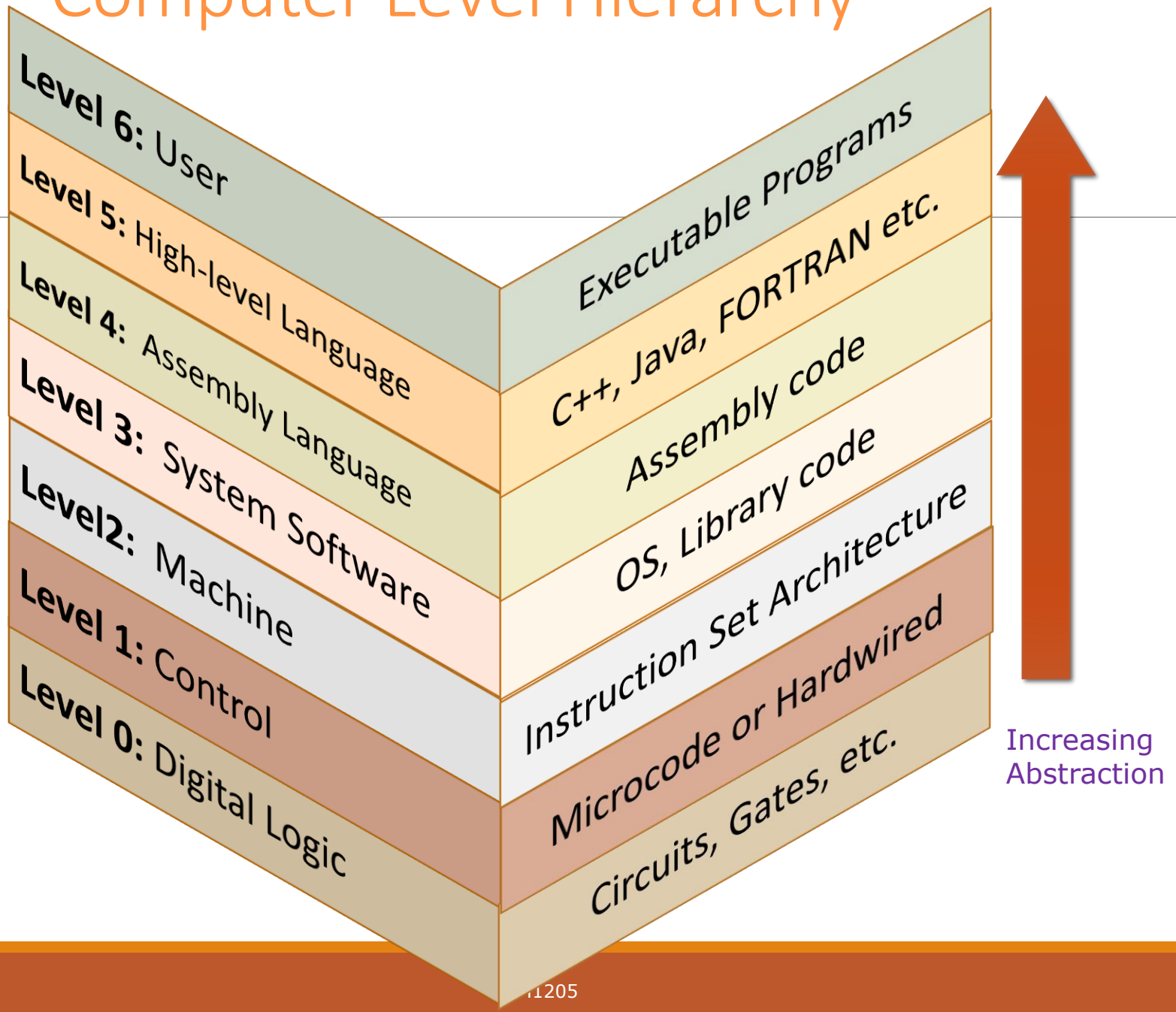
Computer Level Hierarchy



The Instruction Set Architecture Level

We will now look at an overview of the Intel 8086 and Pentium processors Instruction Set Architecture, ISA. The ISA is how the machine appears to a machine-code programmer

Interface between the software and the hardware

- Programs are translated into ISA-level “machine code” instructions
- Hardware executes these directly

A good ISA should be

- Easy to implement in hardware
  - Simple, future-proof, cost-effective…
- Easy to compile programs for
  - Regular, flexible,…

Properties of an ISA

Static properties:

- Memory model, register layout, data-types, instruction set, execution modes

Dynamic properties:

- Instruction performance and semantics

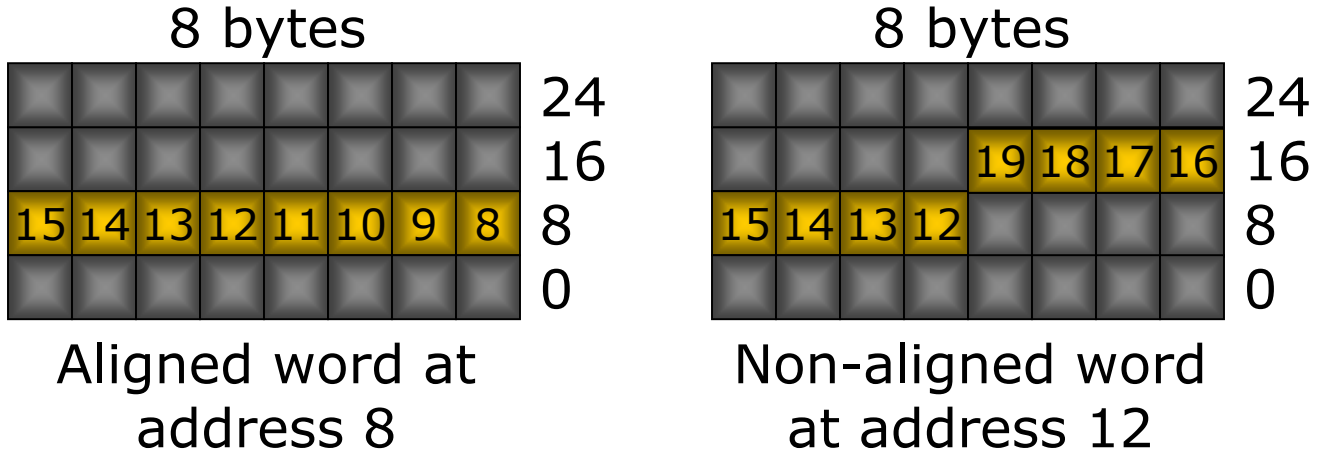
Sometimes formally defined

- E.g. V9 SPARC, JVM

Sometimes deliberately obscured (or protected)

- E.g. Intel Pentium II

Memory Models



Architecture may require words to be *aligned*

- Memory operates more efficiently this way

Pentium II has 36-bit physical addresses, but only 33 address bits are accessible

- Last 3 bits are zero

Enforced alignment can cause problems

- Two reads may be necessary to get non-aligned data

Data Organisation

One of the differences between the Intel line of microprocessors and those made by other manufacturers is Intel's way of storing numbers in memory.

The method was introduced with the 8080 processor and has continued right through the range.

It is a technique known as ***byte-swapping***.

BYTE SWAPPING

This technique is sometimes confusing for those unfamiliar with it, but becomes clear after a little exposure.

When a number larger than 1byte (8-bits) is written into the systems memory, the low byte is written into the first memory location and the high byte into the second location.

This is byte swapping.

Byte swapping is more usually referred to as

little endian/big endian.

Little Endian/Big Endian

Big Endian

- Usually MainFrame based systems and Motorola
- Big-end is stored first (Most significant byte is stored at the lowest address)
  - i.e. ebfe0000 would be stored as
  - eb fe 00 00

Little Endian

- Intel based systems **BYTE SWAP**
- Low-end stored first (most significant byte is stored at the highest address)
- i.e. ebfe0000 would be stored as:
- 00 00 fe eb

Byte Swapping con’t

Byte swapping is one of the most significant differences between Intel processors and other machines such as Motorola’s 68000, which do not byte swap.

Reading the 16-bit number out of memory is performed automatically by the processor. The 8086 knows its reading the low byte first and puts it in the correct place.

Programmers who manipulate data in memory must remember to use the proper practice of byte-swapping or discover that their programs do not give correct, or unexpected results.

Microarchitecture Level

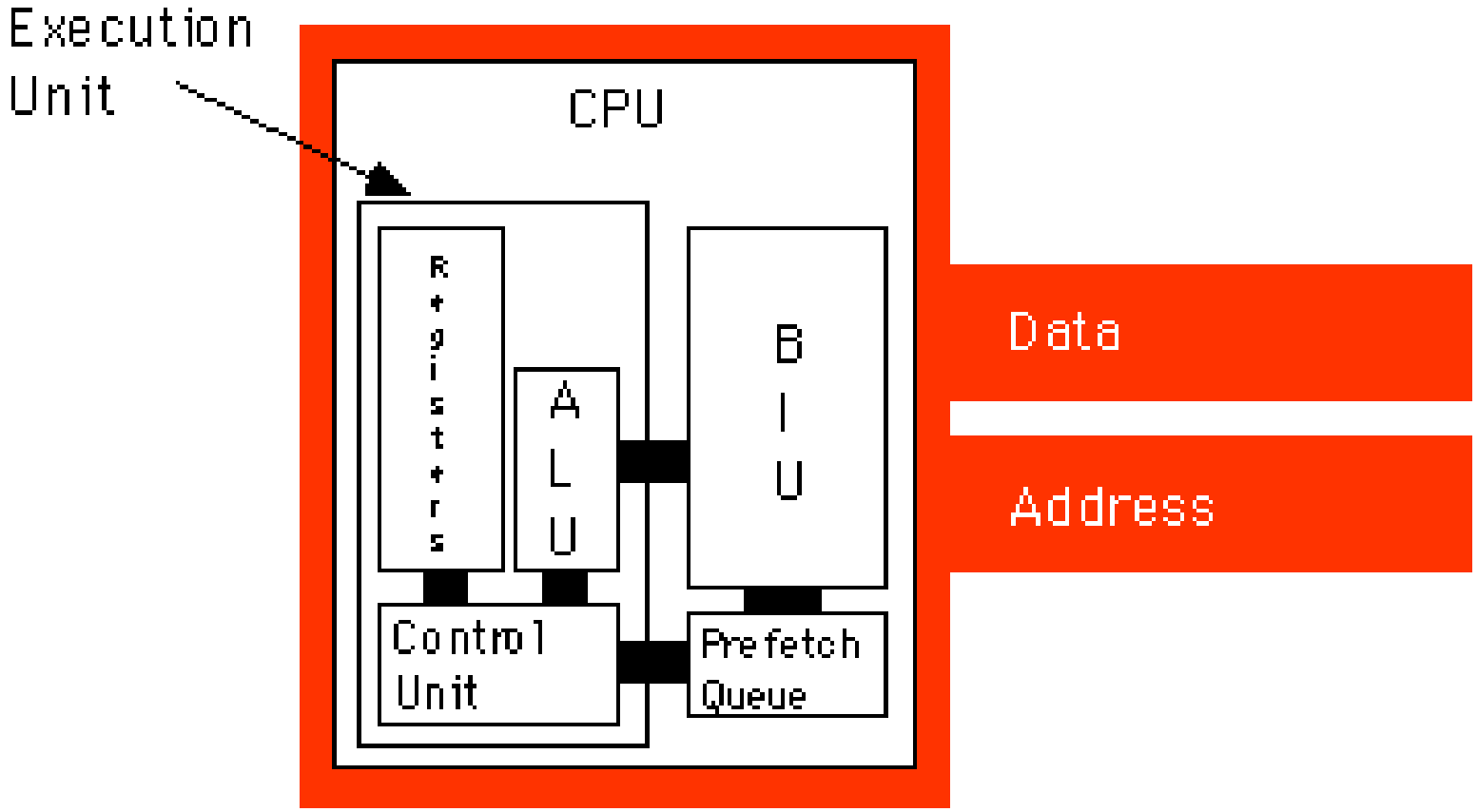
Consists of:

- Local memory (registers)
- Arithmetic Logic Unit (*ALU*)

ALU is connected to the registers by the *data path*

The data path operates by sending data from the registers to the ALU, and storing the result back in some register

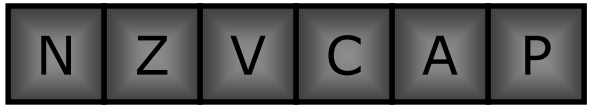
8086 Architecture



Registers

Specialised registers:

- Program counter
- Top of stack
- Program status word (PSW)



General purpose registers

- Hold temporary results (avoid accessing memory)
- Some general-purpose registers may also have specific functions

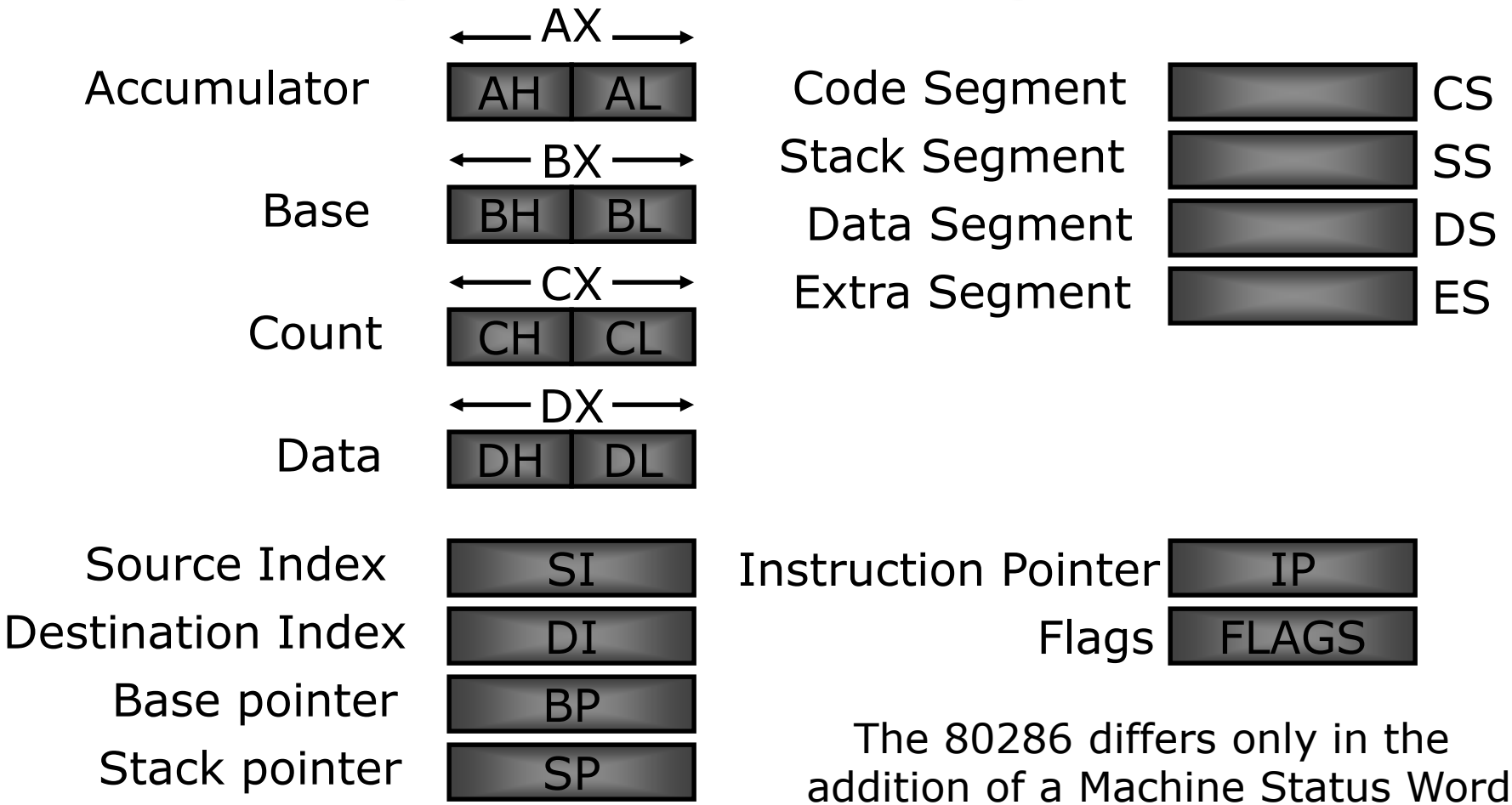
The 8086

The 8086 was INTELS first true 16-bit microprocessor.

Its internal architecture is shown on the next slide.

All of INTELS 80x86 and PENTIUM Microprocessor family are code compatible with the original 8086.

Original 8086 Registers



The 80286 differs only in the addition of a Machine Status Word and extra flag bits

The Pentium ISA

IA-32: Intel's 32-bit architecture

- Used in: 80386, 80486, Pentium, Pentium-Pro, Pentium II, Celeron and Xeon

Three main operating modes:

- *Real mode*: acts like a 8088 (program crash = crash)
- *Virtual 8086 mode*: OS informed on program crash
- *Protected mode*: full 32-bit machine

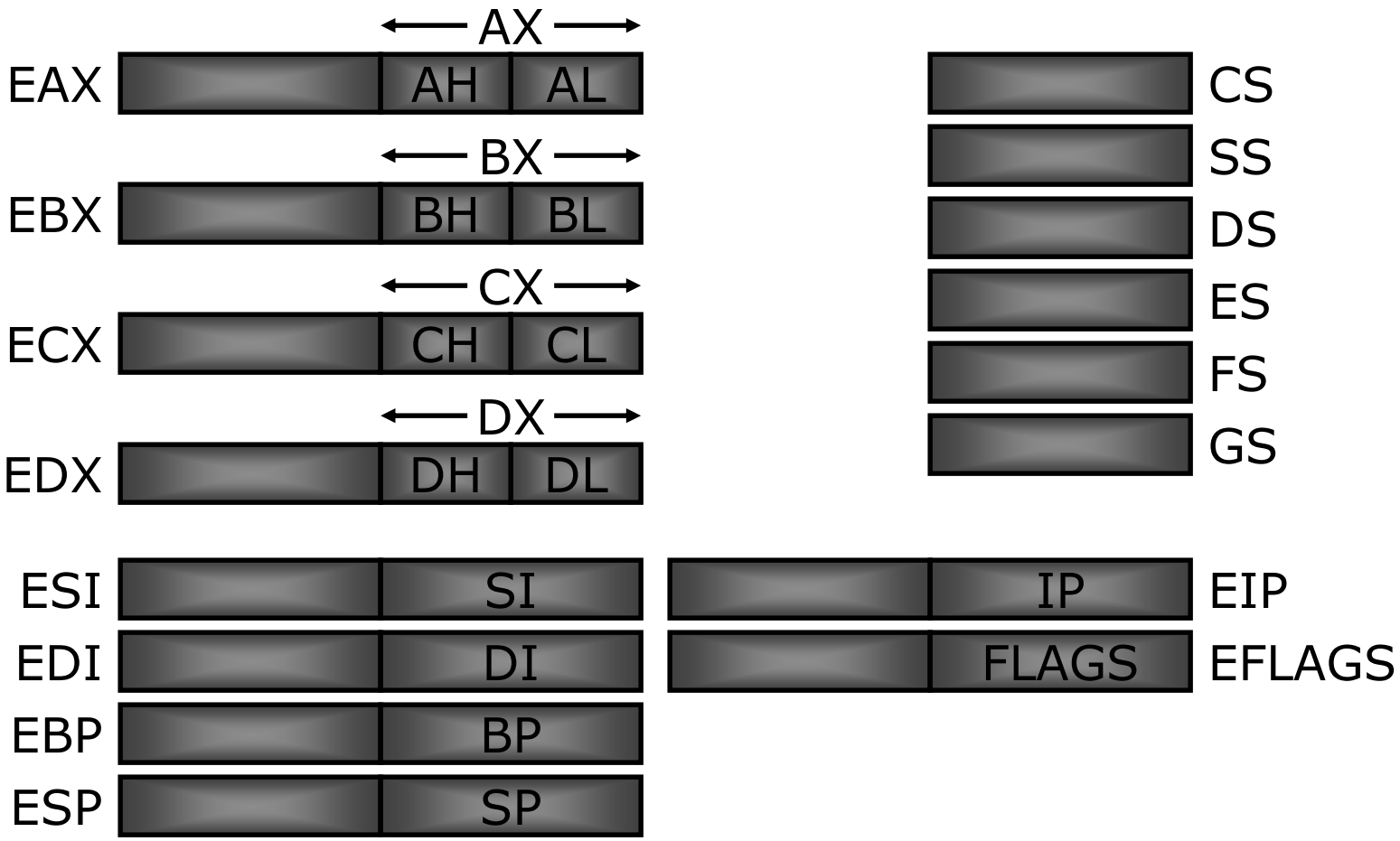
Four privilege levels:

- 0: kernel mode
- 3: user mode (modes 1 and 2 are rarely used)

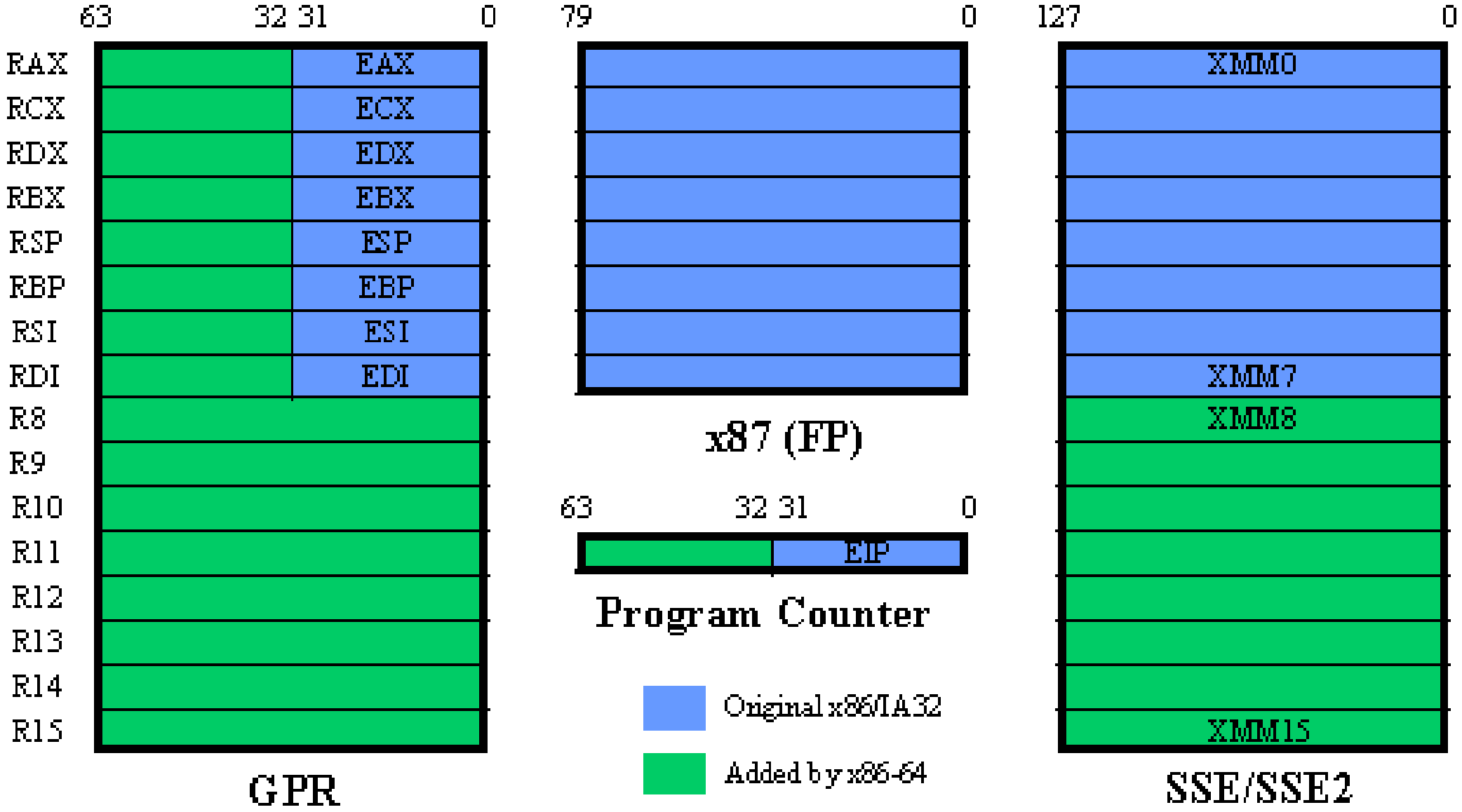
Huge address space:

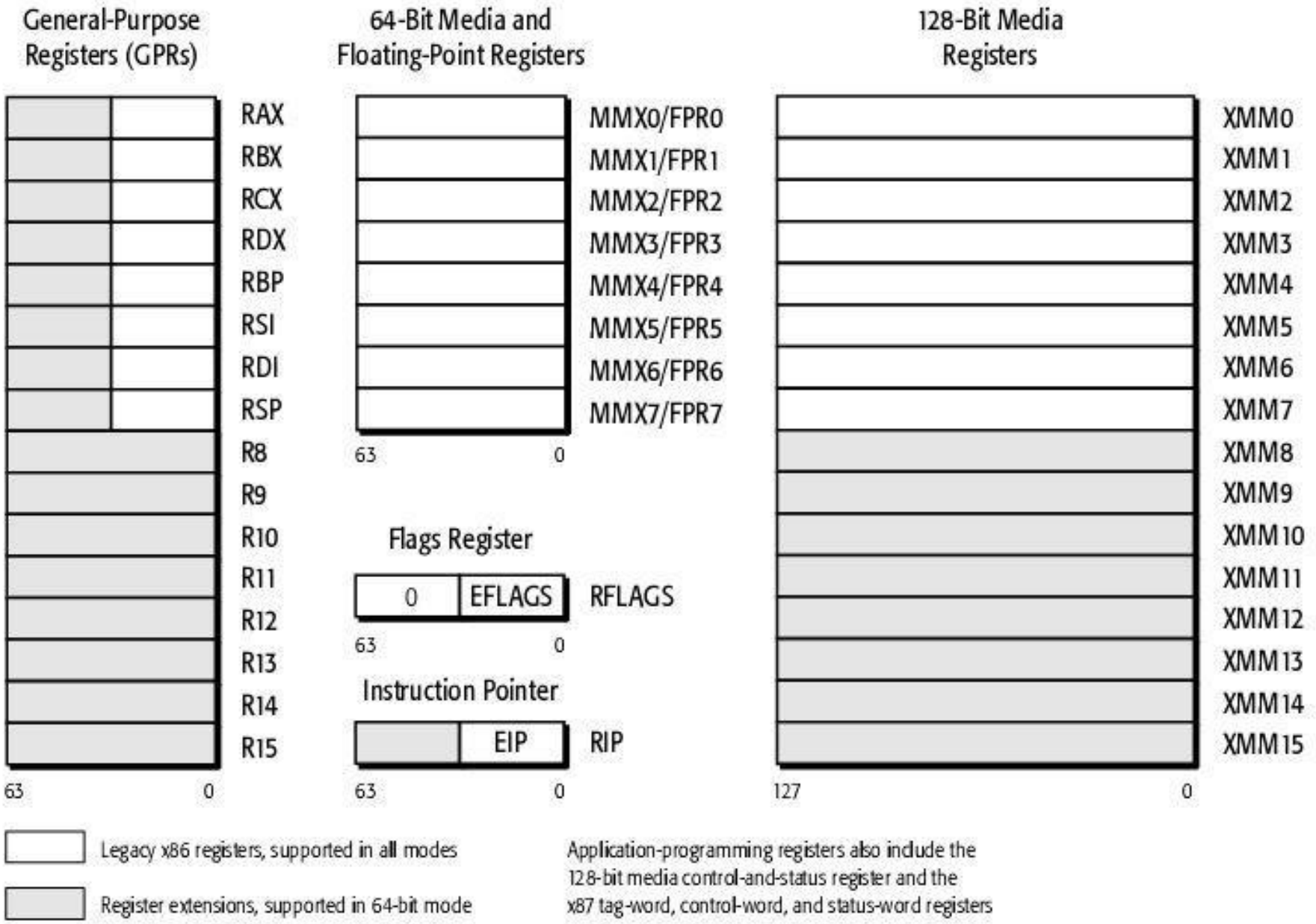
- 2^{16} segments, each with 2^{32}-1 addresses
- Windows, and Linux use only one segment

32 bit Pentium Registers



64 Bit INTEL STRUCTURE




General-Purpose Registers (GPRs)
RAX
RBX
RCX
RDX
RBP
RSI
RDI
RSP
R8
R9
R10
R11
R12
R13
R14
R15
63
0
64-Bit Media and Floating-Point Registers
MMX0/FPR0
MMX1/FPR1
MMX2/FPR2
MMX3/FPR3
MMX4/FPR4
MMX5/FPR5
MMX6/FPR6
MMX7/FPR7
63
0
Flags Register
0
EFLAGS
RFLAGS
63
0
Instruction Pointer
EIP
RIP
63
0
128-Bit Media Registers
XMM0
XMM1
XMM2
XMM3
XMM4
XMM5
XMM6
XMM7
XMM8
XMM9
XMM10
XMM11
XMM12
XMM13
XMM14
XMM15
127
0
Legacy x86 registers, supported in all modes
Register extensions, supported in 64-bit mode
Application-programming registers also include the 128-bit media control-and-status register and the x87 tag-word, control-word, and status-word registers

Assembly Language Level

This is really just a human-readable form of one of the underlying languages

Based on the ISA level

Programs in assembler need to be translated into machine language before execution

INTEL 8086 Assembly Language

We can now start to look at what 8086 instructions are available to us, and how we can use them.

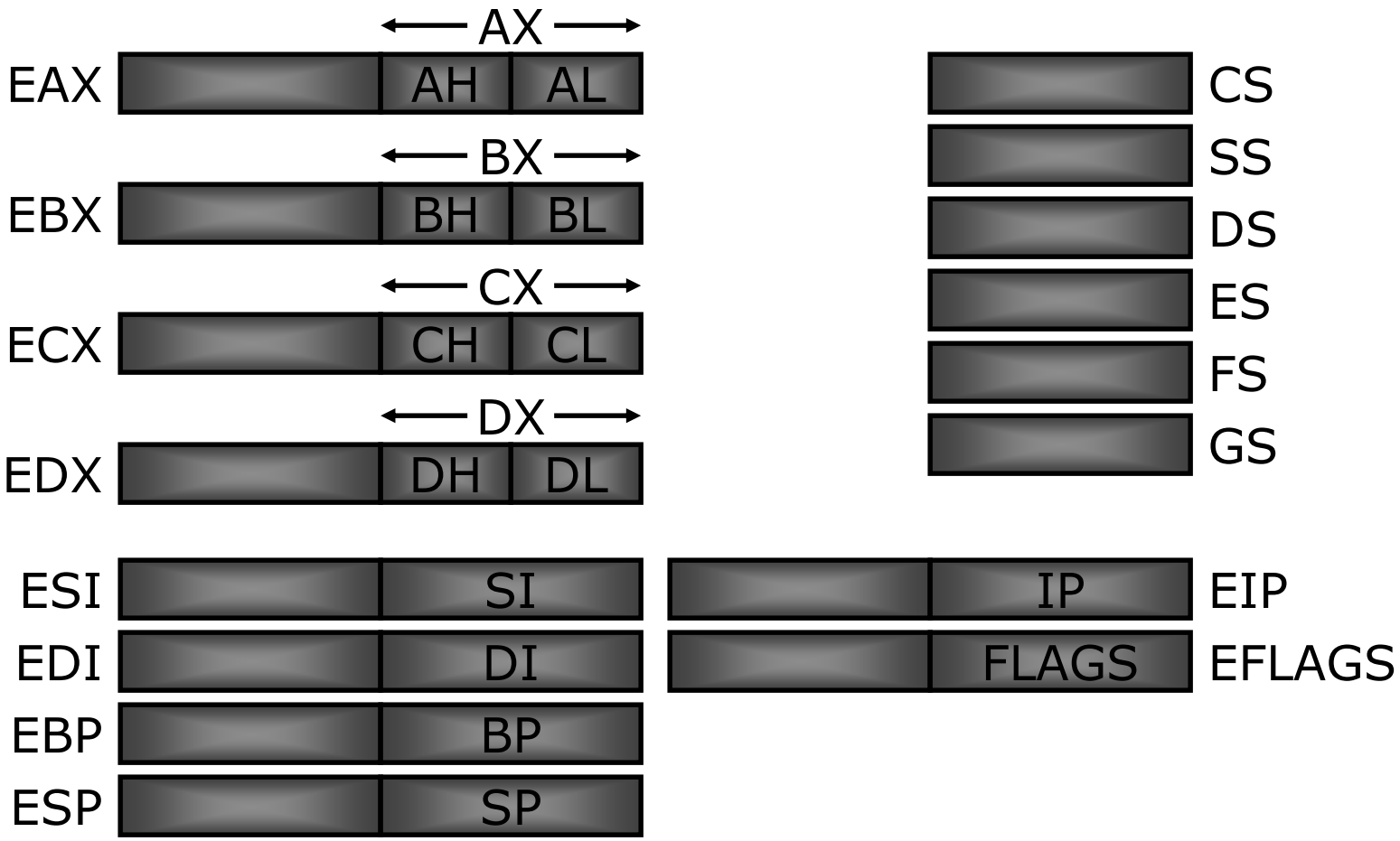
We will concentrate on the 32-bit architecture.

8086 Instructions con’t

The majority of instructions involve **Registers** in some way or other.

Registers are places where data can be processed particularly quickly.

32 bit Pentium Registers



8086 REGISTERS

The basic structure has 4 16-bit registers AX,BX,CX,DX but these can be split into 8 8-bit registers.

AH, AL, BH, BL, CH, CL, DH, DL

The H stands for High Byte

The L stands for Low Byte

So a 16 bit number has the 8 most significant bits stored in AH and the 8 least significant bits in AL etc.

General Purpose Registers

The programmer can use the EAX,EBX,ECX and EDX registers in many different ways but they also have some specific roles assigned to them:

The ACCUMULATOR (EAX) is used in multiply and divide operations and in instructions that access I/O ports.

The ECX register is used as a counter in loop operations, providing 65536 passes through a loop before termination. The lower half of CX, the 8-bit CL register, is also used as a counter in shift/rotate operations

General Purpose Registers con’t

Data register EDX is used in multiplication and division operations and also as a pointer when accessing I/O ports.

EBX is a base pointer used in memory addressing.

The last two registers are source and destination indexes ESI and EDI. These are used as pointers in string operations

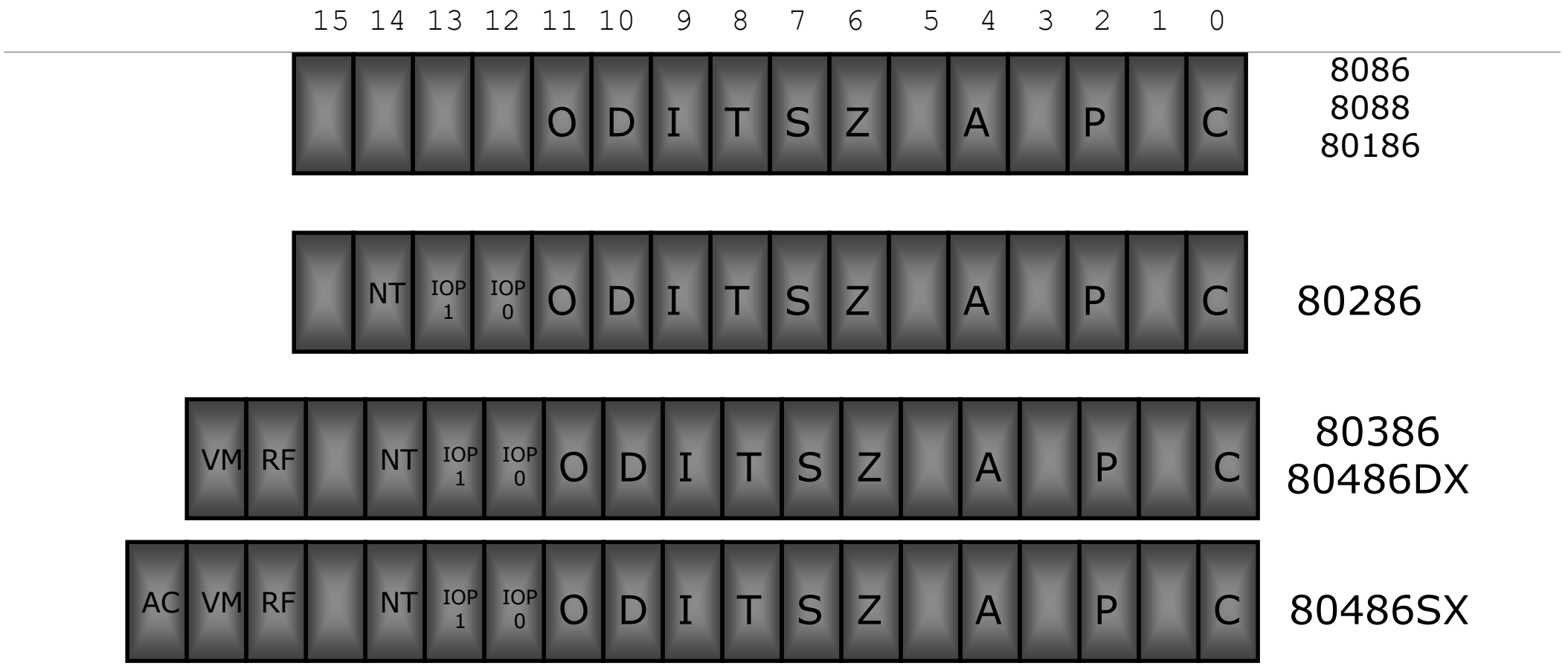
The Flags

The flag register is very important. It is used extensively when controlling the flow of your programs.

The following slide shows the flag registers of all the INTEL 80x86 family.

Note how they are all upward compatible.

80x86 Flag Registers



The Flags con’t

C (carry) indicates a carry after addition or a borrow after subtraction. It can also indicate error conditions in some programs.

P (parity) Is a logic 0 for odd parity and a logic 1 for even parity.

A (Auxiliary carry) holds a carry after addition or a borrow. Highly specialised flag bit tested by DAA and DAS.

Z (zero) indicates that the result of an arithmetic or logic operation is zero. If z=1 the result is zero, z=0 result not zero.

Flags con’t

S (sign) indicates arithmetic sign after addition or subtraction. S=1 when set to negative. S= 0 when positive.

T (trap) when set enables trapping through the on-chip debugger feature.

I (interrupt) Controls the operation of INTR (interrupt request) input pin. If I = 1 the INTR pin is enabled I = 0 INTR disabled.

D (direction) controls selection of increment or decrement for the DI and/or SI registers D=1 registers decremented D = 0 registers incremented

O (overflow) is a condition that occurs when signed numbers are added or subtracted. An overflow indicates that the result has exceeded the capacity of the machine.

The above flags are common to all the INTEL 80x86 family a brief list of the rest follows.

Extra Flags

IOPL (input/output privilege level)

NT (nested task)

RF (resume) used with debugging.

VM (virtual mode) selects virtual mode operation. If set it allows multiple DOS memory partitions

AC (alignment check) if a word or double word is addressed on a non-word boundary. Only the 80486SX contains this flag

Basic Structure of an assembly language program.

```
.586                                ;Recognise 80x86 instructions that use 32-bit
.MODEL FLAT                         ;Generate code for a flat memory model
.STACK 4096                         ;Reserve 4096 bytes for stack operations

.DATA
  keys                  BYTE        128 DUP (?), 0

  msg                   BYTE        'Reading keyboard Example',13,10,0

.CODE
  main          proc
; Your actual assembler mnemonics i.e.
                        mov         eax,23h
                        ……..
main            endp
END             main
```

DIRECTIVES

The program starts by defining a few directives to tell the assembler to take some action, the directives don't actually produce any machine code.

Many of the directives start with a period (.) but others don't.

In the example .586, .stack, .data etc are examples starting with a period.

Example directives not starting with period are END, main PROC

The .stack area

The system stack is used at execution time for procedure calls and local storage.

A stack containing 4096 bytes is large enough for most programs.

The .data area

This is where you define your variables

In the previous example **keys** and **msg** were variable names. The special mnemonic BYTE is an indication that we want to define a series of bytes.

We will come back to this later.

The .code area

This directive tells the assembler the next statements that follow are the executable instructions.

The END on the last line tells the assembler to stop assembling code.

Lets now look at the most common mnemonics.

The MOV instruction

The MOV instruction is probably the one instruction used more than any other

Structure:

<*Mnemonic*> <*Dest*>, <*Src*>(Move Byte or Word)

The MOV instruction is used to move data from place to place within the system. You can move data between memory and registers and visa versa and register to register. But NOT memory to memory directly.

MOV con’t

The different movements of data allowed for this instruction are:

DESTINATION	SOURCE
Memory	Accumulator
Accumulator	Memory
Segment register	Memory/register
Memory/register	Segment register
Register	Register
Register	Memory
Memory	Register
Register	Immediate data
memory	Immediate data

Assignment

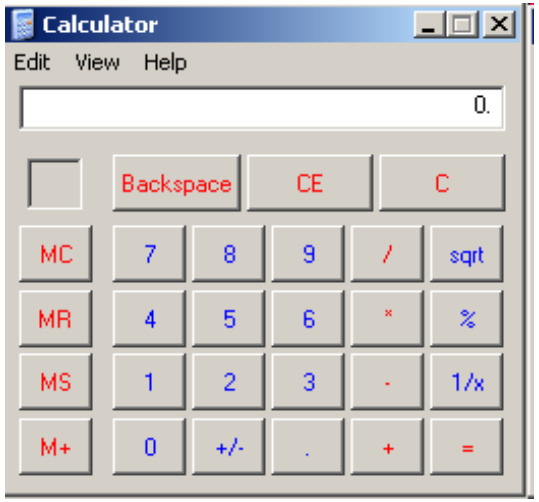
- We use the **MOV** instruction to implement "**A = B**"

```
.DATA
DWORD A ?              ; declare the variable A
DWORD B ?              ; declare the variable B
.CODE
…
MOV EAX, B    ; Can't do memory->memory
MOV A, EAX    ; so need to move to EAX
…
```

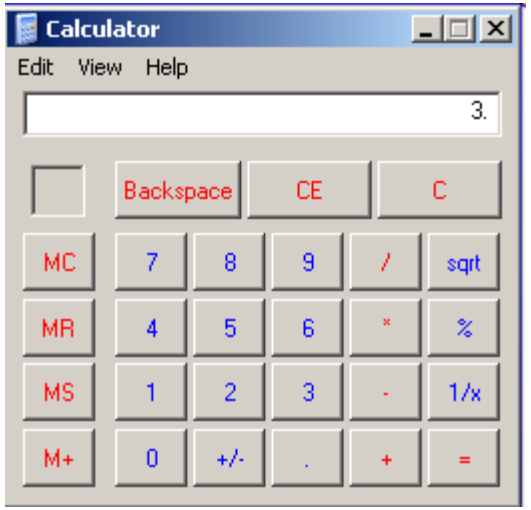
Using the Registers

If you think of a calculator display, as a good model for a register in a microprocessor.

Calculator Example

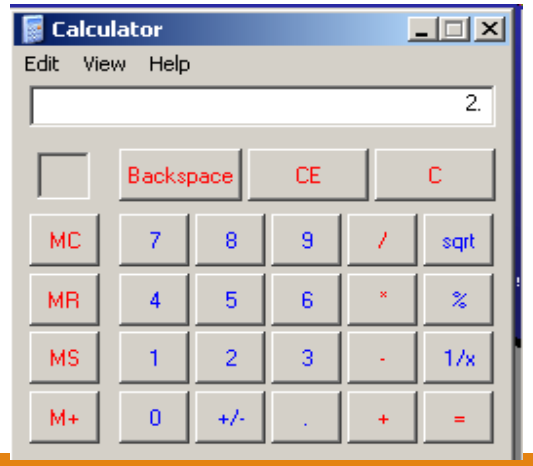


Press 3

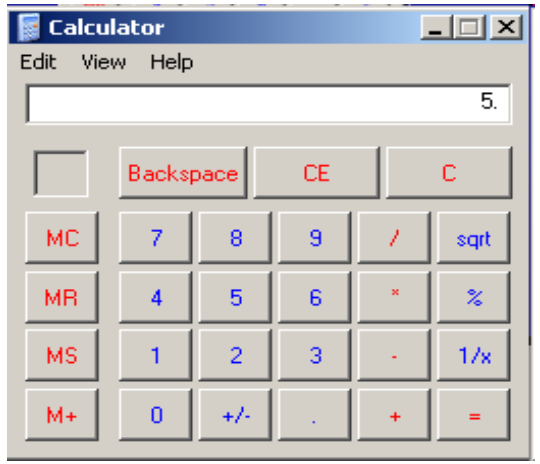


Press +

Press 2



Press =



Using Registers con't

To do the same addition we would do the following piece of machine code:

```
MOV     EAX,3       ;put 3 into register EAX
ADD     EAX,2       ;add 2 to the contents of EAX
```

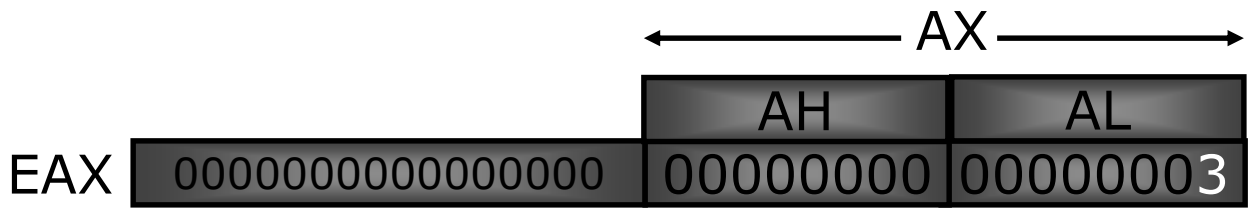
In the above example anything appearing after a semi-colon (;) is a comment.

The answer 5 is left in the EAX register

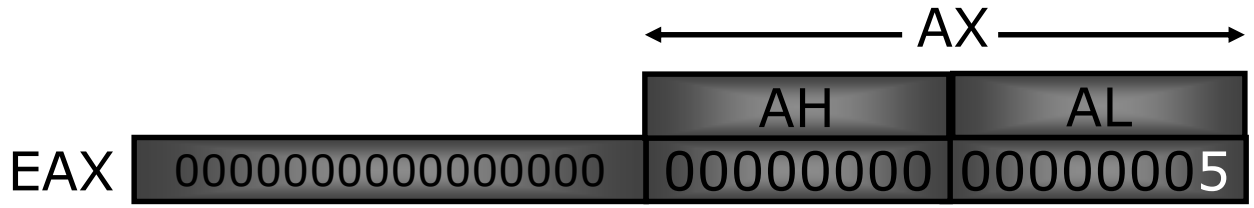
Moving data between Registers is done via the mnemonic MOV.

The mnemonic ADD adds two values together and leaves the result in EAX

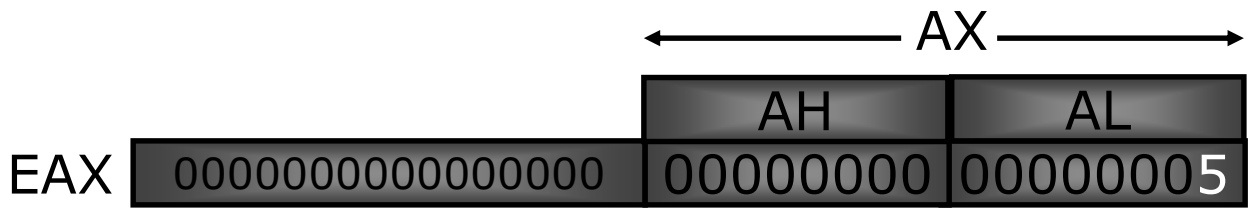
mov eax,3



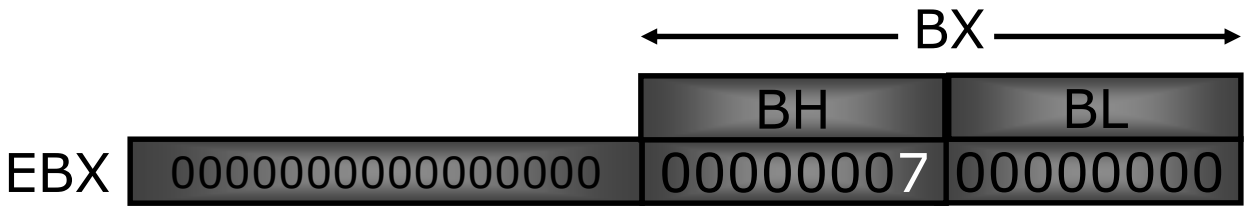
add eax,2



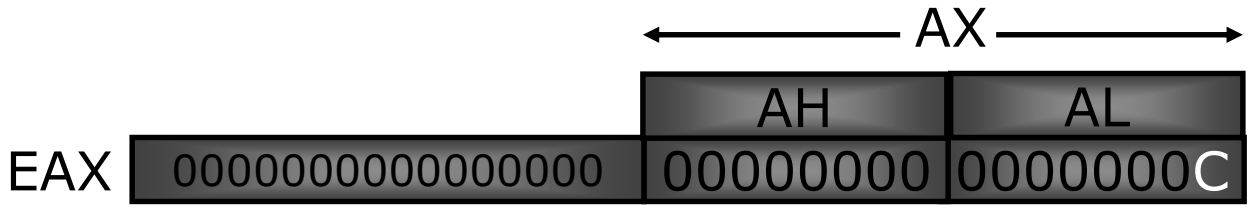
mov al,5



mov bh,7



add al,bh



Using the Registers con’t

The 80x86 instruction set also provides other arithmetic instructions such as SUB, INC,DEC, MUL and DIV. Which we will come back to shortly.

80x86 Instructions

Most instructions have the form

<*Mnemonic*> <*Dest*>, <*Src*>

- i.e. It's generally a 2-address machine
- *Src* and *Dest* can be registers or memory locations
- But they can't *both* be addresses

For example:

```
MOV EAX, EBX            ; move EBX to EAX
MOV myINT, EAX          ; move EAX into myINT
MOV EAX, myINT          ; move myINT into EAX
MOV myINT1, myINT2      ; not allowed
```

There are also 1-address and 0-address instructions:

- E.g. **`JMP <label>, RET, MUL EAX`**,…

Number Format and Removing Ambiguity

As we saw in our calculator demonstration calculators work in DECIMAL (Thank goodness).

BUT! Microprocessors use BINARY.

The Assembler knows about BINARY, DECIMAL and HEXADECIMAL.

The assembler program allows us to write mov EAX,7 and leave it to the assembler to convert 7 to 00000111 binary.

This is fine for numbers upto 9. We now have a problem, if we write mov eax,26 do we mean 26 decimal or 26 hexadecimal (which is 38 in decimal)

To remove ambiguity as far as the assembler is concerned we write 26H for Hexadecimal.

We can also say 26D for Decimal but the assembler assumes decimal if there is no qualifying code letter.

Like wise if we want to directly write binary then 10 can be misinterpreted as decimal TEN so we need to follow the number by B (10B)

Number Format and Removing Ambiguity con't

When we are using hexadecimal we can also run into another problem for example:

MOV DL,AH

Does this mean move the contents of the register AH into register DL or

Does it mean load the register DL with the hexadecimal value A (10D)

To remove this ambiguity ALL HEXADECIMAL NUMBERS BEGINNING with a LETTER must be preceded with a **ZERO** i.e. MOV DL,0AH

"I didn't understand all that stuff he said between 'Good Morning, Class' and 'That concludes my lecture for today'."

Addition/Subtraction

- **ADD/SUB reg, reg**
- **ADD/SUB reg, mem**
- **ADD/SUB mem, reg**
- **ADD/SUB reg, const**
- **ADD/SUB mem, const**
- **ADD/SUB AX/AL, const**
- **INC/DEC reg**
- **INC/DEC mem**
- **INC/DEC reg16**

- **ADD** ***dst, src*** computes

  dst = *dst* + *src*
- **INC** *dst* computes

  dst = *dst* + 1
- These instructions set the flags register in the same way as **CMP**
- Using the **AX** form is both shorter and quicker

SUB

Structure:

- <*Mnemonic*> <*Dest*>, <*Src*>

SUB can be used to subtract 8,16 or 32-bit operands. If the source operand is smaller than the destination operand, the resulting borrow is indicated by setting the carry flag.

SUB example

Example: If AL contains 00 and BL contains 01, what is the result of

- SUB AL,BL?

Solution:

- Subtracting 01 from 00 in 8-bit binary results in FFH, which is the 2's complement representation of -1. So, the contents of AL are replaced with FFH, and both the carry and sign flags are set to indicate a borrow and a negative result

INC

Structure:

<*Mnemonic*> <*Dest*>

Increment Byte or Word by 1

There are many times when we only need to add 1 to a register or contents of a memory location. We could use the ADD instruction, but its simpler to use INC.

In many cases INC generates less machine code, resulting in faster execution

INC example

- ADD EAX,1 and
- INC EAX

Both instructions increment the register EAX by 1.

But, if you look at the resulting machine code that's generated:

EAX,1 assembles into 3 bytes (05 01 00)

INC EAX requires just one byte (40)

As 3 bytes takes longer to execute than 1.

INC EAX executes faster than ADD EAX,1

Multiplication/Division

- **MUL** *OP* or **IMUL** *OP*
  - *OP* must be a register or an address
  - If *OP* is 8-bit, then **AX** := **AL***OP
  - If *OP* is 16-bit, then **DX**:**AX** := **AX***_OP_
- **DIV** *OP* or **IDIV** *OP*
  - *OP* must be a register or an address
  - If *OP* is 8-bit, then **AL** := **AX** div *OP* and **AH**:= **AX** mod *OP*
  - If *OP* is 16-bit, then then **AX** := **DX:AX** div *OP* and **DX**:= **DX:AX** mod *OP*
- Both **MUL** and **DIV** scramble the **Z** and **S** flags

MUL

Structure:

- **<*Mnemonic*> <*Src*> (Multiply Byte or Word Unsigned)**

This unsigned multiply instruction treats byte and word numbers as unsigned binary values. This gives an 8-bit number in the range 0 to 255 and a 16-bit number in the range 0 to 65,535.

The source operand specified in the instruction is multiplied by the value in the accumulator.

If the source is a byte, AL is used as the multiplier, with a 16-bit result left in AX.

If the source is a word, AX is used as the multiplier, and the 32-bit result is returned in registers AX and DX, with DX containing the UPPER 16-bits of the result.

ALL FLAGS ARE AFFECTED

MUL Example 1

Example 1:

What is the result of

- MUL CL

If AL contains 20H and CL contains 80H

Decimal equivalents of 20H and 80H are 32 and 128 respectively. The product of these two numbers is 4096, which is 1000H.

Upon completion, AX will contain 1000H

MUL Example 2

- Example 2:
- What is the result of
  - MUL AX
- If AX contains AO64H ?
- MUL AX, multiplies the accumulator by itself, giving 1,685,923,600 as the result, which is 647D2710H. The upper half of this hexadecimal number (647DH) will be placed into register DX. The lower half (2710H) will replace the contents of AX

IMUL

Structure:

<*Mnemonic*> <*Src*> (Integer Multiply Byte or Word signed)

This multiply instruction is very similar to MUL except that the source operand is assumed to be a signed binary number. This gives byte operands a range -128 to 127 and word operands a range of -32,768 to 32,767. Once again the operand size determines whether the result will be placed into AX or AX and DX

IMUL Example

Example : What is the result of IMUL CL

If AL contains 20H and CL contains 80H?

While this example looks exactly the same as the previous MUL example its not, since the 80H in register CL is interpreted as -128. The product of

-128 and 32 is -4096, which is F000H in 2's complement notation. This is the value placed in AX upon completion.

DIV

Structure:

<*Mnemonic*> <*Src*> (Divide Byte or Word Unsigned)

In this instruction the accumulator is divided by the value represented by the source operand.

All numbers are interpreted as unsigned binary numbers. If the source is a byte, the quotient is placed into AL and the remainder into AH.

If the source is a word, it is divided into the 32-bit number represented by DX and AX with DX holding the upper 16-bits. The 16-bit Quotient is placed into AX and the 16-bit remainder into DX.

If division by Zero is attempted, a type-0 interrupt is generated.

ALL FLAGS ARE AFFECTED

DIV Example

Example:

What is the result of

- DIV BL

If AX contains 2710H and BL contains 32H?

The decimal equivalents of 2710H and 32H are 10,000 and 50 respectively.

The quotient of these two numbers is 200, which is C8H. This is the byte placed in AL. Since the division had no remainder, AH will contain 00.

IDIV

Structure:

<*Mnemonic*> **<*Src*> (Integer Divide Byte or Word signed)**

As with MUL and IMUL, we also see similarities between DIV and IDIV.

In IDIV the operands are treated as signed binary numbers. Everything else remains the same.

Arithmetic Examples

- J=K+M+N+P

```
MOV AX, K
ADD AX, M
ADD AX, N
ADD AX, P
MOV J, AX
```

- J=J-(K+M)

```
MOV AX, K
ADD AX, M
SUB J, AX
```

- J=K/M (unsigned)

```
MOV AL, K
MOV AH, 0
DIV M
MOV J, AL
 (remainder in AH)
```

- J= K/M (signed)

```
MOV AX, K
CWD
IDIV M
MOV J, AX
 (remainder in DX)
```

Variables

Two main types:

- Bytes (keyword **DB**) (8-bit value)
- Words (keyword **DW**) (16-bit value)
- Can also define variables as byte,word,and qword

Format: <*label*><*type*><*value*>

```
myInt DW 1234 or myInt WORD 1234
myChar DB 'a'     ;(initialise to 97)
unInitialisedInt DW ?
unInitialiseByte DB ?
```

There are also directives for declaring and initialising floating point variables

Quick Questions

Name the 4 General Purpose Registers.

What Register is used as the Accumulator.

What Register is used for counting.

When moving data around using the MOV instruction what is the only MOV that cannot be made directly.