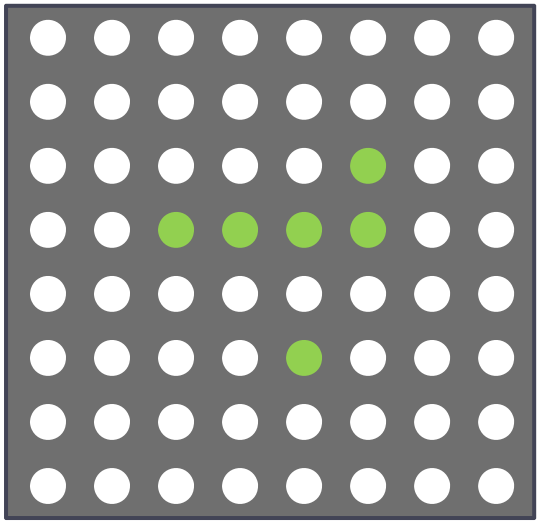
Lecture 6
AVR ATmega

Groups for assignment 2 (Snake)

- Groups of 3-4 students
- Send an e-mail to sanny.syberfeldt@his.se with group members' names
- You will be placed in group A or group B (see scio)



Lecture overview

- The ATmega 328p
- AVR assembly programming
- I/O on LedJoy

ATmega 328p overview

- 8-bit RISC-style processor
- 32 8-bit general-purpose processor registers
- 32K flash instruction memory
- 2K SRAM
- 16-bit instructions (some with 16-bit displacement)
- Built-in timers (2 8-bit timers, 1 16-bit timer)
- Built-in A/D conversion
- 23 programmable I/O ports

Processor registers

- 32 8-bit registers (**`r0`** – **`r31`**)
- Memory-mapped as addresses 0x0000 – 0x001F
- Possible to define your own names for the registers in Atmel Studio, e.g.,

```
.DEF    rRowData = r3
.DEF    rColBits = r4
.DEF    rJoyHigh = r5
.DEF    rJoyLow  = r6
```

- Make sure you don't accidentally give the same register two names!

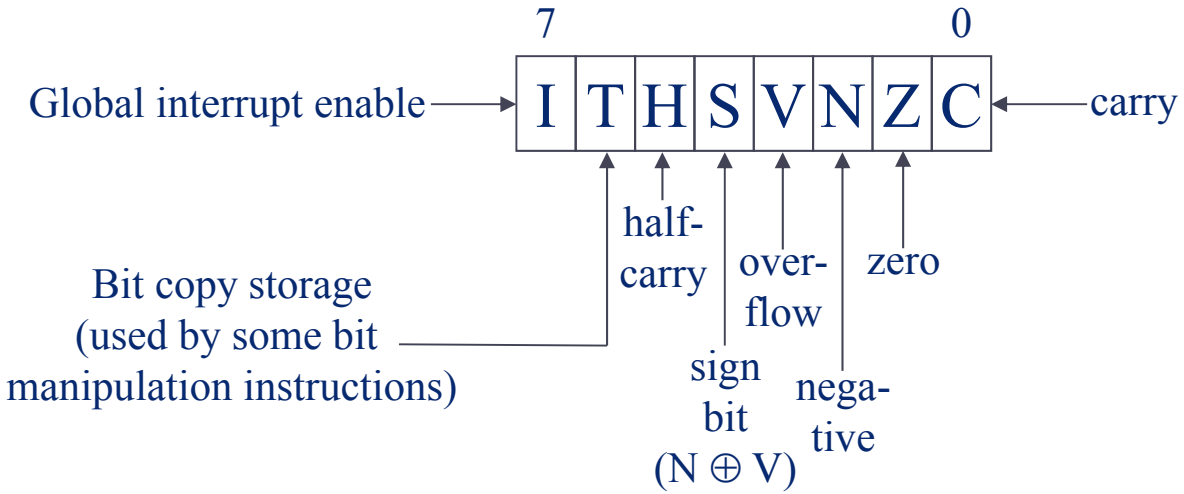
Register usage

- **`r0` – `r15`** cannot use immediate addressing
  - i.e., they **don't allow constant operands**

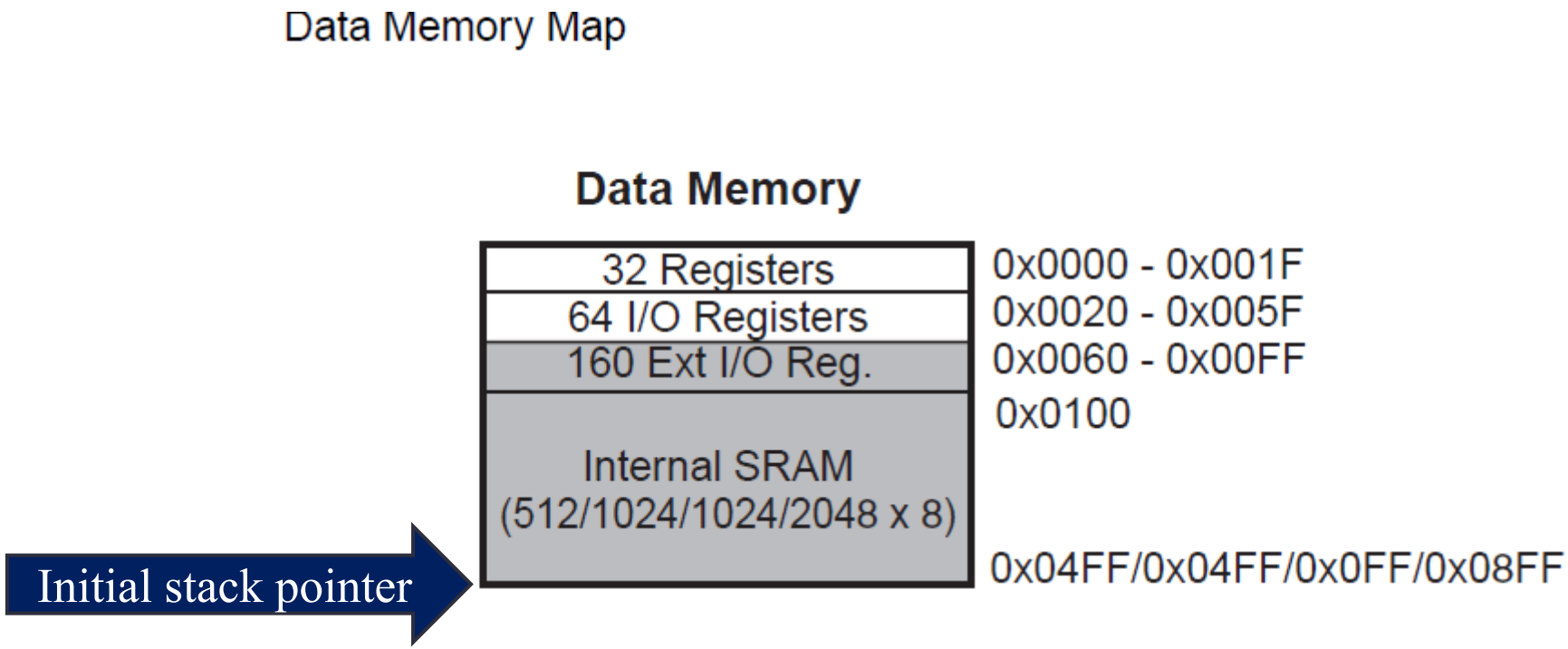
- Three pairs can be used as 16-bit **pointer registers**
  - **`r26`** + **`r27`** are the 16-bit **`X`** register
    - **`r26`** has the alias **`XL`** (**`X`**-register Lower byte)
    - **`r27`** has the alias **`XH`** (**`X`**-register Higher byte)
  - **`r28`** + **`r29`** are the **`Y`** register
  - **`r30`** + **`r31`** are the **`Z`** register
  - Only **`Y`** and **`Z`** allow displacement addressing

Status register (SREG)

- Details results of last instruction
  - Similar registers exist in many processors
- Used to detect e.g., arithmetic overflow
- Common use: to control conditional branches
  - Separate *compare* and *branch* instructions



SRAM structure



Stack pointer

- Made up of two memory registers SPH (Stack Pointer High byte) and SPL (Stack Pointer Low byte)
- Should be initialized before subroutine calls are made and interrupts are enabled
  - Set to highest memory position in SRAM:

```
ldi  rTemp, HIGH(RAMEND)
out  SPH, rTemp
ldi  rTemp, LOW(RAMEND)
out  SPL, rTemp
```

ALU instructions, examples

- **`add`** Rd, Rr Add Rr and Rd, result in Rd
  - E.g., **`add XL, rCurrentRow`**
- **`adiw`** Rdl, K Add constant K to word Rdl
  - E.g., **`adiw X, 1`** ($0 \leq K \leq 63$)
- **`neg`** Rd Negate Rd (two's complement)
- **`com`** Rd Invert Rd (one's complement)
- **`inc`** Rd Increase Rd by one

Immediate addition

- There is no immediate-value addition instruction (”**`addi`**”) for 8-bit registers
  - (**`adiw`** used for 16-bit register pairs)

- Use **`subi`** instead (with constant negated), e.g.,
  - **`subi  rTemp,-4 // add 4 to rTemp`**

Data transfer instructions, examples

- **`mov`** Rd, Rr Copy Rr to Rd
- **`ldi`** Rd, K Set Rd to constant K
- **`ld`** Rd, Y Load Rd from address in Y
- **`st`** Y, Rd Stor Rd contents at address in Y

Addressing modes

- Addressing modes for data memory
  - Direct `lds rTemp, TIMSK0`
  - Indirect with Displacement `ldd rTemp, Y+4`
    - Only for registers **`Y`** and **`Z`**
  - Indirect `ld rTemp, X`
  - Indirect with Pre-decrement `ld rTemp, -X`
  - Indirect with Post-increment `ld rTemp, X+`

- (Registers **`r26`** to **`r31`** pairwise are pointer registers **`X`**, **`Y`**, **`Z`**)
- Direct addressing reaches the entire data space

Bit manipulation instructions, examples

• **`lsl`**	Rd	Logical shift left of bits in Rd
• **`asr`**	Rd	Sign-extended shift right of Rd
• **`bst`**	Rd, K	Set status register T bit to bit K [0-7] in Rd
• **`bld`**	Rd, K	Set bit K [0-7] in Rd to status register T bit

Branching

- Unconditional jumps done with **`jmp`**
  - **`jmp`** lbl Jump to symbol lbl:

- Conditional branches done with **`cp`**/**`cpi`** followed by conditional branch instruction, e.g,.
  - **`brsh`** Branch if same or higher
  - **`breq`** Branch if equal
  - **`brne`** Branch if not equal
- Variant: **`cpse`** (compare, skip if equal)

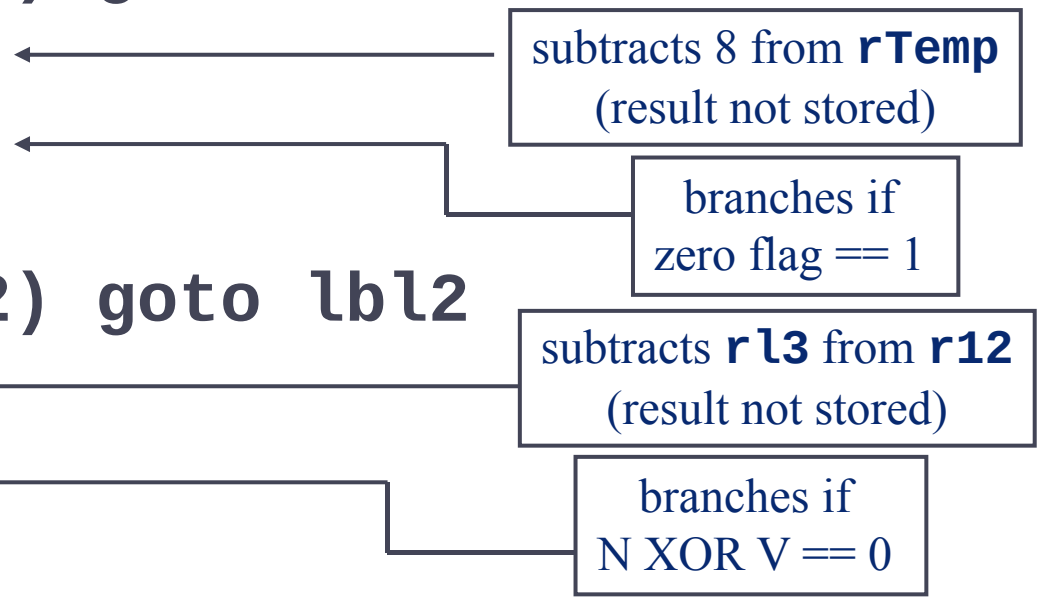
- Subroutine calls done with **`rcall`**
  - Return from subroutine with **`ret`**

More on conditional branches

- **`cp`** is actually a subtraction!
- Branch decision is made based on status register bits
  - E.g.,

```
; if(rTemp == 8) goto lbl1
cpi   rTemp,8
breq  lbl1

; if(r13 >= r12) goto lbl2
cp    r12,r13
brge  lbl2
```



Calling conventions

- Call-used registers (**`r18-r27`**, **`r30-r31`**)
  - ”Temporary” registers; can be used in subroutines
- Call-saved registers (**`r2-r17`**, **`r28-r29`**)
  - Subroutines should save and restore these
- **`r0`** and **`r1`** are assumed (by C compilers) to be *fixed registers*
  - **`r0`** - temporary register
  - **`r1`** - assumed to always be zero

Calling conventions, cont.

- Arguments allocated left to right, **`r25`** to **`r8`**
  - Aligned to start in even-numbered registers
  - Those that don't fit are passed on the stack

- Return values: 8-bit in **`r24`**, 16-bit in **`r25:r24`**
  - 8-bit return values sign-extended to 16 bits by caller

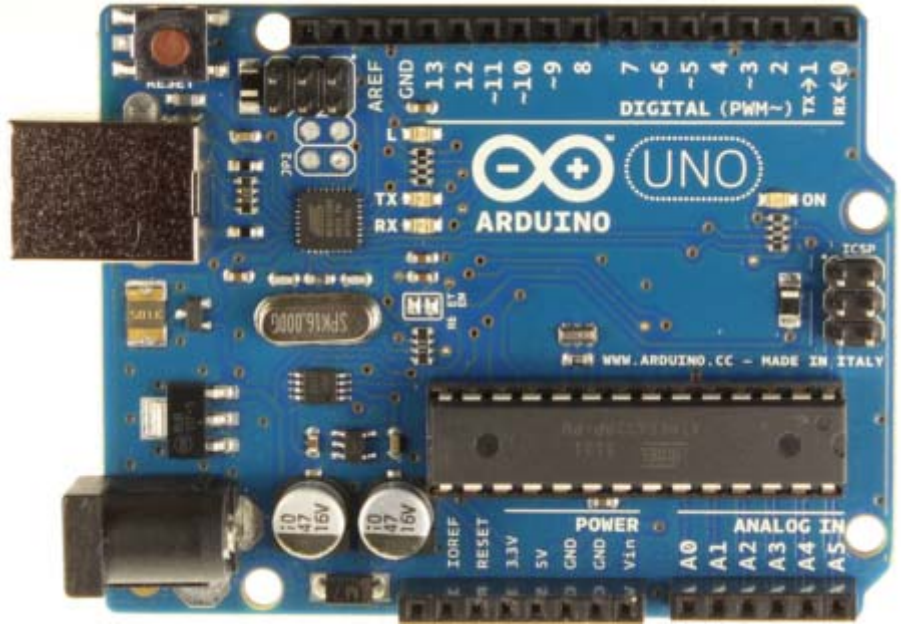
- Variable argument lists (**`printf`** etc.) passed on stack

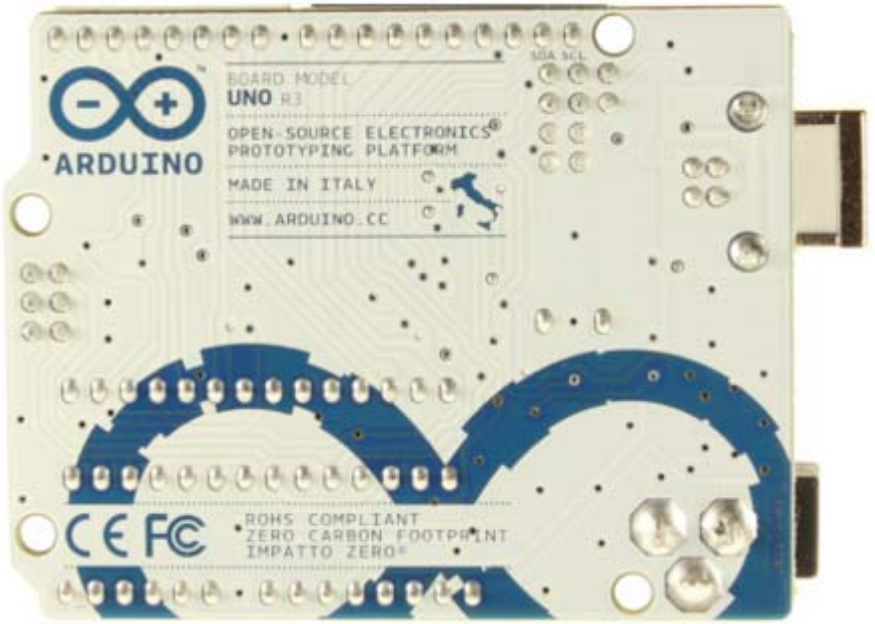
Stack manipulation

- Stack manipulation is done via push and pop
- **push** Rd — Decrease SP, put Rd on top of stack
- **pop** Rd — Set Rd to topmost stack element, increase SP
- Very useful when temporarily storing/restoring values
- E.g., in subroutines and interrupt handlers

Arduino Uno

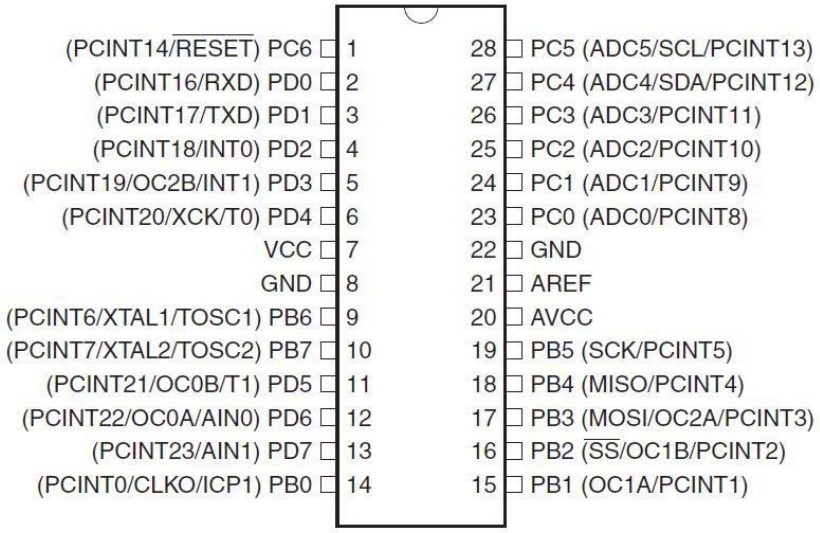
- Hobby development board based on ATmega 328p
- Powered by AC adapter or USB
- Programmable over USB
- 16MHz clock speed





I/O management

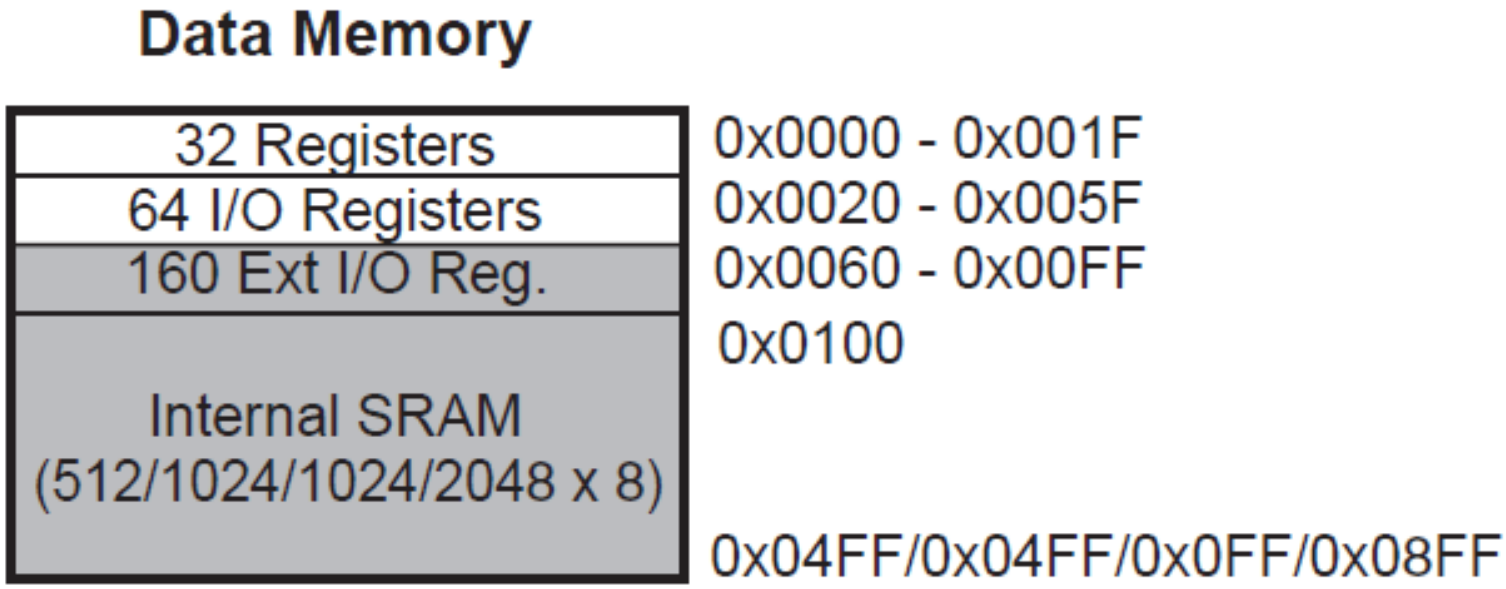
- The memory-mapped addresses that control I/O are called ports
  - Use aliases PORTB, PORTC, PORTD
  - Individual bits on these addresses are read or written to do input or output, respectively



Memory-mapped I/O

- Memory-mapped I/O is done via two types of instructions: **`in/out`** and **`sts/lds`**

Data Memory Map



Memory-mapped I/O, cont.

- The ATmega 328p supports more peripheral units via an **extended** I/O space (0x60 – 0xFF)
  - For example, Timer0 and A/D converter

- To access this space, use **`lds`** and **`sts`**, e.g,.

```
sts  ADCSRA, rTemp

lds  rTemp, TIMSK0
```

Data Direction Registers

- I/O pins on the ATmega must be configured as input or output pins
- Controlled by Data Direction Registers (DDRs)
- Each port has an associated DDR
  - PORTB is associated with DDRB
  - PORTC is associated with DDRC
  - PORTD is associated with DDRD
- To configure a pin as an input, set the associated bit in the DDR to 0; set it to 1 for output

Modifying individual bits

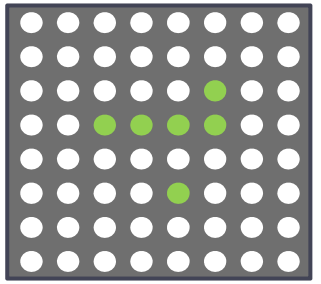
- When modifying bits in memory-mapped I/O registers, it is important to avoid change of other bits
- For primary I/O space, this is easy
  - Use the **`sbi`** and **`cbi`** instructions
- These do not work in the extended I/O space!
- Solution:
  - Read previous value
  - Modify individual bits through masking
  - Write back modified value

Modifying extended I/O space, example

```
/* Set bit TOIE0 in
   in TIMSK0 to 1. */

ldi  rTemp, (1<<TOIE0)
lds  rTemp2, TIMSK0
or   rTemp, rTemp2
sts  TIMSK0, rTemp
```

LED matrix updates



- 8 columns and 8 rows
- Each column and row are mapped to one output pin on the ATmega (mapping shown on next slide)
- To light a LED, activate the row pin and the column pin
  - → the entire display can not be active at once!
  - Solution: Activate rows one by one, changing the column bits
  - Done properly, changes are too quick for the eye to notice
  - It is tricky to ensure that rows are lit an equal amount of time

LED matrix: I/O port mapping

LedJoy	ATmega pin	Arduino-funktion
LED-display Rad 0	PORTC0 (PC0)	Analog input 0 (används som digital)
LED-display Rad 1	PORTC1 (PC1)	Analog input 1 (används som digital)
LED-display Rad 2	PORTC2 (PC2)	Analog input 2 (används som digital)
LED-display Rad 3	PORTC3 (PC3)	Analog input 3 (används som digital)
LED-display Rad 4	PORTD2 (PD2)	Digital 2
LED-display Rad 5	PORTD3 (PD3)	Digital 3
LED-display Rad 6	PORTD4 (PD4)	Digital 4
LED-display Rad 7	PORTD5 (PD5)	Digital 5
LED-display Kolumn 0	PORTD6 (PD6)	Digital 6
LED-display Kolumn 1	PORTD7 (PD7)	Digital 7
LED-display Kolumn 2	PORTB0 (PB0)	Digital 8
LED-display Kolumn 3	PORTB1 (PB1)	Digital 9
LED-display Kolumn 4	PORTB2 (PB2)	Digital 10
LED-display Kolumn 5	PORTB3 (PB3)	Digital 11
LED-display Kolumn 6	PORTB4 (PB4)	Digital 12
LED-display Kolumn 7	PORTB5 (PB5)	Digital 13
Joystick Y-axel	PORTC4 (PC4)	Analog input 4
Joystick X-axel	PORTC5 (PC5)	Analog input 5

LED matrix: suggested progression

- Getting the display to work properly is tricky
  - Don’t do it all at once!
- Suggested progression:

1) Light the first LED on the first row
2) Light the entire first row
3) Light the entire display
4) Light different LEDs on different rows
5) Represent the matrix in memory as described on the next slide

LED matrix update: overview

- Allocate a memory region to the matrix data
  - Suggestion: 8 bytes, each bit representing a LED
    - 1: lit, 0: unlit
- To change the matrix, write to this region
- When updating the display, iterate over the rows:

1) Deactivate current row
2) Read byte for next row from matrix memory region
3) Activate column bits based on the matrix data
4) Activate next row

A/D conversion

- Joystick X- and Y-axes are connected to analogue input pins
- ATmega's built-in A/D converter must be used
  - Translates the analogue input to a 10-bit value (0-1023)
  - Neutral position gives value of ~512 on each axis
    - (May fluctuate due to noise on the input lines)
- A/D converter is controlled by two I/O registers: ADMUX and ADCSRA (see next slide)
- Result in ADCH (upper 2 bits) and ADCL (lower 8 bits)
  - ADCL must be read first, then ADCH

ADMUX and ADCSRA

- ADMUX
  - Bits 0-3: Choose analogue input pin
  - Bit 5: Choose between 8-bit and 10-bit mode
  - Bits 6-7: Choose reference source
- ADCSRA
  - Bits 0-2: Clock pre-scaling (rate of conversion)
  - Bit 6: Start conversion / signal when finished
  - Bit 7: Activate the A/D converter

Suggested procedure

- Initialize the A/D converter once in an initialization routine
  - Set pre-scaling, set reference source, activate the converter

- Do conversion at regular intervals
  - Either once per refreshed row, or once per finished matrix update, or based on timer
  - See LedJoy programming guide for details

Timers

- To make the game update smoothly, a timer is needed
  - Should update independently of program execution
- ATmega 328p has several timers (8-bit and 16-bit)
  - Timer0 is a good choice for the assignment
- Timer0
  - 8-bit timer (0-255)
  - Controlled by TCCR0B and TIMSK0 I/O registers
  - Increases by one every clock cycle
    - Can be *pre-scaled* to increase at slower intervals
  - Can signal when overflowing from 255 to 0 using an *interrupt*

Interrupts

- Interrupts are events that break normal program flow
  - Details will be covered in next lecture

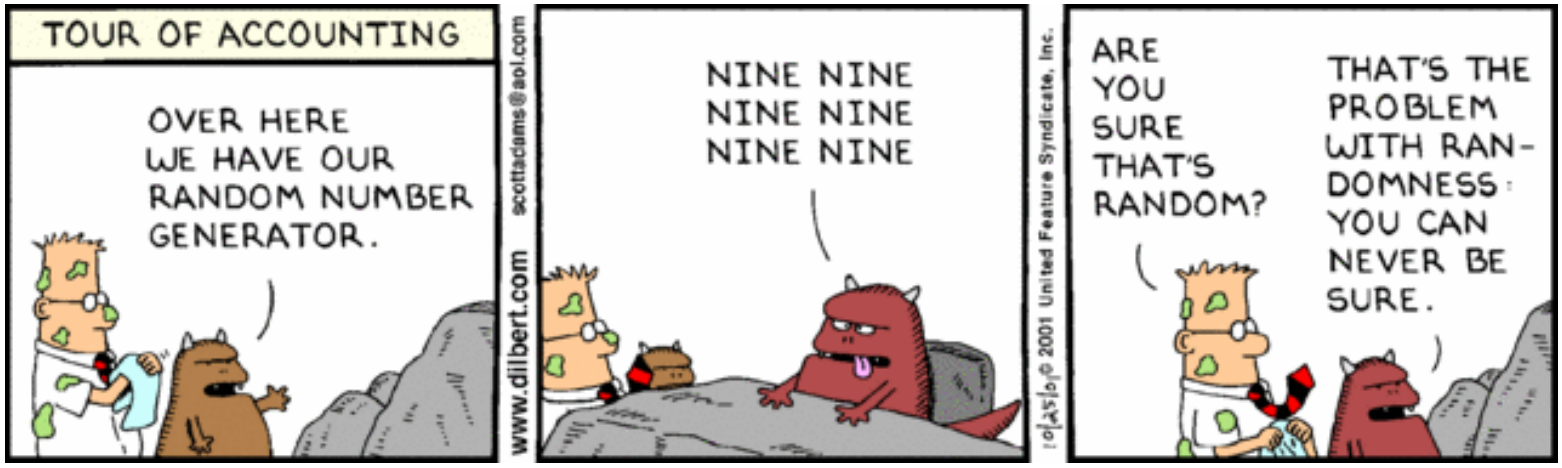
- Timer0 can be configured to cause an interrupt when overflowing
  - This transfers control to an *interrupt handler*
  - The handler can tick up a counter, flag for a game update if a limit is reached

Interrupt handling

- Interrupt handler: a subroutine that can be called at any point in the program
- ”Interrupt Vector Table” at the top of SRAM
- Specific interrupts jump to specific addresses
  - Instructions at these addresses should branch to a user-specified interrupt handler
- Handler must not permanently change any registers!
- Return address is kept on the stack
- An interrupt handler returns to the regular program flow with the `reti` instruction

Random numbers

- Random numbers are actually very tricky to generate
- ATmega has no built-in random number tables
- We need to find a source of randomness
- Suggestion: use noise on the analogue input pins



Random numbers, suggested procedure

- Test the analog inputs to determine which axis fluctuates most
- Use a random number variable
- Add each reading of the chosen analogue input to the variable
  - Allowed to overflow freely
- When generating a random number:
  - Do another reading of the analogue line
  - Run the random number variable through a non-linear function
  - Use the last X bits (e.g., last three bits for a value 0-7)

Example program structure

```
// Register definitions
.DEF rTemp = r16
…

// Constant definitions
.EQU NUM_COLUMNS = 8
…

// Data segment
.DSEG
matrix:   .BYTE 8
…
```

Example program structure, cont.

```
// Code segment
.CSEG
// Interrupt vectors
.ORG 0x0000
     jmp  init_routine   // Reset vector
…
.ORG INT_VECTORS_SIZE // End of vector table
init_routine:
…
```