NASM Basics I
Registers and Basic Instructions

ICS312

Machine-Level and Systems Programming

Henri Casanova (henric@hawaii.edu)

Outline

- 32-bit x86 registers
- x86 basic instructions

The 8086 Registers

- To learn assembly for an ISA you must know the name of registers
  - Because registers are places in which you put data on which 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 16-bit 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 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

32-bit x86

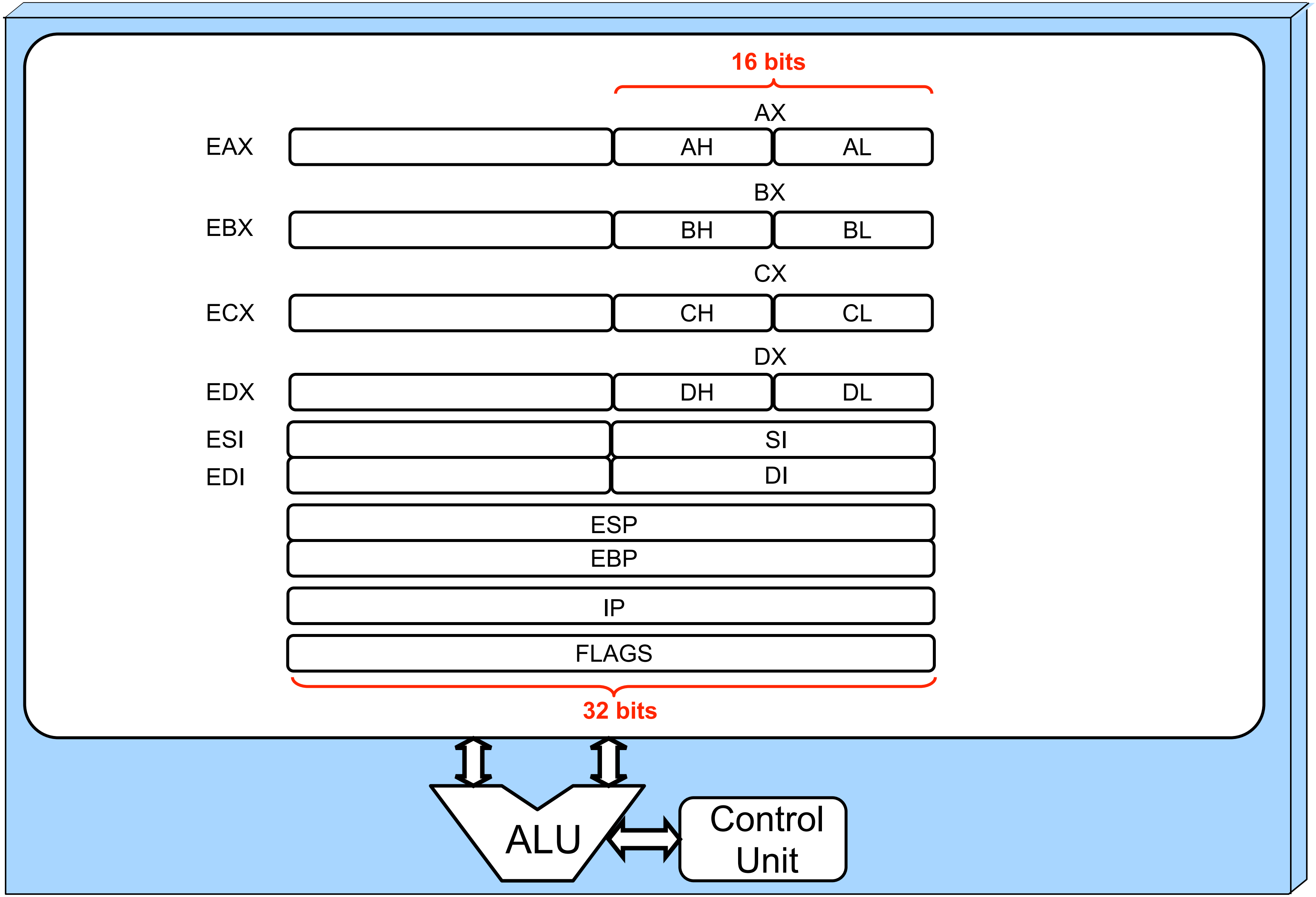
- With the 80386 Intel introduced a processor with 32-bit registers
- Addresses are 32-bit long

- Let’s have a look at the 32-bit registers…

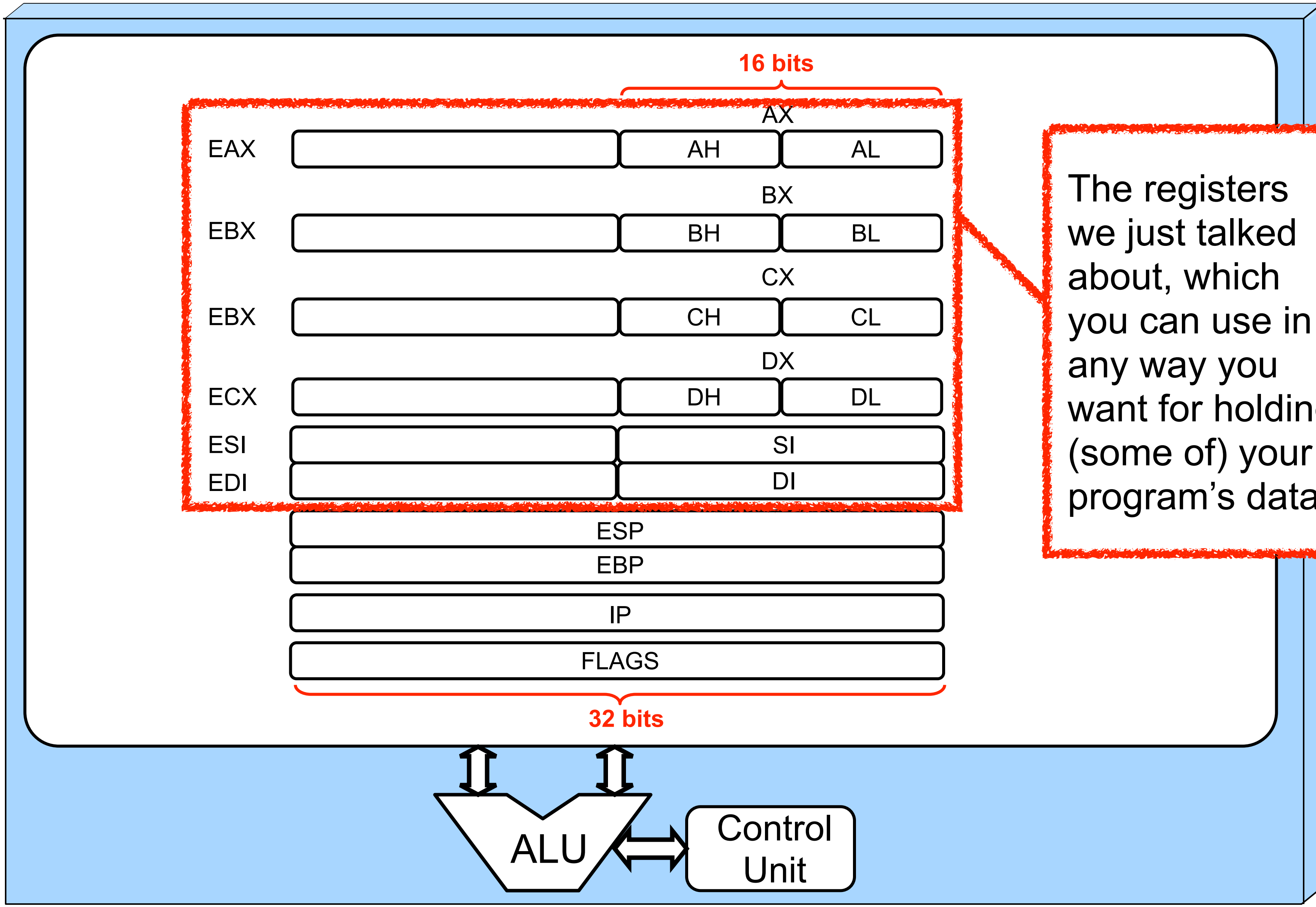
The 80386 32-bit registers

- 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
  - For backward compatibility, the previous names are used to refer to the low 16 bits
- Let's see all this on a picture…

The 32-bit x86 Registers

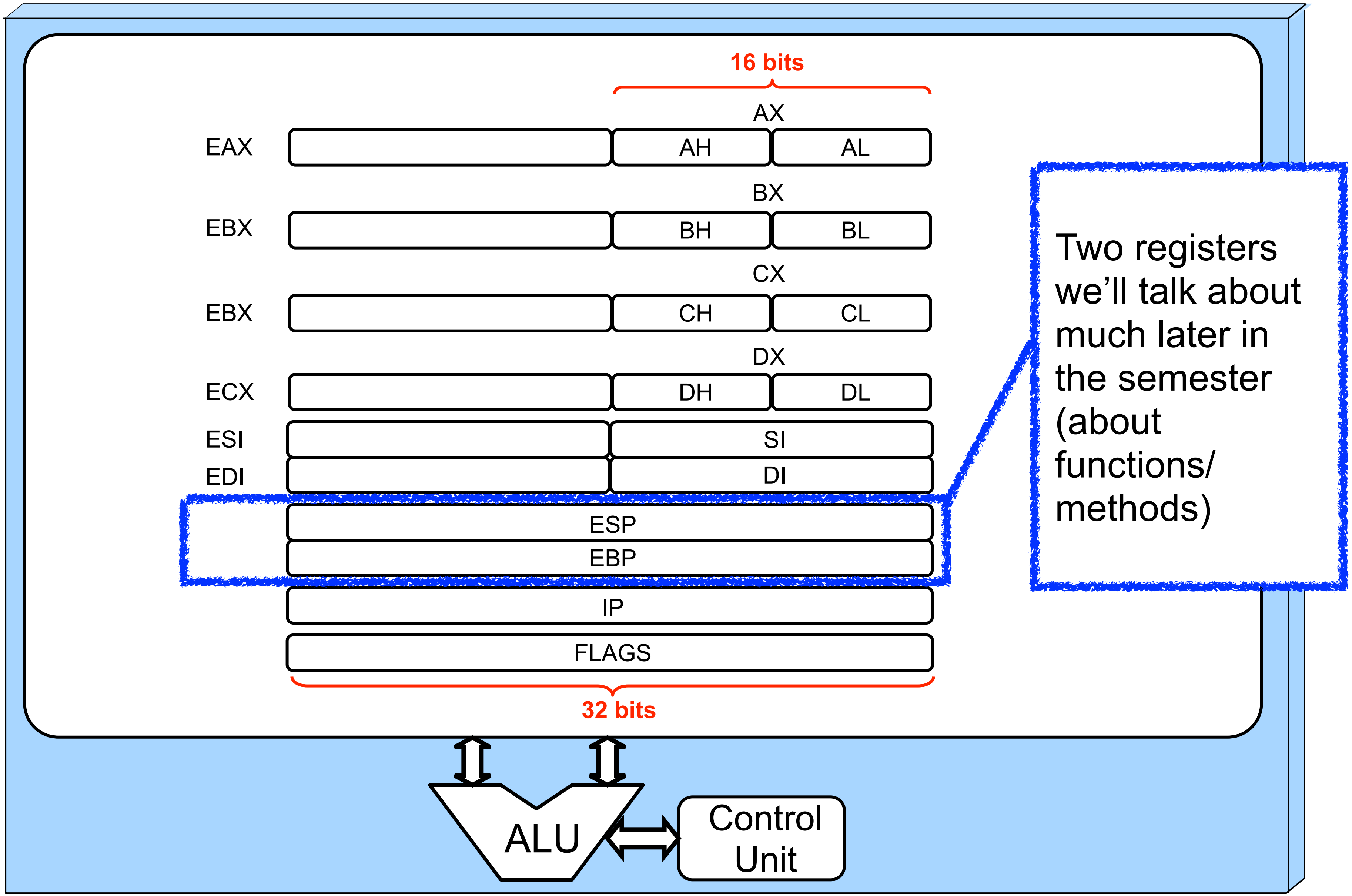


The 32-bit x86 Registers

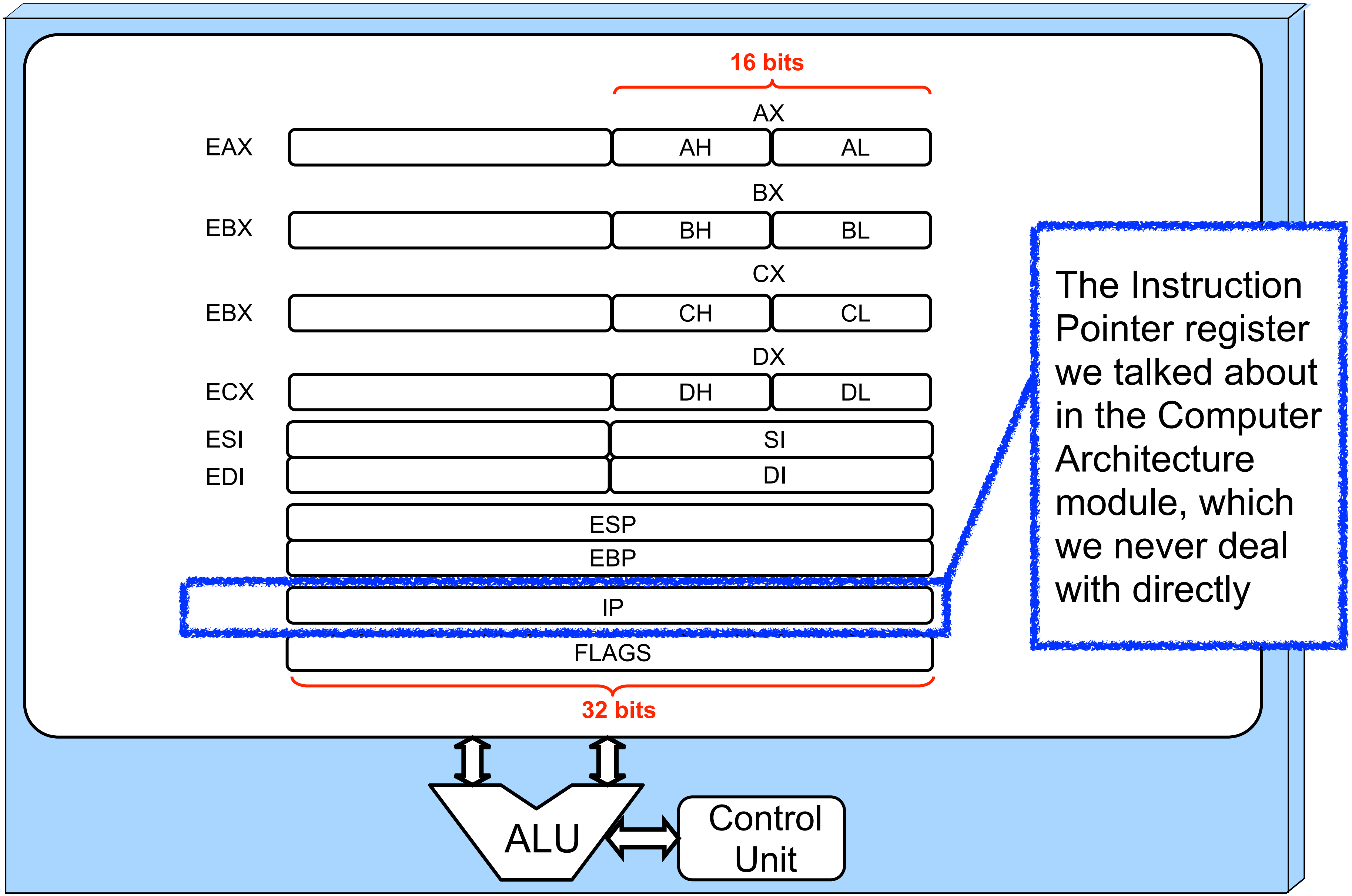


The registers we just talked about, which you can use in any way you want for holding (some of) your program's data

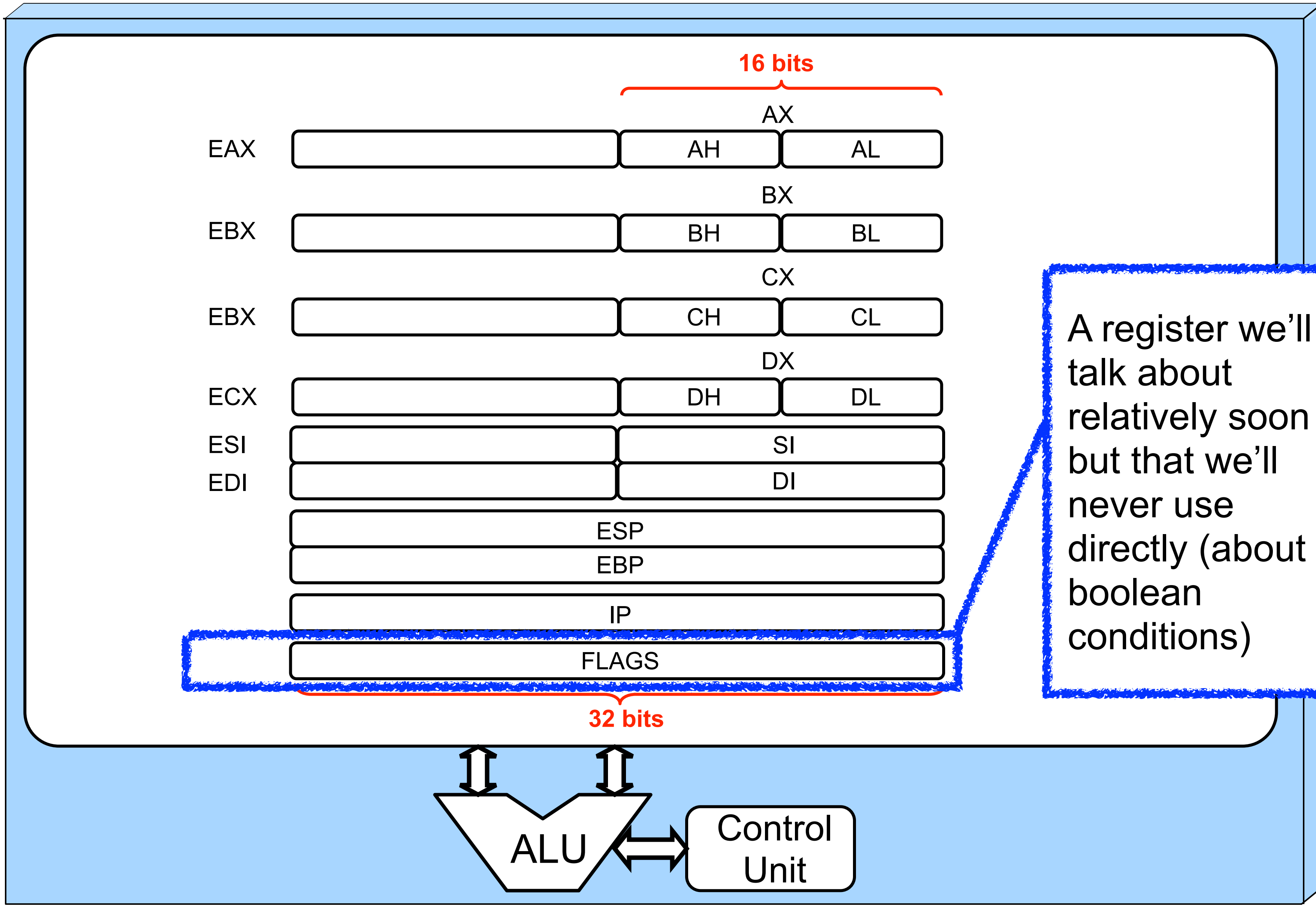
The 32-bit x86 Registers



The 32-bit x86 Registers

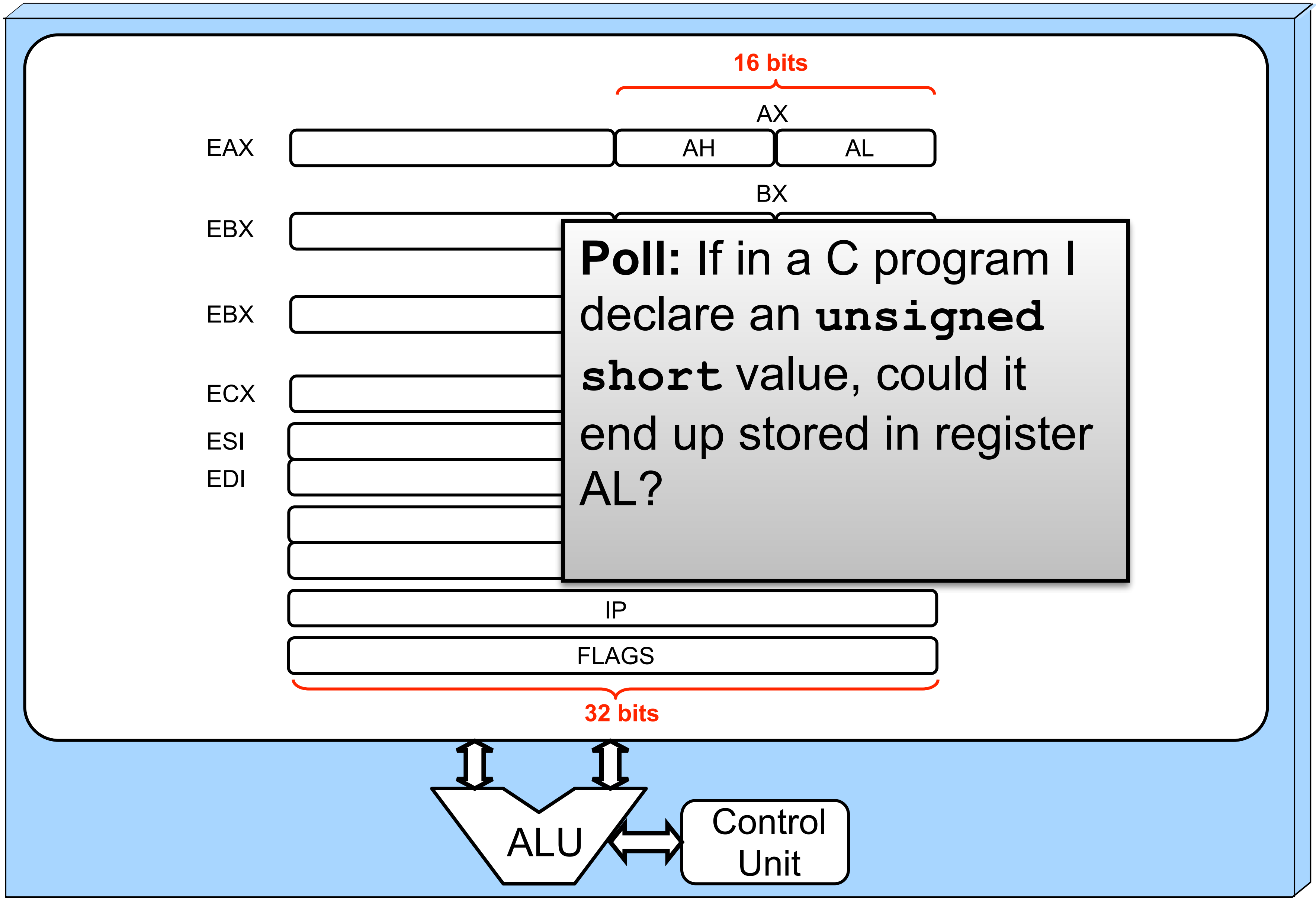


The 32-bit x86 Registers

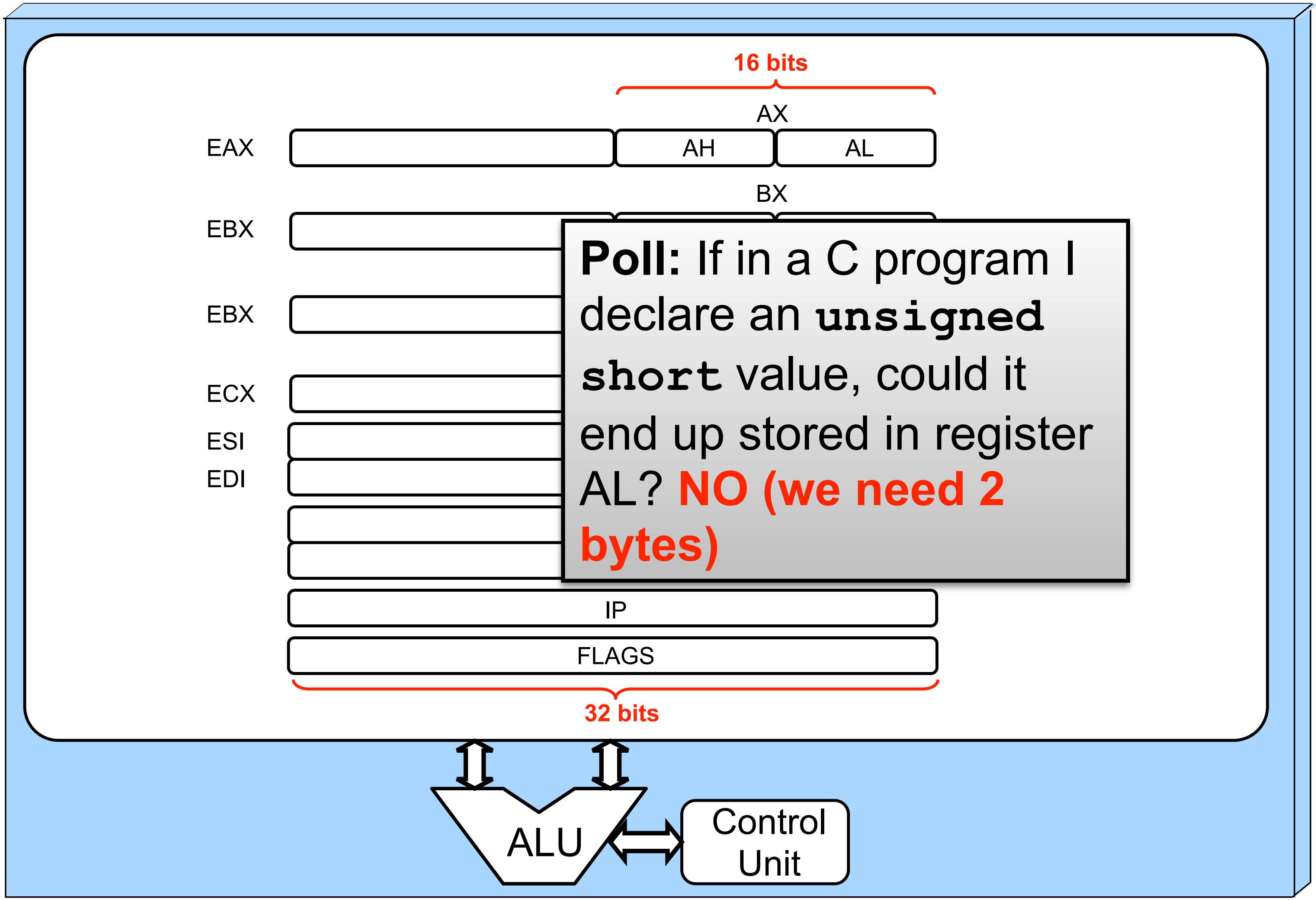


A register we'll talk about relatively soon but that we'll never use directly (about boolean conditions)

The 32-bit x86 Registers

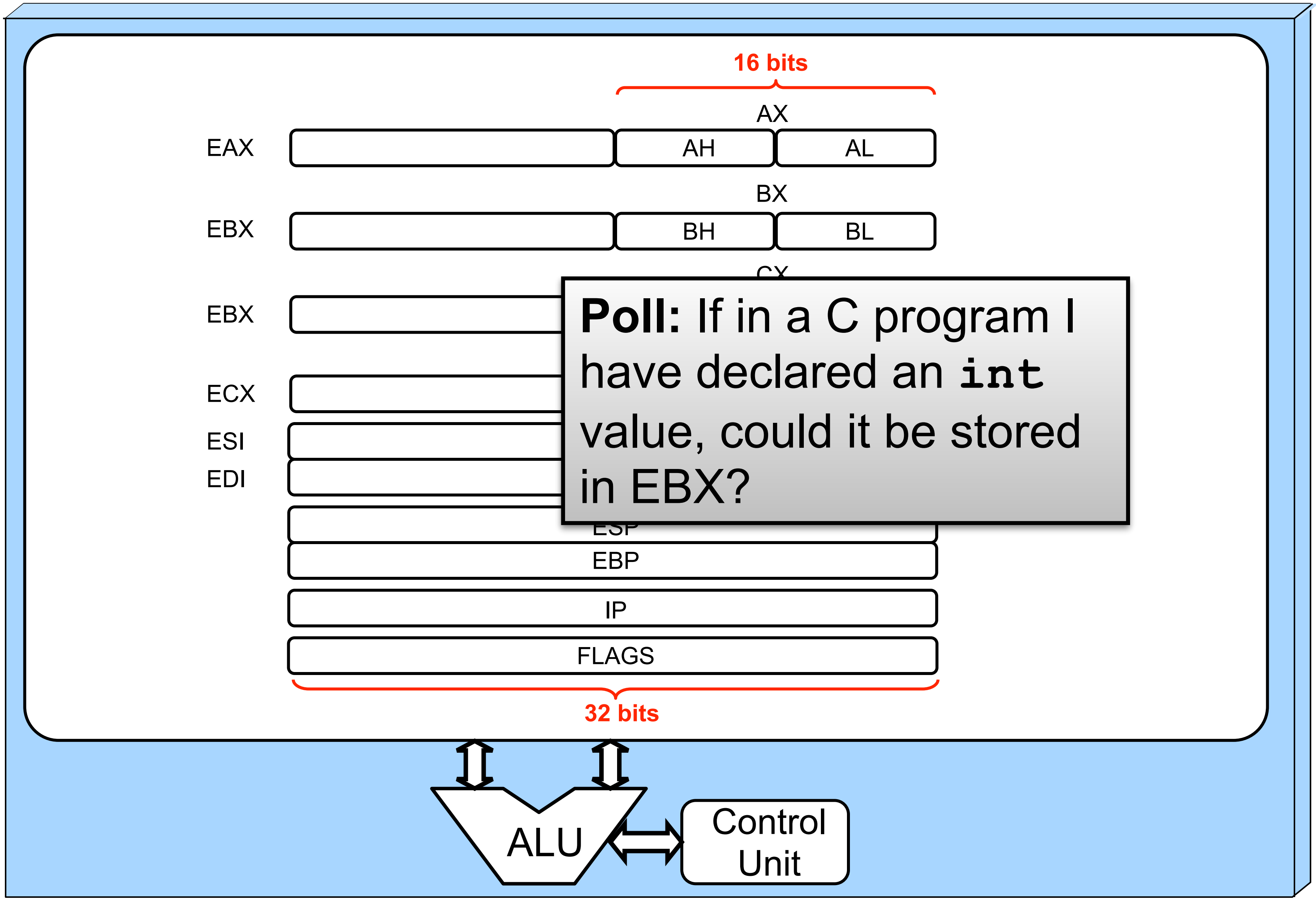


The 32-bit x86 Registers



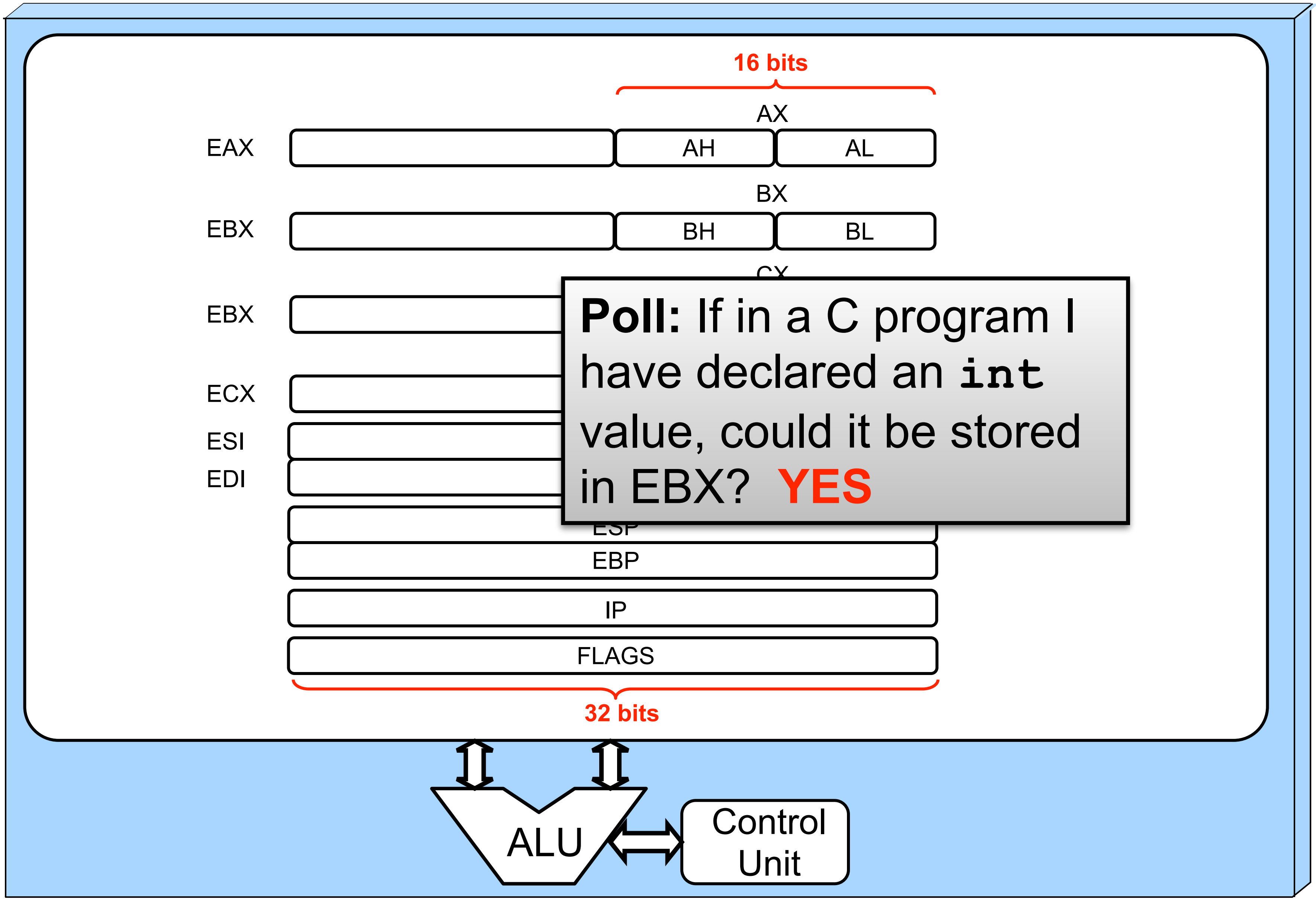
Poll: If in a C program I declare an **`unsigned short`** value, could it end up stored in register AL? **NO (we need 2 bytes)**

The 32-bit x86 Registers



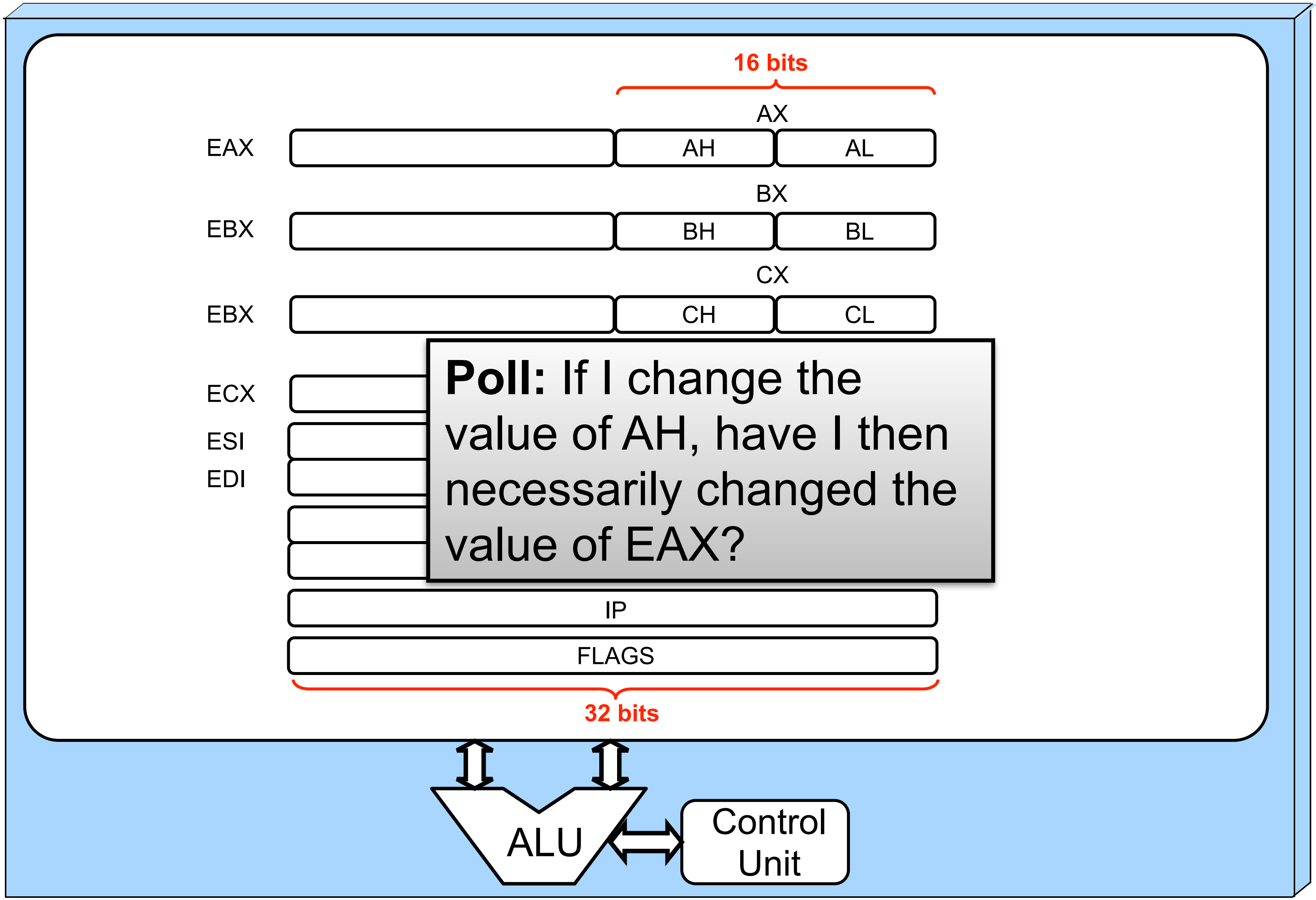
Poll: If in a C program I have declared an `int` value, could it be stored in EBX?

The 32-bit x86 Registers



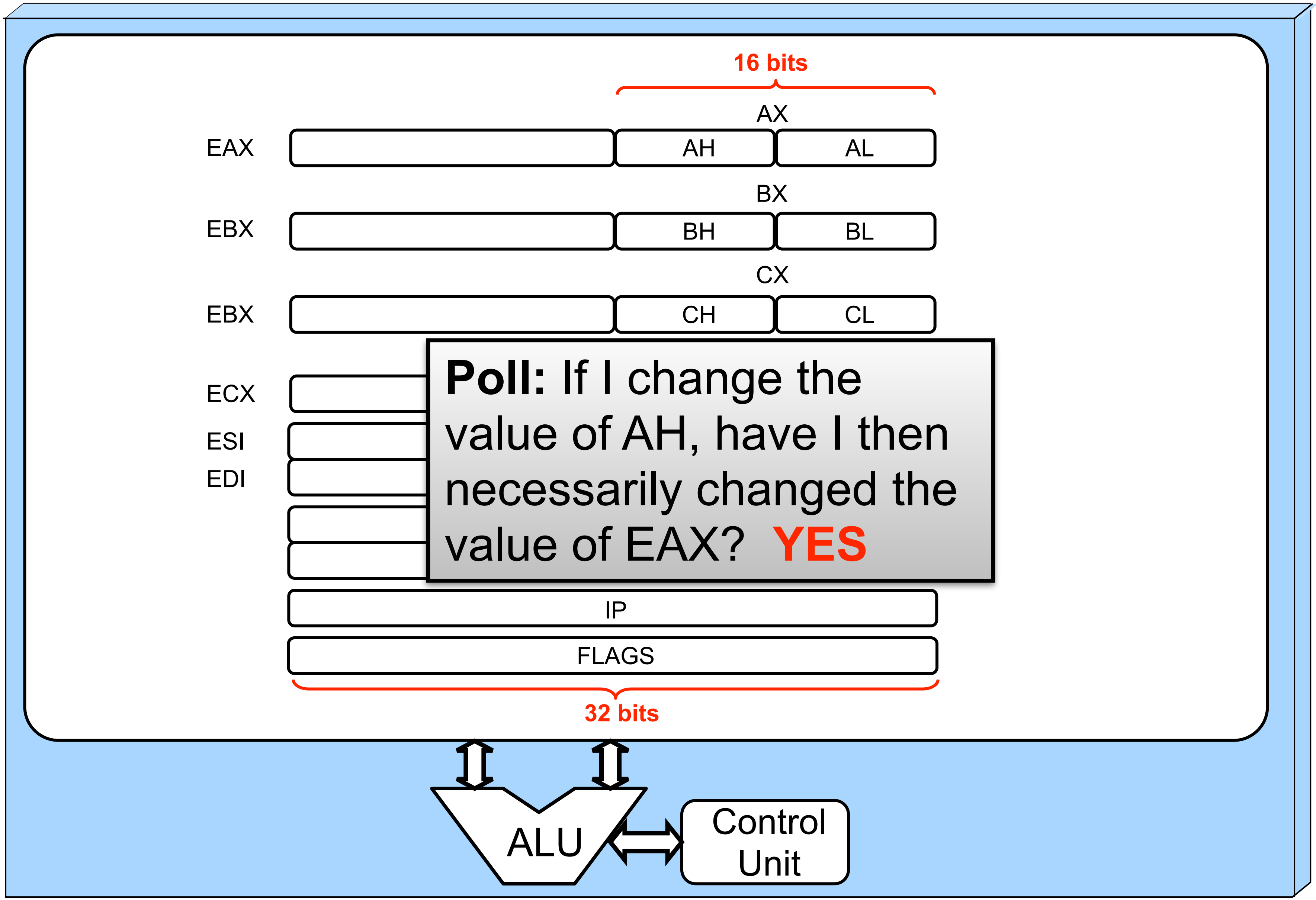
Poll: If in a C program I have declared an `int` value, could it be stored in EBX? **YES**

The 32-bit x86 Registers



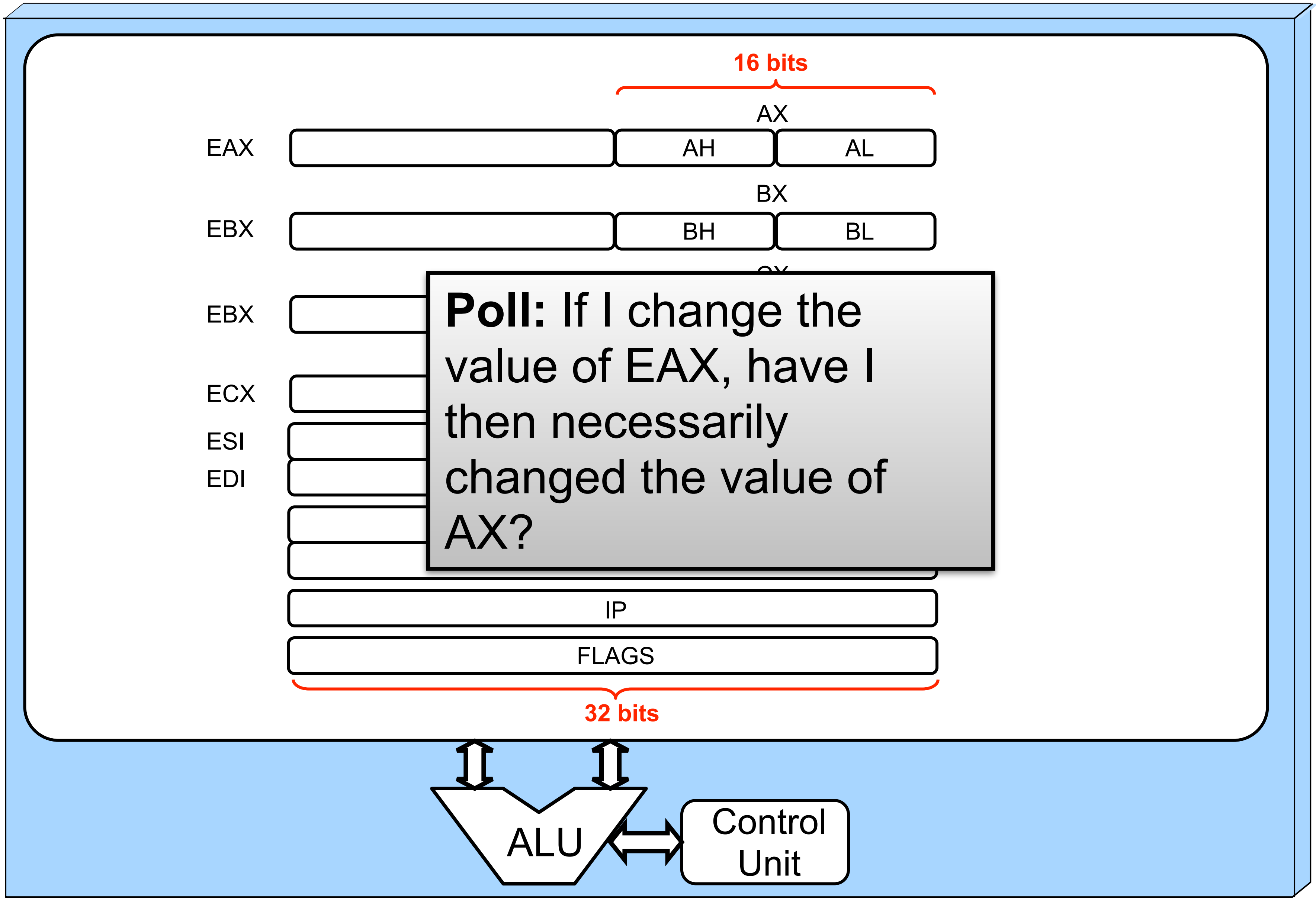
Poll: If I change the value of AH, have I then necessarily changed the value of EAX?

The 32-bit x86 Registers



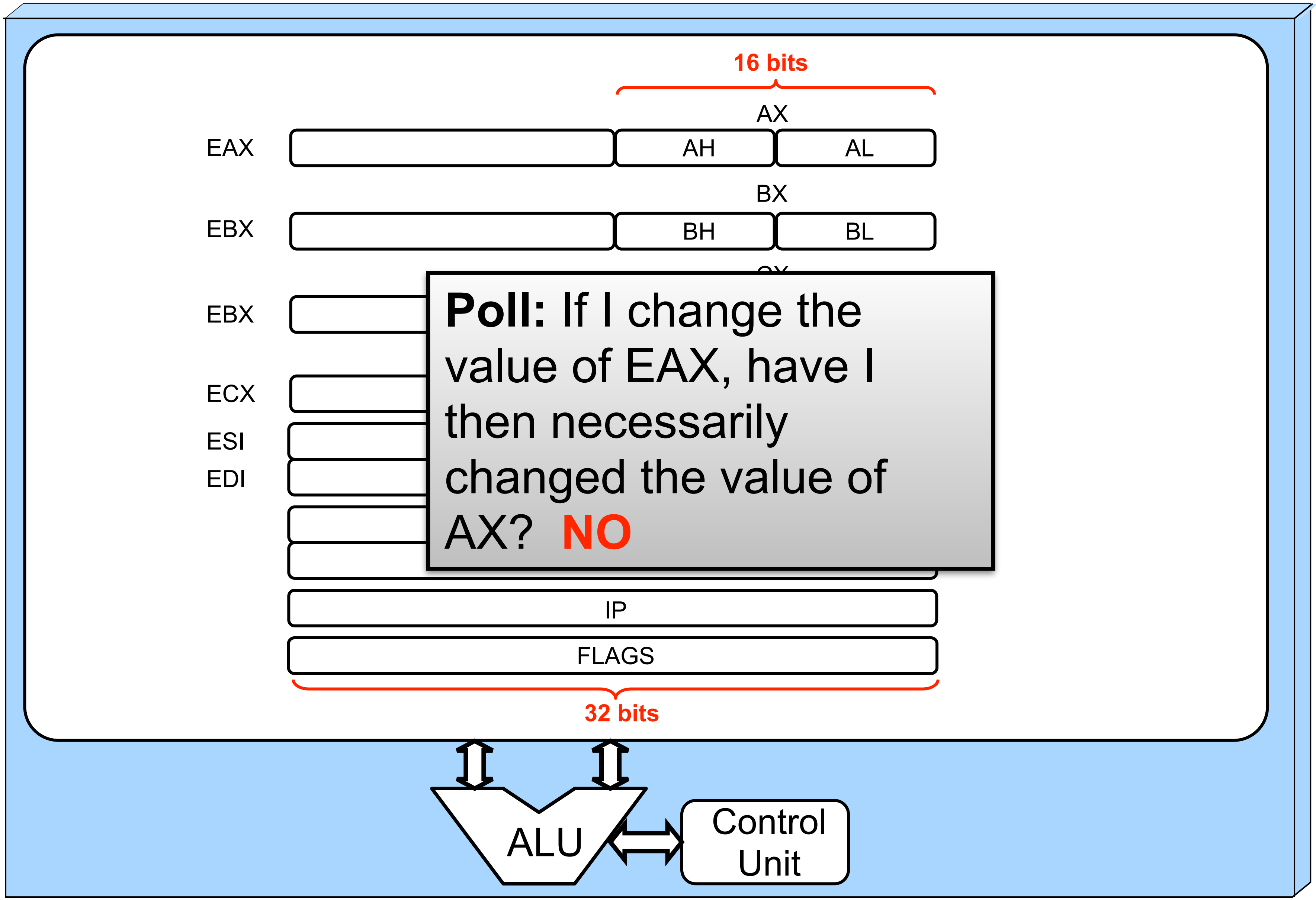
Poll: If I change the value of AH, have I then necessarily changed the value of EAX? **YES**

The 32-bit x86 Registers



Poll: If I change the value of EAX, have I then necessarily changed the value of AX?

The 32-bit x86 Registers



Poll: If I change the value of EAX, have I then necessarily changed the value of AX? **NO**

“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 in this course I am sticking to 32-bit x86

Registers are NOT Variables

- It's tempting to think of the registers as variables
- But they have **no data type** and you can do absolutely whatever you want with them, including horrible mistakes
- So, really, **registers are not variables**, which will be painfully clear when we write assembly

Outline

- 32-bit x86 registers
- x86 basic instructions (NASM)

NASM

- In this course we write assembly using NASM (https://www.nasm.us), which uses the Intel syntax
- The alternative is the ATT syntax
- The two are conceptually completely equivalent, and learning one when one knows the other takes almost zero time
- Here is sample code (we don't know these instructions yet!)

Intel

```
    mov eax, 0
    mov ecx, 1

 .loop:
    add eax, ecx
    add ecx, 1
    cmp ecx, 6
    jne .loop
```

ATT

```
 movl $0, %eax
 movl $1, %ecx

 .loop:
     addl %ecx, %eax
     addl $1, %ecx
     cmpl $6, %ecx
     jne .loop
```

Instruction operands

- Most x86 instructions take in operands
- An operand can be:
  - A register: the operand is the value stored in the register (e.g., ax, ebx)
  - A memory reference: the operand is a value stored in RAM at some address (see the next set of lecture notes)
  - An immediate: an integer constant only as an input operand (e.g., 12, -65)
  - Implicit: always the same operand only as an input operand (see the **`inc`** instruction on the next slide)
- In the Intel syntax, when an operand is a destination, it is listed first and followed by the source operands
- Operand sizes (in bytes) typically must match
- Let's see the **`add`** and **`inc`** instructions….

Adding values with `add`/`inc`

- The `add` instruction: `add op1, op2`
- High-level code equivalent: `op1 = op1 + op2`

```
add eax, ebx     ; eax = eax + ebx (a 4-byte operation)
add dl, cl       ; dl = dl + cl (a 1-byte operation)
add edx, edx     ; edx = edx + edx (a 4-byte operation)
add eax, dx      ; invalid (non-matching sizes)
add eax, 12      ; eax = eax + 12 (a 4-byte operation)
add 12, eax      ; invalid (a constant can't be changed)
```

- The `inc` instruction: `inc op1`
- High-level code equivalent: `op1 += 1`

```
inc eax          ; eax += 1 (a 4-byte operation)
inc al           ; al += 1 (a 1-byte operation)
```

Immediate Operands and Bases

- Immediate operands can be written in decimal (default), binary (b), hex (h), or octal (o), with one syntactic oddity for hex

```
add eax, 12        ; 12 in decimal (as a 4-byte value)
add eax, 1100b     ; 12 in binary (as a 4-byte value)
add eax, 14o       ; 12 in octal (as a 4- byte value)
add eax, 0Ch       ; 12 in hex (as a 4- byte value
```

There must always be a leading 0 for hex!

Even here! It’s because otherwise a hex number could look like a label (see later)

```
add eax, 0AABBCCDDh    ; NOT a 4.5-byte value :)
```

High-level Language

- Say in Java you have code like:

```
short x;
short y;
…
x = x + y;
```

- The compiler will generate byte-code, that will get interpreted by the JVM, that will then execute this machine instruction:

```
add  ax, bx
```

(Assuming the interpreter has stored the value of x in ax and the value of y in bx)

sub, dec, and neg

- Subtraction: just like we have **add** and **inc**, we have **sub** and **dec**
  - They work exactly the same
- There is a “negate” instruction: **neg**

```
mov al, 0FEh   ; FE: signed: -2     unsigned: +254
neg al         ; 02: signed: +2     unsigned: +2
```

```
mov al, 012h   ; 12: signed: +18     unsigned: +18
neg al         ; EE: signed: -18     unsigned: +238
```

The `neg` instruction

- The neg instruction is a very simple instruction to implement in hardware:
  - Flip the bits
  - Add one
- Of course, it does “nonsense” for an unsigned value
- So, when in C you have code like:

```
x = -x;
```

- The compiler will simply generate something like:

```
neg  ebx
```

(Assuming the compiler had “decided” to store the value of variable x in 4-byte register EBX)

Other arithmetic operations

- Multiplication and Division are more cumbersome, and we'll talk about them later, as needed

- There are bitwise operations, and we'll have a whole module on them

- So for now, let's stick to adding and subtracting (and negating signed numbers)

The `mov` Instruction

- The **`mov`** instruction: **`mov op1, op2`**
- Copies the value of op2 into op1 (yes, “mov” isn’t a great name)
- The first operand can be a register or a memory location
- The second operand can be a register, a memory location, or an immediate value
- Both operands must be the exact same size

```
mov eax, ebx      ; eax = ebx (a 4-byte copy)
mov dl, cl        ; dl = cl (a 1-byte copy)
mov edx, dl       ; invalid (non-matching sizes)
mov cl, eax       ; invalid (non-matching sizes)
mov eax, 0FFh     ; eax = 255 (a 4-byte value,
                    without all leading zeros written)
mov al, 0FFh      ; al = 255 or -1 (a 1-byte value)
mov 42, dl        ; invalid (can't change constant)
```

Paying attention to registers

- Let's consider this fragment of code

```
mov  ebx, 011223344h
mov  ax, 0F111h
add  bx, ax
```

In-class exercise: what's the value of register ebx after these three instructions?

Paying attention to registers

- Let's do it step by step (all numbers in hex)

```
mov  ebx, 011223344h
   ; ebx = 11 22 33 44 (bx = 33 44)
mov  ax, 0F111h
   ; ax = F1 11  (eax = ?? ?? F1 11)
add  bx, ax
   ; a 2-byte addition: 33 44 + F1 11
   ; with a result of 24 55 (carry dropped)
   ; and so ebx = 11 22 24 55

   ; (AND NOT 11 23 24 55, which would be
                             wrong and horrible)
```

Important Takeaways

- The set of X86 registers we can use:
  - eax (ax, ah, al), ebx (bx, bh, bl), etc.
  - esi, edi
- Different possible operands to instructions:
  - registers, memory refs, immediate, implicit
- The **`add/inc`** and **`sub/dec`** instructions
- The **`neg`** instruction
- The **`mov`** instruction

Conclusion

- Let’s do some of the posted practice problems….
- Before we can write any useful assembly, we have to learn about using the memory in addition to using the registers
- This is the topic of the next set of lecture notes!