Data movement instructions

READ	WRITE	ADDRESS
mov 0x123, %eax	mov %eax, 0x123	0x123
mov (%ebx), %eax	mov %eax, (%ebx)	%ebx
mov (%ebx, %edx, 4), %eax	mov %eax, (%ebx, %edx, 4)	%ebx + %edx * 4
mov (,%edx, 4), %eax	mov %eax, (, %edx, 4)	%edx * 4
mov 0x100(%ebx, %edx, 4), %eax	mov %eax, 0x100(%ebx, %edx, 4)	0x100 + %ebx + (%edx * 4)

jmp instruction

jmp rel32 // unconditional jump

EIP = EIP + rel32

rel32 is 32 bit signed integer

C equivalent:

goto label;

jmp instruction is an unconditional jump. This instruction sets the EIP to EIP + rel32.

x86 EFLAGS register

- Section-3.4.3 in Intel’s manual-1 (backpack)

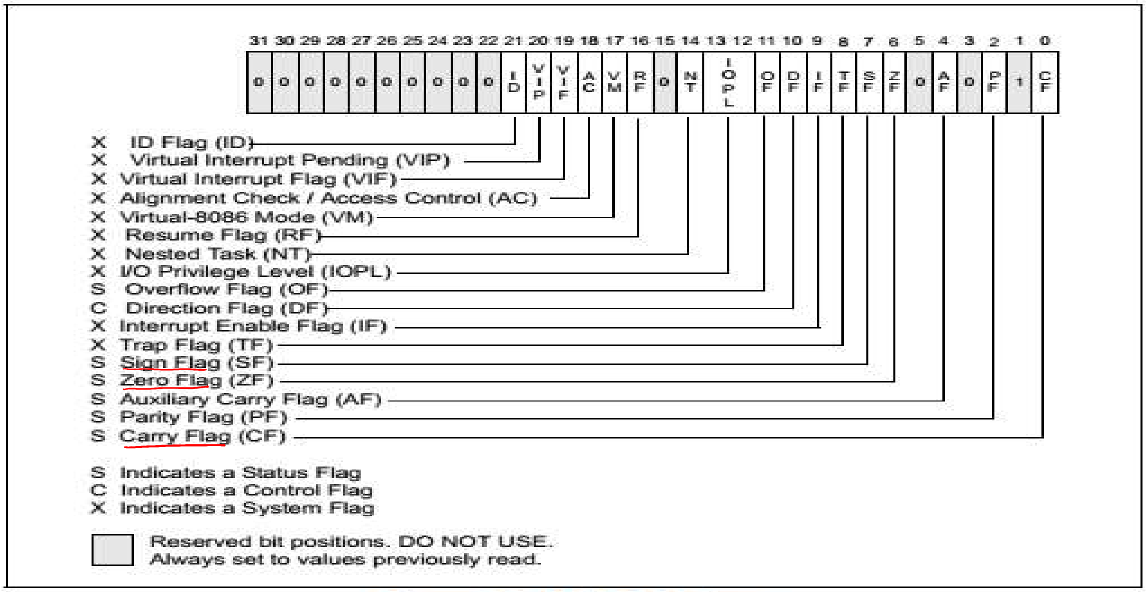


Figure 3-8. EFLAGS Register

EFLAGS

- Some instructions set certain bits in the EFLAGS register
  - Whether the last arithmetic instruction overflowed (OF)
  - Was positive/negative (SF)
  - Was zero / not zero (ZF)
  - Carry/borrow on add/subtract (CF)

Compare

cmp %eax, %edx

Perform %edx - %eax

Sets the flags in the EFLAGS register accordingly

If %edx and %eax are equal the ZF will be set

If %edx > %eax then CF will not be set

if %edx < %eax the CF (borrow) will be set

jcc – jump if condition is met

je rel32 // jump on equal

if (last operation was equal) //(ZF = 1)
    EIP = EIP + rel32

If the ZF is set, then src2 = src1. The x86 processor looks at the eflags register to find the value of zero flag(ZF). If ZF is set, then the x86 processor sets the EIP to the target address (EIP + rel32).

jcc – jump if condition is met

ja rel32 // jump on above

if (in last operation src2 > src1) // CF = 0 and ZF = 0
    EIP = EIP + rel32

If the ZF is set then src2 = src1. ja jumps to the target address (EIP + rel32) only if src2 > src1. The x86 processor looks at the eflags register to find the value of the carry flag(CF) and the zero flag(ZF). If both of them are not set, then it sets the EIP to the target address (EIP + rel32).

jcc – jump if condition is met

jb rel32 // jump on below

if (in last operation src2 < src1) // CF = 1
    EIP = EIP + rel32

jcc – jump if condition is met

jae rel32 // jump on above or equal

if (in last operation src2 >= src1) // CF = 0 or ZF = 1
    EIP = EIP + rel32

Example

```
if (%eax < %edx) {
}
```

Example

```
if (%eax < %edx) {
}

cmp %eax, %edx
jbe label              // jump if %edx <= %eax
// if body
label:
```

Example

```
if (%eax < %edx) {    //if body   }
else {   //else body   }

cmp %eax, %edx
jbe label1                  // jump if %edx <= %eax
// if body
jmp label2:
label1:
// else body
label2:
```

After if body, we need to insert an unconditional jump to skip the else body.

Addition

add %eax, %edx

%edx = %edx + %eax

Addition

add %eax, (%edx)   %edx

((int32)%edx) = *((int32*)%edx) + %eax;

Logical AND

and %eax, %edx

%edx = %edx & %eax;

Logical AND

and %eax, (%edx)

((int32)%edx) = %edx & *((int32*)%eax);

References (available on backpack)

- A Guide to Programming Intel IA32 PC Architecture
- Intel manual - 2

Next homework

- Read Intel manual-2 to find the meaning of some x86 instructions

- The Intel syntax is different from AT&T syntax
  - In Intel's syntax, the first operand is the destination

- Submit your handwritten homework in the submission box placed at the old academic building (2^{nd} floor)

Local variables

```
int foo(int x, int y)
{
   int v1, v2;
   int arr[64];
   ….
   bar(v1, v2);
}
```

Let us come back to our discussion on applications. How do local variables are allocated and deallocated in a C program?

Local variables

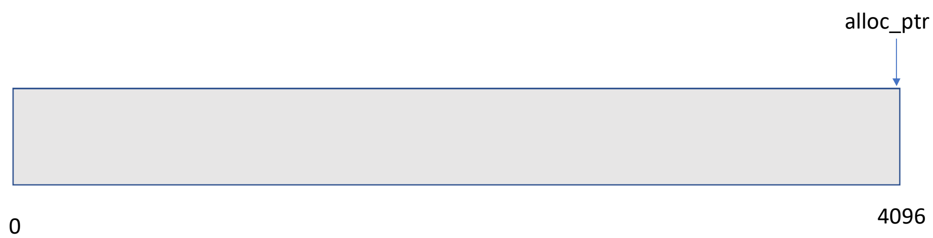
- Local variables are allocated and destroyed automatically by the compiler

- A straightforward strategy is to allocate all the local variables on function entry and deallocate them just before the function exit

Local variable

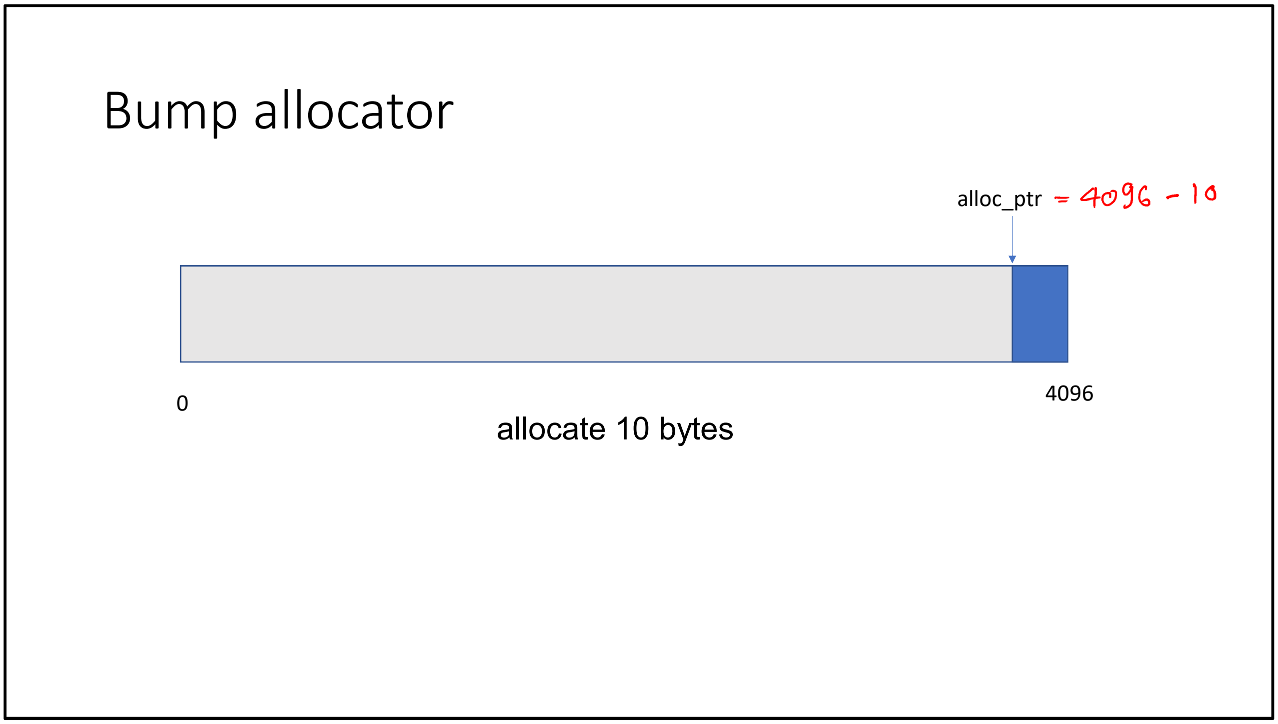
```
void bar() {
  int x1, x2, x3;
  …
}

void foo() {
  int v1, v2, v3;
  bar();
  …
}
```

Bump allocator



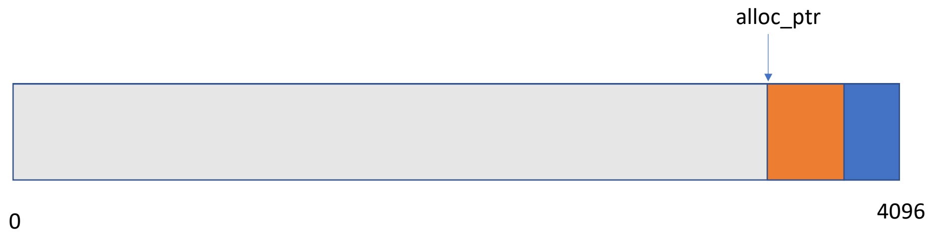
initialize
alloc_from_ram(4096)

Initially, bump allocator allocates a contiguous memory area from the OS. It maintains an alloc_ptr that is set to the end of the memory area.



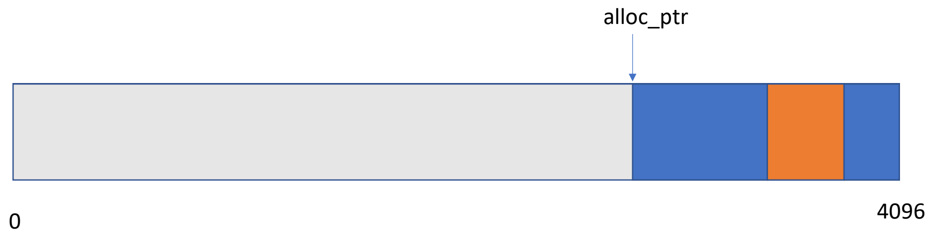
On every allocation, the alloc_ptr is advanced in the reverse direction by the size of the allocation. The new value of the alloc_ptr is returned to the user of the bump allocator.

Bump allocator



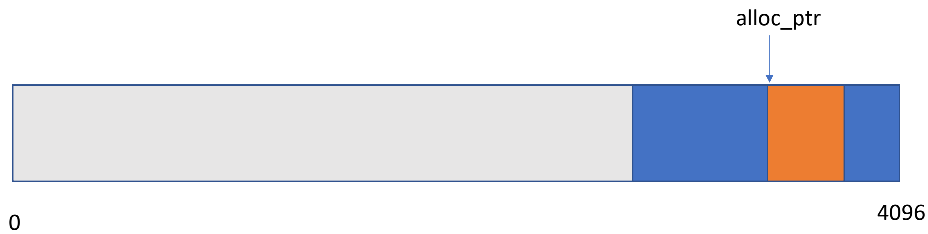
allocate 20 bytes

Bump allocator



allocate 50 bytes

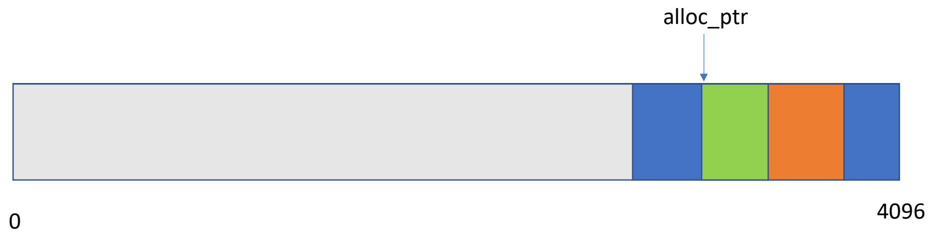
Bump allocator



deallocate 50 bytes

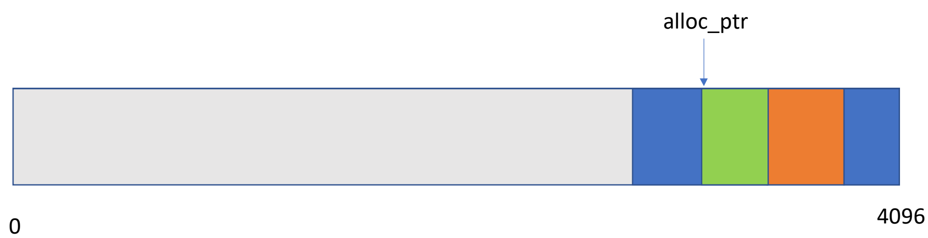
During deallocation, the bump allocator advances the alloc_ptr by the size of the deallocation.

Bump allocator



allocate 20 bytes

Bump allocator




```
void initialize() {
    char *ptr = alloc_from_ram(4096);
    alloc_ptr = ptr + 4096;
}
```

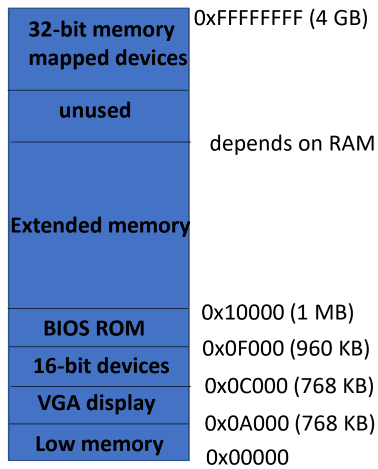
```
char *alloc(size_t size) {
    alloc_ptr -= size;
    return alloc_ptr;
}
```

```
void dealloc(size_t size) {
    alloc_ptr += size;
}
```

OS maintains a simple allocator

- alloc_from_ram(size_t size)
  - applications can use this interface to allocate space from RAM of size multiples of 4096
  - OS divides RAM into chunks of 4096 bytes called pages
  - OS maintains a list of free pages
  - Similar to alloc_pages in Linux kernel

Physical address space



OS divides the entire physical space into the contiguous chunks of 4096 bytes. These chunks are also called pages. We can allocate contiguous pages (i.e., size multiples of 4096) using alloc_from_ram API.

Local variable allocator

```
main () {      → initialize
  int a, b, c; → alloc (12)
  foo();
}              → dealloc (12)
foo() {        → alloc (8)
 int x, y;
 bar();        → dealloc ( 8)
}
bar () {       → alloc(20)
 int a, b, c, x, y;
}              → dealloc (20)
```

Bump allocator

```
void initialize();
char *alloc(size_t size);
void dealloc(size_t size);
```

On function entry, compiler insert calls to allocator to allocate the local variables. Just before returning from a function, the compiler deallocates space allocated for local variables. A bump allocator can be used for allocating and deallocating local variables. Bump allocator requires that the memory deallocation happens in the reverse order of their allocation that is true in this case.

Local variable allocator

```
main () {           // initialize
  int a, b, c;      // alloc(12)
  foo();
}                   // dealloc(12)
foo() {
 int x, y;            // alloc(8)
 bar();
}                     // dealloc(8)
bar () {
 int a, b, c, x, y;       // alloc(20)
}                         // dealloc(20)
```

Bump allocator

```
void initialize();
char *alloc(size_t size);
void dealloc(size_t size);
```

Local variable allocator

```
main () {          // initialize
  int a, b, c;     // alloc(12)
  foo();
}                  // dealloc(12)
foo() {
 int x, y;            // alloc(8)
 bar();
}                     // dealloc(8)
bar () {
 int a, b, c, x, y;        // alloc(20)
}                          // dealloc(20)
```

During execution

initialize
alloc(12)
alloc(8)
alloc(20)
dealloc(20)
dealloc(8)
dealloc(12)

The alloc and dealloc routines are called in this order during the execution of this program.

Parameter passing

```
foo() {
  int x, y;
  bar(x, y);
}
bar(int a1, int a2) {
  return a1 + a2;
}
```

Bump allocator

```
void initialize();
char *alloc(size_t size);
void dealloc(size_t size);
```

Parameters are similar to local variables, except their values are initialized by the caller. For example, before calling bar foo routine sets the values of a1 and a2 to x and y, respectively. It is the caller's (foo in this case) responsibility to allocate space for the parameters and initialize them.

Parameter passing

```
foo() {
  int x, y;            // alloc(8)
  bar(x);  → alloc(4)
           → dealloc(4)
}                      // dealloc(8)
bar(int x) {
  int a, b;            // alloc(8)
  baz(a);  → alloc(4)
           → dealloc(4)
}                      // dealloc(8)
baz(int y) {
  int x, y, a, b;  // alloc(16)
}                  // dealloc(16)
```

During execution

alloc(8)
alloc(4)
~~dealloc(4)~~
~~dealloc(8)~~
alloc(8)
alloc(4)
alloc(16)
dealloc(16)
dealloc(4)
dealloc(8)
dealloc(4)
dealloc(8)

Before every function call, the compiler can call bump allocator APIs to allocate space for parameters and deallocate them after the target function returns.

Bump allocator

- To implement the bump allocator, we need to store the alloc_ptr

- Compilers reserve %esp register to store the alloc_ptr
  - %esp is called the stack pointer
  - RAM space allocated by the bump allocator is called the stack

Calling convention

- Compiler compiles one routine at a time
- Often a routine doesn't know who are the potential callers
  - e.g., printf
- The caller or callee must follow some convention for the parameter passing
  - gcc 32-bit compiler passes parameters via stack

Return address

```
foo() {
  baz();
}
bar(){
 baz();
}
baz(){
}
```

How does baz know where to return (foo or bar). Along with the parameters, the return address is also passed by the caller.

Calling convention

- The contract between caller and callee in x86:
  - On entry to a function call
    - %eip points to the first instruction of a function
    - %esp points to the return address
    - %esp+4 points to the first argument
    - %esp+8 points to the second argument
    - and so on

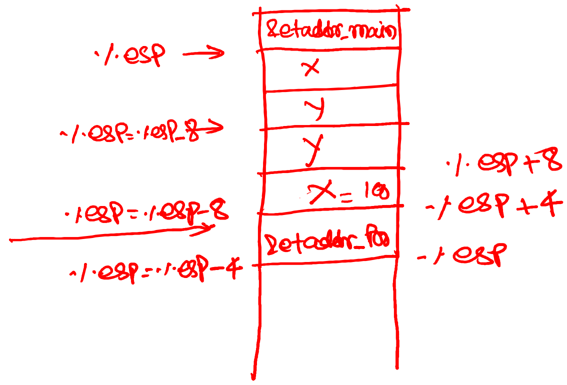
Calling convention

- The contract between caller and callee in x86:
  - After the callee returns
    - %eip points to the return address
    - %esp points to the argument pushed by the caller
    - called function may have trashed arguments
    - %eax (and %edx if the return type is 64 bit) contains the return value

Example

```
void foo() {
  int x = 10;
  int y = 20;
  bar(x, y);
}
int bar(int x, int y) {
   return x + y;
}
```

x = 100



On entry to foo function stack pointer (%esp) points to the return address in the caller (say main). On entry to bar function, %esp points to return address in foo, (%esp +4) points to x (first argument), (%esp + 8) points to y (second argument). After returning from bar %esp points to the argument pushed by foo, and the %eax register contains the return value of the bar routine.

Push

push %eax

sub $4, %esp
mov %eax, (%esp)

- allocates 4-bytes on the stack and initializes them with the value of the %eax register

Pop

pop %eax

mov (%esp), %eax
add $4, %esp

- deallocates 4-bytes from the stack and returns the value of the deallocated memory in the %eax register

call instruction

call rel32

retaddr = EIP + length(EIP)
EIP = EIP + rel32   //rel32 is 32 bit signed integer
push retaddr

C equivalent:
bar();

call instruction set the EIP to EIP + rel32. In addition to this, call instruction also pushes the return address (retaddr) on the stack. x86 instructions are complex instructions and can do multiple tasks in one instruction.

ret instruction

ret

pop %eip

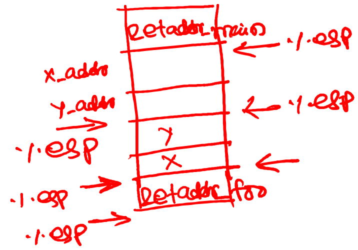
C equivalent:

return

ret instruction pops the return address from the top of the stack and set the %eip to popped value. There is no such instruction pop %eip. The semantics of ret instruction is similar to the possible semantics of “pop %eip” (if it would have existed).

Example

```
void foo() {
  int x = 10;
  int y = 20;
  bar(x, y);
}
int bar(int x, int y) {
   return x + y;
}
```



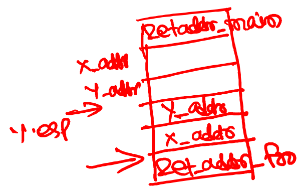

```
foo:
  sub $8, %esp
  mov $10, 4(%esp)
  mov $20, (%esp)
  sub $8, %esp
  mov 12(%esp), %eax
  mov %eax, (%esp)
  mov 8(%esp), %eax
  mov %eax, 4(%esp)
  call bar
  add $8, %esp
  add $8, %esp
  ret
```

```
bar:
  mov 4(%esp), %eax
  mov 8(%esp), %ecx
  add %ecx, %eax
  ret
```

This slide shows one possible code generation that follows the gcc calling convention.

Example

```
void foo() {
  int x = 10;
  int y = 20;
  bar(&x, &y);
}
int bar(int *a, int *b) {
   return (*a) + (*b);
}
```




```
bar:
  mov 4(%esp), %eax
  mov (%eax), %ecx
  mov 8(%esp), %eax
  mov (%eax), %edx
  add %ecx, %edx
  mov %edx, %eax
  ret
```

```
foo:
  sub $8, %esp
  mov $10, 4(%esp)
  mov $20, (%esp)
  sub $8, %esp
  mov %esp, %eax
  add $12, %eax
  mov %eax, (%esp)
  sub $4, %eax
  mov %esp, %eax
  add $8, %eax
  mov %eax, 4(%esp)
  call bar
  add $8, %esp
  add $8, %esp
  ret
```

This is an example of pass by reference. Notice that in this case, we are passing the address of x and address of y to the bar routine. Because x and y are allocated on the stack, their addresses are stack addresses and are at a constant offset from %esp (known to the compiler).