Microprocessor Systems

Lecture by
Piotr M. Szczypinski

Lecturer

Piotr M. Szczypiński, Dr inż.

Instytut Elektroniki Politechniki Lodzkiej
Wolczanska 223, 90-924 Lodz, Poland
Office: 205
http: http://www.eletel.p.lodz.pl/~pms/
email: pms@p.lodz.pl
tel. +4842 631 2638

Lecture scope

- Introduction to AVR IDE
  WinAVR, AVR Studio, MegaLoad
- Historical background
- Digital Electronics
  What are gates, three-state logic, flip-flops, latches
- Microprocessor
  It's functional blocks, buses, machine code vs. assembler
- Memory
  RAM vs. ROM, Dynamic vs. Static, address decoder
- Microprocessor, memory, I/O devices
  How do they communicate?
- Binary data representation
  Integers, floating point numbers, alphanumerics, graphics
- Programming
  Algorithm, languages, compilation, linking, debugging
- Programmable ICs

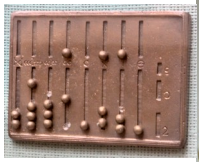
Sources of information

- This lecture: http://www.eletel.p.lodz.pl/~pms/dyda_en.html
- H. Feichtinger, Mikrokomputery – poradnik, WKŁ
- P. Misiurewicz, Podstawy techniki mikroprocesorowej, WNT
- R. Baranowski, Mikrokontrolery AVR ATmega w praktyce, BTC
- J. Doliński, Mikrokontrolery AVR w praktyce, BTC
- T. Łuba, K. Jasiński, B. Zbierzchowski,
  Specjalizowane układy cyfrowe w strukturach PLD i FPGA, WKŁ
- Standard Digital ICs
  - http://www.standardics.philips.com/products/
- AT Mega:
  - http://www.atmel.com/
  - http://www.avrfreaks.net/
  - http://sourceforge.net/projects/winavr/
  - http://www.microsyl.com/megaload/megaload.html
- Programmable ICs
  - http://www.altera.com/html/literature/
  - http://www.latticesemi.com/products/
  - http://www.xilinx.com/partinfo/databook.htm

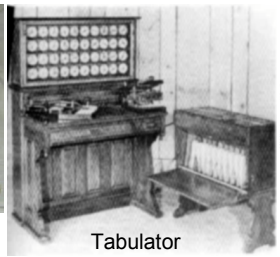
Milestones

Historical Background

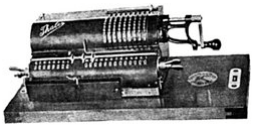
- Abacus - invented 3000 BC in Babylon
- Mechanical arithmometer - independently invented by Wilhem Schnickard in 1623 and by Blaise Pascal in 1652
- Binary system - mathematical theory of binary system, Gottfried Wilhelm Leibnitz in 1674
- Tabulator - a machine for data storage with perforated cards, invented by Herman Hollerith in 1880, used during the census in USA
- Alan Turing Machine - definition of an algorithm
- Relay based computer - 1935 by Conrad Zuse




Roman and Russian Aacus



Tabulator



Arithmometer

(Wikipedia)

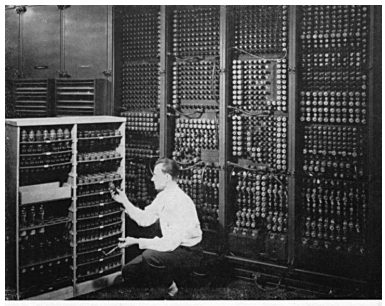
Computer Generations

The I generation of computers

- Eniac - computer composed of 1800 electron tubes, developed in 1946 by US Army researchers (occupied 150m^3, energy consumption 50kW, about 10000 simple operations per second)

The II generation of computers

- In 1956 MIT researchers designed and built the first computer composed of individual transistors



Replacing a bad tube meant checking among ENIAC's 19,000 possibilities.

(Wikipedia)

The III generation of computers

- In the late 60s digital integrated circuits (ICs) were developed (containing several logical gates per one IC). Computers made of such ICs belong to the third generation of computers.

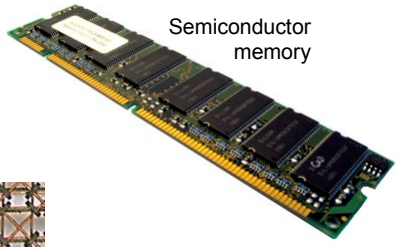
The IV generation of computers

- Very Large Scale of Integration (VLSI) ICs applied.

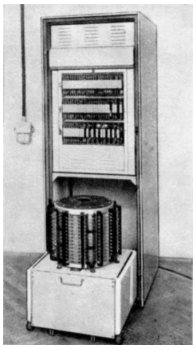
Memories

Mechanical - punch cards
Electromechanical - relay based
Ferrite-core memory
Wire memory
Drum memory
...

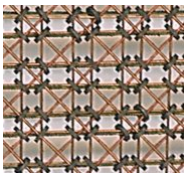
Semiconductor memory



Semiconductor memory



drum memory



ferrite-core memory

(Images: Wikipedia)

Micro...

Processor
a module of computer executing a program, performing computations

Microprocessor
a processor fit in a single IC

Microcomputer
a computer containing a microprocessor

(Wikipedia)

Microprocessors

Microprocessor

- 4-bit microprocessor - 1970, TMS1000 from Texas Instruments
- 8-bit microprocessors - Intel 8008 in 1972, Intel 8080 in 1974, Motorola 6800 ...
- 16-bit... 32-bit... 64-bit microprocessors...
- ...

Personal Computers

Personal computer

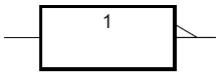
affordable (<$1000) + designed for a home or office use

- Mark 8 (do-it-yourself kit) 1974,
- Altair (do-it-yourself kit), 400$, Intel 8080, 1975,
- Apple, 900$, processor Motorola 6502, 1976,
- IBM PC, 1981,
- Apple, MacIntosh, 1983,
- ...

Digital Logic

Logical Gates

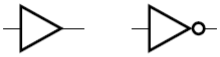
Inverter



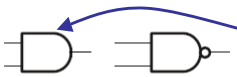
		In 2	
		0	1
In 1	0	0	1
	1	1	1

?

Baffer Inverter

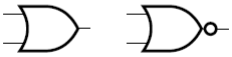


AND NAND



		In 2	
		0	1
In 1	0	0	0
	1	0	1

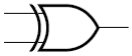
OR NOR

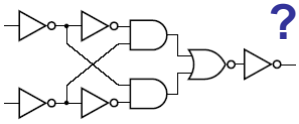


		In 2	
		0	1
In 1	0	0	1
	1	1	0

?

EXOR



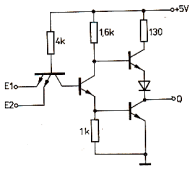


?

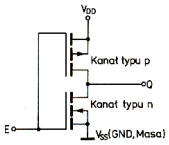
Connect tables and schematic with appropriate gate sybmols

Logical Gates

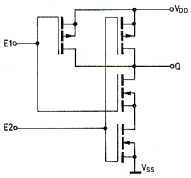
Technology



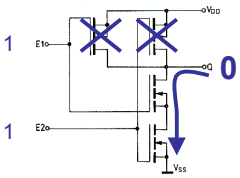
TTL - NAND

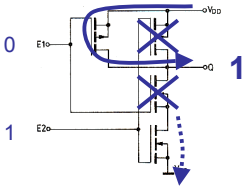


CMOS - Inverter



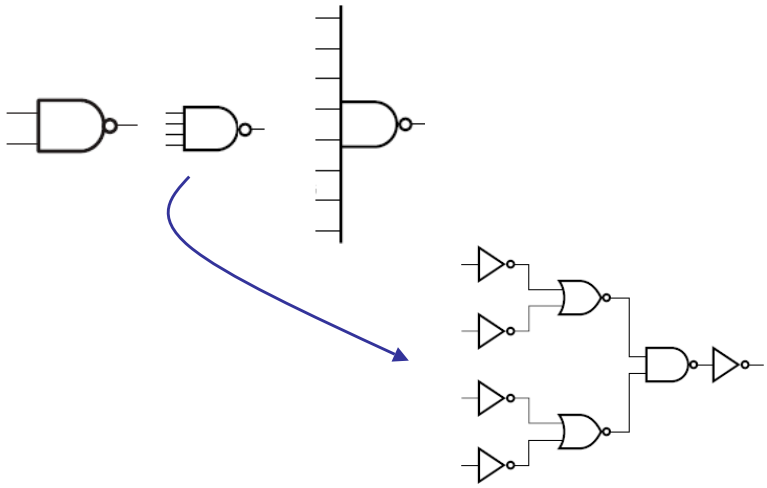
CMOS - NAND





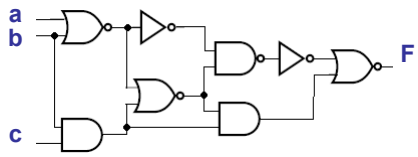
Logical Gates

Multiinput logical gates



Logic functions

Schematic



Function

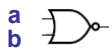
$$F = \overline{\left(\left(\left(b \bullet c\right) \bullet \overline{\overline{a+b} + \left(b \bullet c\right)}\right) + \overline{\overline{\overline{a+b}} \bullet \overline{\left(\overline{a+b} + \left(b \bullet c\right)\right)}}\right)}$$

F=~(((b&c)&(~((~(a|b))|(b&c)))) | ~~((~~(a|b))&~(~(a|b)|(b&c))))

Karnaugh Map

	c	0	1
a b			
0 0		1	1
0 1		0	0
1 1		0	0
1 0		0	0

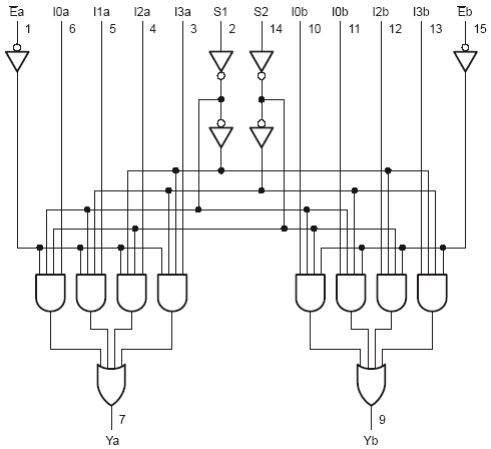
Simplified function and schematic



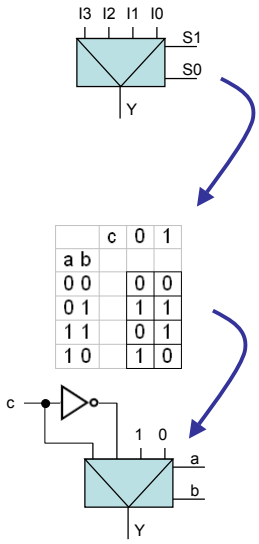
$$F = \overline{a+b}$$

F=~(a|b)

Multiplexers



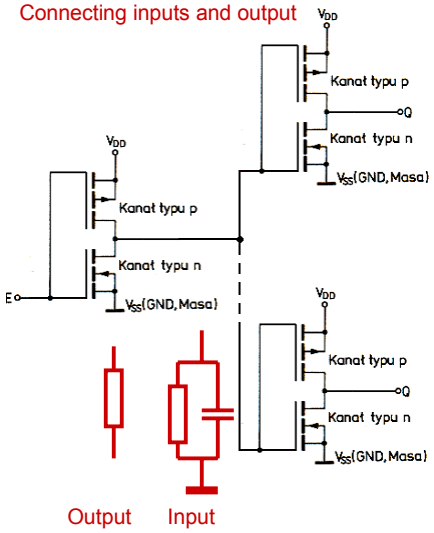
Dual 4-line to 1-line multiplexer with 2 common select inputs
74f153 - http://www.standardics.philips.com/products/



	c	0	1
a b			
0 0		0	0
0 1		1	1
1 1		0	1
1 0		1	0

Logical Gates

Connecting inputs and output

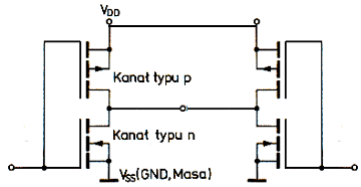


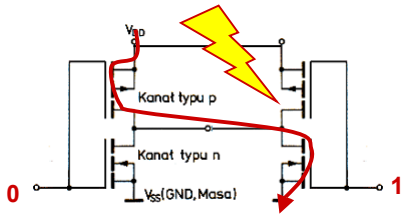
- Find out what are the standard CMOS gates:
  - rise/fall times
  - delays
  - input/output impedances

E.g. see: http://www.standardics.philips.com/products/

Logical Gates

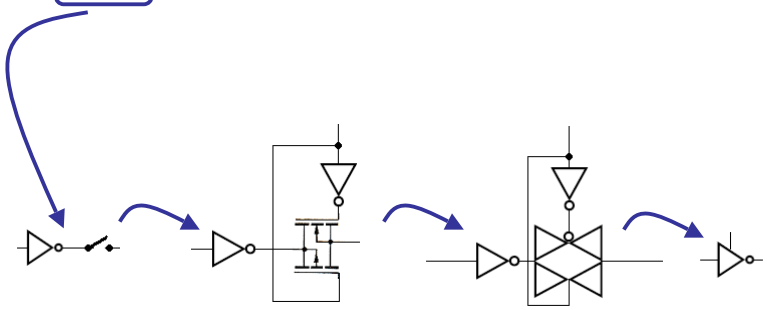
Connecting outputs together





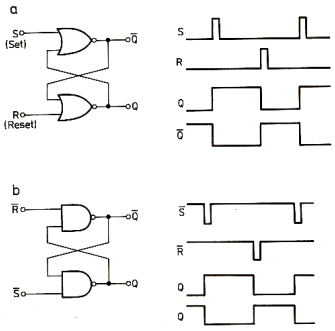
Logical Gates

Three-state logic

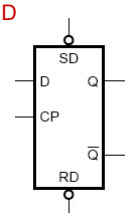


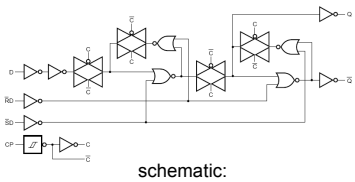
Latches and Flip-flops

RS



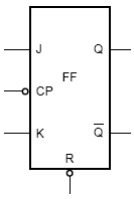
D





schematic:
http://www.standardics.philips.com/products/

JK



- How D and JK Flip-flop work?
- What is Master-Slave Flip-flop?
- Does the schematic show M-S Flip-flop?
- What are synchronic circuits?

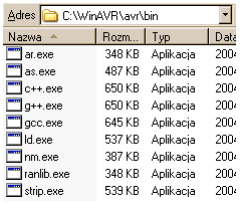
Introduction to AVR IDE

WinAVR + AVR Studio + MegaLoad

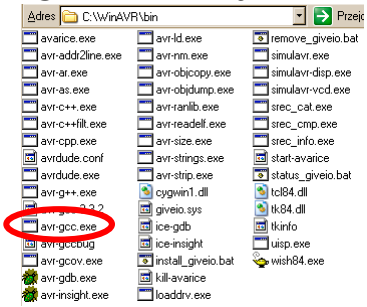
WinAVR

Desktop

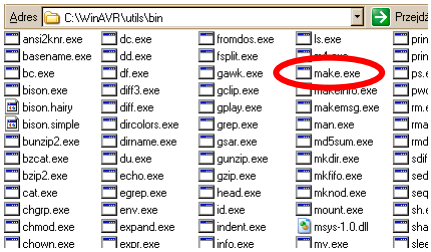
C:\WinAVR\avr\bin



C:\WinAVR\bin

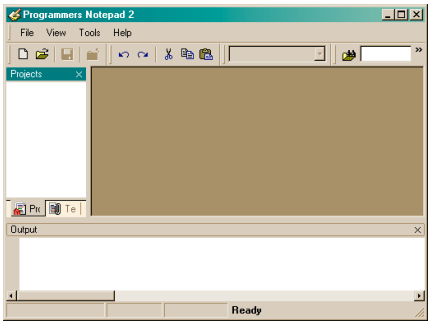


C:\WinAVR\utils\bin



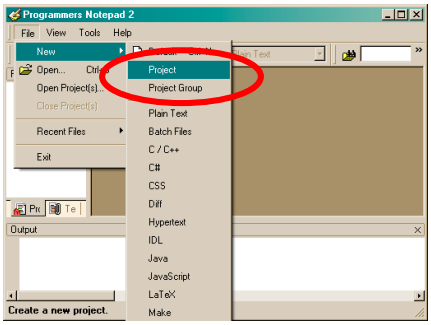
WinAVR: New Project

1. Start Programmers Notepad (PN)
2. Select File->NewProject
3. Create new folder
4. Create new project file
5. Create new C file
6. Save it into a created folder
7. Start MFile
8. Set main file name (created C file)
9. Set MCU to ATMega 128
10. Set optimization level to 0
11. Set Default target to Extended COFF
11. Save makefile into a created folder
12. In PN, add files into the project



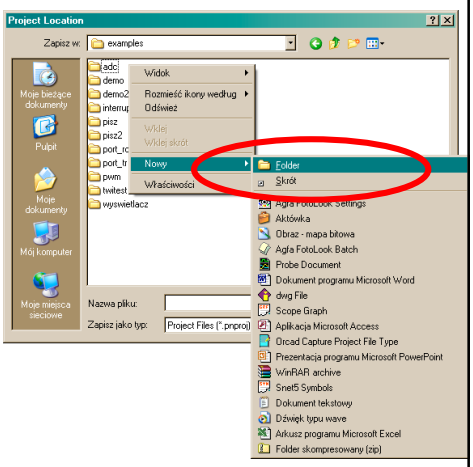
WinAVR: New Project

1. Start Programmers Notepad (PN)
2. Select File->NewProject
3. Create new folder
4. Create new project file
5. Create new C file
6. Save it into a created folder
7. Start MFile
8. Set main file name (created C file)
9. Set MCU to ATMega 128
10. Set optimization level to 0
11. Set Default target to Extended COFF
11. Save makefile into a created folder
12. In PN, add files into the project



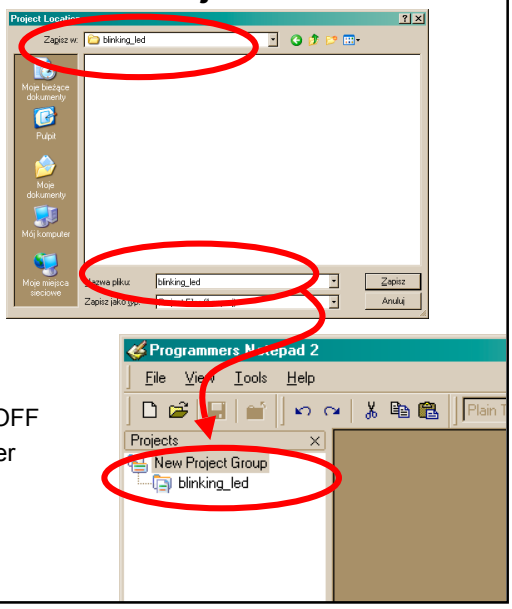
WinAVR: New Project

1. Start Programmers Notepad (PN)
2. Select File->NewProject
3. Create new folder
4. Create new project file
5. Create new C file
6. Save it into a created folder
7. Start MFile
8. Set main file name (created C file)
9. Set MCU to ATMega 128
10. Set optimization level to 0
11. Set Default target to Extended COFF
11. Save makefile into a created folder
12. In PN, add files into the project



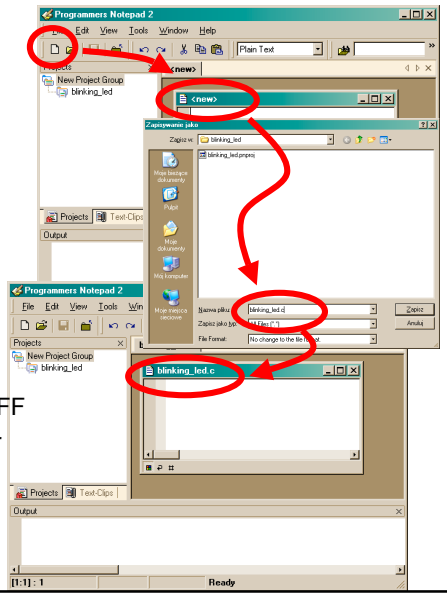
WinAVR: New Project

1. Start Programmers Notepad (PN)
2. Select File->NewProject
3. Create new folder
4. Create new project file
5. Create new C file
6. Save it into a created folder
7. Start MFile
8. Set main file name (created C file)
9. Set MCU to ATMega 128
10. Set optimization level to 0
11. Set Default target to Extended COFF
11. Save makefile into a created folder
12. In PN, add files into the project



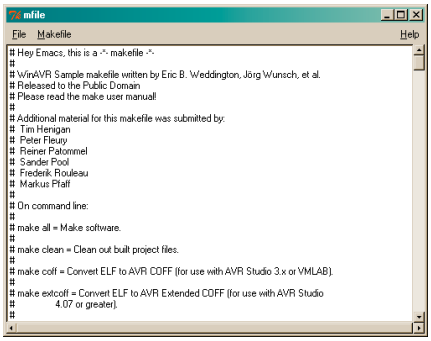
WinAVR: New Project

1. Start Programmers Notepad (PN)
2. Select File->NewProject
3. Create new folder
4. Create new project file
5. Create new C file
6. Save it into a created folder
7. Start MFile
8. Set main file name (created C file)
9. Set MCU to ATMega 128
10. Set optimization level to 0
11. Set Default target to Extended COFF
11. Save makefile into a created folder
12. In PN, add files into the project



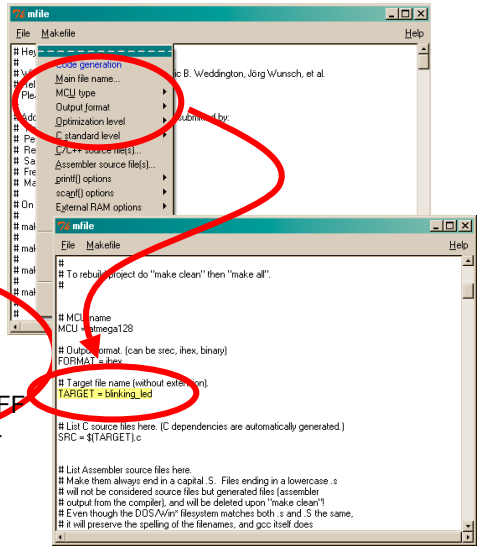
WinAVR: New Project

1. Start Programmers Notepad (PN)
2. Select File->NewProject
3. Create new folder
4. Create new project file
5. Create new C file
6. Save it into a created folder
7. Start MFile
8. Set main file name (created C file)
9. Set MCU to ATMega 128
10. Set optimization level to 0
11. Set Default target to Extended COFF
11. Save makefile into a created folder
12. In PN, add files into the project



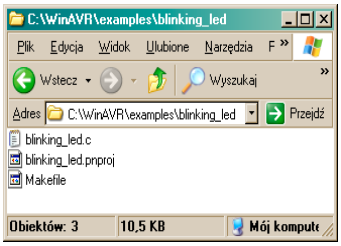
WinAVR: New Project

1. Start Programmers Notepad (PN)
2. Select File->NewProject
3. Create new folder
4. Create new project file
5. Create new C file
6. Save it into a created folder
7. Start MFile
8. Set main file name (created C file)
9. Set MCU to ATMega 128
10. Set optimization level to 0
11. Set Default target to Extended COFF
11. Save makefile into a created folder
12. In PN, add files into the project



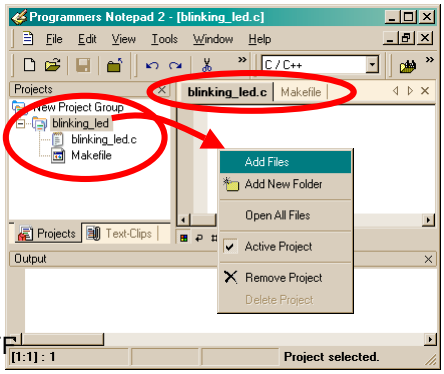
WinAVR: New Project

1. Start Programmers Notepad (PN)
2. Select File->NewProject
3. Create new folder
4. Create new project file
5. Create new C file
6. Save it into a created folder
7. Start MFile
8. Set main file name (created C file)
9. Set MCU to ATMega 128
10. Set optimization level to 0
11. Set Default target to Extended COFF
11. Save makefile into a created folder
12. In PN, add files into the project

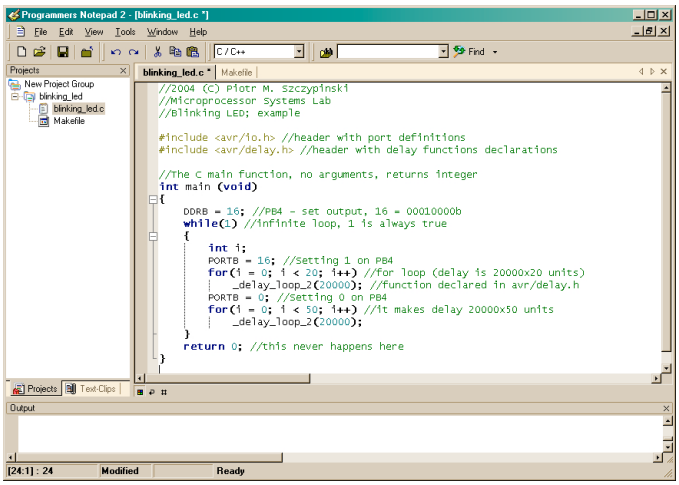


WinAVR: New Project

1. Start Programmers Notepad (PN)
2. Select File->NewProject
3. Create new folder
4. Create new project file
5. Create new C file
6. Save it into a created folder
7. Start MFile
8. Set main file name (created C file)
9. Set MCU to ATMega 128
10. Set optimization level to 0
11. Set Default target to Extended COFF
11. Save makefile into a created folder
12. In PN, add files into the project

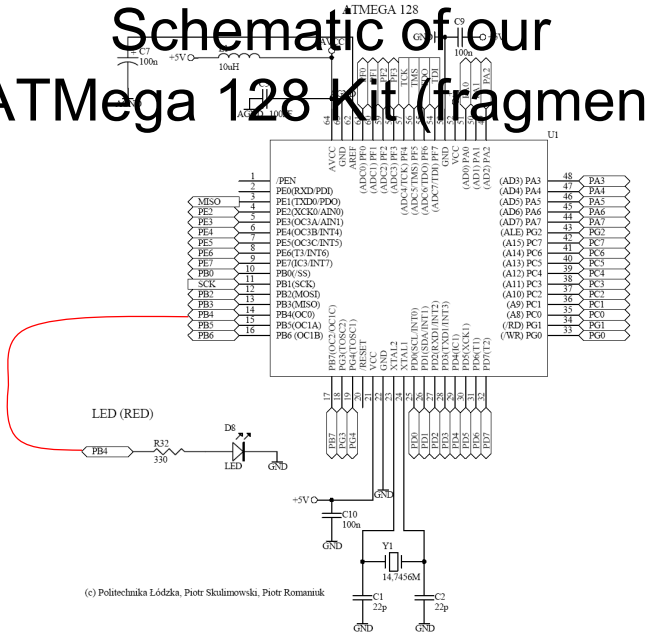


WinAVR: Editing a file

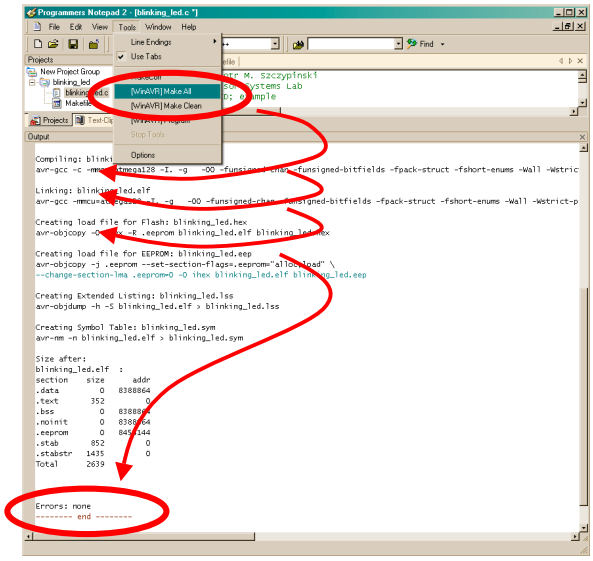


Do not forget to save the edited file!

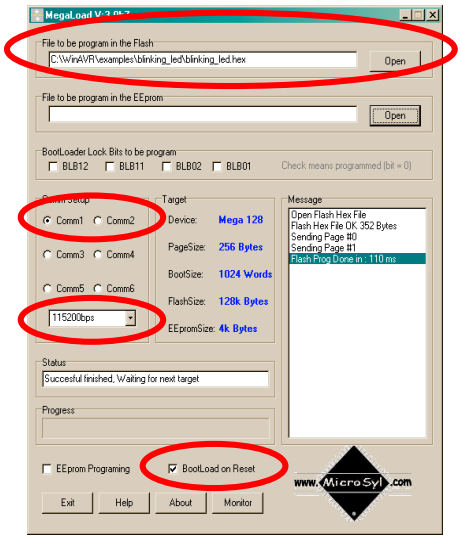
Schematic of our ATMega 128 Kit (fragment)



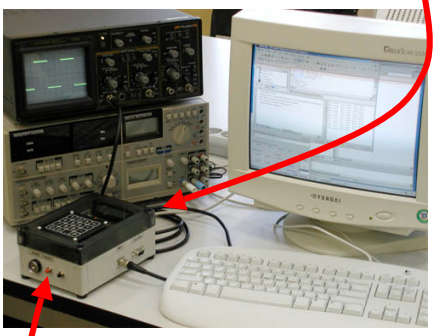
WinAVR: „Making” the project



MegaLoad: Uploading a program



Do not forget to connect ATMega Kit (RS1 connector) with a PC (COM1 or COM2)!



To upload a file push a RESET button

WinAVR → AVR Studio

AVR Studio requires so called *extcoff* files. There are several methods to generate such a file. Here is one of them:

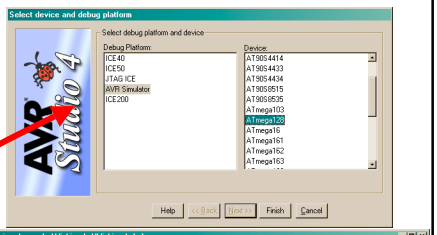
1. In PN select Tools->Options,
2. Select *Tools* from the options list
3. Select scheme *None-global tools*
4. Push *Add* button
5. Create a *MakeCoff* tool (see the picture for details)
6. A new tool will appear on the menu – make use of it!

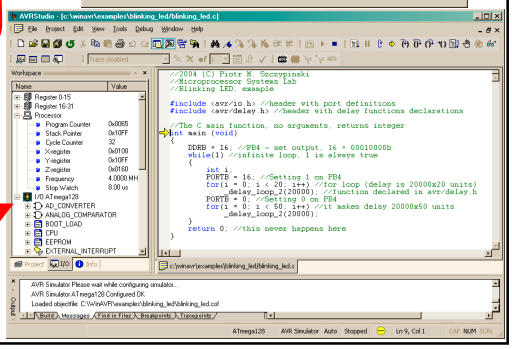
This step is unnecessary if default "make" target in makefile is set to Extended COFF

WinAVR → AVR Studio

AVR Studio is a debugging tool for AVR family. Here is how to load a program created with WinAVR into the AVR Studio:

1. Start the AVR Studio (ver. 4.07 or later),
2. If a *Welcom to* window appears, close it,
3. Select File->Open file… and load a file with a cof extension,
4. Select a device and debug platform: *AVR Simulator* and *ATMega 128*
5. After loading the AVR Studio is ready for debugging

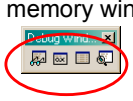




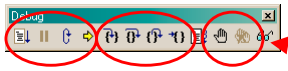
AVR Studio: What you need to know

Refer to the AVR Studio's User Guide and find out:

1. What are these tools for: watch view, register window, memory window and disassembler?



2. How to *run* a program, how to *stop* and *reset* it?



3. What is a difference between *trace into*, *step over* and *step out*?

4. What are *breakpoints*?

Data Representation in Binary System

Binary System

A byte is commonly used as a unit of storage measurement in computers, regardless of the type of data being stored. On modern computers, an eight-bit byte or octet is by far the most common.
(Wikipedia)

Multiples of bytes as defined by IEC 60027-2					
SI prefix			Binary prefixes		
Name	Symbol	Multiple	Name	Symbol	Multiple
kilobyte	kB	10^3 (or 2^{10})	kibibyte	KiB	2^{10}
megabyte	MB	10^6 (or 2^{20})	mebibyte	MiB	2^{20}
gigabyte	GB	10^9 (or 2^{30})	gibibyte	GiB	2^{30}
terabyte	TB	10^{12} (or 2^{40})	tebibyte	TiB	2^{40}
petabyte	PB	10^{15} (or 2^{50})	pebibyte	PiB	2^{50}
exabyte	EB	10^{18} (or 2^{60})	exbibyte	EiB	2^{60}
zettabyte	ZB	10^{21} (or 2^{70})			
yottabyte	YB	10^{24} (or 2^{80})			

In computing, "word" is a term for the natural unit of data used by a particular computer design. A word is simply a fixed-sized group of bits that are handled together by the machine. The word size (or length) is an important characteristic of a computer architecture.
(Wikipedia)

Binary System

The binary numeral system represents numeric values using two symbols, typically 0 and 1. More specifically, binary is a positional notation with a radix of two. (Wikipedia)

Conversion to a decimal system

X	…	1	0	0	1	1	0
2^n	…	2^5	2^4	2^3	2^2	2^1	2^0
		32	16	8	4	2	1

1•32 + 0•16 + 0•8 + 1•4 + 1•2 + 0•1 = 38

4-bit numbers	
Binary	Decimal
1111	15
1110	14
1101	13
1100	12
1011	11
1010	10
1001	9
1000	8
0111	7
0110	6
0101	5
0100	4
0011	3
0010	2
0001	1
0000	0

Conversion to a hexadecimal system

4-bit numbers	
Binary	Hexadecimal
1111	0xF
1110	0xE
1101	0xD
1100	0xC
1011	0xB
1010	0xA
1001	0x9
1000	0x8
0111	0x7
0110	0x6
0101	0x5
0100	0x4
0011	0x3
0010	0x2
0001	0x1
0000	0x0

Binary no.:	1001	1101	0110	0001	1110
Hexadecimal	9	D	6	1	E

Binary System

Fill it in

8-bit numbers	
Binary	Decimal
11111111	255
11111110	254
...	...
11101000	
11100111	
...	...
10000001	129
10000000	128
01111111	127
01111110	126
...	...
00100111	
00100110	
00100101	
...	...
00000011	3
00000010	2
00000001	1
00000000	0

Convert binary numbers to a decimal and hexadecimal numbers

110100100100
111111111111
000100000000
000000010000
000011111111

Convert to binary numbers

Decimal no.: 128, 1276, 2048, 17
Hexadecimal no.: 0xFF, 0x1111, 0x81

Binary System

Endianness

Endianness is an arbitrary convention of byte order, required when integers or any other data are represented with multiple bytes. In such situations, there are different ways those bytes can be arranged in memory or in transmission over some medium.

Example:
00010111 01101100 = Dec. 5996

Addr.	Data
n+2	...
n+1	01101100
n	00010111
n-1	...

Addr.	Data
n+2	...
n+1	00010111
n	01101100
n-1	...

Big-endian
That is, the most significant byte (also known as the msb) is stored at the memory location with the lowest address, the next byte in significance is stored at the next memory location and so on.

Little-endian
That is, least significant byte (also known as lsb) is stored at the memory location with the lowest address.

Signifying negative integers
Two's complement

Let us count down 9-bit no.:

1 0000 0011 = dec. 259
1 0000 0010 = dec. 258
1 0000 0001 = dec. 257
1 0000 0000 = dec. 256
0 1111 1111 = dec. 255
0 1111 1110 = dec. 254
0 1111 1101 = dec. 253

Now, we remove the 9-th bit:

~~1~~ 0 000 0011 = dec. 259 - 256 = 3
~~1~~ 0 000 0010 = dec. 258 - 256 = 2
~~1~~ 0 000 0001 = dec. 257 - 256 = 1
~~1~~ 0 000 0000 = dec. 256 - 256 = 0
~~0~~ 1 111 1111 = dec. 255 - 256 = -1
~~0~~ 1 111 1110 = dec. 254 - 256 = -2
~~0~~ 1 111 1101 = dec. 253 - 256 = -3

Sign

Signifying negative integers
Two's complement

Fill it in

Binary	Signed
01111111	127
01111110	126
...	...
00100111	
00100110	
00100101	
...	...
00000011	3
00000010	2
00000001	1
00000000	0
11111111	–1
11111110	–2
...	...
11101000	
11100111	
...	...
10000001	–127
10000000	–128

Calculating two's complement

Algorithm
1. Invert all the bits (bitwise NOT)
2. Increase the value by one (+1)

Example:

Decimal 12	00001100
Bitwise NOT	11110011
+1	11110100
Decimal -12	
Bitwise NOT	00001011
+1	00001100
Decimal 12	

Binary System

Fixed-point positive numbers

Integer:

X	…	0	0	1	0	0	1	1	0
2^n	…	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
		128	64	32	16	8	4	2	1

= Dec. 38

Let us add a point somewhere in between the binary digits:

X	…	0	0	1	0	0 •	1	1	0
2^k	…	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}
		16	8	4	2	1	1/2	1/4	1/8

Integer part Fraction part

= Dec. 4 $^3/_4$

$$38 \cdot 2^{-3} = 38 / 8 = 4\ ^3/_4$$

Binary System

Floating-point numbers

A floating-point number is a digital representation for a number in a certain subset of the rational numbers, and is often used to approximate an arbitrary real number on a computer. In particular, it represents an integer or fixed-point number (the significand or, informally, the mantissa) multiplied by a base (usually 2 in computers) to some integer power (the exponent). When the base is 2, it is the binary analogue of scientific notation (in base 10). (Wikipedia)

Binary System

Floating-point numbers

Single precision (32-bit) *IEEE 754 Std.* :

S	E	M
sign	exponent (bias +127)	fraction (mantissa)
31	30 23	22 0

Normalised no. = $(-1)^S\, 2^{E-127}\, (1.\mathbf{M})$

unsigned integer converted to a decimal system **(0 < E < 255)**

fraction bits

Exceptions:

S=0, **E**=255, **M**=0	$+\infty$
S=1, **E**=255, **M**=0	$-\infty$
E=255	**NaN**
S=0, **E**=0, **M**=0	**0**
E=0	denormalised no.= $=(-1)^S\, 2^{-126}\, (0.\mathbf{M})$

Binary System

Floating-point numbers

S	E	M
sign	exponent (bias +127)	fraction (mantissa)
31	30 23	22 0

$$FP = (-1)^{S}\, 2^{E-127}\, (1.\mathbf{M})$$

Example 1:

Convert 1 10000010 110100000...0 to a decimal equivalent

S = 1 E = 10000010_2 = 130 1.M = $1.1101_2 = 1\ {}^{13}/_{16}$

FP = $(-1)^1\, 2^{130-127}\, 1.1101_2 = -2^3\, 1.1101_2 = -1110.1_2 = -14\ {}^{1}/_{2}$

Example 2:

Convert a π into a binary single precision number

$\pi = 3.14159265358979 = 3 + 0.\ 14159265358979 \times 2^{23} / 2^{23} =$

$\approx 3 + 1187765 / 2^{23} = 3 + 0.10010000111110110101_2 =$

$= 11.10010000111110110101_2 =$

$= 2^{128-127}\, 1.11001000011111101101010_2$

0 10000000 11001000011111101101010

Binary System

Floating-point numbers

Double precision (64-bit) *IEEE 754 Std.* :

S	E	M
sign	exponent (bias +1023)	fraction (mantissa)
63	62 52	51 0

$$FP = (-1)^{S}\, 2^{E-1023}\, (1.\mathbf{M})$$

Binary System

Binary-coded decimals (BCD)

In BCD (SBCD 8421), a digit is usually represented by four (binary) bits, of which the leftmost (written conventionally) has value 8, and the remaining three have values 4, 2, and 1. Only the combinations of these bits which, when summed, have values in the range 0-9 are valid. (Wikipedia)

Digit	SBCD 8421	Excess-3	BCD 2421	BCD 84-2-1	IBM 702 IBM 705 IBM 7080 IBM 1401 8421
0	0000	0011	0000	0000	1010
1	0001	0100	0001	0111	0001
2	0010	0101	0010	0110	0010
3	0011	0110	0011	0101	0011
4	0100	0111	0100	0100	0100
5	0101	1000	1011	1011	0101
6	0110	1001	1100	1010	0110
7	0111	1010	1101	1001	0111
8	1000	1011	1110	1000	1000
9	1001	1100	1111	1111	1001

(Wikipedia)

Binary System

Binary-coded decimals (BCD)

Example 1:

BCD 1001 0010 0101 0110

Dec. 9 3 5 6

Digit	SBCD 8421
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Example 2:

BCD 1011 0010 1101 0110

Dec. Invalid 3 Invalid 6

Binary System

Alphanumerics and texts

American Standard Code for Information Interchange (ASCII) is the numerical representation of an alphanumeric (e.g.: 'a ', '5' or '#') or an action of some sort.
ASCII was published in 1963 (uppercase letters defined) and in 1967.

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Binary System

Alphanumerics and texts

Example:

10001110	1101111	01101111	01100100	00001101	00001010
G	o	o	d	[CR]	[LF]
1100111	01110101	01111001	01110011	00000000	
g	u	y	s	[NULL]	

End of text string in C

Starting a new line

Binary System

Alphanumerics and texts

Ascii was very simplistic **(7-bit codes)**, and so was extended by adding 'extended' sets by various manufacturers. Apart from being confusing this was still restricted to 256 characters **(8-bit codes)**. Now computers are more widely established around the world the need to show other characters such as Japanese and Chinese languages along with various symbols became more important.

Unicode **(usually 16-bit codes)** is an attempt to standardize every character possible.

(**http://www.lookuptables.com/**)

Refer to:

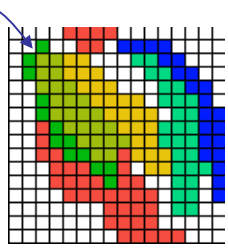
- **http://www.lookuptables.com/**
- **http://www.unicode.org/**

Images and Graphics

Raster Images

A raster graphics image is a data file or structure representing a generally rectangular grid of pixels, or points of color. The color of each pixel is individually defined.

Pixel



Header

```
42 4D 36 05 00 00 00 00 00 00 36 04 00 00 28 00
00 00 10 00 00 00 10 00 00 00 01 00 08 00 00 00
00 00 00 01 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 80
00 00 00 80 80 00 80 00 00 00 80 00 80 00 80 80
```

Image Data

```
FF FF FF FF 4F 4F 4F 4F FF FF FF FF FF FF FF FF
FF FF 31 FF FF 4F 4F FF FC FC FC FC FF FF FF FF
FF 30 35 35 37 37 FF FF FF BA BA FC FC FF FF FF
FF 30 35 35 37 37 37 FF FF FF BA BA FC FC FF FF
FF FF 35 35 35 37 37 37 FF FF FF BA BA FC FC FF
FF FF 31 35 35 35 37 37 37 37 FF BA BA BA FC FF
FF FF 30 35 35 35 35 37 37 37 37 FF BA BA FC FC
FF FF 4F 30 35 35 35 35 37 37 37 BA BA BA FC FC
FF FF 4F 30 30 35 35 35 35 37 37 BA BA BA FC FC
FF FF 4F 4F 30 30 35 35 4F 37 37 37 BA BA FC FC
FF FF FF FF 4F 4F 4F 30 30 4F FF 37 37 BA BA BA FC
FF FF FF 4F 4F 4F 4F 30 4F 4F FF FF BA BA FF FC
FF FF FF FF 4F 4F 4F 4F 4F 4F 4F 4F FF BA BA FF FC
FF FF FF FF FF FF FF FF 4F 4F 4F 4F FF BA BA FF FF
FF FF FF FF FF FF FF FF 4F 4F 4F 4F FF FF FF FF FF
FF FF FF FF FF FF FF FF FF 4F 4F 4F 4F 4F FF FF FF
```

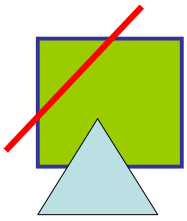
Palette LUT

```
40 00 40
40 60 00
40 40 80
00 40 40
A0 00 40
40 C0 00
40 40 E0
...
...
...
```

Images and Graphics

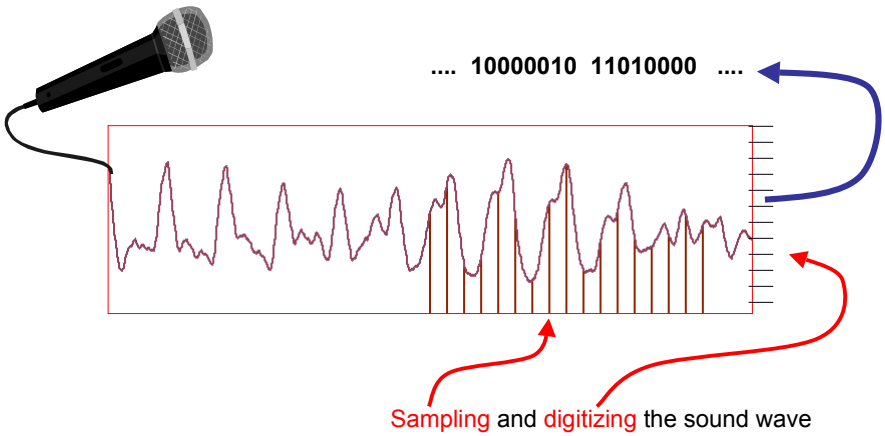
Vector Graphics

Vector graphics or geometric modeling is the use of geometrical primitives such as points, lines, curves, and polygons to represent images in computer graphics. It is used by contrast to the term raster graphics, which is the representation of images as a collection of pixels (dots).



SetPixelV(kontekst, x, y, RGB(r, g, b));
MoveToEx(kontekst, x, y, NULL);
LineTo(kontekst, x, y);
Rectangle(kontekst, x1, y1, x2, y2);

Sound



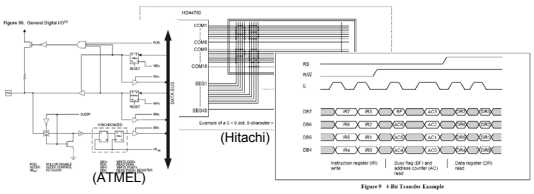
I/O Device Programming

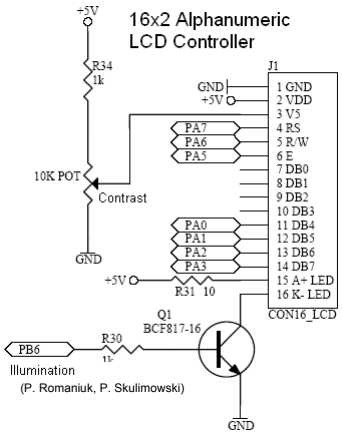
ATMega 128 Kit
Display and Keybord

ATMega 128 Kit
Display

Lecture based on documentation of microcontroller ATMega 128 from ATMEL® and alphanumeric liquid-crystal display driver HD44780U from Hitachi®.

1. Schematic of ATMega 128 ports,
2. I/O Ports registers (PORTx, DDRx),
3. HD44780 pins and their functions,
4. Four and eight bit data bus configuration,
5. Sending instructions, configuring the LCD driver,
6. Sending alphanumeric codes.



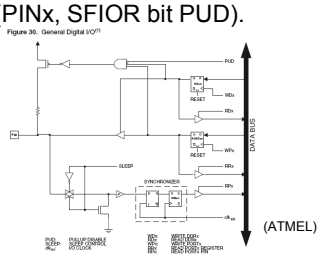


ATMega 128 Kit

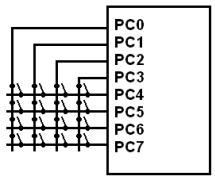
Keybord

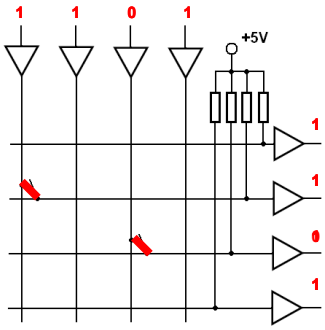
Lecture based on documentation
of microcontroller ATMega 128 from ATMEL®

1. Keybord schematic,
2. Methods for idendification
   the key being pressed,
3. ATMega 128 port schematic
   (pull up resistors),
4. Function of port registers
   (PINx, SFIOR bit PUD).



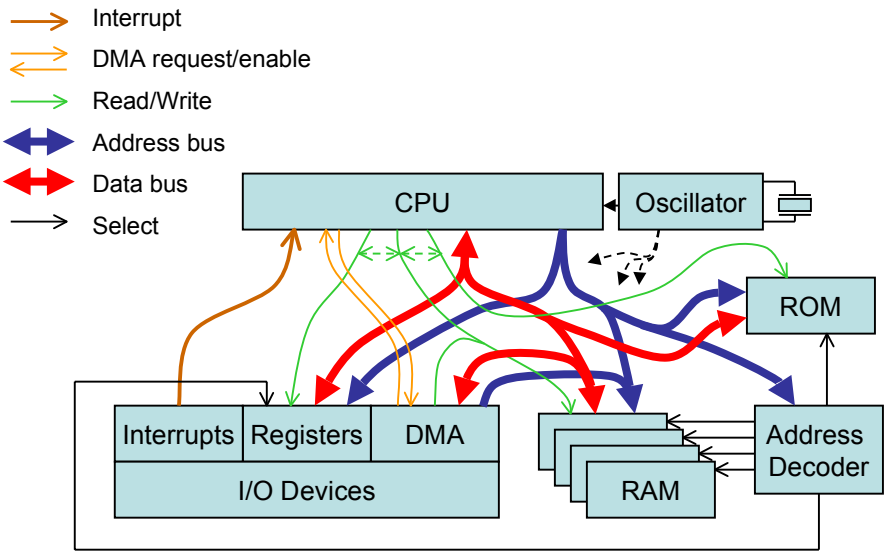
(ATMEL)





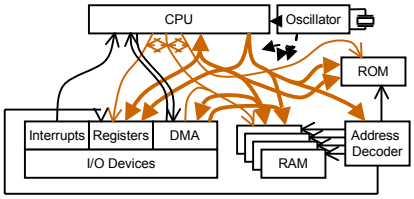
Computer Architecture

Computer Architecture



Computer Architecture

Address, Data & Control Bus (Read/Write signals)



In computer architecture, a bus is a subsystem that transfers data or power between computer components inside a computer or between computers. A bus (address bus, data bus) is usually a group of parallel wires connecting different parts of a circuit with each individual wire carrying a different logic signal.

An address bus is used by CPUs or DMA-capable units for communicating the physical addresses of computer memory elements (location) that the requesting unit wants to access (read or write). The width of an address bus, along with the size of addressable memory elements, determines how much memory can be accessed. **Where?**

A data bus (connections between and within the CPU, memory, and peripherals) is used to carry data. **What?**

A control bus is used by CPUs for communicating with other devices within the computer. While the address bus carries the information on which device the CPU is communicating with and the data bus carries the actual data being processed, the control bus carries commands from the CPU and returns status signals from the devices, for example if the data is being read or written to the device the appropriate line (of a control bus) will be active. **How?**

Computer Architecture

The term von Neumann architecture refers to a computer design model that uses a single storage structure to hold both instructions and data. The term von Neumann machine can be used to describe such a computer, but that term has other meanings as well. The separation of storage from the processing unit is implicit in the von Neumann architecture.

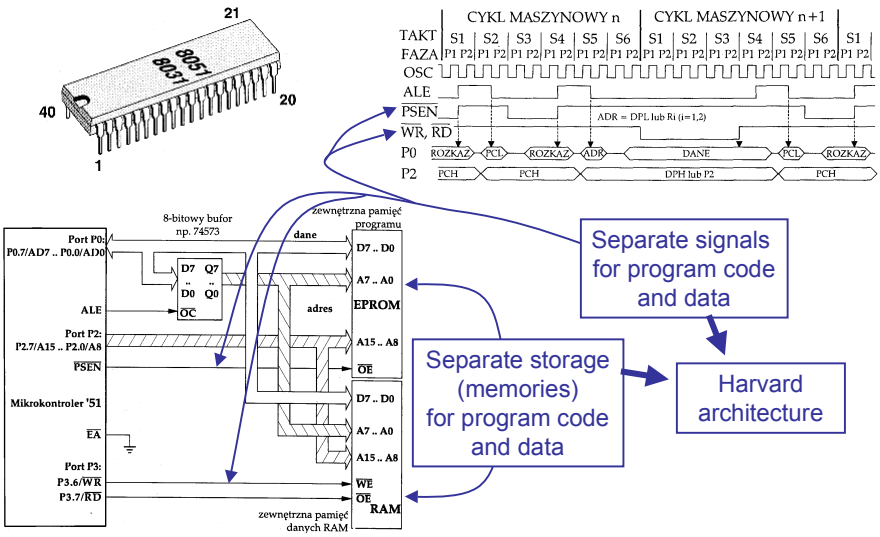
The term Harvard architecture originally referred to computer architectures that used physically separate storage and signal pathways for their instructions and data (in contrast to the von Neumann architecture). The term originated from the Harvard Mark I relay-based computer, which stored instructions on punched tape (24-bits wide) and data in relay latches (23-digits wide).

Memory can be made much faster, but only at high cost. The solution then is to provide a small amount of very fast memory known as a cache. As long as the memory that the CPU needs is in the cache, the performance hit is much smaller than it is when the cache has to turn around and get the data from the main memory. Tuning the cache is an important aspect of computer design.
Modern high performance CPU chip designs incorporate aspects of both Harvard and von Neumann architecture. On chip cache memory is divided into an instruction cache and a data cache. Harvard architecture is used as the CPU accesses the cache. In the case of a cache miss, however, the data is retrieved from the main memory, which is not divided into separate instruction and data sections. Thus a von Neumann architecture is used for off chip memory access.

(Wikipedia)

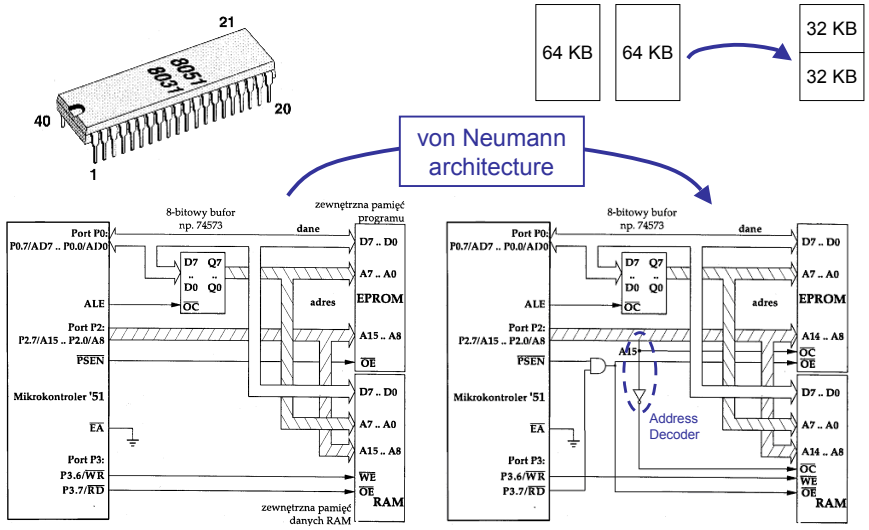
Computer Architecture

Example: Intel 8051



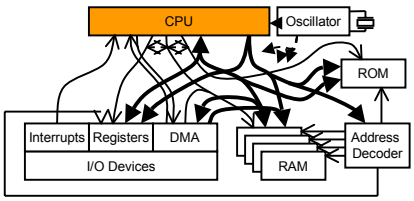
Computer Architecture

Example: Intel 8051

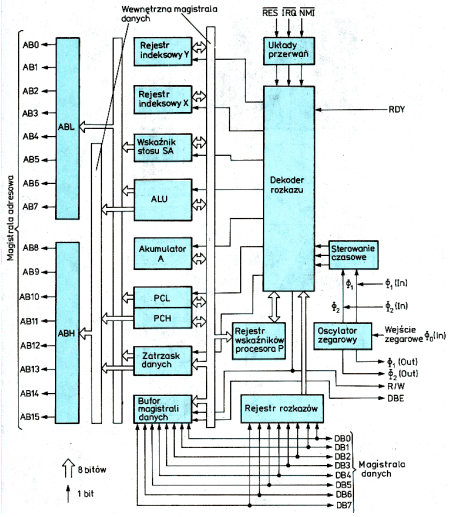


Computer Architecture

Microprocessor (CPU)



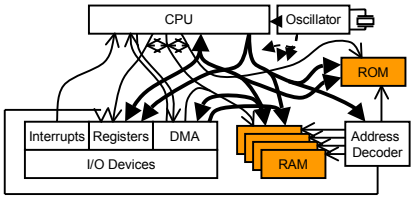
A central processing unit (CPU) refers to part of a computer that interprets and executes instructions

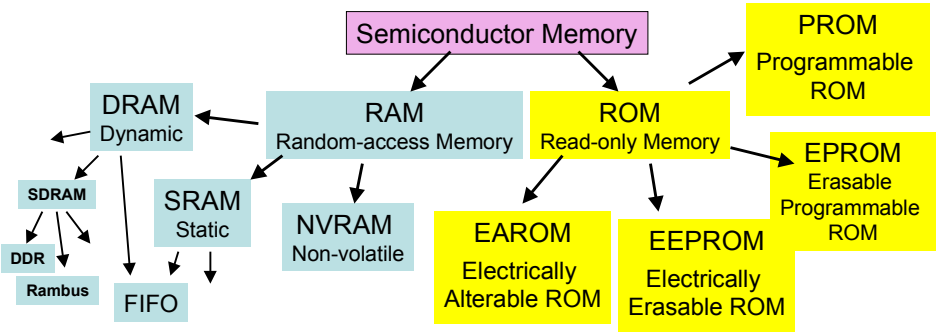


Motorola 6800 (H. Feichtinger, Mikrokomputery – poradnik)

Computer Architecture

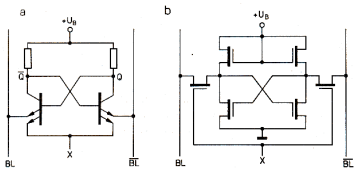
Memory





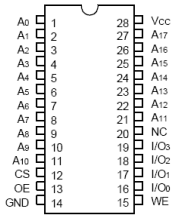
Computer Architecture

RAM - Static vs. Dynamic

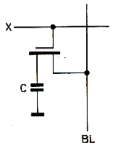


TTL MOS
Static memory cell

The static RAM retains its contents as long as power remains applied.
- Fast,
- Low density.

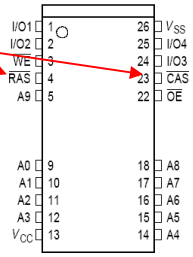


Column and Row Address Strobes



Dynamic memory cell

The Dynamic RAM needs to be periodically refreshed.
- Slow,
- High density.



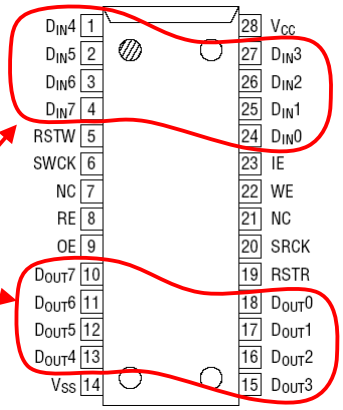
What are the storage sizes (in Bytes) of the above Static and Dynamic memories?

Computer Architecture

FIFO

FIFO (First In, First Out) memory stores the data in queue order so the first input element goes out the first. FIFO memory chips are used in buffering applications between devices that operate at various speeds.

Dual data interface:
- Input
- Output



MSM518221A (OKI Semiconductor)

Computer Architecture

Dynamic RAM: SDRAM, DDR RAM

Synchronous Dynamic (SD) RAM is an improved type of DRAM. Whilst DRAM has an asynchronous interface, meaning that it reacts immediately to changes in its control inputs, SDRAM has a synchronous interface, meaning that it waits for a clock pulse before responding to its control inputs. The clock is used to drive an internal finite state machine that can pipeline incoming commands. This allows the chip to have a more complex pattern of operation than plain DRAM.

Double data rate (DDR) SDRAM is a later development of SDRAM, used in PC memory from 2000 onwards. All types of SDRAM use a clock signal that is a square wave. This means that the clock alternates regularly between one voltage (low) and another (high), usually millions of times per second. Plain SDRAM, like most synchronous logic circuits, acts on the low-to-high transition of the clock and ignores the opposite transition. DDR SDRAM acts on both transitions, thereby halving the required clock rate for a given data transfer rate.

Direct Rambus DRAM (DRDRAM), often called RDRAM, is internally similar to DDR SDRAM, but uses a special method of signaling developed by the Rambus Company that allows faster clock speeds.

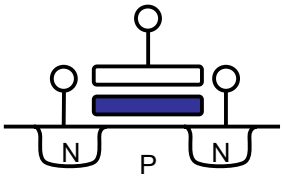
(Wikipedia)

Computer Architecture

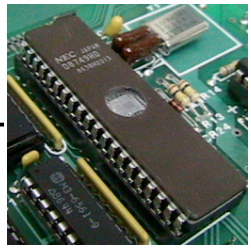
PROM, EPROM, EEPROM

PROMs (Programmable Read-Only Memory) can be programmed via a special device, a PROM programmer. The writing often takes the form of permanently destroying or creating internal links (fuses or antifuses) with the result that a PROM can only be programmed once.

EPROMs (Erasable Programmable Read-Only Memory) can be erased by exposure to ultraviolet light then rewritten via an EPROM programmer. Repeated exposure to ultraviolet light will eventually destroy the EPROM but it generally takes many exposures before the EPROM becomes unusable.



Floating gate MESFET

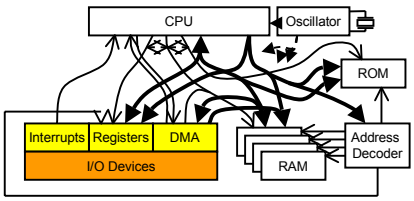


EAROMs (Electrically Alterable Read-Only Memory) can be modified a bit at a time, but writing is intended to be an infrequent operation; most of the time the memory is used as a ROM. EAROM may be used to store critical system setup information in a non-volatile way. For many applications, EAROM has been supplanted by CMOS RAM backed-up by a lithium battery. EEPROM such as Flash memory (Electrically Erasable Read-Only Memory) allow the entire ROM (or selected banks of the ROM) to be electrically erased (flashed back to zero) then written to without taking them out of the computer (camera, MP3 player, etc.). Flashing is much slower than writing to RAM (Random Access Memory) (or reading from any ROM).

(Wikipedia)

Computer Architecture

Communication with I/O Devices



I/O devices are used by a person (or other system) to communicate with a computer. For instance, keyboards and mice are considered input devices of a computer and monitors and printers are considered output devices of a computer. Typical devices for communication between computers are for both input and output, such as modems and network cards.
In computer architecture, the combination of the CPU and main memory is considered the heart of a computer, and any movement of information from or to that complex, for example to or from a disk drive, is considered I/O.

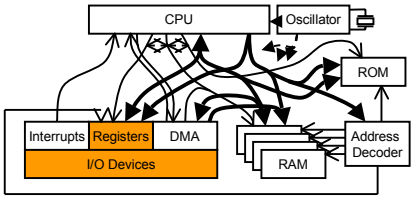
(Wikipedia)

I/O communication methods:

- Memory-mapped I/O (MMIO) and port-mapped I/O (PMIO)
- Direct memory access (DMA)
- Interrupt

Computer Architecture

Communication with I/O Devices



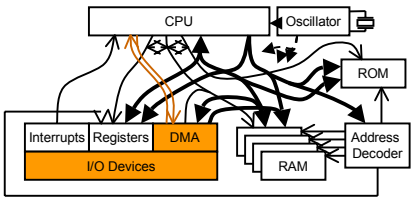
Memory-mapped I/O uses the same bus to address both memory and I/O devices, and the CPU instructions used to read and write to memory are also used in accessing I/O devices. In order to accommodate the I/O devices, areas of CPU addressable space must be reserved for I/O rather than memory. The I/O devices monitor the CPU's address bus and respond to any CPU access of their assigned address space, mapping the address to their hardware registers.

Port-mapped I/O uses a special class of CPU instructions specifically for performing I/O. This is generally found on Intel microprocessors, specifically the IN and OUT instructions which can read and write a single byte to an I/O device. I/O devices have a separate address space from general memory, either accomplished by an extra "I/O" pin on the CPU's physical interface, or an entire bus dedicated to I/O.

(Wikipedia)

Computer Architecture

Communication with I/O Devices



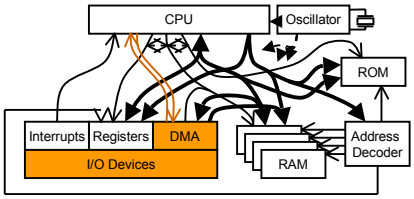
Direct memory access (DMA) allows certain hardware subsystems within a computer to access system memory for reading and/or writing independently of the CPU.

Control signals:

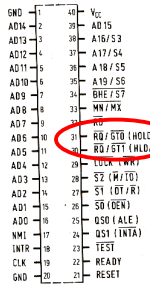
- DMA Request
- DMA Enable

Computer Architecture

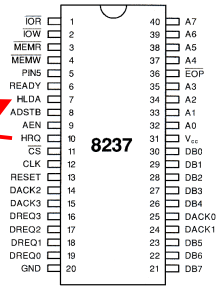
Communication with I/O Devices



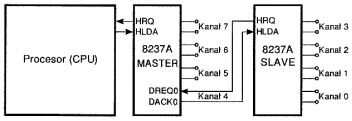
Example: IMB PC DMA system



Intel 8086



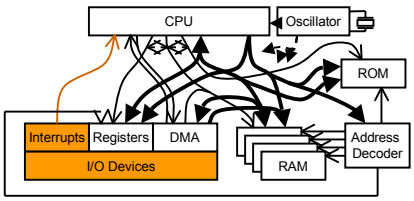
DMA Controller



DMA Controller Master-Slave configuration

Computer Architecture

Communication with I/O Devices



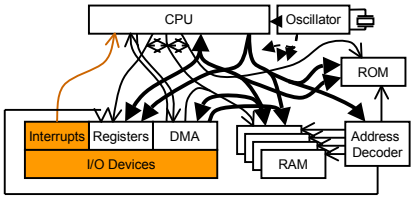
Interrupts: Digital computers usually provide a way to start software routines in response to asynchronous electronic events. These events are signaled to the processor via interrupt requests (IRQ). The processor and interrupt code make a context switch into a specifically written piece of software to handle the interrupt. This software is called the interrupt service routine, or interrupt handler.

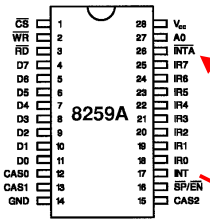
Processors also often have a mechanism referred to as interrupt disable which allows software to prevent interrupts from interfering with communication between interrupt-code and non-interrupt code. Typically, the user can configure the machine using hardware registers so that different types of interrupts are enabled or disabled, depending on what the user wants (maskable interrupts). Some interrupts cannot be disabled - these are referred to as non-maskable interrupts.

(Wikipedia)

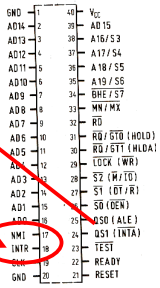
Computer Architecture

Communication with I/O Devices



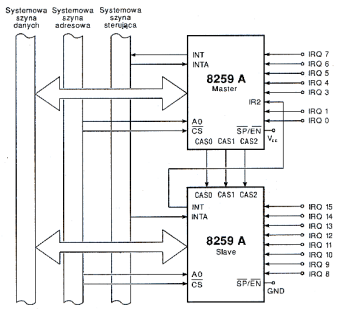


Interrupt Controller



Intel 8086

Example: IMB PC Interrupt system



Interrupt Controller Master-Slave configuration

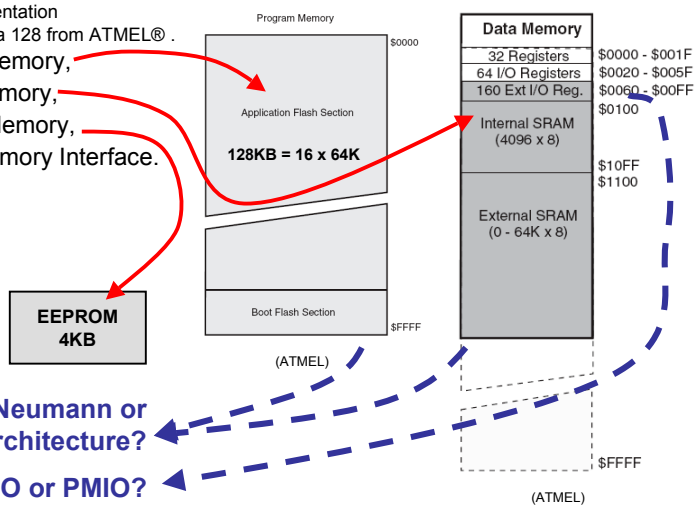
ATMega 128
Memories

ATMega 128

Memories

Lecture based on documentation
of microcontroller ATMega 128 from ATMEL® .

1. Flash Program Memory,
2. SDRAM Data Memory,
3. EEPROM Data Memory,
4. External Data Memory Interface.



Is it a von Neumann or Harvard architecture?

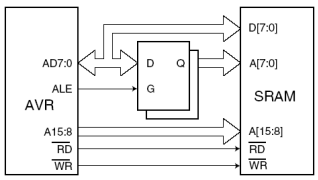
Is it a MMIO or PMIO?

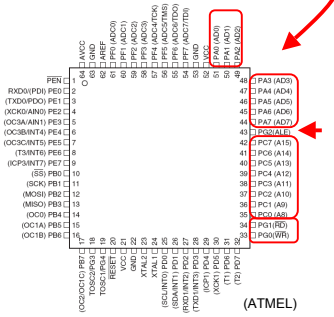
ATMega 128

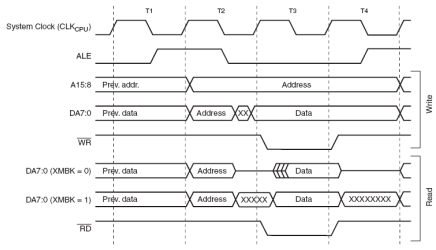
Memories

Lecture based on documentation
of microcontroller ATMega 128 from ATMEL® .

1. Flash Program Memory,
2. SDRAM Data Memory,
3. EEPROM Data Memory,
4. External Data Memory Interface.







(ATMEL)

Interrupt system of ATMega 128

ATMega 128

Interrupt system

Lecture based on documentation
of microcontroller ATMega 128 from ATMEL® .

1. Interrupt vectors table,
2. Table placement,
3. Programming
   the interrupt handler.

Table 23. Reset and Interrupt Vectors

Vector No.	Program Address(2)	Source	Interrupt Definition
1	$0000(1)	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset
2	$0002	INT0	External Interrupt Request 0
3	$0004	INT1	External Interrupt Request 1
4	$0006	INT2	External Interrupt Request 2
5	$0008	INT3	External Interrupt Request 3
6	$000A	INT4	External Interrupt Request 4
7	$000C	INT5	External Interrupt Request 5
8	$000E	INT6	External Interrupt Request 6
9	$0010	INT7	External Interrupt Request 7
10	$0012	TIMER2 COMP	Timer/Counter2 Compare Match
11	$0014	TIMER2 OVF	Timer/Counter2 Overflow
12	$0016	TIMER1 CAPT	Timer/Counter1 Capture Event
13	$0018	TIMER1 COMPA	Timer/Counter1 Compare Match A
14	$001A	TIMER1 COMPB	Timer/Counter1 Compare Match B
15	$001C	TIMER1 OVF	Timer/Counter1 Overflow
16	$001E	TIMER0 COMP	Timer/Counter0 Compare Match
17	$0020	TIMER0 OVF	Timer/Counter0 Overflow
18	$0022	SPI, STC	SPI Serial Transfer Complete
19	$0024	USART0, RX	USART0, Rx Complete
20	$0026	USART0, UDRE	USART0 Data Register Empty
21	$0028	USART0, TX	USART0, Tx Complete
22	$002A	ADC	ADC Conversion Complete
23	$002C	EE READY	EEPROM Ready
24	$002E	ANALOG COMP	Analog Comparator
25	$0030(5)	TIMER1 COMPC	Timer/Counter1 Compare Match C

(ATMEL)

```
//2005 (C) Piotr M. Szczypiński
//Interrupt handling and PWM example
//Microprocessor Systems Lab

#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/signal.h>

//Variables store the current Pulse Width (PW)
int pwm_PB5 = 0;
int pwm_PB6 = 0;

SIGNAL (SIG_OVERFLOW1)
{
//Modifying the PW for PB5 (Red/Green Led)
    pwm_PB5++;
    if(pwm_PB5>=1024) pwm_PB5 = -1023;
    OCR1A = (pwm_PB5<0 ?-pwm_PB5 :pwm_PB5);

//Modifying the PW for PB6 (Display illumination)
    pwm_PB6++;
    if(pwm_PB6>=16384) pwm_PB6 = -16383;
    OCR1B = (pwm_PB6<0 ?(-pwm_PB6)/16 :pwm_PB6/16);
}
int main(void)
{
    TCCR1A = 128|32|2|1; // 10.bit PWM
    TCCR1B = 1; // Prescaler 1-5
    OCR1A = pwm_PB5;
    OCR1B = pwm_PB6/16;
    DDRB = 64|32; //PB6, PB5 outputs
    TIMSK = 4; //Interrupt mask
    sei(); //Enables interrupts
    while(1); //Infinite loop
    return 0;
}
```

Global variables

Interrupt handler

Configuring a device sending interrupt signal

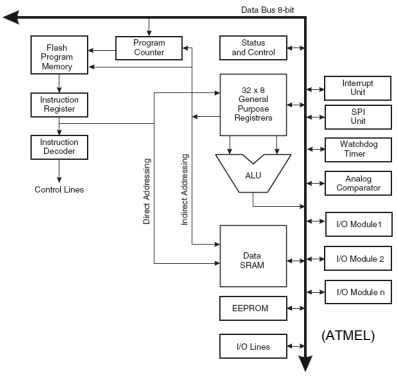
ATMega 128 CPU Core

ATMega 128

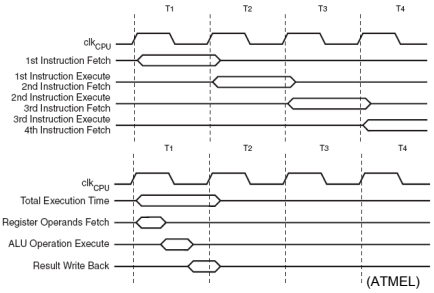
CPU Core

Lecture based on documentation
of microcontroller ATMega 128 from ATMEL® .

1. Arithmetic-Logic Unit,
2. General Purpose Registers,
3. Indexing Registers,
4. Status (Flag) Register,
5. Stack & Stack Pointer Register,
6. Instruction Execution Timings.



(ATMEL)



(ATMEL)

The AVR status Register – SREG – is defined as:

Bit	7	6	5	4	3	2	1	0	
	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	

(ATMEL)

CPUs
Some Definitions

CPUs
Instruction Sets (RISC, CISC...)

Reduced Instruction Set Computer (RISC) is a microprocessor CPU design philosophy that favors a smaller and simpler set of instructions. Properties:

- All the instructions take about the same amount of time to execute
- Typically most of the instruction codes are of a fixed length
- The total number of instructions read from memory is high

Complex Instruction Set Computer (CISC) is a CPU design in which each instruction can execute several low-level operations, such as a load from memory, an arithmetic operation, and a memory store, all in a single instruction. Properties:

- Usually instruction codes are of a variable length
- Smaller program sizes and fewer calls to program memory

Very Long Instruction Word (VLIW) is a design of a microprocessor that packs many simple RISC-like instructions into a much longer internal instruction word format. A VLIW microprocessor will usually have execution units, capable of executing all of the instructions contained in the instruction word, in parallel. Explicitely Parallel Instruction Computing (EPIC) is an Intel's acronym for VLIW.

CPUs

Instruction Pipeline

An instruction pipeline is a technology used to enhance microprocessors performance. Pipelining greatly improves throughput at a small cost in latency.

Execution of CPU instruction consist of a number of steps:

- Read the next instruction
- Read the operands, if any
- Execute the instruction
- Write the results back out

Non-pipelined processors did only one instruction at a time. Since each step of an instruction is performed by a different piece of hardware, the CPU may start executing the next instruction before accomplishing the previous one.

CPUs

Scalar vs. Superscalar

A superscalar CPU architecture implements a form of parallelism on a single chip, thereby allowing the system as a whole to run much faster than it would otherwise be able to at a given clock speed. The term is a modification of scalar, processors that run one instruction per clock cycle, themselves a step up from earlier processors that would take a variable number of cycles to complete any given operation.

CPUs

SIMD

Single Instruction, Multiple Data (SIMD). It is a computing term that refers to a set of operations for efficiently handling large quantities of data in parallel, as in a vector processor or array processor

Example: Intel Pentium II and III SIMD technologies:

Multi-Media eXtensions (MMX) is instructions extension intended to enhance programs that have multi-media capabilities. MMX is SIMD for integer numbers.

Streaming SIMD Extensions (SSE) is a SIMD instructions extension for single-precision floating-point numbers.

Assembly Language Programming

Assembly Programming

Opcode (Machine Code) vs. Mnemonic (Assembly)

A system of codes directly understandable by a computer's CPU is termed this CPU's native or machine language (or machine code).

An Opcode is the portion of a machine language instruction that specifies the operation to be performed. The term is an abbreviation of Operation Code.

Assembly language or simply assembly is a human-readable notation for the machine language that a specific computer architecture uses. Machine language, a pattern of bits encoding machine operations, is made readable by replacing the raw values with symbols called mnemonics.

Mnemonic is a code, usually from 1 to 5 letters, that represents an opcode, a number. A mnemonic is usually followed by a list of arguments.

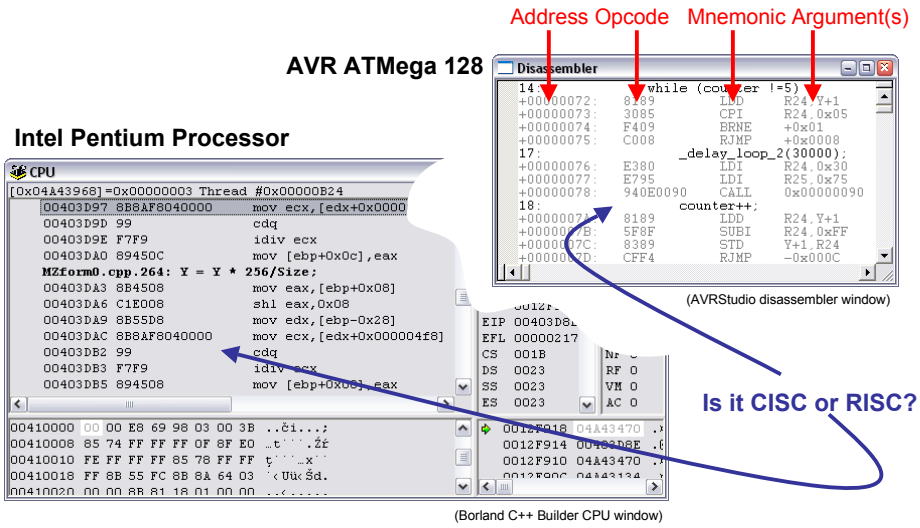
Programming in machine code, by supplying the computer with the numbers of the operations it must perform, can be quite a burden, because for every operation the corresponding number must be looked up or remembered. Looking up all numbers takes a lot of time, and misremembering a number may introduce computer bugs.

Therefore a set of mnemonics was devised. Each number was represented by an alphabetic code. So instead of entering the number corresponding to addition to add two numbers one can enter "add". Although mnemonics differ between different CPU designs some are common, for instance: "sub" (subtract), "div" (divide), "add" (add) and "mul" (multiply).

(Wikipedia)

Assembly Programming

Opcode (Machine Code) vs. Mnemonic (Assembly)



(AVRStudio disassembler window)

(Borland C++ Builder CPU window)

Assembly Programming

Assembly Language

Assembly language or simply assembly is a human-readable notation for the machine language that a specific computer architecture uses. Machine language, a pattern of bits encoding machine operations, is made readable by replacing the raw values with symbols called mnemonics.
An assembler is a computer program for translating assembly language — essentially, a mnemonic representation of machine language — into object code. A cross assembler (see cross compiler) produces code for one type of processor, but runs on another.

Every computer architecture has its own machine language, and therefore its own assembly language. Computers differ by the number and type of operations that they support. They may also have different sizes and numbers of registers, and different representations of data types in storage. While all general-purpose computers are able to carry out essentially the same functionality, the way they do it differs, and the corresponding assembly language must reflect these differences.

Transforming assembly into machine language is accomplished by an assembler, and the reverse by a disassembler. Unlike in high-level languages, there is usually a 1-to-1 correspondence between simple assembly statements and machine language instructions.

(Wikipedia)

Assembly Programming

Instruction Groups

- Arithmetic and Logic Instructions
- Branch Instructions
- Data Transfer Instructions
- Bit and Bit Test Instructions
- MCU Control Instructions

(ATMEL)

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
COM	Rd	One's Complement	Rd ← $FF − Rd	Z,C,N,V	1
NEG	Rd	Two's Complement	Rd ← $00 − Rd	Z,C,N,V,H	1
SBR	Rd,K	Set Bit(s) in Register	Rd ← Rd v K	Z,N,V	1
CBR	Rd,K	Clear Bit(s) in Register	Rd ← Rd • ($FF - K)	Z,N,V	1
INC	Rd	Increment	Rd ← Rd + 1	Z,N,V	1
DEC	Rd	Decrement	Rd ← Rd − 1	Z,N,V	1
TST	Rd	Test for Zero or Minus	Rd ← Rd • Rd	Z,N,V	1
CLR	Rd	Clear Register	Rd ← Rd ⊕ Rd	Z,N,V	1
SER	Rd	Set Register	Rd ← $FF	None	1
MUL	Rd, Rr	Multiply Unsigned	R1:R0 ← Rd x Rr	Z,C	2
MULS	Rd, Rr	Multiply Signed	R1:R0 ← Rd x Rr	Z,C	2
MULSU	Rd, Rr	Multiply Signed with Unsigned	R1:R0 ← Rd x Rr	Z,C	2
FMUL	Rd, Rr	Fractional Multiply Unsigned	R1:R0 ← (Rd x Rr) << 1	Z,C	2
FMULS	Rd, Rr	Fractional Multiply Signed	R1:R0 ← (Rd x Rr) << 1	Z,C	2
FMULSU	Rd, Rr	Fractional Multiply Signed with Unsigned	R1:R0 ← (Rd x Rr) << 1	Z,C	2

Assembly Programming

Instruction Groups

- ~~Arithmetic and Logic Instructions~~
- Branch Instructions (Jumps)
- ~~Data Transfer Instructions~~
- Bit and Bit Te
- MCU Control

(ATMEL)

BRANCH INSTRUCTIONS					
RJMP	k	Relative Jump	PC ← PC + k + 1	None	2
IJMP		Indirect Jump to (Z)	PC ← Z	None	2
JMP	k	Direct Jump	PC ← k	None	3
RCALL	k	Relative Subroutine Call	PC ← PC + k + 1	None	3
ICALL		Indirect Call to (Z)	PC ← Z	None	3
CALL	k	Direct Subroutine Call	PC ← k	None	4
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, N,V,C,H	1
CPC	Rd,Rr	Compare with Carry	Rd – Rr – C	Z, N,V,C,H	1
CPI	Rd,K	Compare Register with Immediate	Rd – K	Z, N,V,C,H	1
SBRC	Rr, b	Skip if Bit in Register Cleared	if (Rr(b)=0) PC ← PC + 2 or 3	None	1 / 2 / 3
SBRS	Rr, b	Skip if Bit in Register is Set	if (Rr(b)=1) PC ← PC + 2 or 3	None	1 / 2 / 3
SBIC	P, b	Skip if Bit in I/O Register Cleared	if (P(b)=0) PC ← PC + 2 or 3	None	1 / 2 / 3
SBIS	P, b	Skip if Bit in I/O Register is Set	if (P(b)=1) PC ← PC + 2 or 3	None	1 / 2 / 3
BRBS	s, k	Branch if Status Flag Set	if (SREG(s) = 1) then PC←PC+k + 1	None	1 / 2
BRBC	s, k	Branch if Status Flag Cleared	if (SREG(s) = 0) then PC←PC+k + 1	None	1 / 2
BREQ	k	Branch if Equal	if (Z = 1) then PC ← PC + k + 1	None	1 / 2
BRNE	k	Branch if Not Equal	if (Z = 0) then PC ← PC + k + 1	None	1 / 2
BRCS	k	Branch if Carry Set	if (C = 1) then PC ← PC + k + 1	None	1 / 2
BRCC	k	Branch if Carry Cleared	if (C = 0) then PC ← PC + k + 1	None	1 / 2
BRSH	k	Branch if Same or Higher	if (C = 0) then PC ← PC + k + 1	None	1 / 2
BRLO	k	Branch if Lower	if (C = 1) then PC ← PC + k + 1	None	1 / 2
BRMI	k	Branch if Minus	if (N = 1) then PC ← PC + k + 1	None	1 / 2
BRPL	k	Branch if Plus	if (N = 0) then PC ← PC + k + 1	None	1 / 2
BRGE	k	Branch if Greater or Equal, Signed	if (N ⊕ V= 0) then PC ← PC + k + 1	None	1 / 2
BRLT	k	Branch if Less Than Zero, Signed	if (N ⊕ V= 1) then PC ← PC + k + 1	None	1 / 2
BRHS	k	Branch if Half Carry Flag Set	if (H = 1) then PC ← PC + k + 1	None	1 / 2
BRHC	k	Branch if Half Carry Flag Cleared	if (H = 0) then PC ← PC + k + 1	None	1 / 2
BRTS	k	Branch if T Flag Set	if (T = 1) then PC ← PC + k + 1	None	1 / 2
BRTC	k	Branch if T Flag Cleared	if (T = 0) then PC ← PC + k + 1	None	1 / 2
BRVS	k	Branch if Overflow Flag is Set	if (V = 1) then PC ← PC + k + 1	None	1 / 2
BRVC	k	Branch if Overflow Flag is Cleared	if (V = 0) then PC ← PC + k + 1	None	1 / 2
BRIE	k	Branch if Interrupt Enabled	if (I = 1) then PC ← PC + k + 1	None	1 / 2
BRID	k	Branch if Interrupt Disabled	if (I = 0) then PC ← PC + k + 1	None	1 / 2

Assembly Programming

Instruction Groups

- Arithmetic and Logic Instructions
- ~~Branch Instructions (Jumps)~~
- Data Transfer Instructions
- ~~Bit and Bit Test Instruction~~
- MCU Control Instructions

(ATMEL)

DATA TRANSFER INSTRUCTIONS			
MOV	Rd, Rr	Move Between Registers	Rd ← Rr
MOVW	Rd, Rr	Copy Register Word	Rd+1:Rd ← Rr+1:Rr
LDI	Rd, K	Load Immediate	Rd ← K
LD	Rd, X	Load Indirect	Rd ← (X)
LD	Rd, X+	Load Indirect and Post-Inc.	Rd ← (X), X ← X + 1
LD	Rd, - X	Load Indirect and Pre-Dec.	X ← X - 1, Rd ← (X)
LD	Rd, Y	Load Indirect	Rd ← (Y)
LD	Rd, Y+	Load Indirect and Post-Inc.	Rd ← (Y), Y ← Y + 1
LD	Rd, - Y	Load Indirect and Pre-Dec.	Y ← Y - 1, Rd ← (Y)
LDD	Rd,Y+q	Load Indirect with Displacement	Rd ← (Y + q)
LD	Rd, Z	Load Indirect	Rd ← (Z)
LD	Rd, Z+	Load Indirect and Post-Inc.	Rd ← (Z), Z ← Z+1
LD	Rd, -Z	Load Indirect and Pre-Dec.	Z ← Z - 1, Rd ← (Z)
LDD	Rd, Z+q	Load Indirect with Displacement	Rd ← (Z + q)
LDS	Rd, k	Load Direct from SRAM	Rd ← (k)
ST	X, Rr	Store Indirect	(X) ← Rr
ST	X+, Rr	Store Indirect and Post-Inc.	(X) ← Rr, X ← X + 1
ST	- X, Rr	Store Indirect and Pre-Dec.	X ← X - 1, (X) ← Rr
ST	Y, Rr	Store Indirect	(Y) ← Rr
ST	Y+, Rr	Store Indirect and Post-Inc.	(Y) ← Rr, Y ← Y + 1
ST	- Y, Rr	Store Indirect and Pre-Dec.	Y ← Y - 1, (Y) ← Rr
STD	Y+q,Rr	Store Indirect with Displacement	(Y + q) ← Rr
ST	Z, Rr	Store Indirect	(Z) ← Rr
ST	Z+, Rr	Store Indirect and Post-Inc.	(Z) ← Rr, Z ← Z + 1
ST	-Z, Rr	Store Indirect and Pre-Dec.	Z ← Z - 1, (Z) ← Rr
STD	Z+q,Rr	Store Indirect with Displacement	(Z + q) ← Rr
STS	k, Rr	Store Direct to SRAM	(k) ← Rr
LPM		Load Program Memory	R0 ← (Z)
LPM	Rd, Z	Load Program Memory	Rd ← (Z)
LPM	Rd, Z+	Load Program Memory and Post-Inc	Rd ← (Z), Z ← Z+1
ELPM		Extended Load Program Memory	R0 ← (RAMPZ:Z)
ELPM	Rd, Z	Extended Load Program Memory	Rd ← (RAMPZ:Z)
ELPM	Rd, Z+	Extended Load Program Memory and Post-Inc	Rd ← (RAMPZ:Z), RAMPZ:Z ← RAMPZ:Z+1
SPM		Store Program Memory	(Z) ← R1:R0
IN	Rd, P	In Port	Rd ← P
OUT	P., Rr	Out Port	P ← Rr
PUSH	Rr	Push Register on Stack	STACK ← Rr
POP	Rd	Pop Register from Stack	Rd ← STACK

Assembly Programming

Instruction Groups

- Arithmetic and Logic Instructions
- Branch Instructions (Jumps)
- ~~Data Transfer Instructions~~
- Bit and Bit Test Instructions
- ~~MCU Control Instructions~~

(ATMEL)

BIT AND BIT-TEST INSTRUCTIONS					
SBI	P,b	Set Bit in I/O Register	I/O(P,b) ← 1	None	2
CBI	P,b	Clear Bit in I/O Register	I/O(P,b) ← 0	None	2
LSL	Rd	Logical Shift Left	Rd(n+1) ← Rd(n), Rd(0) ← 0	Z,C,N,V	1
LSR	Rd	Logical Shift Right	Rd(n) ← Rd(n+1), Rd(7) ← 0	Z,C,N,V	1
ROL	Rd	Rotate Left Through Carry	Rd(0)←C,Rd(n+1)← Rd(n),C←Rd(7)	Z,C,N,V	1
ROR	Rd	Rotate Right Through Carry	Rd(7)←C,Rd(n)← Rd(n+1),C←Rd(0)	Z,C,N,V	1
ASR	Rd	Arithmetic Shift Right	Rd(n) ← Rd(n+1), n=0..6	Z,C,N,V	1
SWAP	Rd	Swap Nibbles	Rd(3..0)←Rd(7..4),Rd(7..4)←Rd(3..0)	None	1
BSET	s	Flag Set	SREG(s) ← 1	SREG(s)	1
BCLR	s	Flag Clear	SREG(s) ← 0	SREG(s)	1
BST	Rr, b	Bit Store from Register to T	T ← Rr(b)	T	1
BLD	Rd, b	Bit load from T to Register	Rd(b) ← T	None	1
SEC		Set Carry	C ← 1	C	1
CLC		Clear Carry	C ← 0	C	1
SEN		Set Negative Flag	N ← 1	N	1
CLN		Clear Negative Flag	N ← 0	N	1
SEZ		Set Zero Flag	Z ← 1	Z	1
CLZ		Clear Zero Flag	Z ← 0	Z	1
SEI		Global Interrupt Enable	I ← 1	I	1
CLI		Global Interrupt Disable	I ← 0	I	1
SES		Set Signed Test Flag	S ← 1	S	1
CLS		Clear Signed Test Flag	S ← 0	S	1
SEV		Set Twos Complement Overflow.	V ← 1	V	1
CLV		Clear Twos Complement Overflow	V ← 0	V	1
SET		Set T in SREG	T ← 1	T	1
CLT		Clear T in SREG	T ← 0	T	1
SEH		Set Half Carry Flag in SREG	H ← 1	H	1
CLH		Clear Half Carry Flag in SREG	H ← 0	H	1

Assembly Programming

Instruction Groups

- Arithmetic and Logic Instructions
- Branch Instructions (Jumps)
- Data Transfer Instructions
- ~~Bit and Bit Test Instructions~~
- MCU Control Instructions

(ATMEL)

MCU CONTROL INSTRUCTIONS					
NOP		No Operation		None	1
SLEEP		Sleep	(see specific descr. for Sleep function)	None	1
WDR		Watchdog Reset	(see specific descr. for WDR/timer)	None	1
BREAK		Break	For On-chip Debug Only	None	N/A

Assembly Programming

Addressing Modes

Addressing modes form part of the instruction set architecture for some particular types of CPU. Some machine languages will need to refer to (addresses of) operands in memory. An addressing mode specifies how to calculate the effective memory address of an operand by using information held in registers and/or constants contained within a machine instruction.

Examples of addressing modes:

Absolute or **Direct** -- effective address given in instruction
Relative -- effective address is a given offset plus address of next instruction
Indirect -- effective address is contents of a memory at a given address
Base plus offset -- effective address is offset plus contents of specified register
Immediate -- operand given within a program code
Register -- argument within a register
Register indirect -- effective address is contents of specified register
Indexed absolute -- effective address is address given in instruction plus contents of specified index register
Autoincrement(decrement) -- base plus index plus the register is in(de)cremented

What are addressing modes of ATMega 128?

Assembly Programming

Program Line

An input line may take one of the four following forms:

```
[label:] directive [operands] [Comment]
[label:] instruction [operands] [Comment]
Comment
Empty line
```

A comment has the following form:

```
; [Text]
```

(AVRStudio Tools User Guide)

Assembly Programming
Labels

A label is usually an address identifier in programming language

Examples:

```
var1: .BYTE 1 ; reserve 1 byte to var1
table: .BYTE tab_size ; reserve tab_size bytes

label:   .EQU var1=100 ; Set var1 to 100

test:    rjmp test ; Infinite loop
```

(AVRStudio Tools User Guide)

Assembly Programming
Constants

The EQU directive assigns a value to an identifier. This identifier can then be used in later expressions. An identifier assigned to a value by the EQU directive is a constant and can not be changed or redefined.

Syntax:

```
.EQU identifier = expression
```

Example:

```
.EQU io_offset = 0x23
.EQU porta = io_offset + 2

.CSEG ; Start code segment
clr r2 ; Clear register 2
out porta,r2 ; Write to Port A
```

(AVRStudio Tools User Guide)

Assembly Programming
Variables

Examples:

```
var1: .BYTE 1 ; reserve 1 byte to var1
table: .BYTE tab_size ; reserve tab_size bytes
```

(AVRStudio Tools User Guide)

Assembly Programming
Procedures

Example:

```
        ; main program
        ...
        rcall  procedure_name
        ...
procedure_name:
        ...
        ; procedure body here
        call other_procedure_name
        ...
        ret
other_procedure_name:
        ...
        ; procedure body here
        ...
        ret
```

What is a stack? How `call` and `ret` instructions use a stack?

Assembly Programming

Macros

Example:

```
.MACRO SUBI16 ; Start macro definition
       subi @1,low(@0) ; Subtract low byte
       sbci @2,high(@0) ; Subtract high byte
.ENDMACRO ; End macro definition

.CSEG ; Start code segment
SUBI16 0x1234,r16,r17 ; Sub.0x1234 from r17:r16
```

(AVRStudio Tools User Guide)

Assembly Programming

Procedures vs. Macros

Procedure:

- slower execution (additional execution of call/ret instructions)
- smaller code (appears only in one location in the object code)

Macro:

- faster execution (it is pasted into a code, immediately executed)
- larger code (appears in many places in the object code)

Assembly Programming

Directives

The Assembler supports a number of directives. The directives are not translated directly into opcodes. Instead, they are used to adjust the location of the program in memory, define macros, initialize memory and so on. An overview of the directives is given in the following table.

(AVRStudio Tools User Guide)

Directive	Description
BYTE	Reserve byte to a variable
CSEG	Code Segment
CSEGSIZE	Program memory size
DB	Define constant byte(s)
DEF	Define a symbolic name on a register
DEVICE	Define which device to assemble for
DSEG	Data Segment
DW	Define Constant word(s)
ENDM	End macro
EQU	Set a symbol equal to an expression
ESEG	EEPROM Segment
EXIT	Exit from file
INCLUDE	Read source from another file
LIST	Turn listfile generation on
LISTMAC	Turn Macro expansion in list file on
MACRO	Begin macro
NOLIST	Turn listfile generation off
ORG	Set program origin
SET	Set a symbol to an expression

Assembly Programming

Expressions

The Assembler incorporates expressions. Expressions can consist of operands, operators and functions.

(AVRStudio Tools User Guide)

Functions

- LOW(expression) returns the low byte of an expression
- HIGH(expression) returns the second byte of an expression
- BYTE2(expression) is the same function as HIGH
- BYTE3(expression) returns the third byte of an expression
- BYTE4(expression) returns the fourth byte of an expression
- LWRD(expression) returns bits 0-15 of an expression
- HWRD(expression) returns bits 16-31 of an expression
- PAGE(expression) returns bits 16-21 of an expression
- EXP2(expression) returns 2 to the power of expression
- LOG2(expression) returns the integer part of log2(expression)

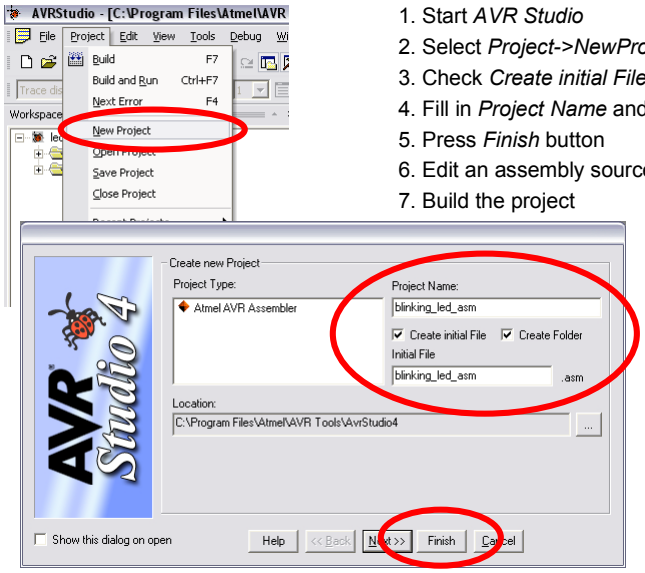
Operators

Symbol	Description
!	Logical Not
~	Bitwise Not
-	Unary Minus
*	Multiplication
/	Division
+	Addition
-	Subtraction
<<	Shift left
>>	Shift right
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal
==	Equal
!=	Not equal
&	Bitwise And
^	Bitwise Xor
|	Bitwise Or
&&	Logical And
||	Logical Or

Is the '+' operator translated into `add` opcode?

Assembly Programming

Assembly Project in AVR Studio



1. Start *AVR Studio*
2. Select *Project->NewProject*
3. Check *Create initial File* and *Create Folder*
4. Fill in *Project Name* and *Initial File*
5. Press *Finish* button
6. Edit an assembly source file
7. Build the project

Assembly Programming

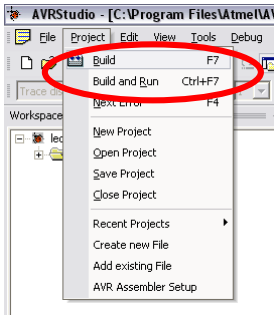
Assembly Project in AVR Studio

1. Start *AVR Studio*
2. Select *Project->NewProject*
3. Check *Create initial File* and *Create Folder*
4. Fill in *Project Name* and *Initial File*
5. Press *Finish* button
6. Edit an assembly source file
7. Build the project

Example:

```
;;2005(C) Piotr M. Szczypinski
;;Microprocessor Systems Lab
;;Assembly programming example
;;Blinking Led in Assembly

.equ    DDRB    =$37
.equ    PORTB   =$38
.equ    SPH     =$3E
.equ    SPL     =$3D
.equ    RAM_TOP =$10FF
.equ    DELAY   =$20

;;Everything starts here
START:
;;Stack initialization
    LDI R16, HIGH(RAM_TOP)
    OUT SPH, R16
    LDI R16, LOW(RAM_TOP)
    OUT SPL, R16

;;Main program
    LDI R18, 16
    LDI R17, 0
    STS DDRB, R18
LOOP:
    STS PORTB, R17
    CALL DELAY_PROC
    STS PORTB, R18
    CALL DELAY_PROC
    RJMP LOOP    ;;Loops forever
```

```
;;Delay procedure
DELAY_PROC:
    LDI R20, DELAY
DELAY_LOOP1:
    LDI R19, $FF
DELAY_LOOP2:
    LDI R16, $FF
DELAY_LOOP3:
    NOP
    NOP
    NOP
    DEC R16
    BRNE DELAY_LOOP3
    DEC R19
    BRNE DELAY_LOOP2
    DEC R20
    BRNE DELAY_LOOP1
    RET
```



Assembly Programming

C and Assembly Language (*WinAvr*)

In C language **asm** keyword can be used to place assembly language statements in the middle of a C source code.

Assembly language statements in GCC compiler take a form:
asm("instruction [operands]");

Example:

```c
//2005 (C) Piotr M. Szczypinski
//Microprocessor Systems Lab
//assembly in C source code

#include <avr/io.h>
#include <avr/delay.h>
int main (void)
{
//  Equivalent of DDRB = 16;
    asm ("LDI R24, 16"); // Loads 16 into R24 register
    asm ("STS 0x37, R24"); // Sets DDRB (address 0x37) register

    while(1)
    {
        int i;
        PORTB = 16;
        for(i = 0; i < 20; i++) _delay_loop_2(20000);
        PORTB = 0;
        for(i = 0; i < 50; i++) _delay_loop_2(20000);
    }
    return 0;
}
```

Assembly Programming

C and Assembly Language (*WinAvr*)

Procedures written in assembly language defined within a separate *.asm file can be added to a *WinAvr* project and called from a C file.

1. Create a new *.asm file and add it to the project,
2. Edit a *.asm file and define a procedure to be called from a C source code,
3. Declare a procedure within a C file,
4. List a *.asm file within the project's Makefile
5. Call a assembly procedure from a C source code

Assembly Programming

C and Assembly Language (*WinAvr*)

Example:

blinking_led.c

```c
//2005 (C) Piotr M. Szczypinski
//Microprocessor Systems Lab
//Calling a assebly procedure example

#include <avr/io.h>
#include <avr/delay.h>
extern void SET_DDRB_16(void);

int main (void)
{
    SET_DDRB_16();
    while(1)
    {
        int i;
        PORTB = 16; //s
        for(i = 0; i <
            _delay_loop
        PORTB = 0; //se
        for(i = 0; i <
            _delay_loop
    }
    return 0; //this n
}
```

set_ddrb.asm

```asm
.equ DDRB    =$37

.global SET_DDRB_16
SET_DDRB_16:
    LDI R24, 16
    STS DDRB, R24
    RET
```

Makefile

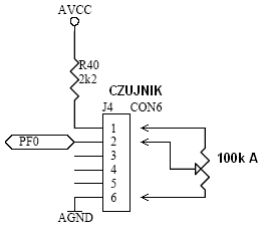
```makefile
# List C source files here. (C dependencies are automatically generated.)
SRC = $(TARGET).c


# List Assembler source files here.
# Make them always end in a capital .S.  Files ending in a lowercase .s
# will not be considered source files but generated files (assembler
# output from the compiler), and will be deleted upon "make clean"!
# Even though the DOS/Win* filesystem matches both .s and .S the same,
# it will preserve the spelling of the filenames, and gcc itself does
# care about how the name is spelled on its command-line.
ASRC = set_ddrb.asm
```

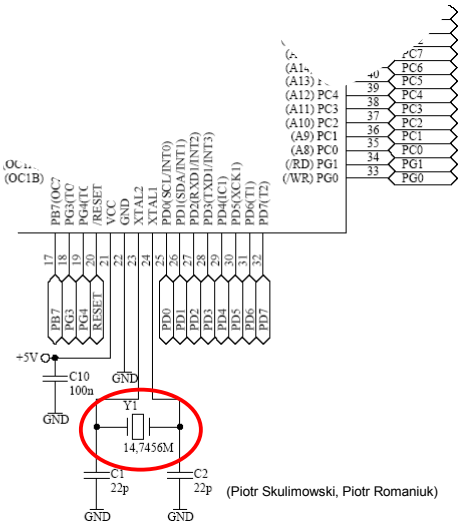
Analog-Digital Converter of ATMega 128

ADC of ATMega 128

Lecture based on documentation
of microcontroller ATMega 128 from ATMEL® .

1. Analog signal connector of the Kit,
2. Clock frequency,
3. ADC characteristics,
4. Analog channels & gain select,
7. ADC voltage reference selection,
5. Single conversion & free running mode,
6. Prescaling and conversion timing,
7. Conversion result,
8. ADC registers: ADMUX, ADCSRA, ADC



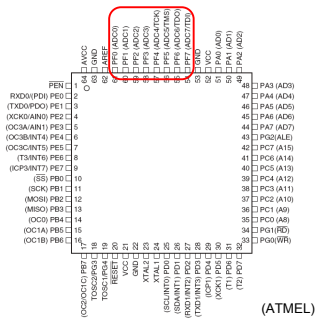


(Piotr Skulimowski, Piotr Romaniuk)

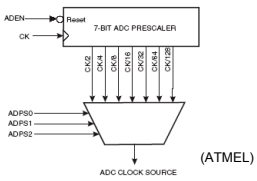
ADC of ATMega 128

Lecture based on documentation
of microcontroller ATMega 128 from ATMEL® .

1. Analog signal connector of the Kit,
2. Clock frequency,
3. ADC characteristics,
4. Analog channels & gain select,
7. ADC voltage reference selection,
5. Single conversion & free running mode,
6. Prescaling and conversion timing,
7. Conversion result,
8. ADC registers: ADMUX, ADCSRA, ADC



(ATMEL)



(ATMEL)

ADC of ATMega 128

Lecture based on documentation
of microcontroller ATMega 128 from ATMEL® .

1. Analog signal connector of the Kit,
2. Clock frequency,
3. ADC characteristics,
4. Analog channels & gain select,
7. ADC voltage reference selection,
5. Single conversion & free running mode,
6. Prescaling and conversion timing,
7. Conversion result,
8. ADC registers:
   ADMUX, ADCSRA, ADC.

ADC Multiplexer Selection Register – ADMUX

Bit	7	6	5	4	3	2	1	0	
	REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0	ADMUX
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	

ADC Control and Status Register A – ADCSRA

Bit	7	6	5	4	3	2	1	0	
	ADEN	ADSC	ADFR	ADIF	ADIE	ADPS2	ADPS1	ADPS0	ADCSRA
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	

(ATMEL)

The ADC Data Register – ADCL and ADCH

ADLAR = 0:

Bit	15	14	13	12	11	10	9	8	
	–	–	–	–	–	–	ADC9	ADC8	ADCH
	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0	ADCL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R	R	R	R	
	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

ADLAR = 1:

Bit	15	14	13	12	11	10	9	8	
	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADCH
	ADC1	ADC0	–	–	–	–	–	–	ADCL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R	R	R	R	
	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

(ATMEL)