IA32 Instruction Set

- General Purpose Register instruction set architecture
  - many general registers
  - there are also some registers with specific uses
- Basic Instruction types:
  - arithmetic/logical
    - add, subtract, and, or, etc.
  - control
    - changing which instruction executes next
  - data movement
    - copying values from one location to another.

Operands

- There are three ways of specifying operands:
  - **register**: operand value is contained in a register
  - **immediate**: operand value is a constant that is encoded as part of the instruction
  - **memory**: operand value is in memory

- Integer operands can be 8, 16 or 32 bits.
- Floating point operands can be 32, 64 or 80 bits.

Integer Registers

- Goofy names:

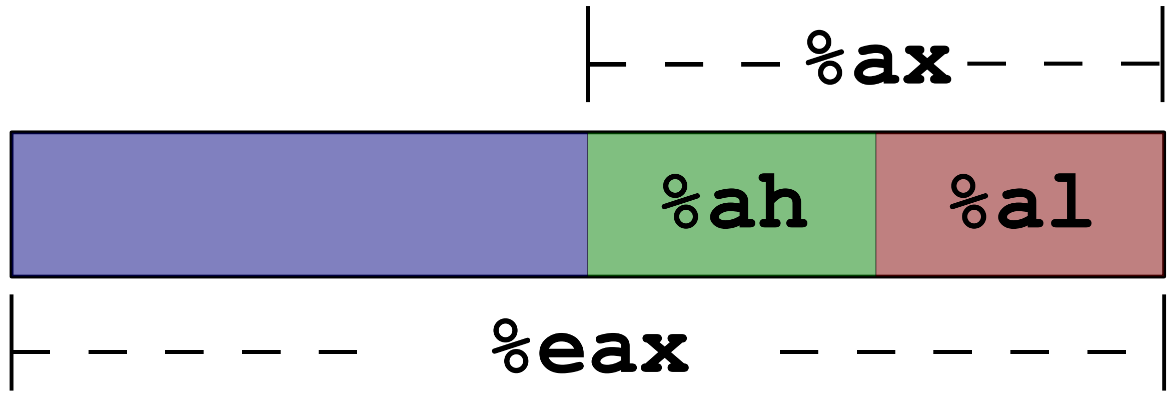
`%eax`	**`%ebx`**	**`%ecx`**	**`%edx`**
`%esi`	**`%edi`**	**`%ebp`**	**`%esp`**

- These are all 32 bit registers.
- **`%ebp`** and **`%esp`** are *special*
  - there are special uses for these, they are typically not used as general purpose registers.

8, 16 and 32 bit registers

- Instead of providing different registers for different operand sizes, there are names for smaller parts of some of the 32 bit registers.
  - this provides compatibility with **`x86`** (16 bit) instruction set.
  - Keep in mind that whenever you change an 8 bit register you are also changing the corresponding 32 bit register!

Registers %eax, %ax, %ah, %al

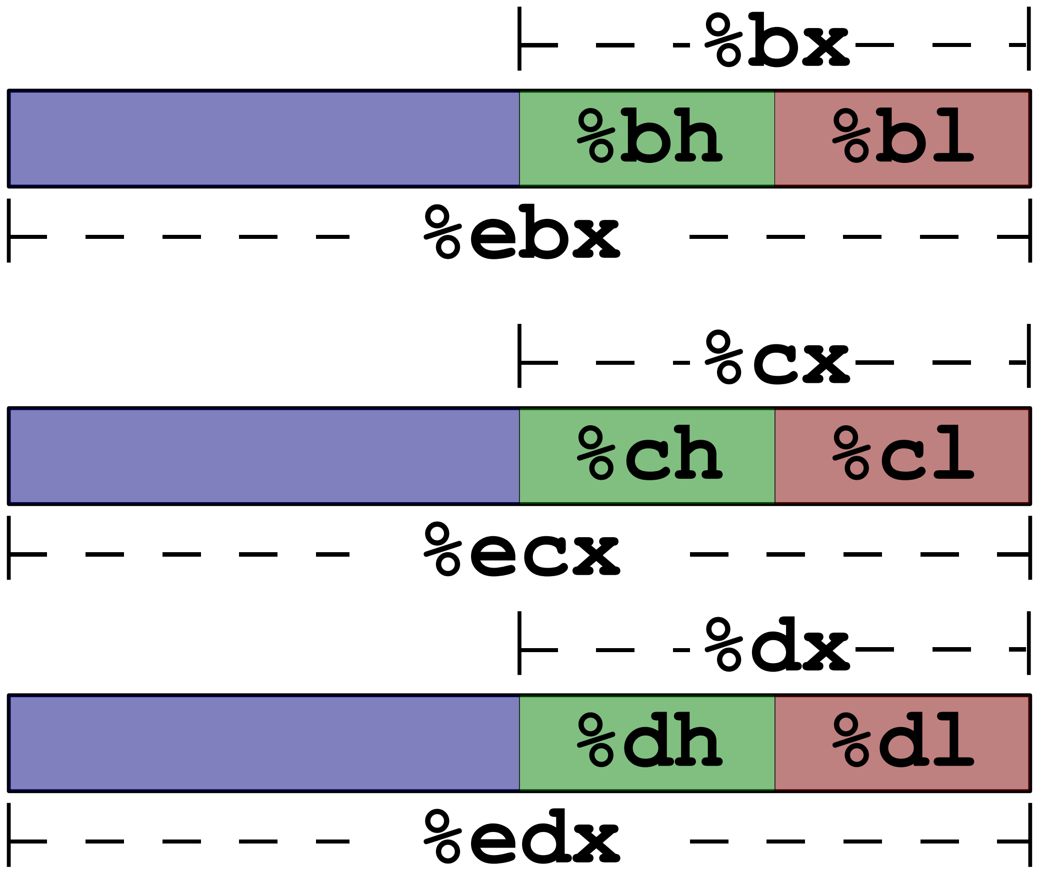


- **`%eax:`** 32 bit register
- **`%ax:`** 16 bit register, *ls* 16 bits of **`%eax`**
- **`%al:`** **8** bit register, *ls* byte of **`%eax, %ax`**
- **`%ah:`** **8** bit register, *ms* byte of **`%ax`**

  also second *ls* byte of **`%eax`**

%ebx, %ecx, %edx

- We also have names for parts of registers %ebx, %ecx, %edx:



First Instruction: `addl`

`addl` ***`srcreg, dstreg`***

- treats the contents of both registers as 32 bit integers.
- adds the contents of the two registers and stores the result in dstreg.
- the original value in *`dstreg`* is overwritten!
- examples:

```
add %ebx, %eax
add %edx, %esi
```

`add` machine code

- There are 8 different possible registers
  - it takes 3 bits to encode a choice from 8 different *things*
- add uses two registers
  - need at least 6 bits to specify the *operands*
- There must also be some bits to distinguish and add instruction from other instructions...

We are not that concerned with machine code, but it's good to keep track of what needs to be encoded in an instruction.

Adding bytes (8 bit registers)

`addb` ***`srcreg, dstreg`***

- treats the contents of both registers as 8 bit integers.
- adds the contents of the two registers and stores the result in ***`dstreg`***.
- the original value in ***`dstreg`*** is overwritten!
- example:

`add %bh, %al`

this changes the ls byte of register **`%eax`***!*

Assemblers

- An assembler is a program that converts from assembly language to machine code.
- Some IA32 assemblers allow you to do this:

`add %eax,%ebx` ← Same as **`addl %eax, %ebx`**

`add %al, %al` ← Same as **`addb %al, %al`**

- The assembler figures out the operand size from the register names used.

Another Instruction: **`and`**

`and` ***`srcreg, dstreg`***

- bitwise logical and of the contents of the two registers and stores the result in ***`dstreg`***.
- the original value in ***`dstreg`*** is overwritten!
- examples:

```
and %ebx, %eax
and %edx, %esi
```

- corresponds to the C **`&`** operator.

Subtraction

`sub` ***`srcreg, dstreg`***

- treats the registers as 32 bit integers, and subtracts ***`srcreg`*** from ***`dstreg`***, stores the result in ***`dstreg`***.

  `dstreg = dstreg - srcreg`

- the original value in ***`dstreg`*** is overwritten!
- examples:

```
sub %ebx, %eax
sub %edx, %esi
```

Other arithmetic/logic instructions

- Same format as add, sub:

op srcreg, dstreg

`imull`: integer multiplication

`or`: bitwise logical or

`xor`: bitwise logical exclusive or

Shift Instructions

sal ***srcreg, dstreg***

shift arithmetic left

dstreg = dstreg << srcreg

sar ***srcreg, dstreg***

shift arithmetic right : sign bit extended

dstreg = dstreg >> srcreg

shr ***srcreg, dstreg***

shift logical right : shift in 0's

dstreg = dstreg >> srcreg

Exercise: IA32 Assembly program

- We can build a sequence of assembly instructions to perform some compuatation.
- We have not yet established how registers initially get a value, for now we assume that they have some value.
- Compute

$$y = 2y – x + z$$

- Assume:

`%eax` holds y, **`%ebx`** holds x, **`%ecx`** holds z

One Solution

$y = 2y - x + z$
y: **%eax**
x: **%ebx**
z: **%ecx**

```
add %eax, %eax      # eax = 2y
sub %ebx, %eax      # eax = 2y - x
add %ecx, %eax      # eax = 2y – x + z
```

comments

Possibly Wrong Solution

$y = 2y - x + z$
y: **`%eax`**
x: **`%ebx`**
z: **`%ecx`**

```
add %eax, %ecx        # ecx = y + z
add %ecx, %eax        # eax = 2y + z
sub %ebx, %eax        # eax = 2y + z - x
```

The problem is that **`%ecx`** no longer holds the value of z!

Quiz: What does this do?

```
xor %eax, %eax
```

IA32 integer Arithmetic

Do **`add`** and **`sub`** instructions deal with signed or unsigned integers?

YES!

Recall that the actual bit manipulations necessary for signed/unsigned addition are identical !

Subtraction is really just addition:

$$x - y = x + (-y)$$

Immediate Operands

- An *immediate* operand is a constant (a number)
  - the actual bit representation is part of the machine code for the instruction.
- In IA32 assembly language, immediate operands are prefixed with '$'
- Default is decimal, you can also use hex using the same syntax as with C.

```
$100        $0x80    $-35
```

Immediate operand usage

op srcreg, dstreg

- You can use an immediate operand in the place of ***srcreg***, but not ***dstreg***
  - it doesn't make sense to say something like:

    `add %eax, $24` since this is saying: 24 = 24 + eax
- Some examples:

```
add $1, %eax   # %eax = %eax + 1
sub $5, %bh    # %bh = %bh - 5
```

Another quiz: what does this do?

```
xor %eax, %eax
add $13, %eax
```

Machine code issues

- For instructions that include an immediate operand, the machine code must include the immediate value.
  - depending on the value, it may require 8, 16 or 32 bits in the actual machine code for the instruction.
  - just saying...

Another quiz? Already?

```
xor %ebx, %ebx
or  %eax, %ebx
and $0x80, %ebx
```

Moving data: `mov` instruction

`mov` ***`src, dstreg`***

- moves data specified by ***`src`*** to the destination register ***`dstreg`***.
  - really *copies* the data.
  - If ***`src`*** is a register it is not modified or *emptied*
    - there is no such thing as *emptied*, every register always has some value!

`mov` examples

```
mov %eax, %ebx        # %ebx = %eax

mov $22, %eax         # %eax = 22

mov $22, %ah          # %ah = 22

mov $65535, %al       # ? no idea!
```

`%al` won't hold a 16 bit value

Quiz-mania

$$y = y - (x^2 + 3)$$

Solution-mania

$y = y - (x^2 + 3)$
y: `%eax`
x: `%ebx`

```
mov %ebx,%ecx           # %ecx = x (a copy)
imull %ecx,%ecx         # %ecx = x^2
add $3, %ecx            # %ecx = x^2+3
sub %ecx, %eax          # %eax = y -(x^2+3)
```

Note that using `%ecx` to hold the intermediate value means that `%ebx` is still x. Sometimes this is important (sometimes it isn't – perhaps we don't need x for anything else).

Memory Operands

- Many instructions support using operands that are located in memory.
  - we always need to specify the *address* of the operand.
- There are a number of ways to specify addresses:
  - as an absolute address (a number, like 204)
  - using a register as a pointer – the register holds the address.
  - using some simple arithmetic to compute the address (add two registers, add a number to a register, etc.)

Addressing *modes*

- An *addressing mode* is a mechanism for specifying an address.
  - **absolute**: the address is provided directly
  - **register**: the address is provided indirectly, but specifying *where* (what register) the address can be found.
  - **displacement**: the address is computed by adding a *displacement* to the contents of a register
  - **indexed**: the address is computed by adding a displacement to the contents of a register, and then adding in the contents of another register times some constant.

Absolute addressing mode

- Actual address is a constant embedded in the program:

  `add 824, %ebx`

  – adds the contents of memory location 824 to register **`%ebx`** and stores the result in **`%ebx`**

  `%ebx = %ebx + Mem[824]` ← *This is not assembly, just a way of describing what is happening*

- Recall that if we want to add 824 to **`%ebx`**, we have to say: **`add $824, %ebx`**

Register Addressing Mode

- Address is found in a register:

```
add (%eax), %ebx
```

  - adds the contents of memory location whose address is in register **%eax** to register **%ebx** and stores the result in **%ebx**

```
%ebx = %ebx + Mem[%eax]
```

- The parens around the register tell the assembler to use the register as a pointer.

Displacement Addressing Mode

- Address is computed as sum of the contents of a register and some constant displacement:

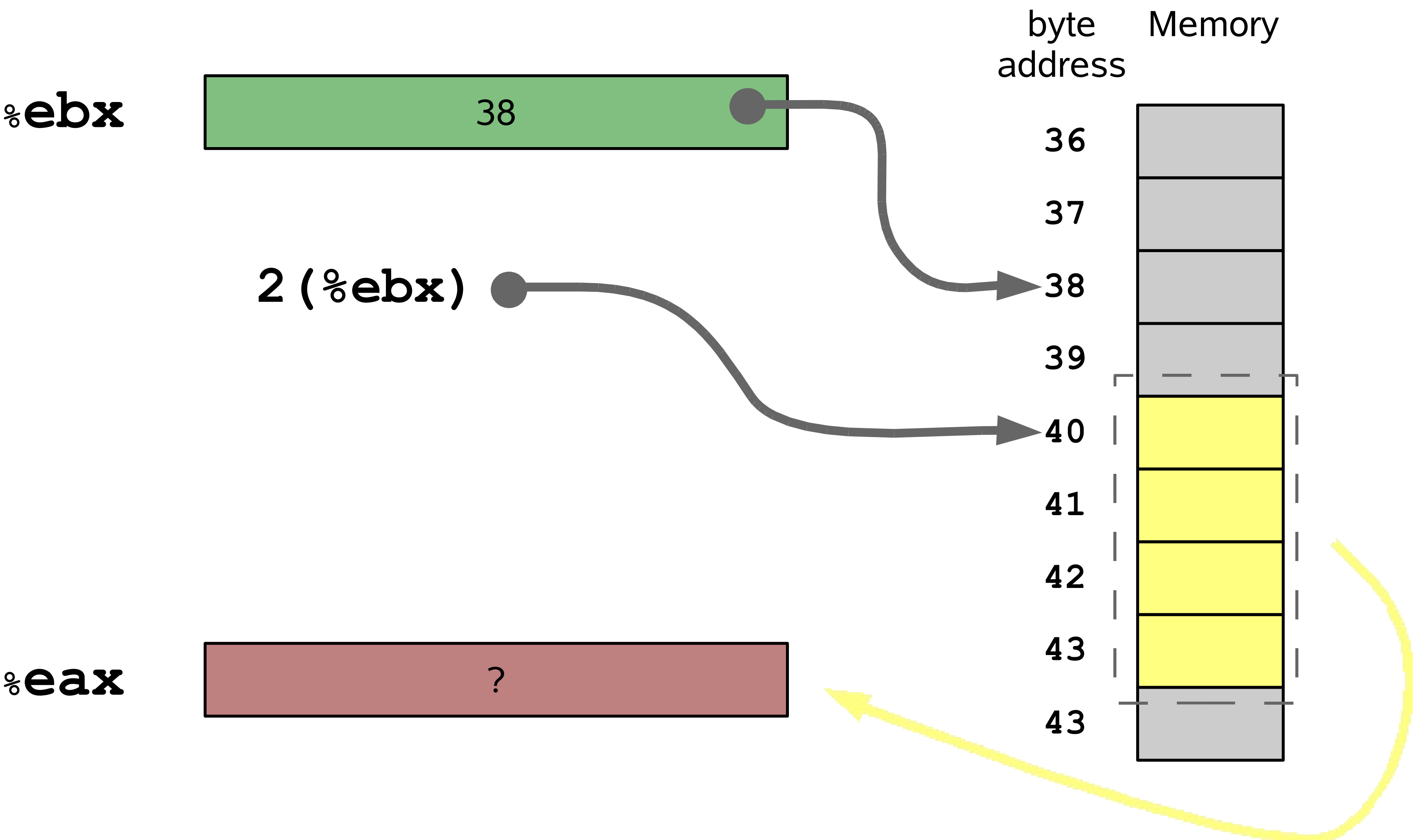
`add 45(%eax), %ebx`

  - adds the contents of memory location whose address is computed as **`%eax+45`** to register **`%ebx`** and stores the result in **`%ebx`**

`%ebx = %ebx + Mem[%eax+45]`

- The register is a pointer, the displacement specified how far from the pointer.

mov 2(%ebx),%eax



IA32

Displacement in action

```
int i=3;
int x[4];
x[0]=i;
x[1]=i+3;
```

```
mov $x,%ebx      # %ebx is x
                 # (address of array)
mov $3,%eax      # %eax is i
mov %eax,(%ebx) # put i in mem[%ebx]
add $3,%eax      # %eax is i+3
mov %eax, 4(%ebx)  # put i in mem[%ebx+4]
```

Notes:

`$x` is the address of the array (the name of an array is it's address)

displacement is 4 since each array element is 4 bytes (each is an **`int`**)

Not a quiz, an *exercise*

```
int a[3];
a[0]=0;
a[1]=1;
a[2]=2;
```

Start with this (puts the address of a in register **`%eax`**):

```
mov $a, %eax
```

Exercise Solution

```
int a[3];
a[0]=0;
a[1]=1;
a[2]=2;
```

```
mov $a, %eax
mov $0,(%eax)
mov $1, 4(%eax)
mov $2, 8(%eax)
```

This would also work

```
mov $a, %eax
mov $0,(%eax)
add $4,%eax
mov $1,(%eax)
add $4, %eax
mov $2, (%eax)
```

Dealing with bytes

- All addresses are byte addresses (each byte in memory has a unique address).
- There is nothing different about addressing a byte operand – same syntax.

```
mov  122(%ebx), %al

mov %al, 85(%esi)

add %bh, 5(%edx)
```

You may need to be explicit:

```
movb $32, 85(%esi)
addb $1, 5(%edx)
```

Some Rules

- **`mov`** and arithmetic/logical instructions cannot have two memory operands (at most one).

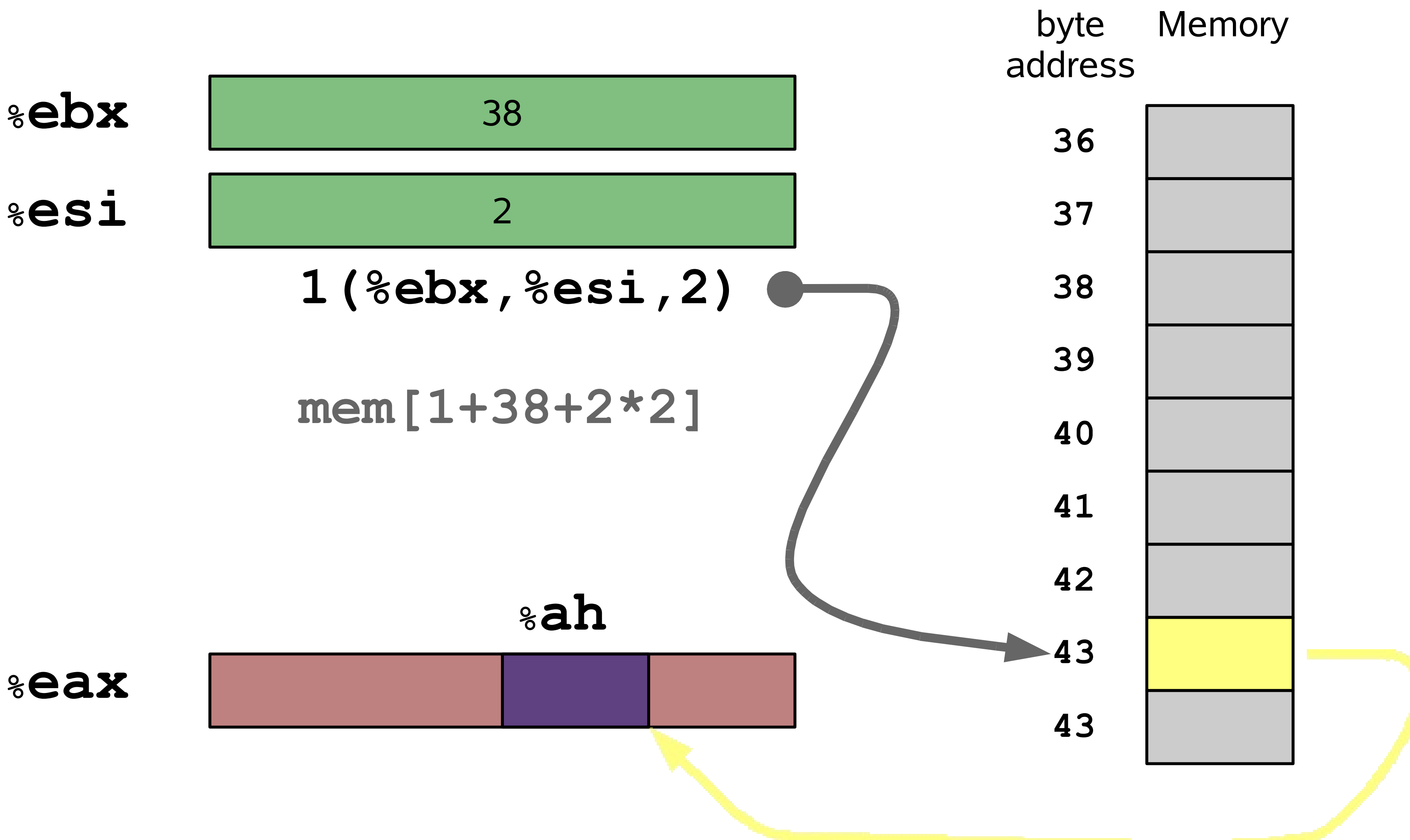
- You can't do this:
  - **`mov (%eax), 14(%eax)`**
  - **`add 100, (%esi)`**

Indexed Addressing Mode

`disp(reg1, reg2, scale)`

- Address is computed as sum of:
  - constant displacement *disp*
  - contents of register *reg1*
  - contents of register *reg2* times the *scale* factor
- **`scale`** can be 1,2,4 or 8 only.
  - size of data types.

movb 1(%ebx,%esi,2),%ah



Why Indexed?

- Indexed addressing mode seems overly complex
  - very CISCish
- There are actually times when it makes sense to use it:
  - structure field is an array.
- The real reason for it is:
  - it is really the only addressing mode, the others are all special cases!

Exercisemania

```
int a[10];
int i;
/* i gets some value */

a[i]=12;
a[i+2]=a[i+1];
```

Solution

```
a[i]=12;
a[i+2]=a[i+1];
```

assume **$a** is in **%edi** and **i** is in **%esi**

```
movl $12,(%edi,%esi,4)      # Mem[%edi+%esi*4]=12
movl 4(%edi,%esi,4),%eax    # %eax= Mem[4+%edi+%esi*4]
movl %eax,8(%edi,%esi,4)    # Mem[8+%edi+%esi*4]=%eax
```

Addressing Modes

- Indexed: ***dist(reg1, reg2, scale)***
- Absolute: ***dist***
- Register: ***(reg1)***
- Displacement: ***dist(reg1)***
- You can also do this:

```
movl (,%eax,2),%ebx    # %ebx = Mem[%eax*2]
movl (%ebx,%eax),%esi    # %esi=Mem[%ebx+%eax]
```

What does this do?

```
mov     $1, %eax
add     $3, %eax
add     $5, %eax
add     $7, %eax
```

How about this?

```
mov     %edx, %eax
add     %ecx, %edx
add     %eax, %ecx
```

OK Smartypants – try this

```
subb 'a',%al
addb 'A',%al
```

No way you figure this one out.

```
xor    %ebx,%eax
xor    %eax,%ebx
xor    %ebx,%eax
```

Fun with addressing modes: What is each address?

```
xor     %eax,%eax
add     $0x22,%eax
movl    %esi,(%eax)
addl    22,%edi
movl    0xffffffff(%eax,%eax,2),%ebx
```

Subroutines

- In C, all code is in a function.
- In Assembly, all code is in a *subroutine*.
- In general, the compiler will generate on subroutine per C function
  - exceptions: inline functions, some optimizations
- We will study the details of subroutines a little later, for now we just need to recognize a few things.

Example Subroutine

```
int increment(int x) {
    x = x + 1;
    return(x);
}
```

```
increment:
        pushl   %ebp
        movl    %esp, %ebp

        incl    8(%ebp)

        movl    8(%ebp), %eax

        popl    %ebp
        ret
```

subroutine setup

body

return value

finish

Subroutine Parameters

- Parameters are passed on *the stack*
  - *we have not yet discussed the stack*
- For now, just remember:
  - first parameter is located in memory at **`8(%ebp)`**
  - second parameter value is at **`12(%ebp)`**
  - third parameter value is at **`16(%ebp)`**
  - and so on...

Example Subroutine

```
int add(int x, int y) {
    return(x+y);
}
```

```
add:
        pushl   %ebp
        movl    %esp, %ebp

        movl    12(%ebp),%eax
        addl    8(%ebp),%eax

        popl    %ebp
        ret
```

subroutine setup

body

return value

finish

Another Subroutine

```
void incr(int *x) {
    *x++;
}
```

```
incr:
    pushl   %ebp
    movl    %esp, %ebp

    movl    8(%ebp),%eax
    incr    (%eax)

    popl    %ebp
    ret
```

subroutine setup

body

finish

SubQuiz – what does this do?

```
foo:
        pushl   %ebp
        movl    %esp, %ebp

        mov     8(%ebp),%eax
        mov     12(%ebp),%edx
        add     %edx,(%eax)

        popl    %ebp
        ret
```

subroutine setup

body

finish

IA32

Two possible functions

```
foo:
    pushl   %ebp
    movl    %esp, %ebp

    mov     8(%ebp),%eax
    mov     12(%ebp),%edx
    add     %edx,(%eax)

    popl    %ebp
    ret
```

```
/* could be either of these */
void foo(int *x, int i) {
   *x = *x + i;
}

void foo(int x[], int i) {
  x[0]+=i;
}
```

Calling a subroutine

- Parameters go on the *stack*
- Use push to put each on the stack
  - push in reverse order: last param pushed first.
- Everything is passed by value!
  - you put a value on the stack
  - an address is a value!

calling `int add(int x,int y)`

- Assume **`x`** is in **`%ecx`**, **`y`** is in **`%edx`**

```
push %edx    # put y on stack
push %ecx    # put x on stack
call add     # call add()
# return value is always in %eax
```

`printf("num is %d\n",x);`

- Assume **`x`** is in **`%eax`**

```
st1:
    .string "num is %d\n"

    push %eax
    push $st1
    call printf
```

All together now...

```
st1: .string "%d + %d ="
st2: .string "%d\n"
    push %edx    # put y on stack
    push %ecx    # put x on stack
    push $st1    # put "%d + %d =" on stack
    call printf
    call add     # call add()
    push %eax    # put add(x,y) on stack
    push $st2    # put "%d\n" on stack
    call printf
```

C to Assembly

gcc can generate assembly for you:

`gcc -S foo.c`

produces the file **`foo.s`**

You can *assemble* foo.s:

`gcc -o foo foo.s`

Compiler generated assembly code

- There are no comments!
- Lots of other things besides code:
  - directives – lines that look like this:

`.globl foo` foo is a global symbol

`.section .rodata` define some read-only data

`.text` define some code

`.type foo, @function` foo came from a C function

`.size foo, .-foo` establishes the size of foo