
x86 Introduction

Philipp Koehn

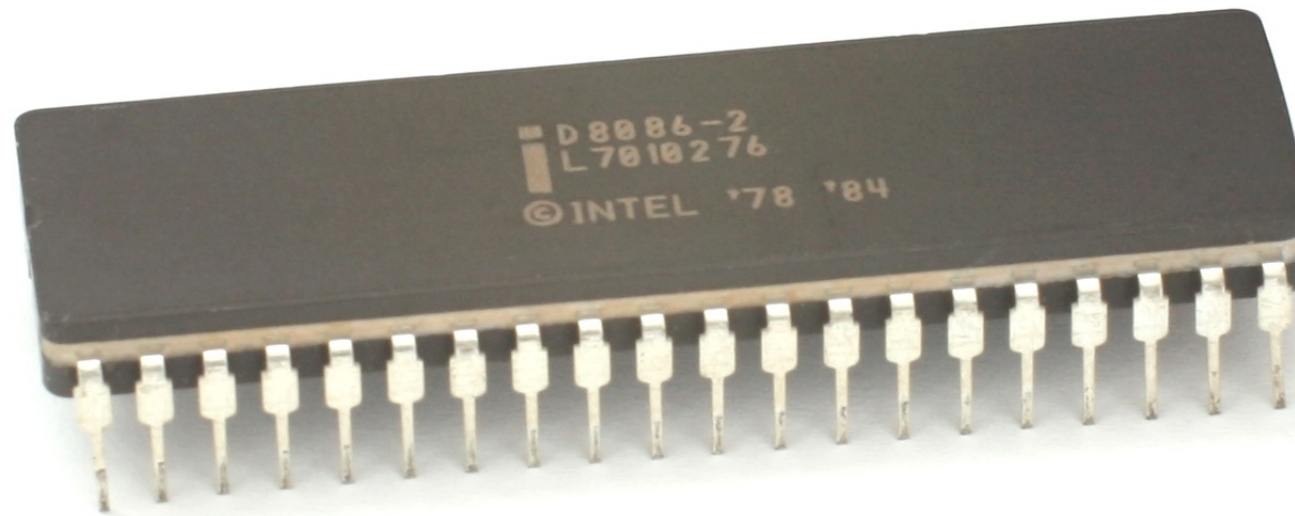
25 October 2019



- Yet another processor architecture...
- Why do we care?
- x86 is the dominant chip in today's computers (Mac, Windows, Linux)
 - 100 million chips sold per year
 - \$5 billion annual development budget
- We will focus on C programs get compiled into x86 machine code

history

8086



- 16-bit processor released in 1978 by Intel
- 8 16-bit internal registers, 20-bit address bus
- Ahead of its time, too expensive, slow sales
- 8-bit processors dominated the market

8088



- Scaled down version of 8068
- 8-bit data bus instead of 16-bit
- But looked the same from programmer's perspective
- Clock speed 4.77 MHz
- Chosen by IBM for its PC, released 1981
 - IBM PC for sale for \$1,265 (\$3,360 in 2016 dollars)
 - Apple][for sale for \$1,355 (\$3,599 in 2016 dollars)

80286

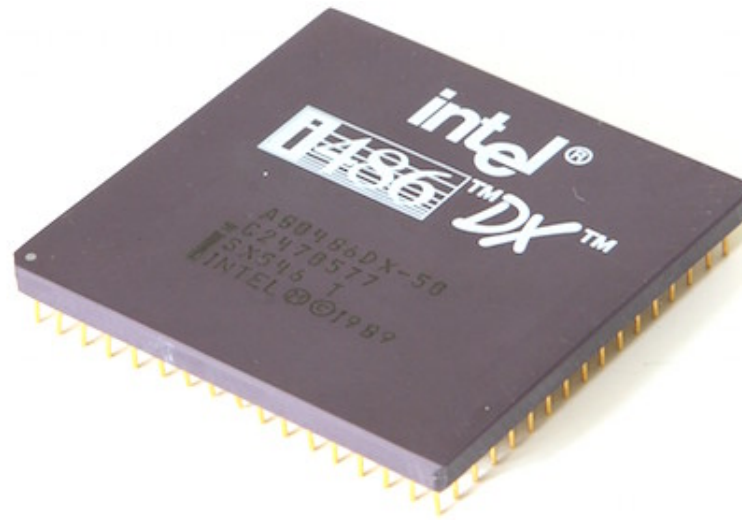


- Released by intel in 1981, used in IBM AT in 1984
- More instructions, e.g., support for multi-tasking
- Faster
 - clock speed 4.77 MHz \rightarrow 6 MHz
 - average number of cycles per instructions 12 \rightarrow 4.5
- Downward compatible: "real" mode vs. "protected" mode

386

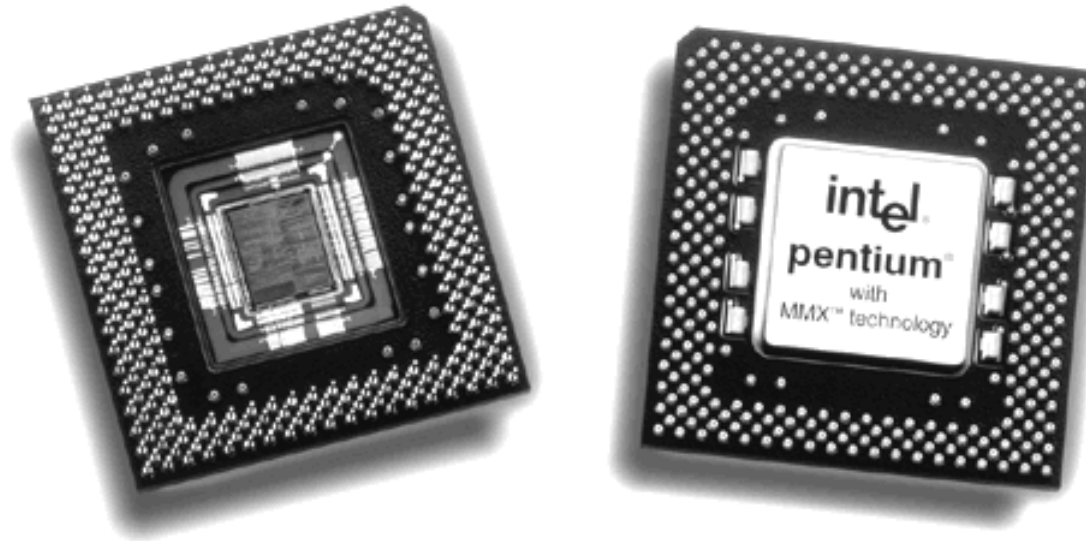


- Released in 1985, in computers late 1986, popular until early 1990s
 - 32-bit processor, but downward compatible to 286, 8086
 - Virtual real mode
 - allows different processes use different parts of memory
 - crashes do not affect whole systems
- true multi-tasking



- Up to 120 MHz
- Average number of cycles per instructions $4 \rightarrow 2$
- Internal L1 cache (hit ratio 90-95%)
- Burst memory (after initial load, 12 bytes transferred in 1 cycle)
- Internal math co-processor
- Enabled graphical user interfaces ("Windows")

586 (Pentium)



- 75–266 MHz
- 2 data paths: can execute 2 instructions in parallel
- 2 internal caches: instruction and data

And so on...



9

- 1995 Pentium Pro: Conditional move instruction
- 1997 Pentium MMX: Instructions for 64 bit vectors of integers
- 1999 Pentium III: Instructions for 128 bit vectors of floats
- 2000 Pentium 4: Double precision floating point
- 2004 Pentium 4E: 64 bit, hyper-threading of 2 processes in parallel
- 2006 Core 2: Multiple cores on chip
- 2008 Core i7: 4 cores \times 2 hyperthreading
- 2011 Core i7: 256 bit vector instructions

Today: Intel Xeon Platinum 8180M

10



- 28 cores, 56 threads
- 2.5–3.8 GHz
- 38.5 MB Cache (L1, L2, L3)
- Can address 1.5 TB RAM
- Uses 205 Watt
- List price \$8399





architecture

RISC vs. CISC

- RISC = Reduced Instruction Set Computer, e.g., MIPS
 - instructions follow simple pattern
 - for instance: no memory lookup and ALU operation in same instruction
 - allows for compact design and pipelining
- CISC = Complex Instruction Set Computer, e.g., x86
 - instructions of different complexity and length (1-15 bytes)
 - some very complex: vector operations on floats
 - complexities, but were increasingly addressed with more hardware (Intel Xeon Platinum 8180M processors have 8 billion transistors)

8 Registers

- 4 general purpose registers: AX, BX, CX, DX (16 bit)
- Stack pointer: SP
- Base pointer: BP
- Address registers: SI, DI
- 8 bit registers: AH/AL, BH/BL, CH/CL, DH/DL
- 32 bit registers: prefix with "E", e.g., EAX
- 64 bit registers: prefix with "R", e.g., RAX
8 additional registers added (R8-R15)
- Additional floating point registers: ST(0)-ST(7)

- As in 6502, operands can be registers and memory locations
- For instance addition
 - `add EAX, EBX` ; add two registers
 - `add EAX, 42` ; add value 42 to register value
 - `add EAX, [ff02]` ; add value from memory location ff02 to register
 - `add [ff02], EAX` ; as above, store result in memory
 - `add [ff02], 20` ; add 20 to value stored in memory location ff02

- Addressing modes similar to 6502
 - `mov [ff02], EAX` ; load from address ff02
 - `mov [ESP], EAX` ; load from address specified in register ESP
 - `mov [ESP+40], EAX` ; address is register value + 40
 - `mov [ESP+EBX], EAX` ; address is sum of register values
- To deal with different data sizes: scaled index
 - `mov [60+EDI*4], EAX` ; scale index register value
 - `mov [60+EDI*4+EBX], EAX` ; scale index register, add base



- Operations work on 8, 16, 32, or 64 bit data sizes
- Examples
 - add AH, BL ; 8 bit
 - add AX, BX ; 16 bit
 - add AX, -1 ; 16 bit (-1 = ffff)■
 - add EAX, EBX ; 32 bit
 - add EAX, -1 ; 32 bit (-1 = ffffffff)■
 - add RAX, RBX ; 64 bit

Data Types

17



C	Intel type	Assembly suffix	Bytes
char	byte	b	1
short	word	w	2
int	double word	l	4
long	quad word	q	8
float	single precision	s	4
double	double precision	d	8

Status Flags

- Same kind of status flags as 6502
 - CF: carry flag
 - ZF: zero flag
 - SF: sign flag
 - OF: overflow flag
- Used in conditional branches
 - jz: jump if zero
 - jc: jump if carry



instructions

- Just one command: `mov`
- Used for
 - load
 - store
 - transfer between registers
 - copy from memory to memory

Stack Operations

- Basic stack operations
 - push: place value on stack
 - pop: retrieve value from stack
- Jumps
 - call: call a subroutine (store return address on stack)
 - ret: return from sub routine

Arithmetic and Logic

- Basic math: `add`, `sub`, `mul`, `div`, `neg`
- Counter: `inc`, `dec`
- Boolean: `and`, `or`, `xor`, `not`
- Shift: `shl`, `shr`

- Compare two values: `cmp`
- Test (Boolean and): `test`
- Map flags to register: `setz, setnz, ...`
- Jump: `jmp`
- Branch: `jz, jnz, ...`
- Conditional move: `cmovz, cmovnz, ...`

Code Example: Fibonacci

- Note: 32 bit indicated by
 - l (long int) in instructions: `movl`
 - extended register names: `%eax`, `%ebx`, `%ecx`, `%edx`

```
        movl $0, %ebx           ; ebx = secondlast = 1
        movl $1, %eax           ; eax = last = 0
loop:
        cmp  $0, %ecx           ; %ecx is input value n
        jne  end                ; if n != 0 loop
        movl %eax, %edx         ; tmp = last
        add  %edx, %ebx         ; tmp += secondlast
        movl %ebx, %eax         ; shift last -> secondlast
        movl %edx, %ebx         ; shift tmp -> last
        dec  %ecx               ; n = n - 1
        jmp  loop
end:
```

Vector Operations

- 128 bit allows encoding of 4 single precision floats (32 bit each)

- Instructions that

- load vector of 4 floats into memory
- multiply each element of a vector
- store vector of 4 floats

- Example

```
movups %xmm0,[%ebx+%ebx] ; loads 4 floats in first register (xmm0)
movups %xmm1,[%eax+%ebx] ; loads 4 floats in second register (xmm1)
mulps %xmm0,%xmm1       ; multiplies both vector registers
movups [%eax+%ebx],%xmm0 ; write back the result to memory
```