THE IA-32 PLATFORM

SEC204

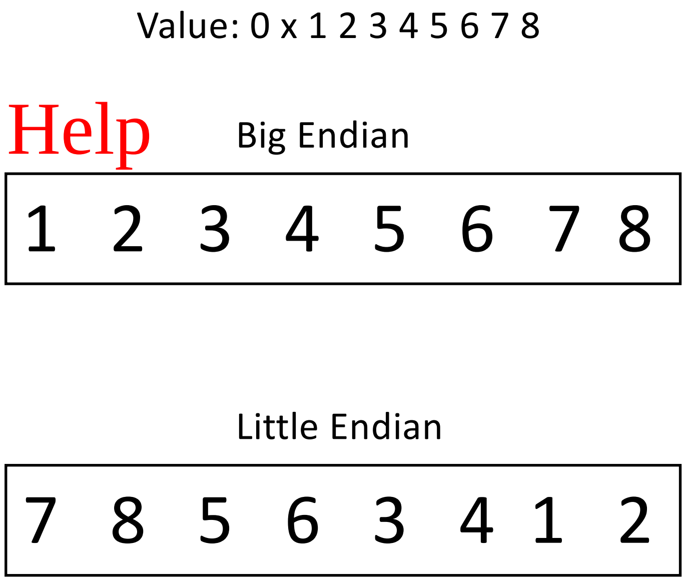
Overview

- Introduction
- Core components
- Summary

INTRODUCTION

BIG AND LITTLE ENDIAN

- Endianness was introduced by Danny Cohen in 1980. It comes from Swift's Gulliver's Travels (1726).
- Big Endian - 0x12345678 stored as is.
  - Network traffic is Big Endian
  - Most architectures (PowerPC, ARM, SPARC, MIPS) are either Big Endian by default or can be configured as either (Bi-Endian)
- Little Endian - 0x12345678 stored in RAM “little end” first. The least significant byte of a word or larger is stored in the lowest address. E.g. 0x78563412
  - Intel (IA-32) is Little Endian



Value: 0 x 1 2 3 4 5 6 7 8

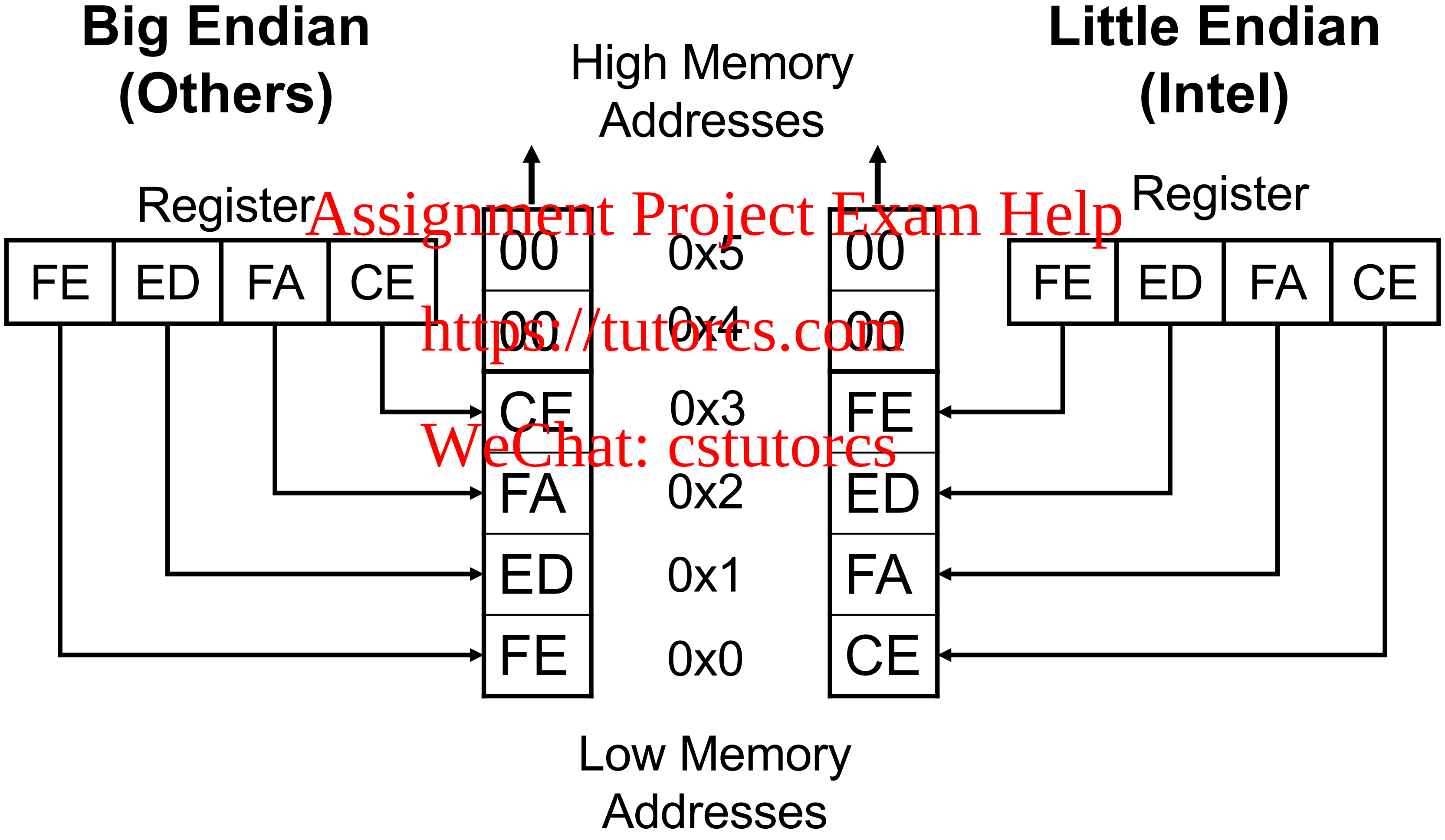
Big Endian

1	2	3	4	5	6	7	8

Little Endian

7	8	5	6	3	4	1	2

ENDIANESS


Big Endian
(Others)
Little Endian
(Intel)
High Memory
Addresses
Register
Register
FE
ED
FA
CE
00
00
CE
FA
ED
FE
0x5
0x4
0x3
0x2
0x1
0x0
00
00
FE
ED
FA
CE
FE
ED
FA
CE
Low Memory
Addresses

BIG AND LITTLE ENDIAN ACTIVITY

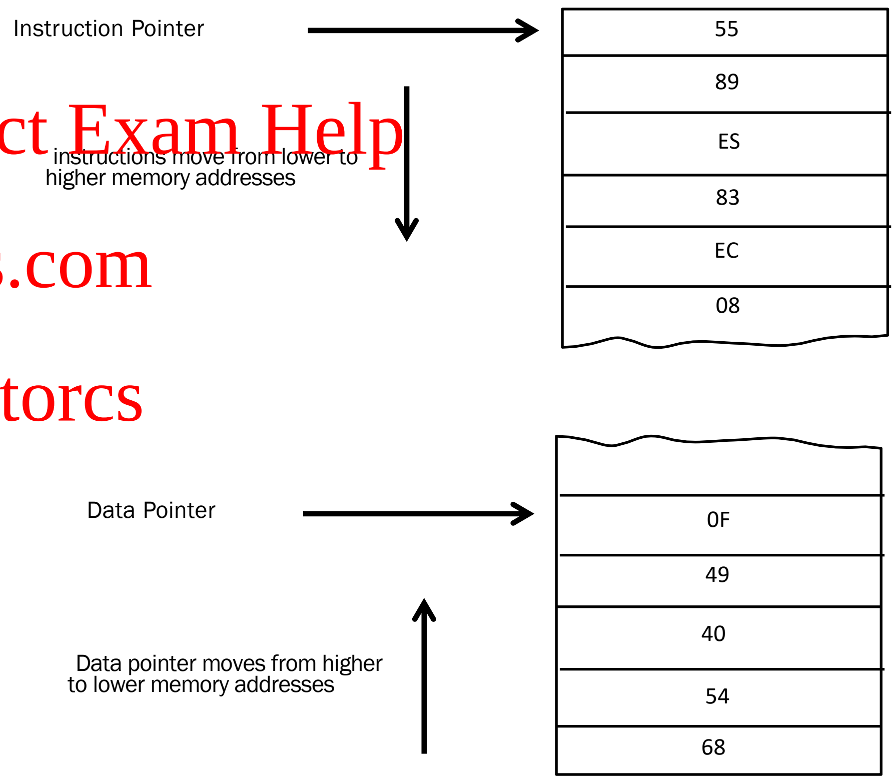
- Use an ASCII to Hex converter to type a message “HELLO WORLD!”
  - https://www.rapidtables.com/convert/number/ascii-to-hex.html

- Use an online Hex converter to see how this is represented in big/little endian environments
  - https://www.scadacore.com/tools/programming-calculators/online-hex-converter/
  - https://hexed.it/

ASCII	Hex	ASCII	HEX
Space	20	M	4D
!	21	N	4E
A	41	O	4F
B	42	P	50
C	43	Q	51
D	44	R	52
E	45	S	53
F	46	T	54
G	47	U	55
H	48	V	56
I	49	W	57
J	4A	X	58
K	4B	Y	59
L	4C	Z	5A

INSTRUCTION CODE HANDLING

- As processor runs, it reads instruction codes that are stored in memory
- As each instruction is decoded, any required data is also copied from memory
- To differentiate between instructions and data, special pointers are used
  - ***Instruction Pointer*** keeps track which instructions have already been processed and which is next to be processed
  - ***Data Pointer*** shows where the data area in memory starts.
- The *Instruction Pointer* moves *from lower to higher addresses,* whereas the *Data pointer* moves *from higher to lower addresses*



Instruction Pointer

55
89
ES
83
EC
08

instructions move from lower to higher memory addresses

Data Pointer

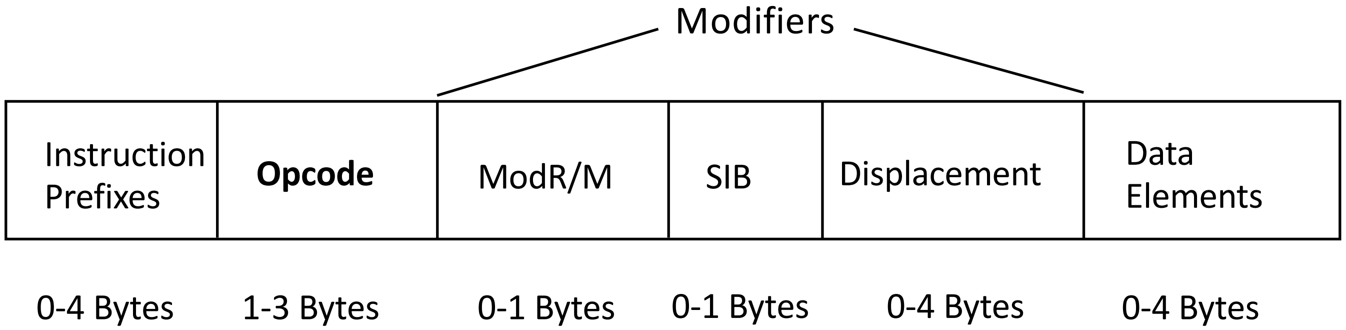
0F
49
40
54
68

Data pointer moves from higher to lower memory addresses

INSTRUCTION CODE FORMAT

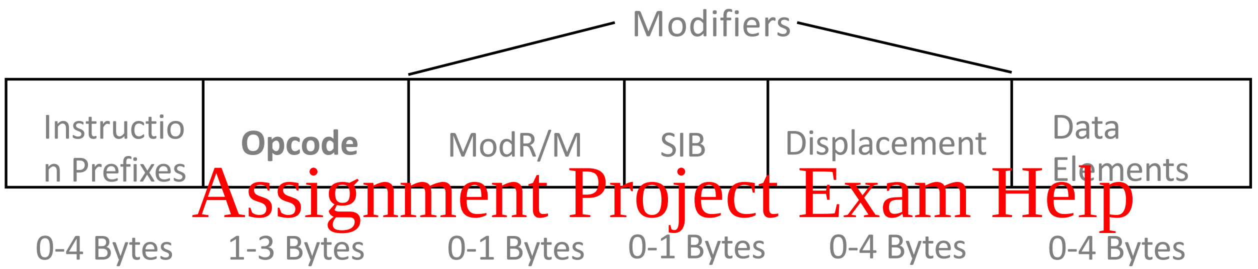
Instructions in the IA-32 family are variable-length and consist of 4 main parts

- (Optional) Instruction prefix: Modify opcode behaviour. Up to 4 prefixes, each 1 byte long
- (Required)**Operational code (opcode):** Defines the task to be performed
- (Optional) Modifier: Define what registers or memory locations are involved in the function
- (Optional) Data element: used by the function. It could be an actual static numerical value or a memory address



		Modifiers			
Instruction Prefixes	**Opcode**	ModR/M	SIB	Displacement	Data Elements
0-4 Bytes	1-3 Bytes	0-1 Bytes	0-1 Bytes	0-4 Bytes	0-4 Bytes

INSTRUCTION CODE EXAMPLE*



C7 45 FC 01 00 00 00

- **C7**: **Opcode**. Move a value to memory location

- **45 FC: Modifier**. The memory location is -4 Bytes (FC) from the memory location pointed to by the value in the EBP register (value 45).
- **01 00 00 00: Data element**. The integer value 00 00 00 01 to be placed in that memory location. (Note the little endian notation)

* There is no expectation you will be able to decode this instruction by yourself. This is only shown as an example.

OPCODE MNEMONICS

We could just about decode MARIE instructions (fixed length and format). How would you feel about having to decode IA 32 instructions yourself?

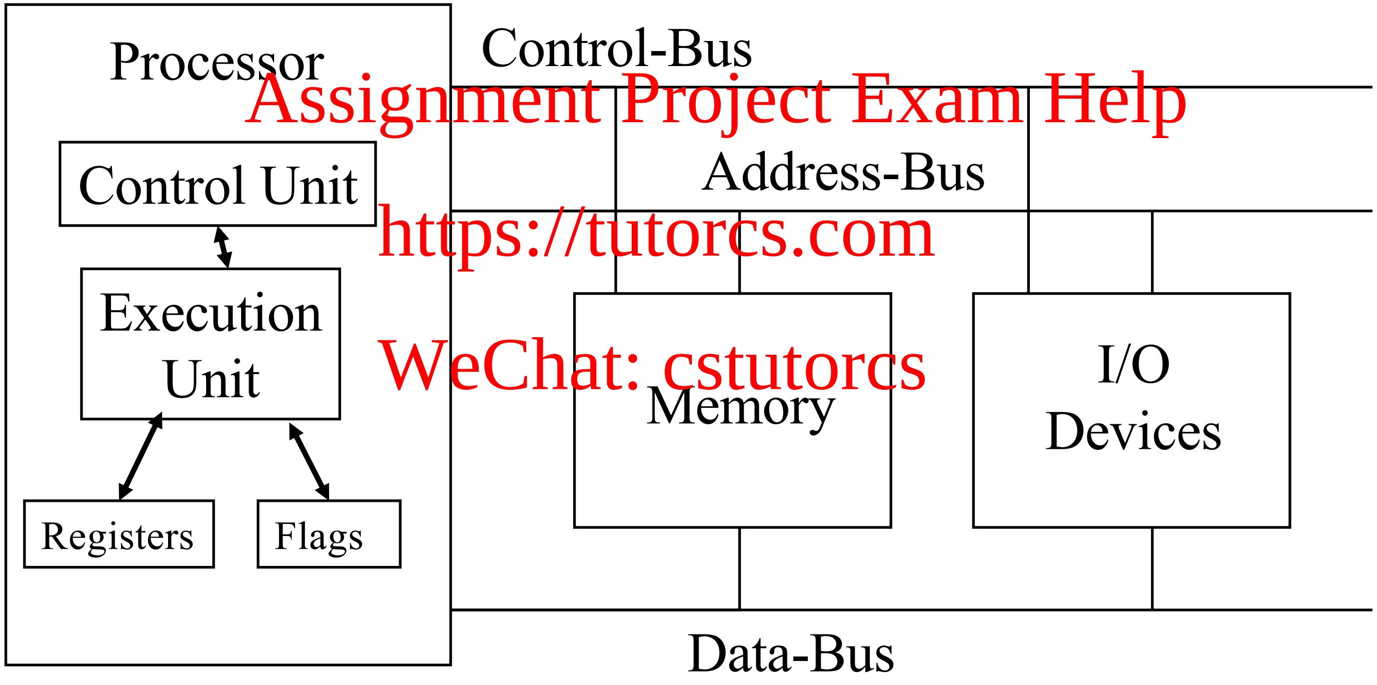
- Lets be grateful for opcode mnemonics

```
55                          push    %ebp
89 E5                       mov     %esp, %ebp
83 EC 08                    sub     $0x8, %esp
C7 45 FC 01 00 00 00        movl    $0x1, -4(%ebp)
83 EC 0C                    sub     $0xc, %esp
6A 00                       push    $0x0
E8 D1 F3 FF FF              call    8048348
```

QUESTIONS?

IA 32 CORE COMPONENTS

CORE PARTS OF AN IA-32 PROCESSOR



CONTROL UNIT

1. Retrieves instructions from memory
2. Decodes instructions for operation
3. Retrieves data from memory as needed
4. Stores results as necessary

- NetBurst functionality speeds up processing by incorporating:
  - Instruction prefetch and decoding pipeline
  - Branch prediction
  - Out-of-order execution
  - Retirement

PIPELINE EXECUTION

- Instruction execution can be divided into stages
- Pipelining makes it possible to start an instruction before completing the execution of previous one

Stages

Cycles

	S1	S2	S3	S4	S5	S6
1	**I-1**					
2		**I-1**				
3			**I-1**			
4				**I-1**		
5					**I-1**	
6						**I-1**
7	**I-2**					
8		**I-2**				
9			**I-2**			
10				**I-2**		
11					**I-2**	
12						**I-2**

Non-pipeline execution

Stages

Cycles

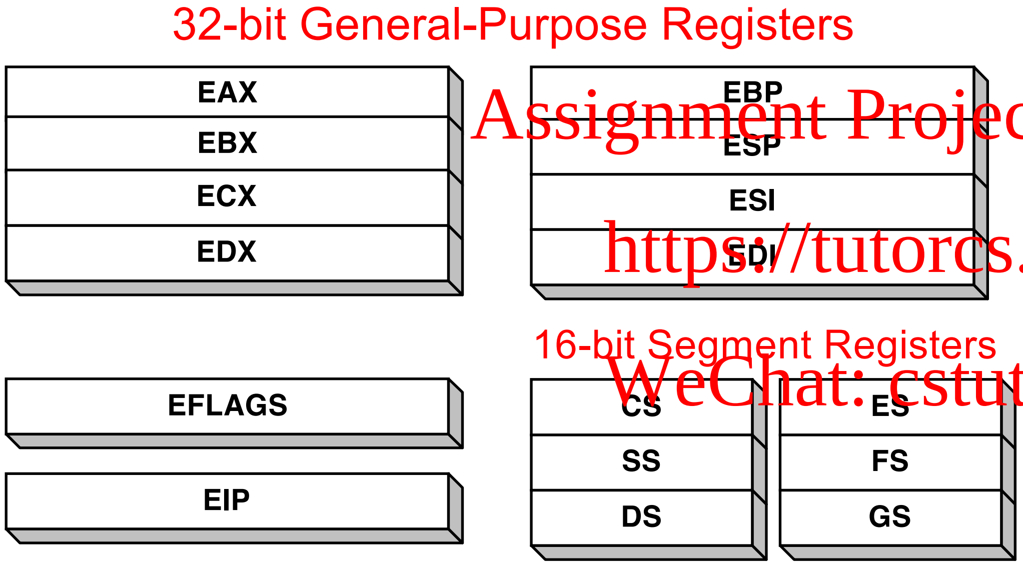
	S1	S2	S3	S4	S5	S6
1	I-1					
2	I-2	I-1				
3		I-2	I-1			
4			I-2	I-1		
5				I-2	I-1	
6					I-2	I-1
7						I-2

Pipeline Execution

EXECUTION UNIT

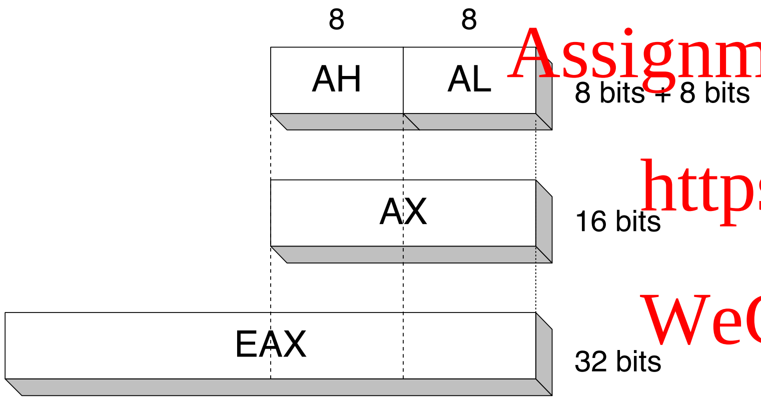
- It consists of one or more ALUs for the following functions:
  1. Simple-integer operations
  2. Complex-integer operations
  3. Floating-point operations

REGISTERS



- Registers are high speed memory inside the CPU
  - Eight 32-bit general-purpose registers
  - Six 16-bit segment registers for handling memory access
  - Instruction Pointer (EIP) pointing to the next instruction

GENERAL PURPOSE REGISTERS



REGISTER	DESCRIPTION
EAX	Accumulator for operands and results data
EBX	Pointer to data in the data memory segment
ECX	Counter for string and loop operations
EDX	I/O pointer
EDI	Data pointer for destination of string operations
ESI	Data pointer for source of string operations
ESP	Stack pointer
EBP	Stack data pointer

SEGMENT REGISTERS, CONTROL REGISTERS

Used to reference memory locations

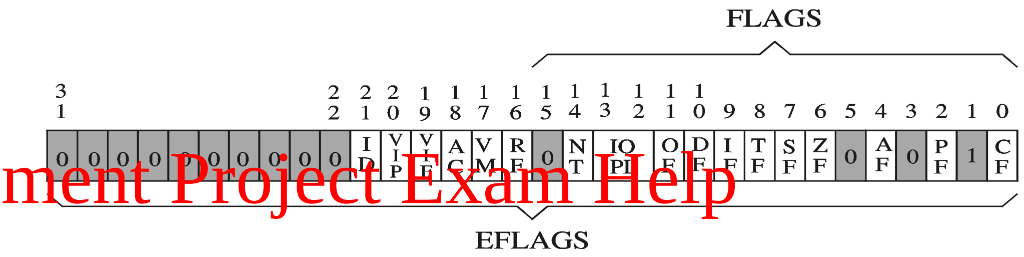
SEGMENT REGISTERS	DESCRIPTION
CS	Code Segment
DS	Data Segment
SS	Stack Segment
ES	Extra segment pointer
PS	Extra segment pointer
GS	Extra segment pointer

Used to determine the operating mode of the processor and the characteristics of the executing task

CONTROL REGISTERS	DESCRIPTION
CR0	System flags to control microprocessor operating mode
CR1	Not used
CR2	Memory page fault information
CR3	Memory page directory information
CR4	Enable processor features

EFLAGS

- Status Flags
  - Indicate status of arithmetic and logical operations
- Control flags
  - Control CPU operations
- System flags
  - Control O/S-level operations



FLAGS

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

EFLAGS

Status flags

CF = Carry flag
PF = Parity flag
AF = Auxiliary carry flag
ZF = Zero flag
SF = Sign flag
OF = Overflow flag

Control flags

DF = Direction flag

System flags

TF = Trap flag
IF = Interrupt flag
IOPL = I/O privilege level
NT = Nested task
RF = Resume flag
VM = Virtual 8086 mode
AC = Alignment check
VIF = Virtual interrupt flag
VIP = Virtual interrupt pending
ID = ID flag

STATUS FLAGS

- Carry Flag
  - Set when unsigned arithmetic result is out of range
- Overflow Flag
  - Set when signed arithmetic result is out of range
- Sign Flag
  - Copy of sign bit, set when result is negative
- Zero Flag
  - Set when result is zero
- Auxiliary Carry Flag
  - Set when there is a carry from bit 3 to bit 4
- Parity Flag
  - Set when parity is even
  - Least-significant byte in result contains even number of 1s

ACTIVITY – SPOT THE DIFFERENCES

- Work in pairs
- Compare the MARIE and IA32 architectures. Spot the differences
  - Core Components
  - Instruction Set Architecture
  - Instruction Code Handling
  - Data types

ADVANCED FEATURES

- X87 Floating Point unit (FPU)
  - perform floating-point mathematical operations

- Multimedia Extensions (MME)
  - Performs complex integer arithmetic operations often found in multimedia applications. Supports the Intel Single Instruction, Multiple Data (SIMD) execution model

- Streaming SIMD extensions (SSE)
  - Enhances performance for complex floating-point arithmetic often used in 3D graphics and motion video
- Hyperthreading
  - Handles multiple program execution threads simultaneously

SUMMARY

- The IA-32 processor consists of the control unit, the execution unit, registers and flags
- The control unit controls how the execution unit processes instructions and data
  - Prefetching and decoding instructions from memory long before execution improves performance
  - Instructions can also be processed out of order, and results are stored until needed later on
- Registers are used as local data storage within a processor
- Instructions are retrieved from memory based on the value of the Instruction Pointer Register.
- The Control Register controls the processor’s behaviour
- Instruction Pointer, Data Pointer

SUMMARY

- Little Endian, Big Endian
- Instruction Code Handling
- Pipeline execution
- Registers

FURTHER READING

- Professional Assembly Language, chapter 2
- http://ecee.colorado.edu/~ecen2120/Manual/ia32summary.pdf
- Reference information on IA 32: http://www.sandpile.org/
- Guide to programming Intel IA32 PC Architecture https://www.cs.princeton.edu/courses/archive/fall04/cos318/docs/pc-arch.html