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)
- We will use 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
  - Unlikely old ones: Sparc, VAX
  - Recent ones: PowerPC, Itanium, MIPS
    - In ICS331/ICS431/EE460 you'd (likely) be exposed to MIPS
- Some courses in some curricula subject students to two or even more ISAs in a single semester, but in this course we'll just focused on one

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

x86 History (partial)

- 1997 - now: improvements, new features galore
  - MMX and 3DNow! extensions
  - New instructions to speed up graphics (integer and float)
  - New cache instructions, new floating point operations
  - Virtualization extensions
  - etc..
- 2015: the “Skylake” code name (6th generation)
  - “All models support: MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX, AVX2, FMA3, Enhanced Intel SpeedStep Technology (EIST), Intel 64, XD bit (an NX bit implementation), Intel VT-x, Intel VT-d, Turbo Boost, AES-NI, Smart Cache, Intel Insider.”
  - 4 cores
  - Around $350 for the i7 model
- The “Icelake” is expected in 2020
- Several manufacturers build x86-compliant processors
  - And have been for a long time

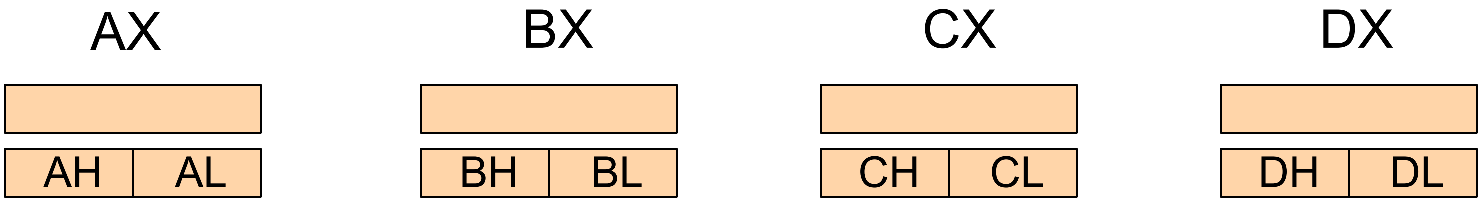
x86 History

- It’s quite amazing that this architecture has witnessed so little (fundamental) change since the 8086
  - All in the name of backward compatibility
  - Imposed early as “the one ISA” (Intel was the first company to produce a 16-bit architecture, which secured its success)
- Many argue that it’s an unsightly ISA
  - Due to it being a set of add-ons rather than a modern re-design
  - Famous quote by Mike Johnson (AMD): “The x86 isn’t all that complex… it just doesn’t make a lot of sense” (1994)
- But it’s relatively easy to implement in hardware, and constructors have been successfully making faster and faster x86 processors for decades, explaining its wide adoption
- This architecture is still in use today in 64-bit processors (dubbed x86-64), e.g., in all our laptops in this classroom today
  - 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
  - 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

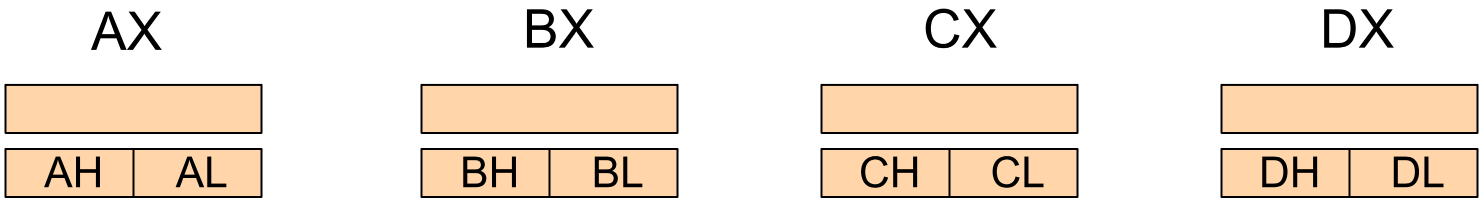


AX BX CX DX

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



- The xH and xL registers can be used as 1-byte registers to store 1-byte values
- Important: both are “tied” to the 16-bit register
  - Changing the value of AX will change the values of AH and/or 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 general-purpose registers
- But by convention they are often used as “pointers”, i.e., they contain addresses instead of data
- And they **cannot** be decomposed into High and Low 1-byte registers

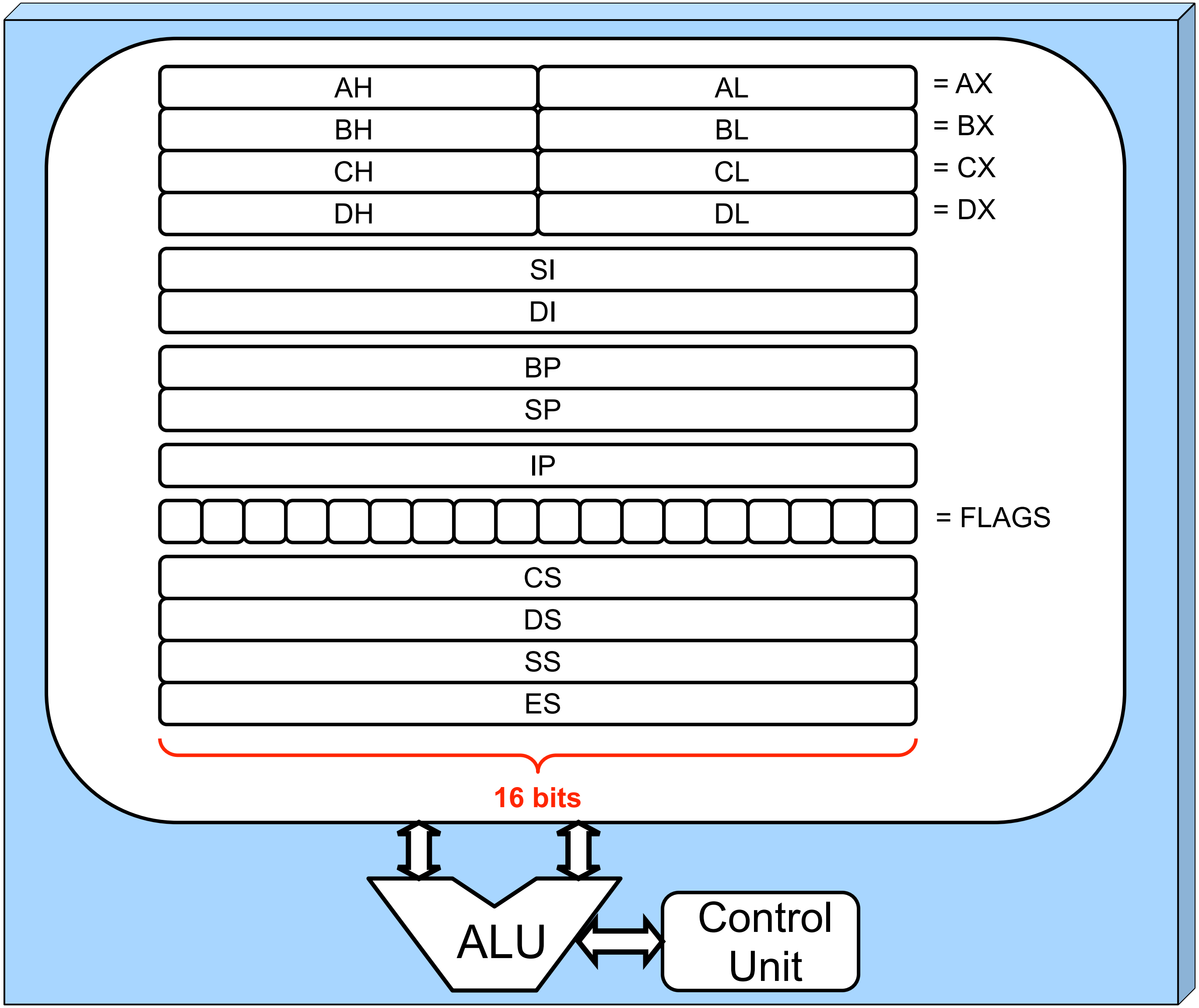
The 8086 Registers

- 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 soon a little bit, but won't use them 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
  - The bits of the FLAGS register contain “status bits” that each has its individual name and meaning
    - It’s really a collection of bits, not a multi-bit value
  - 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

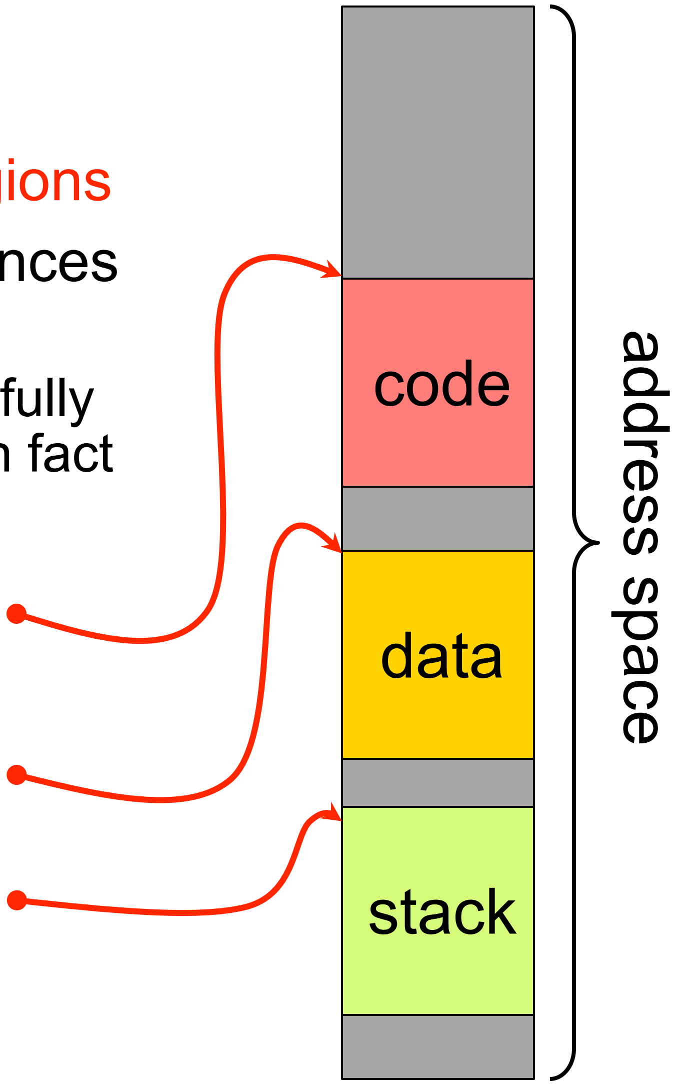
Control Unit

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

Code, Data, Stack

- The address space has three logical 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
- ES: points to the beginning of an “extra” segment
  - used to store/address temporary data



The trouble with segments

- It is well-known that programming with segmented architectures is really a pain
- In the 8086 you constantly had to make sure segment registers are set up correctly
- But if your data/code is more than 64KiB then it becomes awkward
  - You must then switch back and forth between so-called selector values to reference different segments at runtime
- There is an interesting on-line article on the topic called “the curse of segments”
  - 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
- And of course, our code and data are way bigger!
- Given the horror of segmented programming, one may wonder how come it stuck?
- From the "curse of segments" 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 (for you) in this course we use 32-bit x86...

32-bit x86

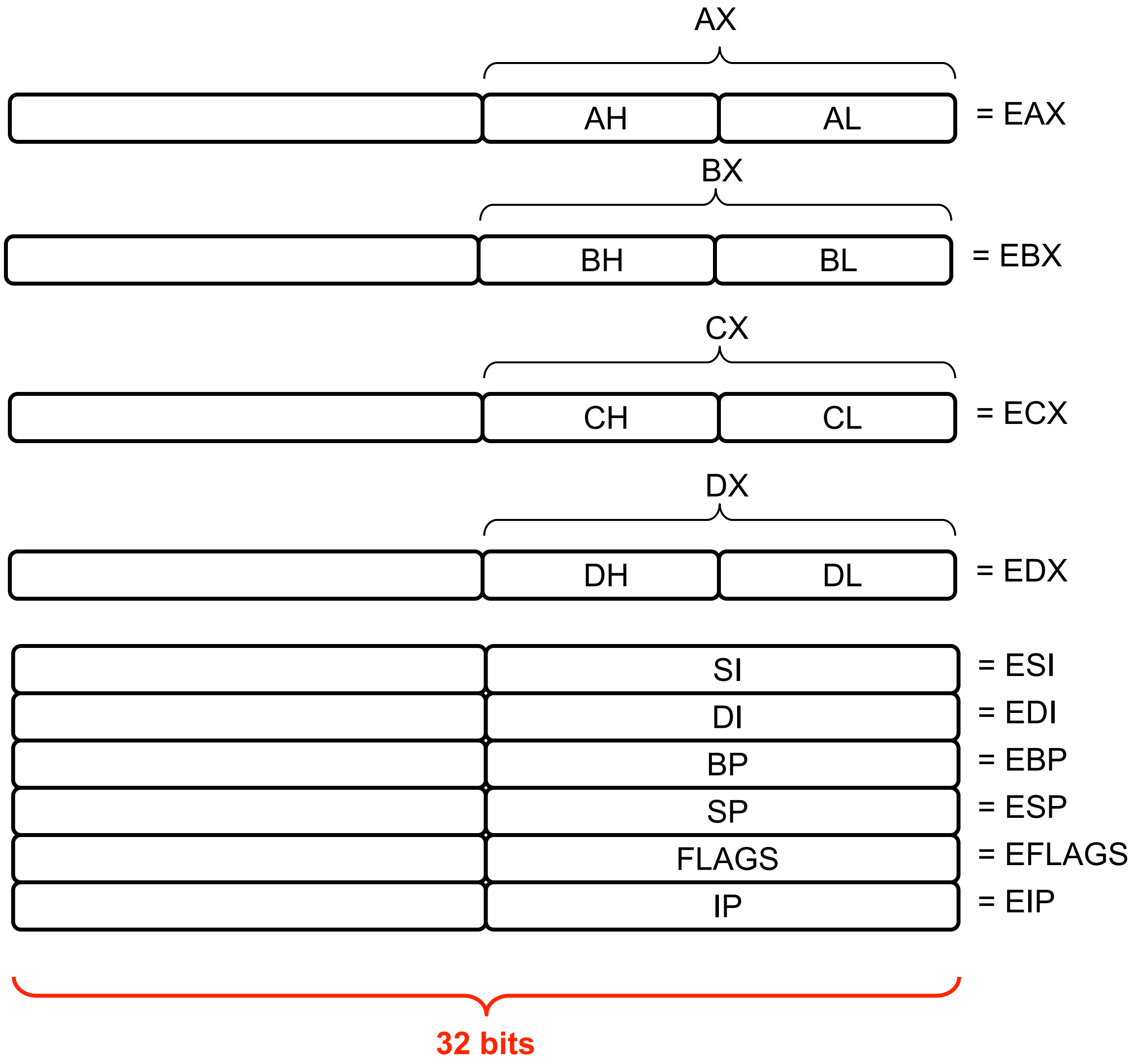
- With the 80386 Intel introduced a processor with 32-bit registers
- Addresses are 32-bit long
  - Segments are 4GiB
  - Meaning that we don’t really need to modify the segment registers very often (or at all), and in fact we’ll call assembly from C so that we won’t see segments at all (you can thank me later)

- 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



“But my machine is 64-bit”

- We now all have 64-bit machines
- So you may wonder why we’re using a 32-bit architecture
    - Of course, a 64-bit machine can handle 32-bit code
- Basically, for what we need to do in this course it does not matter *whatsoever*
    - For the code we’ll write, we wouldn’t learn anything interesting/different by going from 32-bit to 64-bit
- Going to 64-bit would just add more things that are conceptually the same
    - e.g., we’d have 64-bit RAX, RBX, etc. registers that each contain EAX, EBX, etc.
    - just like EAX, EBX, etc. contain AX, BX, etc.
- So for now I am sticking to 32-bit x86

Conclusion

- From now on I’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
        - So, really, they are not variables at all, which will be painfully clear as you do programming assignments
- We’re ready to move on to writing assembly code for the 32-bit x86 architecture
- But before, you have a screencast to watch before the next lecture…
    - Let’s start this now in case we have time remaining today