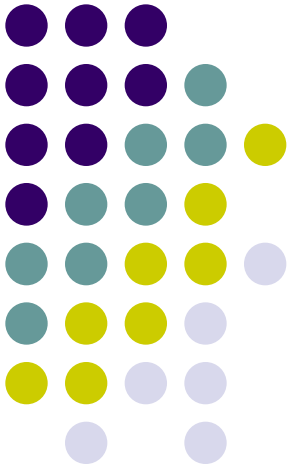
AVR ISA & AVR Programming (I)

Lecturer: Sri Parameswaran

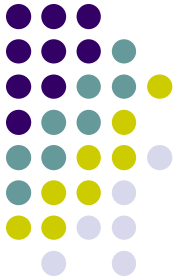
Notes by: Annie Guo



Lecture Overview

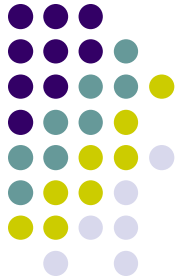
- AVR ISA
- AVR Instructions & Programming (I)
  - Basic construct implementation

Atmel AVR



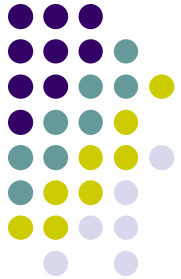
- 8-bit RISC architecture
  - Most instructions have 16-bit fixed length
  - Most instructions take 1 clock cycle to execute.
- Load-store memory access architecture
  - All calculations are on registers
- Two-stage instruction pipelining
- Internal program memory and data memory
- Wide variety of on-chip peripherals (digital I/O, ADC, EEPROM, UART, pulse width modulator (PWM) etc).

AVR Registers



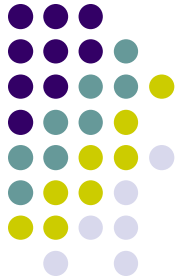
- General purpose registers
  - 32 8-bit registers, r0 ~ r31 or R0 ~ R31
  - Can be further divided into two groups
    - First half group: r0 ~ r15 and second half group: r16 ~ r31
    - Some instructions work only on the second half group r16 ~ r31
      - Due to the limitation of instruction encoding bits
        - Will be covered later
      - E.g. `ldi Rd, #number  ; Rd ∈ r16~r31`

AVR Programming



- Refer to the online AVR Instruction Set documentation for the complete list of AVR instructions
  - http://www.cse.unsw.edu.au/~cs2121/AVR/AVR-Instruction-Set.pdf
- The rest of the lecture covers
  - Programming to implement some basic constructs with examples

Arithmetic Calculation (1/4) -- example

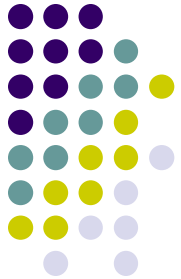


- Expressions

$$z = 2x - xy - x^2$$

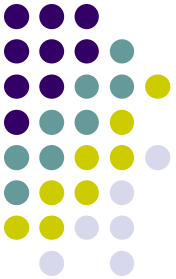
  - where all data including products from multiplications are 8-bit unsigned numbers; and x, y, z are stored in registers r2, r3, and r4.

What instructions do you need?



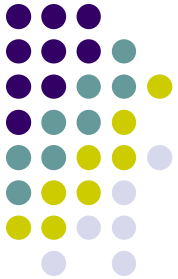
- `sub`
- `mul`
- `ldi`
- `mov`

Subtract without Carry



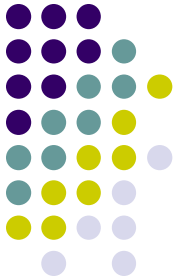
- Syntax: ***sub Rd, Rr***
- Operands: Rd, Rr $\in$ {r0, r1, …, r31}
- Operation: Rd$\leftarrow$Rd–Rr
- Flags affected: H, S, V, N, Z, C
- Words: 1
- Cycles: 1

Multiply Unsigned



- Syntax: ***mul Rd, Rr***
- Operands: Rd, Rr $\in$ {r0, r1, …, r31}
- Operation: **r1:r0**$\leftarrow$Rr*Rd
  - (unsigned$\leftarrow$unsigned * unsigned)
- Flags affected: Z, C
  - C is set if bit 15 of the result is set; cleared otherwise.
- Words: 1
- Cycles: 2

Load Immediate



- Syntax: ***ldi Rd, k***
- Operands: Rd∈{**r16, …, r31**}, 0 ≤ k ≤ 255
- Operation: Rd←k
- Flag affected: None
- Words: 1
- Cycles: 1
- Encoding: 1110 kkkk dddd kkkk
- Example:

```
ldi  r16, 0x42          ; Load 0x42 to r16
```

Copy Register

- Syntax: ***mov Rd, Rr***
- Operands: Rd, Rr $\in${r0,r1,…,r31}
- Operation: Rd$\leftarrow$Rr
- Flag affected: None
- Words: 1
- Cycles: 1

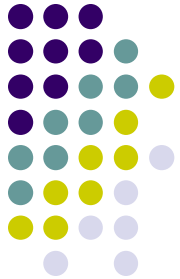
Arithmetic calculation (2/4)

- AVR code for $z = 2x - xy - x^2$
  - where all data including products from multiplications are 8-bit unsigned numbers; and x, y, z are stored in registers r2, r3, and r4.

```
ldi   r16, 2          ; r16 ← 2
mul   r16, r2         ; r1:r0 ← 2x
mov   r5, r0          ; r5 ← 2x
mul   r2, r3          ; r1:r0 ← xy
sub   r5, r0          ; r5 ← 2x-xy
mul   r2, r2          ; r1:r0 ← x²
sub   r5, r0          ; r5 ← 2x-xy- x²
mov   r4, r5          ; r4 ← z
```

  - 8 instructions and 11 cycles

AVR Registers (cont.)

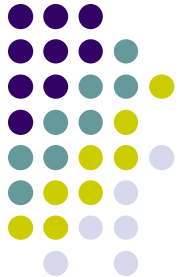


- General purpose registers
  - The following register pairs can work as address indexes
    - X, r26:r27
    - Y, r28:r29
    - Z, r30:r31
  - The following registers can be applied for specific use
    - r1:r0 store the result of multiplication instruction
    - r0 stores the data loaded from the program memory

AVR Registers (cont.)

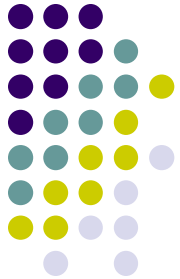
- I/O registers
  - 64+416 8-bit registers
    - Their names are defined in the m2560def.inc file
  - Used in input/output instructions
    - Mainly storing data/addresses and control signal bits
  - Some instructions work only with I/O registers, others with general purpose registers – don't confuse them
    - E.g. `in Rd, port` `; Port must be an I/O register (0-63)`
    - E.g. `lds Rd, port` `; Used for I/O registers (64-415)`
  - Will be covered in detail later
- Status register (SREG)
  - A special I/O register

The Status Register in AVR



- The Status Register (SREG) contains information about the result of the most recently executed arithmetic instruction. This information can be used for altering program flow in order to perform conditional operations.
- SREG is updated after any of ALU operations by hardware.
- SREG is not automatically stored when entering an interrupt routine and restored when returning from an interrupt. This must be handled by software.
  - Use in/out instructions to store/restore SREG

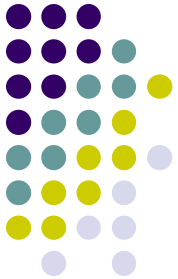
The Status Register in AVR (cont.)

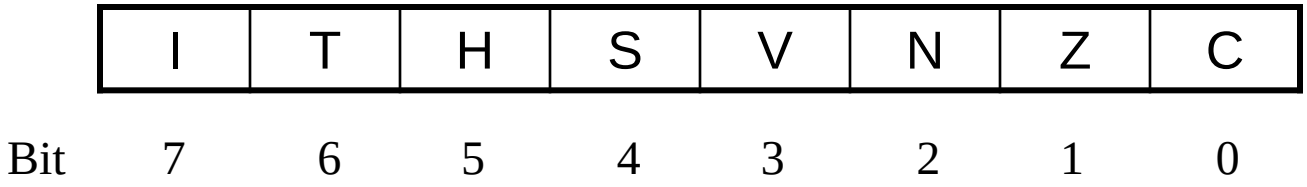


	I	T	H	S	V	N	Z	C
Bit	7	6	5	4	3	2	1	0

- Bit 7 – I: Global Interrupt Enable
  - Used to enable and disable interrupts.
  - 1: enabled. 0: disabled.
  - The I-bit is cleared by hardware after an interrupt has occurred, and is set by the RETI instruction to enable subsequent interrupts.

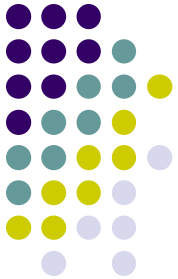
The Status Register in AVR (cont.)





- Bit 6 – T: Bit Copy Storage
  - The Bit Copy instructions `bld` (Bit LoaD) and `bst` (Bit STore) use the T-bit as source or destination for the operated bit. A bit from a register in the Register File can be copied into T by the `bst` instruction, and a bit in T can be copied into a bit in a register in the Register File by the `bld` instruction.

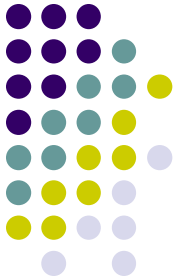
The Status Register in AVR (cont.)



	I	T	H	S	V	N	Z	C
Bit	7	6	5	4	3	2	1	0

- Bit 5 – H: Half Carry Flag
  - The Half Carry Flag H indicates a Half Carry (carry from bit 4) in some arithmetic operations.
  - Half Carry is useful in BCD arithmetic.

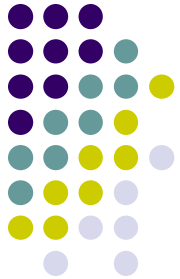
The Status Register in AVR (cont.)



I	T	H	S	V	N	Z	C
Bit	7	6	5	4	3	2	1	0

- Bit 4 – S: Sign Bit
  - Exclusive OR between the Negative Flag N and the Two's Complement Overflow Flag V (S = N $\oplus$ V).
- Bit 3 – V: Two's Complement Overflow Flag
  - The Two's Complement Overflow Flag V supports two's complement arithmetic.

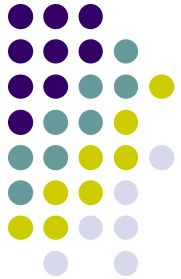
The Status Register in AVR (cont.)



I	T	H	S	V	N	Z	C
Bit	7	6	5	4	3	2	1	0

- Bit 2 – N: Negative Flag
  - N is the most significant bit of the result.
- Bit 1 – Z: Zero Flag
  - Z indicates a zero result in an arithmetic or logic operation. 1: zero. 0: Non-zero.

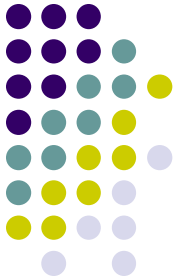
The Status Register in AVR (cont.)



I	T	H	S	V	N	Z	C
Bit	7	6	5	4	3	2	1	0

- Bit 0 – C: Carry Flag
  - Its meaning depends on the operation.
    - For addition x+y, it is the carry from the most significant bit
    - For subtraction x-y, where x and y are unsigned integers, it indicates if x<y. If x<y, C=1; otherwise, C=0.

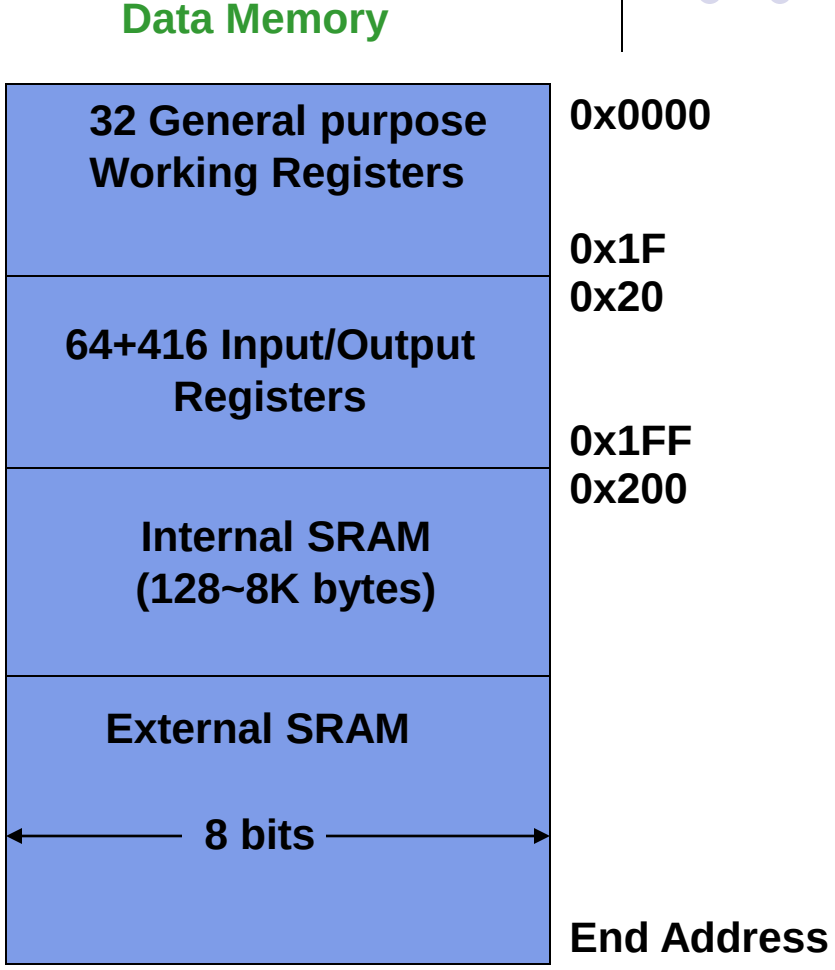
AVR Address Spaces



- Three address spaces
  - Data memory
    - Storing data to be processed
  - Program memory
    - Storing program and sometimes constants
  - EEPROM memory
    - Large permanent data storage

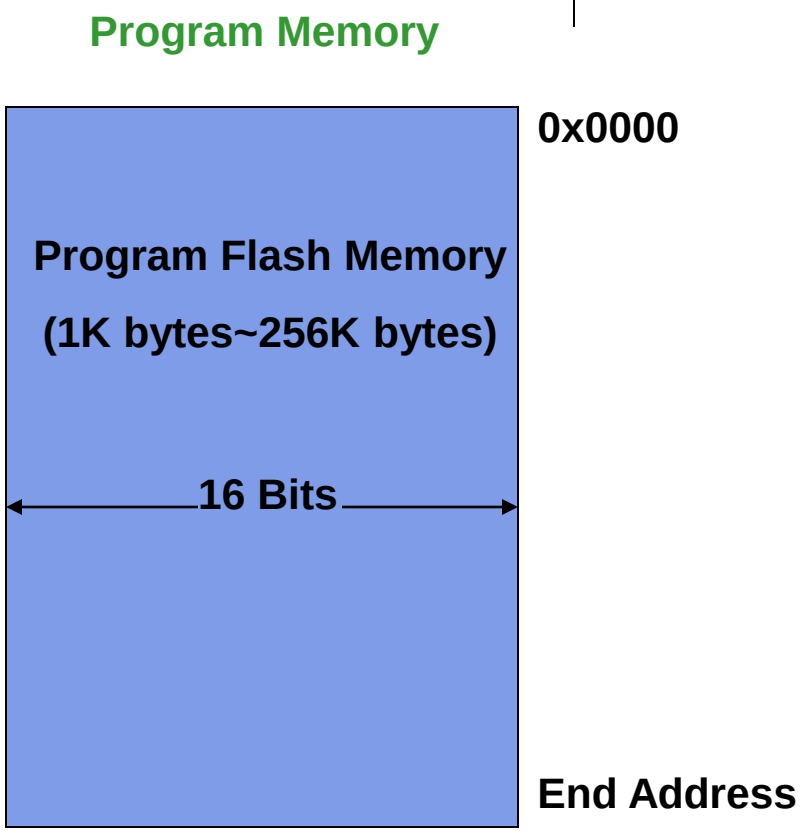
Data Memory Space

- Covers
  - Register file
    - I.e. registers in the register file also have memory address
  - I/O registers
    - I.e. First 64 I/O registers have two addresses
      - I/O addresses
      - Memory addresses
  - SRAM data memory
    - The highest memory location is defined as RAMEND



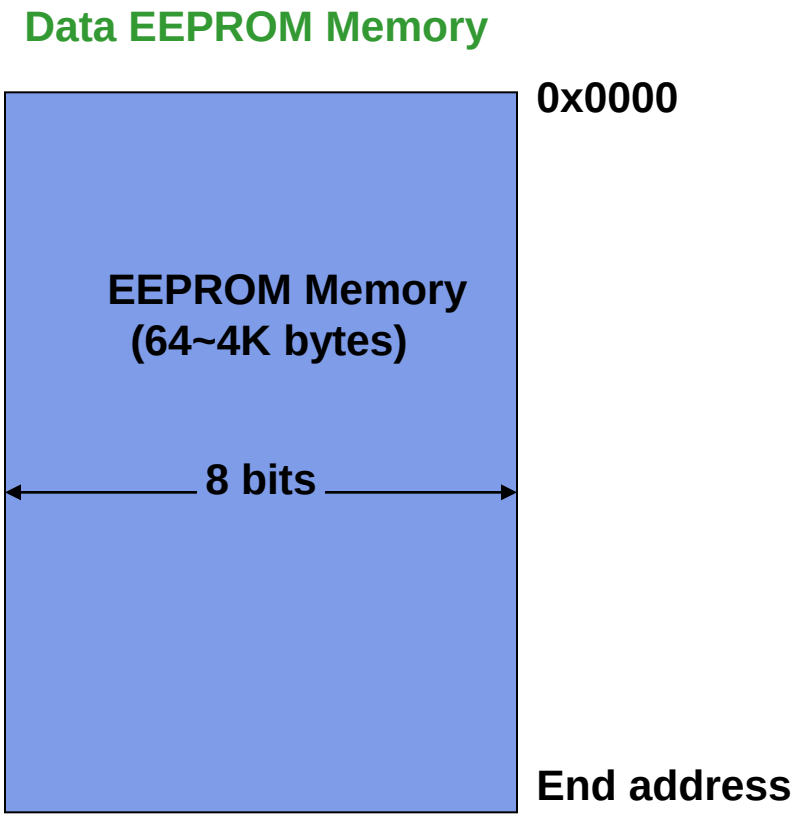
Program Memory Space

- Covers
  - 16 bit Flash Memory
    - Read only
      - Instructions are retained when power off
  - Can be accessed with special instructions
    - `lpm`
    - `spm`

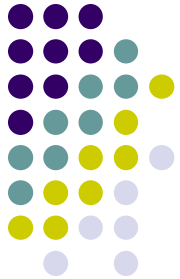


EEPROM Memory Space

- Covers
  - 8-bit EEPROM memory
    - Use to permanently store large set of data
  - Can be accessed using load and store instructions with special control bit settings
- Not covered in this course

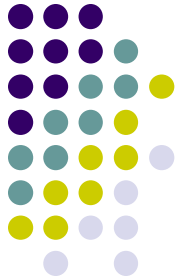


AVR Instruction Format



- For AVR, almost all instructions are 16 bits long
  - `add Rd, Rr`
  - `sub Rd, Rr`
  - `mul Rd, Rr`
  - `brge k`
- Few instructions are 32 bits long
  - `lds Rd, k` ($0 \leq k \leq 65535$)
    - loads 1 byte from the SRAM to a register.

Examples (1) -- 16 bits long



- Clear register instruction

  Syntax: *add Rd, Rr*

  Operand: $0 \leq Rd \leq 31$, $0 \leq Rr \leq 31$

  Operation: $Rd \leftarrow Rd + Rr$

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

15 0

- Instruction format
  - OpCode uses 6 bits (bit 9 to bit 15).
  - The only operand uses the remaining 10 bits (only 5 bits (bit 0 to bit 4) are actually needed).
- Execution time

  1 clock cycle

Examples (2)
-- 32 bit long

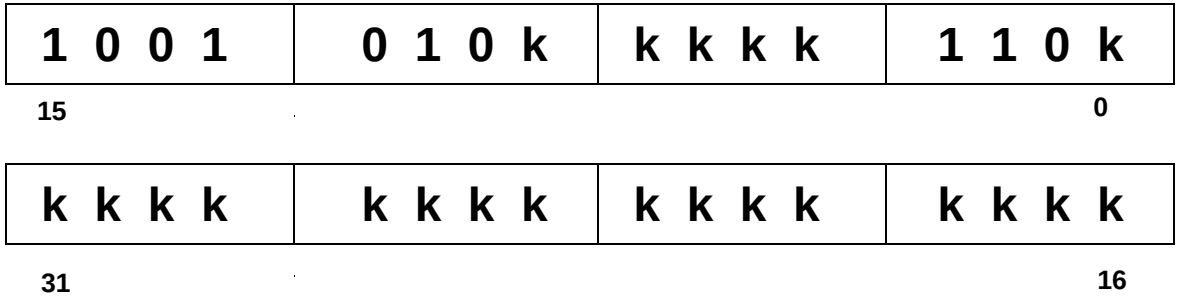
- Unconditional branch

  Syntax: *jmp k*

  Operand: 0 ≤ k < 4M

  Operation: PC ← K

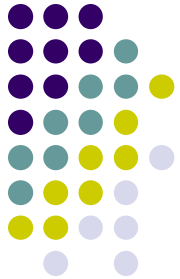
- Instruction format



- Execution time

  3 clock cycles

Examples (3)
-- with variable cycles



- Conditional branch

  Syntax: *breq k*

  Operand: -64 ≤ k < +63

  Operation: If Rd=Rr(Z=1) then PC ← PC+k+1, else PC ←PC+1

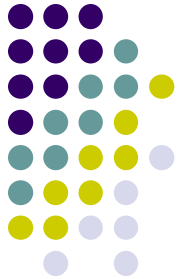
1 1 1 1	0 0 k k	k k k k	k 0 0 1

- Instruction format
- Execution time

  1 clock cycle if condition is false

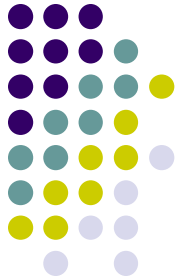
  2 clock cycles if condition is true

AVR Instructions



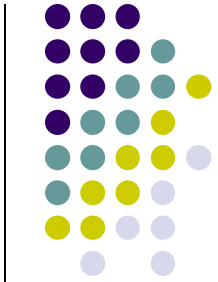
- AVR has the following classes of instructions:
  - Arithmetic and Logic
  - Data transfer
  - Program control
  - Bit and Others
    - Bit set and Bit test
    - MCU Control
- An overview of the instructions are given in the next slides.

Arithmetic and Logic Instructions



- Arithmetic
  - addition
    - E.g. `add Rd, Rr`
  - Subtraction
    - E.g. `sub Rd, Rr`
  - Increment/decrement
    - E.g. `inc Rd`
  - Multiplication
    - E.g. `mul Rd, Rr`
- Logic
  - E.g. `and Rd, Rr`
- Shift
  - E.g. `lsl Rd`

Transfer Instructions



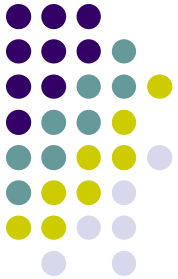
- GP register
  - E.g. `mov Rd, Rr`
- I/O registers
  - E.g. `in Rd, PORTA`
    `out PORTB, Rr`
- Stack
  - `push Rr`
  - `pop Rd`
- Immediate values
  - E.g. `ldi Rd, K`
- Memory
  - Data memory
    - E.g. `ld Rd, X`
    - E.g. `st X, Rr`
  - Program memory
    - E.g. `lpm`
  - EEPROM memory
    - Not covered in this course

Program Control Instructions

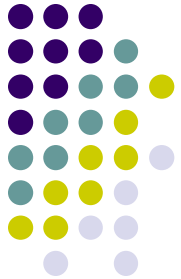
- Branch
  - Conditional
    - Jump to address
      - `breq dest`
        - test ALU flag and jump to specified address if the test was true
    - Skips
      - `sbic k`
        - test a bit in a register or an IO register and skip the next instruction if the test was true.
  - Unconditional
    - Jump to the specified address
      - `rjmp dest`
- Call subroutine
  - E.g. `rcall k`
- Return from subroutine
  - E.g. `ret`

Bit & Other Instructions

- Bit
  - Set bit
    - E.g. `sbi PORTA, b`
  - Clear bit
    - E.g. `cbi PORTA, b`
  - Bit copy
    - E.g. `bst Rd, b`
- Others
  - `nop`
  - `break`
  - `sleep`
  - `wdr`

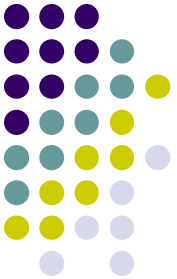


AVR Instructions (cont.)



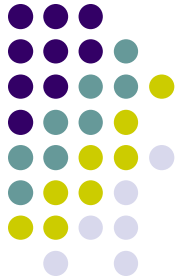
- Not all instructions are implemented in all AVR controllers.
- Refer to the data sheet of a specific microcontroller
- Refer to online AVR instruction document for the detail description of each instruction

AVR Addressing Modes



- Immediate
- Register direct
- Memory related addressing mode
  - Data memory
    - Direct
    - Indirect
    - Indirect with Displacement
    - Indirect with Pre-decrement
    - Indirect with Post-increment
  - Program memory
  - EEPROM memory
    - Not covered in this course

Immediate Addressing

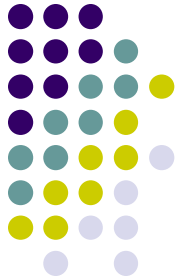


- The operands come from the instructions
- For example

```
andi r16, 0x0F
```

  - Bitwise logic AND operation
    - Clear upper nibble of register r16

Register Direct Addressing



- The operands come from general purpose registers
- For example

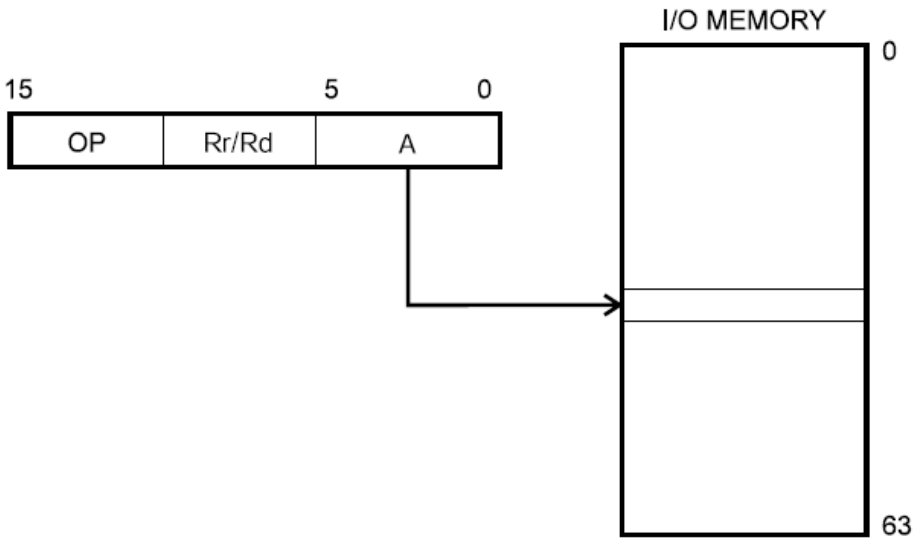
```
add r16, r0
```

  - r16 ← r16 + r0
    - Add register r0 to register r16

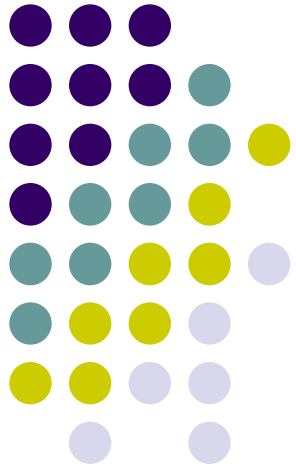
Register Direct Addressing

- The operands come from I/O registers
- For example

```
in r25, PINA
     ; r25 ← PIN A
```



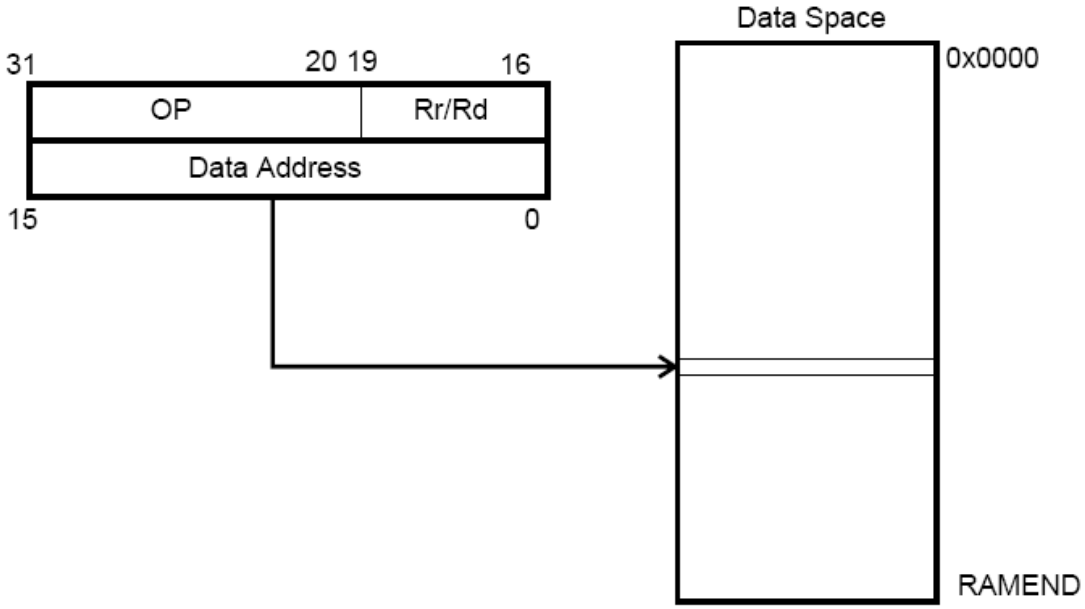
Data Memory Addressing



Data Direct Addressing

- The data memory address is given directly from the instruction
- For example

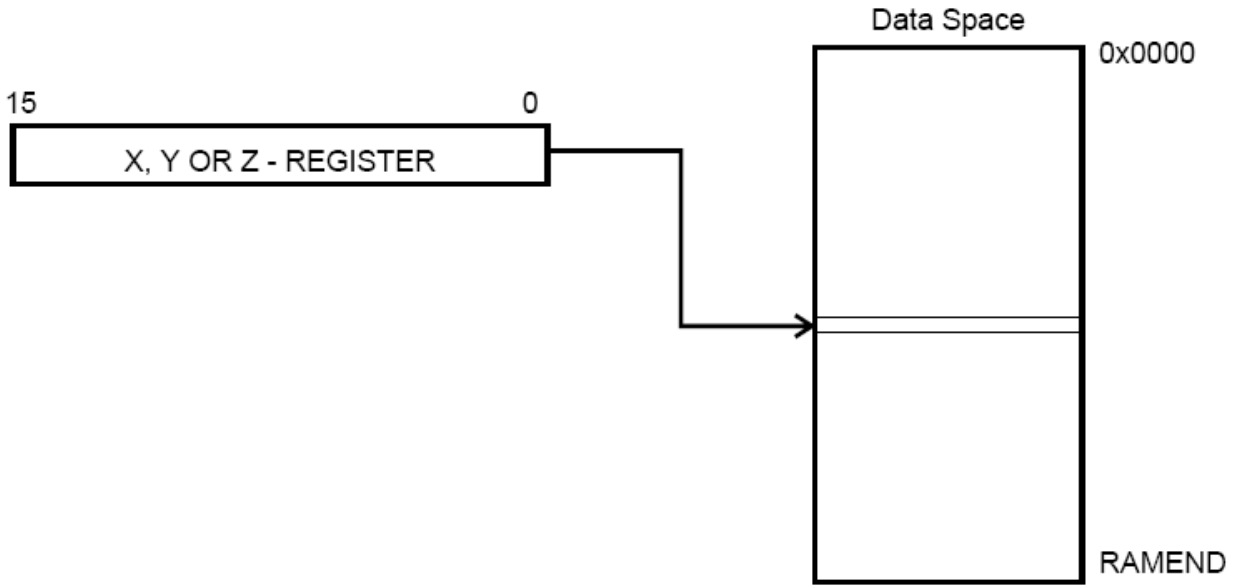
```
lds r5, 0xF123
     ; r5 ← Mem(0xF123), or r5 ←(0xF123)
```



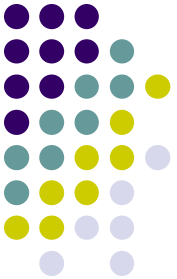
Indirect Addressing

- The data memory address is from an address pointer (X, Y, Z)
- For example

```
ld r11, X
     ; r11 ← Mem(X), or r11 ← (X)
```

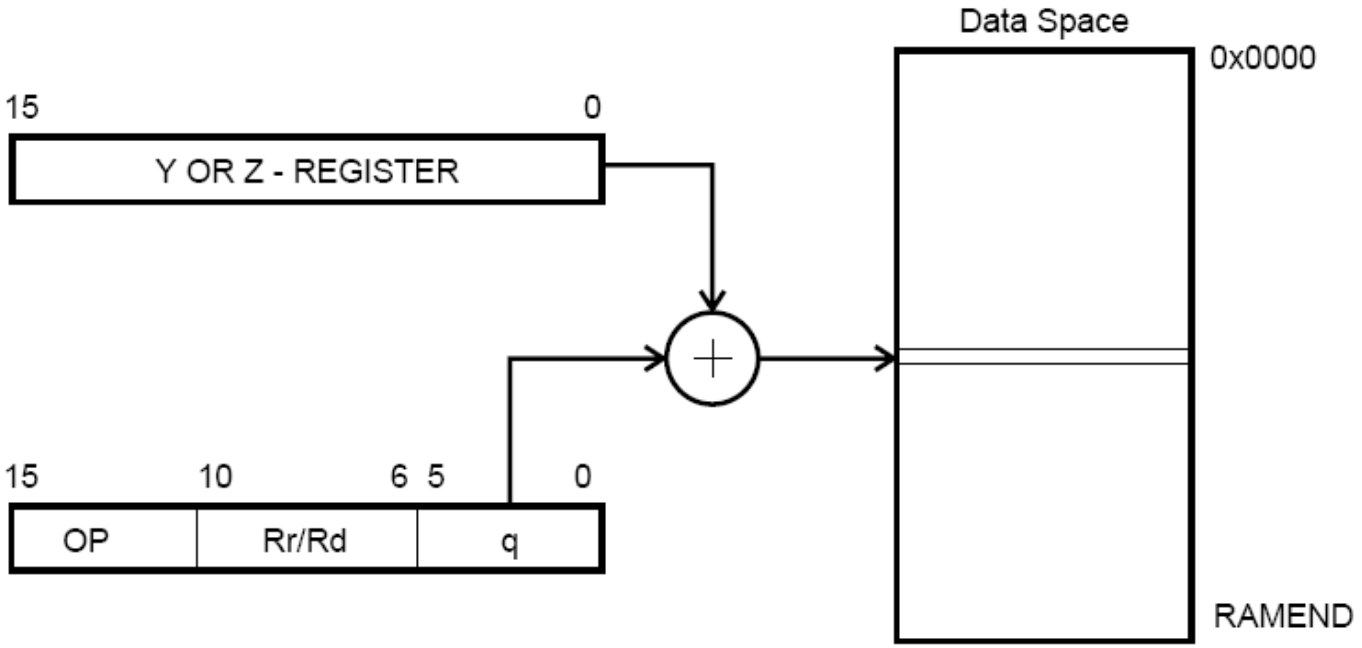


Indirect Addressing with displacement



- The data memory address is from (Y,Z)+q
- For example

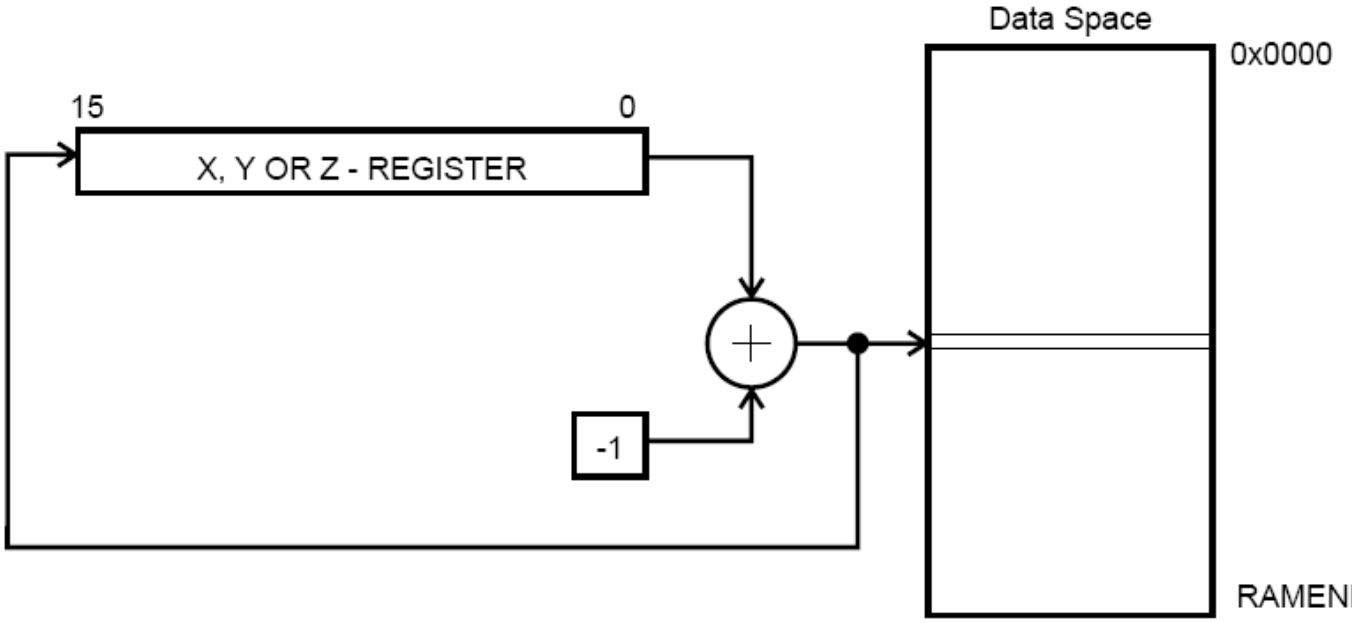
```
std Y+10, r14
      ; (Y+10) ← r14
```



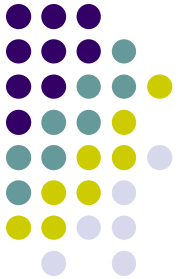
Indirect Addressing with Pre-decrement

- The data memory address is from an address pointer (X, Y, Z) and the value of the pointer is auto-decreased **before** each memory access.
- For example

```
std -Y, r14
     ; Y←Y-1, (Y) ← r14
```

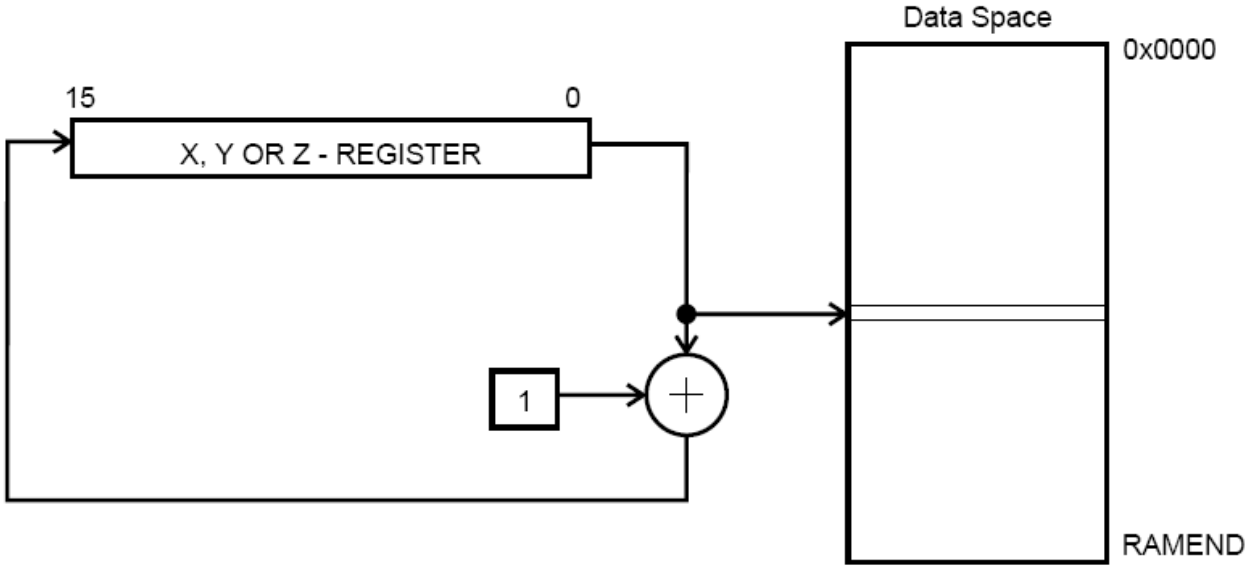


Indirect Addressing with Post-increment

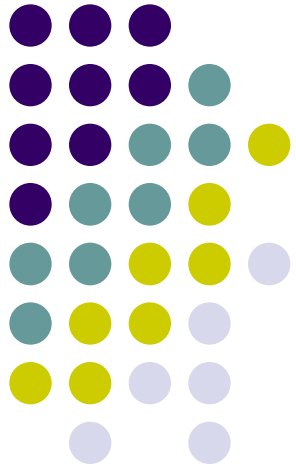


- The address of memory data is from an address pointer (X, Y, Z) and the value of the pointer is auto-increased **after** each memory access.
- For example

```
std Y+, r14
      ; (Y) ← r14, Y ← Y+1
```



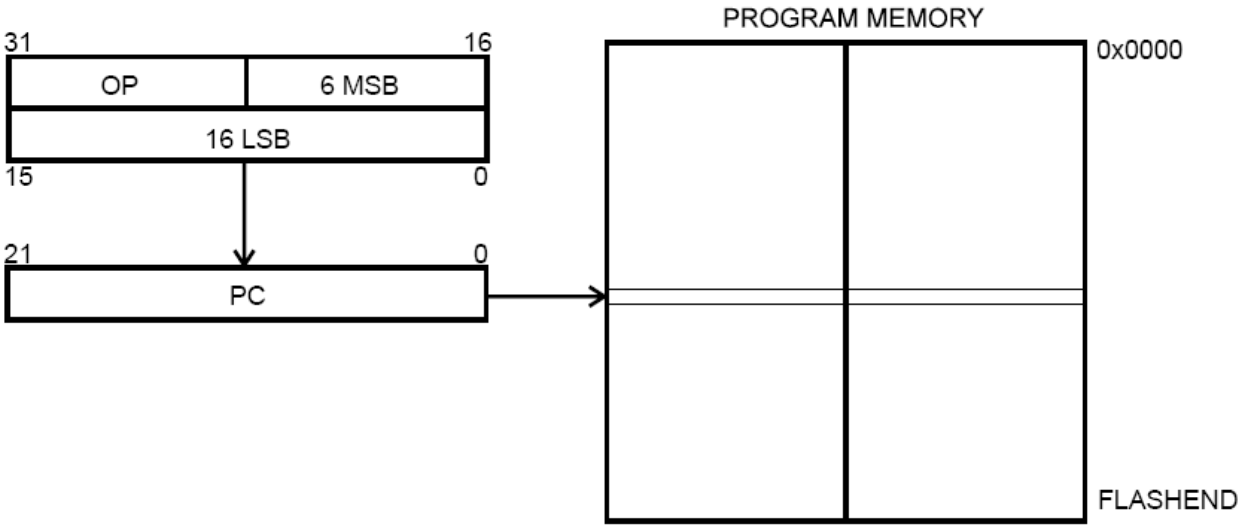
Program Memory Addressing



Direct Program Addressing

- The instruction address is from instruction
- For example

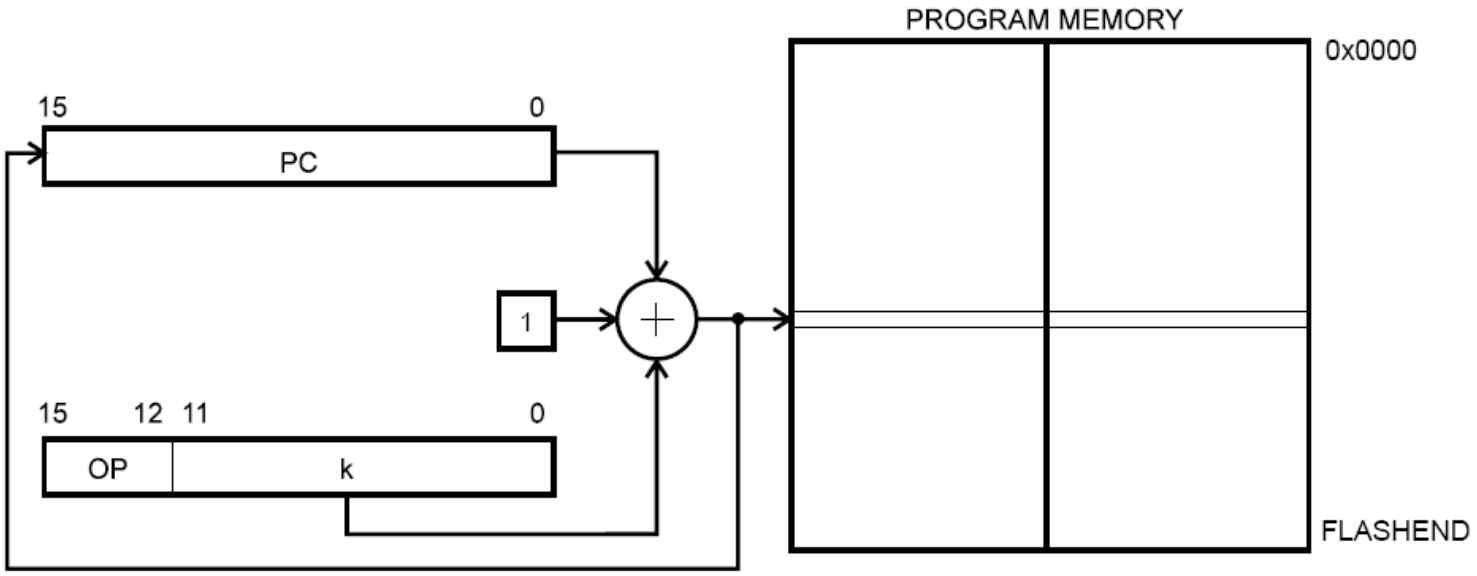
```
jmp k
      ; (PC) ← k
```



Relative Program Addressing

- The instruction address is PC+k+1
- For example

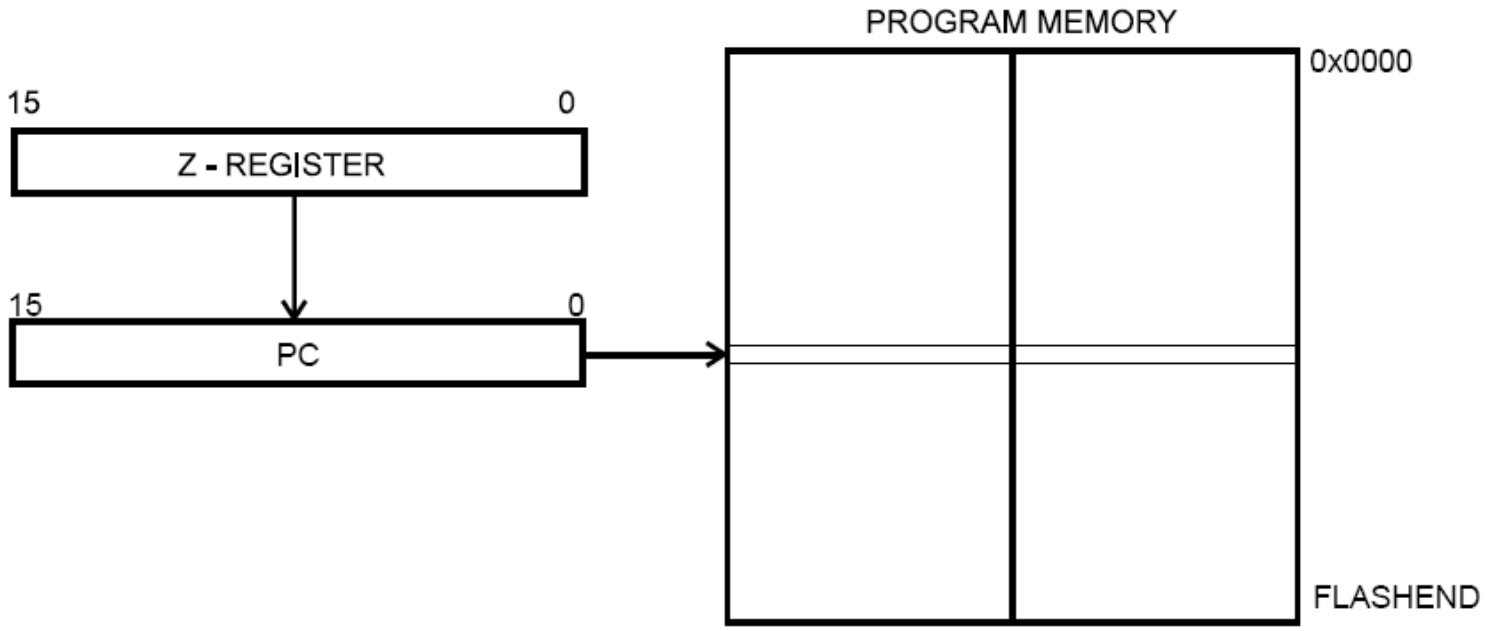
```
rjmp k
      ; (PC) ← (PC)+k+1
```



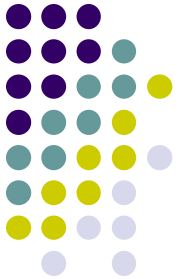
Indirect Memory Addressing

- The instruction address is implicitly stored in Z register

```
icall
     ; PC(15:0) ← (Z), PC(21:16) ←0
```

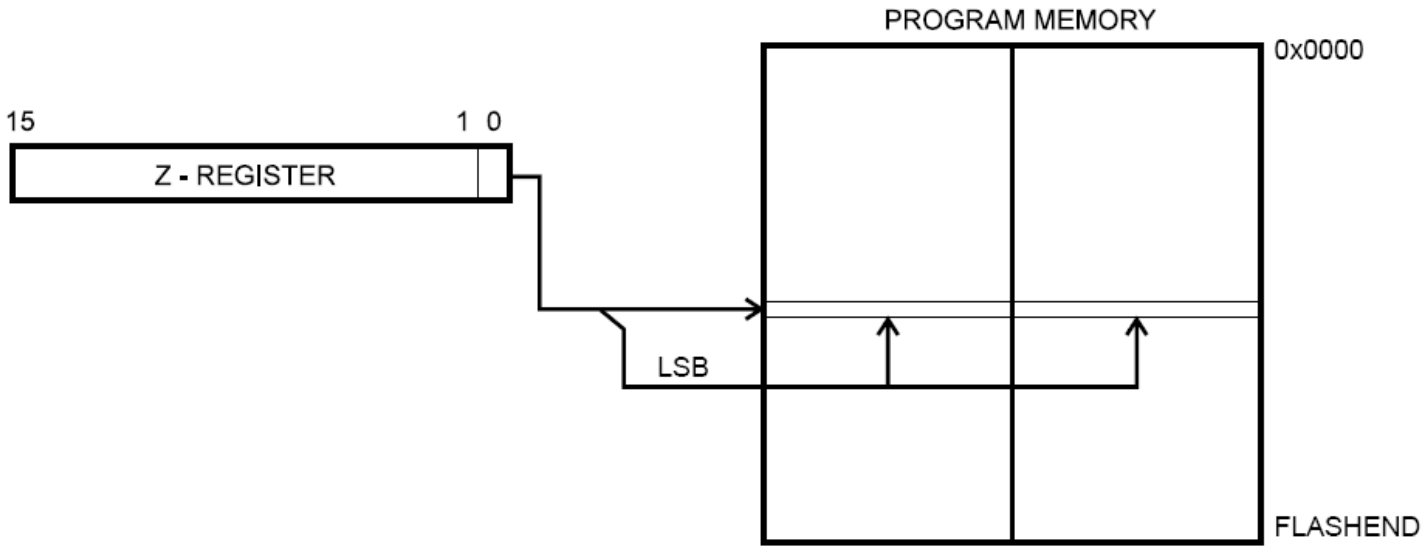


Program Memory Constant Addressing

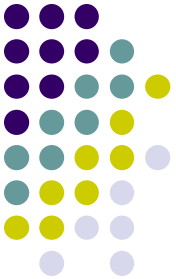


- The address of the constant is stored in Z register
  - The address is a byte address.
- For example:

```
lpm
      ; r0 ← (Z)
```

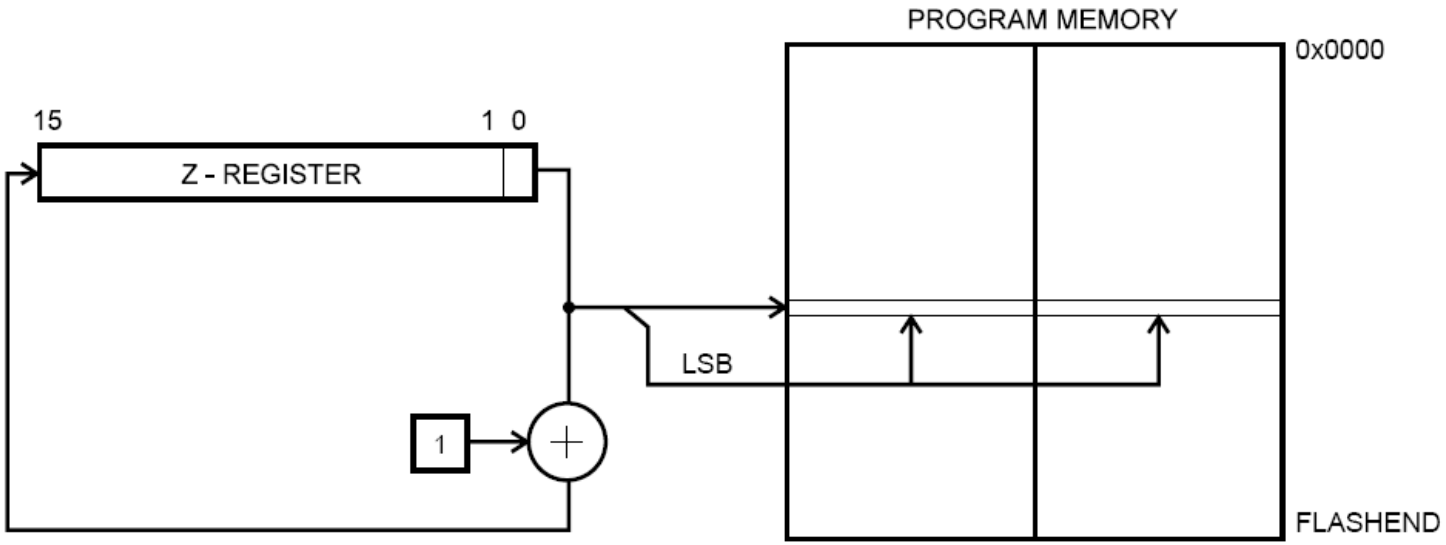


Program Memory Addressing with Post-increment

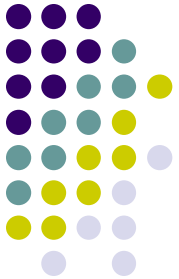


- For example

```
lpm r16, Z+
     ; r16 ← (Z), Z←Z+1
```



Arithmetic calculation (3/4)



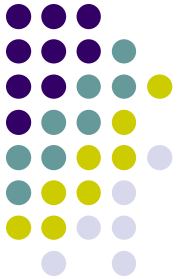
- Expressions

$$z = 2x - xy - x^2$$

$$z = x(2 - (x + y))$$

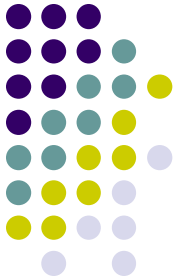
  - where all data including products from multiplications are 8-bit unsigned numbers; and x, y, z are stored in registers r2, r3, and r4.

What instructions do you need?



- `sub`
- `mul`
- `ldi`
- `mov`
- `add`

Add without Carry



- Syntax: ***add Rd, Rr***
- Operands: Rd, Rr $\in$ {r0, r1, …, r31}
- Operation: Rd$\leftarrow$Rd + Rr
- Flags affected: H, S, V, N, Z, C
- Words: 1
- Cycles: 1

Arithmetic calculation (4/4)

- AVR code for $z = 2x - xy - x^2$

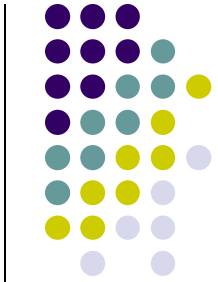
$$z = x(2 - (x + y))$$

  - where all data including products from multiplications are 8-bit unsigned numbers; and x, y, z are stored in registers r2, r3, and r4.

```
mov   r5, r2          ; r5 ← x
add   r5, r3          ; r5 ← x+y
ldi   r16, 2          ; r16 ← 2
sub   r16, r5         ; r16 < 2-(x+y)
mul   r2, r16         ; r1:r0 ← x(2-(x+y))
mov   r4, r0          ; r4 ← z
```

  - 6 instructions and 7 cycles

Control Structure (1/2) -- example



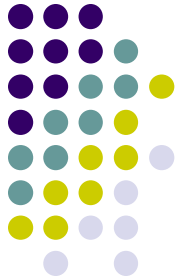
- IF-THEN-ELSE control structure

```
if (x < 0)
        z = 1;
else
        z = -1;
```

  - Numbers x, z are 8-bit signed integers and stored in registers. You need to decide which registers to use.

- Instructions interested
  - Compare
  - Conditional branch
  - Unconditional jump

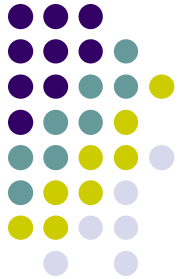
Compare



- Syntax: ***cp Rd, Rr***
- Operands: Rd ∈{r0, r1, …, r31}
- Operation: Rd–Rr (Rd is not changed)
- Flags affected: H, S, V, N, Z, C
- Words: 1
- Cycles: 1
- Example:

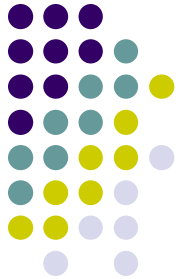
```
   cp r4, r5       ; Compare r4 with r5
   brne noteq      ; Branch if r4 ≠ r5
   ...
noteq: nop         ; Branch destination (do nothing)
```

Compare with Immediate



- Syntax: ***cpi Rd, k***
- Operands: Rd $\in$ {r16, r17, …, r31} and $0 \leq k \leq 255$
- Operation: Rd – k (Rd is not changed)
- Flags affected: H, S, V, N, Z, C
- Words: 1
- Cycles: 1

Conditional Branch



- Syntax: ***brge k***
- Operands: $-64 \le k < 64$
- Operation: If $Rd \ge Rr$ ($N \oplus V=0$) then $PC \leftarrow PC+k+1$,
  else $PC \leftarrow PC+1$ if condition is false
- Flag affected: None
- Words: 1
- Cycles: 1 if condition is false; 2 if condition is true

Relative Jump

- Syntax: ***rjmp k***
- Operands: $-2K \leq k < 2K$
- Operation: PC $\leftarrow$ PC+k+1
- Flag affected: None
- Words: 1
- Cycles: 2

Control (2/2)

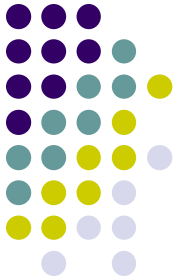
- IF-THEN-ELSE control structure

```
if (a < 0)
        b = 1;
else
        b = -1;
```

  - Numbers x, z are 8-bit signed integers and stored in registers. You need to decide which registers to use.

```
.def   a = r16
.def   b = r17
       cpi    a, 0          ; a - 0
       brge   ELSE          ; if a ≥ 0
       ldi    b, 1          ; b = 1
       rjmp   END           ; end of IF statement
ELSE:  ldi    b, -1         ; b = -1
END:   …
```

Loop (1/2)

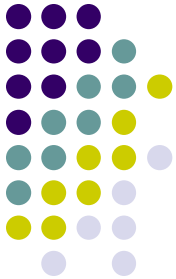


- WHILE loop

```
i = 1;
while (i <= n) {
        sum += i * i;
        i++;
}
```

- Numbers i, sum are 8-bit unsigned integers and stored in registers. You need to decide which registers to use.

Loop (2/2)

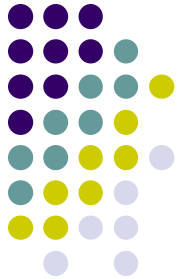


- WHILE loop

```
.def i = r16
.def n = r17
.def sum = r18

        ldi i, 1                        ;initialize
        clr sum
loop:
        cp n, i
        brlo end
        mul i, i
        add sum, r0
        inc i
        rjmp loop
end:
        rjmp end
```

Homework



1. Refer to the AVR Instruction Set documentation (available at http://www.cse.unsw.edu.au/~cs2121/AVR/AVR-Instruction-Set.pdf).

   Study the following instructions:

   - Arithmetic and logic instructions
     - `add, adc, adiw, sub, subi, sbc, sbci, subiw, mul, muls, mulsu`
     - `and, andi, or, ori, eor`
     - `com, neg`

Homework

1. Study the following instructions (cont.)
   - Branch instructions
     - `cp, cpc, cpi`
     - `rjmp`
     - `breq, brne`
     - `brge, brlt`
     - `brsh, brlo`
   - Data transfer instructions
     - `mov`
     - `ldi, ld, st`

Homework

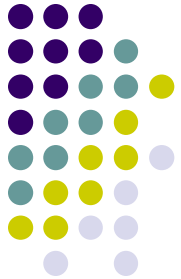
2. Implement the following functions with AVR assembly language
   1) 2-byte addition
   2) 2-byte multiplication
3. Reverse a string of ten characters that is stored in the registers r0~r9; and store the reversed string in registers r10~r19

Homework

4. Translate the following if-then-else statement, where x is an 8-bit unsigned integer.

```
if (x < 0)
        z = 1;
else
        z= 255;
```

Reading Material



- AVR Instruction Set online documentation
  - Instruction Set Nomenclature
  - I/O Registers
  - Program and Data Memory Addressing
  - Arithmetic instructions, program control instructions