

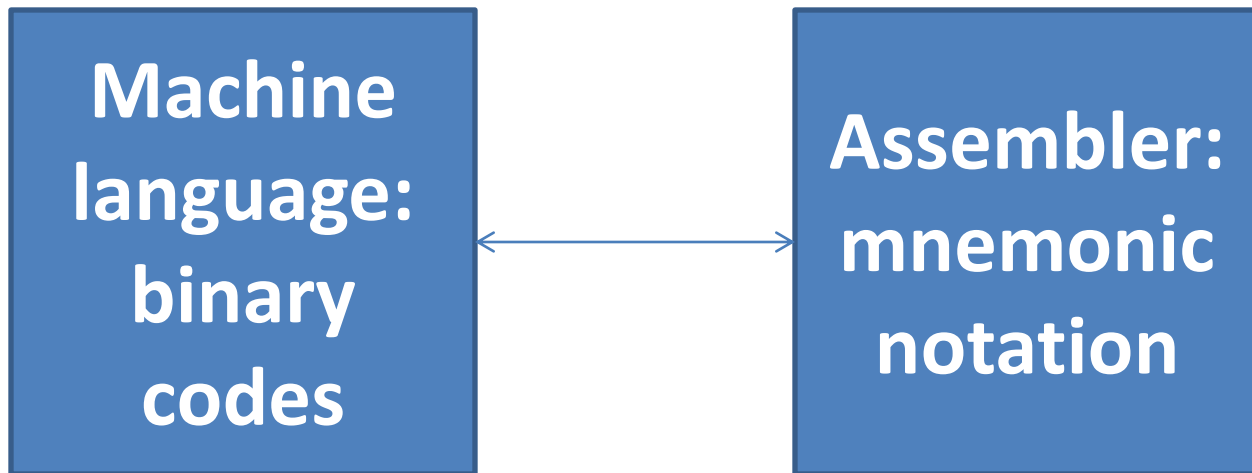
WSIZ, 2023
Computer systems architecture

Lecture 2. Introduction to programming in X86 assembler

Dmitry Zaitsev

<http://daze.ho.ua>

Machine language and assembler



48 bf 19 00 00 00 00 00 00

movq \$25, %rdi

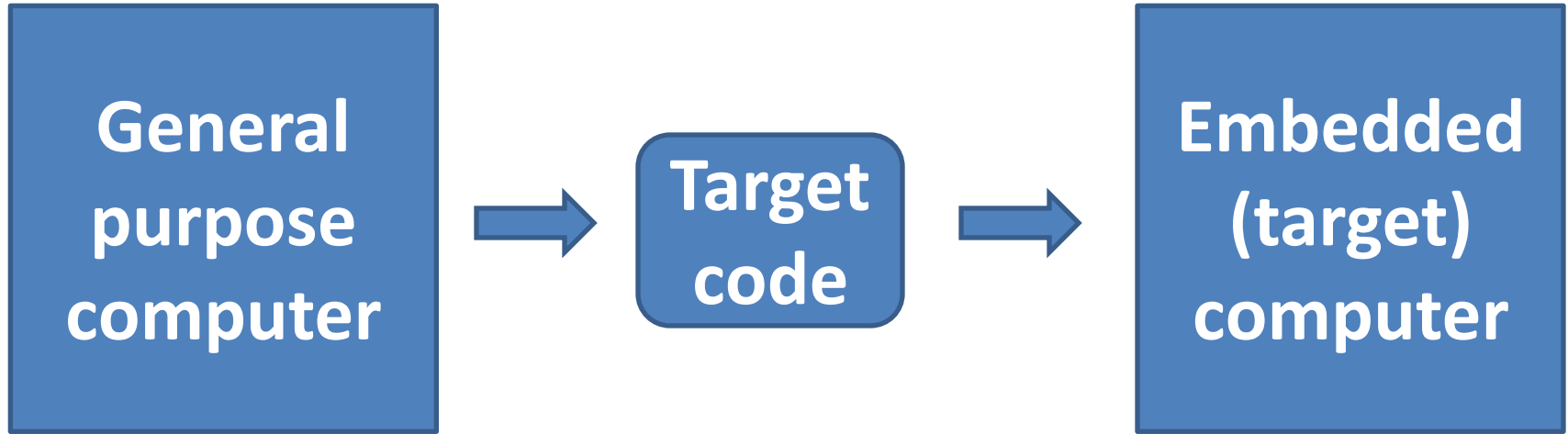
Why program in assembler?

- limited resources of embedded applications
- utmost efficiency of code – shortest time/size when optimizing software
- operating systems design
- first-time implementation of programming language

How to avoid programming in assembler?

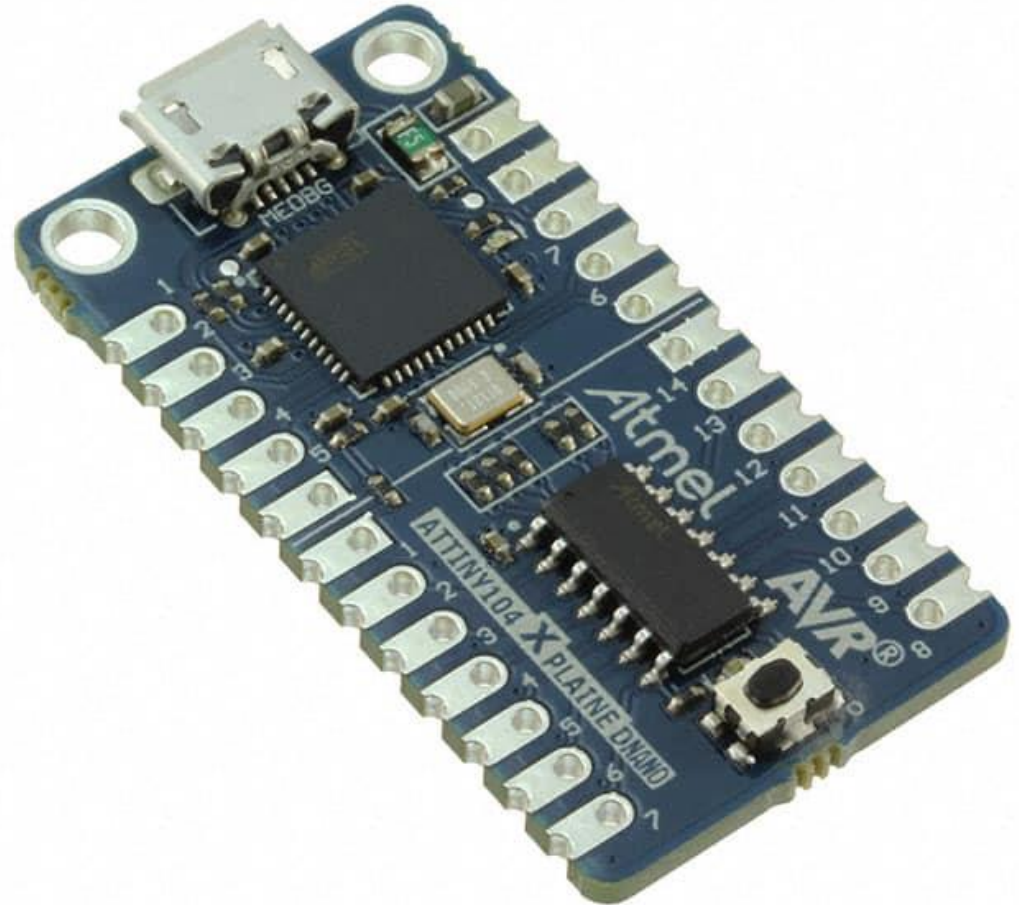
- Programming with optimization in C
- Using cross-platforms for embedded applications
- Bootstrapping compiling (self-compiling compiler) when implementing compiler in the same language

Cross-platforms for embedded applications



Example – [Microchip Studio](#)

AVR Microcontroller board



**NVIDIA Jetson
NANO – AI
powered
embedded
applications**



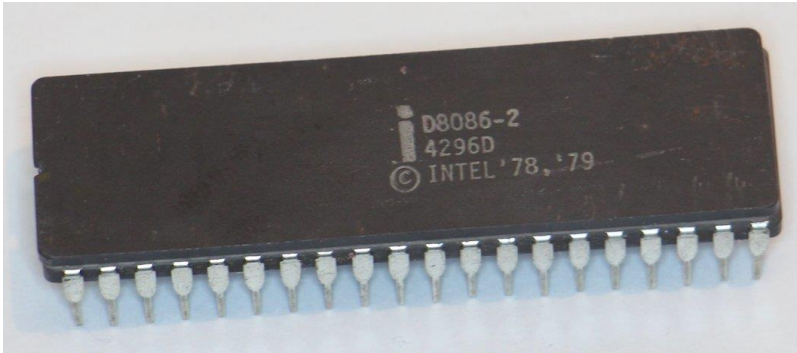
Autonomous driving vehicle JetBot



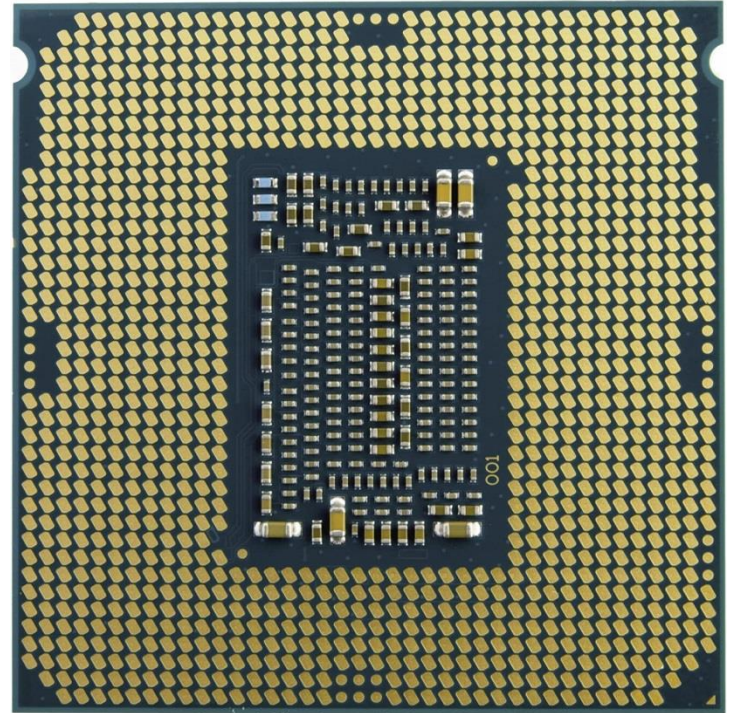
X86 assembler

- 8086 family, Intel, AMD; IBM PC, 1978
- Intel 80186, 80282, 80386, 80486, 1982-1989
- Intel Pentium, 1993
- Intel Core 2, 2006
- Intel Core i3, i5, i7

Intel processors



Intel 8086



Intel Core i9

x86 assemblers

- NASM - Netwide Assembler

```
mov eax, 4
```

- **GAS - GNU Assembler**

```
movl $4, %eax
```

Basic tools

- online compiler: <https://www.jdoodle.com/compile-assembler-gcc-online/>
- online compiler and debugger:
- https://www.onlinegdb.com/online_gcc_assembler
- dedicated assembler – as
- GNU C compiler – gcc

An example of program

%eax=%eax+%ebx, no i/o, actually 2+3

.data

.text

.global main

```
main:    mov $2, %ecx        # ecx=2
         mov $3, %ebx        # ebx=3
         add %ebx, %ecx      # ecx=2+3
         xor %eax, %ecx
         ret
```

Debug program

Online Assembler (GCC) Compile x GDB online Debugger | Compiler x +

onlinegdb.com

Run Debug Stop Share Save { } Beautify

main.S

```
1 # %eax=%eax+%ebx, no i/o
2 .data
3 .text
4 .global main
5 main: mov $2, %ecx
6      mov $3, %ebx
7      add %ebx, %ecx
8      xor %eax, %eax
9      ret
10
```

Call Stack

#	Function	File:Line
0	main	main.S:8

Local Variables

Registers

Register	Value
rbx	0x3 3
rcx	0x5 5
rdx	0x7fffffffce8 140737488350440
rsi	0x7fffffffecd8 140737488350424
rdi	0x1 1
rbp	0x1 0x1
rsp	0x7fffffffefc8 0x7fffffffefc8
r8	0x7ffff7f2f10

input Debug Console

start pause continue step over step into

8 xor %eax, %eax

(gdb)

60°F Mostly cloudy 12:03 PM 3/8/2023

Intel x86 Architecture

- General-Purpose Registers
- Index and Base Registers
- Specialized Registers
- Floating-Point, MMX, XMM Registers

General-Purpose Registers

Named storage locations inside the CPU, optimized for speed.

32-bit General-Purpose Registers

EAX
EBX
ECX
EDX

EBP
ESP
ESI
EDI

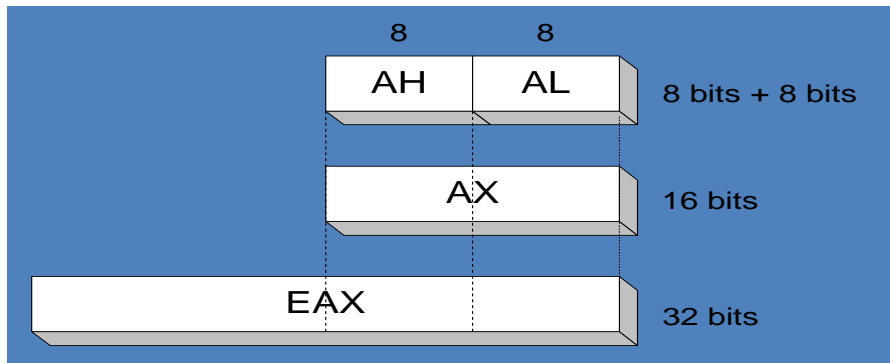
16-bit Segment Registers

EFLAGS
EIP

CS	ES
SS	FS
DS	GS

Accessing Parts of Registers

- Use 8-bit name, 16-bit name, or 32-bit name
- Applies to EAX, EBX, ECX, and EDX



32-bit	16-bit	8-bit (high)	8-bit (low)
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL

Index and Base Registers

- Some registers have only a 16-bit name for their lower half*:

32-bit	16-bit
ESI	SI
EDI	DI
EBP	BP
ESP	SP

* can't access at the byte level

Specialized Registers

- General-Purpose Registers
 - EAX – accumulator (multiplication/division)
 - ECX – loop counter
 - ESP – stack pointer
 - ESI, EDI – index registers (high speed memory transfer)
 - EBP – extended frame pointer (stack)
- Segment Registers (used for pointers)
 - CS – code segment
 - DS – data segment
 - SS – stack segment
 - ES, FS, GS - additional segments

Specialized Registers

- EIP – instruction pointer
 - Contains the address of the next instruction to be executed
- EFLAGS
 - status and control flags
 - each flag is a single binary bit

Status Flags

- Carry
 - unsigned arithmetic out of range
- Overflow
 - signed arithmetic out of range
- Sign
 - result is negative
- Zero
 - result is zero
- Auxiliary Carry
 - carry from bit 3 to bit 4
- Parity
 - sum of 1 bits is an even number

Abstract instruction



x86-64 instruction (up to 15 bytes)

- Legacy prefixes (1-4 bytes, optional)
- Opcode with prefixes (1-4 bytes, required)
- ModR/M (1 byte, if required)
- SIB (1 byte, if required)
- Displacement (1, 2, 4 or 8 bytes, if required)
- Immediate (1, 2, 4 or 8 bytes, if required)

Data movement instruction

- **mov source, destination**
- operands:
 - immediate – constant: \$3
 - register: %eax
 - memory location: num

Arithmetic instructions - 1

Addition	add source, destination	destination = destination+source
Subtraction	sub source, destination	destination = destination-source
Comparison	cmp source, destination	destination-source

Arithmetic instructions - 2

Multiplication	imul source, destination	destination = source*destination
Division	idiv divisor	result = dividend/divisor result=(remainder, quotient)

result registers for **div**

width of <i>divisor</i>	1 byte	2 bytes	4 bytes	8 bytes
<i>dividend</i>	AX	DX ◦ AX	EDX ◦ EAX	RDX ◦ RAX
<i>remainder</i> stored in	AH	DX	EDX	RDX
<i>quotient</i> stored in	AL	AX	EAX	RAX

Arithmetic instructions - 1

Negation	neg arg	$\text{arg} = -\text{arg}$
Carry Arithmetic Instructions: Add with carry; Subtract with borrow	adc src, dest sbb src, dest	$\text{dest} = \text{src} + \text{CF} + \text{dest}$ $\text{dest} = \text{dest} - (\text{src} + \text{CF})$
Increment and decrement	inc arg dec arg	$\text{arg} = \text{arg} + 1$ $\text{arg} = \text{arg} - 1$

Suffices of instructions

- b (byte) — 1 byte,
- w (word) — 2 bytes,
- l (long) — 4 bytes,
- q (quad) — 8 bytes.

Input-output implementation

- hardware: ports, interrupts
- system calls – Linux syscall(...)
- **library functions calls - libc**

Print message using libc

```
message:    .data
            .global data
            .asciz "Hello, WSIZ!"      # asciz puts a 0 byte at the end

main:       .text
            .global main
            # This is called by C library's startup code
            mov    $message, %rdi      # message address
            call   puts                # puts(message)
            ret                        # Return to C library code
```

Printing message




Online Assembler (GCC) Compiler x GDB online Debugger | Compiler x +

jdoodle.com/compile-assembler-gcc-online/

```
1 .data
2
3 message:      .global data
               .asciz "Hello, WSIZ!"      # asciz puts a 0 byte at the end
4
5
6 .text
7 .global main
8 main:        # This is called by C library's startup code
9               mov     $message, %rdi    # message address
10              call    puts              # puts(message)
11              ret                       # Return to C library code
12
```

Execute Mode, Version, Inputs & Arguments

GCC 11.1.0 ▾ ☐ Interactive Stdin Inputs

Execute   

Result

CPU Time: 0.00 sec(s), Memory: 1284 kilobyte(s) compiled and executed in 0.825 sec(s)

```
Hello, WSIZ!
3
```

Activate Windows
Go to Settings to activate Windows

Type here to search 60°F Mostly cloudy 12:23 PM 3/8/2023

Print integer number using libc

```
.data
format:
.asciz "%4d\n"
.global main
.text

main:
    mov $format, %rdi
    mov $1964, %rsi
    xor %rax, %rax
    call printf
    xor %rax, %rax
    ret
```


Print integer number




Online Assembler (GCC) Compile x GDB online Debugger | Compiler x +

jdooodle.com/compile-assembler-gcc-online/

```
1 .data
2 format:
3 .asciz "%4d\n"
4 .global main
5 .text
6 main:
7     mov $format, %rdi
8     mov $1964, %rsi
9     xor %rax, %rax
10    call printf
11    xor %rax, %rax
12    ret
13
```

Execute Mode, Version, Inputs & Arguments

GCC 11.1.0 ☐ Interactive Stdin Inputs

[Execute](#)   

Result

CPU Time: 0.00 sec(s), Memory: 1436 kilobyte(s) compiled and executed in 0.824 sec(s)

1964

main.S

Activate Windows
Go to Settings to activate Windows. [Show all](#)

Earnings upcoming 12:58 PM 3/8/2023

c=a+b

```
.data
a:    .long 2
b:    .long 3
c:    .long 0
format:
      .asciz "%4d\n"
      .text
.global main
main:
      movl a, %ecx
      movl b, %ebx
      addl %ebx, %ecx
      movl %ecx, c
```

```
mov $format, %rdi
mov c, %rsi
xor %rax, %rax
call printf

xor %rax, %rax
ret
```

Arithmetic example over 1 byte integers

Unsigned, range 0..255

+	95
	234
	329

+	0	1	0	1	1	1	1	1
	1	1	1	0	1	0	1	0
1	0	1	0	0	1	0	0	1

↑
Carry bit (CF)

↑
actual result = 73 = 329 - 256

Arithmetic example over 1 byte integers - 2

Signed, range -128..127

