The x86 Architecture

ICS312
**Machine-Level and
Systems Programming**

Henri Casanova (henric@hawaii.edu)

The 80x86 Architecture

- To learn assembly programming we need to pick a processor family with a given ISA (Instruction Set Architecture)
- In this course we pick the Intel 80x86 ISA (x86 for short)
  - The most common today in existing computers
  - For instance in my laptop
- We could have picked other ISAs
  - Old ones: Sparc, VAX
  - Recent ones: PowerPC, Itanium, MIPS
    - In ICS431/EE460 you'd (likely) be exposed to MIPS
- Some courses in some curricula subject students to two or even more ISAs, but in this course we'll just focused on one more in depth

X86 History (partial)

- In the late 70s Intel creates the 8088 and 8086 processors
  - □ 16-bit registers, 1 MiB of memory, divided into 64KiB segments
- In 1982: the 80286
  - □ New instructions, 16 MiB of memory, divided into 64KiB segments
- In 1985: the 80386
  - □ 32-bit registers, 5 GiB of memory, divided into 4GiB segments
- 1989: 486; 1992: Pentium; 1995: P6
  - □ Only incremental changes to the architecture
- 1997: MMX and 3DNow! extensions
  - □ New instructions to speed up graphics (integer and float)
- 1999: SSE extensions
  - □ New cache instructions, new floating point operations
- 2003: 64-bit x86
- 2005: Virtualization extensions
- ...

X86 History

- It's quite amazing that this architecture has witnessed so little (fundamental) change since the 8086
    - Backward compatibility
    - Imposed early as _the_ ISA (Intel was the first company to produce a 16-bit architecture)
- Some argue that it's an ugly ISA
    - Due to it being a set of add-ons rather than a modern re-design
- But it's relatively easy to implement in hardware, and Intel's was successful in making faster and faster x86 processors for decades, explaining its success
- This architecture is still in use today in 64-bit processors (dubbed x86-64)
    - In this course we do 32-bit x86 though

The 8086 Registers

- To write assembly code for an ISA you must know the name of registers
  - Because registers are places in which you put data to perform computation and in which you find the result of the computation (think of them as variables for now)
  - The registers are identified by binary numbers, but assembly languages give them "easy-to-remember" names
- The 8086 offered 16-bit registers
- Four general purpose 16-bit registers
  - AX
  - BX
  - CX
  - DX

The 8086 Registers

AX	BX	CX	DX
AH	AL	BH	BL	CH	CL	DH	DL
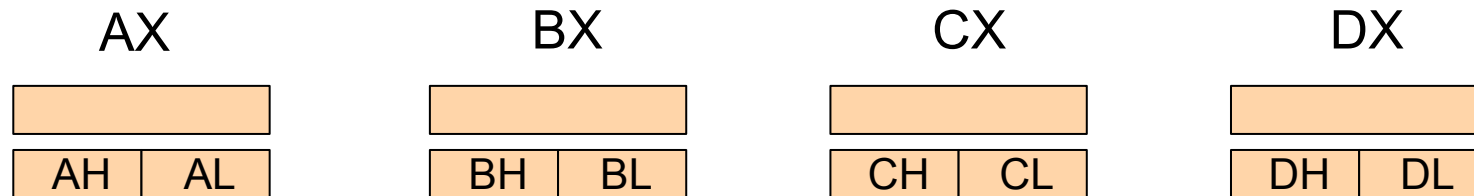
- Each of the 16-bit registers consists of 8 "low bits" and 8 "high bits"
  - Low: least significant
  - High: most significant
- The ISA makes it possible to refer to the low or high bits individually
  - AH, AL
  - BH, BL
  - CH, CL
  - DH, DL

The 8086 Registers

AX	BX	CX	DX

AH	AL	BH	BL	CH	CL	DH	DL
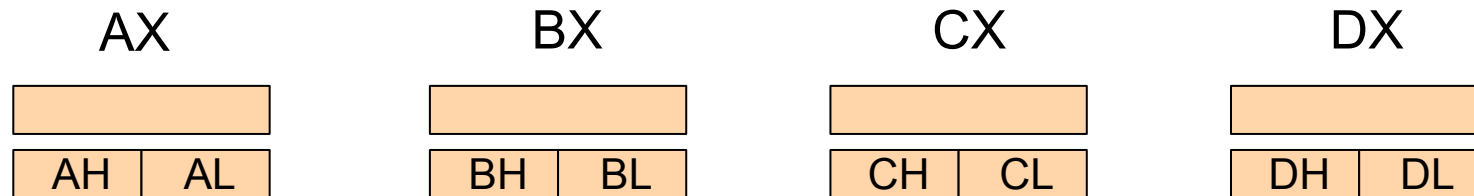
- The xH and xL registers can be used as 1-byte registers to store 1-byte quantities
- Important: both are "tied" to the 16-bit register
  - Changing the value of AX will change the values of AH and AL
  - Changing the value of AH or AL will change the value of AX

The 8086 Registers

- Two 16-bit index registers:
    - SI
    - DI
- These are basically general-purpose registers
- But by convention they are often used as "pointers", i.e., they contain addresses
- And they **cannot** be decomposed into High and Low 1-byte registers
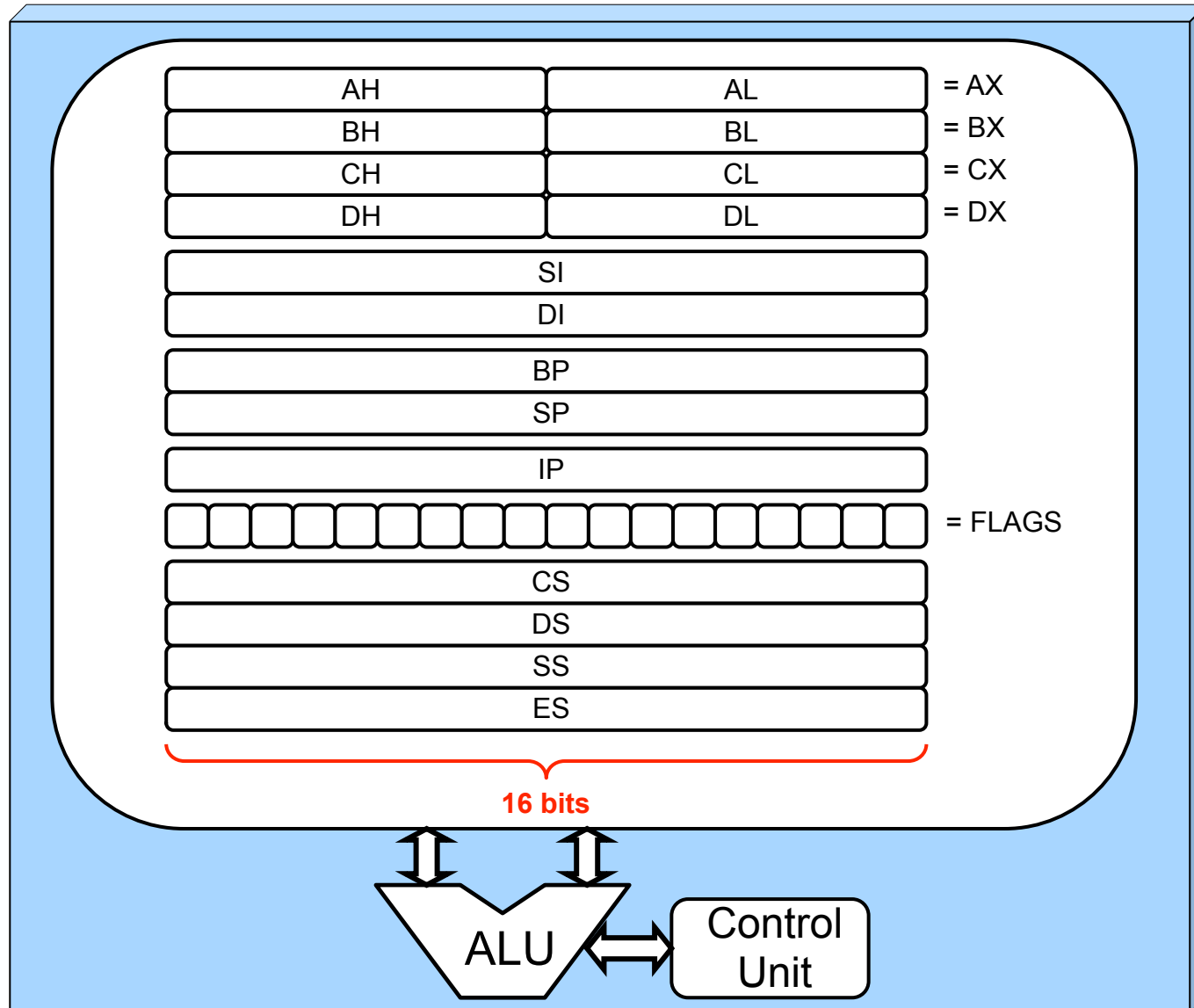
The 8086 Registers

- Two 16-bit special registers:
    - BP: Base Pointer
    - SP: Stack Pointer
    - We'll discuss these at length later
- Four 16-bit segment registers:
    - CS: Code Segment
    - DS: Data Segment
    - SS: Stack Segment
    - ES: Extra Segment
    - We'll discuss these later as well but won't use them much at all

The 8086 Registers

- **The 16-bit Instruction Pointer (IP) register:**
  - Points to the next instruction to execute
  - Typically not used directly when writing assembly code
- **The 16-bit FLAGS registers**
  - Information is stored in individual bits of the FLAGS register
  - Whenever an instruction is executed and produces a result, it may modify some bit(s) of the FLAGS register
  - Example: Z (or ZF) denotes one bit of the FLAGS register, which is set to 1 if the previously executed instruction produced 0, or 0 otherwise
  - We'll see many uses of the FLAGS registers

The 8086 Registers

AH	AL	= AX
BH	BL	= BX
CH	CL	= CX
DH	DL	= DX

SI
DI

BP
SP

IP

	= FLAGS

CS
DS
SS
ES

16 bits

ALU

Control Unit

Addresses in Memory

- We mentioned several registers that are used for holding addresses of memory locations
- Segments:
  - CS, DS, SS, ES
- Pointers:
  - SI, DI: indices (typically used for pointers)
  - SP: Stack pointer
  - BP: (Stack) Base pointer
  - IP: pointer to the next instruction
- Let's look at the structure of the address space

Address Space

- In the 8086 processor, a program is limited to referencing an address space of size 1MiB, that is 2^{20} bytes
- Therefore, addresses are 20-bit long!
- A d-bit long address allows to reference 2^d different "things"
- Example:
  - 2-bit addresses
    - 00, 01, 10, 11
    - 4 "things"
  - 3-bit addresses
    - 000, 001, 010, 011, 100, 101, 110, 111
    - 8 "things"
- In our case, these things are "bytes"
  - One does not address anything smaller than a byte
- Therefore, a 20-bit address makes it possible to address 2^{20} individual bytes, or 1MiB

Address Space

- One says that a running program has a 1MiB address space

- And the program needs to use 20-bit addresses to reference memory content
  - Instructions, data, etc.

- Problem: registers are at 16-bit long! How can they hold a 20-bit address???

- The solution: split addresses in two pieces:
  - The selector
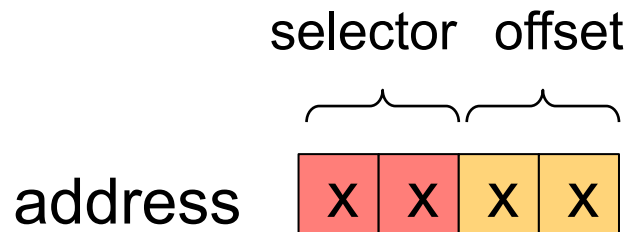 - The offset

Simple Selector and Offset

- Let us assume that we have an address space of size $2^4=16$ bytes
  - Yes, that would not be a useful computer
- Addresses are 4-bit long
- Let's assume we have a 2-bit selector and a 2-bit offset
  - As if our computer had only 2-bit registers

- We take such small numbers because it's difficult to draw pictures with 2^{20} bytes!
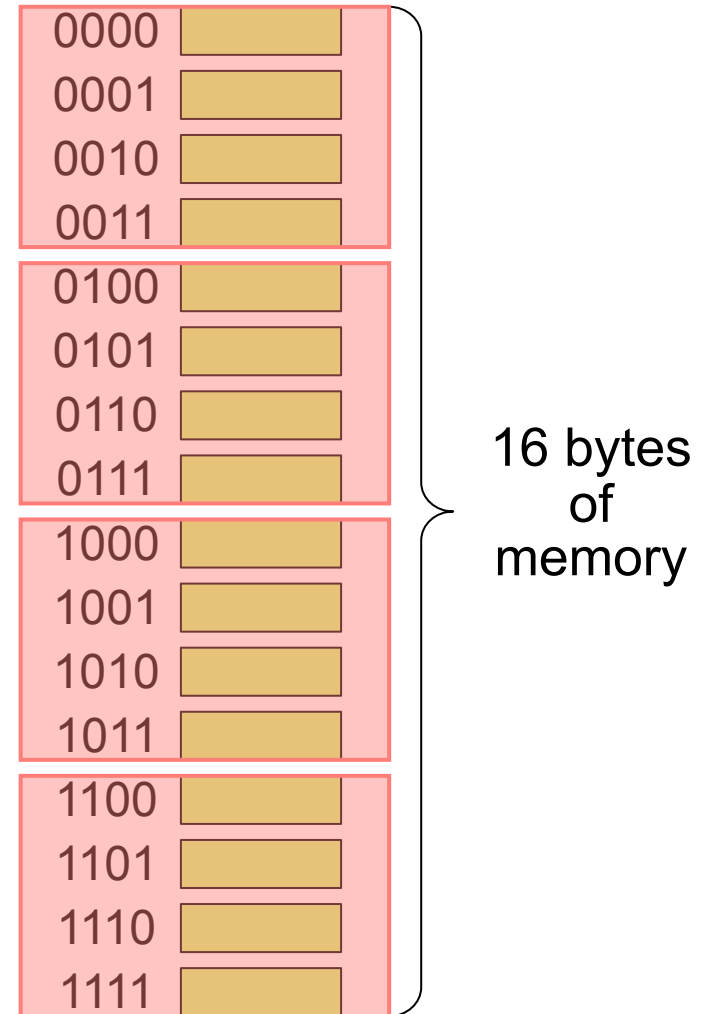
Selector and Offset Example

selector offset

address **0 1 x x**
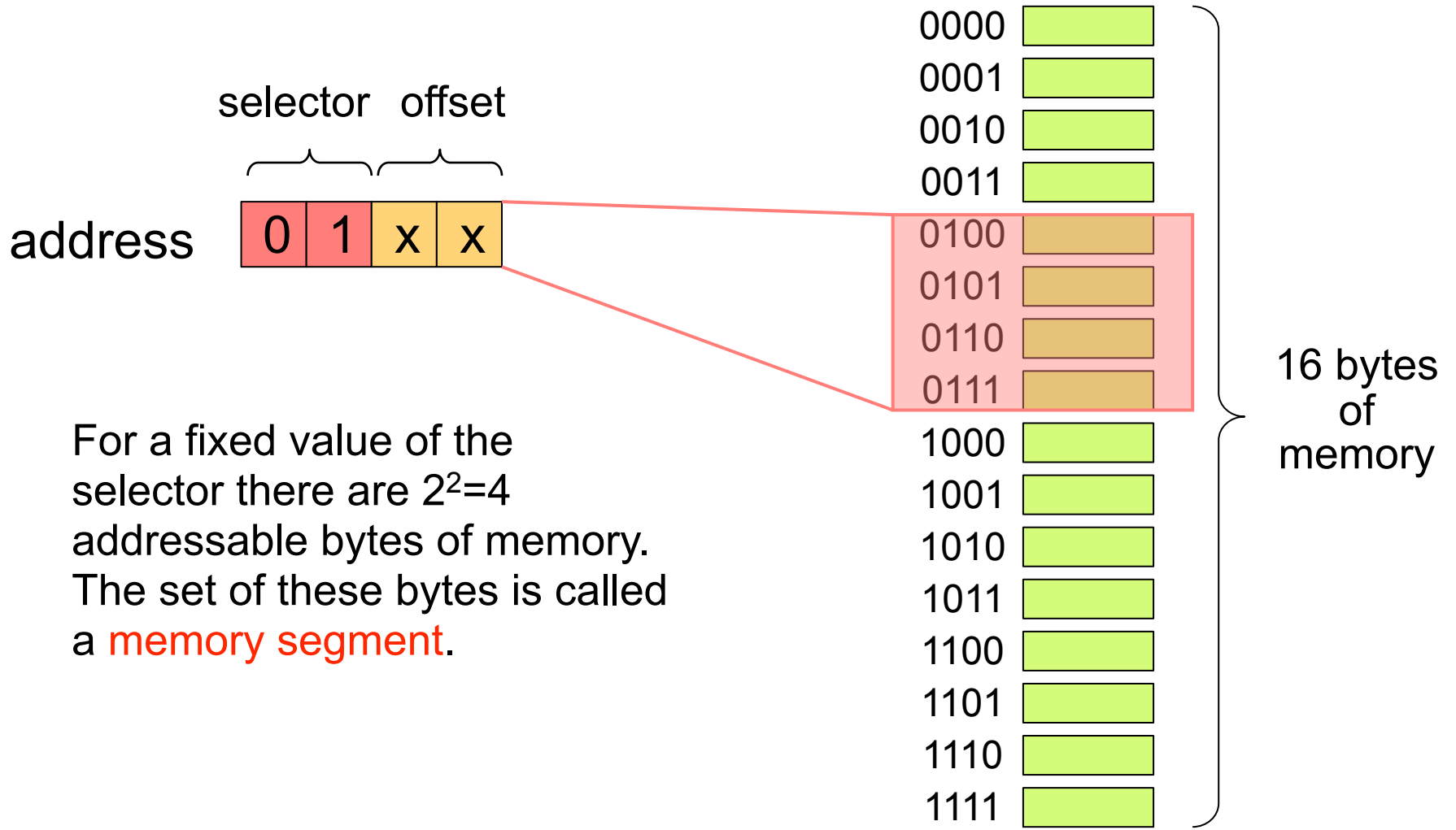
For a fixed value of the selector there are $2^2=4$ addressable bytes of memory. The set of these bytes is called a memory segment.

0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

16 bytes of memory

Selector and Offset Example

selector   offset

address | x | x | x | x |

We have 16 bytes of memory
We have 4-byte segments
We have 4 segments

| 0000 | |
0001	
0010	
0011	
0100	
0101	
0110	
0111	
1000	
1001	
1010	
1011	
1100	
1101	
1110	
1111	

16 bytes of memory
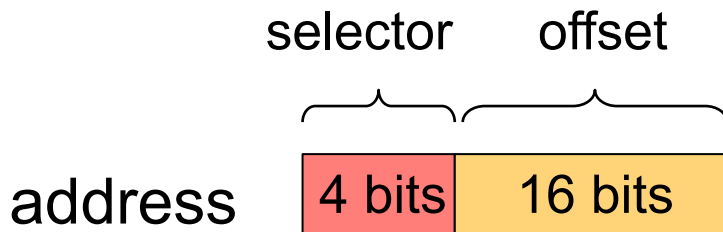
Selector and Offset

- The way in which one addresses the memory content is then pretty straightforward

- First, set the bits of the selector to "pick" a segment

- Second, set the bits of the offset to address a byte within the segment

- This all makes sense because a program typically addresses bytes that are next to each other, that is within the same segment

- So, the selector bits stay the same for a long time, while the offset bits change often
  - Of course, this isn't true for tiny 4-byte segments as in our example…
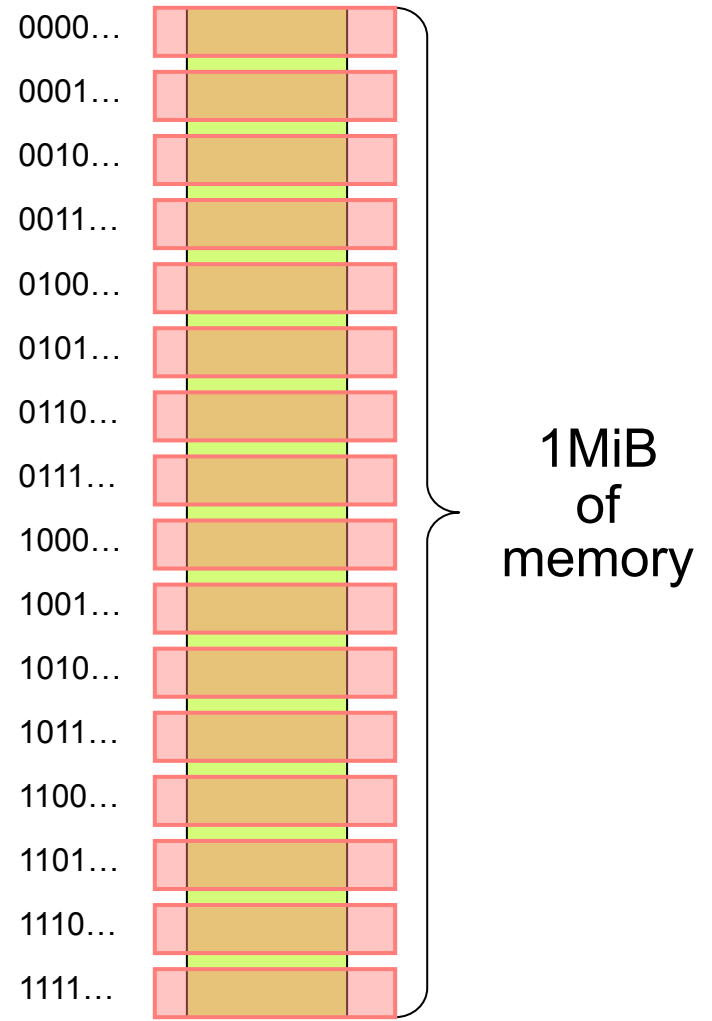
For 20-bit Addresses

selector	offset
4 bits	16 bits
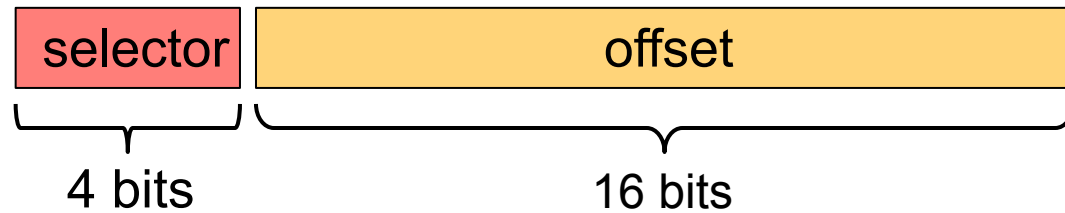
- On the 8086 the offset if 16-bit long
  - And therefore the selector is 4-bit
- We have $2^4 = 16$ different segments
- Each segment is 2^{16} byte = 64KiB
- For a total of 1MiB of memory, which is what the 8086 used

For 20-bit Addresses

selector   offset

address | 4 bits | 16 bits |

We have 1MB of memory
We have 64K segments
We have 16 segments

```
0000…
0001…
0010…
0011…
0100…
0101…
0110…
0111…
1000…
1001…
1010…
1011…
1100…
1101…
1110…
1111…
```

1MiB
of
memory

The 8086 Selector Scheme

- So far we've talked about the selector as a 4-bit quantity, for simplicity
  - And we can clearly store a 4-bit quantity in a 16-bit register
- This leads to 16 non-overlapping segments
- The designers of the 8086 wanted more flexibility
- E.g., if you know that you need only an 8K segment, why use 64K for it? Just have the "next" segment start 8K after the previous segment
  - We'll see why segments are needed in a little bit
- So, for the 8086, the selector is NOT a 4-bit field, but rather the address of the beginning of the segment
- But now we're back to our initial problem: Addresses are 20-bit, how are we to store an address in a 16-bit register???

The 8086 Selector Scheme

- What the designers of the 8086 did is pretty simple
- Enforce that the beginning address of a segment can only be a multiple of 16
- Therefore, its representation in binary always has its four lowest bits set to 0
- Or, in hexadecimal, its last digit is always 0
- So the address of a beginning of a segment is a 20-bit hex quantity that looks like: XXXX0
- Since we know the last digit is always 0, no need to store it at all
- Therefore, we need to store only 4 hex digits
- Which, lo and behold, fit in a 16-bit register!

The 8086 Selector Scheme

- So now we have two 16-bit quantities
  - The 16-bit selector
  - The 16-bit offset
- The selector must be stored in one of the "segment" registers
  - CS, DS, SS, ES
- The offset is typically stored in one of the "index" registers
  - SI, DI
  - But could be stored in a general purpose register
- Address computation is straightforward
- Given a 16-bit selector and a 16-bit offset, the 20-bit address is computed as follows
  - Multiply the selector by 16
    - This simply transforms XXXX into XXXX0, thanks to the beauty of hexadecimal
  - Add the offset

Code, Data, Stack

- Although we'll discuss these at length later, let's just accept for now that the address space has three regions

- Therefore, the program constantly references bytes in three different segments
  - For now let's assume that each region is fully contained in a single segment, which is in fact not always the case

- **CS**: points to the beginning of the code segment

- **DS**: points to the beginning of the data segment

- **SS**: points to the beginning of the stack segment

code

data

stack

address space
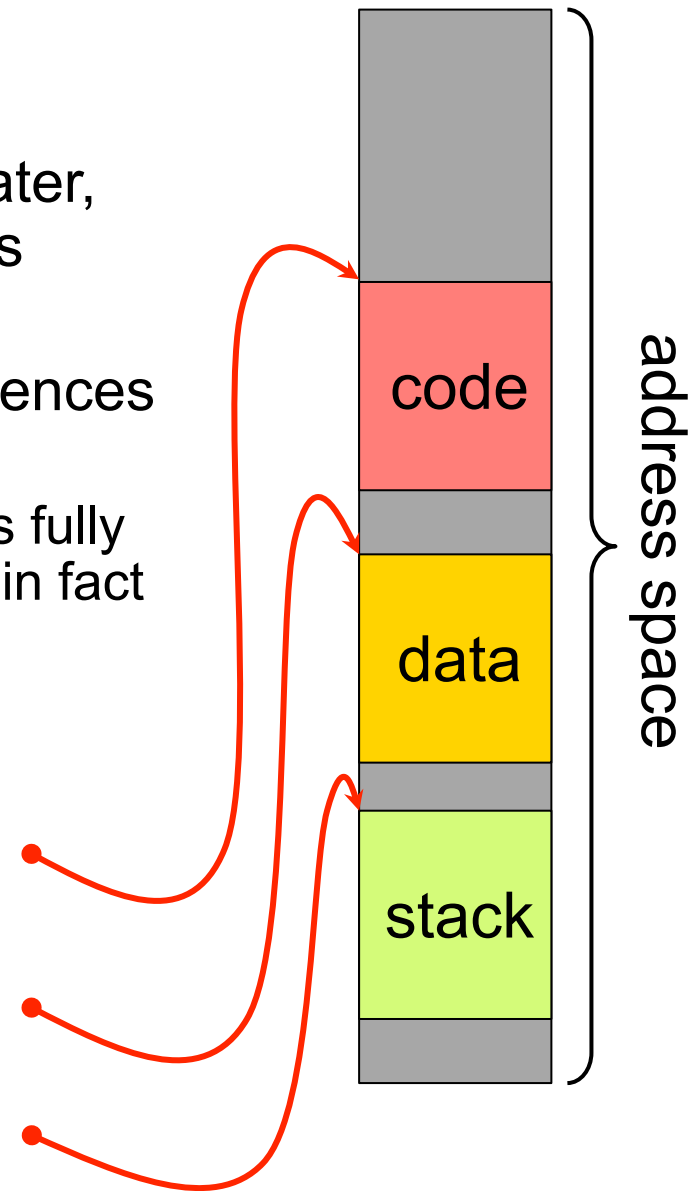
The trouble with segments

- It is well-known that programming with segmented architectures is really a pain
- In the 8086 you constantly have to make sure segment registers are set up correctly
- What happens if you have data/code that's more than 64KiB?
  - You must then switch back and forth between selector values, which can be really awkward
- Something that can cause complexity also is that two different (selector, offset) pairs can reference the same address
  - Example: (a,b) and (a-1, b+16)
- There is an interesting on-line article on the topic: http://world.std.com/~swmcd/steven/rants/pc.html

How come it ever survived?

- If you code and your data are <64KiB, segments are great
- Otherwise, they are a pain
- Given the horror of segmented programming, one may wonder how come it stuck?
- From the linked article: "*Under normal circumstances, a design so twisted and flawed as the 8086 would have simply been ignored by the market and faded away.*"
- But in 1980, Intel was lucky that IBM picked it for the PC!
  - Not to criticize IBM or anything, but they were also the reason why we got stuck with FORTRAN for so many years :/
  - Big companies making "wrong" decisions has impact

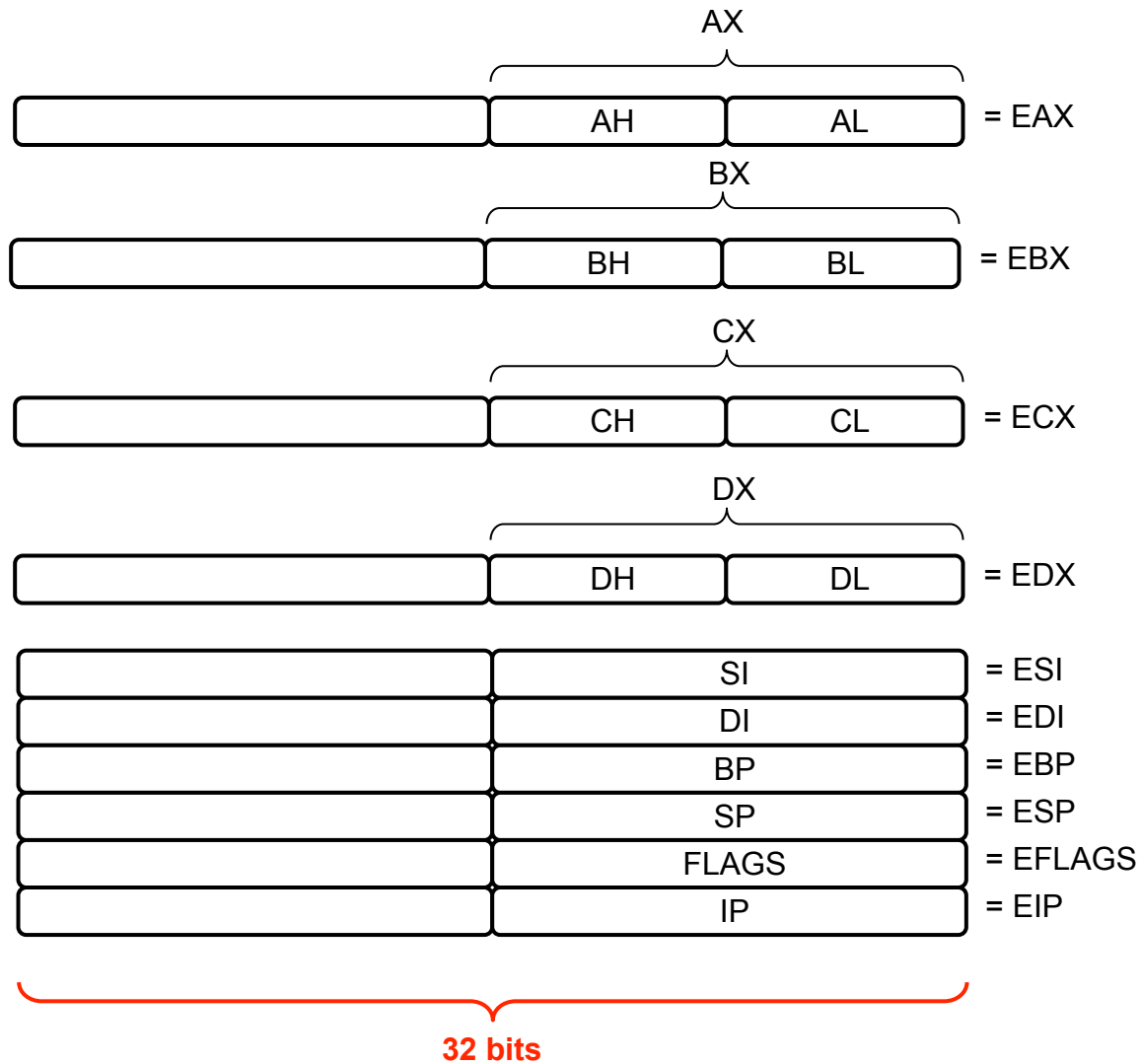- Luckily in this course we use 32-bit x86...

32-bit x86

- With the 80386 Intel introduced a processor with 32-bit registers
- Addresses are 32-bit long
  - Segments are 4GB
  - Meaning that we don't really need to modify the segment registers very often (or at all), and in fact we'll call assembly from C so that we won't see segments at all
  - But we spent a little bit of time on the 8086 segmenting scheme just to get used to this type of address computations
- Let's have a look at the 32-bit registers

The 80386 32-bit registers

- The general purpose registers: extended to 32-bit
  - EAX, EBX, ECX, EDX
  - For backward compatibility, AX, BX, CX, and DX refer to the 16 low bits of EAX, EBX, ECX, and EDX
  - AH and AL are as before
  - There is no way to access the high 16 bits of EAX separately
- Similarly, other registers are extended
  - EBX, EDX, ESI, EDI, EBP, ESP, EFLAGS
  - For backward compatibility, the previous names are used to refer to the low 16 bits

The 8386 Registers

| | | AX | |
| | AH | AL | = EAX |

| | | BX | |
| | BH | BL | = EBX |

| | | CX | |
| | CH | CL | = ECX |

| | | DX | |
| | DH | DL | = EDX |

	SI	= ESI
	DI	= EDI
	BP	= EBP
	SP	= ESP
	FLAGS	= EFLAGS
	IP	= EIP

32 bits

Conclusion

- From now on we'll keep referring to the register names, so make sure you absolutely know them

    - The registers are, in some sense, the variables that we can use

    - But they have no "type" and you can do absolutely whatever you want with them, meaning that you can do horrible mistakes

- We're ready to move on to writing assembly code for the 32-bit x86 architecture