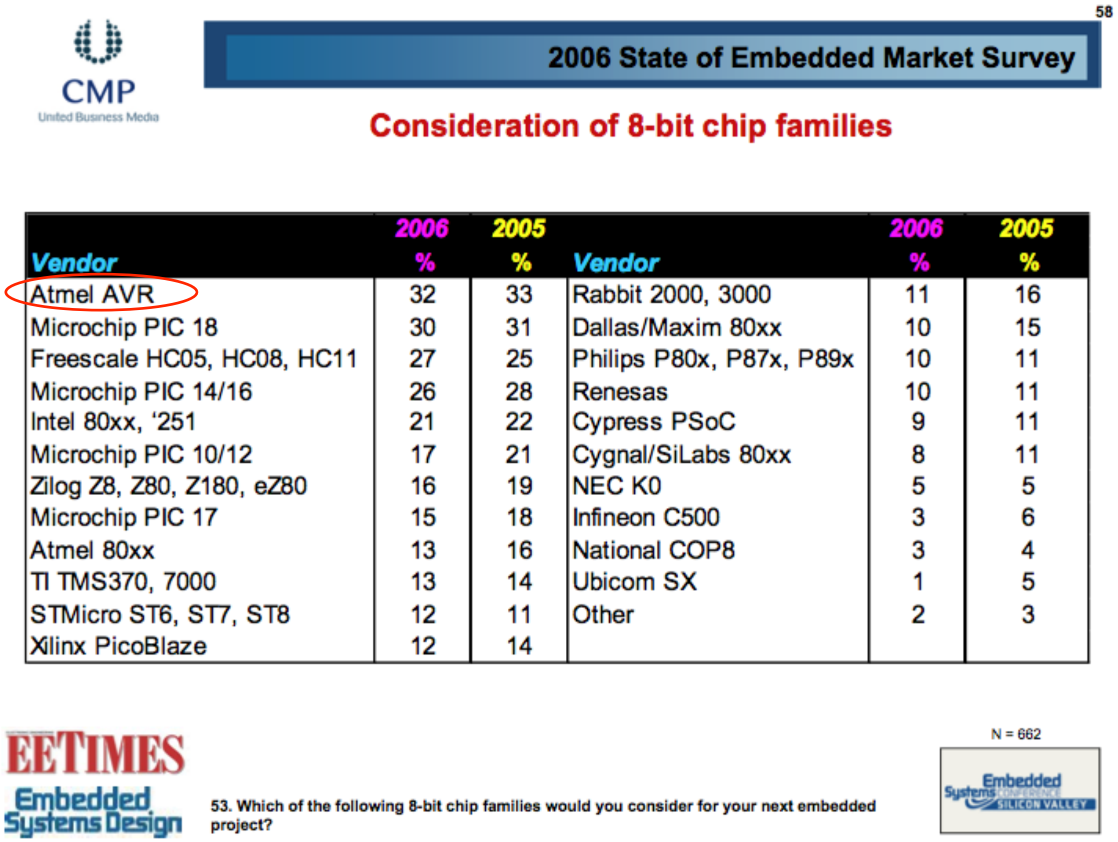
School of Electrical Engineering and
Computer Science

Why Microcontrollers?

- Ratio of Embedded Devices / Desktop PCs is greater than 100.
- The typical house may contain over 50 embedded processors.
- A high-end car can have over 50 embedded processors.
- Embedded systems account for most of the world’s production of microprocessors!

Why AVR Microcontroller?

The most popular 8-bit processor!!



58

2006 State of Embedded Market Survey

Consideration of 8-bit chip families

	2006	2005		2006	2005
Vendor	%	%	Vendor	%	%
Atmel AVR	32	33	Rabbit 2000, 3000	11	16
Microchip PIC 18	30	31	Dallas/Maxim 80xx	10	15
Freescale HC05, HC08, HC11	27	25	Philips P80x, P87x, P89x	10	11
Microchip PIC 14/16	26	28	Renesas	10	11
Intel 80xx, ‘251	21	22	Cypress PSoC	9	11
Microchip PIC 10/12	17	21	Cygnal/SiLabs 80xx	8	11
Zilog Z8, Z80, Z180, eZ80	16	19	NEC K0	5	5
Microchip PIC 17	15	18	Infineon C500	3	6
Atmel 80xx	13	16	National COP8	3	4
TI TMS370, 7000	13	14	Ubicom SX	1	5
STMicro ST6, ST7, ST8	12	11	Other	2	3
Xilinx PicoBlaze	12	14			

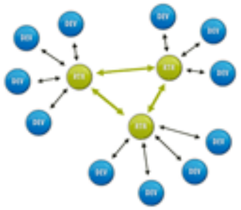
N = 662

53. Which of the following 8-bit chip families would you consider for your next embedded project?

Source: EETimes and Embedded Systems Design Magazine 2006 Embedded Market Survey

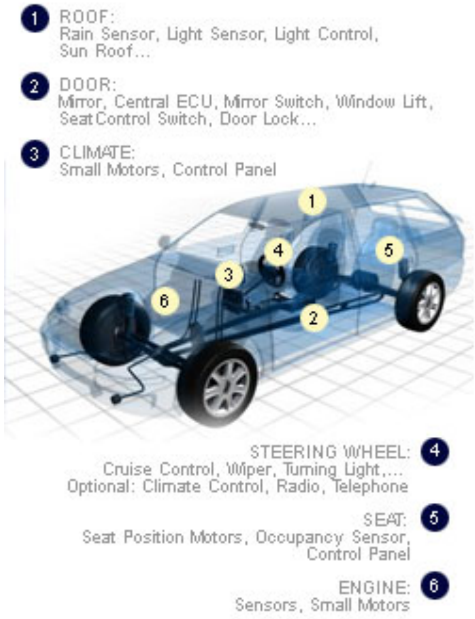
Some AVR-based Products

802.15.4/ZigBee
LR-WPAN Devices



Wireless Sensor Networks

Automotive Applications


1 ROOF: Rain Sensor, Light Sensor, Light Control, Sun Roof…
2 DOOR: Mirror, Central ECU, Mirror Switch, Window Lift, Seat Control Switch, Door Lock…
3 CLIMATE: Small Motors, Control Panel
1
4
5
3
6
2
STEERING WHEEL: Cruise Control, Wiper, Turning Light,… Optional: Climate Control, Radio, Telephone 4
SEAT: Seat Position Motors, Occupancy Sensor, Control Panel 5
ENGINE: Sensors, Small Motors 6

RFID

Motor Control

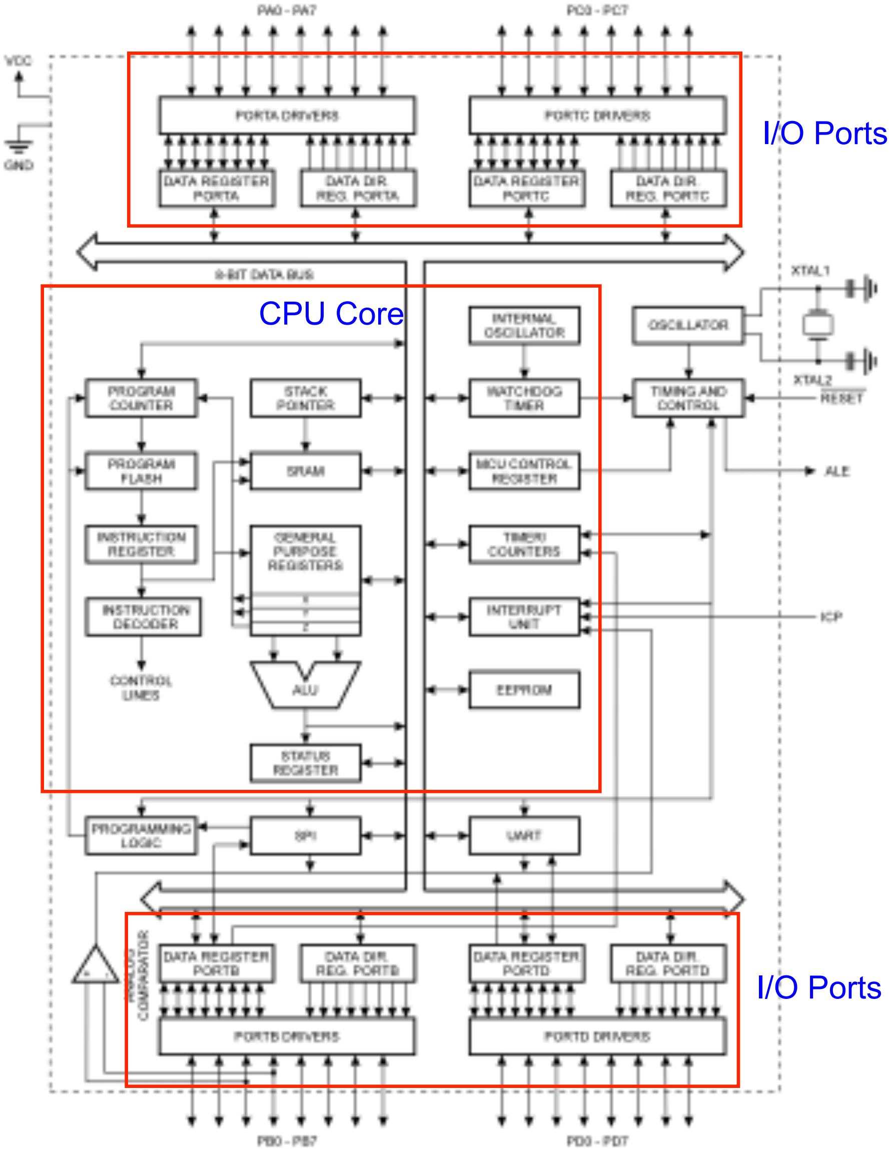
USB controller

Remote Access Controller

General Characteristics

- RISC (Reduced Instruction Set Computing) architecture.
  - Instructions same size
  - Load/store architecture
  - Only few addressing modes
  - Most instructions complete in 1 cycle
- 8-bit AVR Microcontroller
  - 8-bit data, 16-bit instruction
  - Used as embedded controller
    - Provides convenient instructions to allow easy control of I/O devices.
  - Provides extensive peripheral features
    - Timer/counter, Serial UART, Serial interface, etc.

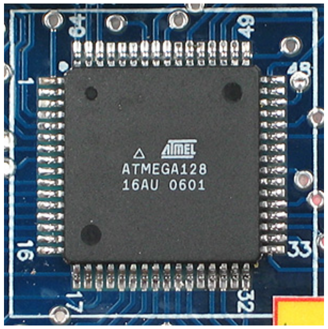
General Architecture



I/O Ports

CPU Core

I/O Ports

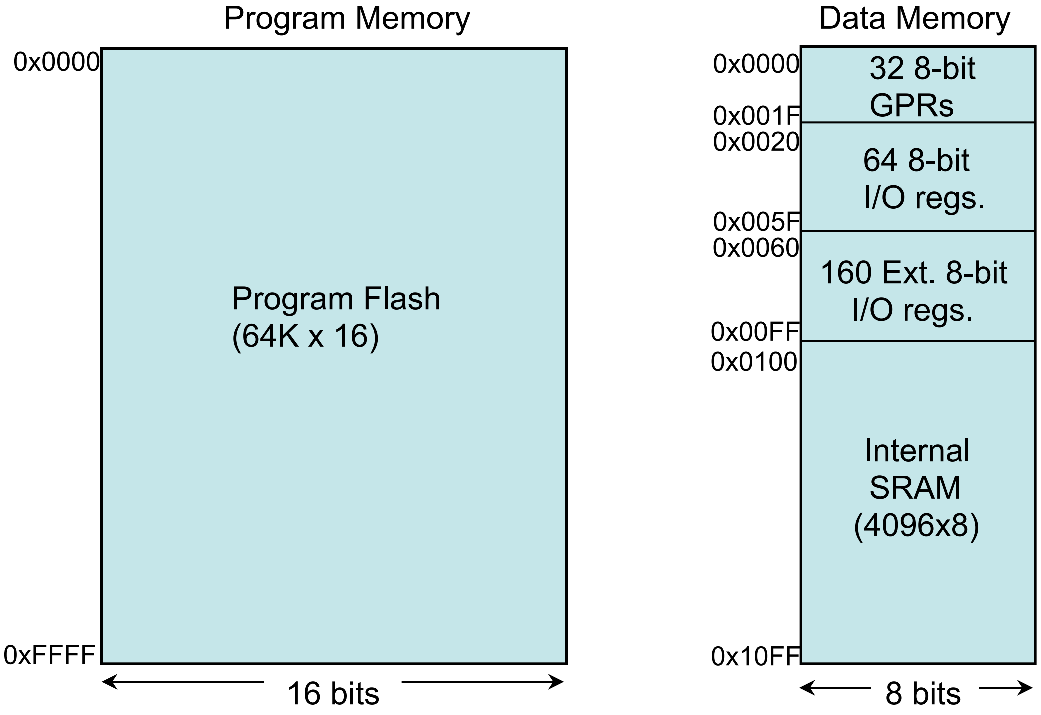


ATmega128

- Many versions of AVR processors.
- Our discussion based on ATmega128.
  - See AVRStarterGuide.pdf (Lab website)
  - See http://www.atmel.com/Images/doc2467.pdf (373 pages!)
- ATmega128 characteristics:
  - 128-Kbyte program memory on-chip
  - 4-Kbyte on-chip SRAM data memory (expandable to 64-Kbyte external)
  - 7 I/O ports

Memory

- Separate Program Memory and Data Memory.
  - FLASH memory for program => non-volatile
  - SRAM for data => volatile



Program Memory

0x0000

Program Flash
(64K x 16)

0xFFFF

16 bits

Data Memory

0x0000

32 8-bit GPRs

0x001F

0x0020

64 8-bit I/O regs.

0x005F

0x0060

160 Ext. 8-bit I/O regs.

0x00FF

0x0100

Internal SRAM (4096x8)

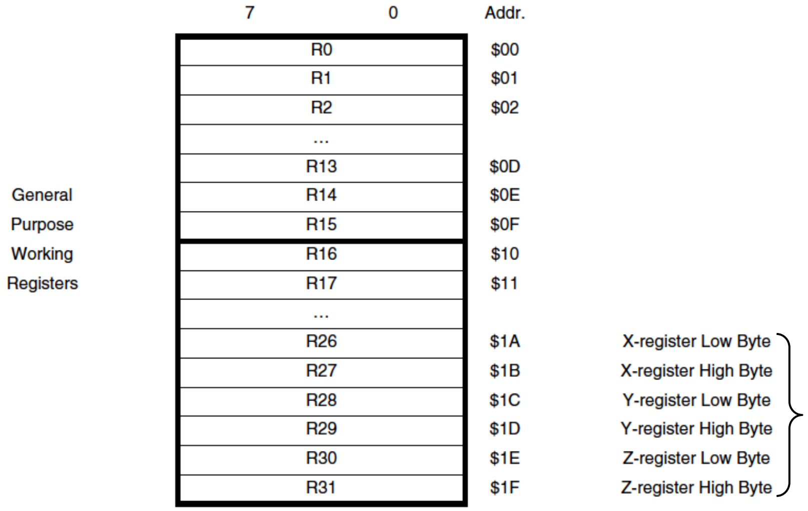
0x10FF

8 bits

Registers

- 32 8-bit GPRs (General Purpose Registers)
  - R0 - R31
- 16-bit X-, Y-, and Z-register
  - Used as address pointers
- PC (Program Counter)
- 16-bit SP (Stack Pointer)
- 8-bit SREG (Status Register)
- 7 8-bit I/O registers (ports)

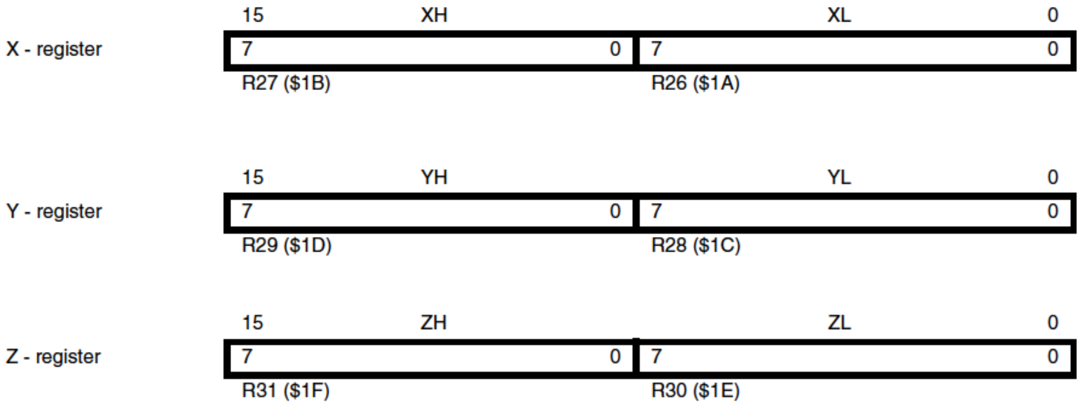
GPRs and X, Y, Z Registers



General Purpose Working Registers

7 0	Addr.	
R0	$00	
R1	$01	
R2	$02	
…		
R13	$0D	
R14	$0E	
R15	$0F	
R16	$10	
R17	$11	
…		
R26	$1A	X-register Low Byte
R27	$1B	X-register High Byte
R28	$1C	Y-register Low Byte
R29	$1D	Y-register High Byte
R30	$1E	Z-register Low Byte
R31	$1F	Z-register High Byte

X, Y, Z registers are mapped to R26-R31



Status Register

I	T	H	S	V	N	Z	C

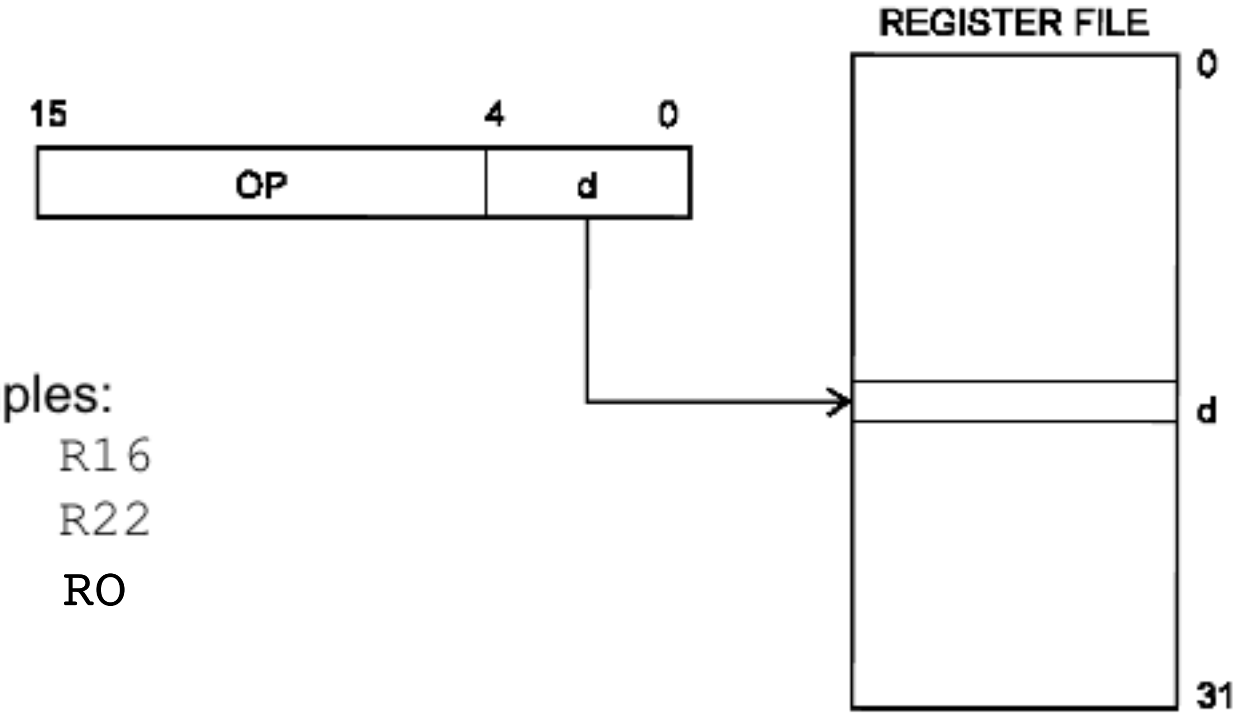
I/O Register $3F

- I - Global Interrupt Enable
- T - Bit Copy Storage
- H - Half Carry Flag
- S - Sign Bit
- V - 2’s Complement Overflow Flag
- N - Negative Flag
- Z - Zero Flag
- C - Carry Flag

Addressing Modes

- Addressing mode defines the way operands are accessed.
- Gives the programmer flexibility by providing facilities such as
  - Pointers to memory, counters for loop control, indexing of data, and program relocation
- Addressing modes in AVR:
  - Register (with 1 and 2 registers)
  - Direct
  - Indirect
    - w/Displacement (also referred to as indexed)
    - Pre-decrement
    - Post-increment
  - Program Memory Constant Addressing
  - Direct Program Memory Addressing
  - Indirect Program Memory Addressing
  - Relative Program Memory Addressing

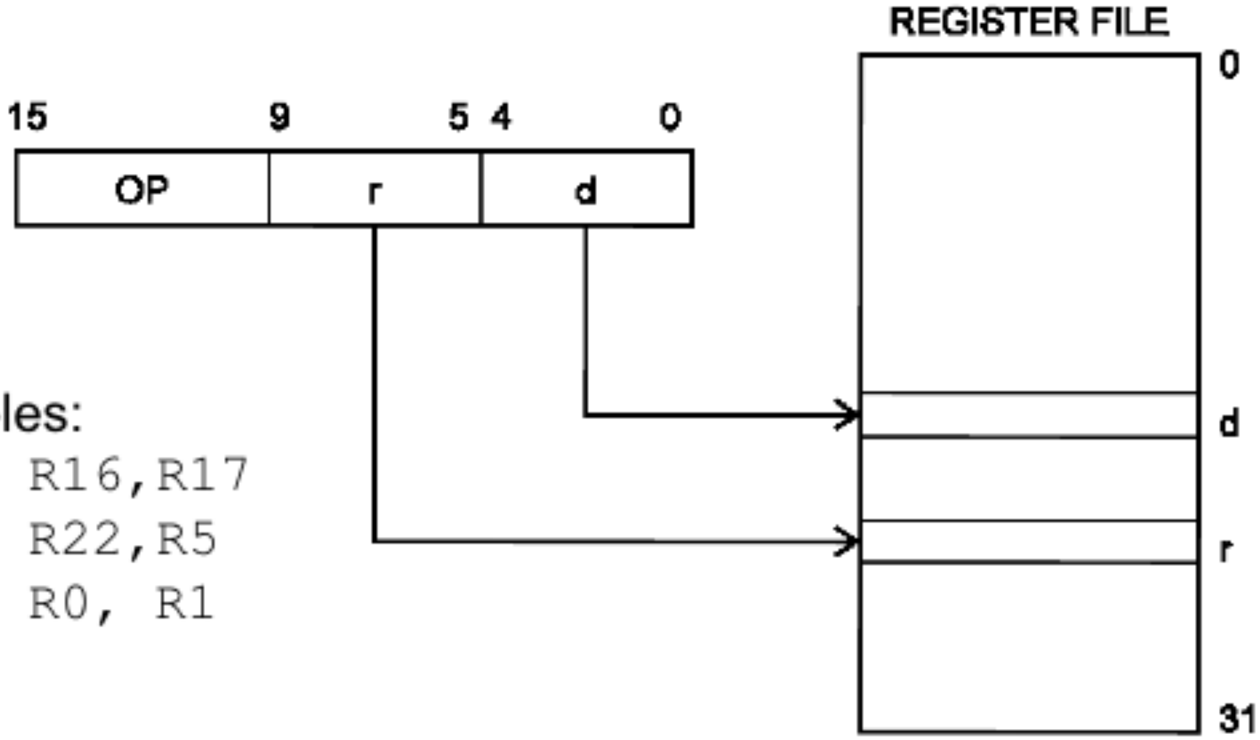
Register Addressing (1 register)



Examples:

```
INC    R16
CLR    R22
DEC    RO
```

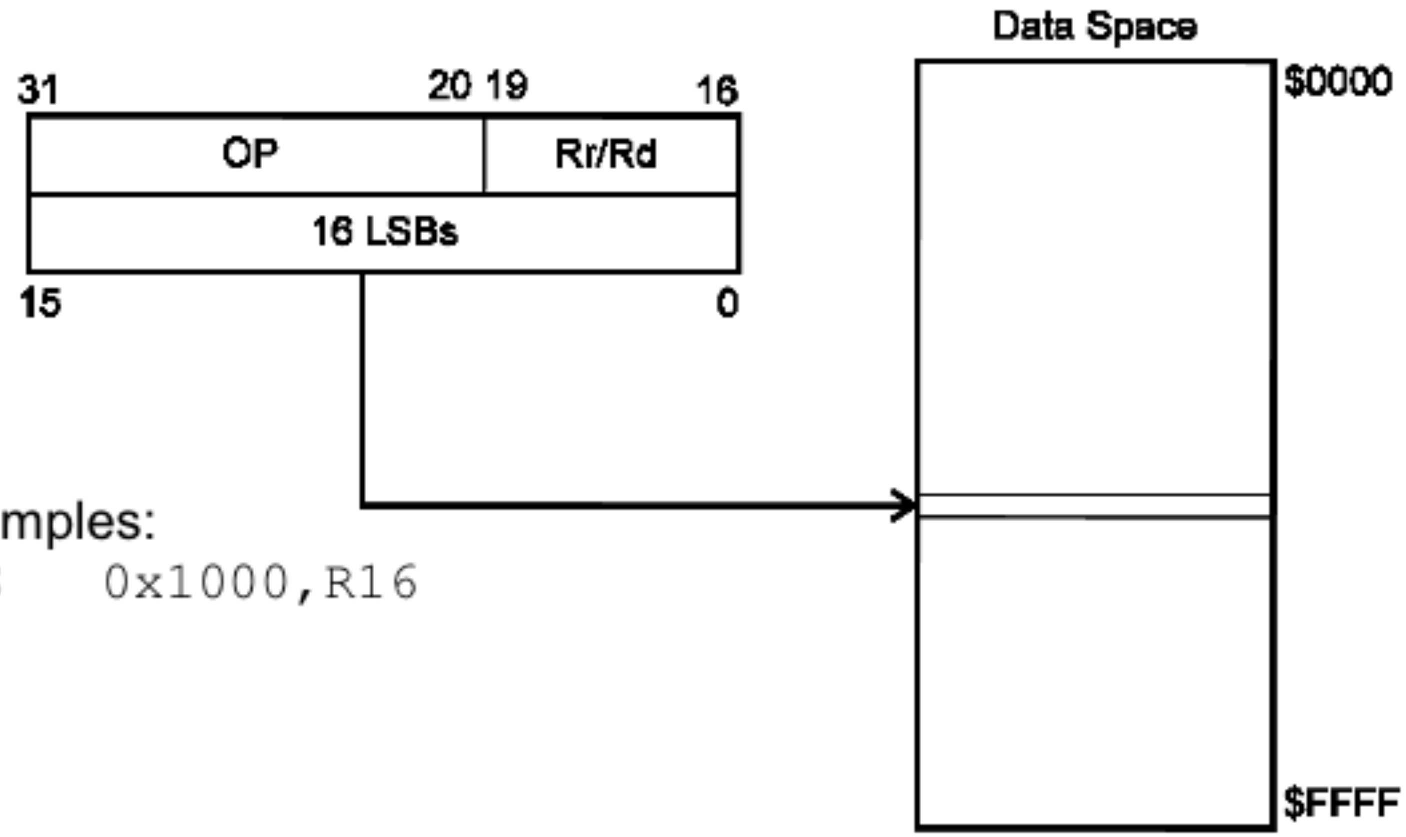
Register Addressing (2 register)



Examples:

```
ADD    R16,R17
CP     R22,R5
MOV    R0, R1
```

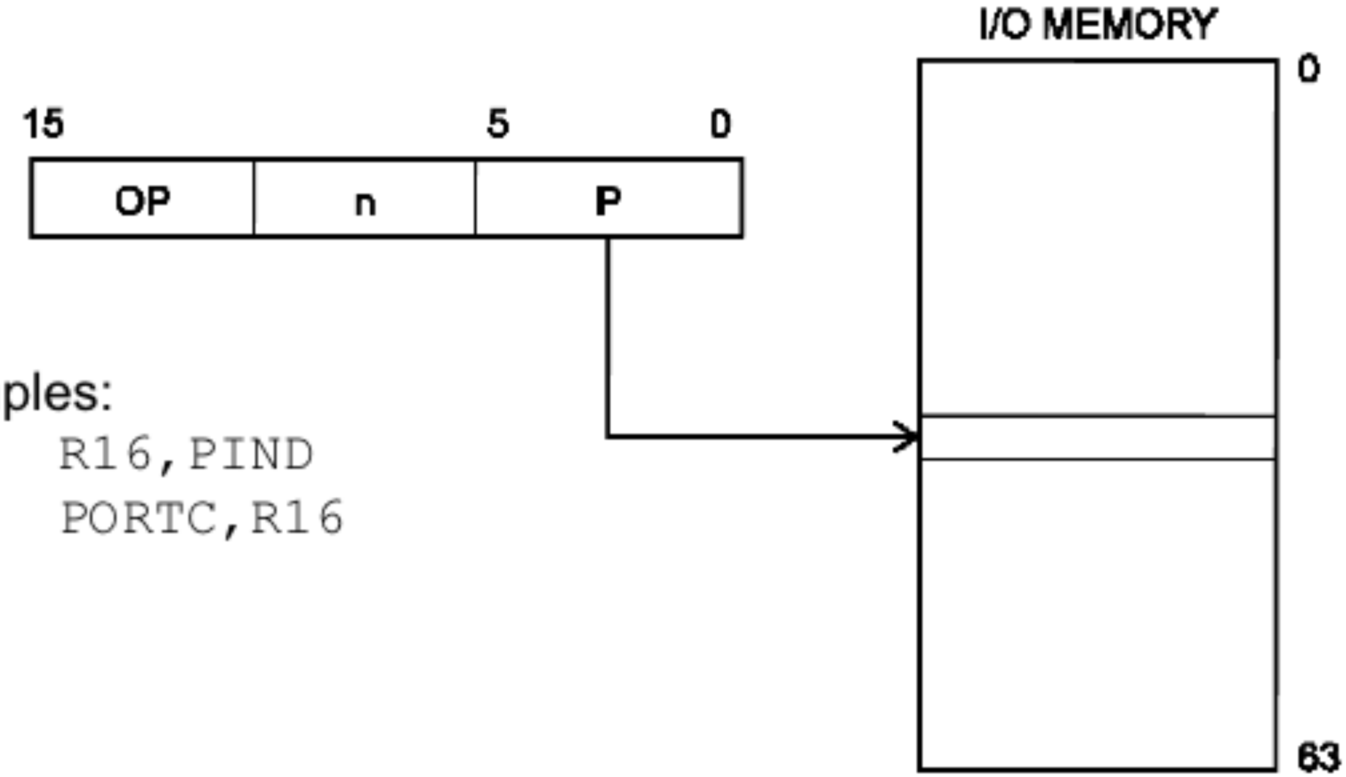
Direct Addressing



Examples:

```
STS    0x1000,R16
```

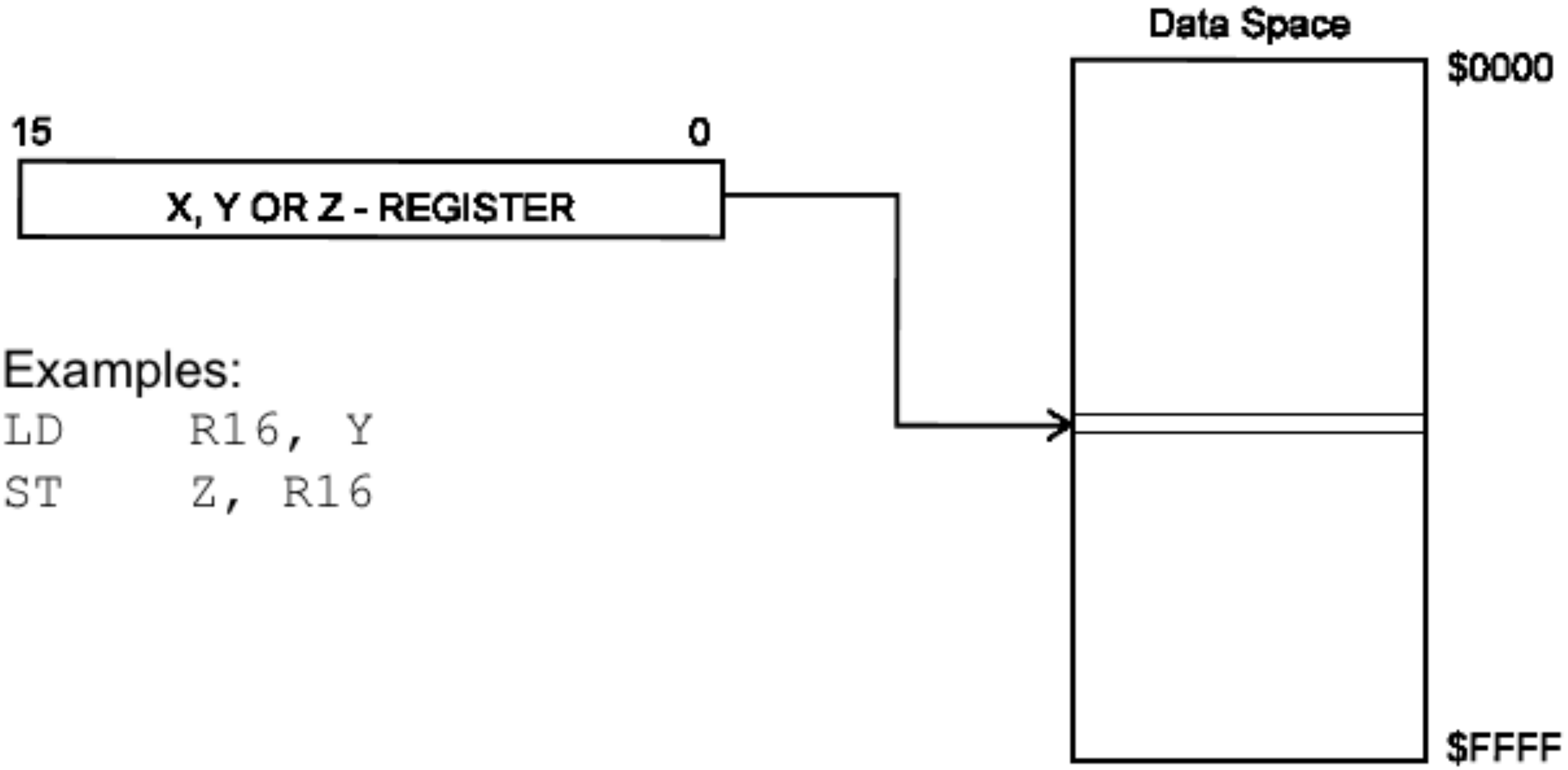
I/O Direct Addressing



Examples:

```
IN     R16,PIND
OUT    PORTC,R16
```

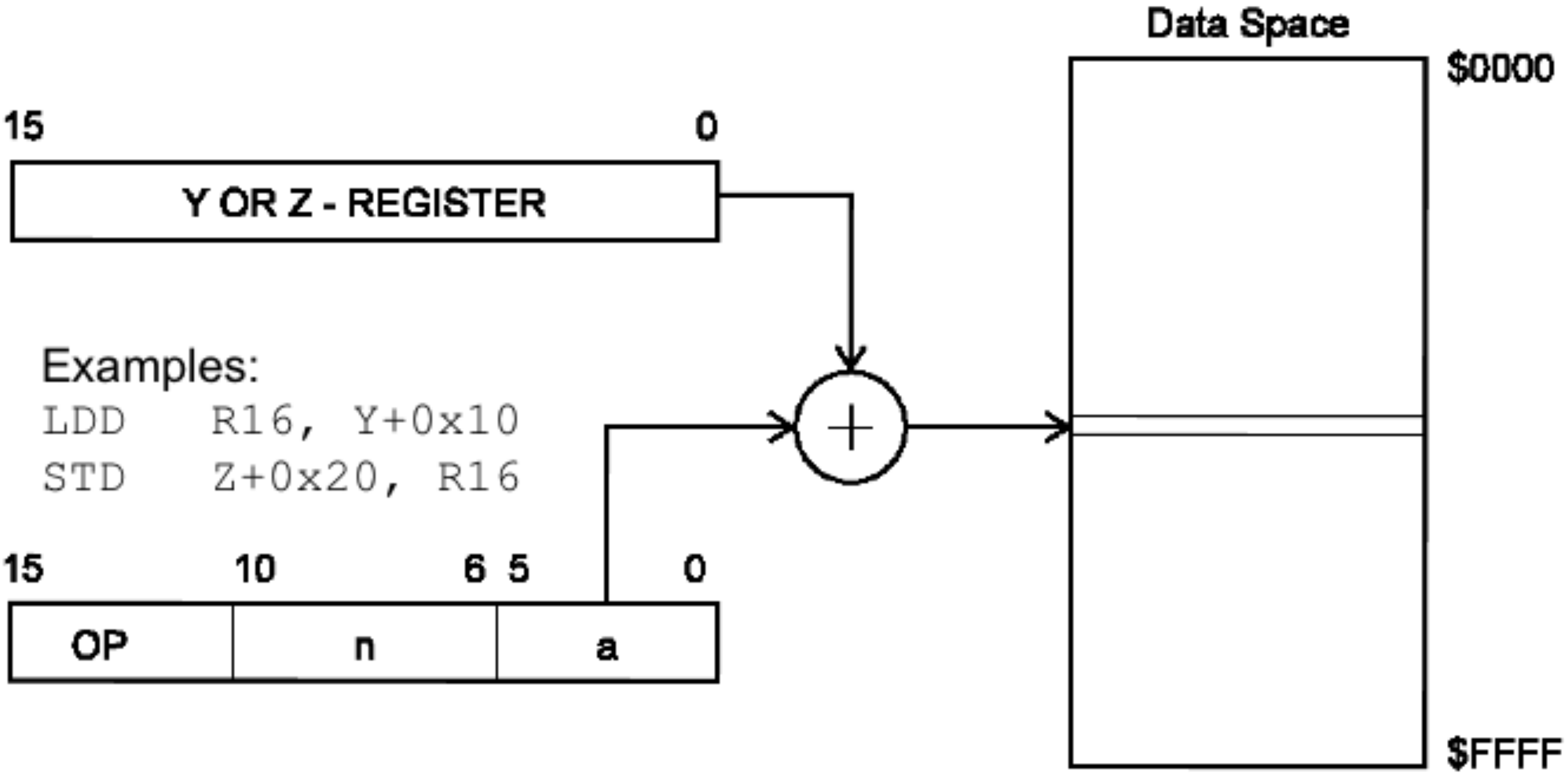
Indirect Addressing



Examples:

```
LD    R16, Y
ST    Z, R16
```

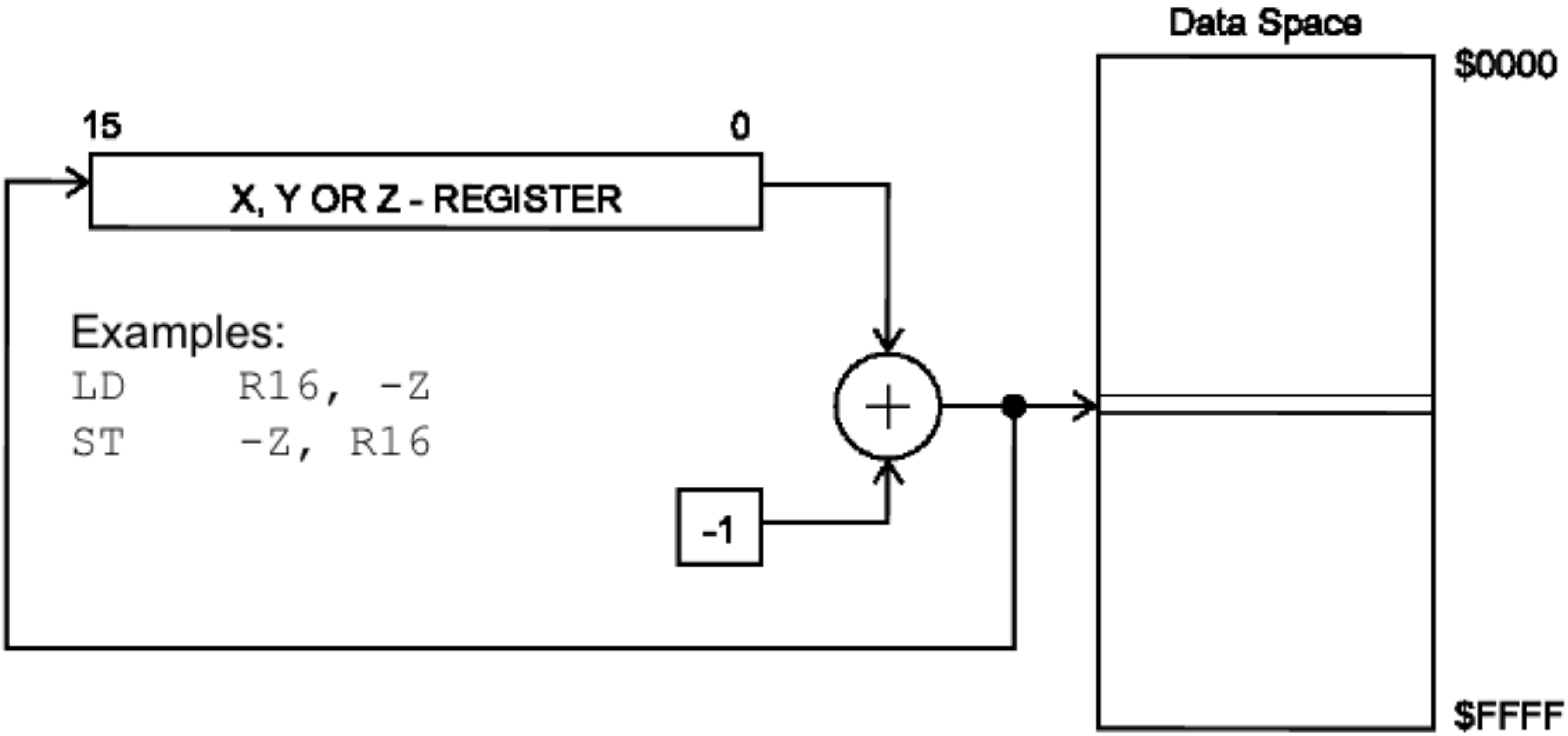
Indirect with Displacement



Examples:

```
LDD    R16, Y+0x10
STD    Z+0x20, R16
```

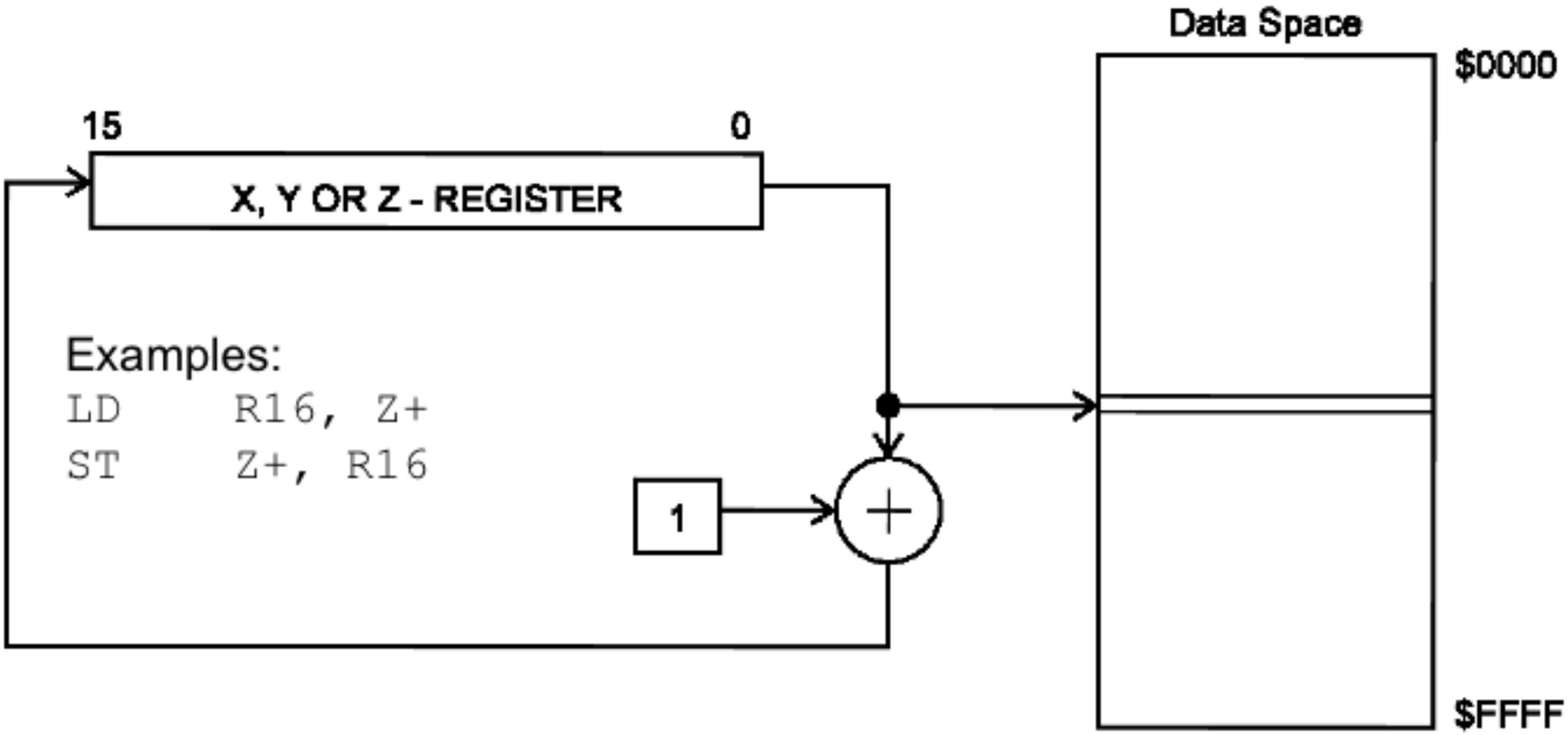
Indirect with Pre-Decrement



Examples:

```
LD     R16, -Z
ST     -Z, R16
```

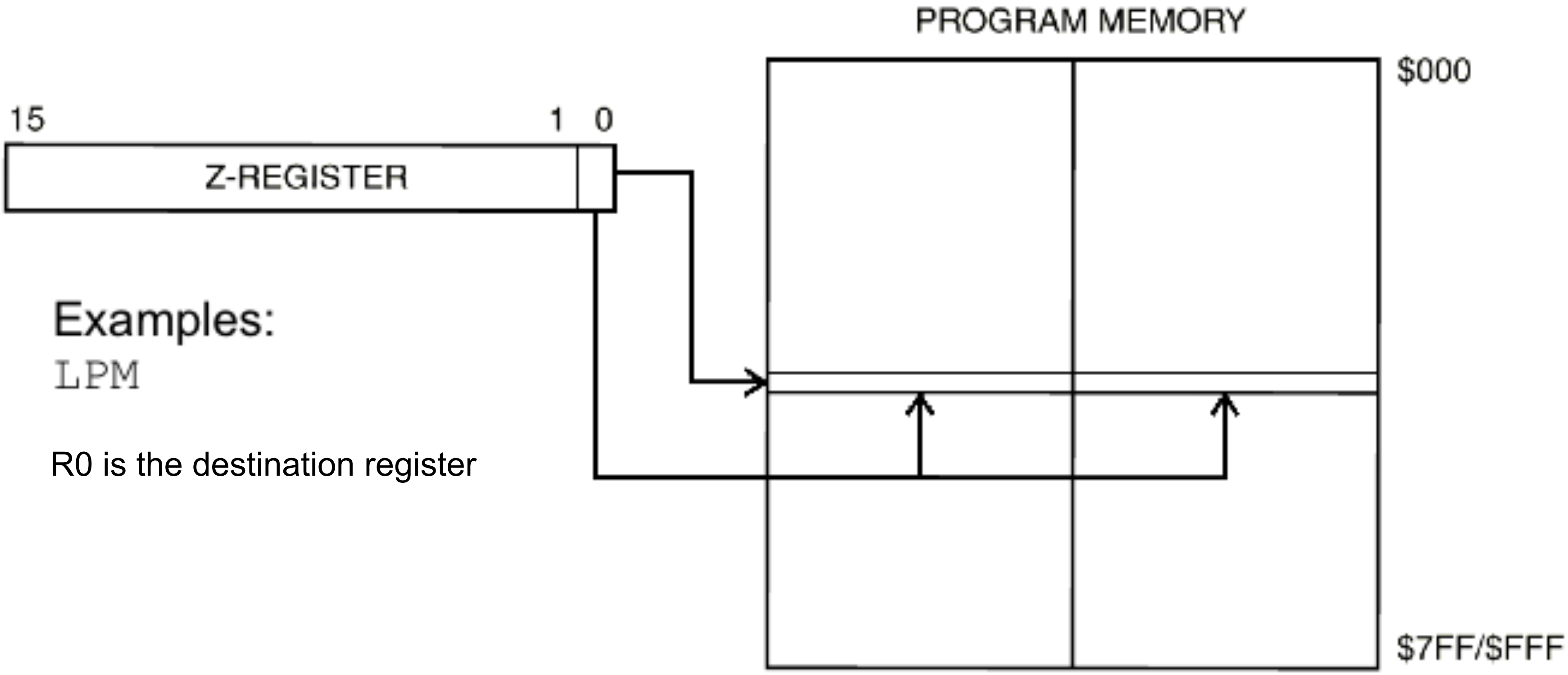
Indirect with Post-Increment



Examples:

```
LD    R16, Z+
ST    Z+, R16
```

Program Memory Addressing

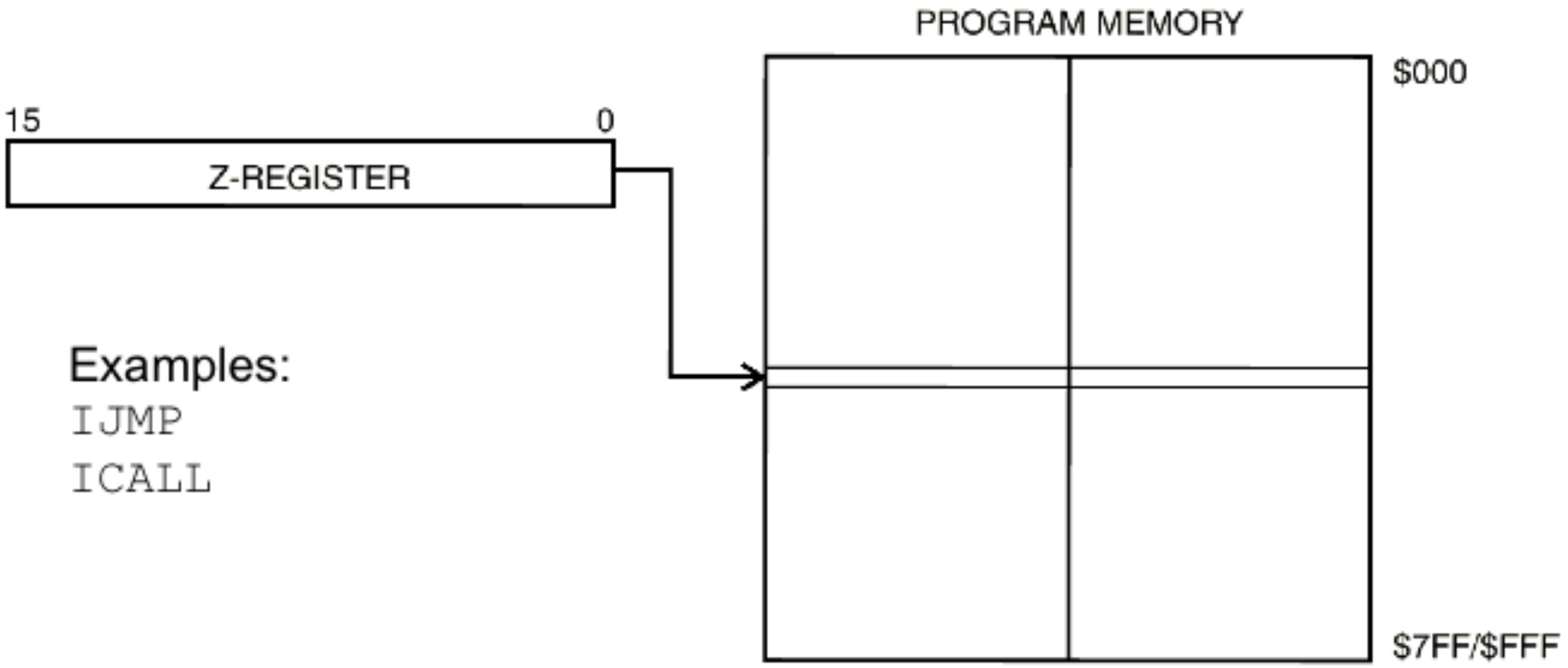


Examples:

`LPM`

R0 is the destination register

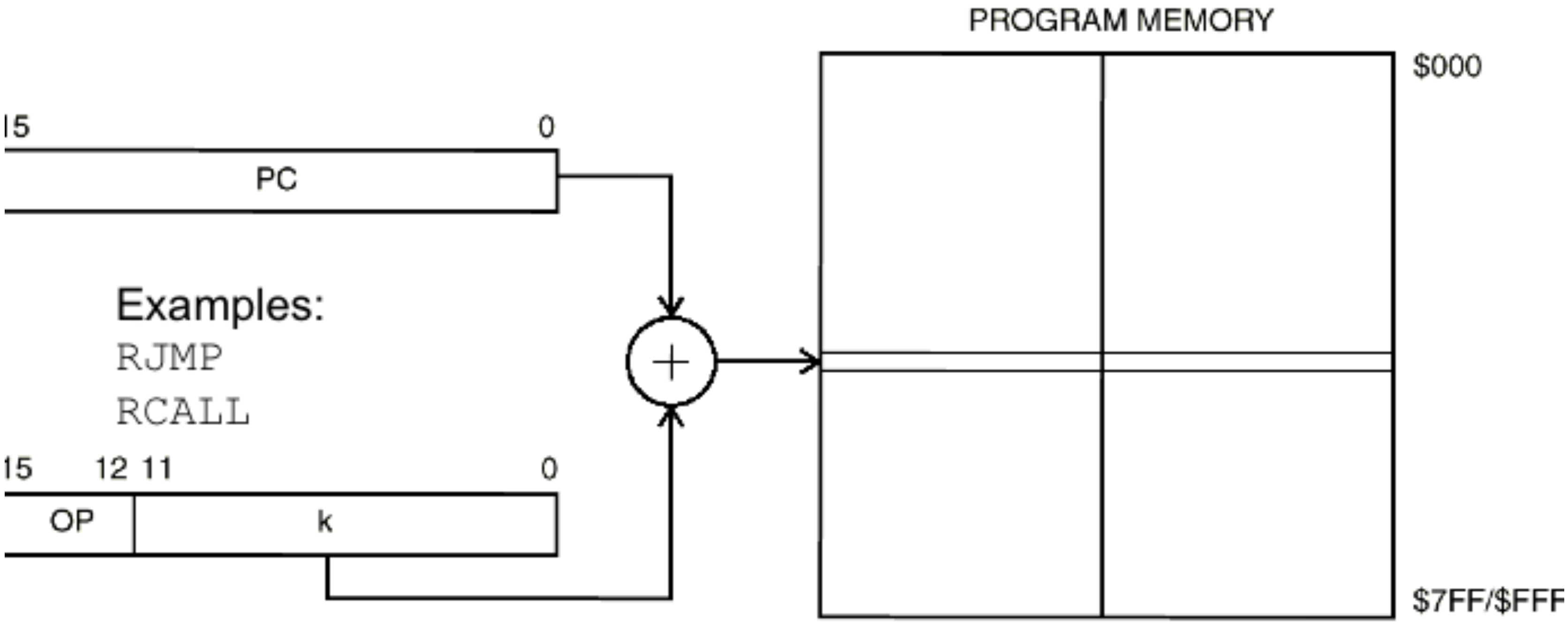
Indirect Program Addressing



Examples:

```
IJMP
ICALL
```

Relative Addressing



Examples:

RJMP

RCALL

Called PC relative jump

AVR Instructions

- AVR has 133 different instructions
- Instruction Types
  - Data Transfer
  - Arithmetic and Logic
  - Control Transfer (branch/jump)
  - Bit and bit-test

Data Transfer Instructions (1)

- MOV - transfers data between two registers.
  - MOV Rd, Rr
- LD - loads data from memory or immediate value (Indirect Addressing):
  - LD *dest*, *src* (Load)
    - *dest* = Rd
    - *src* = X, X+, -X, Y, Y+, -Y, Z, Z+, -Z
  - LDD *dest*, *src* (Load with Displacement)
    - *dest* = Rd
    - *src* = Y+displacement, Z+displacement
  - LDI Rd, *immediate* (Load Immediate)
    - Binary => 0b00001010
    - Decimal => 10
    - Hexadecimal => 0X0A, $0A
  - LDS Rd, k (Load SRAM)

Data Transfer Instructions (2)

- LPM - load program memory
  - LPM *dest*, *src*
    - *dest* = Rd
    - *src* = Z, Z+
  - LPM
    - *dest* = R0 (implied)
    - *src* = Z (implied)
- ST - stores data to memory
  - ST *dest*, *src*
    - *dest* = X, X+, -X, Y, Y+, -Y, Z, Z+, -Z
    - *src* = Rr
  - STD *dest*, *src*
    - *dest* = Y+displacement, Z+displacement
    - *src* = Rr
  - STS k, Rr

Data Transfer Instructions (3)

- IN/OUT - input/output
  - IN Rd, A
  - OUT A, Rr
- PUSH/POP - stack operations
  - PUSH Rr
  - POP Rd

Data Transfer Examples (1)

- LD Rd, Y
  - LD R16, Y; R16 ← M(Y)
  - Y is implied in the opcode

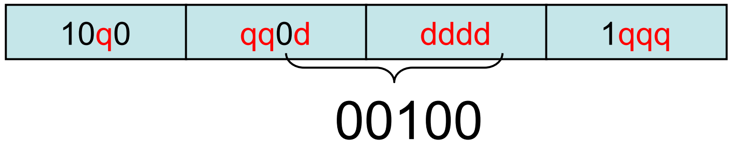
$$1000\quad 000\underbrace{d\quad dddd}_{10000}\quad 1001$$

- LDI Rd, K
  - LDI R30, 0xF0; R30 ← 0xF0 (hexadecimal)
  - Destination register can only be R16-R31
  - 8 bit constant K ($0 \leq K \leq 255$)

$$1110\quad \underbrace{KKKK}_{1111}\quad dddd\quad \underbrace{KKKK}_{0000}$$

Data Transfer Examples (2)

- LDD Rd, Y+q
  - LDD R4, Y+2; R4 ← M(Y+2)
  - Y is implied in the opcode
  - 6 bit displacement q (0 ≤ q≤ 63)

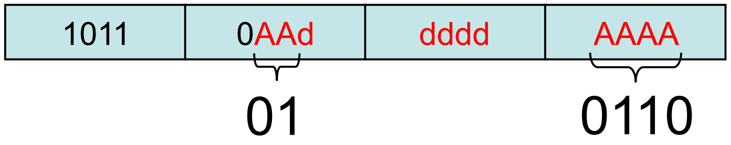


10q0	qq0d	dddd	1qqq

00100

qqqqqq=000010

- IN Rd, A
  - IN R25, $16; R25 ← I/O($16)
  - $16 is an I/O register connected to Port B (PIN7-PIN0)
  - 6 bit A (0 ≤ A≤ 63) => 64 I/O registers.



1011	0AAd	dddd	AAAA

01

0110

ALU Instructions (1)

Arithmetic instructions

- ADD/SUB
  - ADD *dest*, *src*
  - *dest* = Rd
  - *src* = Rr
  - Also ADC/SBC -> add/subtract with carry
- MUL - Multiply unsigned
  - MUL Rd, Rr
  - R1<-Result(high), R0<-Result(low)
  - Also MULS -> multiply signed
- INC/DEC - increment/decrement register
- CLR/SER - clear/set register

ALU Instructions (2)

<u>Logic instructions</u>

- AND/ANDI - logical AND & /w immediate
  - AND Rd, Rr
  - ANDI Rd, *immediate*
- OR/ORI - logical OR & /w immediate
  - OR Rd, Rr
  - ORI Rd, *immediate*
  - Also EOR -> exclusive OR
- COM/NEG - one’s/two’s complement
  - COM RD
- TST - test for zero or minus
  - TST Rd

ALU Examples

- MUL Rd, Rr
  - MUL R15, R16; R1:R0 ← R15 x R16
  - Both operand unsigned (signed version => MULS)
  - Product high and low stored in R1 and R0, respectively.

1001	11rd	dddd	rrrr

rrrrr = 10000

ddddd = 01111

Branch Instructions

- BR*cond* - Branch on condition
  - BR*cond*   address
  - *cond* = EQ, NE, CS, CC, SH, LO, MI, PL, GE, LT, HS, HC, TS, TC, VS, VC, IE, ID
- CP - compare
  - CP   Rd, Rr
  - CPC   Rd, Rr
  - CPI   Rd, immediate
- COM/NEG - one's/two's complement
  - COM   RD
- TST - test for zero or minus
  - TST   Rd
- …many more

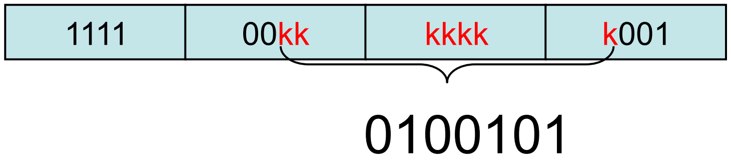
Branch Example

- Example
  - Branches are PC-relative: if (Z=1) then PC <-PC + address +1

```
0232              CP       R0, R0   ; compare
0233              BREQ     SKIP
0234              next inst.
                  …
                  …
0259     SKIP:    …
```

0259H - 0234H = 0025H

BREQ SKIP =>



1111	00kk	kkkk	k001

0100101

Address range => -64 ≤ k ≤ +63

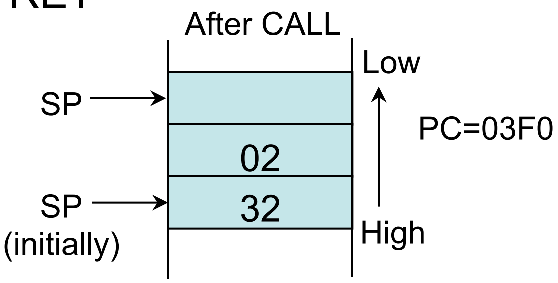
Jump Instruction

- JMP instructions
  - JMP address
  - Also, RJMP, IJMP
- CALL/RET - subroutine call/return
  - CALL address
  - RET
  - Stack implied
  - Also RETI -> return from interrupt
- …and few more

Jump Example

- Example
  - Subroutines are implemented using CALL and RET

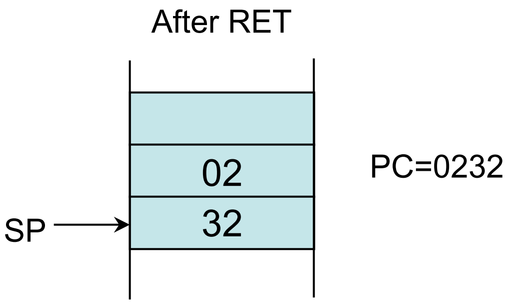
```
0230                CALL     SUBR
0232                next inst.


03F0     SUBR:      …
                             {my subroutine}
                             …
                             RET
```



  - CALL is 32-bit instruction

1001	010k	kkkk	111k
kkkk	kkkk	kkkk	kkkk

$\underbrace{\hspace{6cm}}_{0000001111110000}$



Bit and Bit Test Instructions

- LSL/LSR - Logical Shift Left/Right
  - LSL Rd
- ROL/ROR - Rotate Left/Right through carry
  - ROL Rd
- SE*flag*/CL*flag* - Set/Clear flag
  - *flag* = C, N, Z, I, S, V, T, H
  - SEZ => set zero flag
- …many more

AVR Assembly Directives

- Called pseudo opcodes
  - ORG: tells the assembler where to put the instructions that follow it.

    ```
    .ORG 0x37
    ```
  - EXIT: tells the assembler to stop assembling the file
  - EQU: Assigns a value to a label.

    Syntax: .EQU label = expression

    ```
    .EQU io_offset = 0x23
    ```
  - BYTE: reserves memory in data memory

    Syntax: LABEL: .BYTE expression

    ```
    var:    .BYTE 1
    ```
  - DB: Allows arbitrary bytes to be placed in the code.

    Syntax: LABEL: .DB expressionlist

    ```
    consts: .DB 0, 255, 0b01010101, -128, 0xaa
    Text: .DB “This is a text.”
    ```
  - DW: Allows 16-bit words to be placed into the code

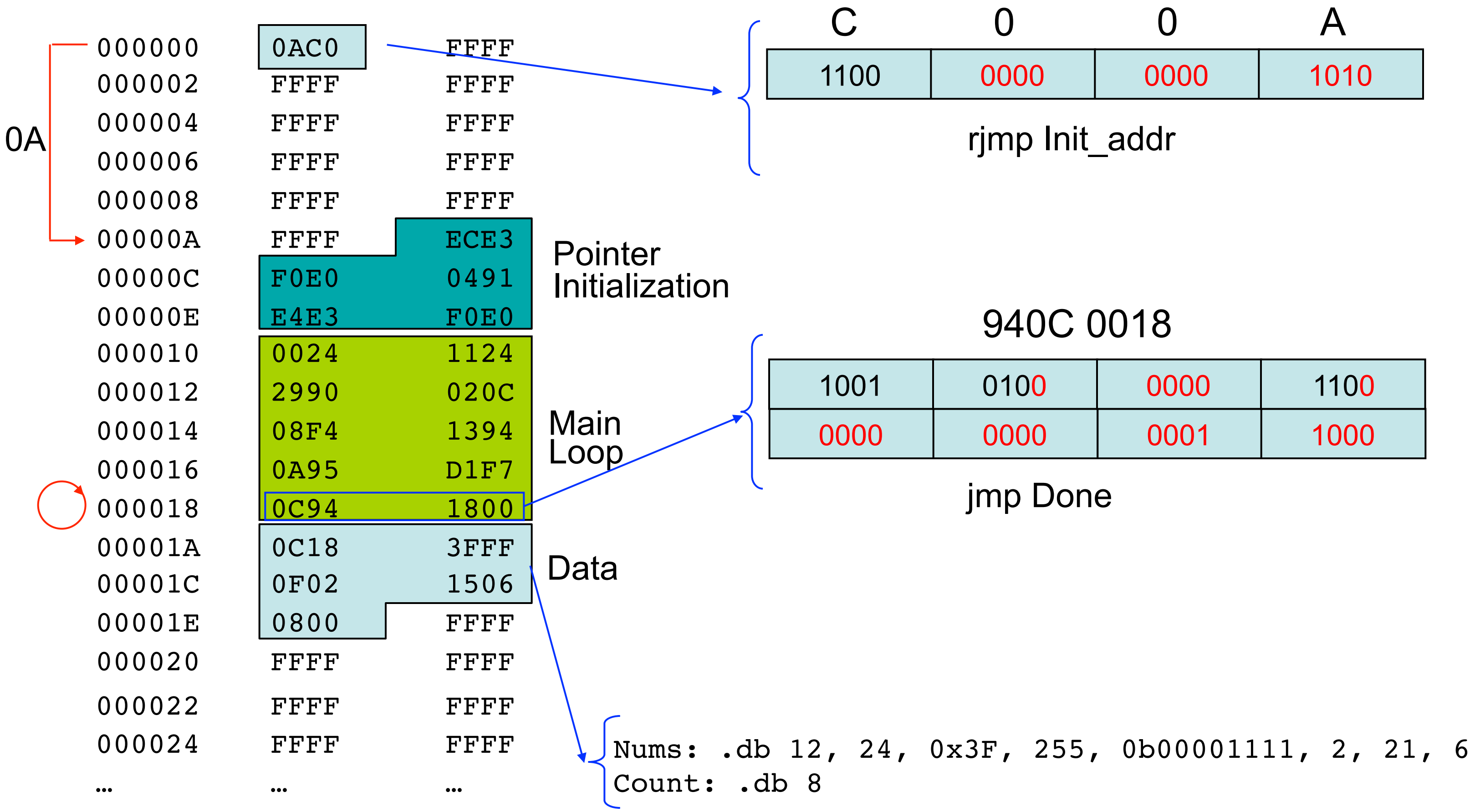
    Syntax: LABEL: .DW expressionlist

    ```
    varlist: .DW 0,0xffff,0b1001110001010101,-32768,65535
    ```

Addition of 8 Numbers

```
            .include   “m128def.inc”
            .ORG       $0000
            rjmp       Init_addr
            .ORG       $000B
Init_addr:  ldi        ZL,low(Count<<1)    ;
            ldi        ZH,high(Count<<1)   ;
            lpm        R16,Z               ; R16 =8
            ldi        ZL,low(Nums<<1)     ;
            ldi        ZH,high(Nums<<1)    ; Z points to 12
Init_acc:   clr        R1                  ; Accumulate the result in R1(H) and R0(L)
            clr        R0                  ;
Loop:       lpm        R2,Z+               ; Load data to R2 and post inc. pointer
            add        R0, R2              ; Add R2 to R0(L)
            brcc       Skip                ; No carry, skip next step
            inc        R1                  ; Add carry R1(H)
Skip:       dec        R16                 ; Count down the # words to add
            brne       Loop                ; If not done loop
Done:       jmp        Done                ; Done.  Loop forever.
Nums:       .DB        12, 24, 0x3F, 255, 0b00001111, 2, 21, 6
Count:      .DB        8
```

Program Memory



```
000000    0AC0    FFFF
000002    FFFF    FFFF
000004    FFFF    FFFF
000006    FFFF    FFFF
000008    FFFF    FFFF
00000A    FFFF    ECE3
00000C    F0E0    0491
00000E    E4E3    F0E0
000010    0024    1124
000012    2990    020C
000014    08F4    1394
000016    0A95    D1F7
000018    0C94    1800
00001A    0C18    3FFF
00001C    0F02    1506
00001E    0800    FFFF
000020    FFFF    FFFF
000022    FFFF    FFFF
000024    FFFF    FFFF
…         …       …
```

C	0	0	A
1100	0000	0000	1010

rjmp Init_addr

940C 0018

1001	0100	0000	1100
0000	0000	0001	1000

jmp Done

```
Nums: .db 12, 24, 0x3F, 255, 0b00001111, 2, 21, 6
Count: .db 8
```

Main Loop

```
; Assume Z points to the first word & R16 points to # of words

Init_acc:
      clr         R1        ; Accumulate the result in R1(H) and R0(L)
      clr         R0
Loop:
      lpm         R2,Z+     ; Load data to R2 and post inc. pointer
      add         R0 R2     ; Add R2 to R0(L)
      brcc        Skip      ; No carry, skip next step
      inc         R1        ; Add carry R1(H)
Skip:
      dec         R16       ; Count down the # words to add
      brne        Loop      ; If not done loop
Done:
      jmp         Done      ; Done.  Loop forever.
```

Pointer Initialization

- Where are the 8 numbers & number of words stored and how do we point to it?
- Depends on where data is stored: program memory or data memory.

Stored in Program memory as constants:

```
    .ORG         $000B
    ldi  ZL,low(Count<<1)       ;
    ldi  ZH,high(Count<<1)      ;
    lpm  R16,Z                  ;R16 =8
    ldi  ZL,low(Nums<<1)        ;
    ldi  ZH,high(Nums<<1)       ;Z points to 12
    …
NUMS:
    .DB  12, 24, 0x3F, 255, 0b00001111, 2, 21, 6
COUNT:
    .DB  8
```

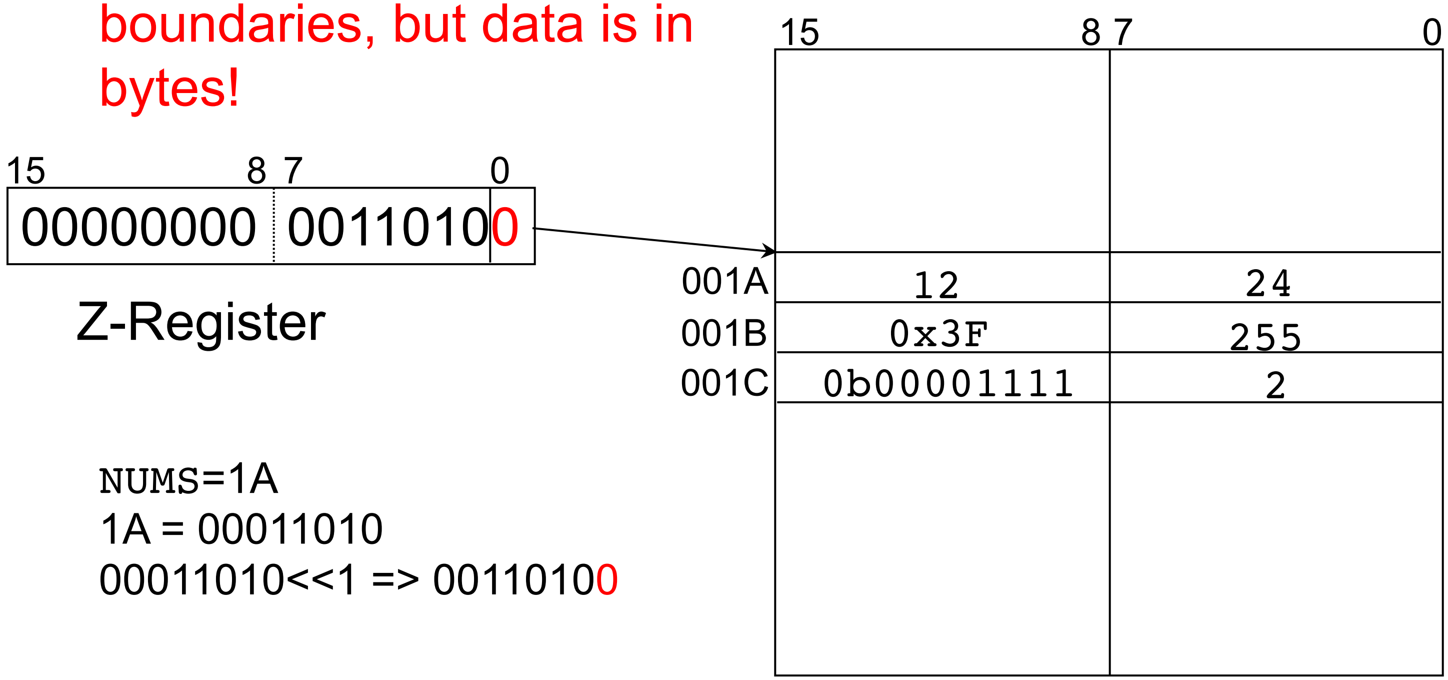
`low()`and `high()` are macros that extracts low and high byte (defined in “`m128def.inc`”)

Pointer Initialization

- “<<“ means shift left (multiply by 2)

Instructions are on word (2 bytes) boundaries, but data is in bytes!

Program Memory



	15 8	7 0
001A	12	24
001B	0x3F	255
001C	0b00001111	2

`NUMS`=1A
1A = 00011010
00011010<<1 => 00110100

- We could have also used `high(2*NUMS)` and `low(2*NUMS)`

Data Memory

- What if data is not predefined (i.e., constants)? Then, we need to allocate space in data memory using `.BYTE`.

Data memory:

```
Nums:
   .BYTE       8; Sets storage in data memory
Count:
   .BYTE       1; Data generated on the fly or read in
   ldi  YL,low(Count)  ;
   ldi  YH,high(Count) ;
   ld   R16,Y          ;R16 =8
   ldi  YL,low(Nums)   ;
   ldi  YH,high(Nums)  ;Y points to first word
   …
   …
Loop:
   ld   R2,Y+          ; Load data and post inc. pointer
   …
```

Result

- Registers R1 (H) and R0 (L) hold the result.

12

24

63 (3F)

255

15 (0b00001111)

2

21

+ 6

398 => 018E

R1=01 & R0=8E

Testing for Overflow

- Suppose we want to detect overflow and call a subroutine (e.g., to print error message)

```
   …
Loop:
   lpm  R2,Z+   ; Load data to R2 and post inc. pointer
   add  R0, R2  ; Add R2 to R0(L)
   brcc Skip    ; No carry, skip next step
   inc  R1      ; Add carry R1(H)
   brvc Skip    ; If V=0 branch to skip
                ; (Branch if overflow cleared)
   rcall        ERROR   ; We could also use CALL
Skip:  …
   …
   …
ERROR:
   …            ; My subroutine would go here
   RET
```