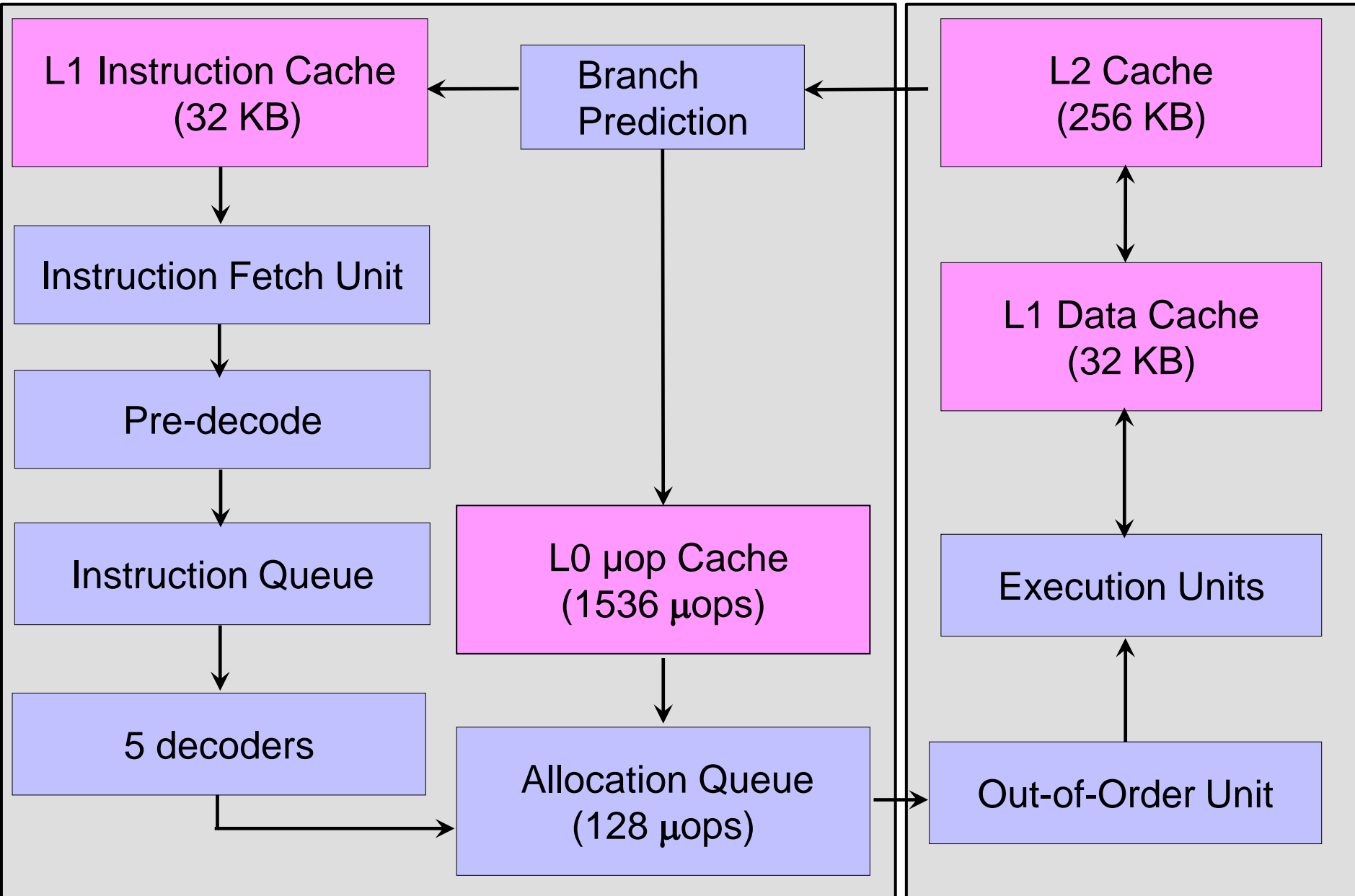


# Front-end

# Individual core

# Back-end



# Front-end

## Pre-decode

### Goal:

- identify instruction borders (variable instruction length – 1 to 17 bytes)
- decode prefixes (e.g. repetition of the string instruction, change of the default operand size, ...)

# Decoding

Processor divides the instruction into basic steps – micro-ops ( $\mu$ ops) that allow

- better parallelization and
- simpler execution units.

For example:

```
add eax, [mem1]
```

generates 2  $\mu$ ops:

1. read from memory to a temporary register,
2. add the contents of the temporary register to EAX

Instruction Fetch Unit is looking for the next instruction simultaneously in L1 instruction cache and L0  $\mu$ op cache. If the instruction is contained within the  $\mu$ op cache, the decoding step can be skipped.

## Back-end

### Out-of-order execution

The out-of-order execution logic analysis the stream of decoded instructions in Allocation Queue and schedules them for execution in whatever order best matches the available computer resources (execution units). Out-of-order execution allows execution units to be kept as busy as possible executing independent instructions that are ready to execute.

```
mov ecx,2000
```

```
mov ebx,Address
```

```
mov eax,[ebx]
```



```
mov ebx,Address
```

```
mov ecx,2000
```

```
mov eax,[ebx]
```

Two techniques enable out-of-order execution:

1. decomposition of the instruction into micro-ops

```
mov eax,[mem1]
```

```
imul eax,5; eax = eax * 5
```

```
add eax,[mem2]; generates 2  $\mu$ ops (1. read from  
memory to a temporary register, 2. add the contents  
of the temporary register to EAX)
```

Processor can fetch the value from location mem2 simultaneously with the execution of the multiplication.

## 2. register renaming

Processor has physical registers that are invisible to a programmer. Operations are performed using these „alias“ registers. Every time an instruction writes to a logical register, processor assigns a new alias register to this logical register.

In the following example, let us suppose that [mem1] is in L1 cache, [mem2] is not. Thanks to register renaming the multiplication can start before the addition (imul works with other physical register than add; old value of EAX remains in the physical register until EBX is ready for addition).

```
mov  eax,[mem1]
mov  ebx,[mem2]
add  ebx,eax
imul eax,5
```

The retirement logic reorders the instructions back to the original program order:

- The result of the instruction is stored to the destination register or to Level 1 Data Cache after all the previous instructions have been finished.
- An internal interrupt (exception) is served only if the instruction causing the exception is the oldest, non-retired operation in the machine.

This logic also reports branch history information to the branch predictors at the front end of the machine.

## Superscalar architecture

- The processor has several execution units.

Skylake has 8 ports for connection of execution units, which means, that up to 8 micro-ops can be executed in parallel. There are 27 execution units.



# Execution Units

- **computation units**
  - Arithmetic and Logic Units (ALU) - integer math, logic operations
  - separate units for integer multiplication and division
  - units for integer vector operations in SIMD technology
  - Floating Point Units (FPU)
  - branch units - handle branch instructions
- **memory units**
  - load and store address calculation (Address Generation Unit)

# Hyper-Threading Technology (HTT)

Intel introduced HTT in 2002.

HTT supports parallelization of computations. The main function of hyper-threading is to increase the number of independent instructions in the pipeline and thus to maximize the utilization of execution units in the core.

With HTT, one physical core appears as two processors to the operating system, allowing concurrent scheduling of two processes per core. If resources for one process are not available, then another process can continue if its resources are available. This technology is transparent to operating systems and programs.

Each core will see instructions from up to two threads at the same time.

# SIMD technology

Multimedia and communications applications:

- usually process 8-bit (pictures) and 16-bit (sound) data,
- frequently read and write to memory,
- the same operation (addition, multiplication) is repeated over and over for multiple pieces of data.

SIMD technology (Single Instruction Multiple Data) - the same operation is performed on independent data entries in parallel.

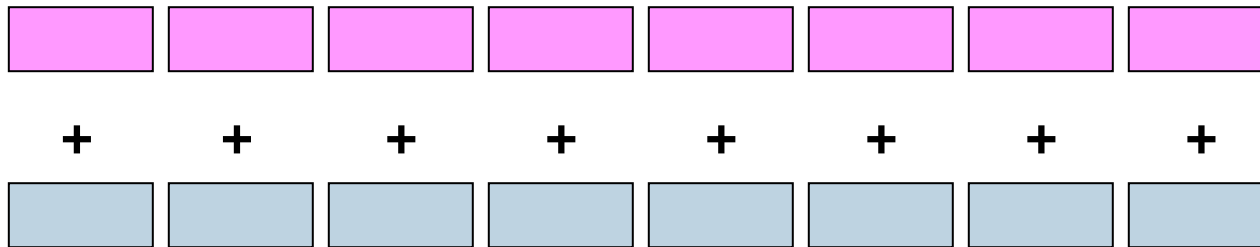
Intel: MMX, SSE, SSE2, SSE3, SSE4, AVX, AVX2

## MMX technology (MultiMedia eXtension)

An arithmetic or logic operation is performed in parallel on all entries (integers) of 64-bit operands (vectors).

MMX registers are identical with FPU registers.

64 bits = 8 bytes:



4 words:



2 doublewords:



## SSE technology (Streaming Simd Extension)

- 8 new 128-bit registers (XMM0 až XMM7)
- each register packs together four 32-bit single-precision floating point numbers
- 70 new instructions

## SSE2 technology

XMM registers can contain:

- two double-precision (64-bit) floating point numbers
- integers (bytes, words, doublewords, quadwords)

## SSE3, SSE4 technology

- new instructions

## AVX technology (Advanced Vector Extensions)

- 256-bit extension of SSE (registers YMM)
- new instructions with 3 operands

# Architecture Skylake brings new instructions

## SGX - Software Guard Extension

- new instructions that allow user-level code to allocate private regions of memory, called enclaves, that are protected from processes running at higher privilege levels

## MPX - Memory Protection Extension

- new bounds registers and instructions that allow checking pointer references at runtime

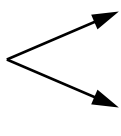
# Operating modes of a processor

## Real mode

Software has an unlimited direct access to all addressable memory, I/O addresses and peripheral hardware. Real mode provides no support for memory protection, multitasking, or code privilege levels.

**Segmentation** = memory management scheme

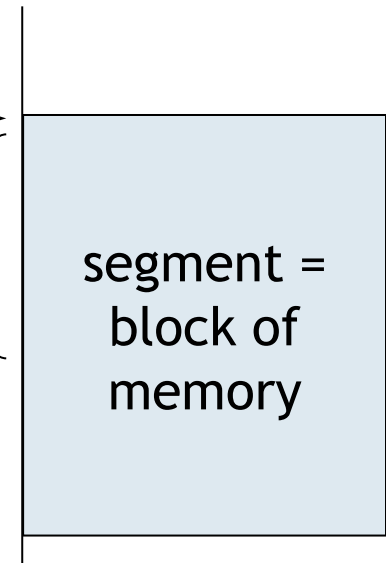
Logical  
(virtual)  
address



base address  
offset  
(displacement  
relative to the  
beginning of the  
segment)

base address

offset



## Advantages of the segmentation

- shorter addressing part of the instruction (only the offset of the operand is encoded in the instruction, the base address is in a pre-described register)
- instructions are separated from data

## Segment types

- code segment - holds machine instructions
- data segment
- stack segment - holds:
  - return addresses, parameters and local variables of procedures,
  - temporary results of mathematical operations



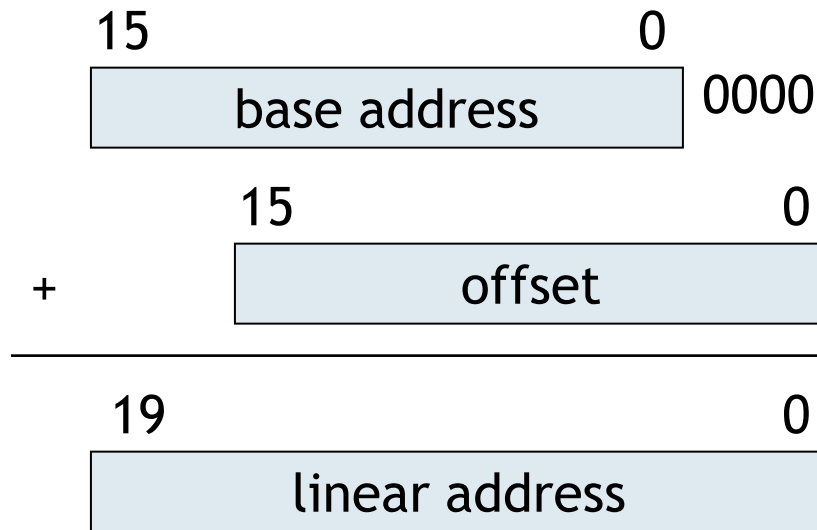
## Linear address calculation

Linear address is a combination of base address and offset.

Base address: 16 bits long

Offset: 16 bits => maximum size of a segment is  $2^{16}$  bytes = 64 kB.

Linear address = base address \* 16 + offset



Linear address

=

Physical address

## Protected mode (32-bit processor)

Base address: 32 bits, offset: 16/32 bits

Linear address = base address + offset



**Paging** - another memory management scheme.

- Allows regions of memory to be mapped to different locations in the physical address space.
- In addition, the memory access rights can be controlled, much as they can for segments (paging enables multitasking - concurrent execution of multiple processes).

## Virtual mode (32-bit processor)

- Processor runs in protected mode, but simulates real mode: a 20-bit linear address is translated by paging to a 32-bit physical address.
- A processor is switched to virtual mode when running a MS-DOS application under 32-bit Windows operating system.

## Long mode (64-bit processor)

- Operating system and applications can access 64-bit registers.
- Applications written in protected mode can run in sub-mode compatibility mode.
- 16-bit applications cannot run (DOS emulator like DOSBox is needed).

## Put it all together...

Mode		Operating system required	Type of code being run	Linear address [b]	General purpose registers [b]
Long mode	64-bit mode	64-bit	64-bit code	64	64
	Compatibility mode		32/16-bit protected mode	32	32
Legacy mode	Protected mode	32-bit	32/16-bit protected mode	32	32
	Virtual 8086 mode	32-bit	16-bit real mode	20	32
	Real mode	16-bit (starting mode for 16-, 32- and 64-bit OS)	16-bit real mode	20	16

## *Comment*

**16-bit code:** uses 16-bit general purpose registers and x86 instruction set

**32-bit code:** uses 32-bit general purpose registers and IA-32 instruction set

**64-bit code:** uses 64-bit general purpose registers and x86-64 instruction set

# Registers

- memory locations inside the processor.
- user registers - used by user applications
- system registers - used by the operating system to control the CPU

## User registers:

- general-purpose registers
- segment registers
- instruction pointer
- flags register

# General-purpose registers

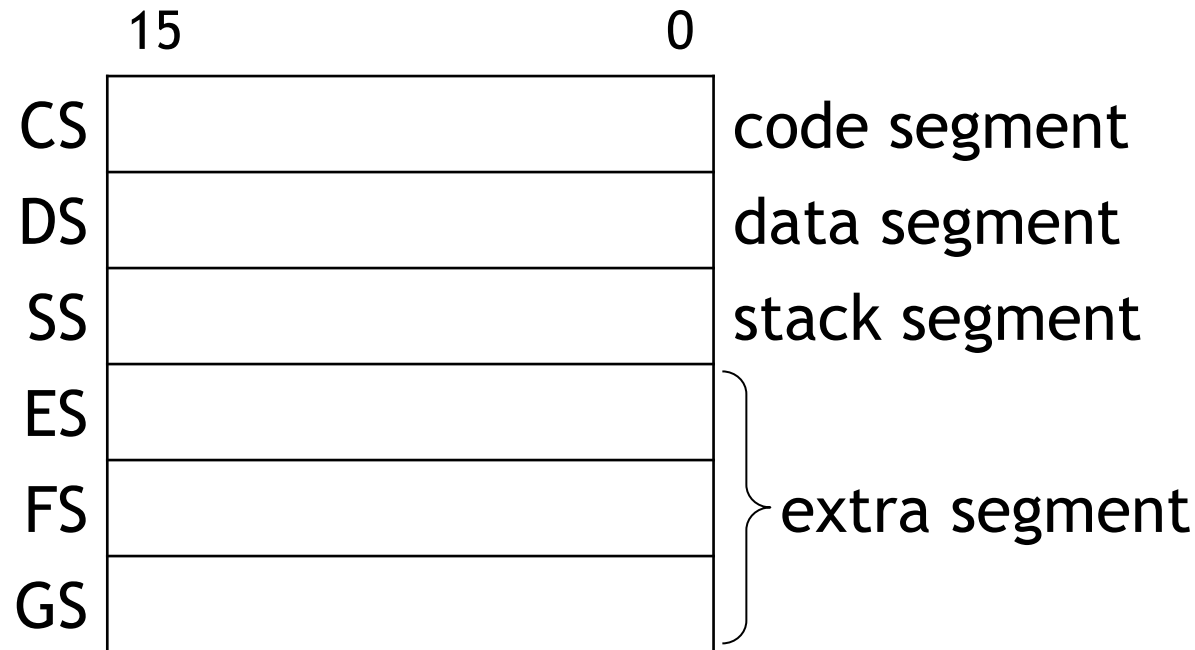
- hold instruction operands
- contain the address (offset) of an operand or participate in its calculation

	31	16	15	8	7	0
EAX		AH		AL		
EBX		BH		BL		
ECX		CH		CL		
EDX		DH		DL		
ESI				SI		
EDI				DI		
EBP				BP		
ESP				SP		

Stack  
Pointer



# Segment registers



- In **real mode** they contain the base address of a segment.
- In **protected mode** they contain indexes (selectors) to a table where segment descriptors are stored; they are initialized automatically by the operating system.

# Instruction Pointer - EIP

- 32-bit register
- EIP **points to the next instruction** that is to be fetched from memory.
- EIP is **initialized** automatically by the operating system at the moment of the program start to the offset of the first instruction.
- As one instruction is fetched from memory to the CPU, EIP is **advanced** to point to the next instruction. Jumps, procedure calls and returns change the EIP value as well.
- A programmer cannot set it directly.

Microsoft Visual Studio, Disassembly window:

```
mov edx, offset myMessage  
004033F5 mov edx,406000h
```

Microsoft Visual Studio, Registers window:

```
EIP = 004033F5
```

## Flags register

- 32-bit register
- It contains information on
  - the result of the recent arithmetic or logical operation
  - the state of the processor
  - the state of the current task

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	0	0	0	0	0	0	0	0	0						RF
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0				OF	DF	IF	TF	SF	ZF	0	AC	0	PF	1	CF

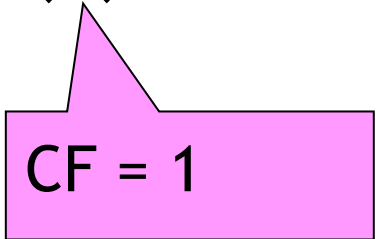
 used by the operating system

## CF (Carry Flag)

CF is set to 1 if the result of an arithmetic operation on **unsigned** numbers is too big to be held in the specified register or memory location, i.e. the operation has produced a carry from the most significant bit.

Otherwise it is set to 0.

```
mov al,0FFh; store 0FFh to register al
add al,4; add 4 to al
```

$$\begin{array}{r} 1111 \ 1111 \ (= \text{FFh}) \\ \underline{\quad \quad 100 \ (= \text{4h})} \\ (1)0000 \ 0011 \ (= \text{103h} \notin \langle 0; \text{FFh} \rangle) \end{array}$$


CF = 1

```
mov al,127
add al,4
```

$$\begin{array}{r} 0111\ 1111\ (= 127) \\ \quad \quad 100\ (= 4) \\ \hline 1000\ 0011\ (= 131 \in \langle 0; 255 \rangle) \end{array}$$

CF = 0

## OF (Overflow Flag)

It is checked after an operation on **signed** numbers. OF is set to 1 if an arithmetic operation gives a result which is out of range, i.e.

- a) the operation has produced a carry to the most significant (sign) bit but not from the most significant bit,
- b) or the operation has produced a carry only from the most significant bit.

OF is set if a carry **to** the most significant bit does not equal to the carry **from** the most significant bit.

```
mov al, -1
```

```
add al, 4
```

$$\begin{array}{r} 1111 \ 1111 \ (= \ -1) \\ \quad \quad \quad 100 \ (= \ 4) \\ \hline (1)0000 \ 0011 \ (= \ 3 \in \langle -128; 127 \rangle) \end{array}$$

OF = 0

```
mov al,127
add al,4
```

```
  0111 1111 (= 127)
    100 (= 4)
  -----
  1000 0011 (= 131 ∉ <-128;127>)
```

OF = 1



## SF (Sign Flag)

- is only useful when operating with **signed** numbers.

SF is set to 1 if the result of an operation is negative, otherwise it is set to 0.

SF takes the same value as the most significant bit of the result.

```
mov al, -1
add al, 4
```

```
1111 1111 (= -1)
      100 (= 4)
-----
0000 0011 (= 3)
```

SF = 0

```
mov al, -1
add al, -4
```

```
1111 1111 (= -1)
1111 1100 (= -4)
-----
1111 1011 (= -5)
```

SF = 1

## AC (Auxiliary Carry flag)

- is useful when operating with **decimal numbers represented in packed BCD form**.

AC is set to 1 if an arithmetic operation has produced a carry from the 3<sup>rd</sup> to the 4<sup>th</sup> bit.

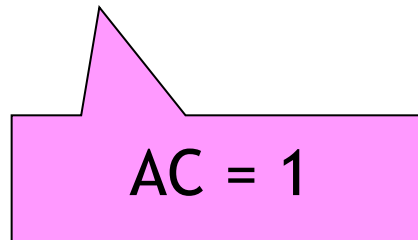
```
mov al,28h  
add al,9
```

```
0010 1000 (= 28 BCD)
```

```
0000 1001 (= 09 BCD)
```

---

```
0011 0001 (= 31?)
```



AC = 1

## ZF (Zero Flag)

ZF is set to 1 if the result of an operation is zero, otherwise it is set to 0.

```
mov al, -1
add al, 4
```

```
1111 1111 (= -1)
   100 (= 4)
-----
0000 0011 (= 3)
```

ZF = 0

```
mov al, -1
add al, 1
```

```
1111 1111 (= -1)
   1 (= 1)
-----
0000 0000
```

ZF = 1

- Calculate  $83 + 51$  in the binary numbering system ( $1010011 + 110011$ ). What will be in flags CF, OF, SF, ZF a AC?

## Parity Flag PF (Parity Even)

PF is set to 1 if the lowest byte of the result of an operation contains an even number of 1s, otherwise it is set to 0. It is mainly useful when transmitting data between devices.

## DF (Direction Flag)

DF is useful when manipulating a data array (string, vector etc.). It is set by instructions in a program.

If **DF is set to 0**, the string instruction **increments** the index by the size of the array entry and so progress is forward through memory (the string is manipulated from the left to the right).

If **DF is set to 1**, the string instruction **decrements** the index and so progress is backward through memory (the string is manipulated from the end to the beginning).

## IF (Interrupt Enable Flag)

IF is set by instructions in a program.

If IF is set to 1, interruption to normal processor operation can be initiated from I/O devices (like keyboard, mouse, disc, modem).

If IF is set to 0, external interrupts are ignored.

IF is automatically set to 0 at the entry to the interrupt service routine and its original value is restored at the exit.

## TF (Trap Flag)

## RF (Resume Flag)

These flags are useful when debugging programs.

They are set by instructions in a program (debugger).

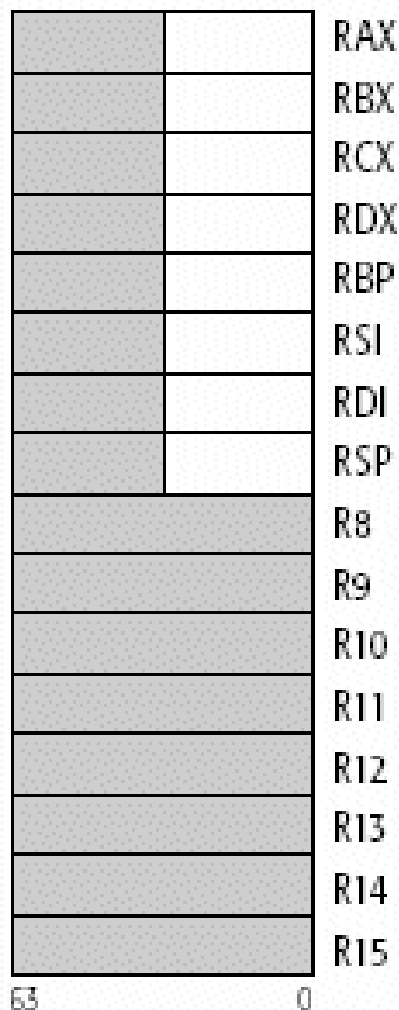
By setting TF to 1 the processor is forced to operate in single step mode in which an internal interrupt is generated after every instruction.

By setting RF to 1, a potential breakpoint on the next instruction will be ignored.

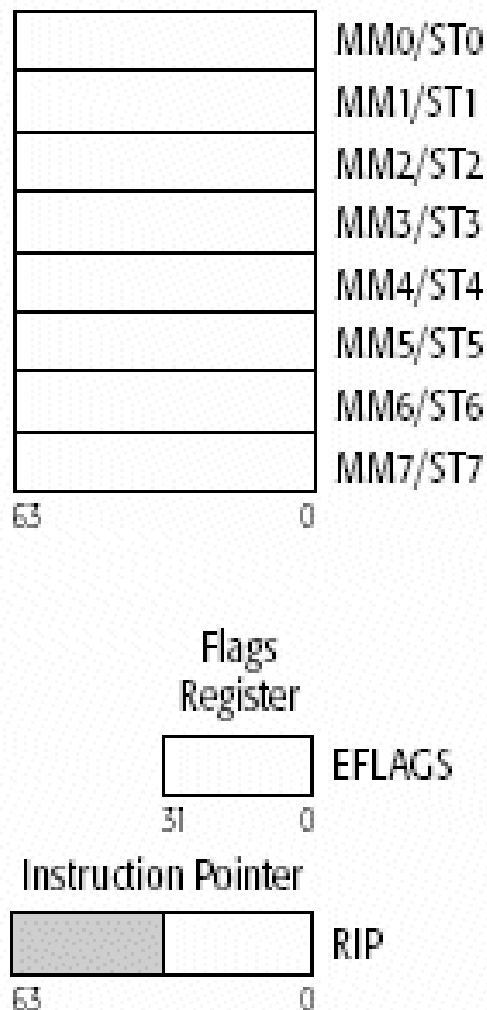
Debugger sets RF to 1 after the breakpoint service routine has finished so the program can continue from the instruction labeled by breakpoint. After the instruction has been executed, RF is set to 0.



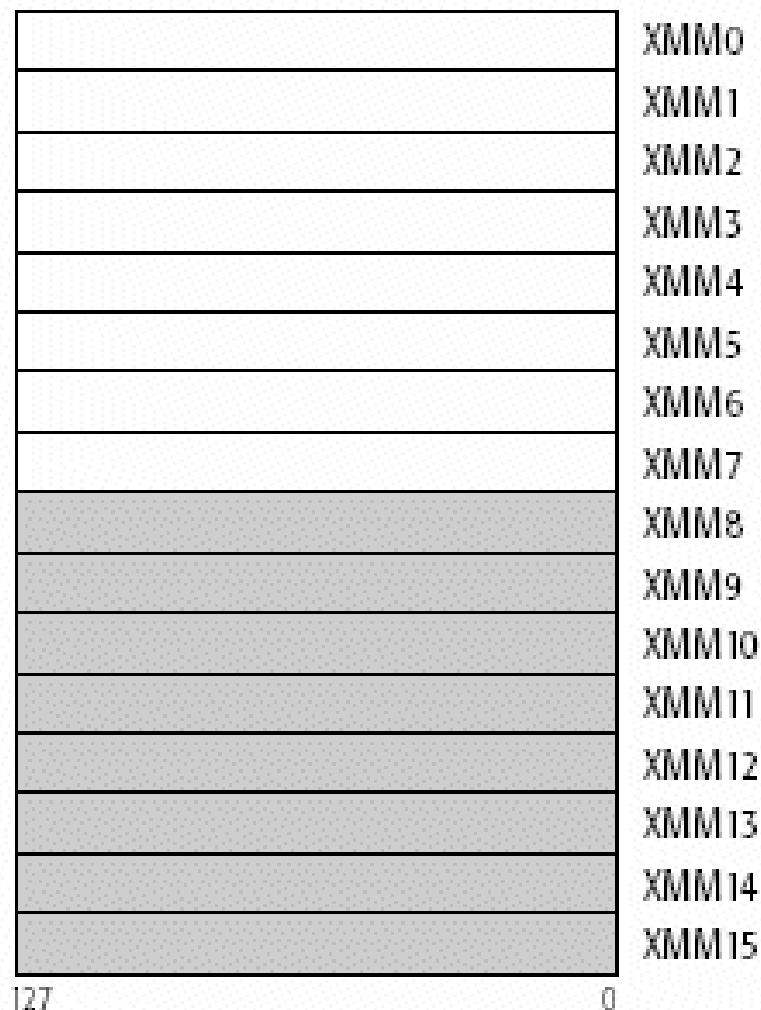
### General-Purpose Registers (GPRs)



### Multimedia Extension and Floating-Point Registers



### Streaming SIMD Extension (SSE) Registers



-  Legacy x86 Registers, supported in all modes
-  Register Extensions, supported in 64-Bit Mode