# Chapter 4: Atmel's AVR 8-bit Microcontroller
# Part 1 – Assembly Programming

Prof. Ben Lee
Oregon State University
School of Electrical Engineering and Computer Science

## Chapter Goals

- Understand how to program AVR assembly:
  - AVR architecture
  - Assembly syntax
  - Coding techniques
  - Relation between assembly and high-level language
- Bring us closer to processor hardware:
  - Assembly instructions to machine instructions
  - Binary code in memory

# Contents





# 4.1 Introduction

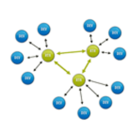
# Why Microcontrollers?

- 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 the most of the worlds production of microprocessors!
- This chapter discusses one of the most popular microcontrollers - Atmel's AVR 8-bit microcontrollers.



# Some AVR-based Products

802.15.4/ZigBee LR-WPAN Devices



Wireless Sensor Networks

Automotive Applications


RFID

Motor Control

USB controller

Remote Access Controller

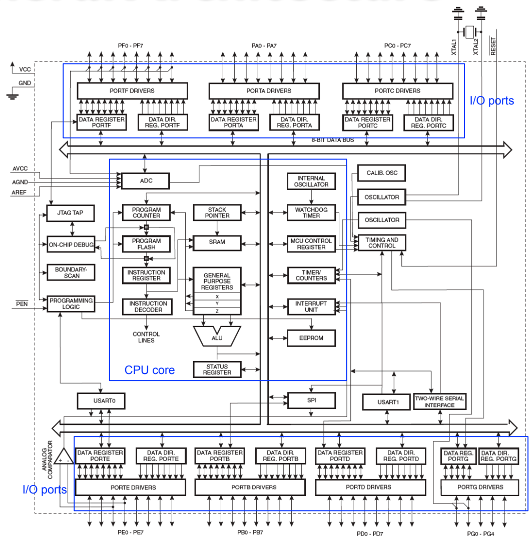
# 4.2 General Characteristics



# 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, USART, ADC, Serial interface, etc.

# General Architecture



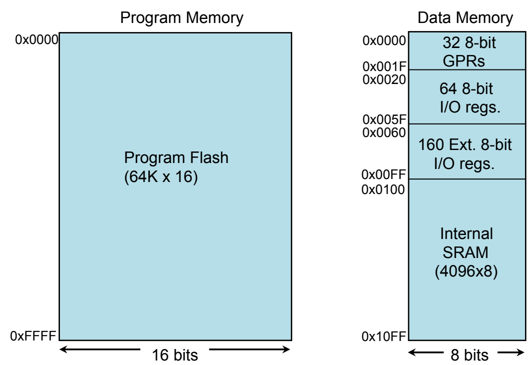



# ATmega128

- Many versions of AVR processors.
- Our discussion based on ATmega128
  - See my textbook
  - See http://www.atmel.com/atmel/acrobat/doc2467.pdf (386 pages!)
- ATmega128 characteristics:
  - 128-Kbyte on-chip Program Memory
  - 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

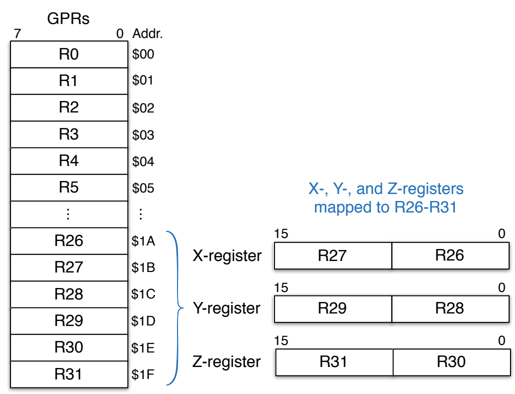





# Registers

- 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






# Status Register

**Status Register (SREG)** I/O Register $3F

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| I | T | H | S | V | N | Z | C |
| R/W (0) | R/W (0) | R/W (0) | R/W (0) | R/W (0) | R/W (0) | R/W (0) | R/W (0) |

Bit 7 - Global Interrupt Enable
Bit 6 - Bit- Copy Storage
Bit 5 - Half Carry Flag
Bit 4 - Sign Bit
Bit 3 - Two's Complement Overflow Flag
Bit 2 - Negative Flag
Bit 1 - Zero Flag
Bit 0 - Carry Flag

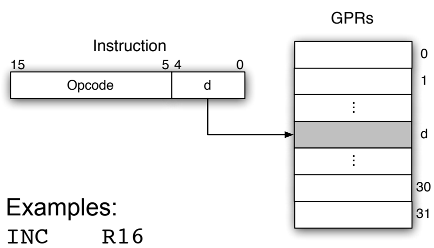
# 4.3 Addressing Modes



# 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)

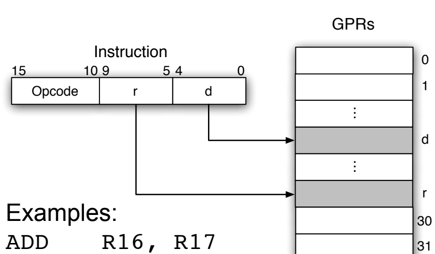



Examples:

```
INC    R16
DEC    R17
CLR    R22
```



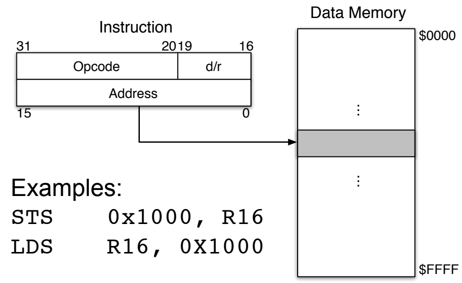
# Register Addressing (2 registers)




Examples:

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

# Direct Addressing

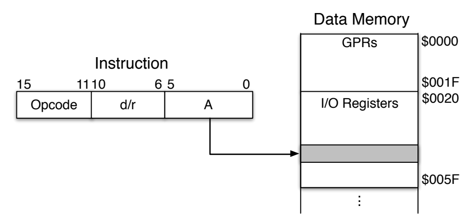



Examples:

```
STS    0x1000, R16
LDS    R16, 0X1000
```



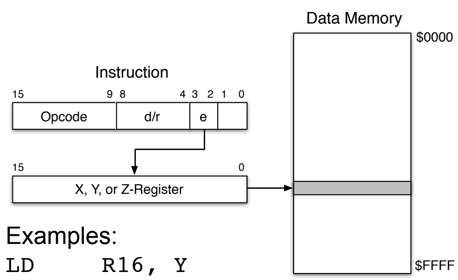
# I/O Direct Addressing




Examples:

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

# Indirect Addressing

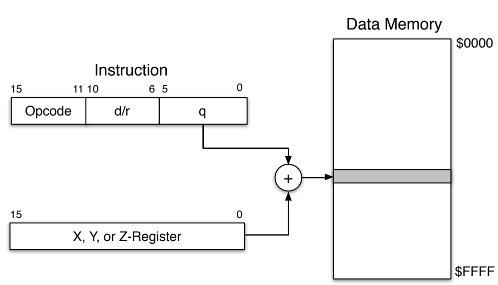



Examples:

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



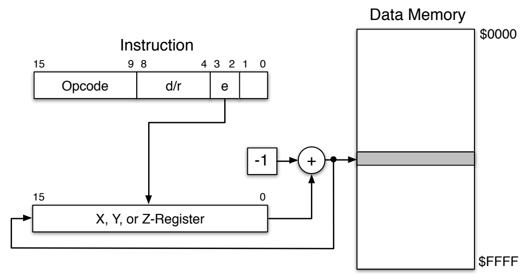
# Indirect with Displacement




Examples:

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

# Indirect with Pre-Decrement

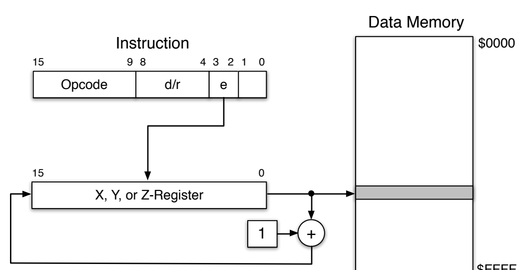



Examples:

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



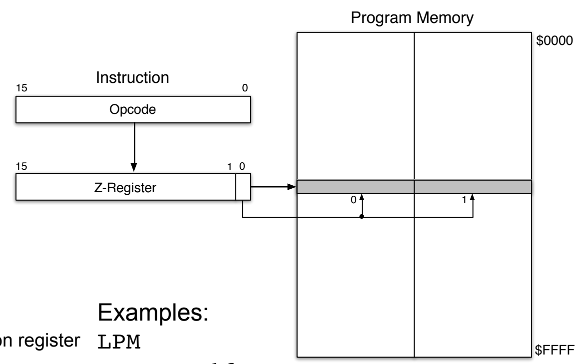
# Indirect with Post-Increment




Examples:

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

# Program Memory Constant Addressing



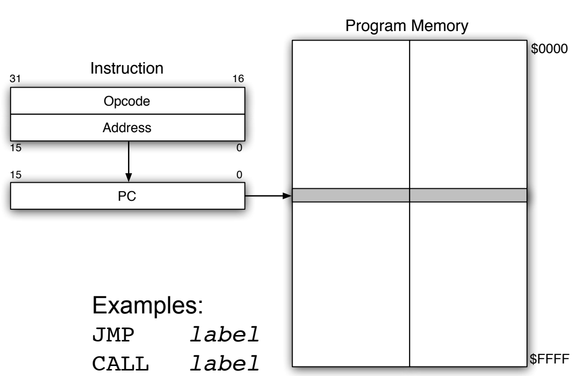

R0 is the destination register

Examples:

```
LPM
LPM    R16, Z
LPM R16, Z+
```



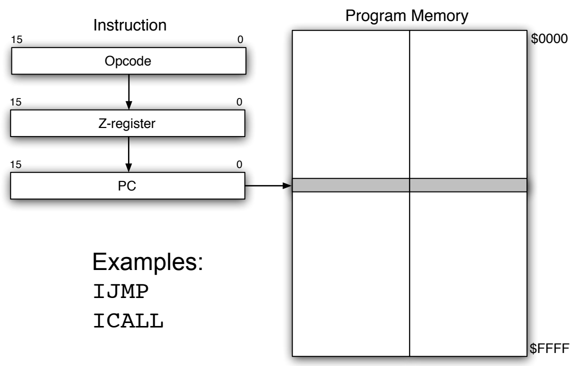
# Direct Program Memory Addressing




Examples:

```
JMP    label
CALL   label
```

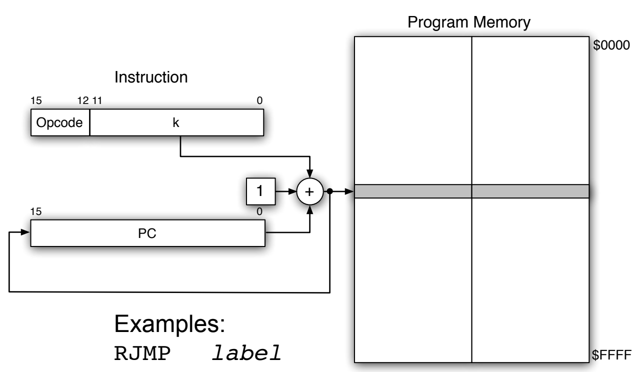
# Indirect Program Memory Addressing




Examples:
`IJMP`
`ICALL`



# Relative Program Memory Addressing




Examples:
`RJMP` *label*
`RCALL` *label*

Called PC-relative jump

## 4.4 Instructions



## AVR Instructions

- AVR has 134 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 (Register Addressing):
  - MOV Rd, Rr
  - d, r = 0, 1, …, 31
- MOVW - transfers data between two pairs of registers.
  - MOVW Rd, Rr
  - d, r = 0, 2, …, 30

- Example:

```
; AVR assembly code – Move Y to Z
 MOVW R30, R28  ; Move R29:R28 (Y) to R31:R30 (Z)
```



## Data Transfer Instructions (2)

- LD - loads data from memory or immediate value (Indirect Addressing):
  - LD Rd, *src* (Load Indirect)
    - d = 0, 1, …, 31
    - *src* = X, X+, -X, Y, Y+, -Y, Z, Z+, -Z
  - LDD Rd, *src* (Load Indirect with Displacement)
    - d = 0, 1, …, 31
    - *src* = Y+q, Z+q
    - q = 6-bit displacement (0 ≤ q ≤ 63)
  - LDI Rd, K (Load Immediate)
    - d = 16, 17, …, 31
    - K = 8-bit constant (0 ≤ K ≤ 255)
    - Can be represented in decimal (e.g., 10), binary (e.g., 0b00001010), or Hexadecimal (e.g., 0x0A or $0A)
  - LDS Rd, k (Load Direct from SRAM)
    - d = 0, 1, …, 31
    - k = 16-bit address

# Data Transfer Instructions (3)

- Example:

```
; AVR assembly code — Add constant to register
  LDI  R16, 24         ; Load 24 into R16
  ADD  R1, R16         ; Add it to R1
```

- Example:

```
; AVR assembly code — Load value from address $0F10
  LDI  R29, $0F        ; Load upper byte of address to R29
  LDI  R28, $10        ; Load lower byte of address to R28
  LD   R16, Y          ; Load value
```



# Data Transfer Instructions (4)

- ST - stores data to memory (Indirect Addressing):
  - ST *dst*, Rr (Store Indirect)
    - *dst* = X, X+, -X, Y, Y+, -Y, Z, Z+, -Z
    - r = 0, 1, …, 31
  - STD *dst*, Rr (Store Indirect with Displacement)
    - *dst* = Y+q, Z+q
    - r = 0, 1, …, 31
    - q = 6-bit displacement ($0 \le q \le 63$)
  - STS k, Rr (Store Direct from SRAM)
    - k = 16-bit address
    - r = 0, 1, …, 31
- Example:

```
; AVR assembly code — Store value to element of structure
; Assume Y points to beginning of a structure
  ADD  R1, R2          ; Add two values
  STD  Y+4, R1         ; Store result to Y offset by 4 bytes
```

## Data Transfer Instructions (5)

- LPM - load program memory
  - LPM Rd, *src*
    - d = 0, 1, …, 31
    - *src* = Z, Z+
  - LPM
    - *dest* = R0 (implied)
    - *src* = Z (implied)
  - Useful for accessing constants stored in Program memory



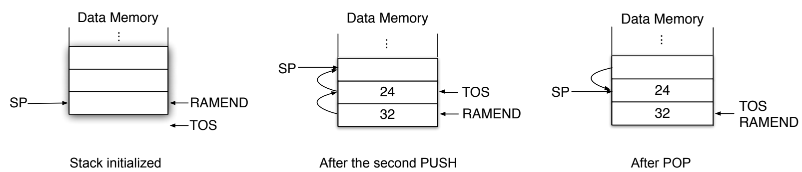
## Data Transfer Instructions (6)

- IN/OUT – I/O instructions
  - IN Rd, A
    - d = 0, 1, …, 31
    - A = 0, 1, …, 63
  - OUT A, Rr
    - A = 0, 1, …, 63
    - r = 0, 1, …, 31
  - Will be discussed in more detail later
- PUSH/POP - stack operations
  - PUSH Rr
    - r = 0, 1, …, 31
  - POP Rd
    - d = 0, 1, …, 31

# Data Transfer Instructions (7)

- Example:

```
; AVR assembly code – Push $32 and $24 onto the stack
    LDI   R22, $32    ; Load $32 into R22
    PUSH  R22         ; Push $32 on to the stack
    LDI   R22, $24    ; Load $24 into R22
    PUSH  R22         ; Push $24 to the stack
    …
    POP   R22         ; Pop TOS (i.e., $24) to R22
```






# ALU Instructions (1)

<u>Arithmetic instructions</u>

- ADD/SUB
  - ADD Rd, Rr
  - d, r = 0, 1, …, 31
  - Also ADC/SBC -> add/subtract with carry
- MUL - Multiply unsigned
  - MUL Rd, Rr
  - d, r = 0, 1, …, 31
  - R1<-Result(high), R0<-Result(low)
  - Also MULS -> multiply signed
- INC/DEC - increment/decrement register
- CLR/SER - clear/set register

# ALU Instructions (2)

Logic instructions

- AND/OR - logical bitwise AND/OR & /w immediate
  - AND/OR Rd, Rr
    - d, r = 0, 1, …, 31
  - ANDI/ORI Rd, K
    - d = 16, 17, …, 31
    - K = 8-bit constant ($0 \le K \le 255$)
    - Can be represented in decimal (e.g., 10), binary (e.g., 0b00001010), or Hexadecimal (e.g., 0x0A or $0A)
  - Also EOR -> Exclusive OR
- COM/NEG – one's/two's complement
  - COM Rd
    - d = 0, 1, …, 31
- TST - test for zero or minus
  - TST Rd
    - d = 0, 1, …, 31



# ALU Instructions (3)

- ADIW/SBIW – Add/subtract Immediate to/from Word
  - ADIW/SBIW Rd+1:Rd, K
    - d = 24, 26, 28, 30
    - K = 8-bit constant ($0 \le K \le 255$)
    - Can be represented in decimal (e.g., 10), binary (e.g., 0b00001010), or Hexadecimal (e.g., 0x0A or $0A)
  - Useful for manipulating 16-bit pointers
  - Example

```
; AVR assembly code - Increment X by 4
      ADIW  R27:R26, 4 ; Add 4 to R27:R26 (X)

; AVR assembly code – Without ADIW
      LDI   R0, 4       ; Load 4
      CLR   R1          ; Clear R1
      ADD   R26, R0     ; Add 4 to lower byte of X
      ADC   R27, R1     ; Add carry to upper byte of x
```

More instructions are required!

## Branch Instructions

- BR*cond* - Branch on condition
  - BR*cond label*
  - *cond* = EQ, NE, GE, LT, CC, CS, PL, MI, VC, VS (and many more)
  - *label* = PC + 1 + k
  - k = 7-bit displacement ($-64 \le k \le 63$)
- CP - compare
  - CP Rd, Rr
  - CPI Rd, immediate
- Compare and conditional branch used together



## Branch Instructions

- Example:

```
; AVR assembly code - Equivalent assembly for IF statement
     CP    R0, R1      ; Compare R0 with R1
     BRGE  NEXT        ; Jump to NEXT if R0 >= R1
     ...               ; If R0 < R1, do something
NEXT:
     ...               ; Otherwise, continue on
```

# Skip Instructions

- Skip if bit in register set(S)/cleared(C)
  - SBRS/C Rd, *bit*
    - d = 16, 17, …, 31
    - *bit* = 0, 1, …7
  - SBIS/C A, *bit*
    - A = 0, 1, …, 63
    - *bit* = 0, 1, …7
- Useful for testing flags
- Example:

```
; AVR assembly code – Test and loop on TOV0
LOOP:
     SBIS $38, 0        ; Skip next inst. if TOV0 is set
     RJMP LOOP          ; Jump back if not set
     ...                ; Do something if flag set
```



# Jump Instruction

- JMP instructions
  - JMP *label*
    - *label* = k
    - k = 16-bit address
  - RJMP *label*
    - *label* = PC + 1 + k
    - k = 7-bit displacement (-64 ≤ k ≤ 63)
  - IJMP
    - *label* = Z
- CALL/RET - subroutine call/return
  - CALL *label*
  - RCALL *label*
  - ICALL
  - RET
  - Stack implied
  - Also RETI -> return from interrupt

# Jump Example

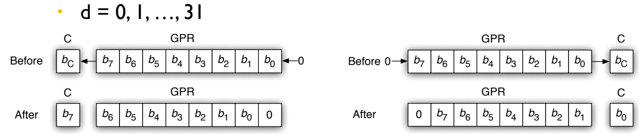
- Example

```
; AVR assembly code - Subroutine call and return
     …
     RCALL SUBR        ; First call to SUBR
     …
     RCALL SUBR        ; Second call to SUBR
     …
SUBR:
     …
     …Do something…    ; Subroutine
     …
     RET
```

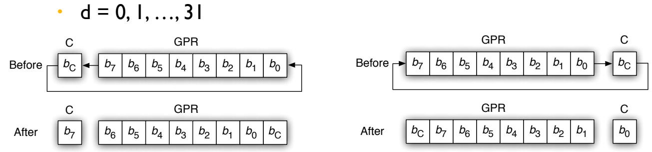


# Bit and Bit Test Instructions

- LSL/LSR - Logical Shift Left/Right
  - LSL Rd
    - d = 0, 1, …, 31




- ROL/ROR - Rotate Left/Right through carry
  - ROL R
    - d = 0, 1, …, 31



## Bit and Bit Test Instructions

- SBI/CBI – Set/Clear Bit in I/O register
  - SBI/CBI A, *bit*
    - A = 0, 1, …, 63
    - *bit* = 0, 1, …, 7
- SE*f*/CL*f* - Set/Clear flag
  - *f* = C, N, Z, I, S, V, T, H
  - SEI => Turn on global interrupt



## 4.5 Assembly to Machine Instruction Mapping

# Introduction

- Discussed assembly instructions using *mnemonics*.
  - Easier to remember names of instructions, registers, and memory locations.
- When computers were first developed, they were programmed using 0's and 1's via mechanical switches, i.e., *machine instructions*!
- Will discuss assembly to machine instruction mapping.
- Knowing how various information of instructions are encoded is crucial for understanding how to design and develop processor hardware.



# ALU Instructions

- INC Rd
  - `INC R1          ; R1 <- R1 + 1`

| | | | |
|---|---|---|---|
| 1001 | 010d | dddd | 0011 |

d dddd = 0 0001

- ADD Rd, Rr
  - `ADD R15, R16  ; R15 <- R15 + R16`

| | | | |
|---|---|---|---|
| 0000 | 11rd | dddd | rrrr |

d dddd = 0 1111
r rrrr = 1 0000

# Load Instructions (1)

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

| 1001 | 000d | dddd | 1000 |
|---|---|---|---|

d dddd = 1 0000

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

| 1110 | KKKK | dddd | KKKK |
|---|---|---|---|

1 dddd = 1 1110
KKKK KKKK = 1111 0000



# Load Instructions (2)

- LDD Rd,Y+q
  - `LDD  R4, Y+2   ; R4 <- M(Y+2)`
  - Y is implied in the opcode
  - 6-bit displacement q ($0 \le q \le 63$)

| 10q0 | qq0d | dddd | 1qqq |
|---|---|---|---|

d dddd = 0 0100
q qq qqq = 0 00 010

# I/O Instructions

- 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 |
|---|---|---|---|

d dddd = 1 1001
AA AAAA = 01 0110



# 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

- BRcc *label*
  - if (cc) then PC <-PC + k +1 (PC-relative)
  - Example:

```
Address                              Code
0232                  CP      R0, R1            ; compare
0233                  BREQ    SKIP
0234                  …                         ; Next inst.
                      …
0259       SKIP:…
```

k = $0259 - $0234 = $0025

BREQ SKIP =>

| | | | |
|---|---|---|---|
| 1111 | 00kk | kkkk | k001 |

kk kkkk k = 01 0010 1

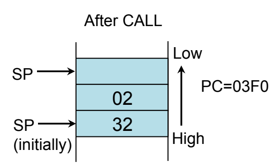
Address range => -64 ≤ k ≤ +63

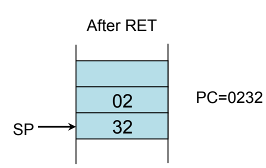


# Jump Instructions

- CALL *label*
  - PC <- k

```
Address                Code
0230                  CALL     SUBR
0232                  next inst.
…                     …
03F0       SUBR:      …
                      …my subroutine…
                      …
                      RET
```




  - CALL is 32-bit instruction

| | | | |
|---|---|---|---|
| 1001 | 0100 | 0000 | 1110 |
| kkkk | kkkk | kkkk | kkkk |

kkkk kkkk kkkk kkkk = 0000 0011 1111 0000

# 4.6 Assembler Directives



# 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`
  - INCLUDE: Tells the assembler to read from the specified file.
    Syntax: .INCLUDE *filename*
    `.INCLUDE "m128def.inc"`

# 4.7 Expressions



## AVR Assembly Expressions (1)

- Expression can consist of operands, operators, and functions:
- Operands can be
  - labels
  - Integer constants
  - Constants defined using .EQU
- Support all C/C++ type operators
- Provides predefined functions

# AVR Assembly Expressions (2)

- Example:

```
; AVR assembly code - Example use of operators
.EQU  RXEN1 = 4
.EQU  TXEN1 = 3
.DEF  mpr = R16
...
      LDI    mpr, (1<<RXEN1|1<<TXEN1)
```

After preprocessing this becomes
`LDI R16, 0b00011000`

- Example:

```
; AVR assembly code – Example use of functions
.DSEG
Array:
      .BYTE array_size
.CSEG
      LDI    R27, HIGH(Array)      ; Load high byte address to R27
      LDI    R26, LOW(Array)       ; Load low byte address to R26
      LD     R7, X
```



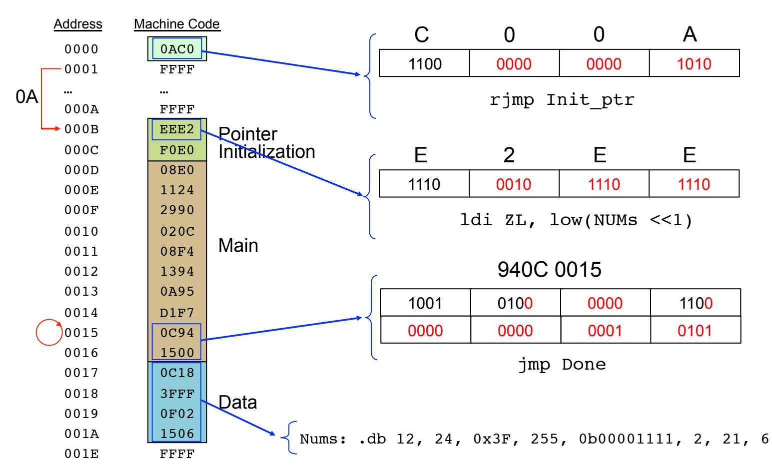
# 4.10 Anatomy of an Assembler Program

# Addition of 8 Numbers

```
.include        "m128def.inc"
.ORG    $0000
        RJMP    Init_ptr
.ORG    $000B
Init_ptr:
        LDI     ZL, low(Nums<<1)        ;
        LDI     ZH, high(Nums<<1)       ; Z points to 12
Main:   LDI     R16, 8                  ; Load loop count
        CLR     R1                      ; Clear accumulator R1:R0
        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
```



# Code in Program Memory



# Main Loop

```
; Assume Z points to the first word
Main: LDI   R16, 8       ; Load loop count
      CLR   R1           ; Clear accumulator R1:R0
      CLR   R0
Loop: LPM   R2,Z+        ; Load data to R2 and post-inc ptr
      ADD   R0 R2        ; Add R2 to R0(L)
      BRCC  Skip         ; No carry, skip next step
      INC   R1           ; Add carry R1(H)
Skip: DEC   R16          ; Decrement loop count
      BRNE  Loop         ; If not done loop
Done: JMP   Done         ; Done.  Loop forever.
```

Can be replaced by

```
      CLR   R3           ; Clear R3
      …
      ADD   R0, R2       ; Add R2 to R0(L)
      ADC   R1, R3       ; Add carry
```



# 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:

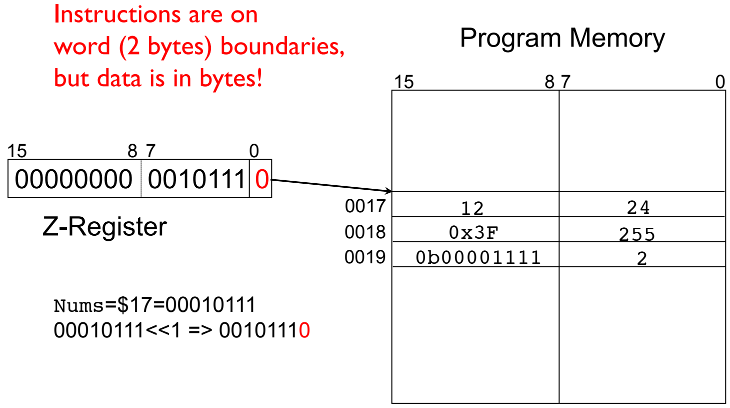
```
.ORG  $000B
      LDI   ZL, low(Nums<<1)    ;
      LDI   ZH, high(Nums<<1)   ;Z points to 12
      …
NUMS: .DB   12, 24, 0x3F, 255, 0b00001111, 2, 21, 6
```

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!




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 & 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 & 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
      rcall ERROR       ; We could also use CALL
Skip: …
      …
ERROR:
      …                 ; My subroutine would go here
      RET
```

# Questions?