Lecture 5: Assembly Boot

Professor G. Sandoval

Some slides adapted by G. Sandoval for CS3224, from Tanenbaum & Bo, Modern Operating Systems:4th ed., (c) 2013 Prentice-Hall, Inc.
Also some Slides by Brendan Dolan-Gavitt and Bryant and O’Hallaron, Computer Systems: A programmer’s Perspective, Third Edition

Today: Machine Programming Basics

- **Assembly**
  - More examples
  - Function calls
- **GDB**
- **Booting the OS**
  - BIOS

Fill in the blanks

Address	Value	Register	Value
0x100	0xFF	%rax	0x100
0x104	0xAB	%rcx	0x1
0x108	0x13	%rdx	0x3
0x10C	0x11		

Operand	Value	Comment
%rax		
0x104		
$0x108		
(%rax)		
4(%rax)		

Fill in the blanks

Address	Value	Register	Value
0x100	0xFF	%rax	0x100
0x104	0xAB	%rcx	0x1
0x108	0x13	%rdx	0x3
0x10C	0x11		

Operand	Value	Comment
%rax	0x100	Register
0x104	0xAB	Absolute Address
$0x108	0x108	Immediate
(%rax)	0xFF	Address 0x100
4(%rax)	0xAB	Address 0x104

x64 Assembly: Jumps

Instruction	Effect
`jmp`	Always jump
`je/jz`	Jump if eq / zero
`jne/jnz`	Jump if !eq / !zero
`jg`	Jump if greater
`jge`	Jump if greater / eq
`jl`	Jump if less
`jle`	Jump if less / eq

x64 Assembly: A Quick Drill

```
cmp $0x15213, %r12
jge deadbeef
```

If _____, jump to addr `0xdeadbeef`

```
cmp %rax, %rdi
jne 15213b
```

If _____, jump to addr `0x15213b`

```
test %r8, %r8
jnz (%rsi)
```

If _____, jump to _____.

x64 Assembly: A Quick Drill

```
cmp $0x15213, %r12
jge deadbeef
```

If `%r12 >= 0x15213`, jump to `0xdeadbeef`

```
cmp %rax, %rdi
jne 15213b
```

```
test %r8, %r8
jnz (%rsi)
```

x64 Assembly: A Quick Drill

```
cmp $0x15213, %r12
jge deadbeef

cmp %rax, %rdi
jne 15213b

test %r8, %r8
jnz (%rsi)
```

If the value of `%rdi` is not equal to value of `%rax`, jump to `0x15213b`.

x64 Assembly: A Quick Drill

```
cmp $0x15213, %r12
jge deadbeef

cmp %rax, %rdi
jne 15213b

test %r8, %r8
jnz (%rsi)
```

If `%r8 & %r8` is not zero, jump to the address stored in `%rsi`.

Function Calls

- Supports creation of functions
- call <f>: push the address of next instruction on the stack and jump to f

- Example:

```
1000 call <f>          // pushes 5 and jumps to f
1005 movl %eax, %ebx
```

- ret: pop return address off the stack and jump to it

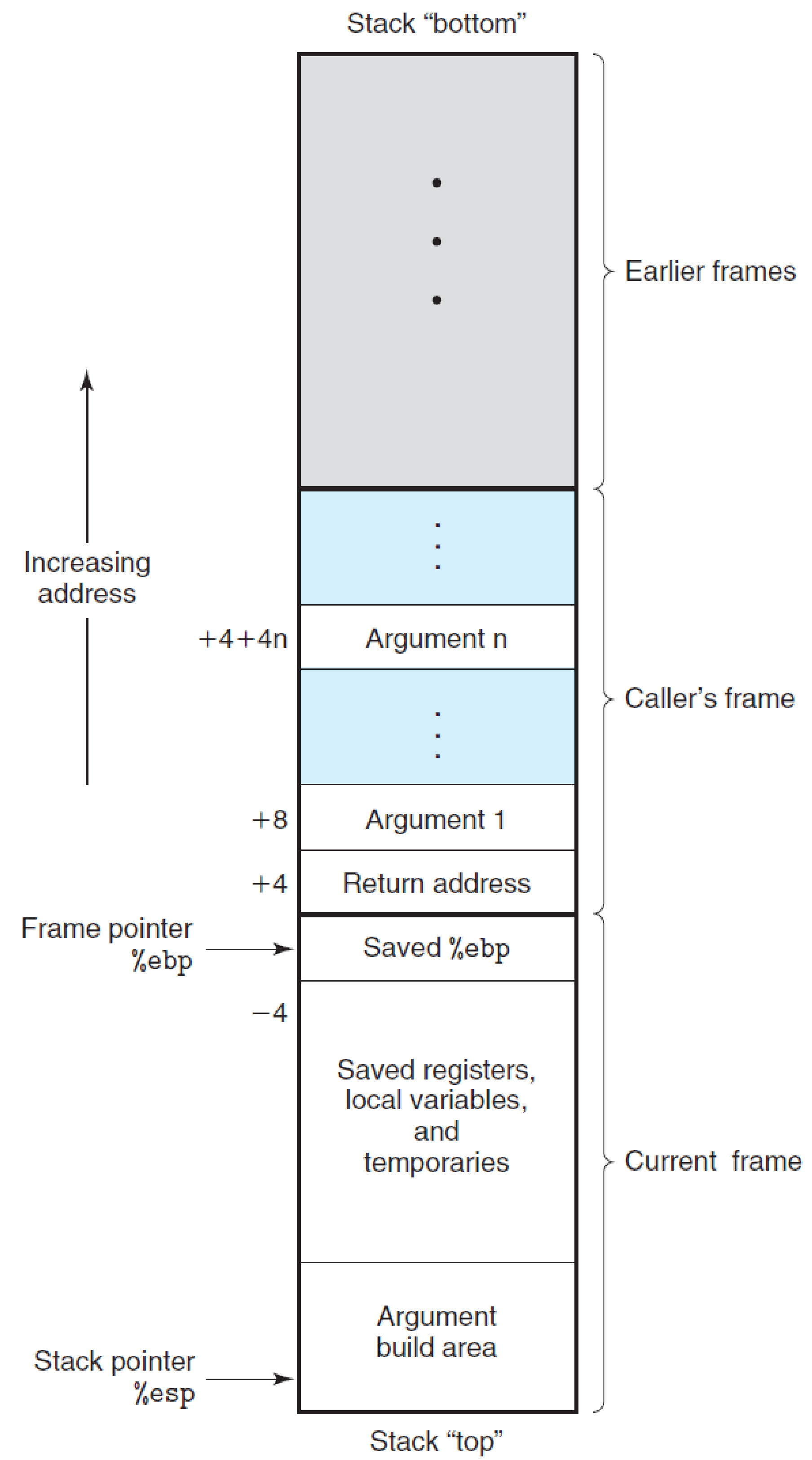
Stack Frames

- **IA32 programs use the stack to support function calls.**
- **The machine uses the stack to:**
  - Pass arguments
  - store return information
  - Save registers
  - Local Storage
- **Therefore to use arguments and return information we need to deal with the stack.**
- **The portion of memory used in the stack for a single procedure call is called a stack frame.**

Stack Frames

- The topmost stack frame is delimited by two register pointers:
  - **%ebp** is the frame pointer (base)
  - **%esp** is the stack pointer

Stack Frame



Stack "bottom"
Earlier frames
Increasing address
+4+4n
Argument n
Caller's frame
+8
Argument 1
+4
Return address
Frame pointer %ebp
Saved %ebp
−4
Saved registers, local variables, and temporaries
Current frame
Argument build area
Stack pointer %esp
Stack "top"

Function Prologue & Epilogue

- The need to set up a stack frame gives rise to a common function prologue:

```
push %ebp           // save old value of ebp
mov %esp, %ebp      // set base p
sub $X, %esp        // reserve space for locals
```

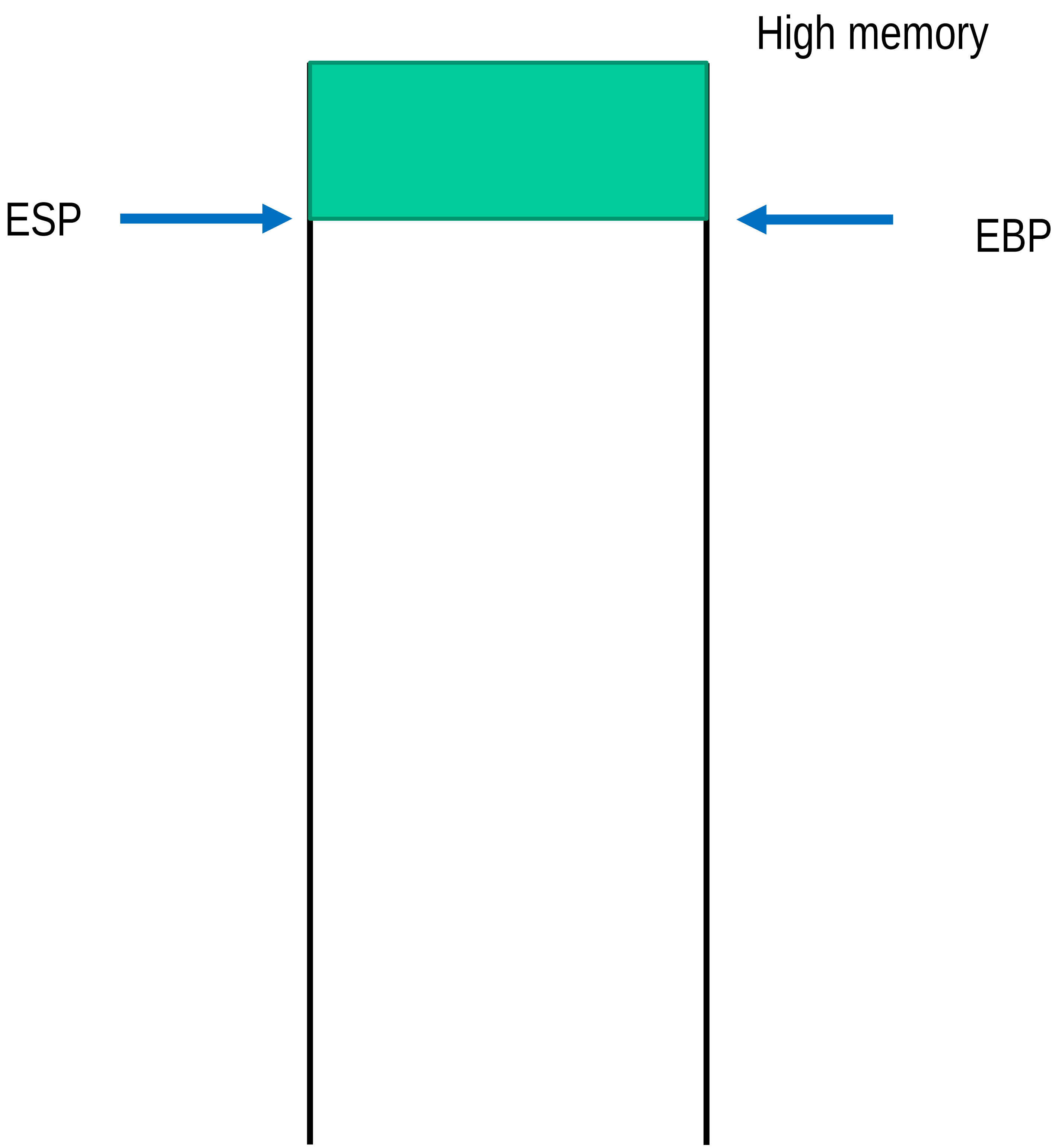
- At the end, we can clean up with just the epilogue

```
add %ebp, %esp      // restore stack pointer
pop %ebp            // restore base pointer
ret                 // return pops return address
```

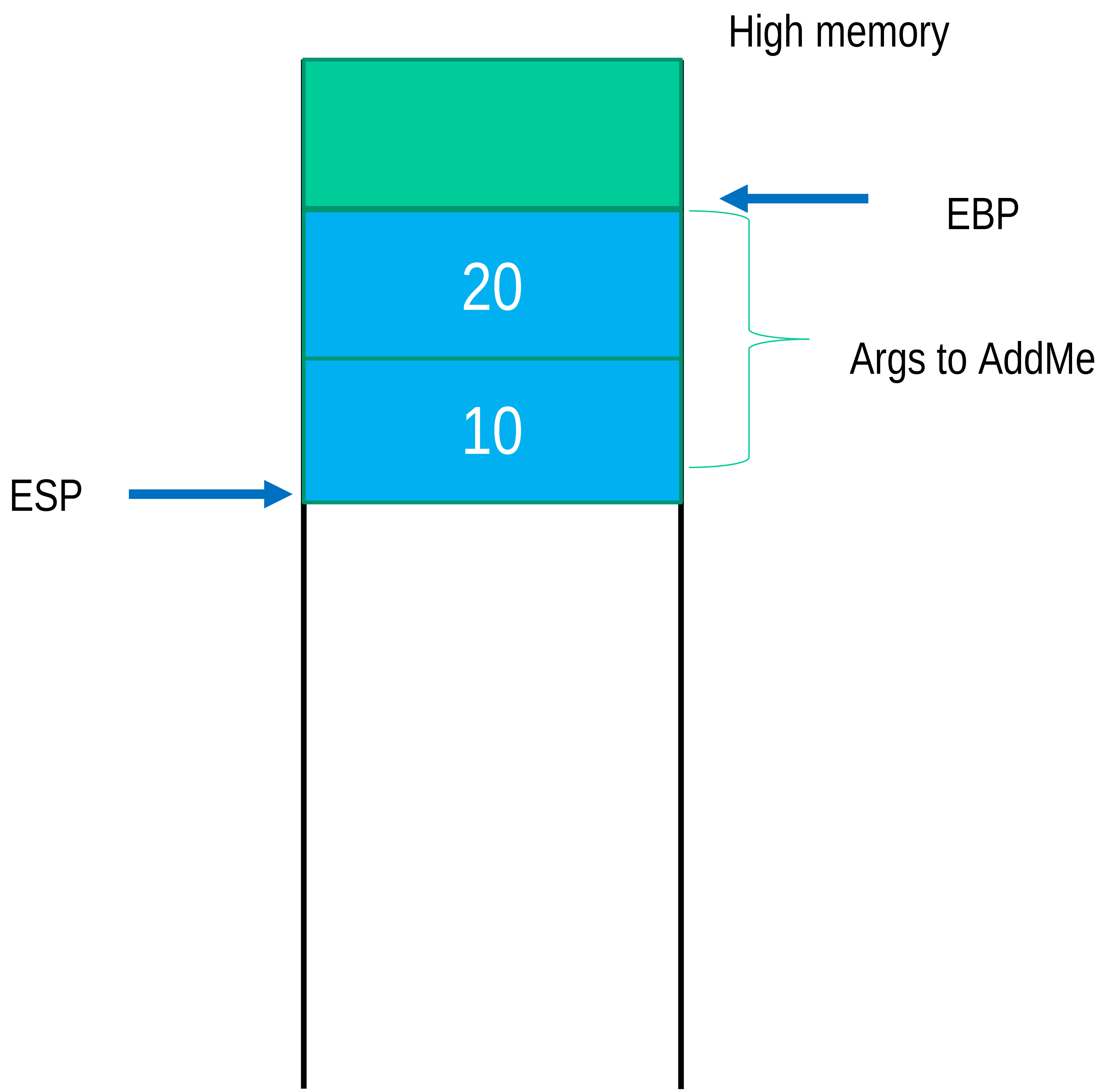
Walking the Call Stack

```
int AddMe(int a, int b)
{
    int c;
    c = a + b;
    return c;
}

main()
{
    AddMe(10,20);
    print…
}
```



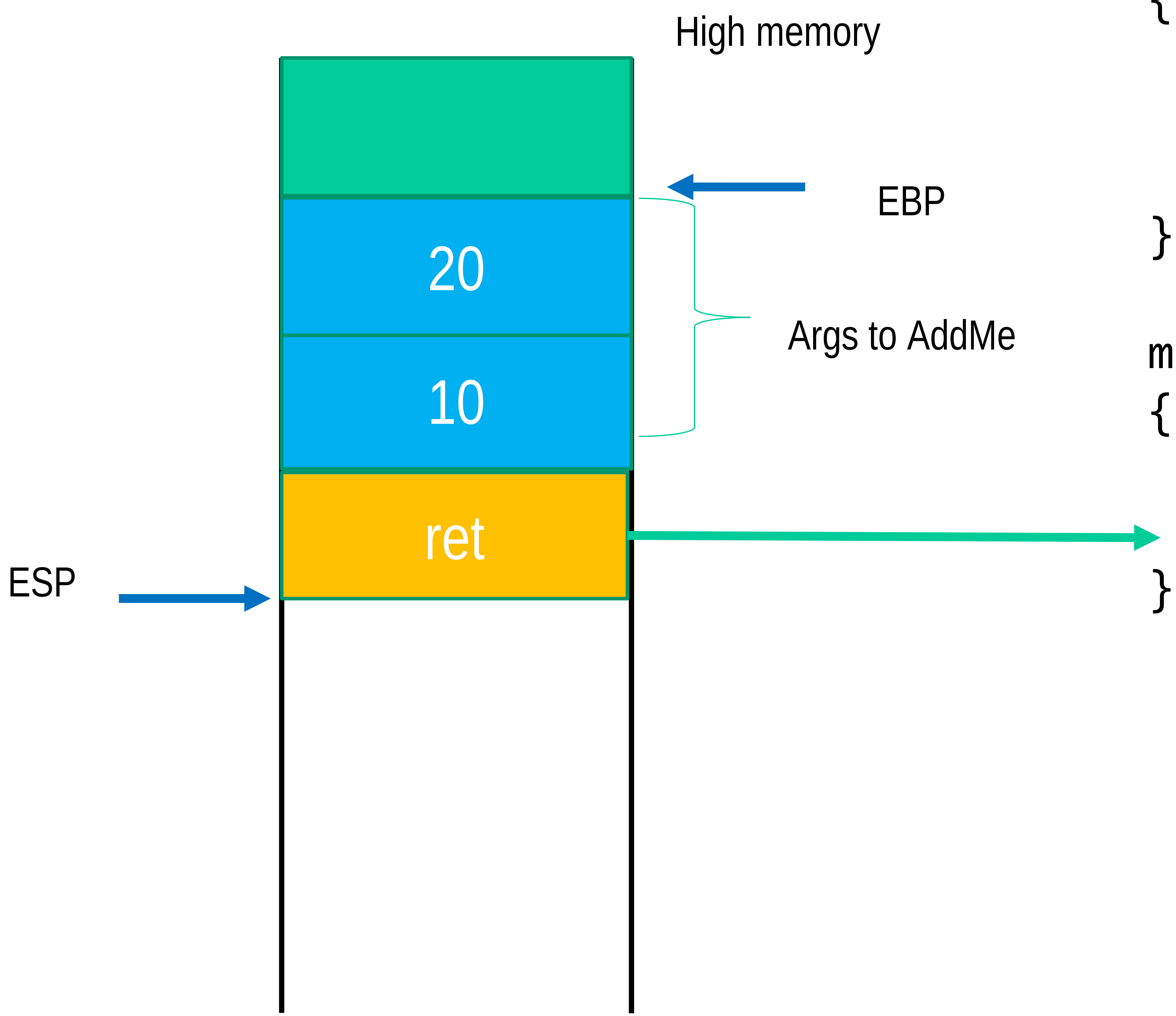
Walking the Call Stack




```
int AddMe(int a, int b)
{
    int c;
    c = a + b;
    return c;
}

main()
{
    AddMe(10,20);
    print…
}
```

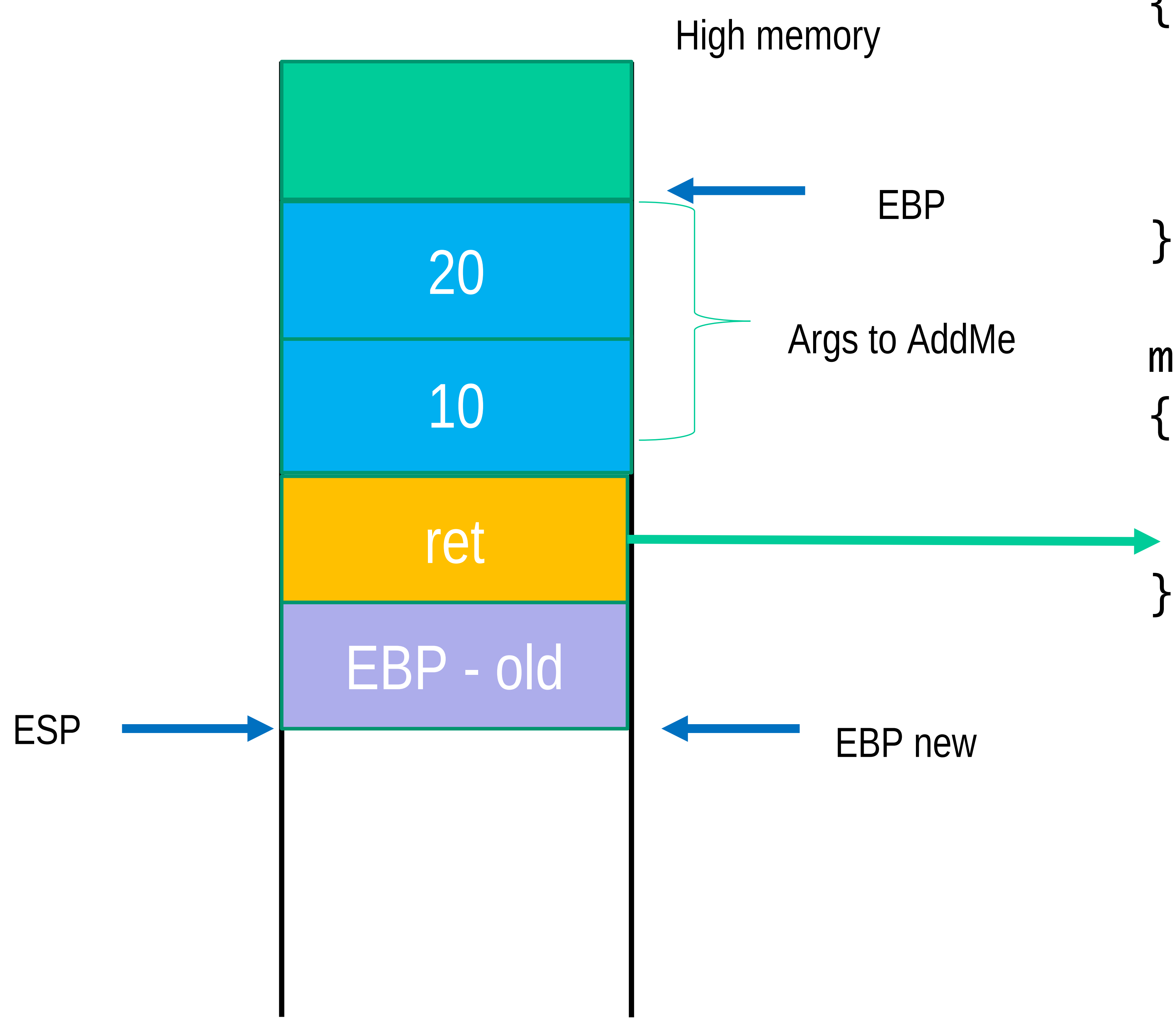
Walking the Call Stack




```
int AddMe(int a, int b)
{
    int c;
    c = a + b;
    return c;
}

main()
{
    AddMe(10,20);
    print…
}
```

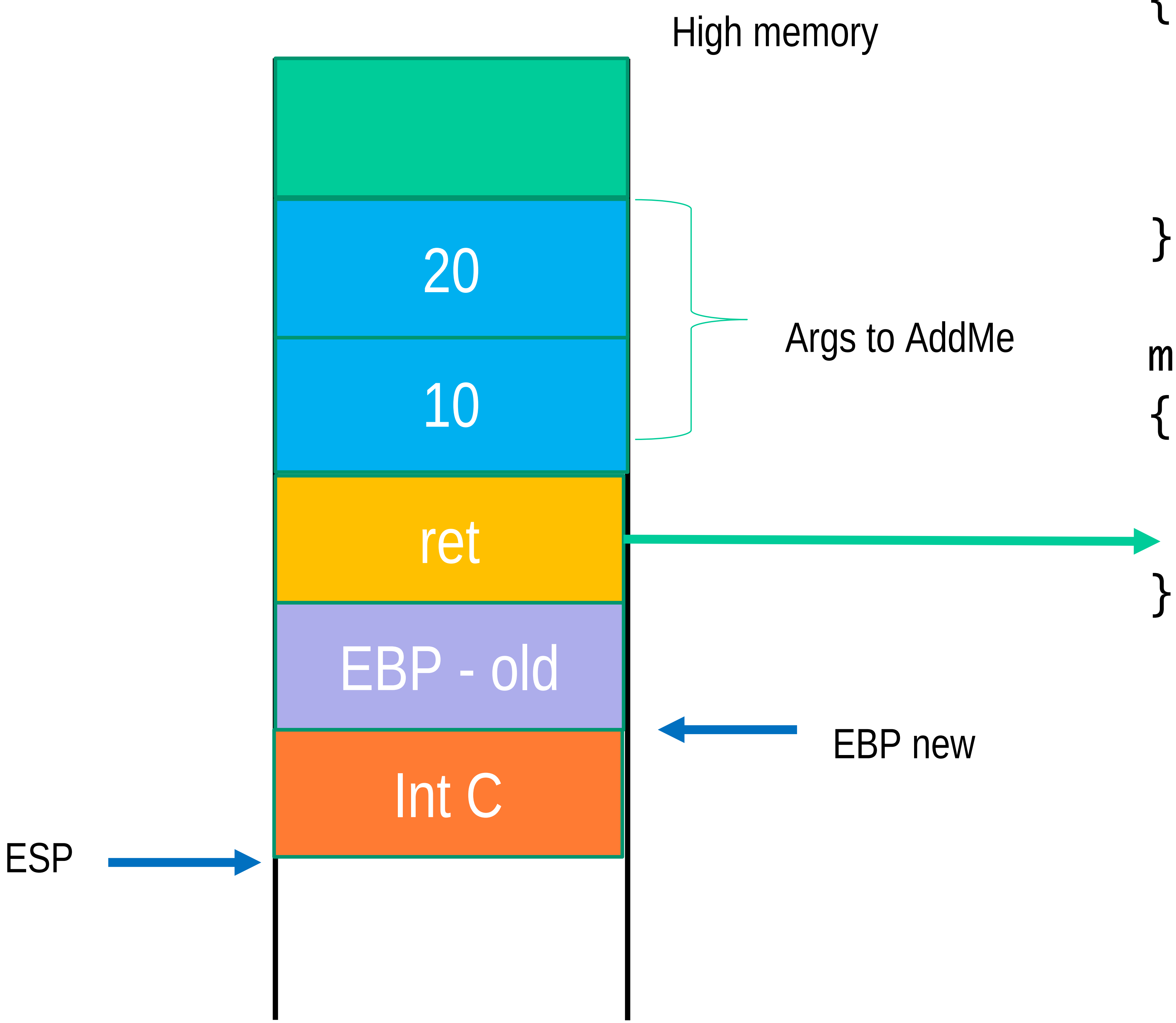
Walking the Call Stack




```
int AddMe(int a, int b)
{
    int c;
    c = a + b;
    return c;
}

main()
{
    AddMe(10,20);
    print…
}
```

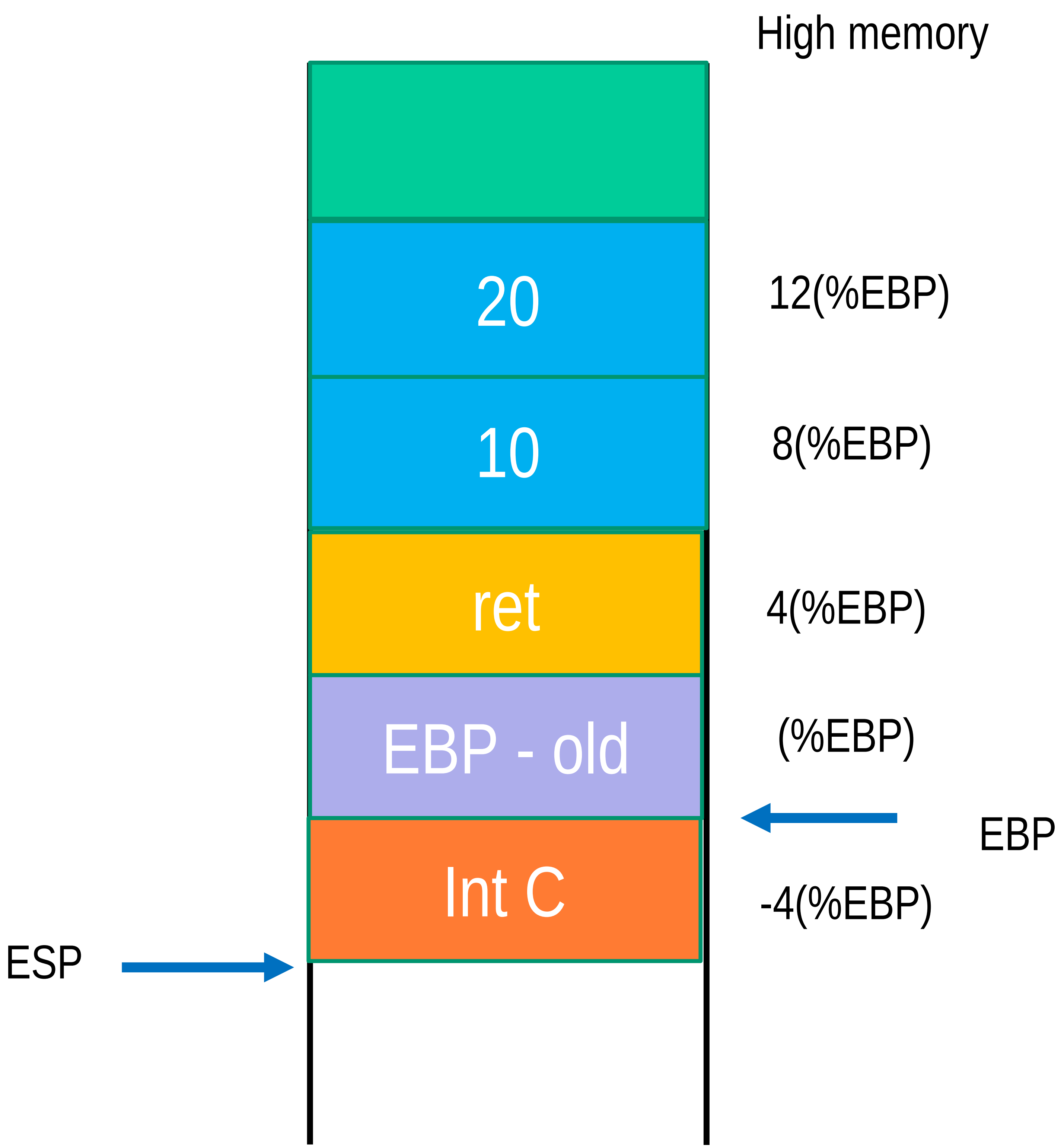
Walking the Call Stack




```
int AddMe(int a, int b)
{
    int c;
    c = a + b;
    return c;
}

main()
{
    AddMe(10,20);
    print…
}
```

To reference data on the stack



Calling Conventions

- **A *calling convention* is a system-defined convention that tells you how to call a function**
- **In general a calling convention will:**
  - Say how to pass arguments to the function
  - Say who is responsible for cleaning up arguments
  - Specify where the return value will be placed
  - Say which registers will be preserved by the callee and which might be trashed

gcc Calling Convention (32-bit)

- **Return value: %eax**
- **Arguments are passed on the stack:**
  - f(1, 2, 3)
  - push 3
push 2
push 1
call <f>

3

2

1

retaddr

gcc Calling Convention (32-bit)

- **Caller is responsible for cleaning arguments off the stack**
  - Note that this allows functions with variable numbers of arguments (varargs)
- **%eax %ecx, %edx may be trashed by callee (must be saved by caller)**
- **%ebp, %ebx, %esi, %edi must be preserved by callee**

Aside: Buffer Overflow

- **Consider a function like:**
```
int f(int arg1, int arg2) {
    char x[4];
    ...
}
```
- **Stack frame for function shown at right**
- **Suppose inside f, we write past the end of array x?**
- **What gets overwritten? What happens when we execute ret?**

Arg2
Arg2
Return address
Saved base pointer
X[3]
X[2]
X[1]
X[0]

Example from XV6: cat.asm

Example Function Call

```c
#include <stdio.h>

int g(int a) {
    return a*2;
}

int f(int a, int b) {
    return g(a) + g(b);
}

int main(void) {
    printf("%d\n", f(1, 2));
    return 0;
}
```

main

```c
int main(void) {
    printf("%d\n", f(1,2));
    return 0;
}
```

```
08048428 <main>:
 8048428:       8d 4c 24 04             lea    0x4(%esp),%ecx
 804842c:       83 e4 f0                and    $0xfffffff0,%esp
 804842f:       ff 71 fc                pushl  -0x4(%ecx)
 8048432:       55                      push   %ebp
 8048433:       89 e5                   mov    %esp,%ebp
 8048435:       51                      push   %ecx
 8048436:       83 ec 04                sub    $0x4,%esp
 8048439:       6a 02                   push   $0x2
 804843b:       6a 01                   push   $0x1
 804843d:       e8 c3 ff ff ff          call   8048405 <f>
 8048442:       83 c4 08                add    $0x8,%esp
 8048445:       83 ec 08                sub    $0x8,%esp
 8048448:       50                      push   %eax
 8048449:       68 00 85 04 08          push   $0x8048500
 804844e:       e8 7d fe ff ff          call   80482d0 <printf@plt>
 8048453:       83 c4 10                add    $0x10,%esp
 8048456:       b8 00 00 00 00          mov    $0x0,%eax
 804845b:       8b 4d fc                mov    -0x4(%ebp),%ecx
 804845e:       c9                      leave
 804845f:       8d 61 fc                lea    -0x4(%ecx),%esp
 8048462:       c3                      ret
```

f

```c
#include <stdio.h>

int g(int a) {
    return a*2;
}

int f(int a, int b) {
    return g(a) + g(b);
}

int main(void) {
    printf("%d\n", f(1,2));
    return 0;
}
```

```
08048405 <f>:
 8048405:       55                      push   %ebp
 8048406:       89 e5                   mov    %esp,%ebp
 8048408:       53                      push   %ebx
 8048409:       ff 75 08                pushl  0x8(%ebp)
 804840c:       e8 ea ff ff ff          call   80483fb <g>
 8048411:       83 c4 04                add    $0x4,%esp
 8048414:       89 c3                   mov    %eax,%ebx
 8048416:       ff 75 0c                pushl  0xc(%ebp)
 8048419:       e8 dd ff ff ff          call   80483fb <g>
 804841e:       83 c4 04                add    $0x4,%esp
 8048421:       01 d8                   add    %ebx,%eax
 8048423:       8b 5d fc                mov    -0x4(%ebp),%ebx
 8048426:       c9                      leave
 8048427:       c3                      ret
```

g

```c
#include <stdio.h>

int g(int a) {
    return a*2;
}

int f(int a, int b) {
    return g(a) + g(b);
}

int main(void) {
    printf("%d\n", f(1,2));
    return 0;
}
```

```
080483fb <g>:
 80483fb:       55                      push   %ebp
 80483fc:       89 e5                   mov    %esp,%ebp
 80483fe:       8b 45 08                mov    0x8(%ebp),%eax
 8048401:       01 c0                   add    %eax,%eax
 8048403:       5d                      pop    %ebp
 8048404:       c3                      ret
```

Walkthrough

- A great way to understand what's going on is to step through a program in gdb
- You'll also use gdb to debug the OS, so it's good to become familiar with it
- On OS X the default debugger is lldb, which has different syntax for commands (it is possible to install gdb instead, though)

Today: Machine Programming Basics

- Assembly
  - More examples
  - Function calls
- **GDB**
- Booting the OS
  - BIOS

Using gdb

- **`break <location>`**
  - Stop execution at function name or address
  - Reset breakpoints when restarting `gdb`
- **`run <args>`**
  - Run program with args `<args>`
  - Convenient for specifying text file with answers
- **`disas <fun>`, but not `dis`**
- **`stepi / nexti`**
  - Steps / does not step through function calls

Using gdb

- `info registers`
  - Print hex values in every register
- `print` (`/x` or `/d`) `$eax` - Yes, use `$`
  - Print hex or decimal contents of `%eax`
- `x $register, x 0xaddress`
  - Prints what's in the register / at the given address
  - By default, prints one word (4 bytes)
  - Specify format: /s, /[num][size][format]
    - `x/8a 0x15213`
    - `x/4wd 0xdeadbeef`

GDB

- **Use gdbtui**
  - Nice display for viewing source/executing commands

DEMO

Today: Machine Programming Basics

- **Assembly**
  - More examples
  - Function calls
- **GDB**
- **Booting the OS**
  - **BIOS**
  - Assembly Bootstrap
  - C Bootstrap

BIOS

- **When an x86 PC boots, it starts executing the BIOS.**
- **BIOS**
  - stored on the motherboard
  - Prepares the hardware
  - Transfers control to the OS (Bootsector)
- **BootSector:**
  - First 512 bytes of the boot disk.
  - Contains the boot loader

PC Physical Address Space

```
+------------------+  <- 0xFFFFFFFF (4GB)
|      32-bit      |
|  memory mapped   |
|     devices      |
|                  |
/\/\/\/\/\/\/\/\/\/\

/\/\/\/\/\/\/\/\/\/\
|                  |
|      Unused      |
|                  |
+------------------+  <- depends on amount of RAM
|                  |
|                  |
| Extended Memory  |
|                  |
|                  |
+------------------+  <- 0x00100000 (1MB)
|     BIOS ROM     |
+------------------+  <- 0x000F0000 (960KB)
|  16-bit devices, |
|  expansion ROMs  |
+------------------+  <- 0x000C0000 (768KB)
|   VGA Display    |
+------------------+  <- 0x000A0000 (640KB)
|                  |
|    Low Memory    |
|                  |
+------------------+  <- 0x00000000
```

xv6 Bootloader

- Note: x86 processors start out in 16-bit *real mode* (pretends to be an 8088 processor from 1979)
- BIOS loads the bootloader (512 bytes) at 0x7c00
- Loader's job:
  - Put the processor in a more modern operating mode(386 from 8088),
  - Load the xv6 kernel from disk into memory,
  - Transfer control to the kernel.

- The xv6 boot loader comprises two parts:
  - **Assembly Bootstrap(bootasm.S:** written in a combination of 16-bit and 32-bit x86 assembly
  - **C Bootstrap(bootmain.c);** written in C

Today: Machine Programming Basics

- Assembly
  - More examples
  - Function calls
- GDB
- Booting the OS
  - BIOS
  - **Assembly Bootstrap**
  - C Bootstrap

Preliminaries

- **During the BIOS's hardware initialization, it enabled hardware interrupts**
  - We don't know what its interrupt handlers look like, and we don't want to deal with interrupts yet
  - Use **cli** instruction to disable interrupts.

- **We also don't know the state of segment registers, so we clear them**

Preliminaries

```
.code16                     # Assemble for 16-bit mode
.globl start
start:
  cli                       # BIOS enabled interrupts; disable

  # Zero data segment registers DS, ES, and SS.
  xorw    %ax,%ax           # Set %ax to zero
  movw    %ax,%ds           # -> Data Segment
  movw    %ax,%es           # -> Extra Segment
  movw    %ax,%ss           # -> Stack Segment
```

The Ugly A20 Hack

- At boot, one of the memory address lines (A20) is forced to 0
- This is for backward compatibility with the original 8088/8086, where addresses were not supported
- So an unused pin on the keyboard controller was attached to the A20 line

I/O Port for Keyboard Status

```
  # Physical address line A20 is tied to zero so that the first PCs
  # with 2 MB would run software that assumed 1 MB.  Undo that.
seta20.1:
  inb     $0x64,%al               # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.1

  movb    $0xd1,%al               # 0xd1 -> port 0x64
  outb    %al,$0x64

seta20.2:
  inb     $0x64,%al               # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.2

  movb    $0xdf,%al               # 0xdf -> port 0x60
  outb    %al,$0x60
```

Write next byte to Controller Output Port

```
  # Physical address line A20 is tied to zero so that the first PCs
  # with 2 MB would run software that assumed 1 MB.  Undo that.
seta20.1:
  inb     $0x64,%al               # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.1

  movb    $0xd1,%al               # 0xd1 -> port 0x64
  outb    %al,$0x64

seta20.2:
  inb     $0x64,%al               # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.2

  movb    $0xdf,%al               # 0xdf -> port 0x60
  outb    %al,$0x60
```

PS/2 Data Port

```
  # Physical address line A20 is tied to zero so that the first PCs
  # with 2 MB would run software that assumed 1 MB.  Undo that.
seta20.1:
  inb     $0x64,%al               # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.1

  movb    $0xd1,%al               # 0xd1 -> port 0x64
  outb    %al,$0x64

seta20.2:
  inb     $0x64,%al               # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.2

  movb    $0xdf,%al               # 0xdf -> port 0x60
  outb    %al,$0x60
```

A20 Enable

0xdf = 11011111

```
  # Physical address line A20 is tied to zero so that the first PCs
  # with 2 MB would run software that assumed 1 MB.  Undo that.
seta20.1:
  inb     $0x64,%al                     # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.1

  movb    $0xd1,%al                     # 0xd1 -> port 0x64
  outb    %al,$0x64

seta20.2:
  inb     $0x64,%al                     # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.2

  movb    $0xdf,%al                     # 0xdf -> port 0x60
  outb    %al,$0x60
```

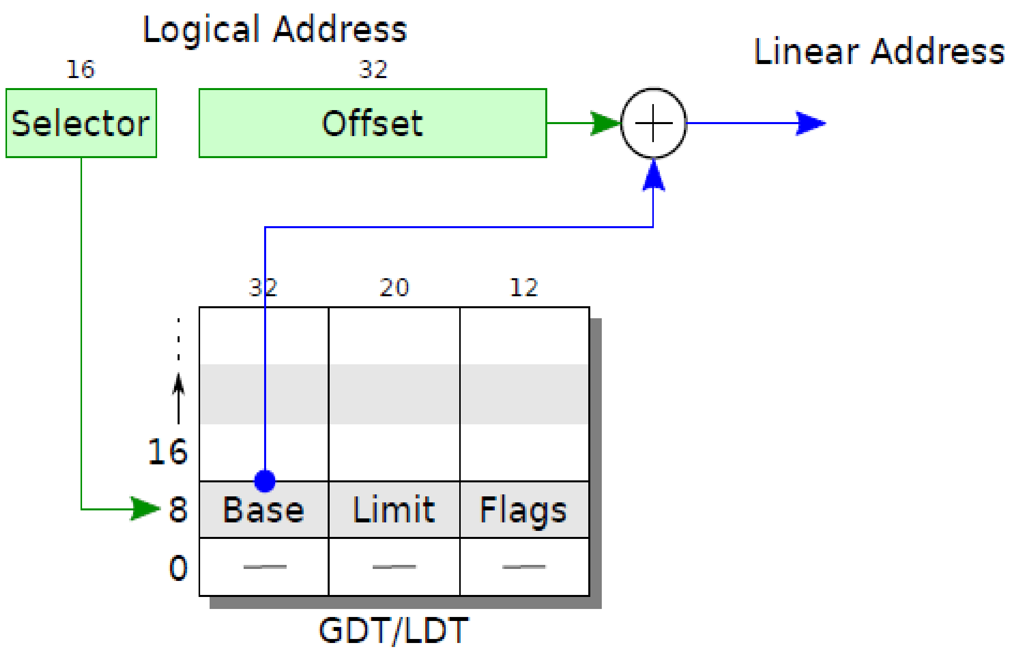
Transitioning into Protected Mode

- To run 32-bit code, we need to transition into ***protected*** *mode*
- A key feature here is that the meaning of the segment registers (%cs, %ds, etc.) changes
- Instead of just holding extra bits for memory addresses, they *select* a particular *segment descriptor*
  - *Segment descriptor* gives the base address, size, and permissions of that memory segment

Segments in xv6

- As with most modern OSes, xv6 makes very little use of segmentation
- The segments it defines just give a 1-1 mapping between logical and physical addresses
- Later, it will set up the *paging hardware* and use *virtual addresses*

Segments



Logical Address
16
32
Selector
Offset
Linear Address
32
20
12
16
8
0
Base
Limit
Flags
GDT/LDT

```
# Switch from real to protected mode.
# Use a bootstrap GDT that makes
# virtual addresses map directly to
# physical addresses so that the
# effective memory map doesn't change
# during the transition.
lgdt    gdtdesc
movl    %cr0, %eax
orl     $CR0_PE, %eax
movl    %eax, %cr0
```

Protected Mode

- **Finally we can jump into *protected mode.* The change happens only once a segment register is modified, so we execute a *long jump* to set %cs**

- **From here on out, we're executing 32-bit x86 code**

```
ljmp    $(SEG_KCODE<<3), $start32
```

```
.code32  # Tell assembler to generate 32-bit code now.
start32:
  # Set up the protected-mode data segment registers
  movw    $(SEG_KDATA<<3), %ax    # Our data segment selector
  movw    %ax, %ds                # -> DS: Data Segment
  movw    %ax, %es                # -> ES: Extra Segment
  movw    %ax, %ss                # -> SS: Stack Segment
  movw    $0, %ax                 # Zero segments not ready for use
  movw    %ax, %fs                # -> FS
  movw    %ax, %gs                # -> GS
```

Onward to C

- **Finally, we can set up the stack and jump into C code**
- **The stack pointer is set to 0x7c00 – the address of the start of the bootloader**

```
# Set up the stack pointer and call into C.
movl    $start, %esp
call    bootmain
```

Today: Machine Programming Basics

- **Assembly**
  - More examples
  - Function calls
- **GDB**
- **Booting the OS**
  - BIOS
  - Assembly Bootstrap
  - **C Bootstrap**

bootmain.c

- Reads in the kernel header from the hard disk
- Parses the kernel headers – it's a standard ELF (Executable and Linkable Format) file that gcc emits
- Reads each section (kernel code, data) into memory at the address specified in the ELF headers
- Finally, calls the *entry point* defined in the ELF file

- **Bootmain.c**

```c
void
bootmain(void)
{
  struct elfhdr *elf;
  struct proghdr *ph, *eph;
  void (*entry)(void);
  uchar* pa;

  elf = (struct elfhdr*)0x10000;  // scratch space

  // Read 1st page off disk
  readseg((uchar*)elf, 4096, 0);

  // Is this an ELF executable?
  if(elf->magic != ELF_MAGIC)
    return;  // let bootasm.S handle error

  // Load each program segment (ignores ph flags).
  ph = (struct proghdr*)((uchar*)elf + elf->phoff);
  eph = ph + elf->phnum;
  for(; ph < eph; ph++){
    pa = (uchar*)ph->paddr;
    readseg(pa, ph->filesz, ph->off);
    if(ph->memsz > ph->filesz)
      stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
  }

  // Call the entry point from the ELF header.
  // Does not return!
  entry = (void(*)(void))(elf->entry);
  entry();
}
```

kernel.ld

- The kernel will eventually be loaded at 0x80100000 (virtual)
- For now though, we have only physical memory, not virtual memory
- Luckily, we can use a *linker script* to specify the physical load address, along with the entry point

```
/* Simple linker script for the JOS kernel.
   See the GNU ld 'info' manual ("info ld") to learn the syntax. */

OUTPUT_FORMAT("elf32-i386", "elf32-i386", "elf32-i386")
OUTPUT_ARCH(i386)
ENTRY(_start)

SECTIONS
{
    /* Link the kernel at this address: "." means the current address */
        /* Must be equal to KERNLINK */
    . = 0x80100000;

    .text : AT(0x100000) {
        *(.text .stub .text.* .gnu.linkonce.t.*)
    }
```

Note that 0x80100000 = 0x100000 + 0x80000000
So during early boot we can translate between "virtual" and physical addresses by simple addition & subtraction

Reading from the Hard Drive

- **You can talk to an IDE hard drive using port I/O (PIO)**
  - This mode is not very fast, but it's simple to implement
  - An alternative is DMA (Direct Memory Access), which allows data transfer to bypass the CPU
  - xv6 only implements PIO mode

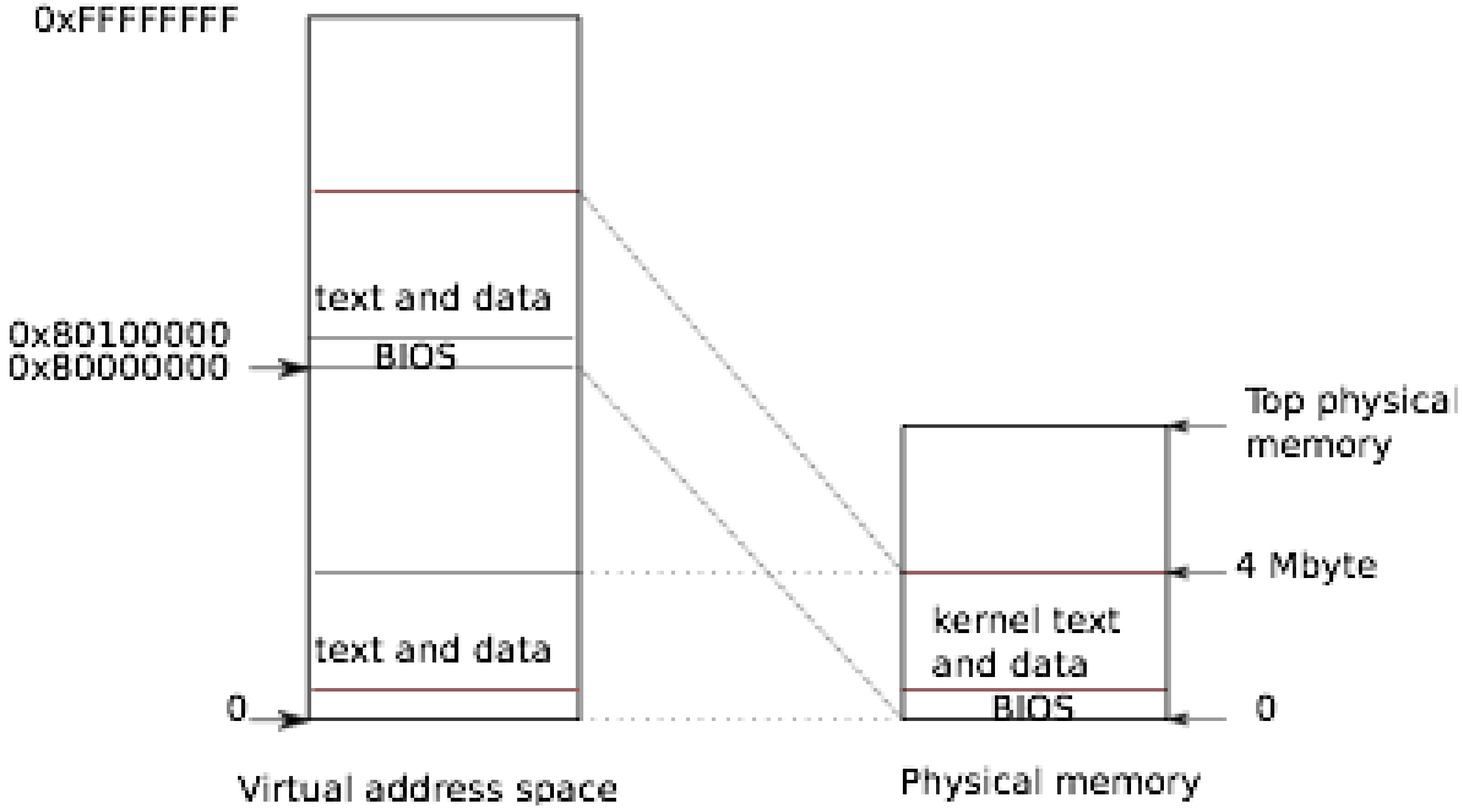
```c
// Read a single sector at offset into dst.
void
readsect(void *dst, uint offset)
{
  // Issue command.
  waitdisk();
  outb(0x1F2, 1);   // count = 1
  outb(0x1F3, offset);
  outb(0x1F4, offset >> 8);
  outb(0x1F5, offset >> 16);
  outb(0x1F6, (offset >> 24) | 0xE0);
  outb(0x1F7, 0x20);  // cmd 0x20 - read sectors

  // Read data.
  waitdisk();
  insl(0x1F0, dst, SECTSIZE/4);
}
```

Kernel Entry Point

- Defined in entry.S
- This is where we finally turn on paging, but for now we just use a very simple mapping:

  0x80000000 => 0x00000000


0xFFFFFFFF
text and data
0x80100000
BIOS
0x80000000
text and data
0
Virtual address space
Top physical memory
4 Mbyte
kernel text and data
BIOS
0
Physical memory

```
# Entering xv6 on boot processor, with paging off.
.globl entry
entry:
  # Turn on page size extension for 4Mbyte pages
  movl    %cr4, %eax
  orl     $(CR4_PSE), %eax
  movl    %eax, %cr4
  # Set page directory
  movl    $(V2P_WO(entrypgdir)), %eax
  movl    %eax, %cr3
  # Turn on paging.
  movl    %cr0, %eax
  orl     $(CR0_PG|CR0_WP), %eax
  movl    %eax, %cr0
```

Kernel Main

```
// Bootstrap processor starts running C code here.
// Allocate a real stack and switch to it, first
// doing some setup required for memory allocator to work.
int
main(void)
{
  kinit1(end, P2V(4*1024*1024)); // phys page allocator
  kvmalloc();      // kernel page table
  mpinit();        // collect info about this machine
  lapicinit();
  seginit();       // set up segments
  cprintf("\ncpu%d: starting xv6\n\n", cpu->id);
  picinit();       // interrupt controller
  ioapicinit();    // another interrupt controller
  consoleinit();   // I/O devices & their interrupts
  uartinit();      // serial port
  pinit();         // process table
  tvinit();        // trap vectors
  binit();         // buffer cache
  fileinit();      // file table
  ideinit();       // disk
  if(!ismp)
    timerinit();   // uniprocessor timer
  startothers();   // start other processors
  kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
  userinit();      // first user process
  // Finish setting up this processor in mpmain.
  mpmain();
}
```

Creating the First Process

- **This happens in userinit() – after we've initialized lots of other hardware [proc.c:79]**
- **The first process will run the code in initcode.S**
- **To do that it needs:**
  - A process structure filled out and assigned a slot in the process table (allocproc())
  - An address space (setupkvm())

initcode.S

```
# exec(init, argv)
.globl start
start:
  pushl $argv
  pushl $init
  pushl $0  // where caller pc would be
  movl $SYS_exec, %eax
  int $T_SYSCALL
[...]
# char init[] = "/init\0";
init:
  .string "/init\0"
```