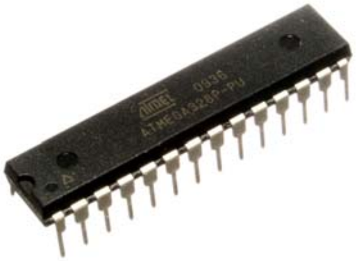
# Embedded Systems

## Lecture 2-4: AVR Architecture



# AVR Architecture

The AVR architecture was conceived by two students at the Norwegian Institute of Technology (NTH).

**Q: What does AVR stand for?**

https://www.youtube.com/watch?v=VUyEFr0YHJs

**AVR == "Alf (Egil Bogen) and Vegard (Wollan) 's Risc processor“**

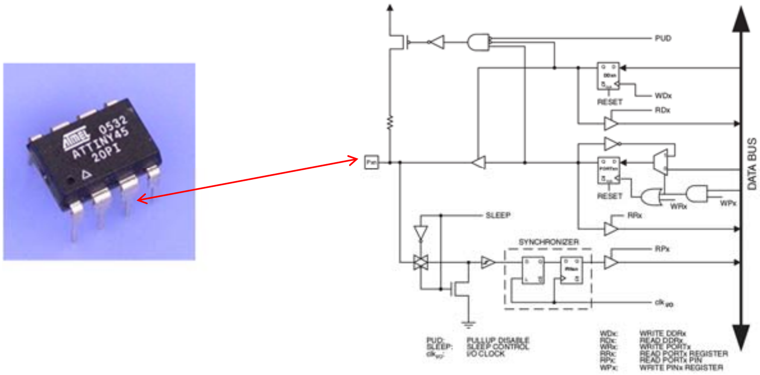
## AVR, ATmega88PA, ATtiny45, etc.

- In the first part of the course, the lab activities will use an 8-pin ATtiny45PU controller
- More complex lab activities will use the 28-pin ATmega88PA controller
- In the lectures we will discuss the core AVR architecture, but draw examples from both ATtiny45PU, and ATmega88PA.
- Atmel → Microchip

## From the ATtiny45 Datasheet …

Atmel ATtiny25/45/85 [DATASHEET] 53
2586Q–AVR–08/2013

**Figure 10-2.** General Digital I/O[1]

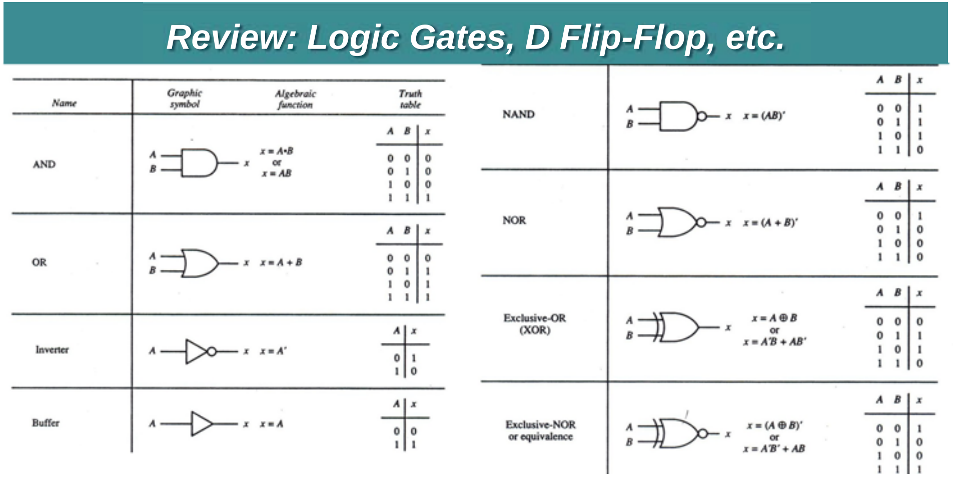


Note: 1. WRx, WPx, WDx, RRx, RPx, and RDx are common to all pins within the same port. $clk_{I/O}$, SLEEP, and PUD are common to all ports.

### 10.2.1 Configuring the Pin

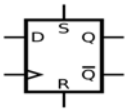
Each port pin consists of three register bits: DDxn, PORTxn, and PINxn. As shown in "Register Description" on page 64, the DDxn bits are accessed at the DDRx I/O address, the PORTxn bits at the PORTx I/O address, and the PINxn bits at the PINx I/O address.

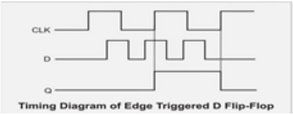
## Review: Logic Gates, D Flip-Flop, etc.



| Name | Graphic symbol | Algebraic function | Truth table |
|---|---|---|---|
| AND | | x = A•B<br>or<br>x = AB | A B \| x<br>0 0 \| 0<br>0 1 \| 0<br>1 0 \| 0<br>1 1 \| 1 |
| OR | | x = A + B | A B \| x<br>0 0 \| 0<br>0 1 \| 1<br>1 0 \| 1<br>1 1 \| 1 |
| Inverter | | x = A' | A \| x<br>0 \| 1<br>1 \| 0 |
| Buffer | | x = A | A \| x<br>0 \| 0<br>1 \| 1 |
| NAND | | x = (AB)' | A B \| x<br>0 0 \| 1<br>0 1 \| 1<br>1 0 \| 1<br>1 1 \| 0 |
| NOR | | x = (A + B)' | A B \| x<br>0 0 \| 1<br>0 1 \| 0<br>1 0 \| 0<br>1 1 \| 0 |
| Exclusive-OR (XOR) | | x = A ⊕ B<br>or<br>x = A'B + AB' | A B \| x<br>0 0 \| 0<br>0 1 \| 1<br>1 0 \| 1<br>1 1 \| 0 |
| Exclusive-NOR or equivalence | | x = (A ⊕ B)'<br>or<br>x = A'B' + AB | A B \| x<br>0 0 \| 1<br>0 1 \| 0<br>1 0 \| 0<br>1 1 \| 1 |

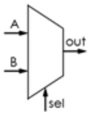
D Flip-Flop





Timing Diagram of Edge Triggered D Flip-Flop

Multiplexer



Task: Design an 8-bit data register.

## Number Notation

The AVR documentation and assembler uses several notations for numbers, and we will follow these in the lecture notes.

By default, numbers are decimal. Thus “**`10`**” means “ten, decimal”.

Hexadecimal numbers are written like **`0xAB`**, or **`0xaB,`** or **`0xAb`**. The leading character is a zero. Sometimes hexadecimal number are written as $AB.

Binary numbers are written as **`0b10101011.`** The leading character is a zero.

Octal numbers begin with “**`0`**” (zero) are octal. For example: **`0253`**

## AVR Architecture

- RISC (as Opposed to CISC). RISC implies
  - Fewer instructions (however, ironically, AVR processors have 100+ instructions …)
  - Manufacturers can use chip "real-estate" to make instruction's execution efficient → RISC processors are normally thought of as fast and efficient
  - A programmer using a RISC processor must construct complex instructions, that may be available in a CISC processor, using a reduced set of instructions
  - Many registers (ATmega88P has 32 general purpose registers). This means program variables can be kept in registers, rather than memory. Register access is faster than memory access → RISC processors are normally thought of as fast and efficient
- Most instructions execute in 1 clock cycle
- Terminology: MIPS = Millions Instructions Per Second
- AVR with 8 MHz clock can approach 8 MIPS

## RISC (AVR) vs. CISC Architecture

**Calculate: A = ((A and 0x84) + (B eor C)) or 0x80**

and: bitwise and

eor: bitwise exclusive or

or: bitwise or

| *AVR* code | CISC code |
|---|---|
| `EOR  B,C` | `MOV  ACC,C` |
| `ANDI A,#84h` | `EOR  ACC,B` |
| `ADD  A,B` | `MOV  TMP,ACC` |
| `ORI  A,#80h` | `MOV  ACC,A` |
| | `AND  ACC,#84h` |
| | `ADD  ACC,TMP` |
| | `OR   ACC,#80h` |
| | `MOV  A,ACC` |
| 8 bytes<br>4 clocks | 12-16 bytes<br>48-96 clocks |

A, B, C, ACC, and TMP: 8-bit registers

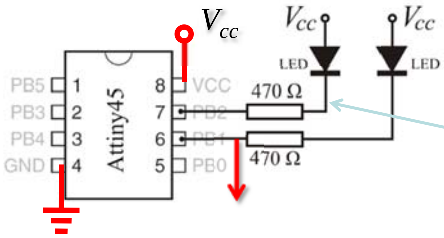
# AVR Architecture

- **The AVR Enhanced RISC Microcontrollers offer an architecture concept for high performance and low-power consumption simultaneously**

- **RISC architecture and instruction set optimized for efficient code density with built-in support for high-level languages (i.e., C).**

# AVR Microcontroller

**Example.** Turning LEDs on and off



Note LEDs will light up when ports pins are pulled LOW

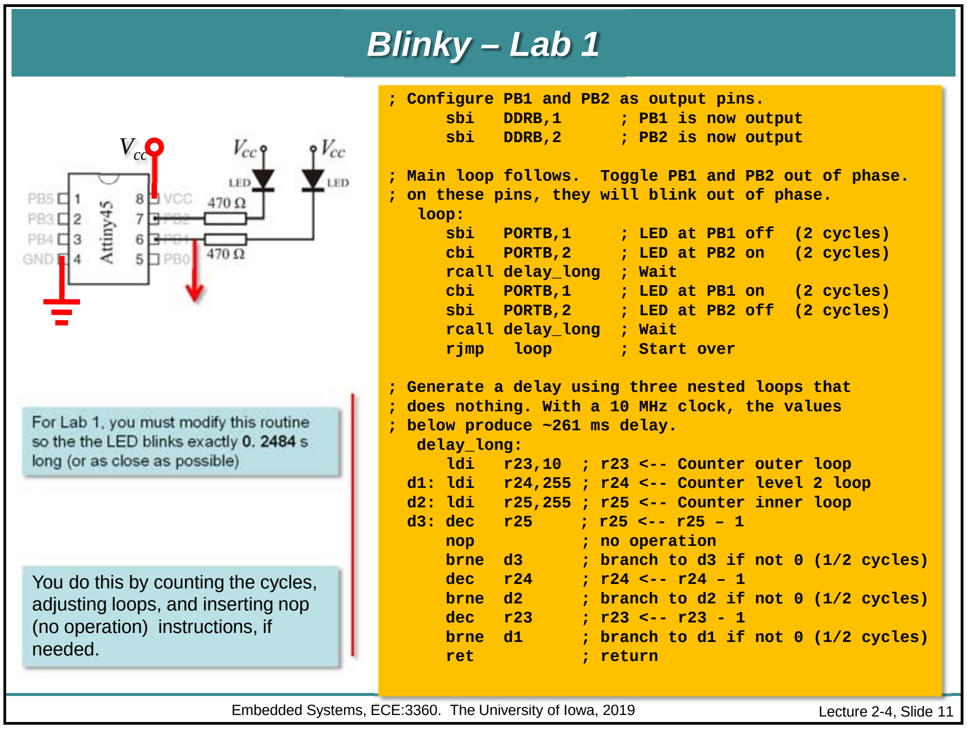
Let's set **bit 2 of port B high**, turning the corresponding **LED off**:

```
PORTB = PORTB | 0x04
```

Microcontrollers such as PIC/AVR, etc. are optimized for such tasks and one would use

```
SBI   0x18,2
```

On the AVR, this instruction is executed in 2 clock cycles



Vcc
Vcc
Vcc
LED
LED
470 Ω
470 Ω
Attiny45
PB5 1
PB3 2
PB4 3
GND 4
8 VCC
7 PB2
6 PB1
5 PB0

# Blinky – Lab 1

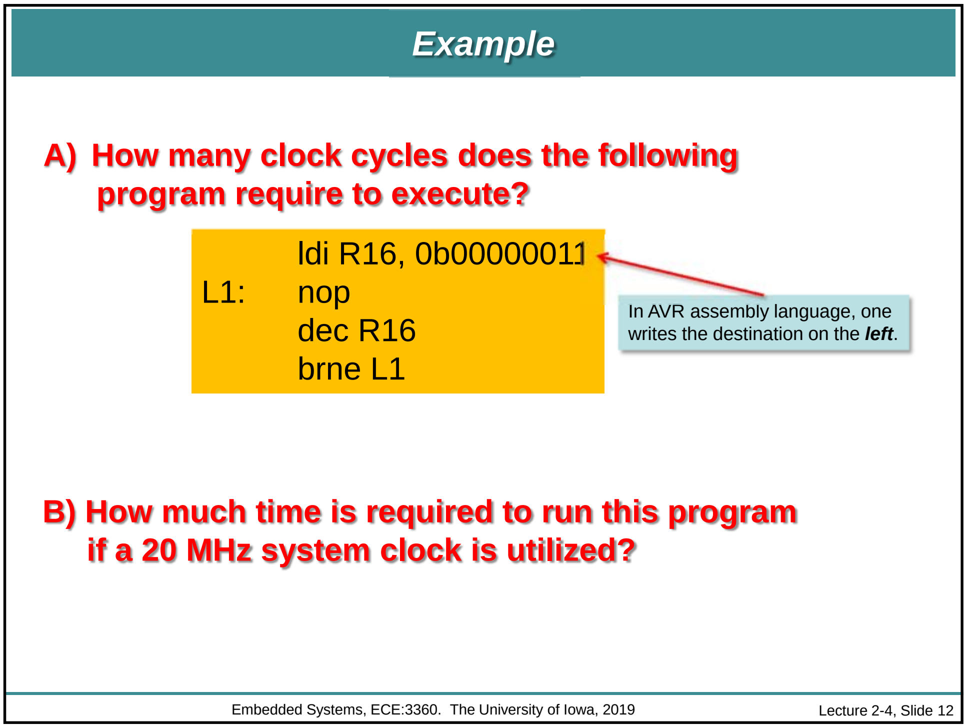
```
; Configure PB1 and PB2 as output pins.
      sbi   DDRB,1      ; PB1 is now output
      sbi   DDRB,2      ; PB2 is now output

; Main loop follows.  Toggle PB1 and PB2 out of phase.
; on these pins, they will blink out of phase.
   loop:
      sbi   PORTB,1     ; LED at PB1 off  (2 cycles)
      cbi   PORTB,2     ; LED at PB2 on   (2 cycles)
      rcall delay_long  ; Wait
      cbi   PORTB,1     ; LED at PB1 on   (2 cycles)
      sbi   PORTB,2     ; LED at PB2 off  (2 cycles)
      rcall delay_long  ; Wait
      rjmp   loop       ; Start over

; Generate a delay using three nested loops that
; does nothing. With a 10 MHz clock, the values
; below produce ~261 ms delay.
   delay_long:
      ldi   r23,10   ; r23 <-- Counter outer loop
  d1: ldi   r24,255 ; r24 <-- Counter level 2 loop
  d2: ldi   r25,255 ; r25 <-- Counter inner loop
  d3: dec   r25      ; r25 <-- r25 – 1
      nop            ; no operation
      brne  d3       ; branch to d3 if not 0 (1/2 cycles)
      dec   r24      ; r24 <-- r24 – 1
      brne  d2       ; branch to d2 if not 0 (1/2 cycles)
      dec   r23      ; r23 <-- r23 - 1
      brne  d1       ; branch to d1 if not 0 (1/2 cycles)
      ret            ; return
```

For Lab 1, you must modify this routine so the the LED blinks exactly **0. 2484** s long (or as close as possible)

You do this by counting the cycles, adjusting loops, and inserting nop (no operation) instructions, if needed.



# Example

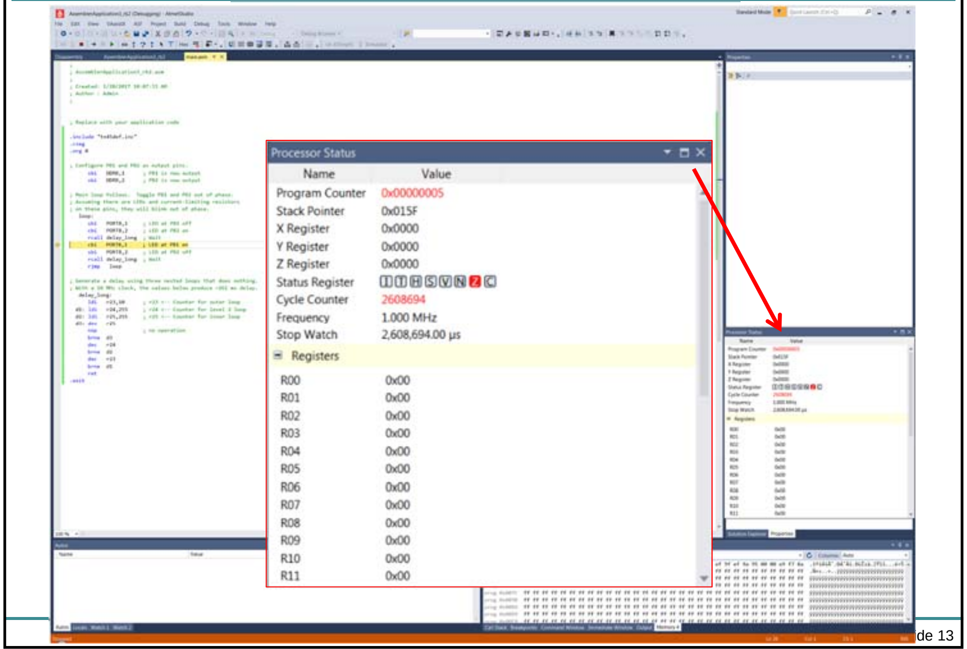
**A) How many clock cycles does the following program require to execute?**

```
        ldi R16, 0b00000011
L1:     nop
        dec R16
        brne L1
```

In AVR assembly language, one writes the destination on the ***left***.

**B) How much time is required to run this program if a 20 MHz system clock is utilized?**

## Simulation in Atmel Studio7



Processor Status

| Name | Value |
|---|---|
| Program Counter | 0x00000005 |
| Stack Pointer | 0x015F |
| X Register | 0x0000 |
| Y Register | 0x0000 |
| Z Register | 0x0000 |
| Status Register | I T H S V N Z C |
| Cycle Counter | 2608694 |
| Frequency | 1.000 MHz |
| Stop Watch | 2,608,694.00 µs |
| Registers | |
| R00 | 0x00 |
| R01 | 0x00 |
| R02 | 0x00 |
| R03 | 0x00 |
| R04 | 0x00 |
| R05 | 0x00 |
| R06 | 0x00 |
| R07 | 0x00 |
| R08 | 0x00 |
| R09 | 0x00 |
| R10 | 0x00 |
| R11 | 0x00 |

## Review: Types of Memory

- ROM – Read Only Memory. Microcontrollers generally don’t have ROM. Rather they implement some EEPROM.
- EEPROM – Electrically Erasable Programmable ROM. Functionally, this is non-volatile memory that one can reprogram. One can program individual bytes. Most microcontrollers have a small amount of EEPROM
- RAM – Random Access Memory. On desktop machines this is normally DRAM for “dynamic” RAM. This means that the data must be continually be refreshed. RAM is used for program variables.
- SRAM – Static RAM. A type of memory where the word *static* indicates that, unlike *dynamic* RAM (DRAM), it does not need to be periodically refreshed, SRAM is still *volatile* in the sense that data is eventually lost when the memory is not powered.
- Microcontrollers (except for very low-end parts) have some SRAM.

## Review: Types of Memory

- FLASH – A specific type of EEPROM that is erased and programmed in large blocks.
- It is a technology that is primarily used in memory cards and USB memory sticks for general storage and transfer of data between computers and other digital products.
- Flash memory costs far less than byte-programmable EEPROM and therefore has become the dominant technology wherever a significant amount of non-volatile, solid-state storage is needed.
- Flash memory has slow write, but fast read times.
- On microcontrollers, flash memory is generally used for program code.
- Flash is non-volatile → retains its contents when power is cycled.

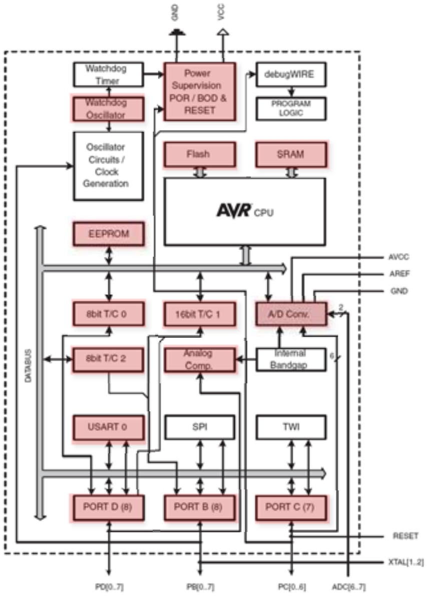
## “Smaller” Microntroller: ATmega48P/88P/168P/328P

Block Diagram of ATmega48P/88P/168P/328P

This refers to size of Flash or program memory: 4K Bytes, 8 K Bytes, …

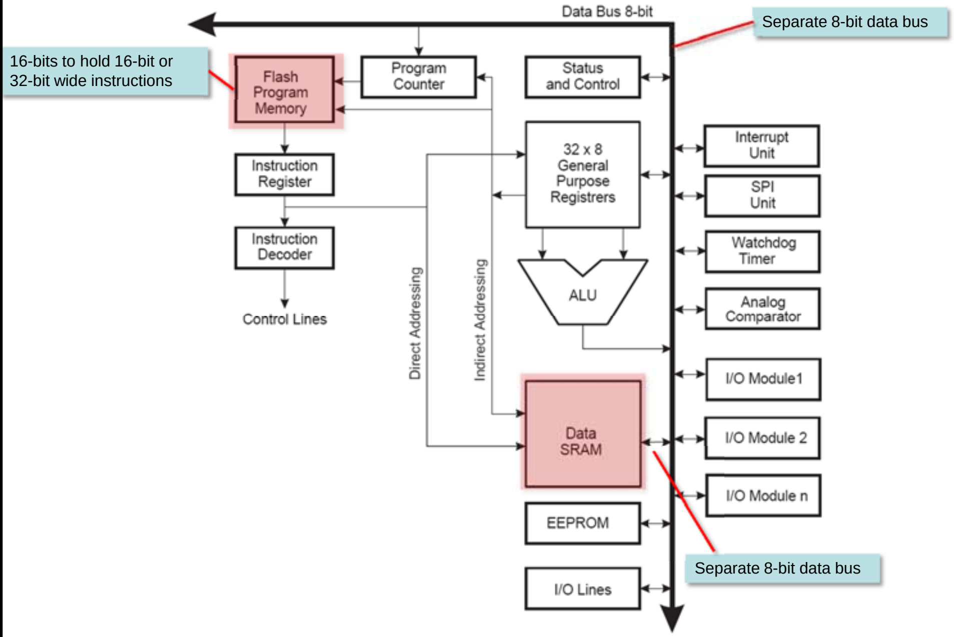
**Atmega88P Highlights**

- 32 general-purpose registers
- 1K SRAM
- 512 Bytes EEPROM
- Two 8-bit counters/timers + prescalers
- One 16-bit counter/timer + prescaler
- Three I/O ports: 23 programmable lines
- Analog comparator and multiplexed 10-bit A/D converter
- Programmable USART
- Power supervision POR, BOD, RESET
- Watchdog
- Six PWM Channels
- $2.30



## AVR Architecture

- Harvard Architecture (as Opposed to Von Neuman)
  - This means there are separate data and program (instruction) memories (bus)
  - Thus, even though one can in general use executable code to write to Flash memory, on AVRs this is not possible, and Flash memory is strictly read-only when the AVR runs
- All AVR processors have on-board In System Programmable (ISP) Flash memory
  - Flash memory is non-volatile and reprogrammable
  - No need for external EEPROM or ROM to hold program code
  - The code space is 16-bits wide to hold the 16-bit or 32-bit instructions. Most instructions are 16-bits wide.
- In System Programmable (ISP) means one can program the controller while it is in the circuit (very useful).
- Separate, 8-bit SRAM is for program variables.

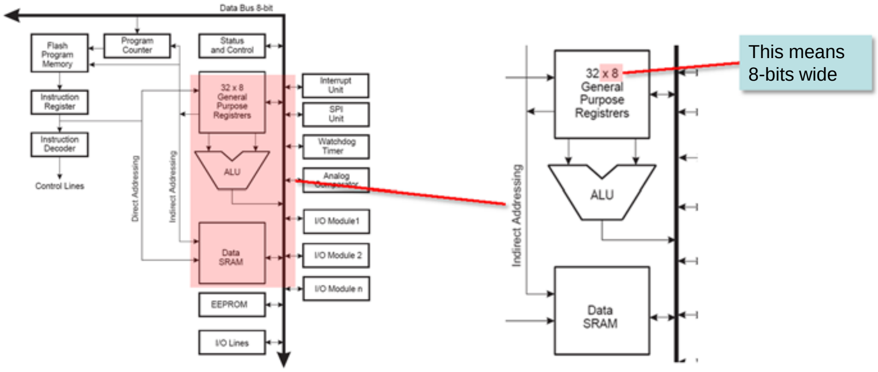


Separate 8-bit data bus

16-bits to hold 16-bit or 32-bit wide instructions

Separate 8-bit data bus

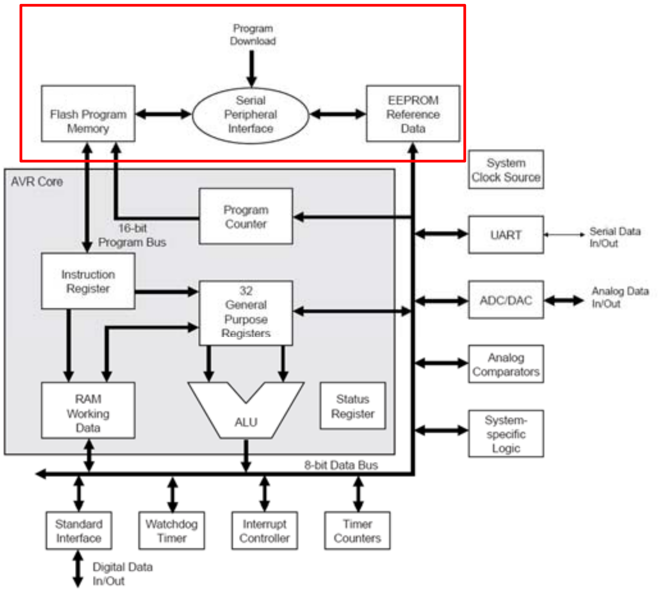
## Arithmetic Logic Unit ALU



This means 8-bits wide

The ALU performs the basic logic and arithmetic operations of the controller: test bits, add register contents, … and updates the destination. AVRs do not divide their clock, and most ALU operations are performed in a single clock cycle.

The ALU updates the Status Register, or **SREG**. For example, if the result of a subtraction is zero, the ALU sets the Z bit (flag) in **SREG**.

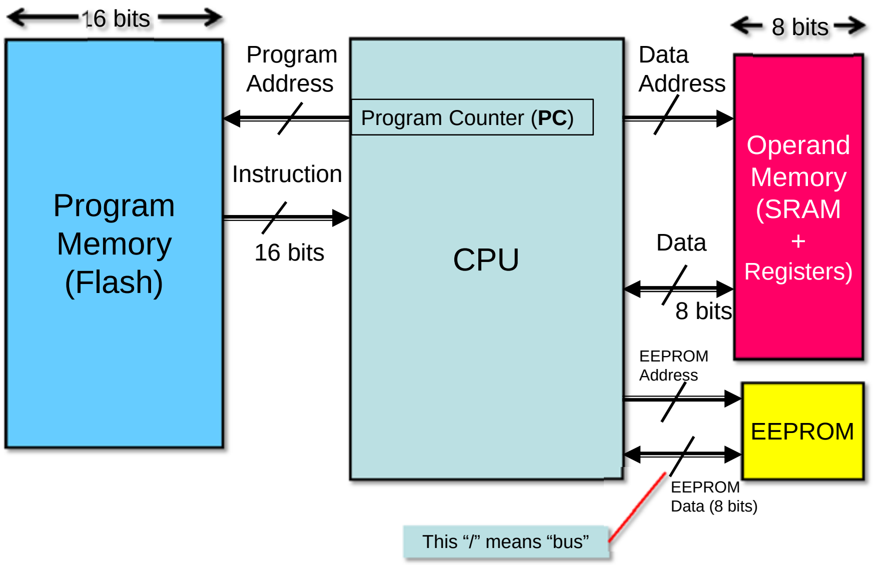


The AVR core is based on a Harvard architecture with separate memories and buses for program and data (Figure 2). The memory spaces in the AVR architecture are all linear and regular memory maps.

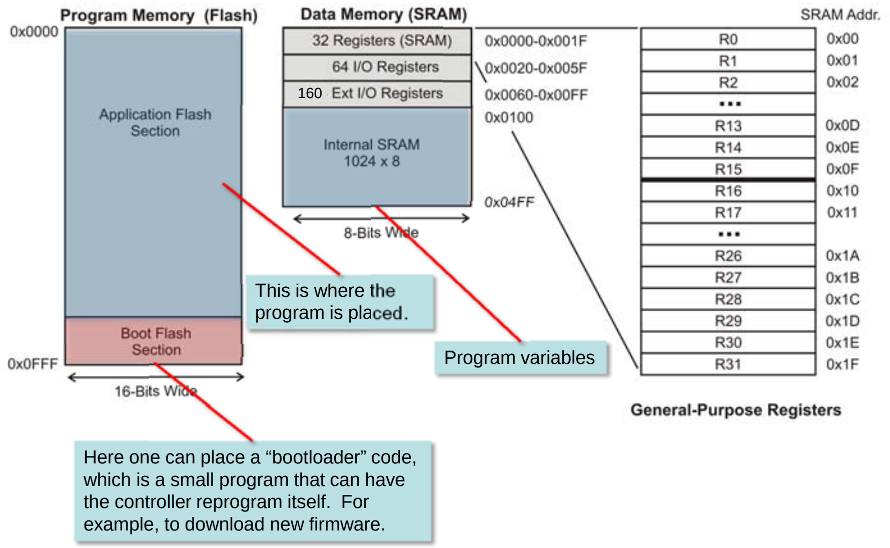
## AVR Architecture

- AVRs have a two stage, single level pipeline design
  - This means the next machine instruction is fetched as the current one is executing.
  - Since almost all operations on registers R0 - R31 are single cycle, the AVR can achieve up to 1 MIPS per MHz.
- The AVR family of processors were designed with the efficient execution of compiled C code in mind
  - Has several built-in pointers for the task
- Variable instruction word width: 16 or 32 bits.
- Most instructions are 16 bits wide

## AVR Memories



# ATmega88 Memory Map



This is where the program is placed.

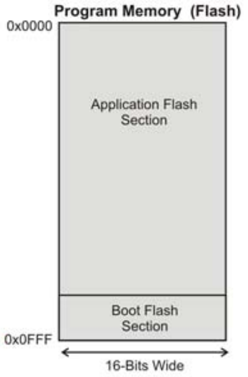
Program variables

Here one can place a “bootloader” code, which is a small program that can have the controller reprogram itself. For example, to download new firmware.

# ATmega88 Memory Map

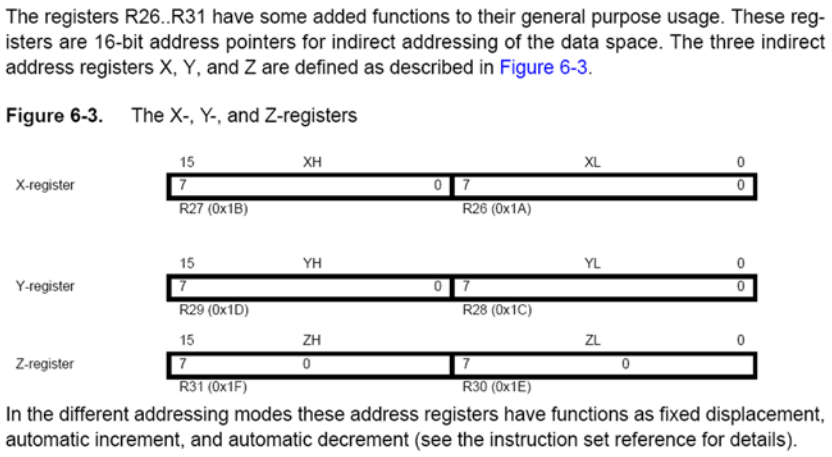
**Question:** What is the largest program that can fit in an ATmega88P?

**Answer:** The ATmega88 has 8K Bytes or Flash/program memory. Because AVR instructions are 16- or 32-bits wide, program memory is organized as 4K x 16. Thus, assuming only 16-bit instructions are used, then the largest program is 4K or 4096 instructions.



0x0FFF = 4,095 decimal

# Index Registers



The registers R26..R31 have some added functions to their general purpose usage. These registers are 16-bit address pointers for indirect addressing of the data space. The three indirect address registers X, Y, and Z are defined as described in Figure 6-3.

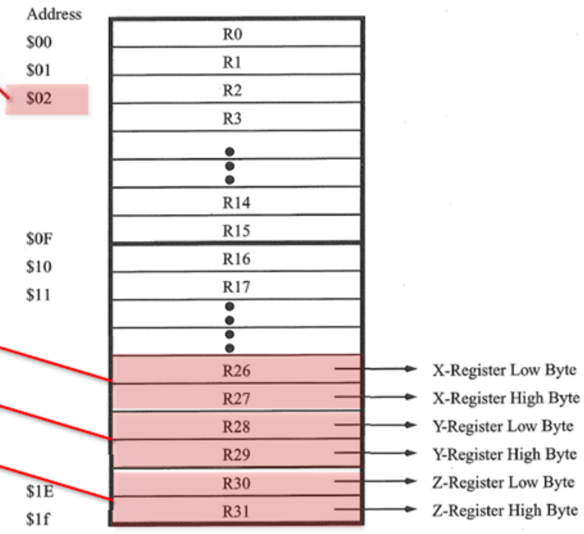
**Figure 6-3.** The X-, Y-, and Z-registers

X-register
15
XH
7
0
R27 (0x1B)
XL
0
7
0
R26 (0x1A)
Y-register
15
YH
7
0
R29 (0x1D)
YL
0
7
0
R28 (0x1C)
Z-register
15
ZH
7
0
R31 (0x1F)
ZL
0
7
0
R30 (0x1E)

In the different addressing modes these address registers have functions as fixed displacement, automatic increment, and automatic decrement (see the instruction set reference for details).

# Index Registers

Note how we indicate hexadecimal numbers as starting with either 0x or $"

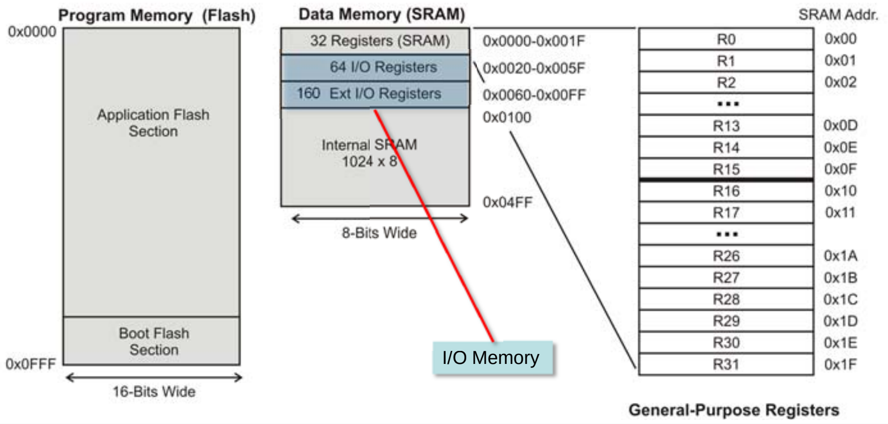


R26 and R27 form the *X*-register

R28 and R29 form the *Y*-register

R30 and R31 form the *Z*-register

## I/O Memory



The input/output (I/O) memory is the gateway to the various peripheral component on AVR processors, such as the ADC, UART, Timers/Counters, and the I/O ports.

The I/O memory is implemented in SRAM and one can access it as I/O register or SRAM

For example, the I/O port PORTC is I/O register 0x08. The equivalent SRAM address is (0x28)

## I/O Memory

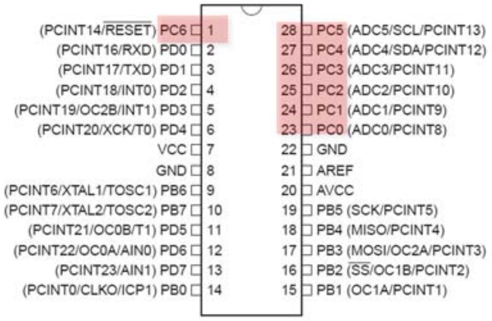
The input/output (I/O) memory is the gateway to the various peripheral components on AVR processors, such as the ADC, UART, Timers/Counters, and the I/O ports.

The I/O memory is implemented in SRAM and one can access it as I/O registers or SRAM

For example, the I/O port PORTC is I/O register 0x08. The equivalent SRAM address is (0x28)

The **IN** and **OUT** instructions are used to access the I/O registers. For example, assuming data direction for PORTC (0x08) is set up properly as output, the following code snippet sends the bit pattern 0b00000010 to PORTC. This will pull all the pins low, except PC1

```
LDI   R16,0b00000010 ; load R16
OUT   0x08,R16       ; out to PORTC
```

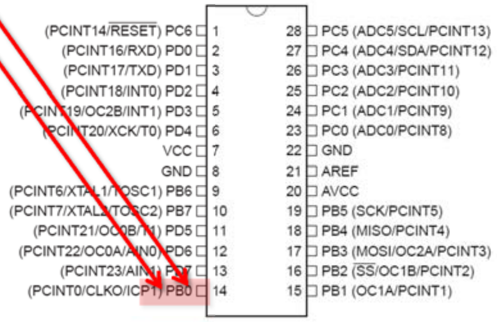


# I/O Memory

In addition to the **IN** and **OUT** instructions, the AVR also supports the **SBI** (Set Bit in I/O Register) and **CBI** (Clear Bit in I/O Registers) instructions to set or clear individual bits in I/O memory.

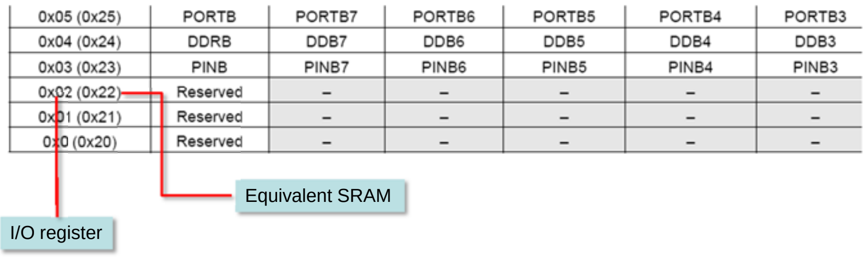
For  example, the I/O memory for **PORTB** is 0x05 the following code snippet sets **PB0**.  That is, it makes it “1” or “high” .  Assuming **PB0** has been properly configured, after the instruction, **PB0** will have a high voltage.  For example, close to +5 V if the controller is running off a 5 V supply.

```
SBI  0x05,0   ; Set bit in I/O register
```



# I/O Memory

The Atmel datasheets show the equivalent SRAM address in parenthesis



| | | | | | | |
|---|---|---|---|---|---|---|
| 0x05 (0x25) | PORTB | PORTB7 | PORTB6 | PORTB5 | PORTB4 | PORTB3 |
| 0x04 (0x24) | DDRB | DDB7 | DDB6 | DDB5 | DDB4 | DDB3 |
| 0x03 (0x23) | PINB | PINB7 | PINB6 | PINB5 | PINB4 | PINB3 |
| 0x02 (0x22) | Reserved | – | – | – | – | – |
| 0x01 (0x21) | Reserved | – | – | – | – | – |
| 0x0 (0x20) | Reserved | – | – | – | – | – |

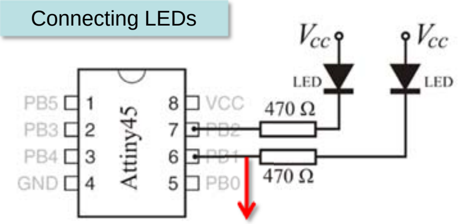
Equivalent SRAM

I/O register

# I/O Memory

**Example.** Turning LEDs on and off

Connecting LEDs



Note LEDs will light up when ports pins are pulled LOW

Data Direction Register

Configuring port pins for output

```
sbi    DDRB,1      ; PB1 is now output
sbi    DDRB,2      ; PB2 is now output
```

Turn LED at **PB1** OFF

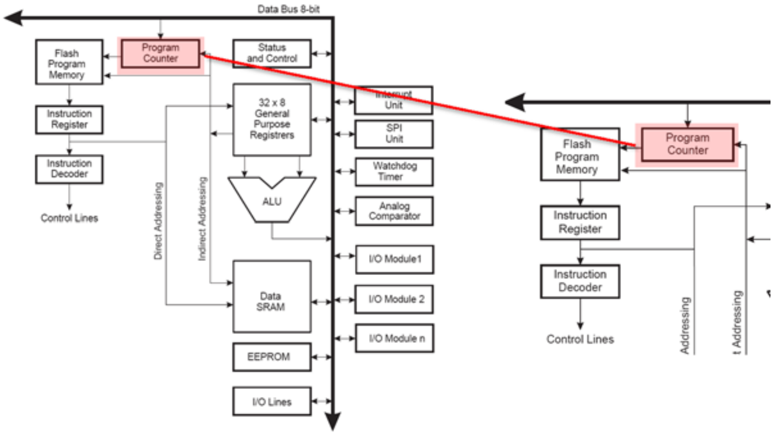
```
sbi    PORTB,1     ; LED at PB1 off
```

Turn LED at **PB2** ON

```
cbi    PORTB,2     ; LED at PB2 on
```

# Program Counter or PC

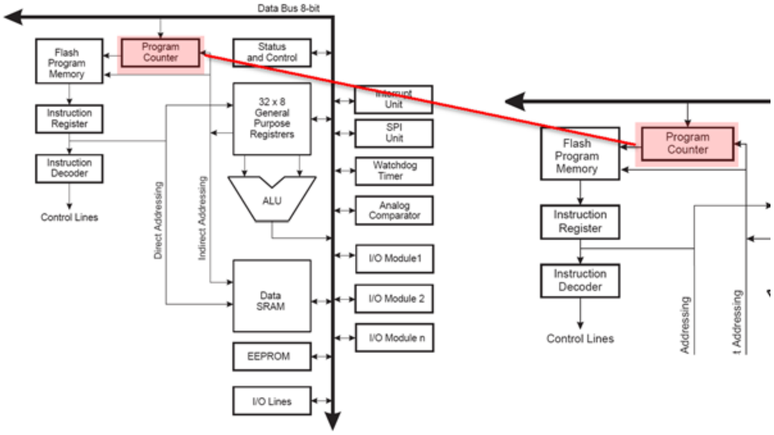
The program counter or **PC** points to the next instruction in program memory



The width of the **PC** depends on the size of program memory for the controller. For example, on the ATmega88PA (8K Flash), **PC** is 12 bits wide, while on the ATtiny45 (4K Flash), **PC** is 11 bits wide.

## Program Counter or PC

Branch instructions (**BRNE**, **RCALL**, **RJMP, …**) change the **PC** and points it to what is required by the call, jump, etc.
During interrupts and subroutine calls, the return address **PC** is stored on the stack. The **ret** and **reti** instructions pop the **PC** off the stack.



## Logic States

Digital signals may be in one of three states:

***State 1: High*, or "1".** Using positive logic polarity, this corresponds to a voltage level closer to the power supply than to ground.

**Example.** For an ATmega88PA and Vcc = 5 V, the data sheet indicates that an input voltage larger than 0.6×Vcc = 3 V is considered high, or logic "1".

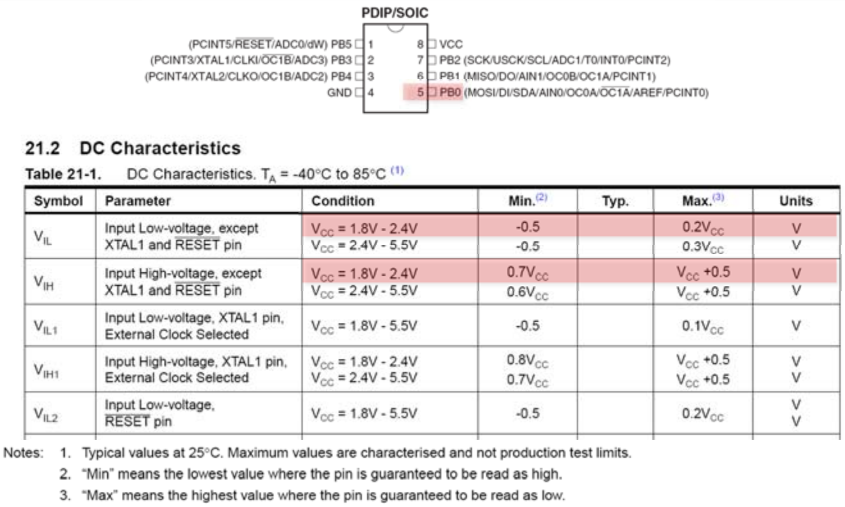
***State 2: Low*, or "0".** Using positive logic polarity, this corresponds to a voltage level closer to ground than to the power supply.

**Example.** For an ATmega88PA and Vcc = 5 V, the data sheet indicates that an input less than 0.3×Vcc = 1.5 V is treated as a logic low or "0"

**State 3: Tristate or high-Z, or simply Z.** In this case the digital line is in a high-impedance state or "floating". External components, often a pull-up resistor can pull the buss high or low.

**In negative logic polarity**, voltages levels closer to the supply than to ground are "0", and voltage levels closer to ground than to the power supply are "1". This is used in some serial communication protocols.

**Example.** Consider the table below, extracted from the “DC Characteristics” section of the ATtiny45/V microcontroller. Assume the power supply voltage is 2 V, and that PB0 is configured as digital input. Determine the threshold voltages for logic 1 and 0 on input.


PDIP/SOIC
(PCINT5/RESET/ADC0/dW) PB5 □ 1
(PCINT3/XTAL1/CLKI/OC1B/ADC3) PB3 □ 2
(PCINT4/XTAL2/CLKO/OC1B/ADC2) PB4 □ 3
GND □ 4
8 □ VCC
7 □ PB2 (SCK/USCK/SCL/ADC1/T0/INT0/PCINT2)
6 □ PB1 (MISO/DO/AIN1/OC0B/OC1A/PCINT1)
5 □ PB0 (MOSI/DI/SDA/AIN0/OC0A/OC1A/AREF/PCINT0)

### 21.2 DC Characteristics

**Table 21-1.** DC Characteristics. $T_A$ = -40°C to 85°C [1]

| Symbol | Parameter | Condition | Min.[2] | Typ. | Max.[3] | Units |
|---|---|---|---|---|---|---|
| $V_{IL}$ | Input Low-voltage, except XTAL1 and RESET pin | $V_{CC}$ = 1.8V - 2.4V<br>$V_{CC}$ = 2.4V - 5.5V | -0.5<br>-0.5 | | $0.2V_{CC}$<br>$0.3V_{CC}$ | V<br>V |
| $V_{IH}$ | Input High-voltage, except XTAL1 and RESET pin | $V_{CC}$ = 1.8V - 2.4V<br>$V_{CC}$ = 2.4V - 5.5V | $0.7V_{CC}$<br>$0.6V_{CC}$ | | $V_{CC}$ +0.5<br>$V_{CC}$ +0.5 | V<br>V |
| $V_{IL1}$ | Input Low-voltage, XTAL1 pin, External Clock Selected | $V_{CC}$ = 1.8V - 5.5V | -0.5 | | $0.1V_{CC}$ | V |
| $V_{IH1}$ | Input High-voltage, XTAL1 pin, External Clock Selected | $V_{CC}$ = 1.8V - 2.4V<br>$V_{CC}$ = 2.4V - 5.5V | $0.8V_{CC}$<br>$0.7V_{CC}$ | | $V_{CC}$ +0.5<br>$V_{CC}$ +0.5 | V<br>V |
| $V_{IL2}$ | Input Low-voltage, RESET pin | $V_{CC}$ = 1.8V - 5.5V | -0.5 | | $0.2V_{CC}$ | V<br>V |

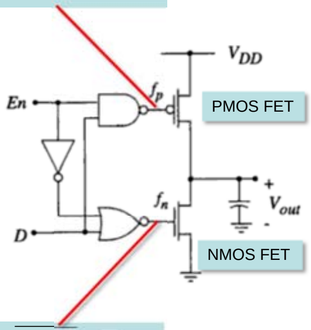
Notes:
1. Typical values at 25°C. Maximum values are characterised and not production test limits.
2. “Min” means the lowest value where the pin is guaranteed to be read as high.
3. “Max” means the highest value where the pin is guaranteed to be read as low.

**Solution.** **PB0** is not XTAL or RESET, so $V_{IL}$ and $V_{IH}$ apply.

$V_{I0}$ = 0.2Vcc = 0.2×2 = 0.4 V, and $V_{I1}$ = 0.7Vcc = 0.7×2 = 1.4 V

# Tristate Circuit (in CMOS)

$f_p = \overline{D \bullet En} = \overline{D} + \overline{En}$



PMOS FET

NMOS FET

$f_n = \overline{D + \overline{En}} = \overline{D} \bullet En$

Buffer with enable

$En = 0$

$f_p = \overline{D} + \overline{En} = \overline{D} + 1 = 1$ PMOS FET is off

$f_n = \overline{D} \bullet En = \overline{D} \bullet 0 = 0$ NMOS FET is off

Both FETs off ➔ Tristate when *En* =0

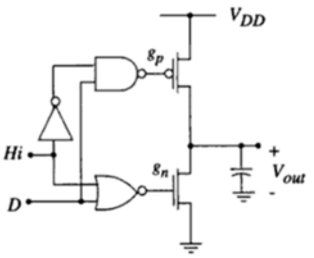
$En = 1$

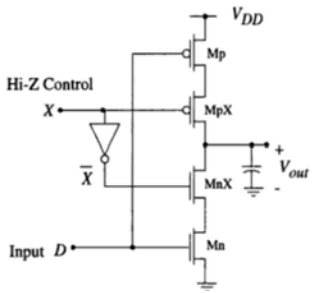
$f_p = \overline{D} + \overline{En} = \overline{D} + 0 = \overline{D}$

$f_n = \overline{D} \bullet En = \overline{D} \bullet 1 = \overline{D}$

Signal at the gate of both FETs is /*D*

## Other Tristate Circuits



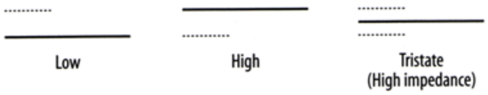


Self-Study

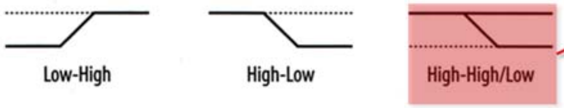
Self-Study

## Sidebar: Timing Diagrams

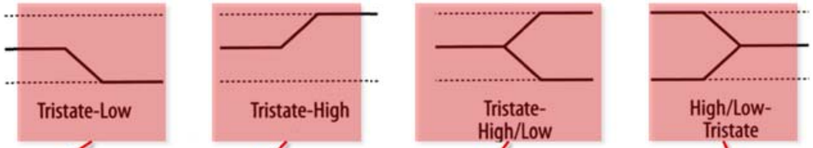
Representation of logic states



Transition of between states



Line is high, stays high or goes low

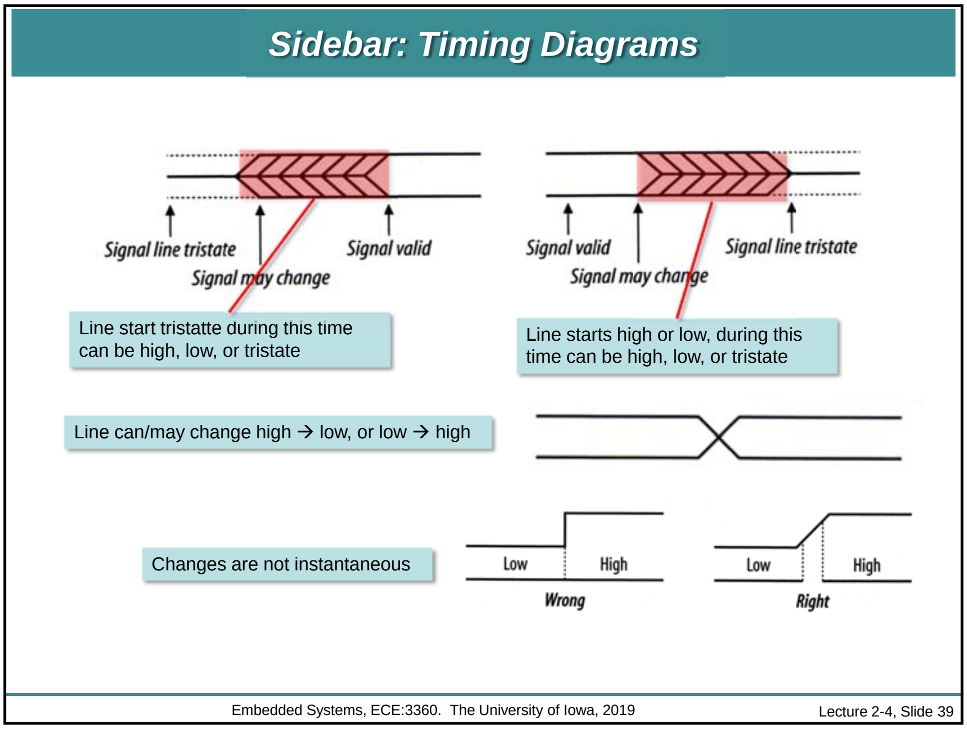


Line is tristate, is then pulled low

Line is tristate, is then pulled high

Line is tristate, and then pulled high or low

Line is how or low, and is then released to float → tristate



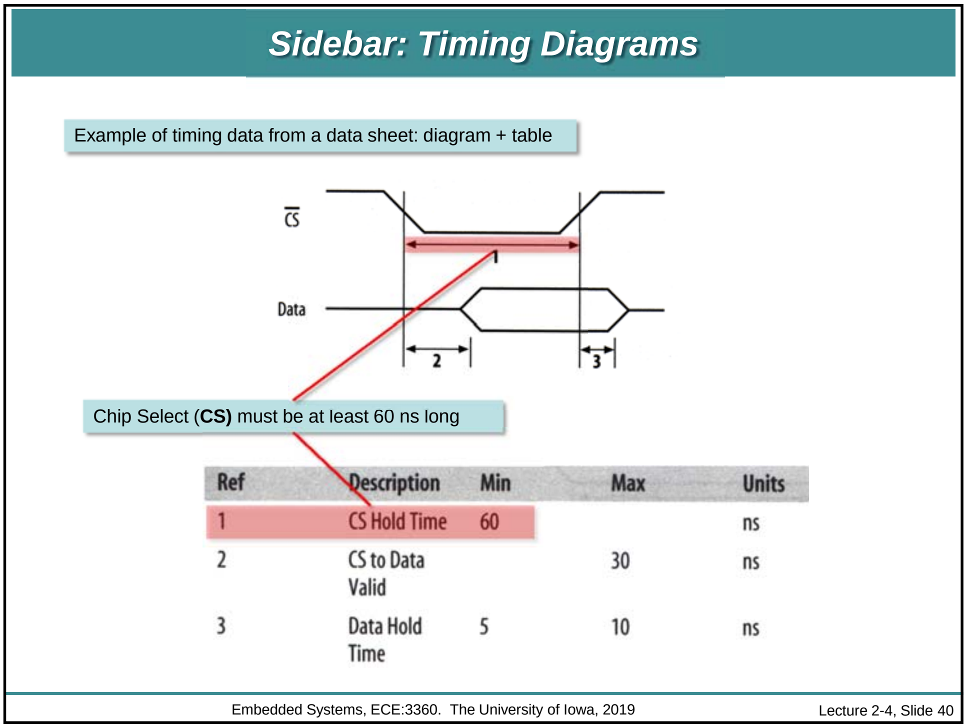
## Sidebar: Timing Diagrams

Line start tristatte during this time can be high, low, or tristate

Line starts high or low, during this time can be high, low, or tristate

Line can/may change high → low, or low → high

Changes are not instantaneous

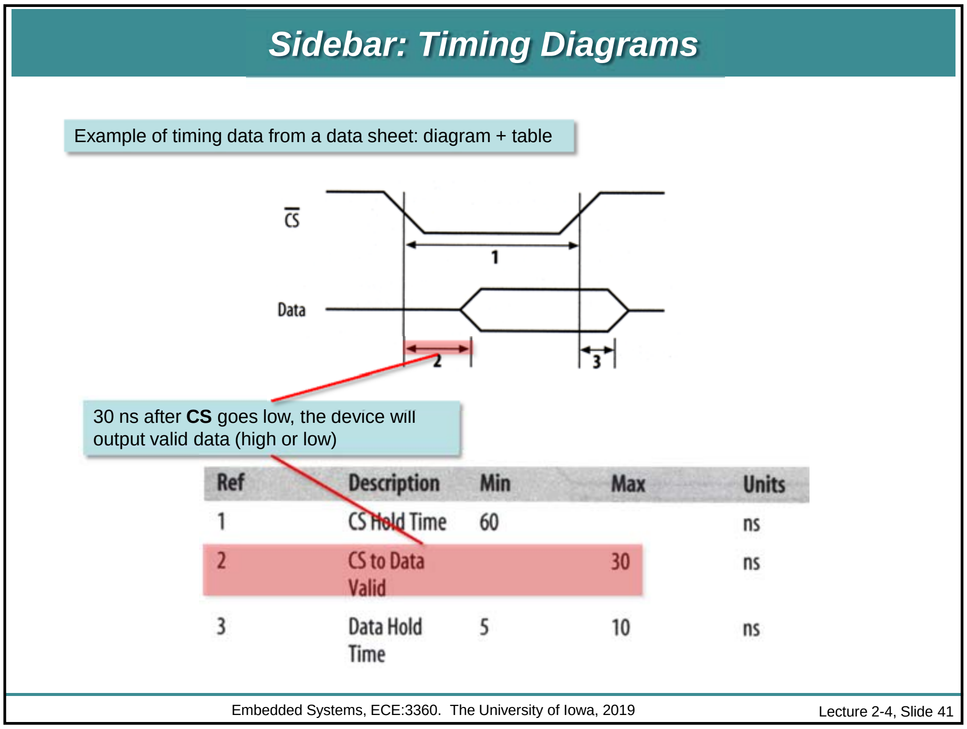


## Sidebar: Timing Diagrams

Example of timing data from a data sheet: diagram + table

Chip Select (**CS)** must be at least 60 ns long

| Ref | Description | Min | Max | Units |
|---|---|---|---|---|
| 1 | CS Hold Time | 60 | | ns |
| 2 | CS to Data Valid | | 30 | ns |
| 3 | Data Hold Time | 5 | 10 | ns |

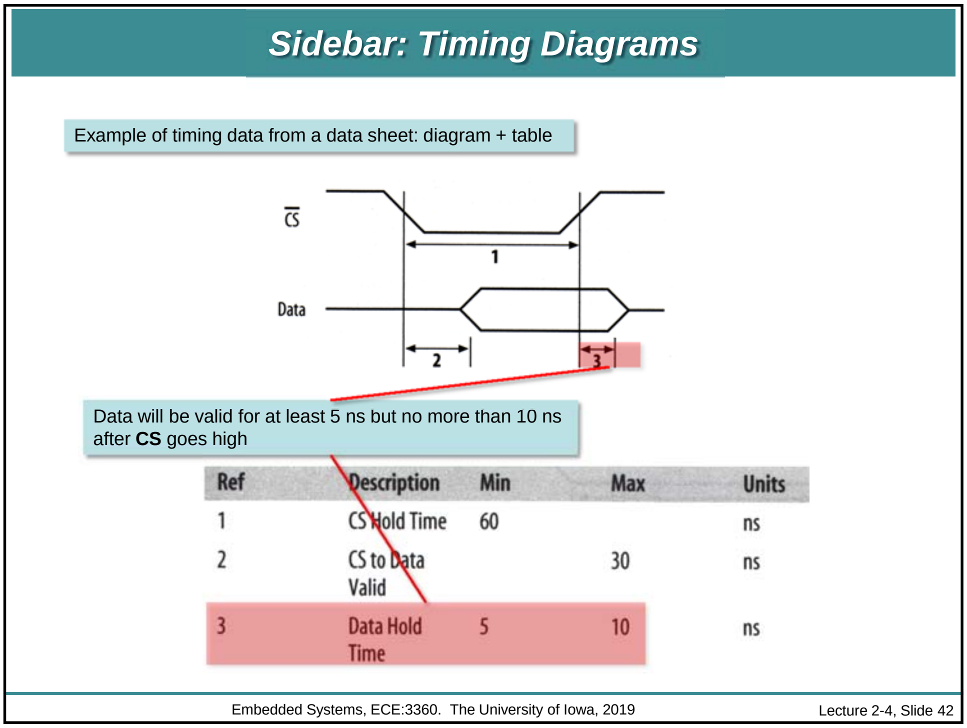


## Sidebar: Timing Diagrams

Example of timing data from a data sheet: diagram + table

30 ns after **CS** goes low, the device will output valid data (high or low)

| Ref | Description | Min | Max | Units |
|---|---|---|---|---|
| 1 | CS Hold Time | 60 | | ns |
| 2 | CS to Data Valid | | 30 | ns |
| 3 | Data Hold Time | 5 | 10 | ns |



## Sidebar: Timing Diagrams

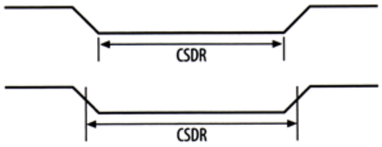
Example of timing data from a data sheet: diagram + table

Data will be valid for at least 5 ns but no more than 10 ns after **CS** goes high

| Ref | Description | Min | Max | Units |
|---|---|---|---|---|
| 1 | CS Hold Time | 60 | | ns |
| 2 | CS to Data Valid | | 30 | ns |
| 3 | Data Hold Time | 5 | 10 | ns |

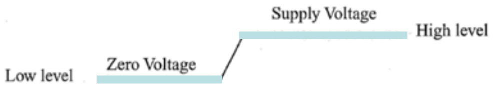
## Sidebar: Timing Diagrams

Other method of specifying timing data: directly on timing diagram

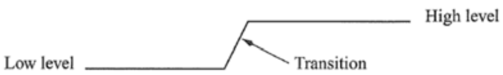


## Sidebar: Timing Diagrams

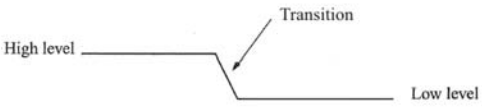
1. A low-level signal is zero voltage (actually a range of voltage around zero), and a high-level signal is supply voltage (or a range around it).



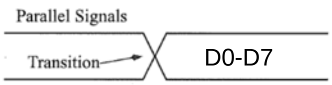
2. Transition of a low-level signal to high level.



3. Transition of a high-level signal to a low level.

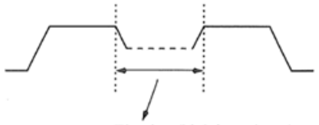


4. Transition of a bunch of parallel signals (called BUS) from one level to another.



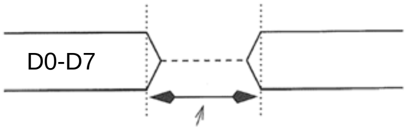
## Sidebar: Timing Diagrams

5. A signal that goes in a high-impedance state, also called a floating signal.



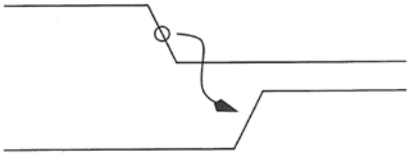
Floating (high impedence) duration of the signal

6. A BUS with floating signals.



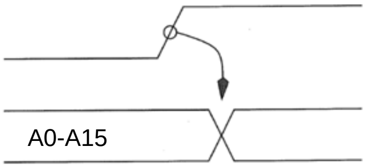
Duration of the signal when it is in high impedance, floating state

7. A change of condition on one signal causes a transition on another signal. The example shows a high-to-low signal transition causing a high-to-low-level transition on another signal.

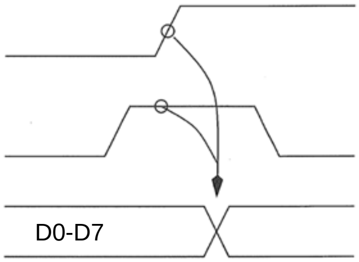


## Sidebar: Timing Diagrams

8. A transition on a signal causes a transition on a BUS.

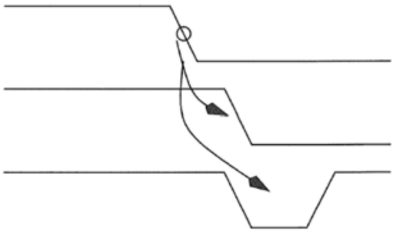


9. More than 1 condition must exist to force a transition of signals on the BUS. Example shows that a transition on one signal during the time when the other signal is at high level causes a transition on the BUS signals.



# Sidebar: Timing Diagrams

10. A condition on a signal causes changes on more than one signal level. The example shows that a high-to-low transition on one signal causes a high-to-low transition on the second signal and a pulse on another signal.

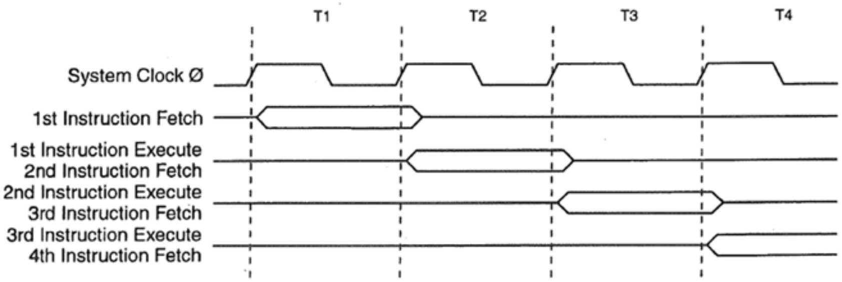


# AVR Memory Access

Clock on AVR can be internal RC (Resistor-Capacitor) or the clock can be external

The clock can be used without any division

The AVR has a two-stage pipeline, and the next instruction is fetched and decoded while the current instruction is executed.



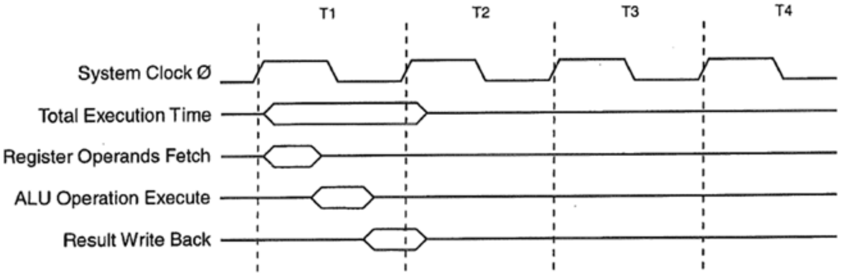
Instruction fetch/decode/execute sequence

## ALU Execution

ALU execution: register fetch, execute, write back. This is the sequence when the instruction is an ALU instruction. For example:

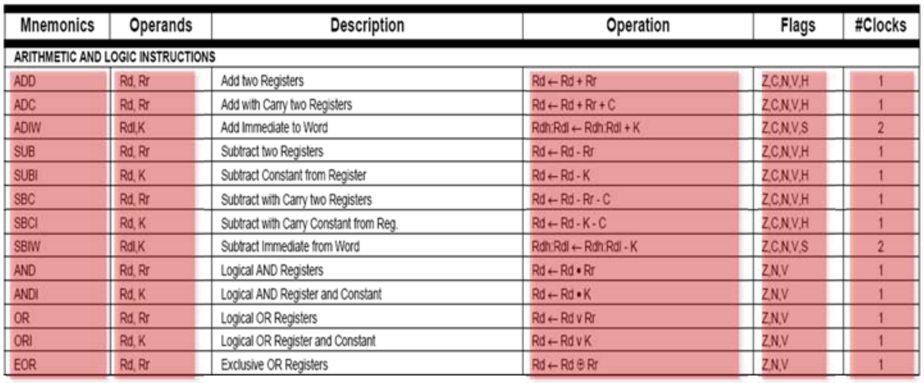
```
ADD R0, R1 ; Add registers R0 and R1.  The Result goes to R0
```

Note the destination is on the left



## Instruction Timing

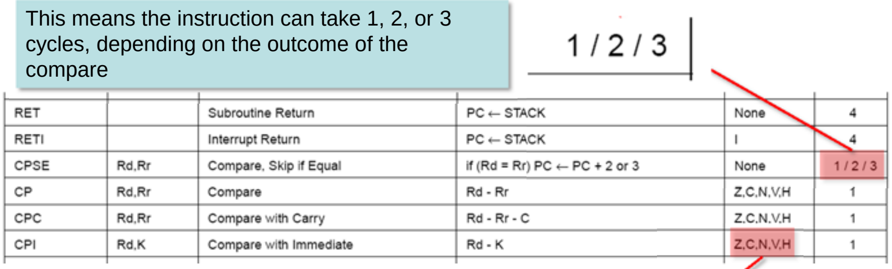
- **Why is this important?**
  - **Because precise timing is often very important in embedded systems**
  - **Precise timing is important in serial communications**
  - **…**



| Mnemonics | Operands | Description | Operation | Flags | #Clocks |
|---|---|---|---|---|---|
| ARITHMETIC AND LOGIC INSTRUCTIONS | | | | | |
| ADD | Rd, Rr | Add two Registers | Rd ← Rd + Rr | Z,C,N,V,H | 1 |
| ADC | Rd, Rr | Add with Carry two Registers | Rd ← Rd + Rr + C | Z,C,N,V,H | 1 |
| ADIW | Rdl,K | Add Immediate to Word | Rdh:Rdl ← Rdh:Rdl + K | Z,C,N,V,S | 2 |
| SUB | Rd, Rr | Subtract two Registers | Rd ← Rd - Rr | Z,C,N,V,H | 1 |
| SUBI | Rd, K | Subtract Constant from Register | Rd ← Rd - K | Z,C,N,V,H | 1 |
| SBC | Rd, Rr | Subtract with Carry two Registers | Rd ← Rd - Rr - C | Z,C,N,V,H | 1 |
| SBCI | Rd, K | Subtract with Carry Constant from Reg. | Rd ← Rd - K - C | Z,C,N,V,H | 1 |
| SBIW | Rdl,K | Subtract Immediate from Word | Rdh:Rdl ← Rdh:Rdl - K | Z,C,N,V,S | 2 |
| AND | Rd, Rr | Logical AND Registers | Rd ← Rd • Rr | Z,N,V | 1 |
| ANDI | Rd, K | Logical AND Register and Constant | Rd ← Rd • K | Z,N,V | 1 |
| OR | Rd, Rr | Logical OR Registers | Rd ← Rd v Rr | Z,N,V | 1 |
| ORI | Rd, K | Logical OR Register and Constant | Rd ← Rd v K | Z,N,V | 1 |
| EOR | Rd, Rr | Exclusive OR Registers | Rd ← Rd ⊕ Rr | Z,N,V | 1 |

# Instruction Timing

This means the instruction can take 1, 2, or 3 cycles, depending on the outcome of the compare



1 / 2 / 3

| | | | | | |
|---|---|---|---|---|---|
| RET | | Subroutine Return | PC ← STACK | None | 4 |
| RETI | | Interrupt Return | PC ← STACK | I | 4 |
| CPSE | Rd,Rr | Compare, Skip if Equal | if (Rd = Rr) PC ← PC + 2 or 3 | None | 1 / 2 / 3 |
| CP | Rd,Rr | Compare | Rd - Rr | Z,C,N,V,H | 1 |
| CPC | Rd,Rr | Compare with Carry | Rd - Rr - C | Z,C,N,V,H | 1 |
| CPI | Rd,K | Compare with Immediate | Rd - K | Z,C,N,V,H | 1 |

These are the bits in the status register, or **SREG.**

**Example**. Determine how many cycles the snippet below will take

```
      ldi   R27,0x04        ; R27 <-- 0x04   1 cycle
more:
      nop                   ; No Operation:  1 cycle
      dec   R27             ; R27 <-- R27-1: 1 cycle
      brne  more            ; Branch to more if not zero:   1/2 cycles
      ldi   R28,0xAB        ; R28 <-- 0xAB:  1 cycle
```

**Solution.** This is a simple loop in AVR assembly language. The first instruction initializes the loop counter with the number 4.

The **NOP** takes 1 cycle. ("**NOP**" is the no-operation instruction that does nothing)

The **DEC R27**, decrements **R27**, and sets the **Z** flag in **SREG** if **R27** is zero. This takes 1 cycle.

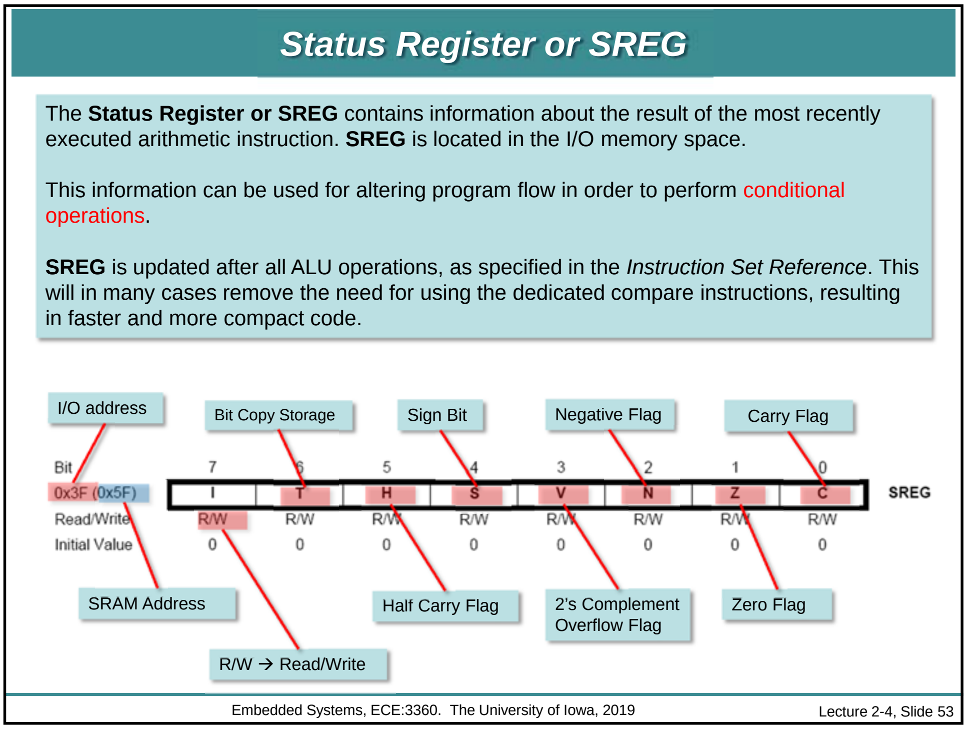
The **BRNE MORE** checks the **Z** flag. If it is not set (1), **R27** is not zero and the program branches to the label **MORE** and the loop starts over. A branch takes 2 cycles.

Thus, every time through the loop + branch takes 4 cycles.

The loop will be executed 4 times, but there are only three branches. On the 4th time though the loop, when **DEC R27** makes **R27** zero, the **Z** flag is set. This time there is no branch, and the **brne more** instruction takes 1 cycle.

The last instruction (**LDI R28,0xAB**) takes 1 cycle**.**

The total number of cycles is 1 + 3×4 + 3 + 1 = 17 cycles.



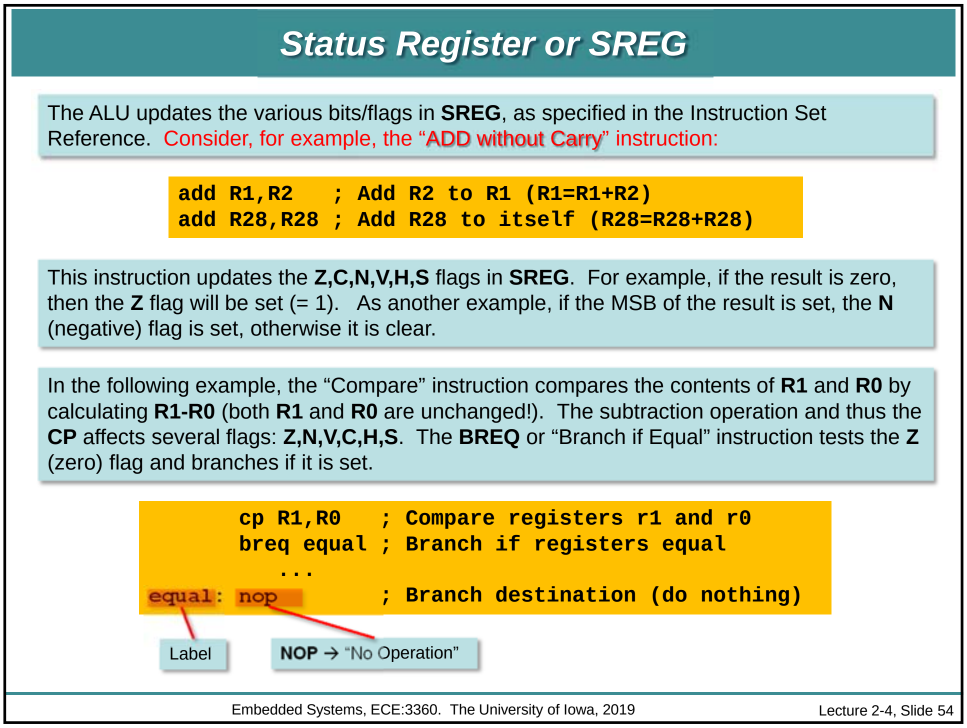
## Status Register or SREG

The **Status Register or SREG** contains information about the result of the most recently executed arithmetic instruction. **SREG** is located in the I/O memory space.

This information can be used for altering program flow in order to perform conditional operations.

**SREG** is updated after all ALU operations, as specified in the *Instruction Set Reference*. This will in many cases remove the need for using the dedicated compare instructions, resulting in faster and more compact code.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x3F (0x5F) | I | T | H | S | V | N | Z | C | SREG |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |



## Status Register or SREG

The ALU updates the various bits/flags in **SREG**, as specified in the Instruction Set Reference. Consider, for example, the "ADD without Carry" instruction:

```
add R1,R2   ; Add R2 to R1 (R1=R1+R2)
add R28,R28 ; Add R28 to itself (R28=R28+R28)
```

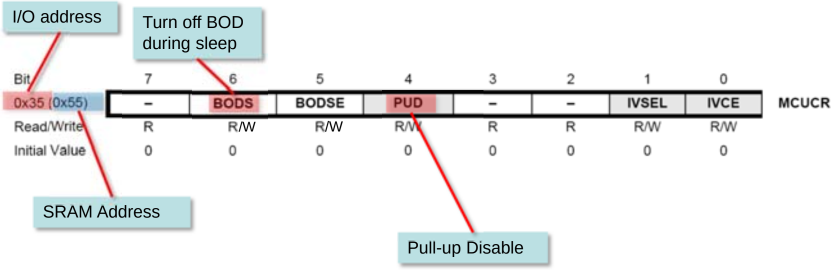
This instruction updates the **Z,C,N,V,H,S** flags in **SREG**. For example, if the result is zero, then the **Z** flag will be set (= 1). As another example, if the MSB of the result is set, the **N** (negative) flag is set, otherwise it is clear.

In the following example, the "Compare" instruction compares the contents of **R1** and **R0** by calculating **R1-R0** (both **R1** and **R0** are unchanged!). The subtraction operation and thus the **CP** affects several flags: **Z,N,V,C,H,S**. The **BREQ** or "Branch if Equal" instruction tests the **Z** (zero) flag and branches if it is set.

```
       cp R1,R0   ; Compare registers r1 and r0
       breq equal ; Branch if registers equal
          ...
equal: nop        ; Branch destination (do nothing)
```

# MCUCR: MCU General Control Register

The **MCUCR** is located in I/O memory. Its various bits allow general control of the microcontroller: brown-out, pull-ups etc.



| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x35 (0x55) | – | BODS | BODSE | PUD | – | – | IVSEL | IVCE | MCUCR |
| Read/Write | R | R/W | R/W | R/W | R | R | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

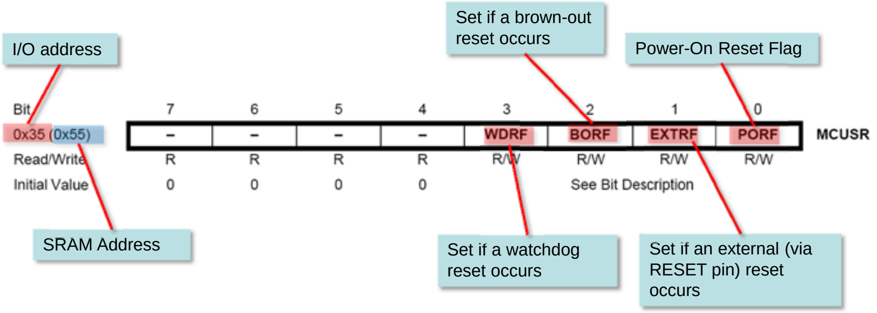
What is BOD? If the voltage in an embedded system drops below some level (brown-out condition), the microntroller may still run, but other circuits attached to the controller may not.

In a brown-out, it may be more reliable to reset the whole system.

A BOD or Brown-out Detector is on-chip circuitry that monitors the power supply voltage. When the BOD is enabled, and the supply VCC decreases to a value below the trigger level, the Brown-out Reset is immediately activated.

# MCUSR: MCU Status Register

The MCU Status Register **MCUSR** provides information on which reset source caused an MCU reset. It is located in I/O memory.



| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x35 (0x55) | – | – | – | – | WDRF | BORF | EXTRF | PORF | MCUSR |
| Read/Write | R | R | R | R | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | See Bit Description | | | | |

I/O address
Set if a brown-out reset occurs
Power-On Reset Flag
SRAM Address
Set if a watchdog reset occurs
Set if an external (via RESET pin) reset occurs

To make use of the Reset Flags to identify a reset condition, the user should read and then Reset the **MCUSR** as early as possible in the program. If the register is cleared before another reset occurs, the source of the reset can be found by examining the Reset Flags.

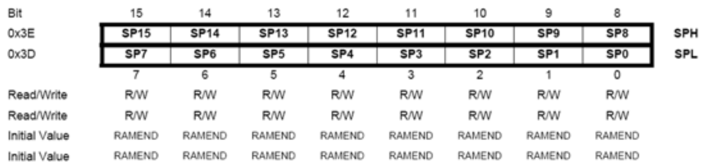
What is a “watchdog” ? - more later

## Stack Pointer or SP

AVRs have a stack, and it is mainly used for storing temporary data, for storing local variables and for storing return addresses after interrupts and subroutine calls.

On RISC machines, the stack is much less used than some other architectures.  This is because there are many registers where one can store temporary data.

The AVR Stack Pointer or **SP** is consists of two 8-bit registers in the **I/O space**, namely **SPH** and **SPL**.



| Bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x3E | SP15 | SP14 | SP13 | SP12 | SP11 | SP10 | SP9 | SP8 | SPH |
| 0x3D | SP7 | SP6 | SP5 | SP4 | SP3 | SP2 | SP1 | SP0 | SPL |
| | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | RAMEND | RAMEND | RAMEND | RAMEND | RAMEND | RAMEND | RAMEND | RAMEND | |
| Initial Value | RAMEND | RAMEND | RAMEND | RAMEND | RAMEND | RAMEND | RAMEND | RAMEND | |

**SP** on ATtiny45

The stack is implemented in SRAM, and **SP** always points to the top of the stack and decrements with a **push** instruction.

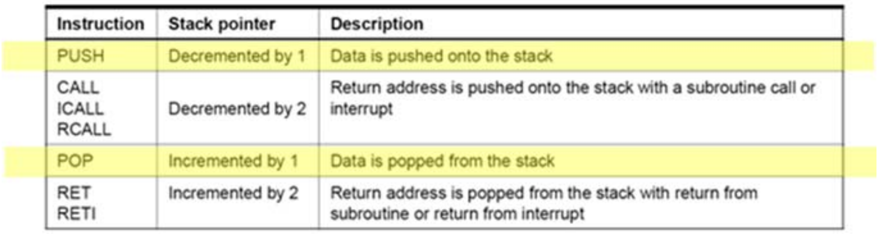
## Stack Pointer or SP

AVR processors automatically **push** the return address during a subroutine call (**CALL**, **RCALL**, …)

AVR processors automatically **pops** the return address as part of the return (**RET**) instruction

AVR processors automatically **push** the return address when an **interrupt** occurs

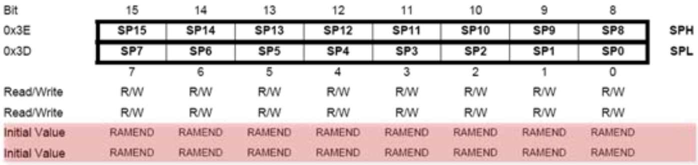
**Interrupt service routines** (ISPs) should return with an **RETI** instruction, which **pops** the return address



| Instruction | Stack pointer | Description |
|---|---|---|
| PUSH | Decremented by 1 | Data is pushed onto the stack |
| CALL<br>ICALL<br>RCALL | Decremented by 2 | Return address is pushed onto the stack with a subroutine call or interrupt |
| POP | Incremented by 1 | Data is popped from the stack |
| RET<br>RETI | Incremented by 2 | Return address is popped from the stack with return from subroutine or return from interrupt |

AVR Instructions that affect **SP**

## Stack Pointer or SP



| Bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x3E | SP15 | SP14 | SP13 | SP12 | SP11 | SP10 | SP9 | SP8 | SPH |
| 0x3D | SP7 | SP6 | SP5 | SP4 | SP3 | SP2 | SP1 | SP0 | SPL |
| | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | RAMEND | RAMEND | RAMEND | RAMEND | RAMEND | RAMEND | RAMEND | RAMEND | |
| Initial Value | RAMEND | RAMEND | RAMEND | RAMEND | RAMEND | RAMEND | RAMEND | RAMEND | |

**SP** on ATtiny45

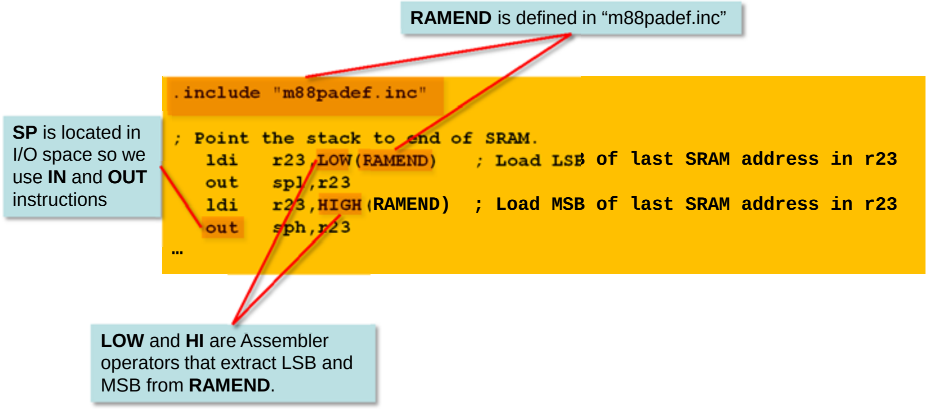
The stack is implemented in SRAM, and **SP** always points to the top of the stack and **decrements** with a **PUSH** instruction.

Thus, to utilize the stack using the **PUSH** and **POP** instructions, one must initialize **SP** properly.

On AVRs, **SP** is initialized to the end of SRAM (see diagram above)

Other AVR documentation suggest that at reset, **SP** is initialized to 0x00, which is where the 32 general purpose registers start—see the AVR memory map, so **programmer has to initialize SP.**

## Stack Pointer or SP



**RAMEND** is defined in “m88padef.inc”

```
.include "m88padef.inc"

; Point the stack to end of SRAM.
   ldi   r23,LOW(RAMEND)    ; Load LSB of last SRAM address in r23
   out   spl,r23
   ldi   r23,HIGH(RAMEND)   ; Load MSB of last SRAM address in r23
   out   sph,r23
…
```

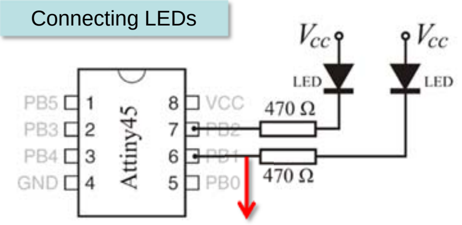
**SP** is located in I/O space so we use **IN** and **OUT** instructions

**LOW** and **HI** are Assembler operators that extract LSB and MSB from **RAMEND**.

# Recall Earlier Slide

**Example.** Turning LEDs on and off

Connecting LEDs



Note LEDs will light up when ports pins are pulled LOW

Data Direction Register

Configuring port pins for output

```
sbi    DDRB,1        ; PB1 is now output
sbi    DDRB,2        ; PB2 is now output
```

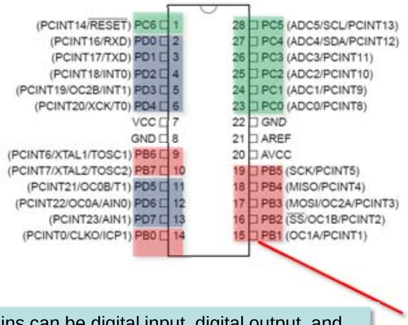
Turn LED at PB1 OFF

```
sbi    PORTB,1       ; LED at PB1 off
```

Turn LED at PB1 ON

```
cbi    PORTB,2       ; LED at PB2 on
```

# DDRB, PORTB, PINB,…



I/O ports on ATmega88P: **PORTB, PORTC**, and **PORTD**

Individual pins of the ports are **PB1**, **PD4**, etc.

Complex circuitry sits behind each port pin.

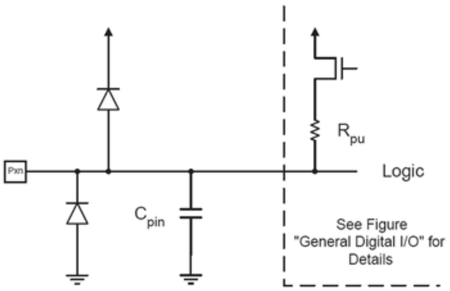
Pins can be digital input, digital output, and analog Input

Three registers located in I/O space

**DDRB** – Data Direction Register for PORTB

**PORTB**– Digital Output Register for PORTB

**PINB**– Input Register for PORTB



Simplified I/O equivalent schematic

## DDRB, PORTB, PINB,…

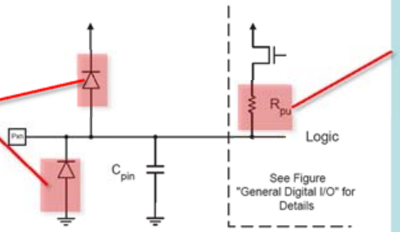
All AVR ports have **true Read-Modify-Write functionality** when used as general digital I/O ports (**SBI** and **CBI** instructions). Also, the **direction** of one port pin can be changed without unintentionally changing the direction of any other pin.

The pin driver is strong enough to drive LED displays directly. All port pins have individually selectable pull-up resistors with a supply-voltage invariant resistance.

Each output buffer has symmetrical drive characteristics with both high sink and source capability.

All I/O pins have protection diodes to both VCC and Ground

Protection Diodes

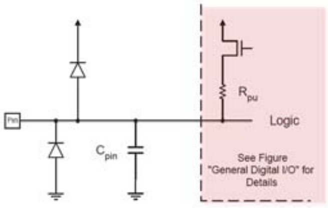


Pull-up resistor ensures that is at expected logic levels if external devices are disconnected.

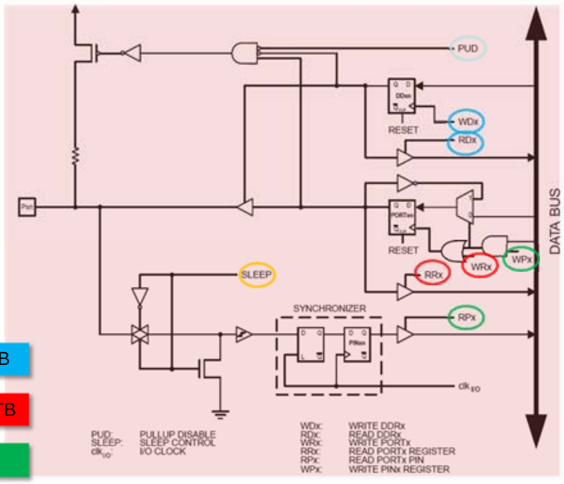
The idea is that is that it weakly "pulls" the pin to Vcc.

The resistor is high-resistance enough that, if something else strongly pulls the wire toward 0V, the pin will go to 0V.

## DDRB, PORTB, PINB,…



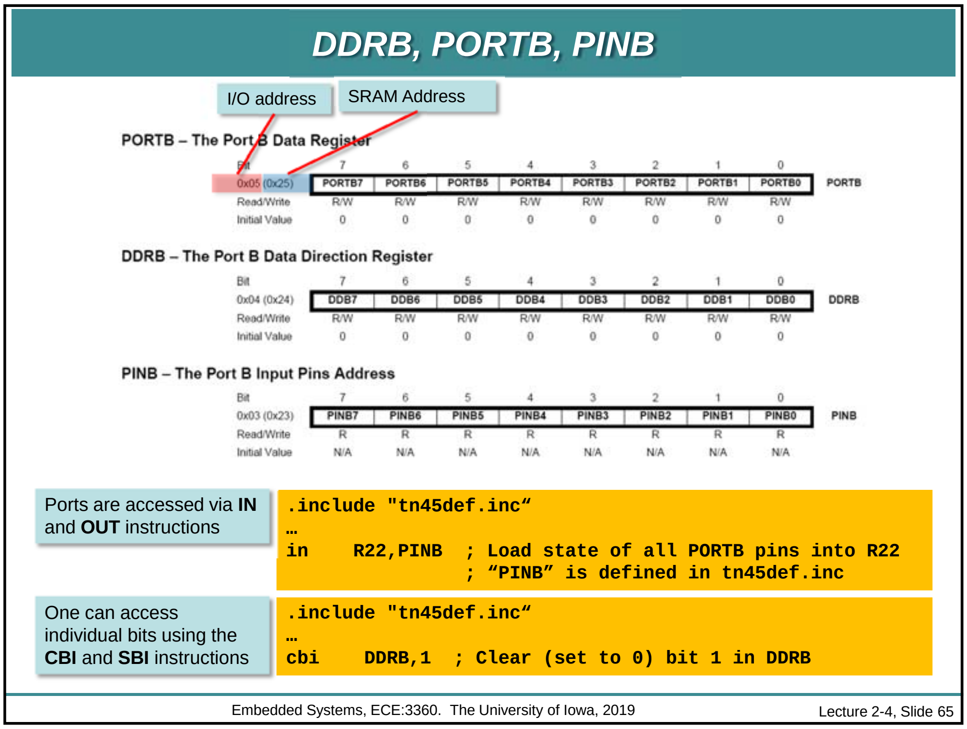
Configuring a PIN



**DDRB** – Data Direction Register for PORTB

**PORTB** – Digital Output Register for PORTB

**PINB** – Input Register for PORTB



# DDRB, PORTB, PINB

I/O address

SRAM Address

**PORTB – The Port B Data Register**

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x05 (0x25) | PORTB7 | PORTB6 | PORTB5 | PORTB4 | PORTB3 | PORTB2 | PORTB1 | PORTB0 | PORTB |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

**DDRB – The Port B Data Direction Register**

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x04 (0x24) | DDB7 | DDB6 | DDB5 | DDB4 | DDB3 | DDB2 | DDB1 | DDB0 | DDRB |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

**PINB – The Port B Input Pins Address**

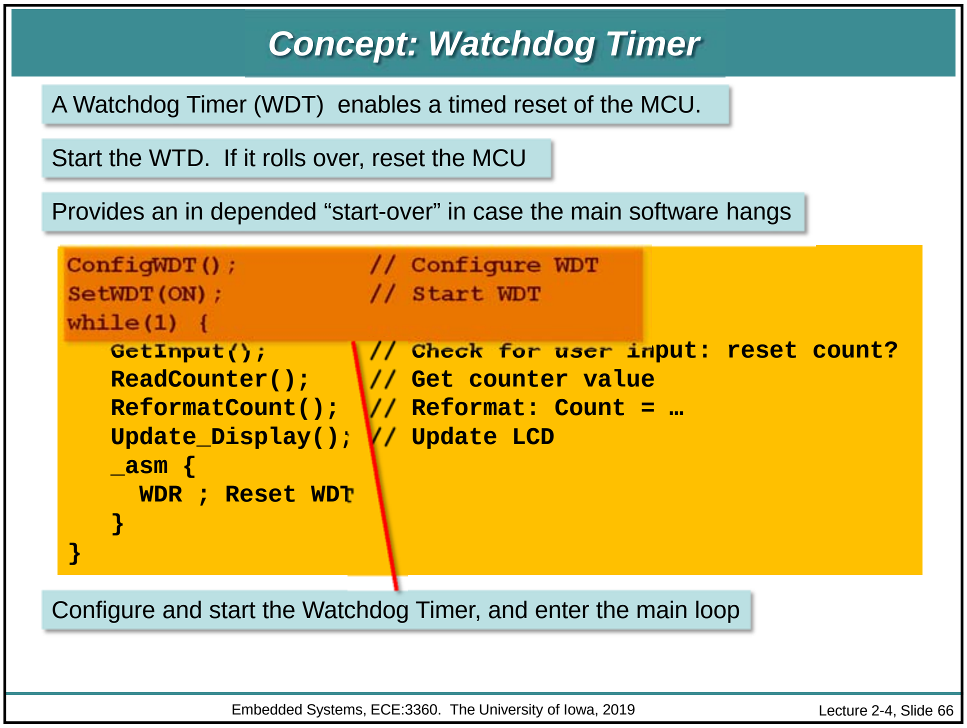
| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x03 (0x23) | PINB7 | PINB6 | PINB5 | PINB4 | PINB3 | PINB2 | PINB1 | PINB0 | PINB |
| Read/Write | R | R | R | R | R | R | R | R | |
| Initial Value | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | |

Ports are accessed via **IN** and **OUT** instructions

```
.include "tn45def.inc“
…
in    R22,PINB  ; Load state of all PORTB pins into R22
                ; “PINB” is defined in tn45def.inc
```

One can access individual bits using the **CBI** and **SBI** instructions

```
.include "tn45def.inc“
…
cbi    DDRB,1  ; Clear (set to 0) bit 1 in DDRB
```



# Concept: Watchdog Timer

A Watchdog Timer (WDT) enables a timed reset of the MCU.

Start the WTD. If it rolls over, reset the MCU

Provides an in depended “start-over” in case the main software hangs

```
ConfigWDT();            // Configure WDT
SetWDT(ON);             // Start WDT
while(1) {
   GetInput();          // Check for user input: reset count?
   ReadCounter();       // Get counter value
   ReformatCount();     // Reformat: Count = …
   Update_Display();    // Update LCD
   _asm {
     WDR ; Reset WDT
   }
}
```

Configure and start the Watchdog Timer, and enter the main loop

## Concept: Watchdog Timer

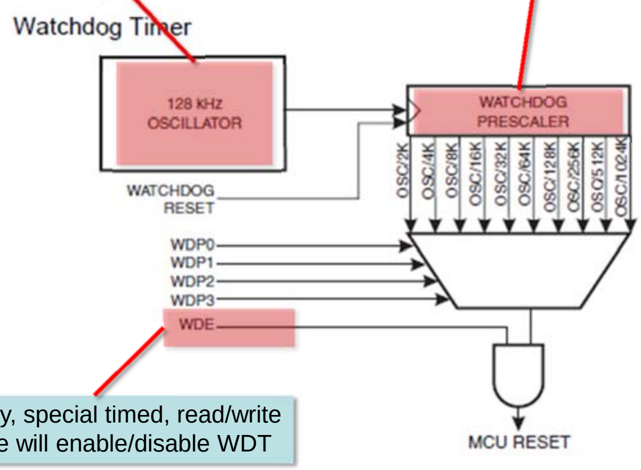
```
ConfigWDT();          // Configure WDT
SetWDT(ON);           // Start WDT
while(1) {
   GetInput();        // Check for user input: reset count?
   ReadCounter();     // Get counter value
   ReformatCount();   // Reformat: Count = …
   Update_Display();  // Update LCD
  _asm {
     WDR ; Reset WDT
   }
}
```

If anything in the main loop hangs the MCU, the MCU will time out and reset before this instruction is reached. If everything is OK, then the WDT will restart after this instruction.

## ATtiny45 Watchdog Timer

Independent, on-board WTD oscillator

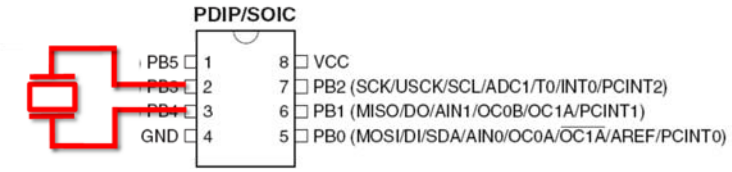
Presaler allows one to select various WDT time-out intervals



For safety, special timed, read/write sequence will enable/disable WDT

## Microcontroller Configuration/Fuses

AVR ATtiny45 ® 8-Pin Microcontroller

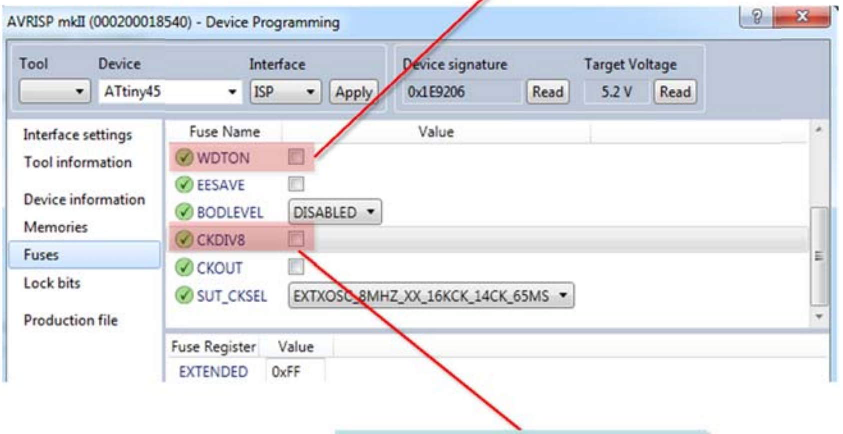

PDIP/SOIC
PB5 1
PB3 2
PB4 3
GND 4
8 VCC
7 PB2 (SCK/USCK/SCL/ADC1/T0/INT0/PCINT2)
6 PB1 (MISO/DO/AIN1/OC0B/OC1A/PCINT1)
5 PB0 (MOSI/DI/SDA/AIN0/OC0A/OC1A/AREF/PCINT0)

- Except for GND and VCC all other pins can perform at least 4 functions
- PB5 can be hardware RESET or an ADC input
- PB3 & PB4 can be ADC inputs or where crystal for external oscillator goes
- How does one configure controller for the external environment?

## Microcontroller Configuration/Fuses

- The Configuration Fuses (Configuration Bits) are the settings that configure a microcontroller for the external environment it is expecting to find
- Typical settings include
  - Oscillator Type, REST pin usage
  - Code Protection
  - Brown-Out and Watchdog Timer usage
  - Low Voltage Programming
  - …
- Different microcontrollers, and members within a device family → different fuses

Enables use of WDT



AVRISP mkII (000200018540) - Device Programming

| Tool | Device | Interface | | Device signature | | Target Voltage | |
|---|---|---|---|---|---|---|---|
| | ATtiny45 | ISP | Apply | 0x1E9206 | Read | 5.2 V | Read |

| Fuse Name | Value |
|---|---|
| WDTON | ☐ |
| EESAVE | ☐ |
| BODLEVEL | DISABLED |
| CKDIV8 | ☐ |
| CKOUT | ☐ |
| SUT_CKSEL | EXTXOSC_8MHZ_XX_16KCK_14CK_65MS |

| Fuse Register | Value |
|---|---|
| EXTENDED | 0xFF |

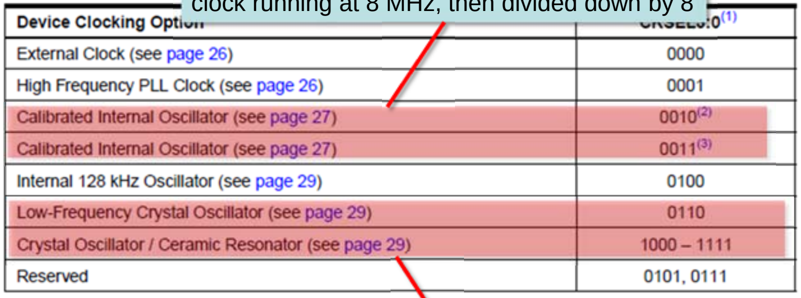
This chooses if the clock is divided by 8 or not, start-up time,…

## Clock Sources

ATtiny45 and ATmega88PA support a wide range of clock sources

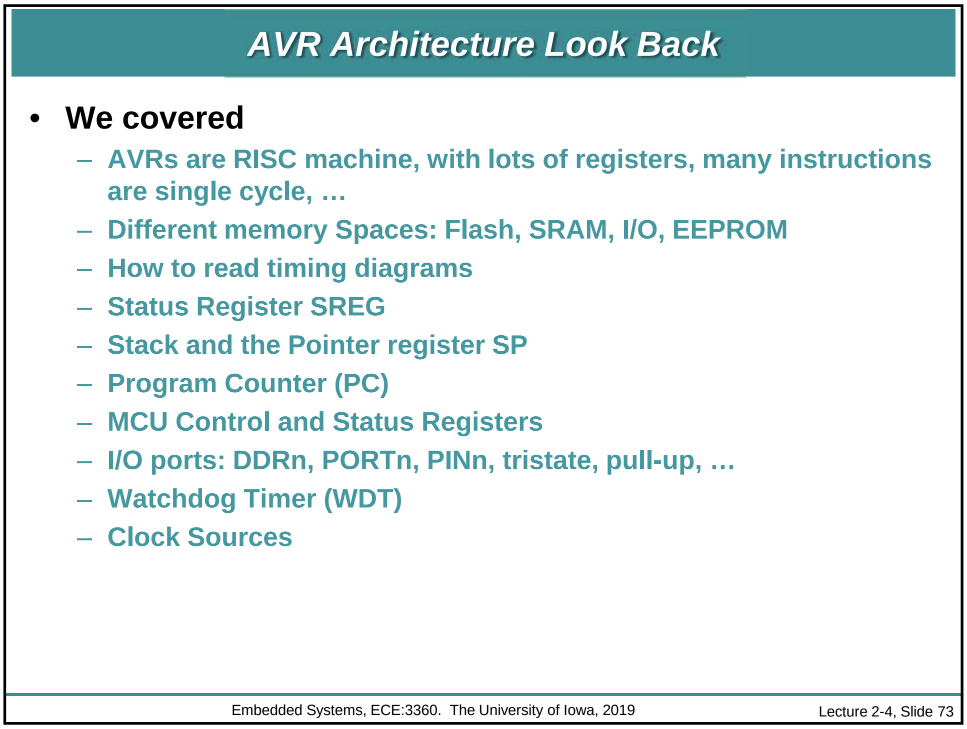
From the ATtiny45 documentation:

Factory default for ATtiny45 uses internal RC clock running at 8 MHz, then divided down by 8



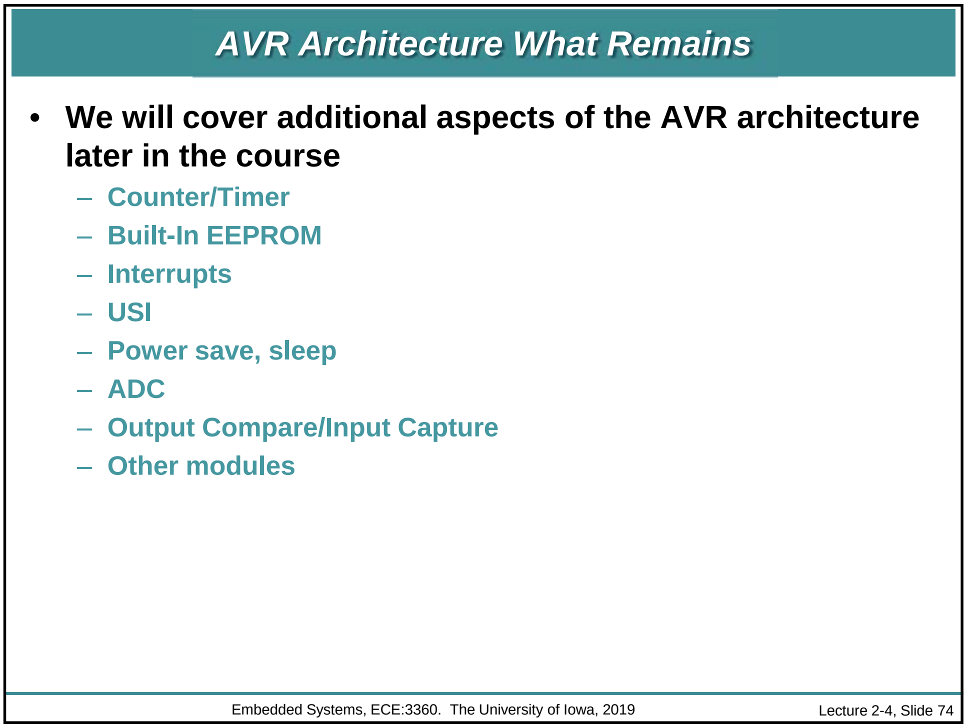
| Device Clocking Option | CKSEL3:0[1] |
|---|---|
| External Clock (see page 26) | 0000 |
| High Frequency PLL Clock (see page 26) | 0001 |
| Calibrated Internal Oscillator (see page 27) | 0010[2] |
| Calibrated Internal Oscillator (see page 27) | 0011[3] |
| Internal 128 kHz Oscillator (see page 29) | 0100 |
| Low-Frequency Crystal Oscillator (see page 29) | 0110 |
| Crystal Oscillator / Ceramic Resonator (see page 29) | 1000 – 1111 |
| Reserved | 0101, 0111 |

One must configure ATtiny45 to use external crystal (e.g., in labs we use 10 MHz crystal)



## AVR Architecture Look Back

- We covered
  - AVRs are RISC machine, with lots of registers, many instructions are single cycle, …
  - Different memory Spaces: Flash, SRAM, I/O, EEPROM
  - How to read timing diagrams
  - Status Register SREG
  - Stack and the Pointer register SP
  - Program Counter (PC)
  - MCU Control and Status Registers
  - I/O ports: DDRn, PORTn, PINn, tristate, pull-up, …
  - Watchdog Timer (WDT)
  - Clock Sources



## AVR Architecture What Remains

- We will cover additional aspects of the AVR architecture later in the course
  - Counter/Timer
  - Built-In EEPROM
  - Interrupts
  - USI
  - Power save, sleep
  - ADC
  - Output Compare/Input Capture
  - Other modules

**… EOL**