Intel 8086

CSC 236

IBM PC body of knowledge 1978-present

Hardware

- 8086
- 80186
- 80286
- 80386
- 80486
- Pentium 1-4
- Core i3, i5, i7, i9

Software

- DOS
- Windows 1-3
- Windows 95, 98
- OS/2 Warp
- NT, 2000, XP
- Vista, 7
- 8, 10


Why do I have to understand history?

Core i7 memory
hex word =
0001

Java or C code:
short int x = ?

What decimal value creates

hex value = 0001

Binary = 0000 0000 0000 0001

This is **not** a trick question

Core i7 memory
hex word =
0001

Java or C code:
short int x = 1

Wrong!

What decimal value creates

hex value = 0001

Binary = 0000 0000 0000 0001

Core i7 memory
hex word =
0001

Java or C code:
short int x = ?

What decimal value creates

hex value = 0001

Binary = 0000 0000 000

To know the correct answer we need to understand history

Decisions that shaped the PC revolution

IBM	Motorola	Intel

Decisions that shaped the PC revolution

IBM	Motorola	Intel
Internal uprocessor		

Largest manufacturer of hardware and software

Decisions that shaped the PC revolution

IBM	Motorola	Intel
Internal uprocessor		
Purchase • hardware		

Decisions that shaped the PC revolution

IBM	Motorola	Intel
Internal uprocessor		
Purchase • hardware • software		

Decisions that shaped the PC revolution

IBM	Motorola	Intel
Internal uprocessor		
Purchase ● hardware ● software		

Why would the largest producer of

chips and software
BUY
chips and software

Decisions that shaped the PC revolution

IBM	Motorola	Intel
Internal uprocessor		
Purchase ● hardware ● software		

IBM sold $10M mainframes

Target price for PCs ~$1K

Decisions that shaped the PC revolution

IBM	Motorola	Intel
Internal uprocessor	16-bit 6800	
Purchase • hardware • software	New start • 68000 • Full 32-bit • Excellent architecture • ... but developers must re-write their software	

Decisions that shaped the PC revolution

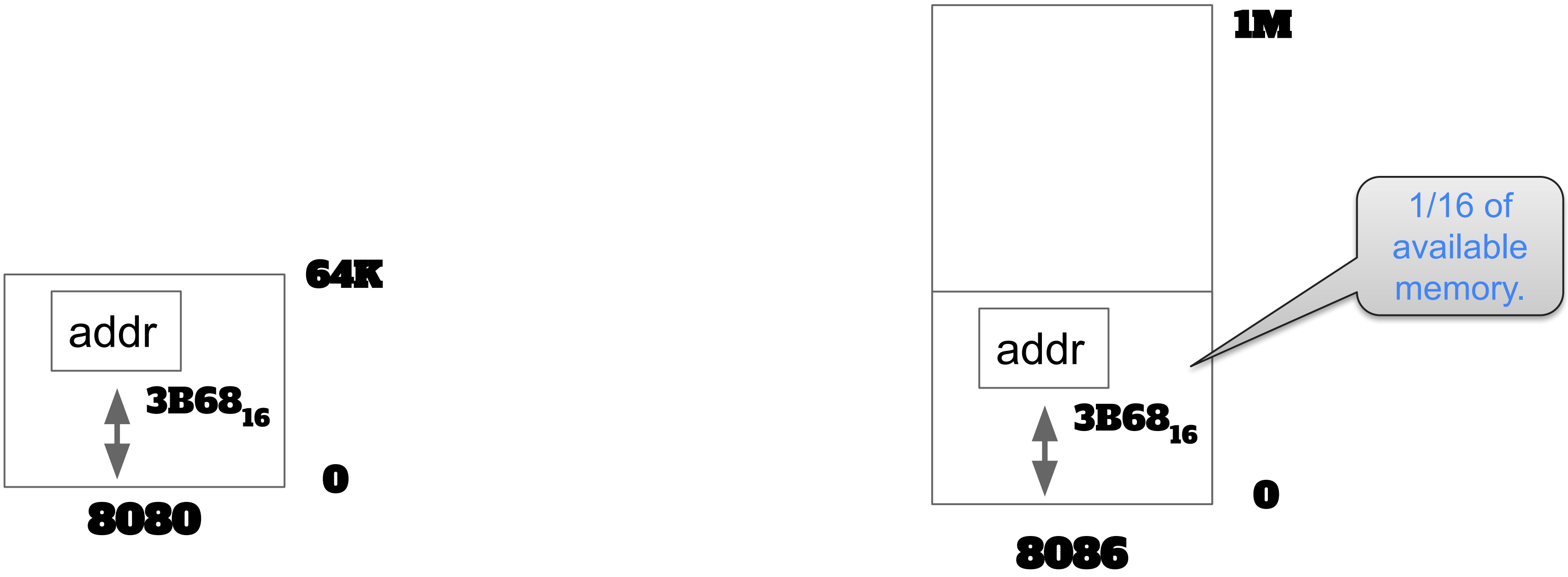
IBM	Motorola	Intel
Internal uprocessor	16-bit 6800	16-bit 8080
Purchase ● hardware ● software	New start ● 68000 ● Full 32-bit ● Excellent architecture ● ... but developers must re-write their software	Conflicting requirements ● Preserve ○ 8080 code ○ 16-bit regs ● support > 64KB memory

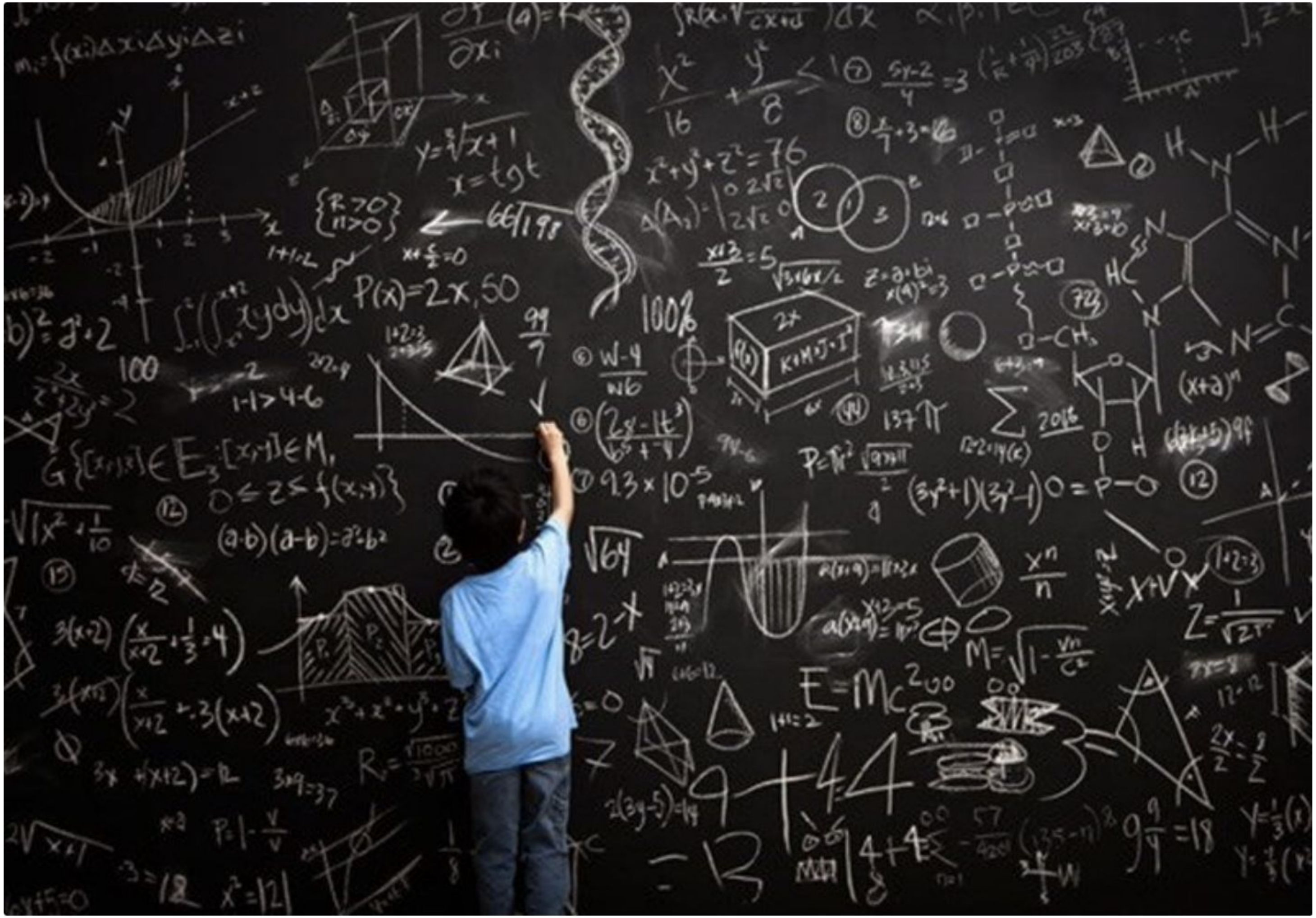
Conundrum

(a confusing and difficult problem)

Conundrum

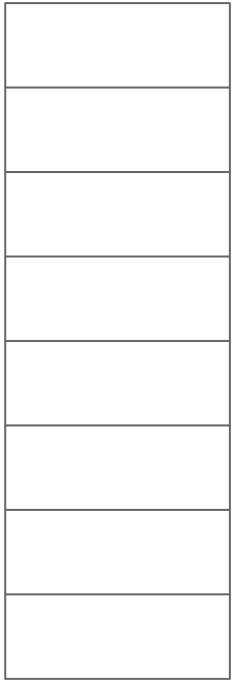
How can a program written for 64K memory and 16 bit registers use a whole Megabyte of memory?





P(x)=2x,50
x²+y²+z²=76
100%
(a·b)(a-b)=a²-b²
E=Mc²

Segmentation



Divide memory into blocks, called
segments
(16 to 64K bytes each)

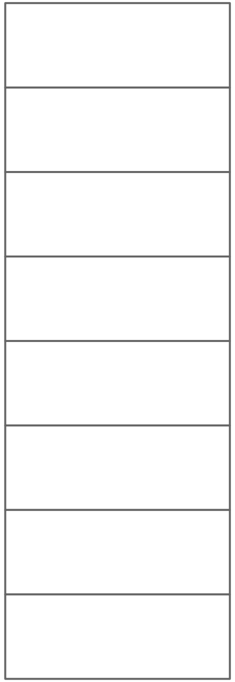
A program can reference

- CS — code segment
- DS — data segment
- SS — stack segment

3 segments ⇒ 2 bits

Why not have 4 segments?

Segmentation



Divide memory into blocks, called **segments**
(16 to 64K bytes each)
A program can reference

- CS — code segment
- DS — data segment
- SS — stack segment
- ES — extra segment

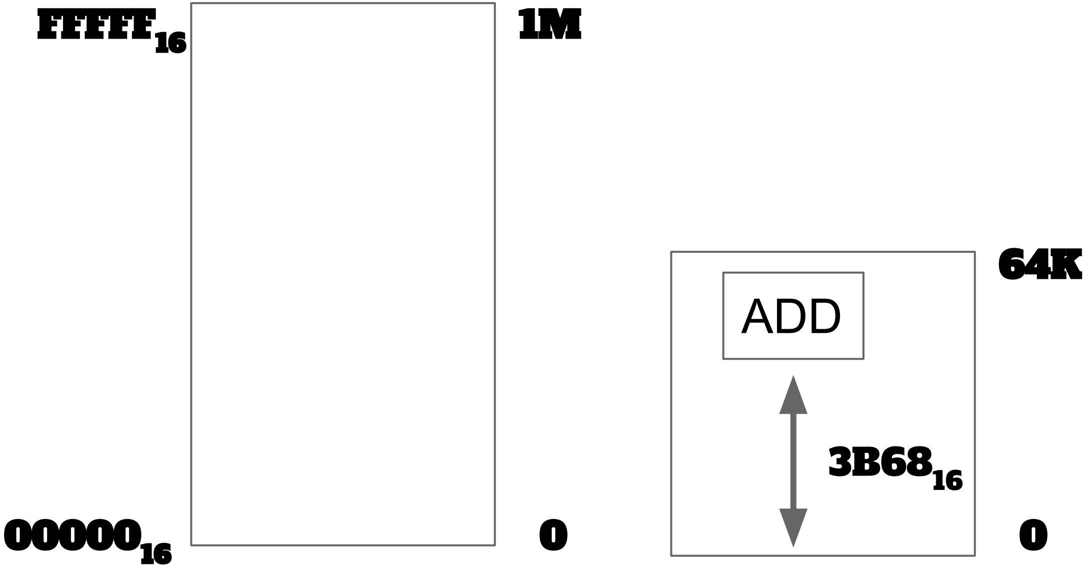
How segmentation works

1MB ranges from

- 00000_{16} **to**
- $FFFFF_{16}$

Segment address is 5 hex digits

But 16-bits just give us 4 digits



How segmentation works

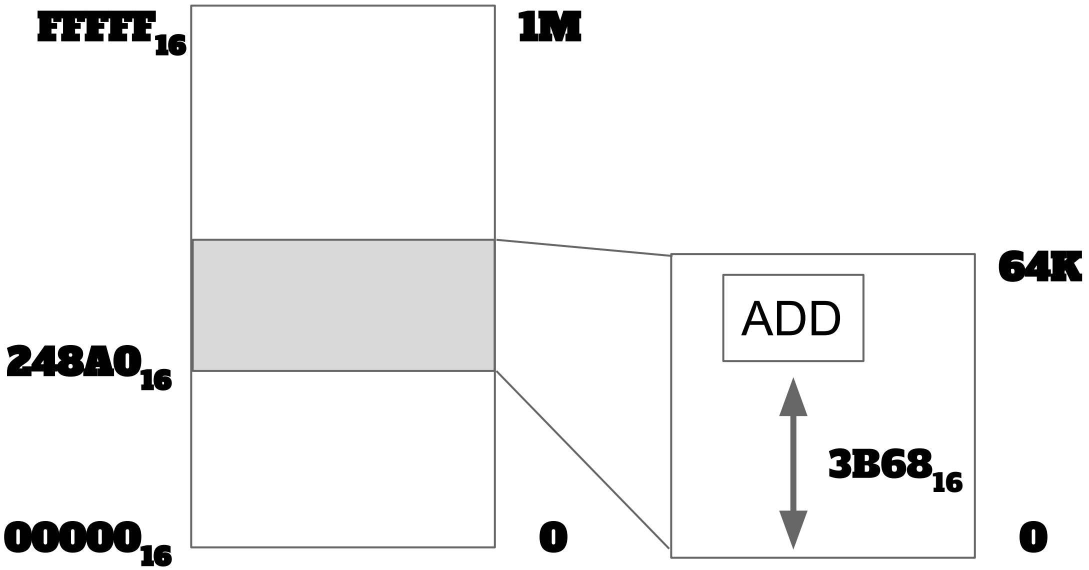
1MB ranges from

- 00000_{16} **to**
- $FFFFF_{16}$

Segment address is 5 hex digits

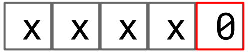
But 16-bits just give us 4 digits

Force lowest to be 0



How segmentation works

16-bit segments register holds 4 hex digits



processor implicitly adds `0`

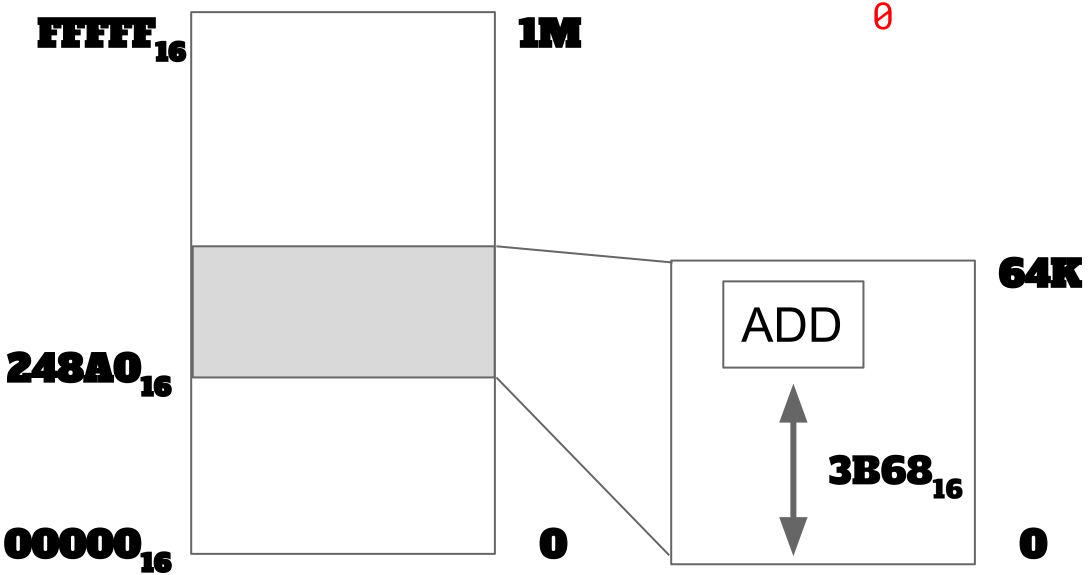
1MB ranges from

- 00000_{16} **to**
- $FFFFF_{16}$

Segment address is 5 hex digits

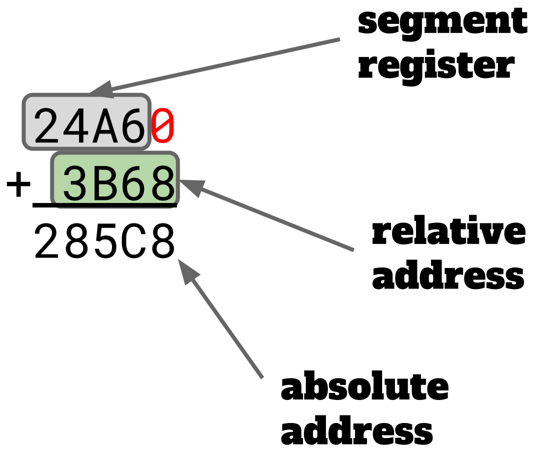
But 16-bits just give us 4 digits

Force lowest to be `0`



How segmentation works

- CPU uses a segment register
  - 1 of 4 hardware registers
- On every memory reference
  - Instructions and data
  - Hardware calculates the absolute address
- Absolute address
  - The actual location in 1MB memory space
  - The start-of-segment plus a relative address
  - That's the real address



Summary

- Low-order hex digit of 5-hex digit (20-bit) segment address is always `0`
- The smallest possible segment is 16 bytes
- Hardware calculates address for <u>every</u> reference
  - Instructions and data
  - ADD x, y, z ⇐ 4 memory references
- Data cannot exceed 64KB
  - Without changing segment register value

Dynamic memory

- How can we be sure program does not violate 64KB limit?
- Static memory
  - Assign data to a segment
  - Use appropriate segment register
  - Can add many instructions
- Dynamic memory allocation
  - Requires runtime checks
  - Slows the program down by at least 10x

Who manages DS register?

- Programmer
  - Or compiler

Who manages DS register?

- Programmer
  - Or compiler
- Ruh Roh

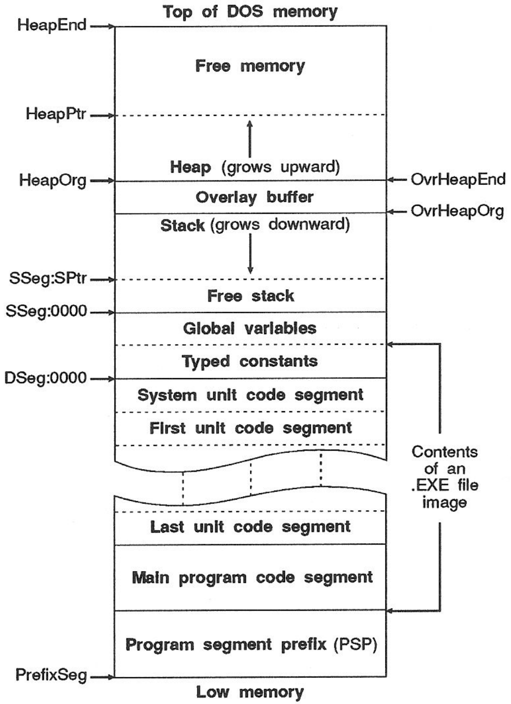
Who manages DS regis

- Programmer
  - Or compiler
- Ruh Roh

Early compiler documentation

The DS register is never changed during program execution. The size of the data segment cannot exceed 64K.

Figure 16.1
Turbo Pascal memory map



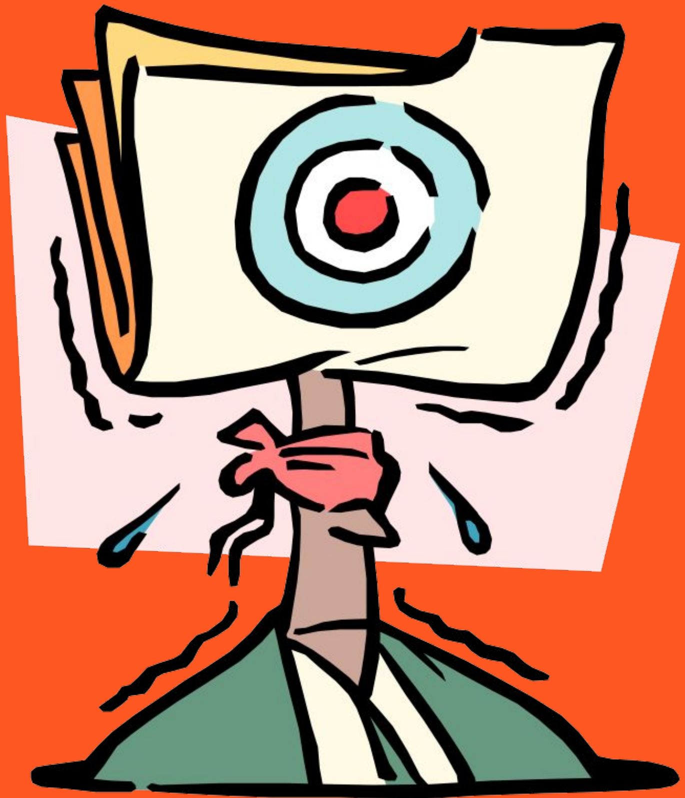
The data segment (addressed through DS) contains all typed constants followed by all global variables. The DS register is never changed during program execution. The size of the data segment cannot exceed 64K.

On entry to the program, the stack segment register (SS) and the stack pointer (SP) are loaded so that SS:SP points to the first byte past the stack segment. The SS register is never changed during

In 1985

- Intel released the 80386
- True 32-bit processor
- Max segment size 4GB
- Removed the need to update the segment registers

Is there anything else to worry about?



Begin Tangent

Gulliver's Travels

All Lilliputians must break their eggs only on the little end.

Those who broke eggs at the big end were angry.

Civil war broke out between

- the Little-Endians and
- the Big-Endians.

GULLIVER IN LILLIPUT.

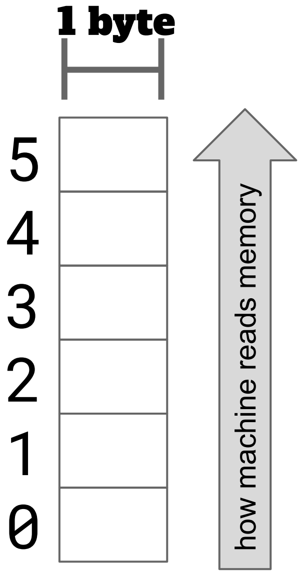
End Tangent

How data is stored in memory

- Instruction

  ADD 1000 | A5 | 03 | E8 |

- $1000_{10} = 03E8_{16}$

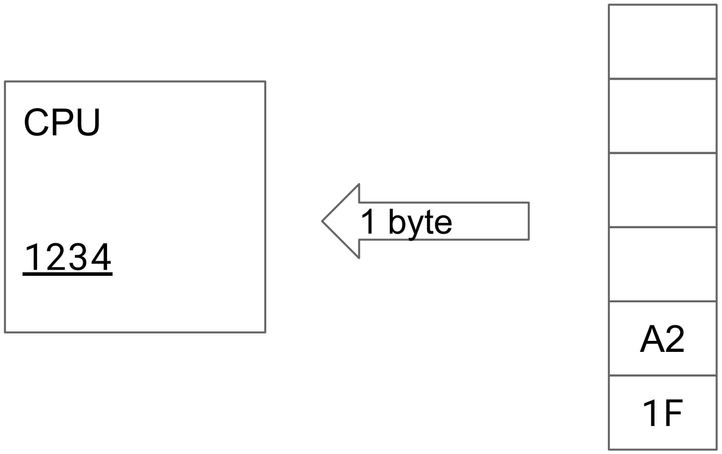


A problem

A problem

- How to add two 16-bit values

  Add 1FA2 from memory
  To <u>1234</u> in the CPU
  xxxx

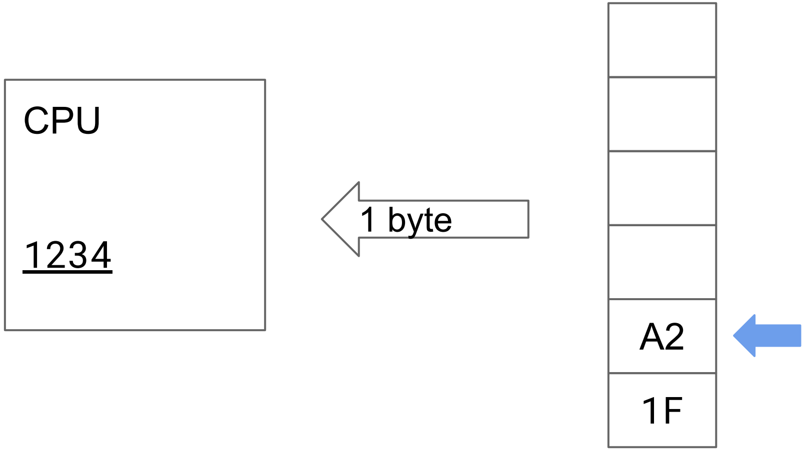


Can only move 1 byte at a time

A problem

- How to add two 16-bit values

  Add 1FA2 from memory
  To 1234 in the CPU
  xxxx

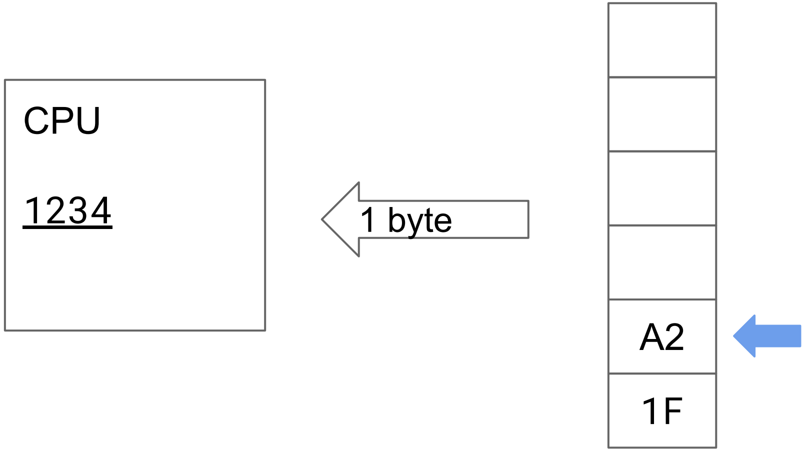


When adding 2-byte values, want to move least significant byte

A problem

- How to add two 16-bit values

  Add 1FA2 from memory
  To <u>1234</u> in the CPU
  xxxx



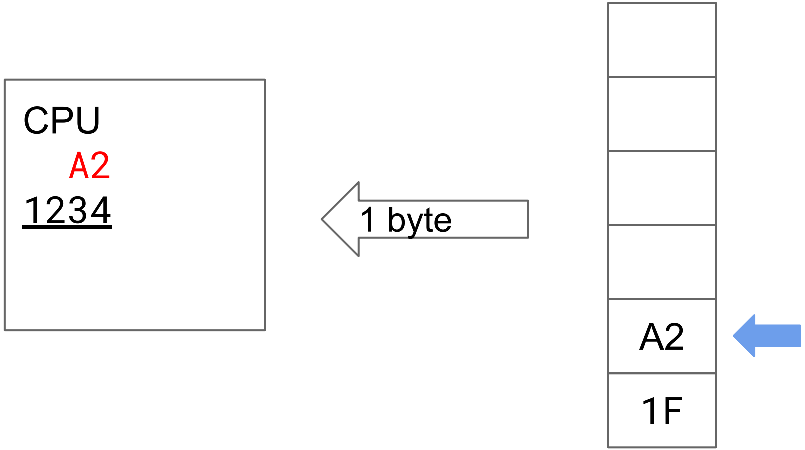
When adding 2-byte values, want to move least significant byte.

Must skip first byte

A problem

- How to add two 16-bit values

  Add 1FA2 from memory
  To <u>1234</u> in the CPU
  XXXX



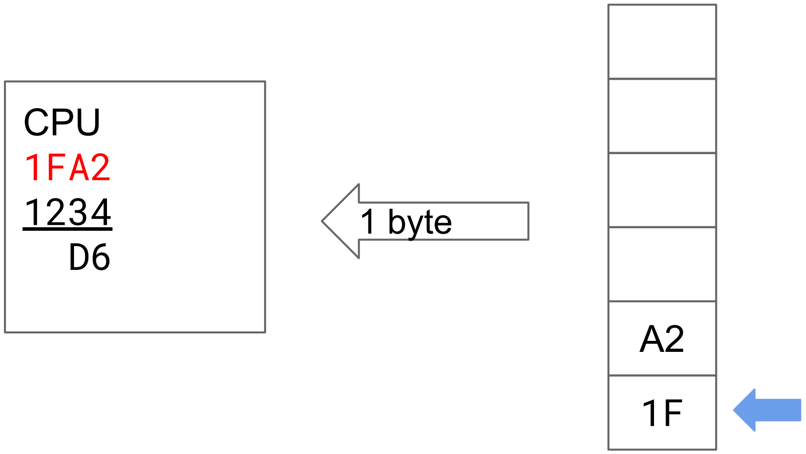
When adding 2-byte values, want to move least significant byte.

Must skip first byte

A problem

- How to add two 16-bit values

Add 1FA2 from memory
To 1234 in the CPU
xxxx



Then go back to first byte.

One-byte data bus

- Limitation of the 8080
  - Remember 8086 was an upgrade to 8080
- Data in memory
  - Store lowest-order byte first
  - “Little Endian”
  - To avoid skipping memory
- In order to maintain backward compatibility
  - 8086 (which is 16-bit) is little endian

Little Endian

ADD 1000

A5	E8	03

03
E8
A5

Byte swapping

Big Endian

ADD 1000

A5	03	E8

E8
03
A5

$1000_{10} = 03E8_{16}$

Problems with Little Endian

- Sharing data with Big Endian machines
- When humans read code

Hex 00 01 in memory is 256_{10} in CPU

00 01 swapped is 01 00 = 256_{10}

Problems with Little Endian

- Sharing data with Big Endian machines
- When humans read code
- Latest CPUs (from Intel) are still little endian
    - Even though it has been 40 years since last byte-wide memory chip

Byte reversal only occurs in memory

- Declare a word variable of 1000_{10}
- Created value is $03E8_{16}$
- Stored in memory as $E803_{16}$
- Reversed whenever moved between memory and CPU
  - Either direction
  - Always

Byte reversal only occurs in memory

- Only swap multi-byte data
- Byte: 0A ⇒ 0A
- Word: 0102 ⇒ 0201
- Double word: 01020304 ⇒ 04030201
- String: “abc”
    - 616263 (in ASCII hex) becomes
    - 616263
    - No swapping because each character is one byte wide
- Swapping is <u>context sensitive</u>

8086 Programmer's Model

8086 Programmer's Model

- How the compiler writer or assembler programmer views the system
- Starts with the register set

8086 Programmer's Model

- How the compiler writer or assembler programmer views the system
- Starts with the register set
  - Special purpose registers

Instruction ptr	IP

Code segment	CS
Data segment	DS
Stack segment	SS
Extra segment	ES

Stack pointer	SP
Base pointer	BP
Source index	SI
Destination idx	DI

8086 Programmer's Model

- How the compiler writer or assembler programmer views the system
- Starts with the register set
  - Special purpose registers

Instruction ptr	IP

Code segment	CS
Data segment	DS
Stack segment	SS
Extra segment	ES

Stack pointer	SP
Base pointer	BP
Source index	SI
Destination idx	DI

8086 Programmer's Model

- How the compiler writer or assembler programmer views the system
- Starts with the register set
  - Special purpose registers

Instruction ptr	IP

Code segment	CS
Data segment	DS
Stack segment	SS
Extra segment	ES

Stack pointer	SP
Base pointer	BP
Source index	SI
Destination idx	DI

8086 Programmer's Model

- How the compiler writer or assembler programmer views the system
- Starts with the register set
    - Special purpose registers

Instruction ptr	IP

Code segment	CS
Data segment	DS
Stack segment	SS
Extra segment	ES

Stack pointer	SP
Base pointer	BP
Source index	SI
Destination idx	DI

8086 Programmer's Model

- How the compiler writer or assembler programmer views the system
- Starts with the register set
  - Special purpose registers

Register	Name
Instruction ptr	**IP**

Register	Name
Code segment	**CS**
Data segment	**DS**
Stack segment	**SS**
Extra segment	**ES**

Register	Name
Stack pointer	**SP**
Base pointer	**BP**
Source index	**SI**
Destination idx	**DI**

Segment registers

- Used to calculate absolute address
- Who sets segment registers
  - OS
  - programmer/compiler
- OS
  - CS
  - SS

  program execution
- Writer
  - DS
  - ES

  data access

Absolute address

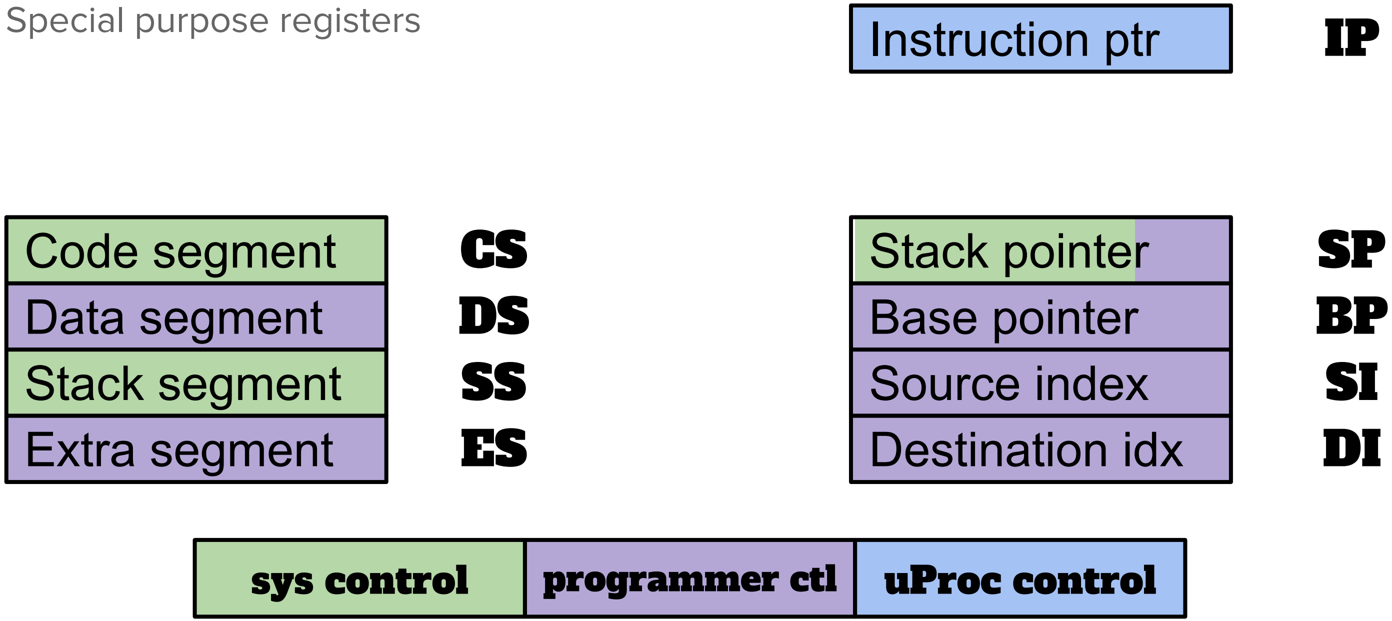
segment register

```
  24A60
+  3B68
  -----
  285C8
```

relative address

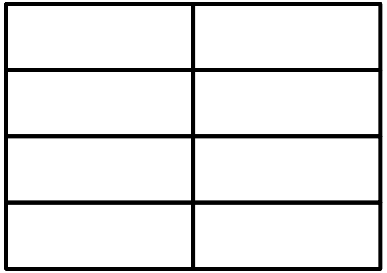
8086 Programmer's Model

- How the compiler writer or assembler programmer views the system
- Starts with the register set
  - Special purpose registers



8086 Programmer's Model

- General purpose registers
  - For use by the programmer
  - Byte and word sizes



16-bit registers

ax	**accumulator**
bx	**base**
cx	**count**
dx	**data**

8086 Programmer's Model

- General purpose registers
  - For use by the programmer
  - Byte and word sizes

When used as 8-bit register, high and low are disconnected

8-bit registers **16-bit registers**

ah	**al**	**ax**	**accumulator**
bh	bl	**bx**	**base**
ch	cl	**cx**	**count**
dh	dl	**dx**	**data**

GP registers

AX = FFFF
0001 Add 1 to AX

AX = 0000 All 16 bits affected

AX = FFFF
01 Add 1 to AL

AX = FF00 Low 8 bits affected

Condition codes

- Gives status of result
  - Negative
  - Overflow
  - Zero
  - …

Condition codes

- SF — sign flag
  - SF = 0 — result of last operation is positive (or zero)
  - SF = 1 — result of last operation is negative
- ZF — zero flag
  - ZF = 1 — result of last operation is zero
  - ZF = 0 — result of last operation is not zero
- OF — overflow flag
  - OF = 1 — a signed overflow occurred

    carry out of sign bit ≠ carry in of sign bit
- CF — carry flag
  - CF = 1 — an unsigned overflow occurred

    carry out (on add) or borrow (on subtract)

Significant question

- Intel add instruction works on
  - Unsigned and
  - Signed (2-comp)
- And hardware does not
  - Know or
  - Care
- How can it set the overflow and carry flags?
  - Answer: it answers both questions
  - The programmer has to check the relevant flags … based on what they are trying to do.

Condition code summary

- SF — equals left most bit of result
- ZF — equals 1 iff all bits are zero
- OF — equals (carry out ≠ carry in) of left-most bit
- CF — equals
  - Carry out of left-most bit on add
  - Borrow of left-most bit on sub

Use OF w/ <u>signed</u> values

Use CF w/ <u>unsigned</u> values

Example 1

```
              1111
AX             FFFF
Add to AX      0001
               ----
               0000

SF = 0
ZF = 1
OF = 0
CF = 1
```

Example 2

```
                11
AX            FF FF
Add to AL     00 01
              FF 00

SF = 0
ZF = 1
OF = 0
CF = 1
```

Example 3

```
AX            7000
Add to AX     1000
              8000

SF = 1
ZF = 0
OF = 1
CF = 0
```

Example 3

```
                                   111
AX              7000        0111 0000 0000 0000
Add to AX       1000        0001 0000 0000 0000
                8000        1000 0000 0000 0000

SF = 1
ZF = 0
OF = 1
CF = 0
```

Example 4

```
                1111
AX               FFFF
Add to AX        FFFF
                 ----
                 FFFE

SF = 1
ZF = 0
OF = 0
CF = 1
```