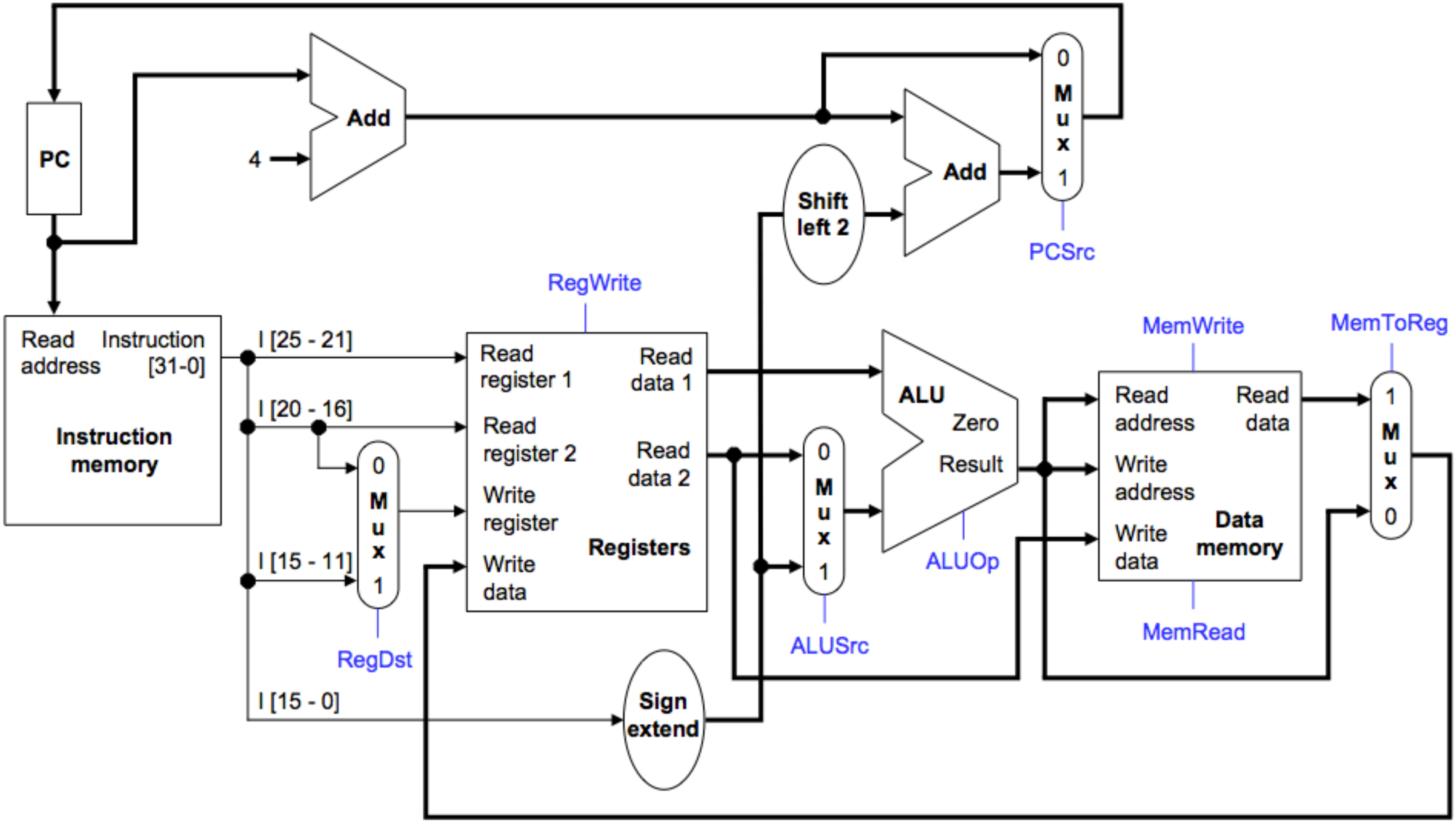
Software Security
Part 1

C.V. Wright

CS 491/591

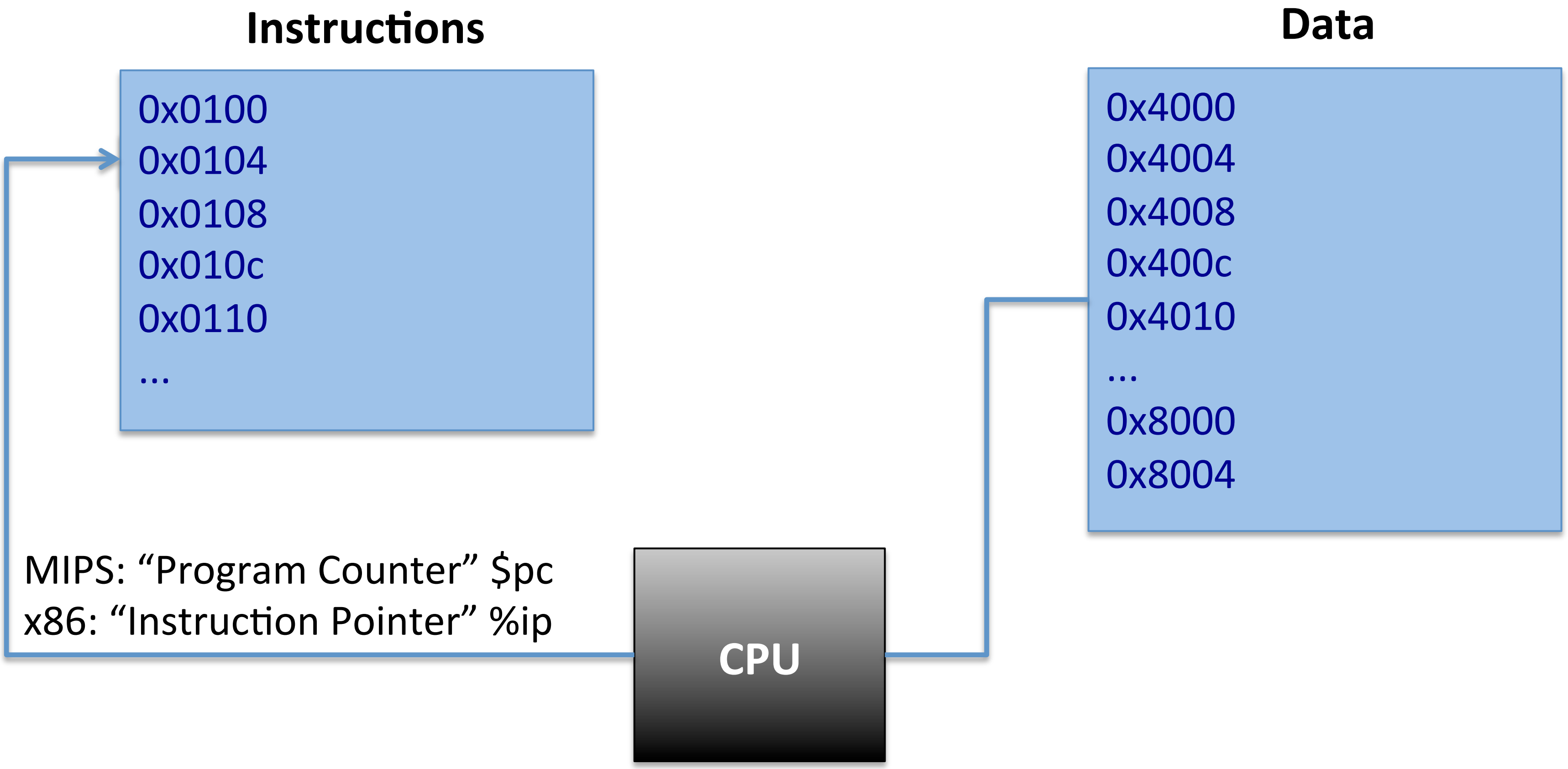
Fall 2015

Datapath for a Simple MIPS CPU



Hennessey and Patterson, *Computer Organization and Design*, 2nd ed. P. 358.

Low-Level View of Program Execution



Instructions

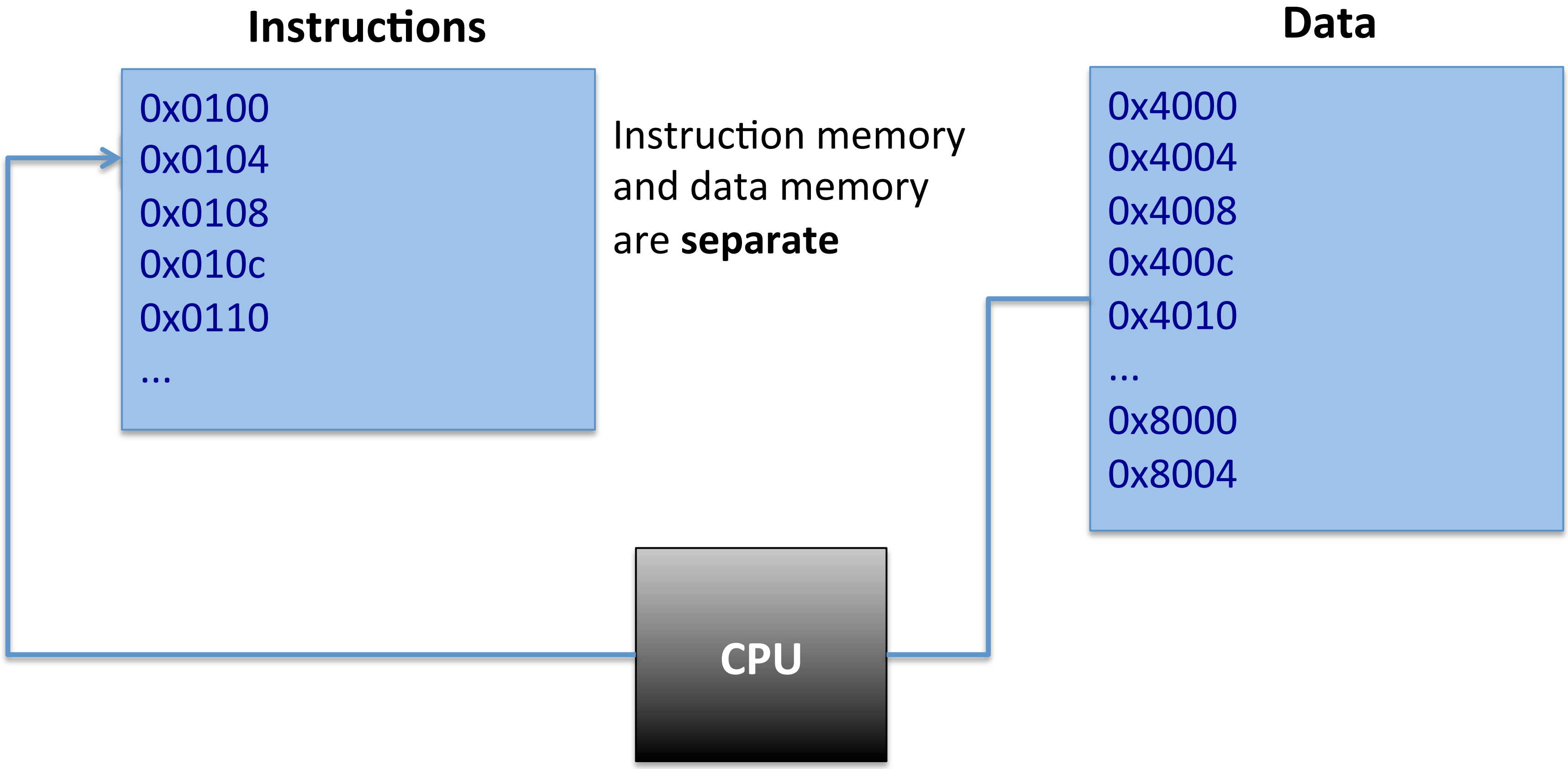
0x0100
0x0104
0x0108
0x010c
0x0110
…

Data

0x4000
0x4004
0x4008
0x400c
0x4010
…
0x8000
0x8004

MIPS: “Program Counter” $pc
x86: “Instruction Pointer” %ip

Harvard Architecture



Instructions

0x0100
0x0104
0x0108
0x010c
0x0110
…

Instruction memory
and data memory
are **separate**

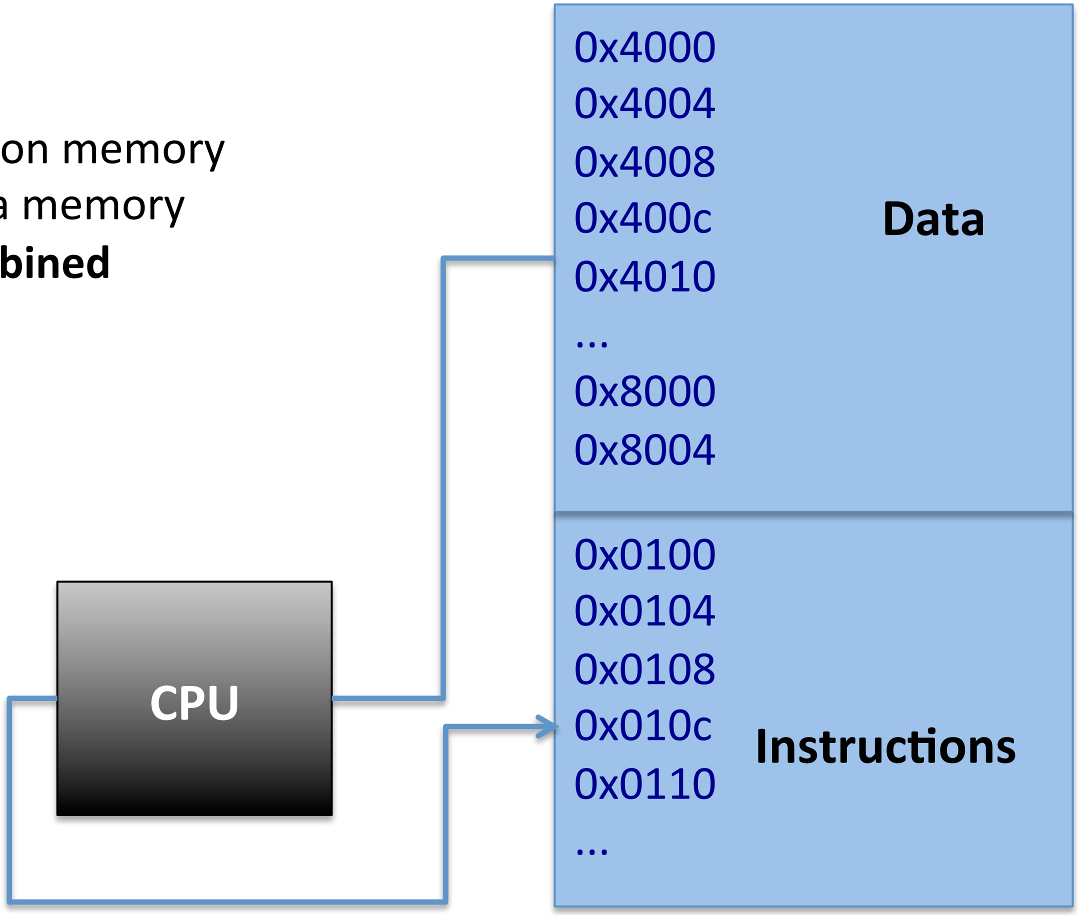
Data

0x4000
0x4004
0x4008
0x400c
0x4010
…
0x8000
0x8004

CPU

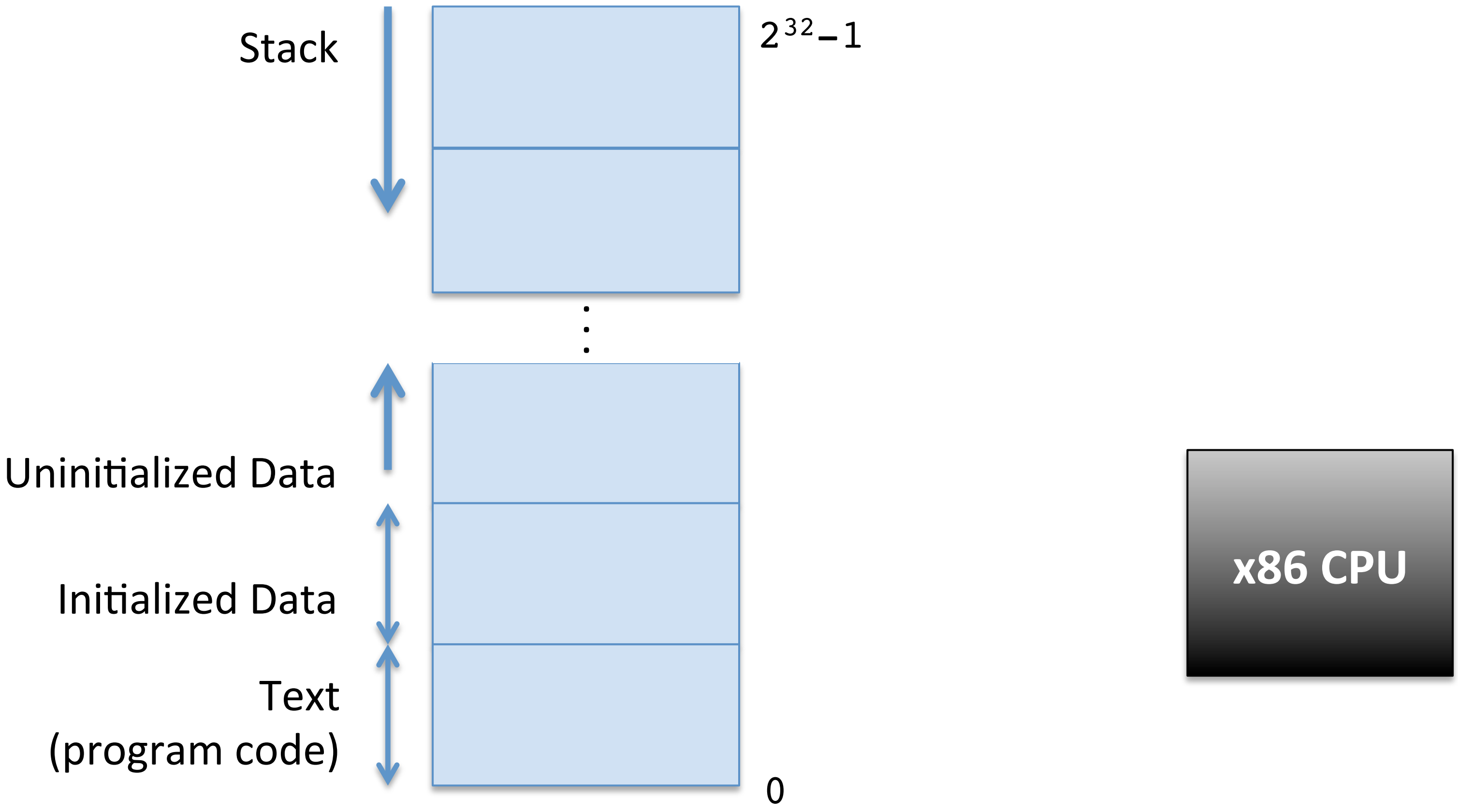
Von Neumann Architecture

Instruction memory
and data memory
are **combined**



In-Memory Layout of a Process

Process Virtual Address Space



Stack

$2^{32}-1$

Uninitialized Data

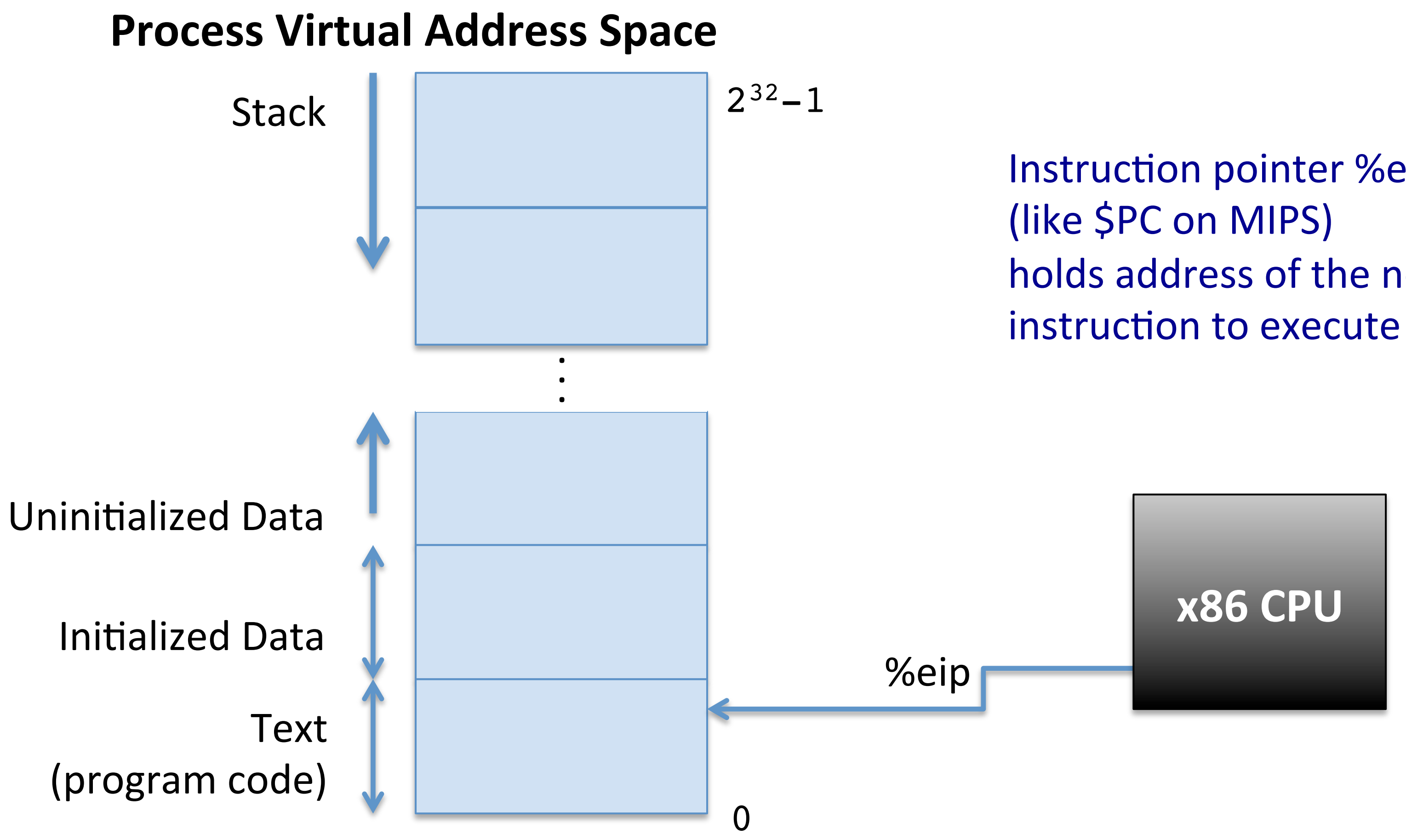
Initialized Data

Text
(program code)

0

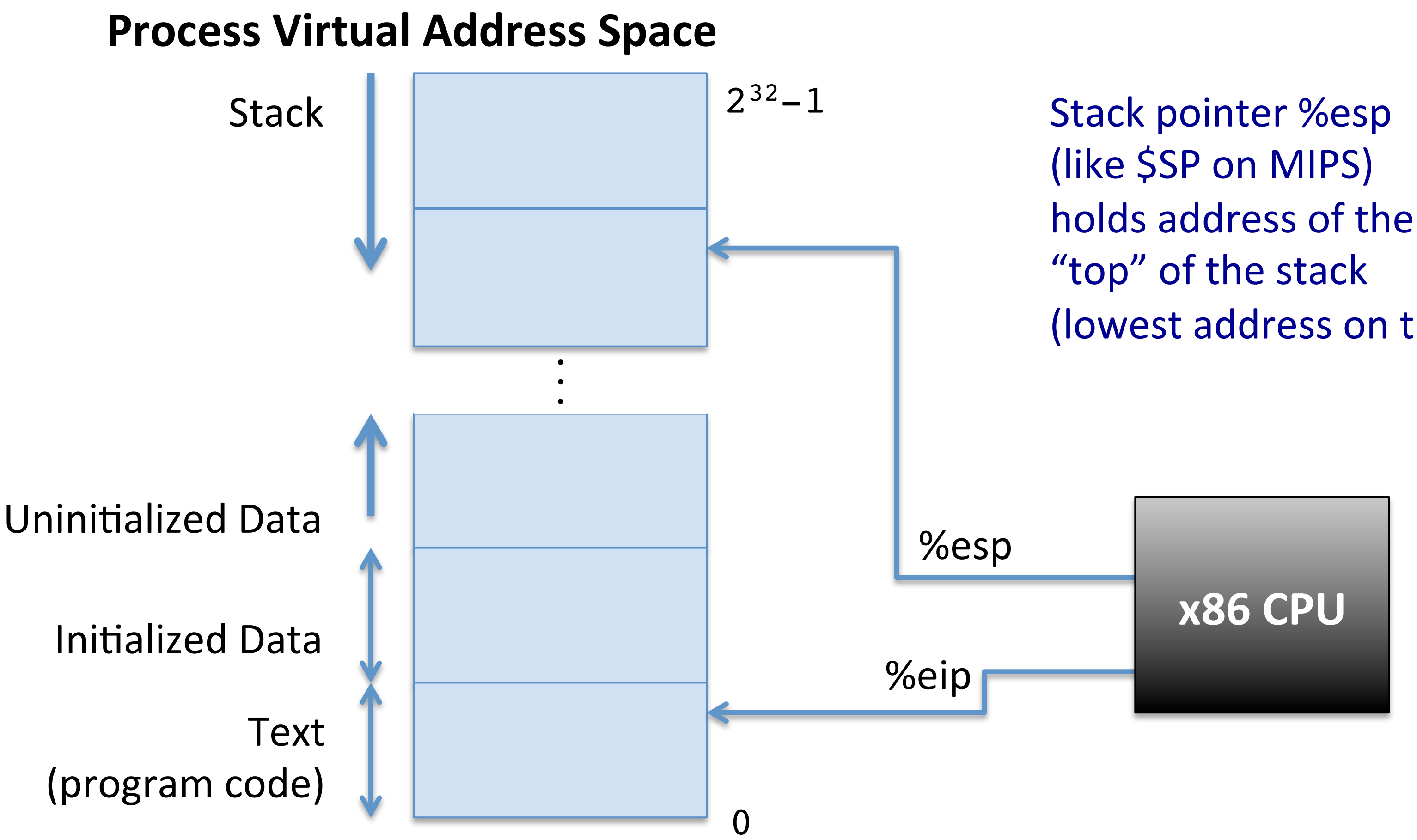
x86 CPU

In-Memory Layout of a Process



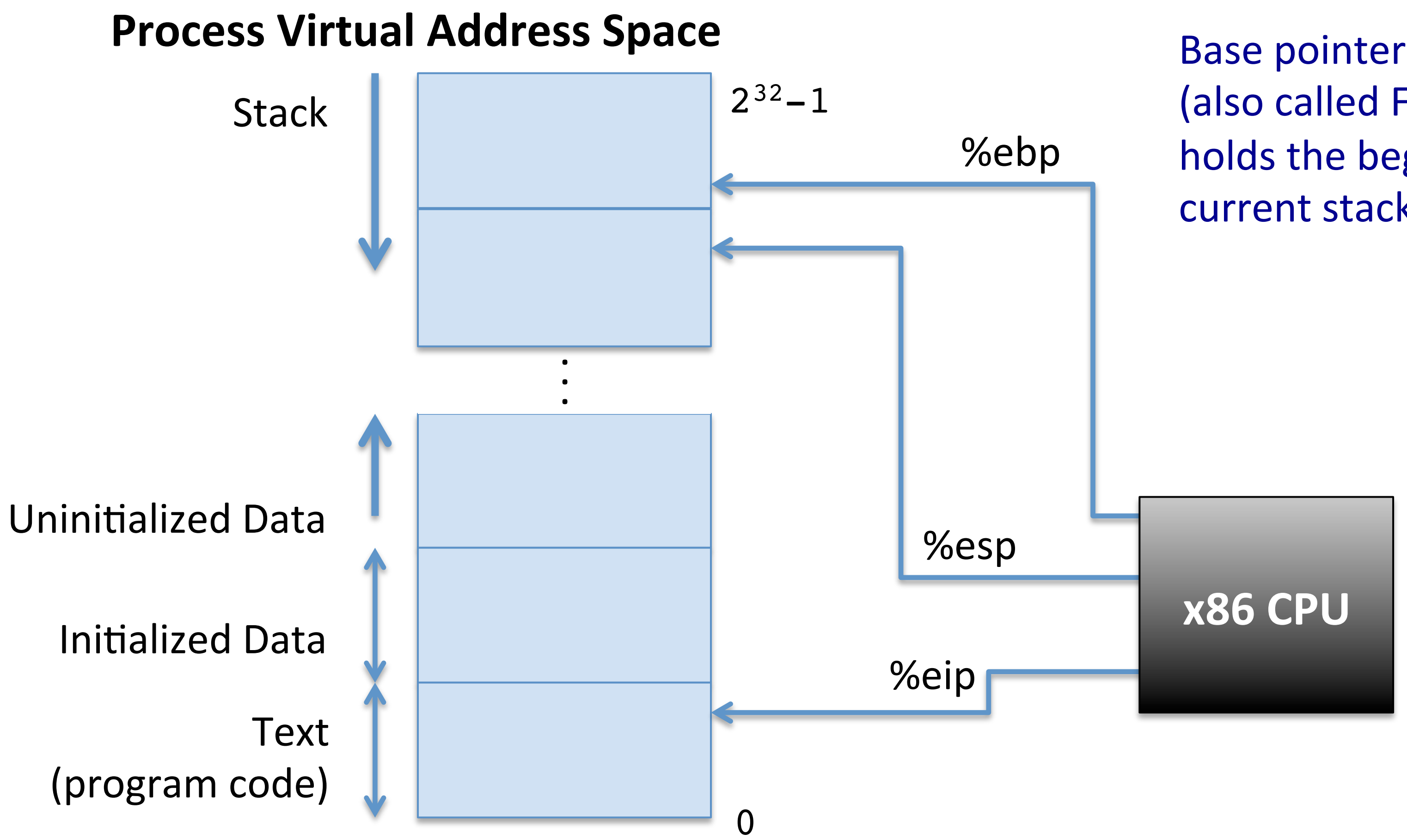
Instruction pointer %eip (like $PC on MIPS) holds address of the next instruction to execute

In-Memory Layout of a Process



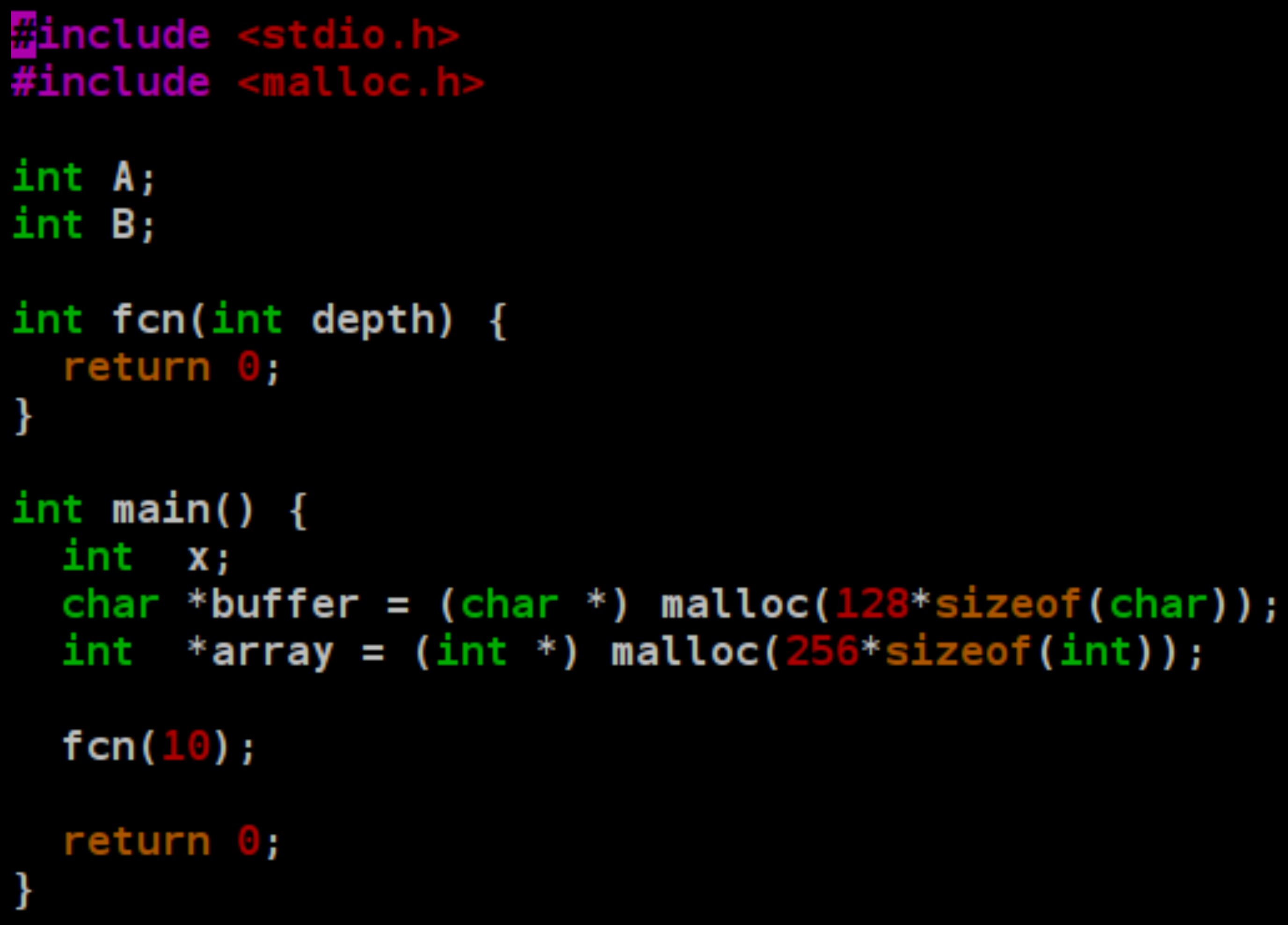
Stack pointer %esp
(like $SP on MIPS)
holds address of the
“top” of the stack
(lowest address on the stack)

In-Memory Layout of a Process



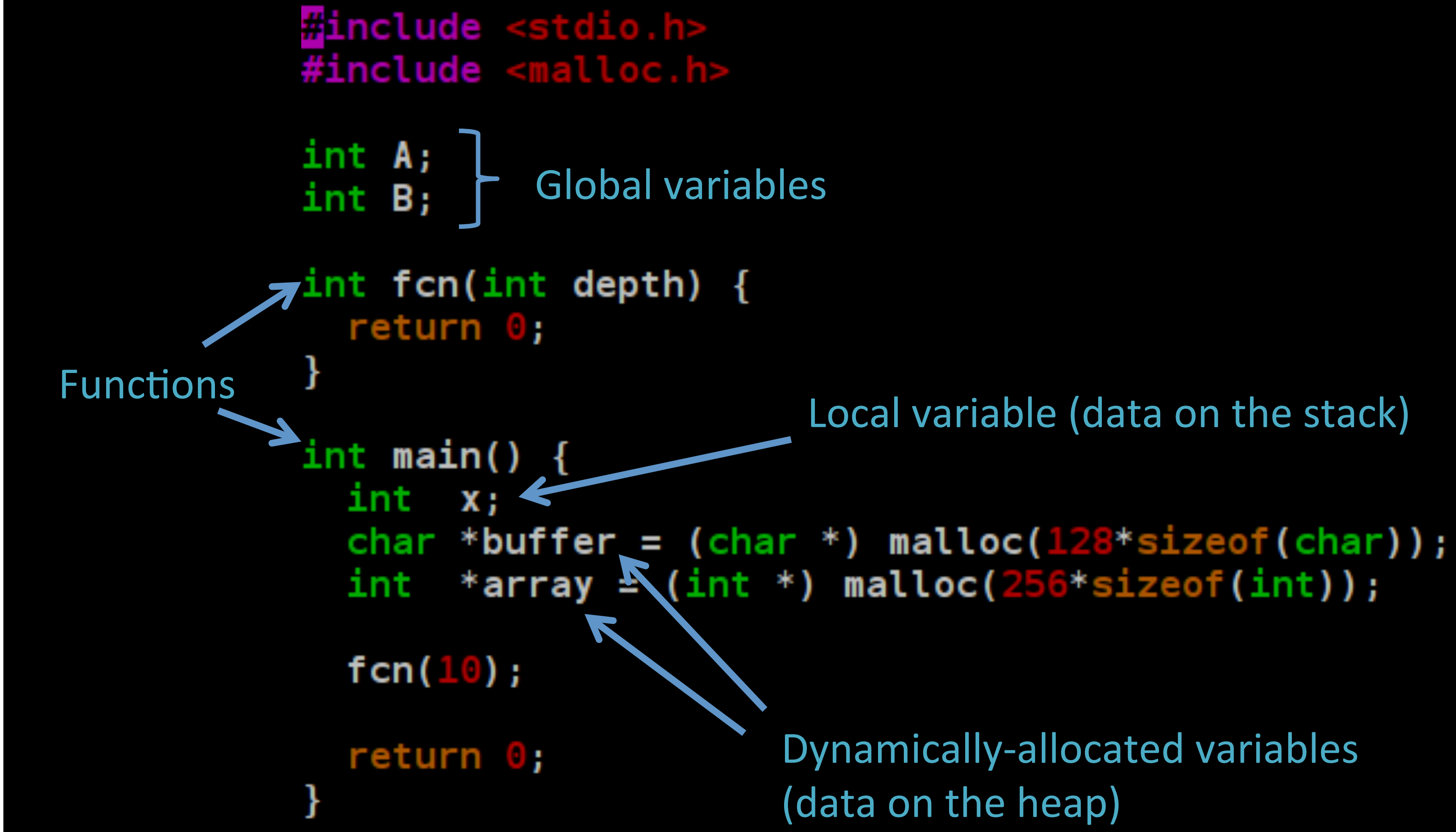
Base pointer %ebp
(also called Frame Pointer)
holds the beginning of the
current stack frame

Example Program



```c
#include <stdio.h>
#include <malloc.h>

int A;
int B;

int fcn(int depth) {
  return 0;
}

int main() {
  int  x;
  char *buffer = (char *) malloc(128*sizeof(char));
  int  *array = (int *) malloc(256*sizeof(int));

  fcn(10);

  return 0;
}
```

Example Program



```
#include <stdio.h>
#include <malloc.h>

int A;
int B;

int fcn(int depth) {
  return 0;
}

int main() {
  int  x;
  char *buffer = (char *) malloc(128*sizeof(char));
  int  *array = (int *) malloc(256*sizeof(int));

  fcn(10);

  return 0;
}
```

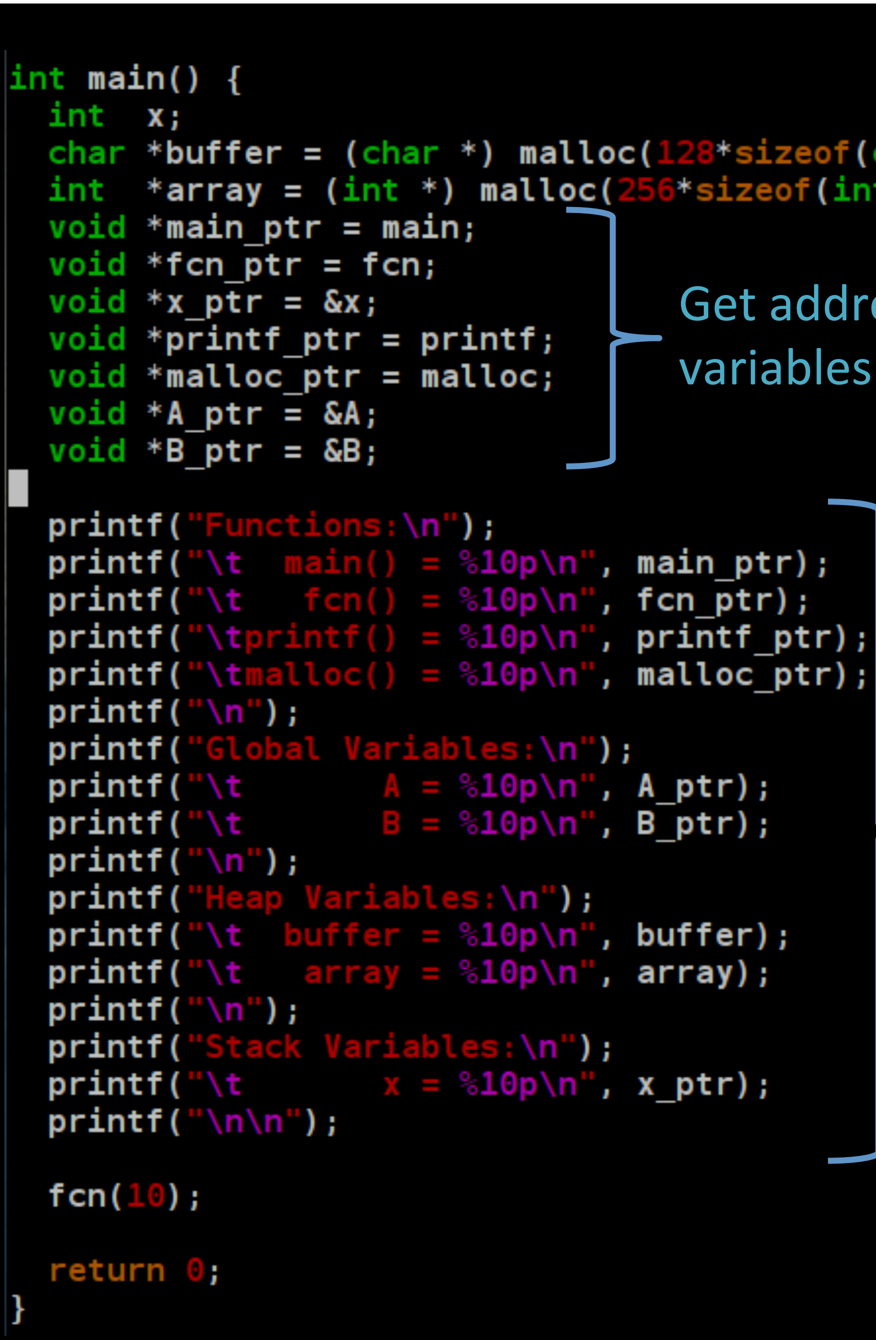
Global variables

Functions

Local variable (data on the stack)

Dynamically-allocated variables (data on the heap)

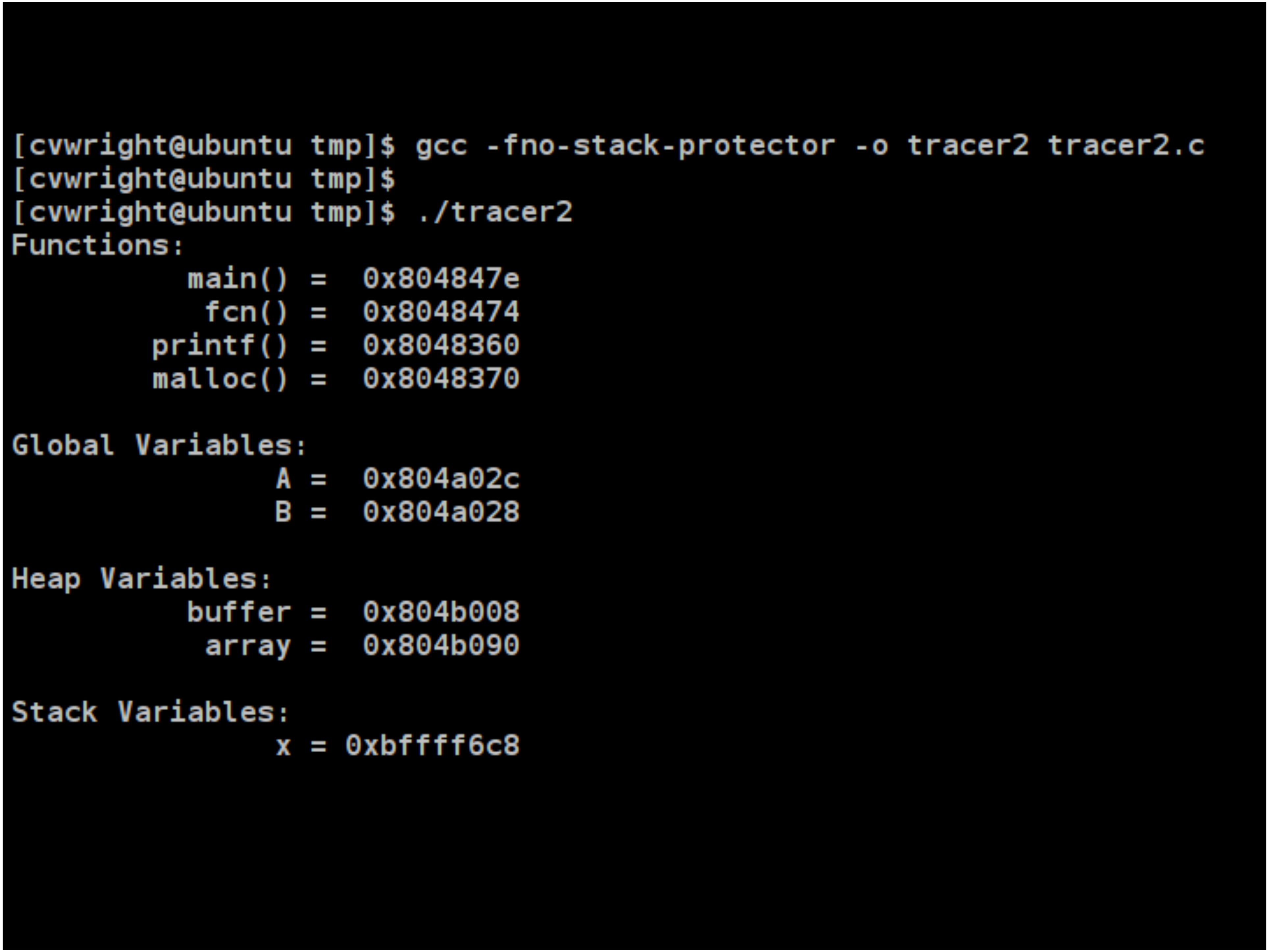
Let’s add some instrumentation to help us see what’s going on



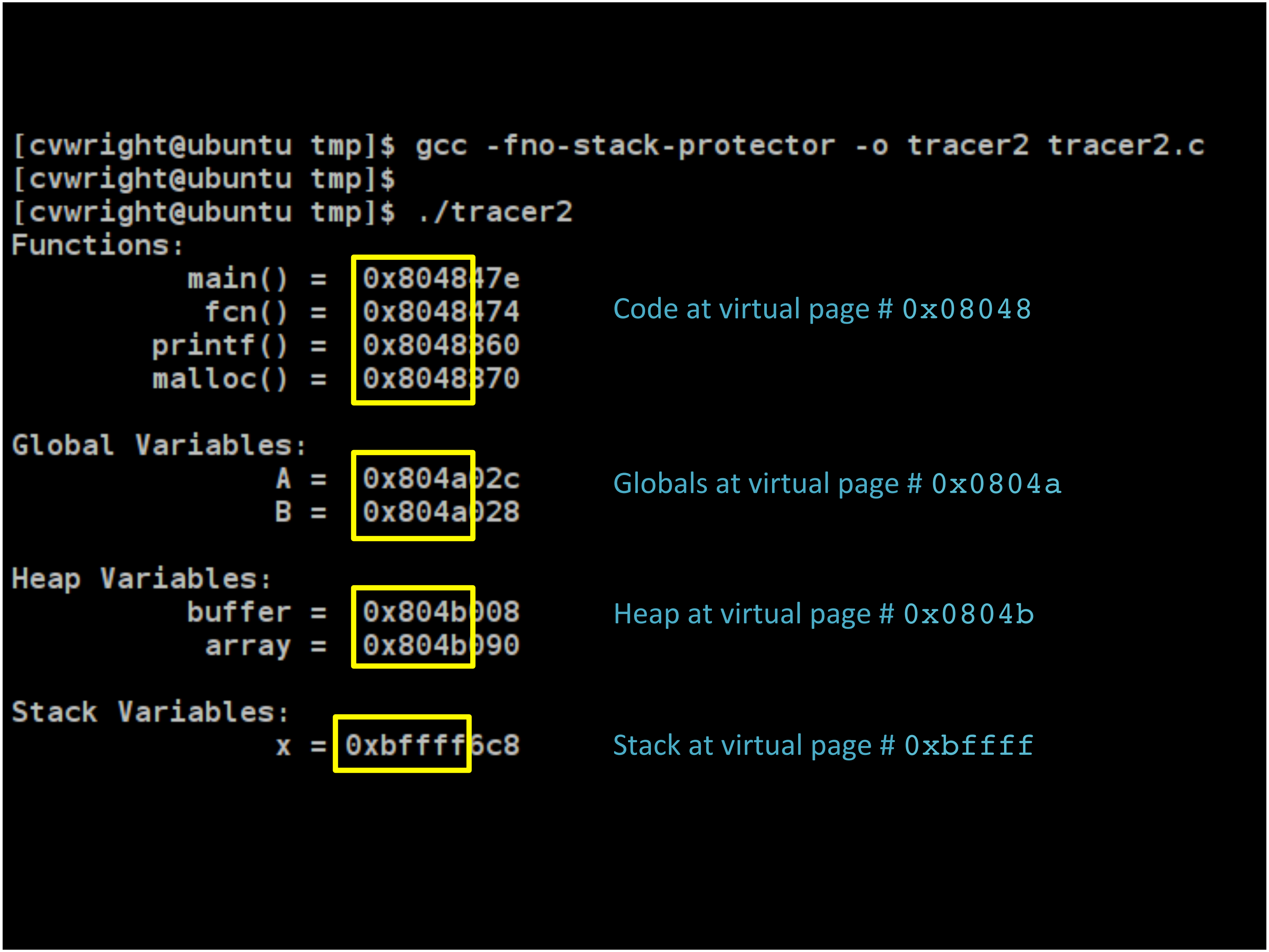
```
int main() {
  int  x;
  char *buffer = (char *) malloc(128*sizeof(char));
  int  *array = (int *) malloc(256*sizeof(int));
  void *main_ptr = main;
  void *fcn_ptr = fcn;
  void *x_ptr = &x;
  void *printf_ptr = printf;
  void *malloc_ptr = malloc;
  void *A_ptr = &A;
  void *B_ptr = &B;

  printf("Functions:\n");
  printf("\t  main() = %10p\n", main_ptr);
  printf("\t   fcn() = %10p\n", fcn_ptr);
  printf("\tprintf() = %10p\n", printf_ptr);
  printf("\tmalloc() = %10p\n", malloc_ptr);
  printf("\n");
  printf("Global Variables:\n");
  printf("\t       A = %10p\n", A_ptr);
  printf("\t       B = %10p\n", B_ptr);
  printf("\n");
  printf("Heap Variables:\n");
  printf("\t  buffer = %10p\n", buffer);
  printf("\t   array = %10p\n", array);
  printf("\n");
  printf("Stack Variables:\n");
  printf("\t       x = %10p\n", x_ptr);
  printf("\n\n");

  fcn(10);

  return 0;
}
```

Get addresses of variables in memory

Print addresses in hex



```
[cvwright@ubuntu tmp]$ gcc -fno-stack-protector -o tracer2 tracer2.c
[cvwright@ubuntu tmp]$
[cvwright@ubuntu tmp]$ ./tracer2
Functions:
          main() =  0x804847e
           fcn() =  0x8048474
        printf() =  0x8048360
        malloc() =  0x8048370

Global Variables:
               A =  0x804a02c
               B =  0x804a028

Heap Variables:
          buffer =  0x804b008
           array =  0x804b090

Stack Variables:
               x = 0xbffff6c8
```



```
[cvwright@ubuntu tmp]$ gcc -fno-stack-protector -o tracer2 tracer2.c
[cvwright@ubuntu tmp]$
[cvwright@ubuntu tmp]$ ./tracer2
Functions:
          main() =  0x804847e
           fcn() =  0x8048474
        printf() =  0x8048360
        malloc() =  0x8048370

Global Variables:
               A =  0x804a02c
               B =  0x804a028

Heap Variables:
          buffer =  0x804b008
           array =  0x804b090

Stack Variables:
               x = 0xbffff6c8
```

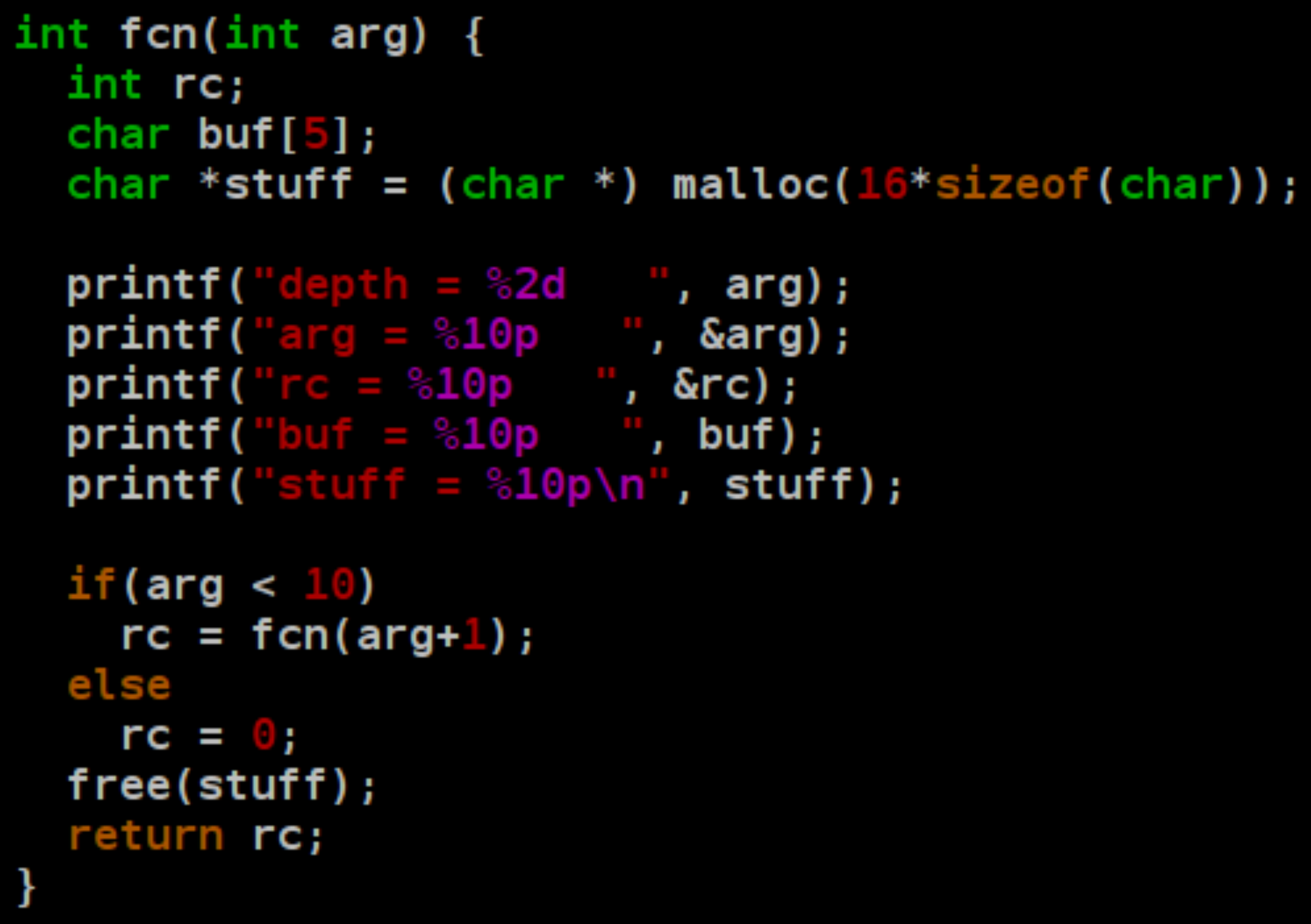
Code at virtual page # 0x08048

Globals at virtual page # 0x0804a

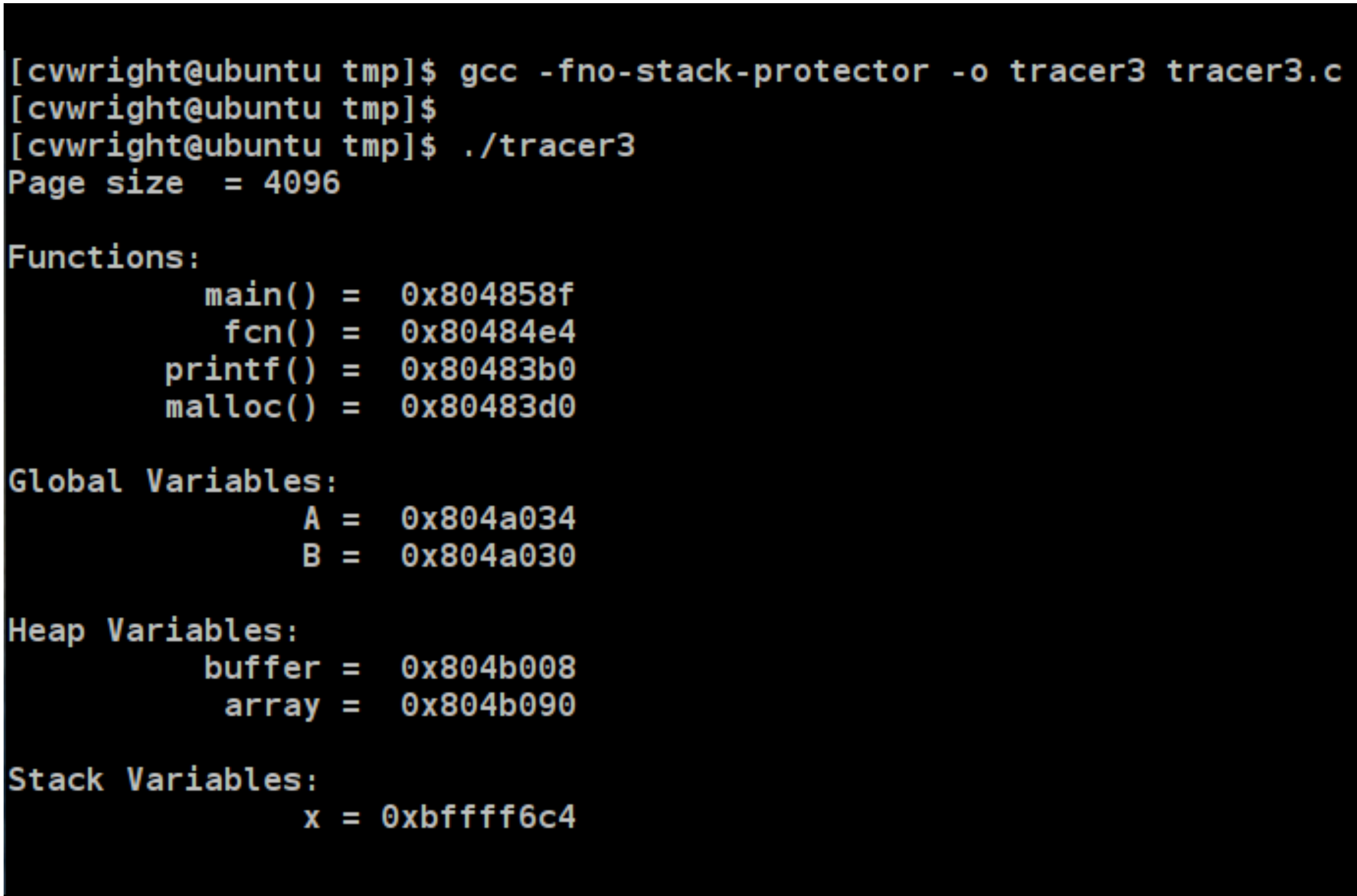
Heap at virtual page # 0x0804b

Stack at virtual page # 0xbffff

More instrumentation for function calls

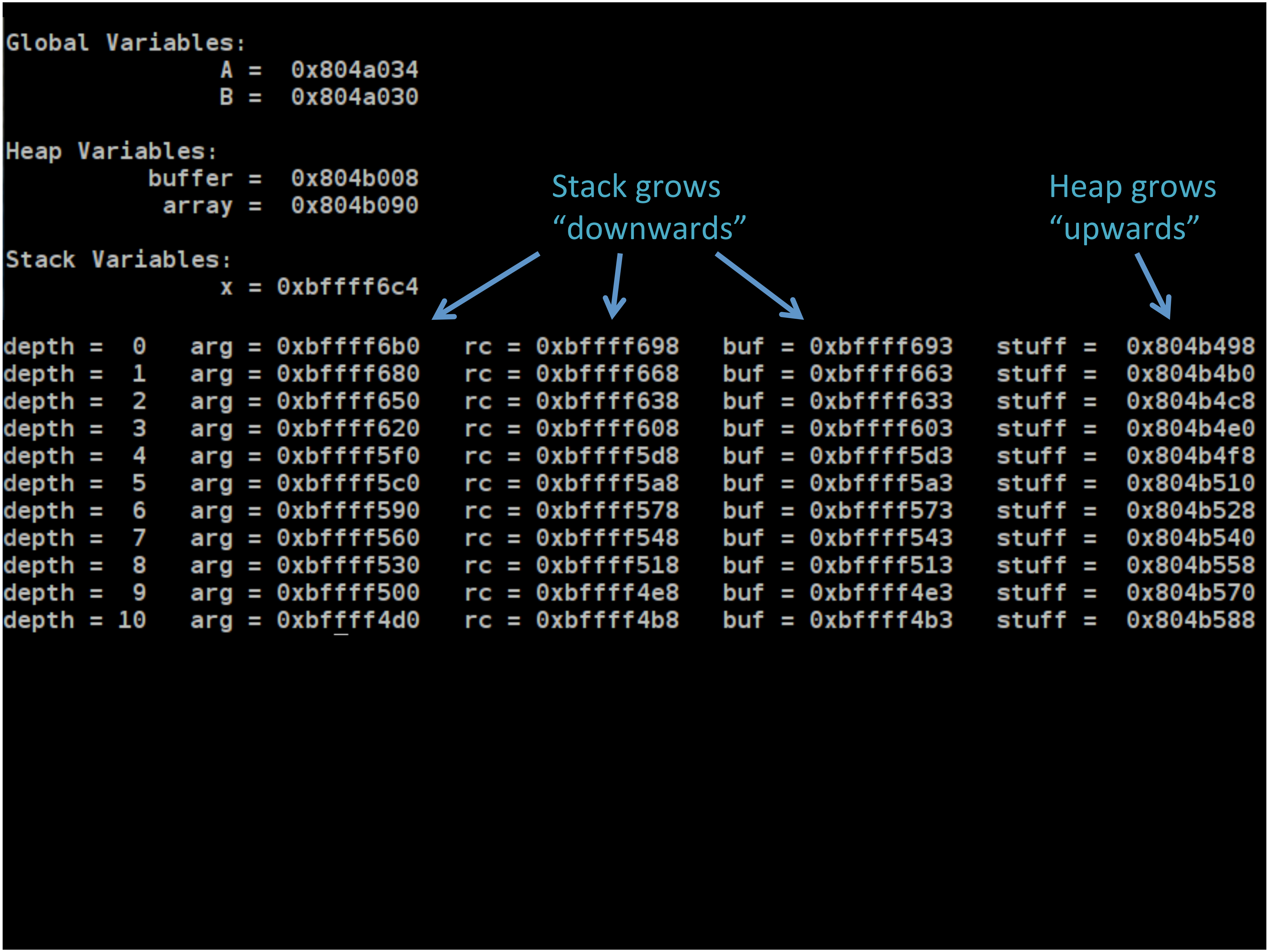


```
int fcn(int arg) {
  int rc;
  char buf[5];
  char *stuff = (char *) malloc(16*sizeof(char));

  printf("depth = %2d   ", arg);
  printf("arg = %10p   ", &arg);
  printf("rc = %10p   ", &rc);
  printf("buf = %10p   ", buf);
  printf("stuff = %10p\n", stuff);

  if(arg < 10)
    rc = fcn(arg+1);
  else
    rc = 0;
  free(stuff);
  return rc;
}
```



```
[cvwright@ubuntu tmp]$ gcc -fno-stack-protector -o tracer3 tracer3.c
[cvwright@ubuntu tmp]$
[cvwright@ubuntu tmp]$ ./tracer3
Page size  = 4096

Functions:
          main() =  0x804858f
           fcn() =  0x80484e4
        printf() =  0x80483b0
        malloc() =  0x80483d0

Global Variables:
               A =  0x804a034
               B =  0x804a030

Heap Variables:
          buffer =  0x804b008
           array =  0x804b090

Stack Variables:
               x = 0xbffff6c4
```

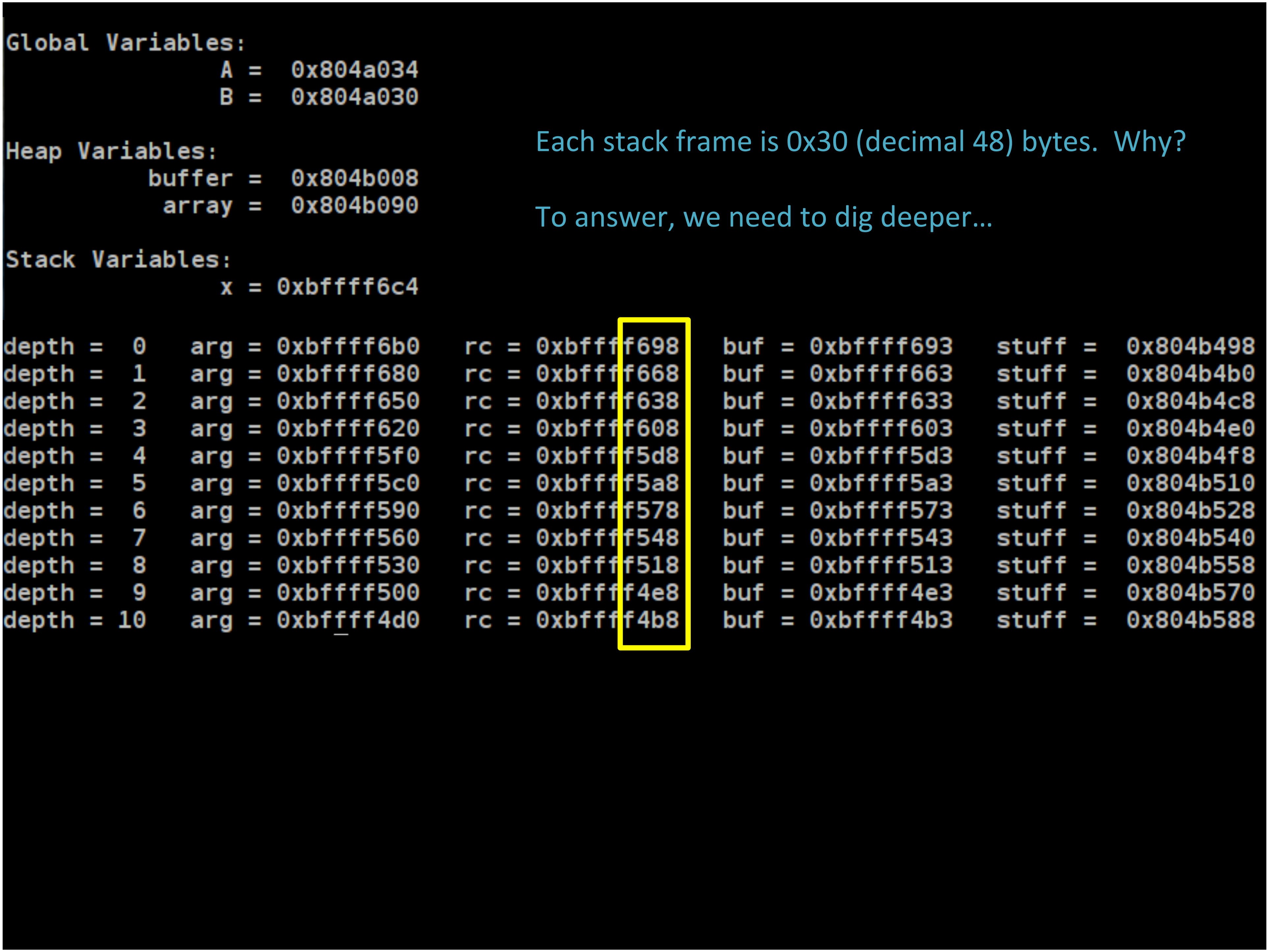
(output continues on next slide)



```
Global Variables:
              A =  0x804a034
              B =  0x804a030

Heap Variables:
         buffer =  0x804b008
          array =  0x804b090

Stack Variables:
              x = 0xbffff6c4

depth =  0   arg = 0xbffff6b0   rc = 0xbffff698   buf = 0xbffff693   stuff =  0x804b498
depth =  1   arg = 0xbffff680   rc = 0xbffff668   buf = 0xbffff663   stuff =  0x804b4b0
depth =  2   arg = 0xbffff650   rc = 0xbffff638   buf = 0xbffff633   stuff =  0x804b4c8
depth =  3   arg = 0xbffff620   rc = 0xbffff608   buf = 0xbffff603   stuff =  0x804b4e0
depth =  4   arg = 0xbffff5f0   rc = 0xbffff5d8   buf = 0xbffff5d3   stuff =  0x804b4f8
depth =  5   arg = 0xbffff5c0   rc = 0xbffff5a8   buf = 0xbffff5a3   stuff =  0x804b510
depth =  6   arg = 0xbffff590   rc = 0xbffff578   buf = 0xbffff573   stuff =  0x804b528
depth =  7   arg = 0xbffff560   rc = 0xbffff548   buf = 0xbffff543   stuff =  0x804b540
depth =  8   arg = 0xbffff530   rc = 0xbffff518   buf = 0xbffff513   stuff =  0x804b558
depth =  9   arg = 0xbffff500   rc = 0xbffff4e8   buf = 0xbffff4e3   stuff =  0x804b570
depth = 10   arg = 0xbffff4d0   rc = 0xbffff4b8   buf = 0xbffff4b3   stuff =  0x804b588
```

Stack grows
“downwards”

Heap grows
“upwards”



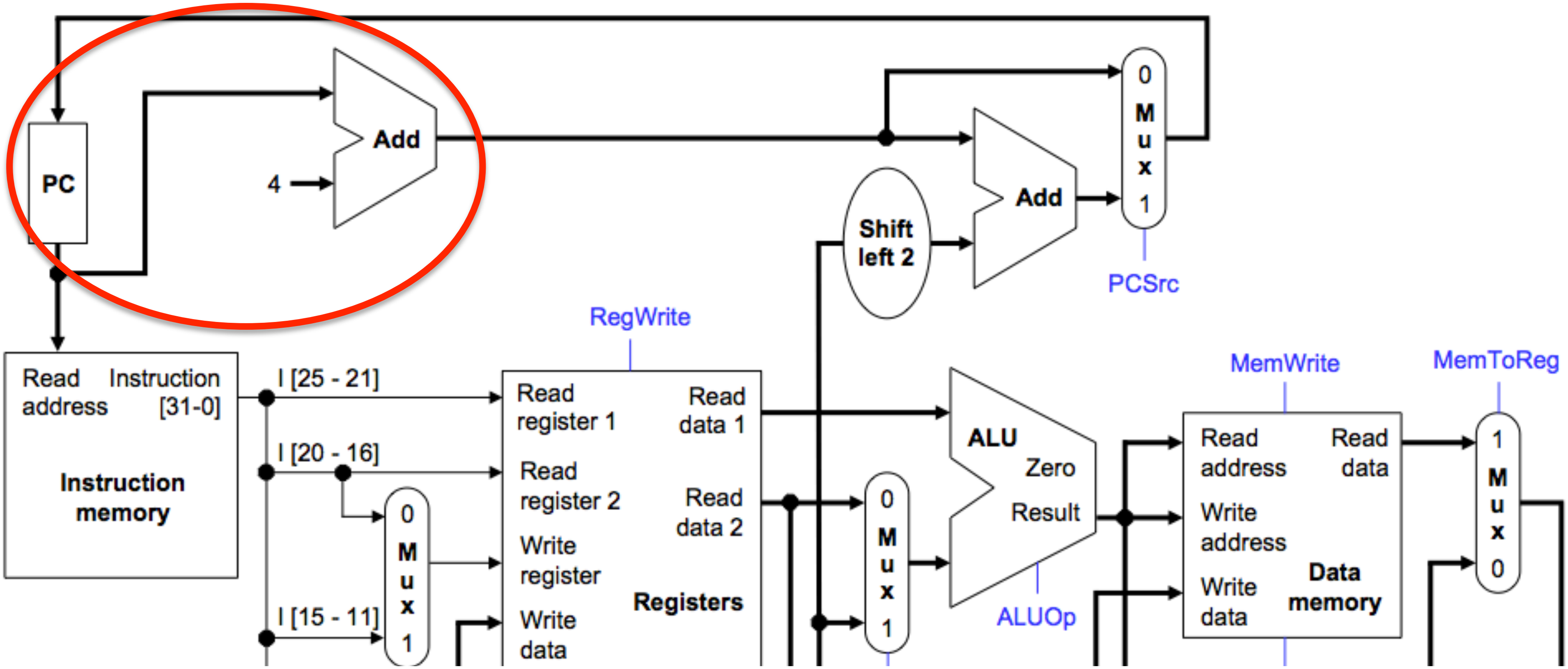
```
Global Variables:
               A =  0x804a034
               B =  0x804a030

Heap Variables:
          buffer =  0x804b008
           array =  0x804b090

Stack Variables:
               x = 0xbffff6c4

depth =  0   arg = 0xbffff6b0   rc = 0xbffff698   buf = 0xbffff693   stuff =  0x804b498
depth =  1   arg = 0xbffff680   rc = 0xbffff668   buf = 0xbffff663   stuff =  0x804b4b0
depth =  2   arg = 0xbffff650   rc = 0xbffff638   buf = 0xbffff633   stuff =  0x804b4c8
depth =  3   arg = 0xbffff620   rc = 0xbffff608   buf = 0xbffff603   stuff =  0x804b4e0
depth =  4   arg = 0xbffff5f0   rc = 0xbffff5d8   buf = 0xbffff5d3   stuff =  0x804b4f8
depth =  5   arg = 0xbffff5c0   rc = 0xbffff5a8   buf = 0xbffff5a3   stuff =  0x804b510
depth =  6   arg = 0xbffff590   rc = 0xbffff578   buf = 0xbffff573   stuff =  0x804b528
depth =  7   arg = 0xbffff560   rc = 0xbffff548   buf = 0xbffff543   stuff =  0x804b540
depth =  8   arg = 0xbffff530   rc = 0xbffff518   buf = 0xbffff513   stuff =  0x804b558
depth =  9   arg = 0xbffff500   rc = 0xbffff4e8   buf = 0xbffff4e3   stuff =  0x804b570
depth = 10   arg = 0xbffff4d0   rc = 0xbffff4b8   buf = 0xbffff4b3   stuff =  0x804b588
```

Each stack frame is 0x30 (decimal 48) bytes. Why?

To answer, we need to dig deeper…

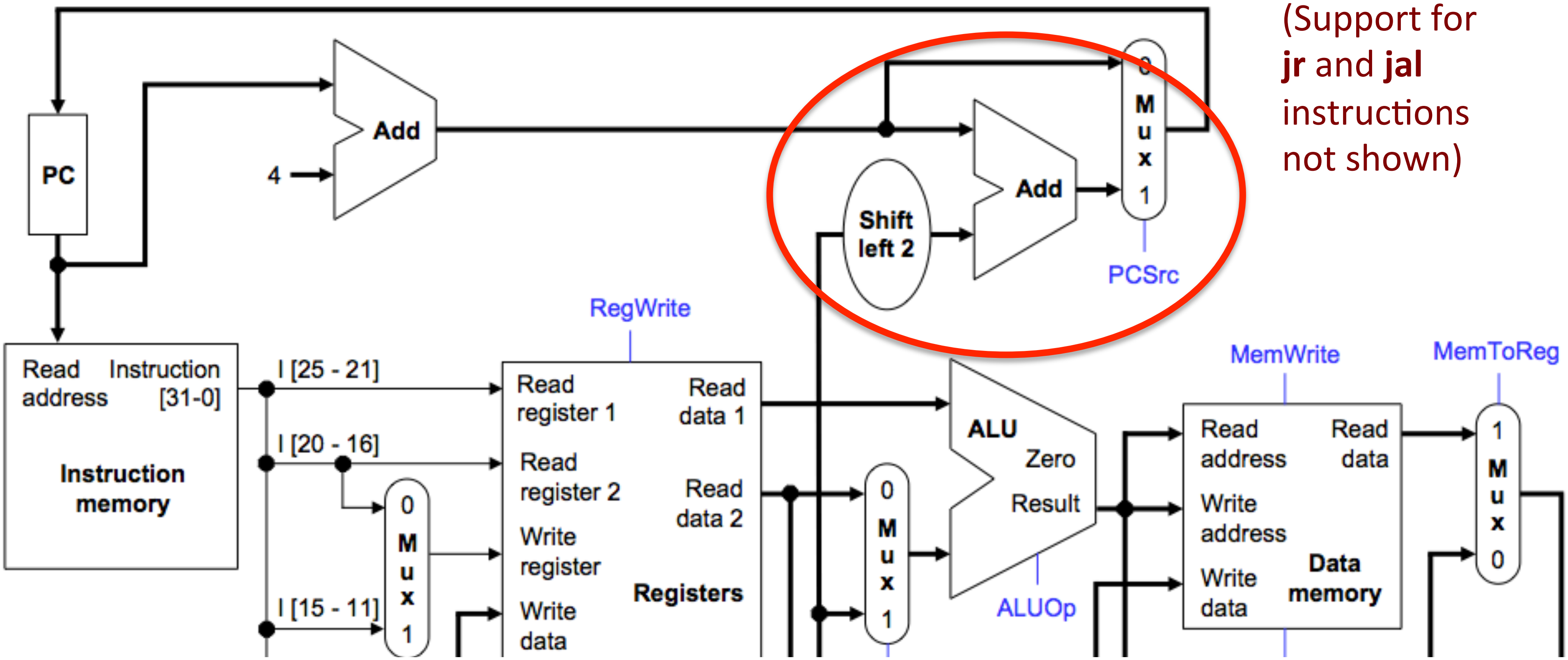
Function Call Fundamentals

- Normally, instructions are executed in order of increasing address
  - MIPS CPU adds 4 to $PC on each instruction



Function Call Fundamentals

- Function call changes control flow
  - Sets %eip (on x86) or $PC (on MIPS) to some other address
  - Starts executing code at new address



(Support for **jr** and **jal** instructions not shown)

Function Call Requirements

- Need to send arguments to the function
  - Set up register values and the stack

- Need to be able to return!
  - Remember where we were (save %eip or $PC)

Function Calls in MIPS

- Function call: “Jump and Link” Instruction
  - `jal reg`
  - Saves address of next instruction ($PC+4) in $RA
  - Sets $PC to the 32-bit value in register *reg*

- Return: no special instruction
  - Use the “jump register” instruction with $RA
  - `jr $ra`
  - Sets $PC to the 32-bit value in register $RA

Function Calls in x86/Linux

- Arguments are passed on the stack
  - Example

```
subl $4, %esp
movl %eax, (%esp)
```

Decrease %esp to extend the stack
(Like subi $sp, $sp, 4 in MIPS)

Store the value in register %eax into memory at the location held in register %esp
(Like sw $t0, $sp in MIPS)

Function Calls in x86

- Call instruction
  - **`call`** *`label`*
  - Pushes address of next instruction onto the stack
    - Sets %esp to %esp – 4
    - Stores next %eip in memory at %esp
  - Sets %eip to the address specified by *label*

Function Calls in x86

- Functions that use the stack typically start by updating the stack registers
  - Base pointer %ebp (aka “frame pointer”)
    - Points to the “bottom” (highest address) of the function’s stack frame
  - Stack pointer %esp
    - Points to the “top” (lowest address) of the stack

Function Calls in x86

- "Leave" instruction
  - **`leave`**
  - Sets %esp to the 32-bit address in %ebp
  - Loads the saved frame pointer from the stack
    - Sets %ebp to the value stored at address %esp
    - Sets %esp to %esp + 4

Function Calls in x86

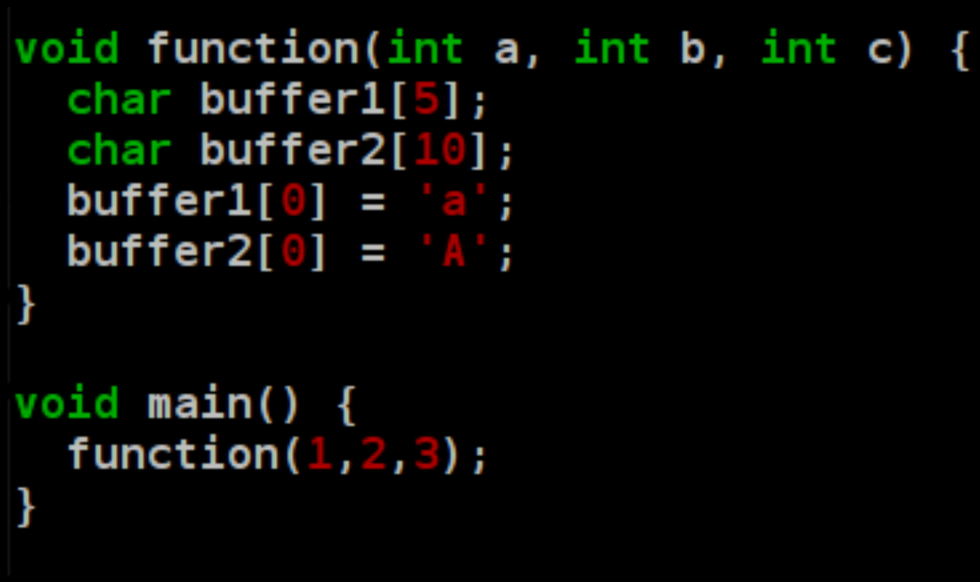
- Ret instruction
  - **`ret`** `[`*`val`*`]`

  - Loads %eip from the stack
    - Loads the 32-bit value from address %esp into %eip

  - Pops the stack *val* times (default: 0)
    - Sets %esp to %esp + val

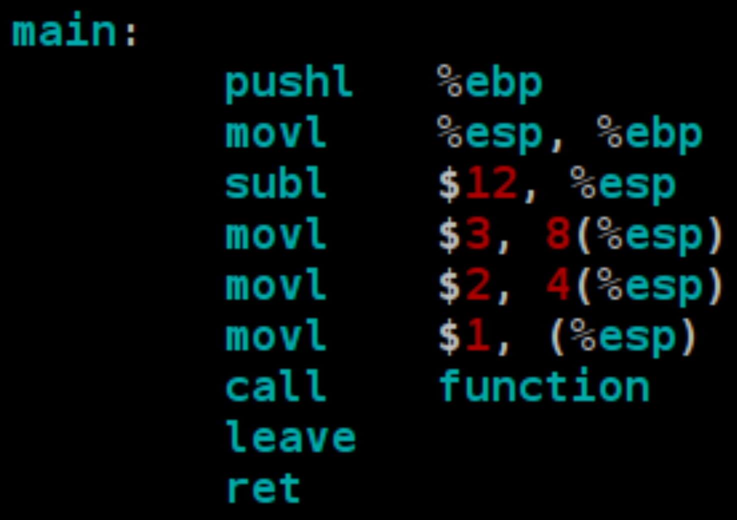
A Simpler Example Program

example1.c



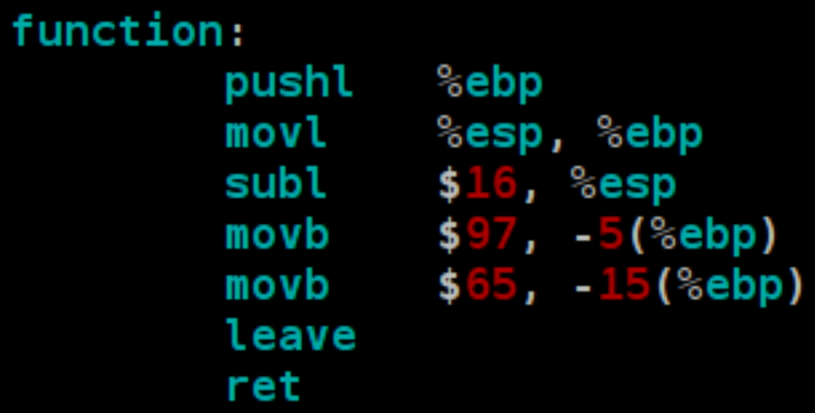
```c
void function(int a, int b, int c) {
  char buffer1[5];
  char buffer2[10];
  buffer1[0] = 'a';
  buffer2[0] = 'A';
}

void main() {
  function(1,2,3);
}
```

In Assembly: example1.s



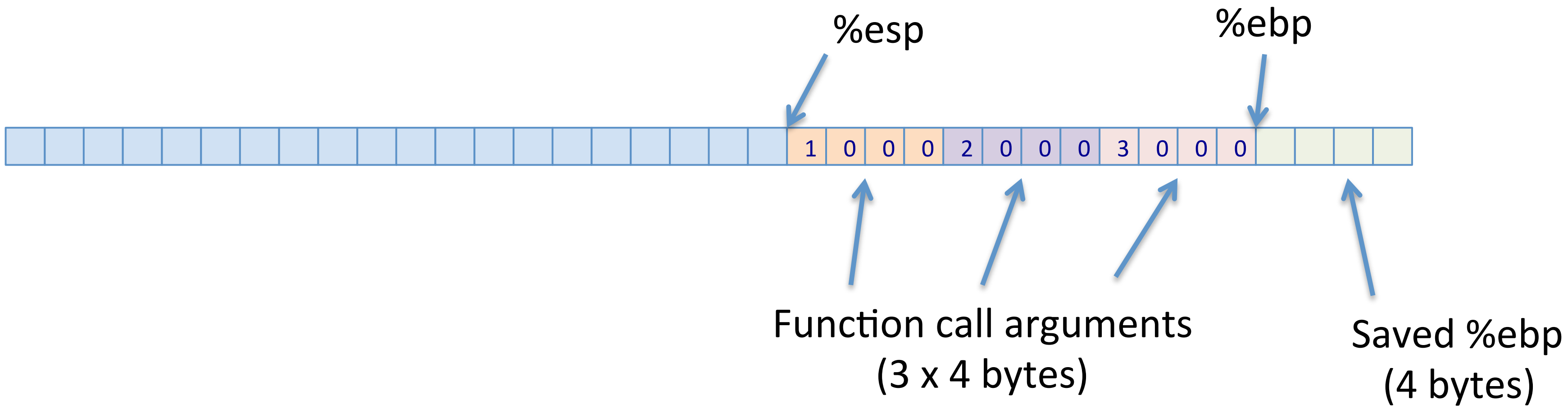
```
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $12, %esp
        movl    $3, 8(%esp)
        movl    $2, 4(%esp)
        movl    $1, (%esp)
        call    function
        leave
        ret
```

In Assembly: example1.s



```
function:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movb    $97, -5(%ebp)
        movb    $65, -15(%ebp)
        leave
        ret
```

Example1 Stack Frame Layout: main()



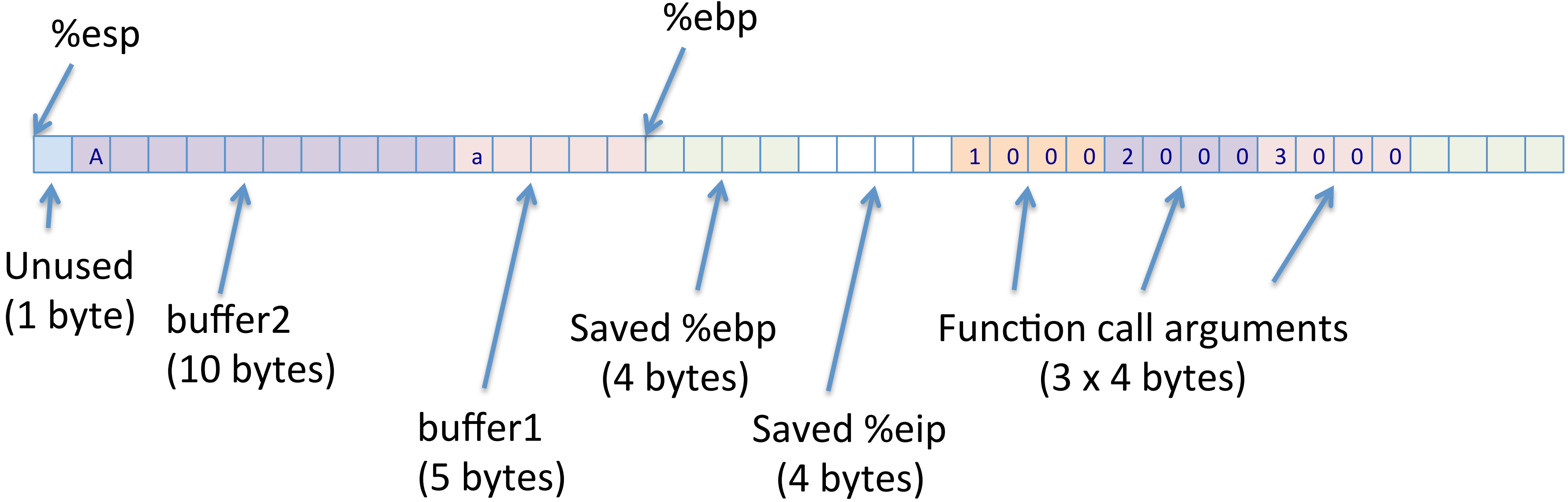
%esp

%ebp

Function call arguments
(3 x 4 bytes)

Saved %ebp
(4 bytes)

Example1 Stack Frame Layout: function()



L33t h4x0r sk1llz

Obligatory black background

Words spelled with numbers in place of letters

```
L33t h4x0r sk1llz
```

well, sort of ….

Reminder: Ethics

- Use your powers only for good!
- Seriously. Adventuring on other people's networks is no longer safe
  - See the recent case of Aaron Swartz
    - Co-inventor of RSS, co-founder of Reddit
    - Prosecuted for downloading millions of scholarly articles from JSTOR/MIT
    - Threatened with 35 years in jail
  - Contrast this to events in the not-so-distant past
    - Edward Tufte hacked AT&T phone network in 1962
    - AT&T politely asked him to stop. No penalties, no prosecution.
  - http://danwin.com/2013/01/edward-tufte-aaron-swartz-marvelously-different/

Software-based Attacks

- Vulnerabilities
  - Stack overflow
  - Heap overflow
  - Format string
  - Others (integer overflow, …)

- Exploits
  - Code Injection
    - Shellcode
    - Payload
  - Return Oriented Programming (ROP)

- Defenses
  - Language-based
  - Compiler tricks
  - System-level

Another Example (narnia0)

```
#include <stdio.h>
#include <stdlib.h>

int main(){
        long val=0x41414141;
        char buf[20];

        printf("Correct val's value from 0x41414141 -> 0xdeadbeef!\n");
        printf("Here is your chance: ");
        scanf("%24s",&buf);

        printf("buf: %s\n",buf);
        printf("val: 0x%08x\n",val);

        if(val==0xdeadbeef)
                system("/bin/sh");
        else {
                printf("WAY OFF!!!!\n");
                exit(1);
        }

        return 0;
}
```

Narnia examples from http://www.overthewire.org/wargames/narnia/ (License: GPL)

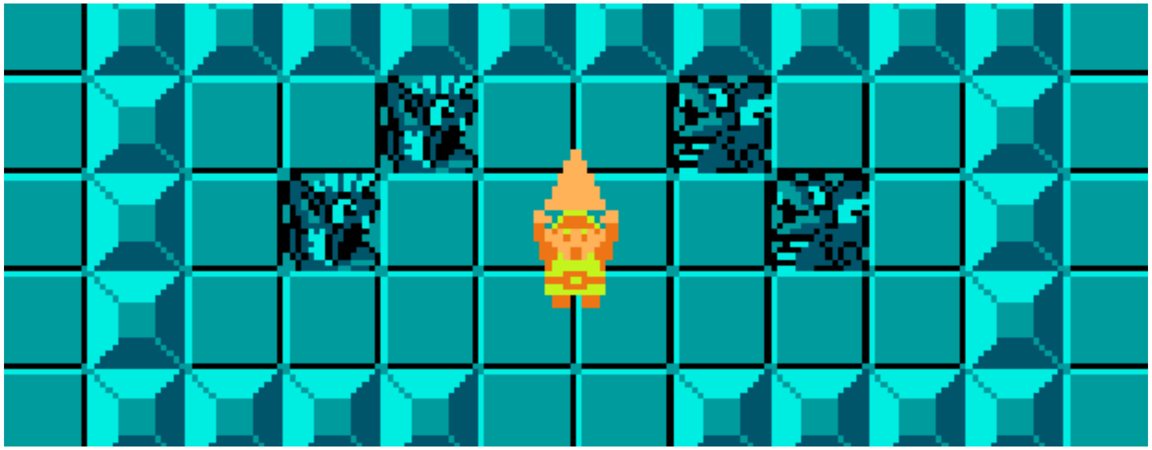
Capture the Flag (CTF)

Not *this* kind of CTF

More like this

Capture the Flag (CTF)

- Narnia is a “wargame” from overthewire.org
  - And by “wargame”, we mean “a series of puzzles”
  - The game is a series of “levels”. Solve the puzzle to progress to the next level.



  - For fun, see how far you can get

Interested in CTF?

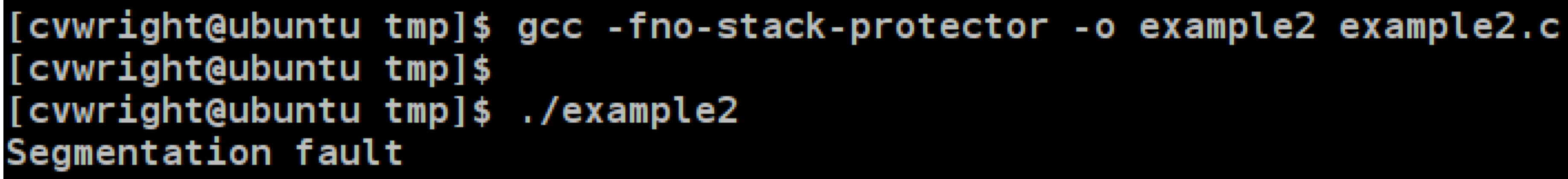
- We started a “hacking club” at PSU

- Weekly get-togethers
  - Where: FAB 145 (Intel Systems & Networking Lab)
  - When: Fridays at 1:30pm

- Mailing list: ctf@cs.pdx.edu
  - Sign up here if interested:
  - https://mailhost.cecs.pdx.edu/mailman/listinfo/ctf

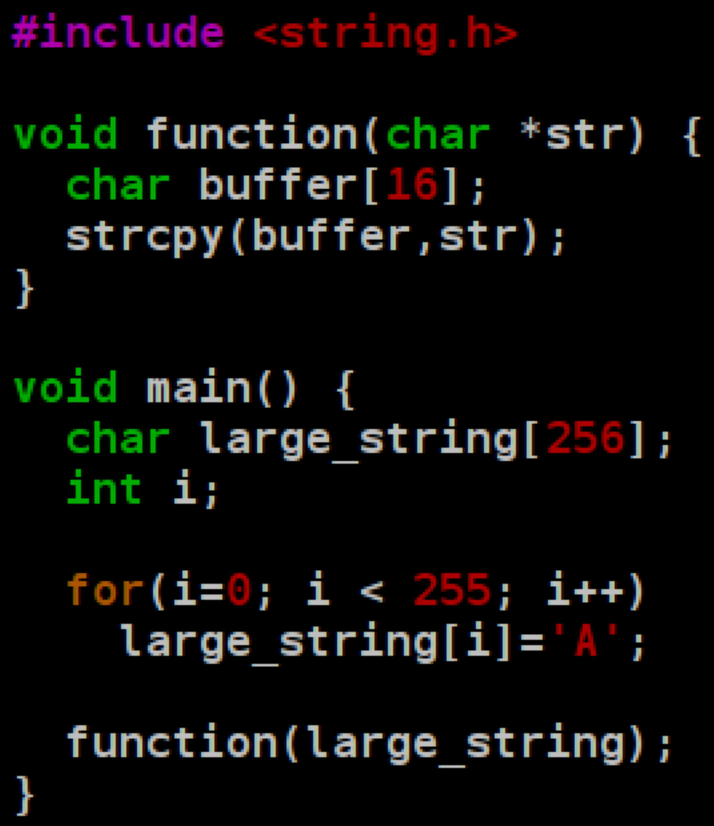
Example 2

Don’t you hate it when this happens?



```
[cvwright@ubuntu tmp]$ gcc -fno-stack-protector -o example2 example2.c
[cvwright@ubuntu tmp]$
[cvwright@ubuntu tmp]$ ./example2
Segmentation fault
```

Example 2

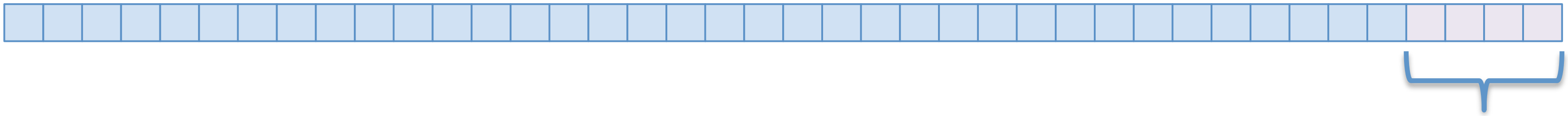


```
#include <string.h>

void function(char *str) {
  char buffer[16];
  strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;

  for(i=0; i < 255; i++)
    large_string[i]='A';

  function(large_string);
}
```

Example 2 Stack View

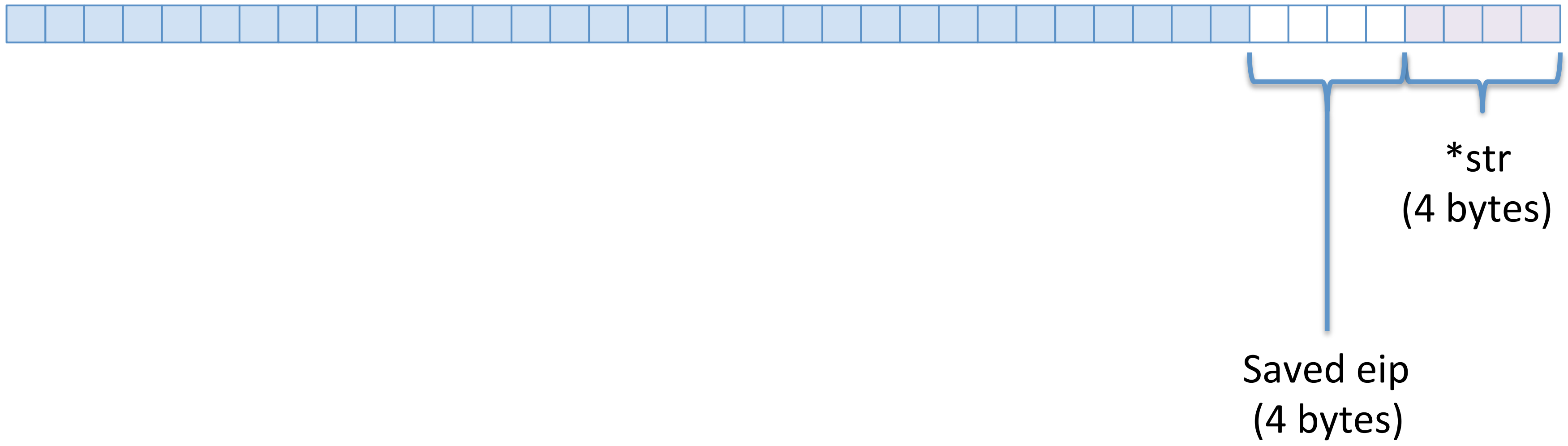
Can you fill this in for function() ?

Example 2 Stack View

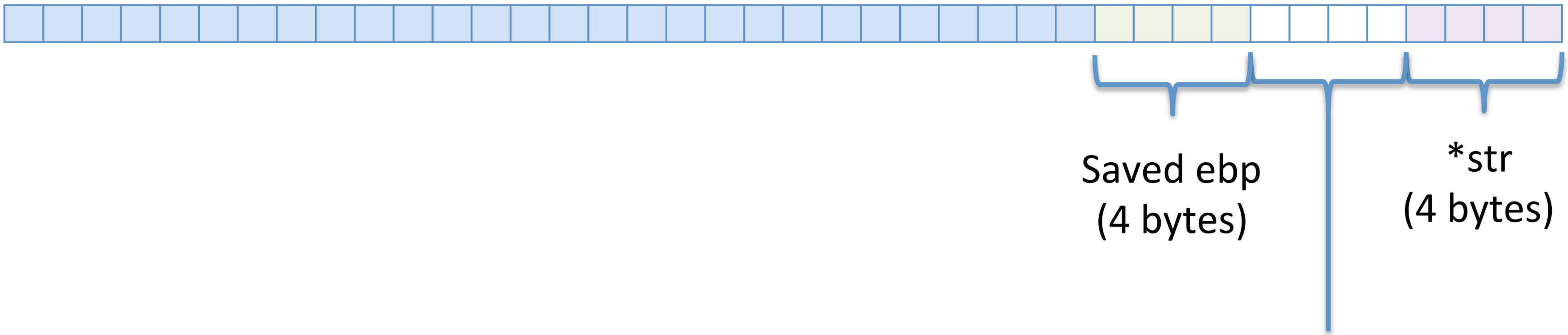


*str
(4 bytes)

Example 2 Stack View



Example 2 Stack View

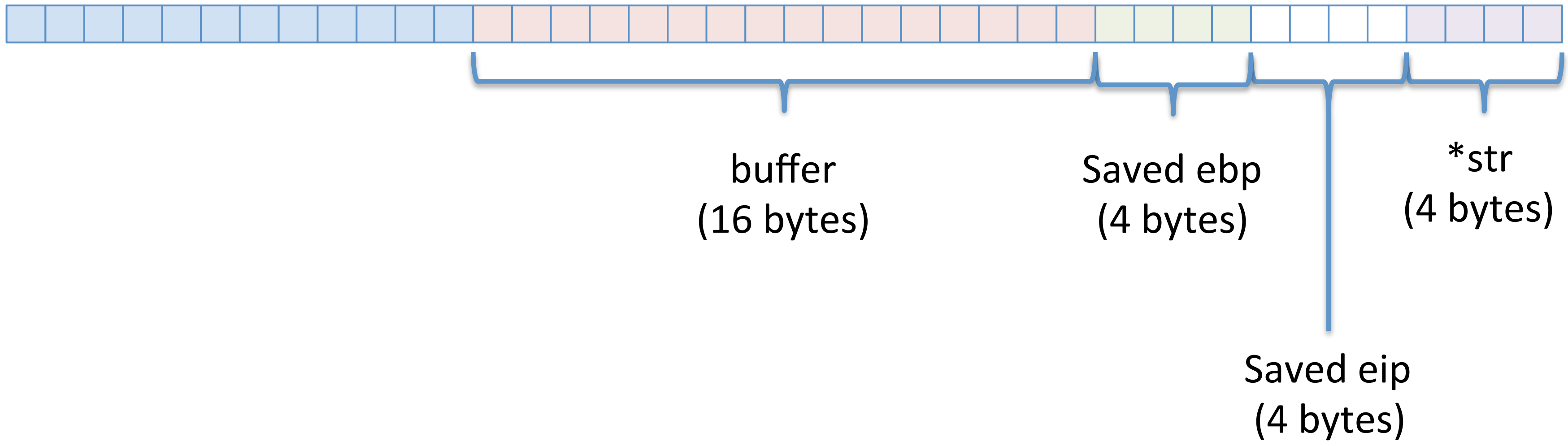


Saved ebp
(4 bytes)

*str
(4 bytes)

Saved eip
(4 bytes)

Example 2 Stack View



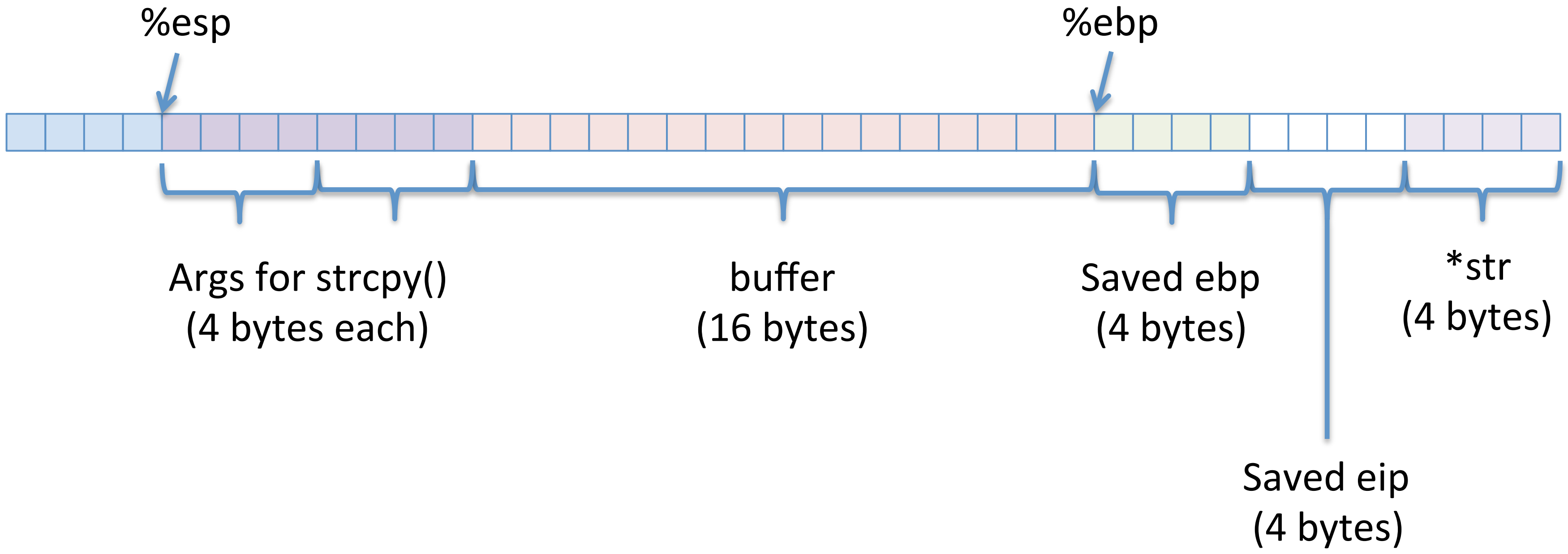
buffer
(16 bytes)

Saved ebp
(4 bytes)

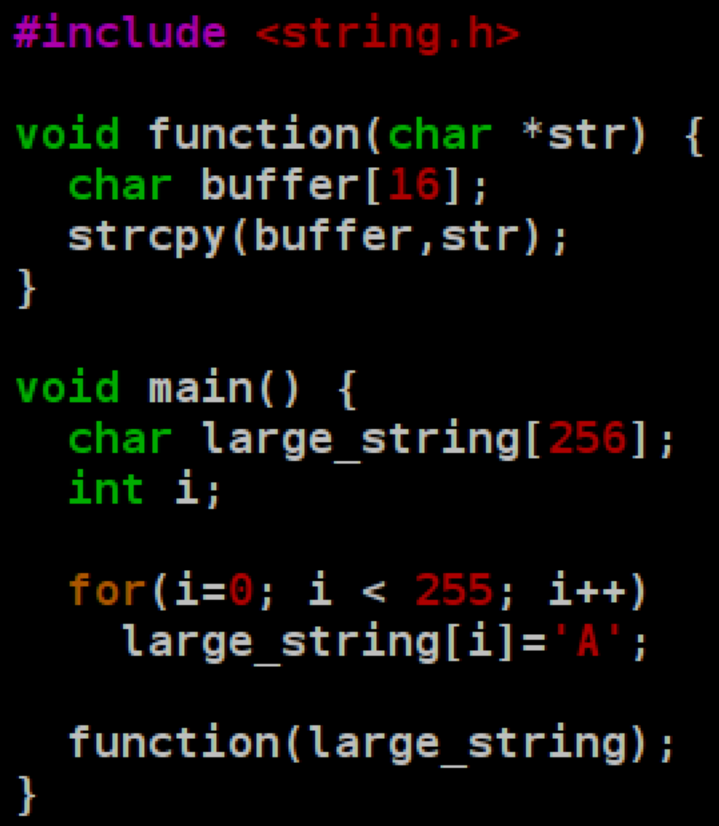
Saved eip
(4 bytes)

*str
(4 bytes)

Example 2 Stack View

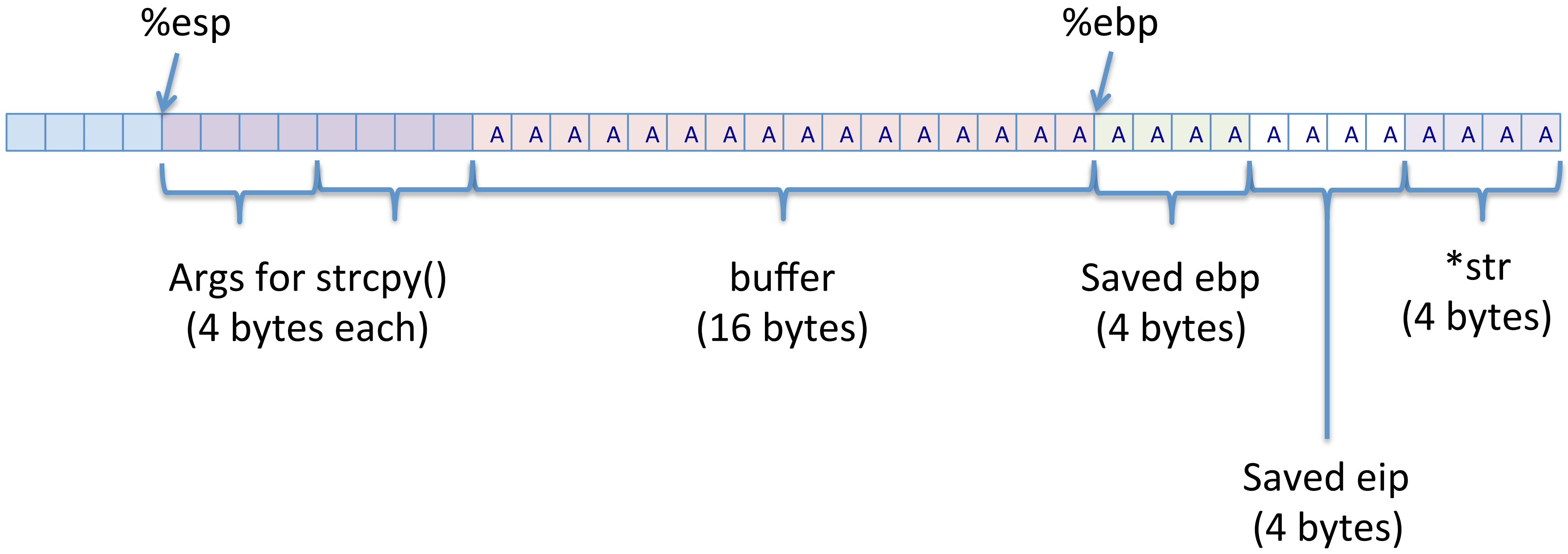


Example 2

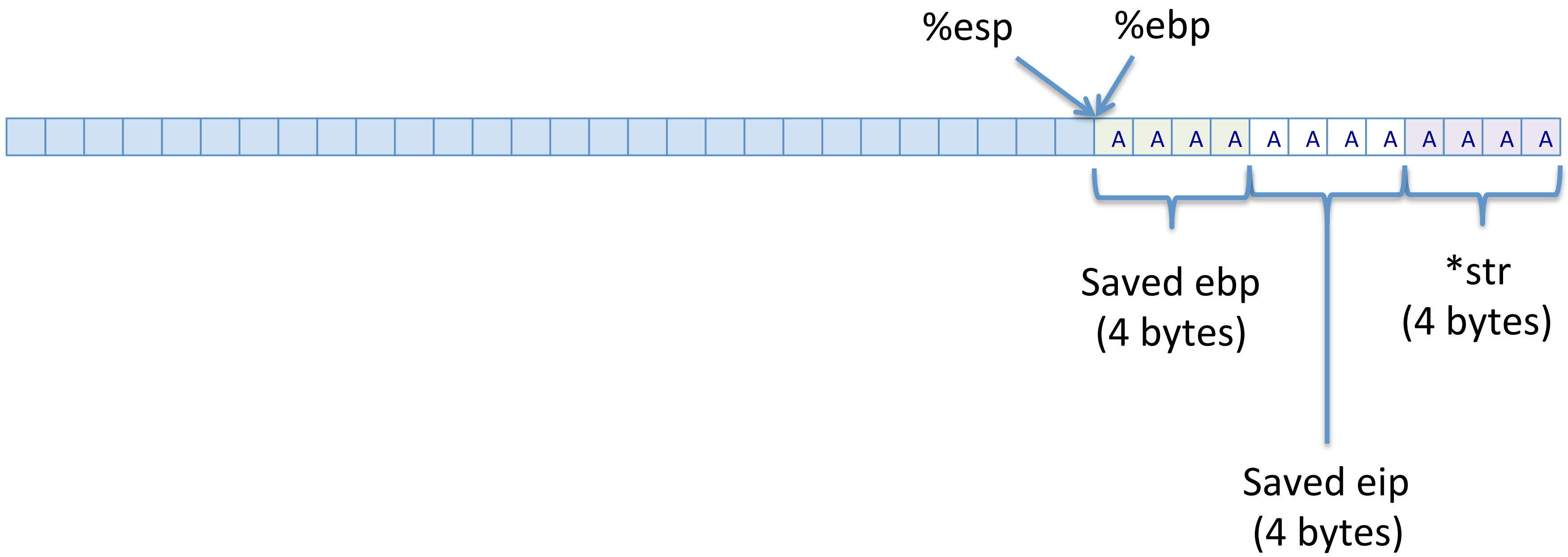


```
#include <string.h>

void function(char *str) {
  char buffer[16];
  strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;

  for(i=0; i < 255; i++)
    large_string[i]='A';

  function(large_string);
}
```

Example 2 – After strcpy()



Example 2 – **leave** (1)



%esp

%ebp

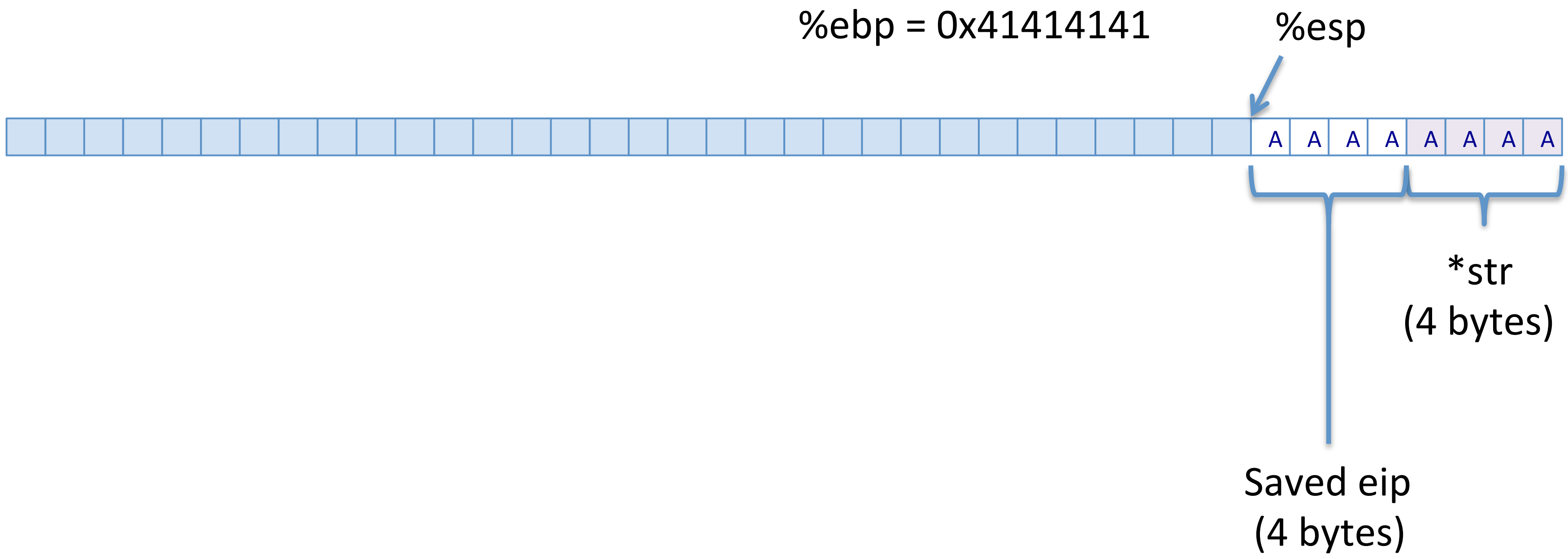
A A A A A A A A A A A A

Saved ebp
(4 bytes)

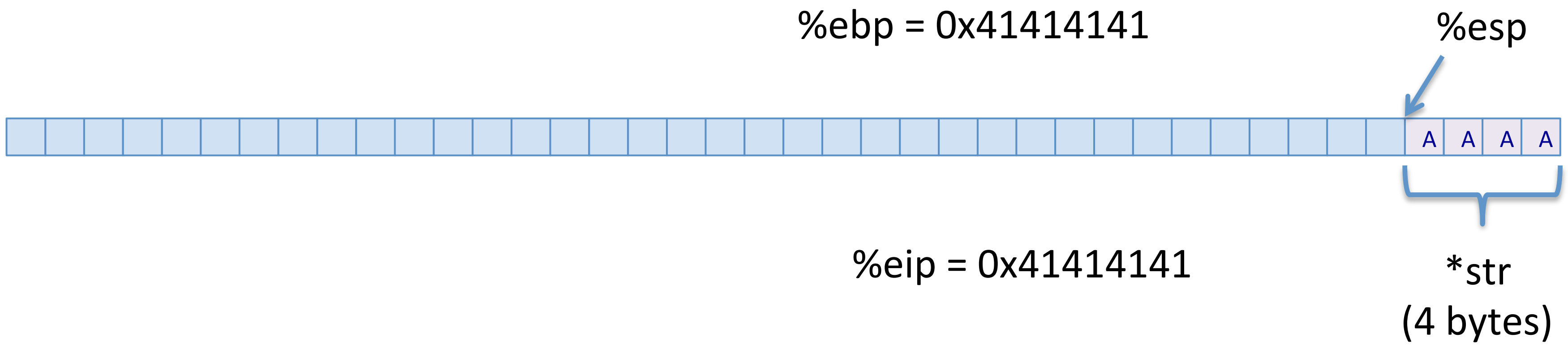
*str
(4 bytes)

Saved eip
(4 bytes)

Example 2 – **leave** (2)



Example 2 – **ret**



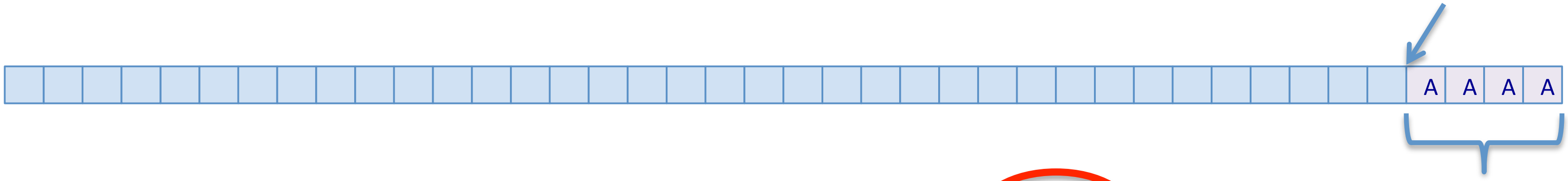
%ebp = 0x41414141

%eip = 0x41414141

Example 2 – **ret**

%ebp = 0x41414141

%esp



*str
(4 bytes)

%eip = 0x41414141

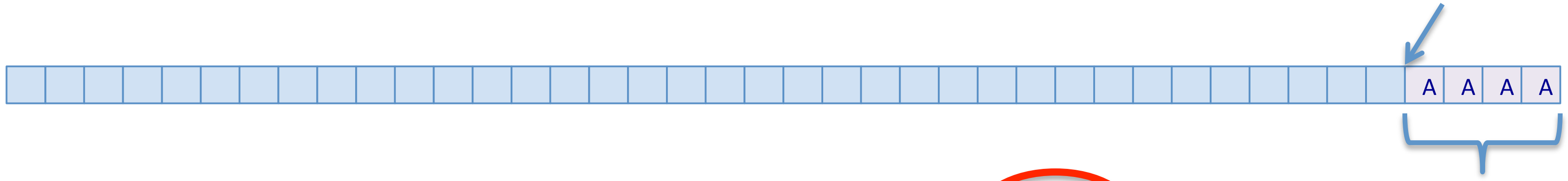
Hey! Wait just a minute!

Do we have an entry
in our page table for
virtual page # 0x41414 ?

Example 2 – **ret**

%ebp = 0x41414141

%esp



%eip = 0x41414141

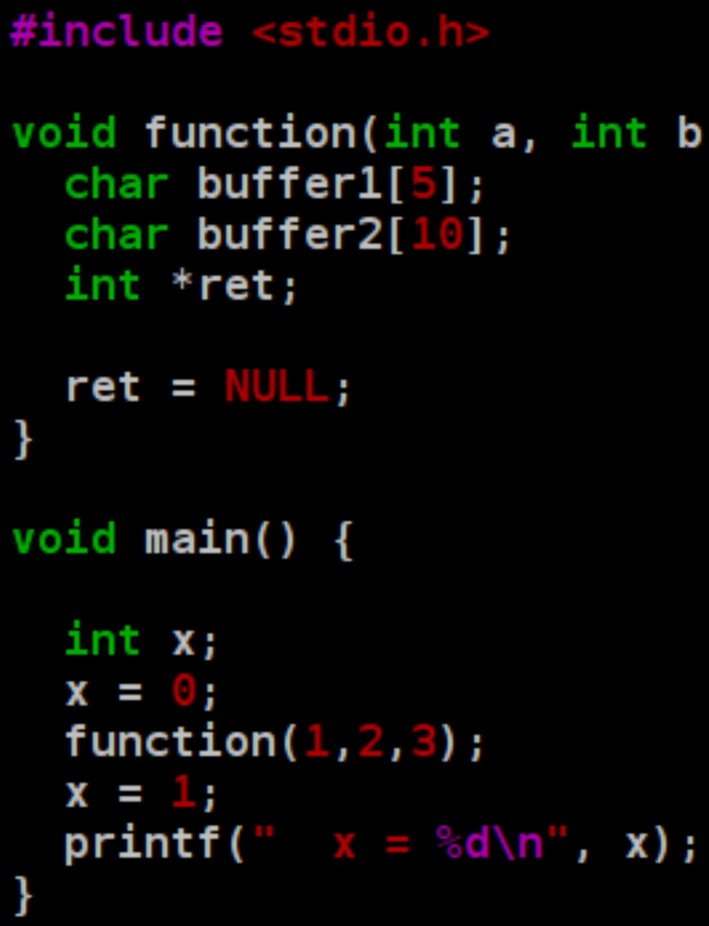
*str
(4 bytes)

Hey! Wait just a minute!

Do we have an entry
in our page table for
virtual page # 0x41414 ?

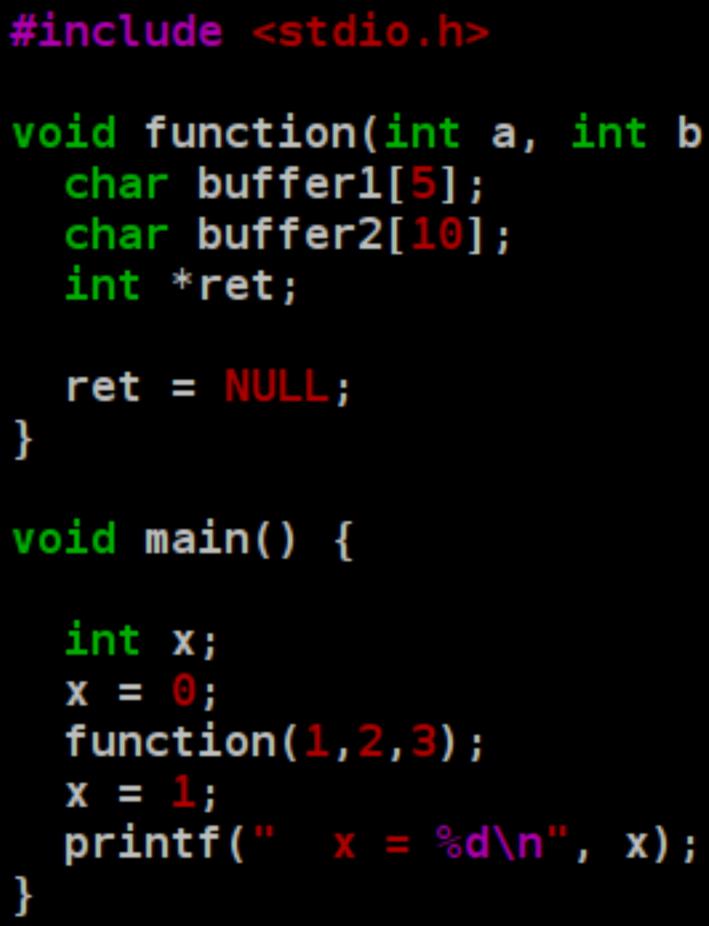
MMU says “No” → Segmentation Fault!

Example 3



```
#include <stdio.h>

void function(int a, int b, int c) {
  char buffer1[5];
  char buffer2[10];
  int *ret;

  ret = NULL;
}

void main() {

  int x;
  x = 0;
  function(1,2,3);
  x = 1;
  printf("  x = %d\n", x);
}
```

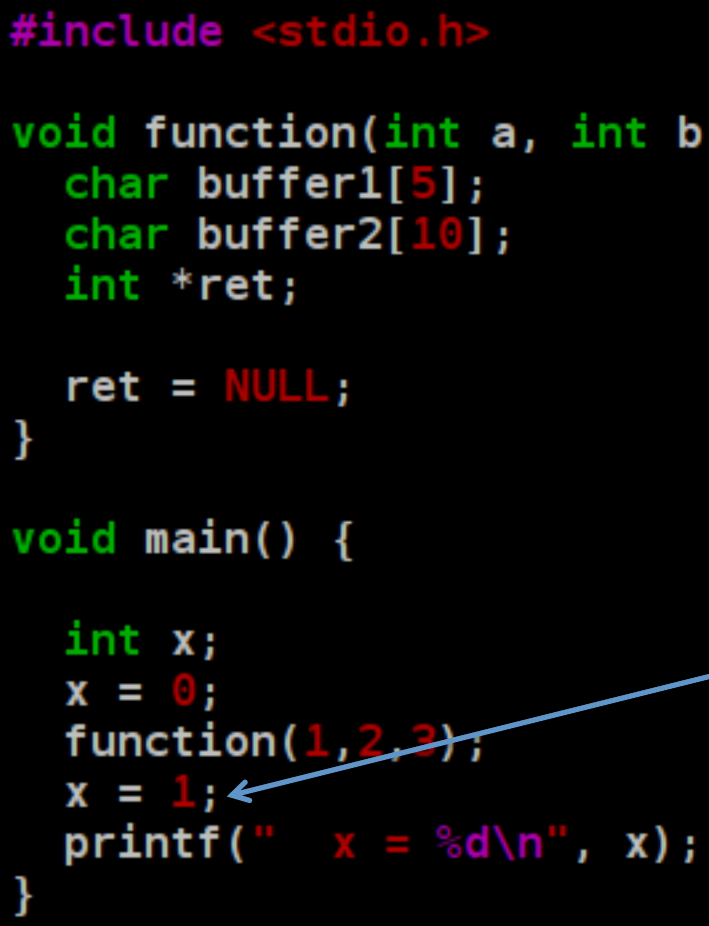
Example 3



```
#include <stdio.h>

void function(int a, int b, int c) {
  char buffer1[5];
  char buffer2[10];
  int *ret;

  ret = NULL;
}

void main() {

  int x;
  x = 0;
  function(1,2,3);
  x = 1;
  printf("  x = %d\n", x);
}
```

Goal: Modify function() so that the program prints “x = 0” instead of “x = 1”

Example 3

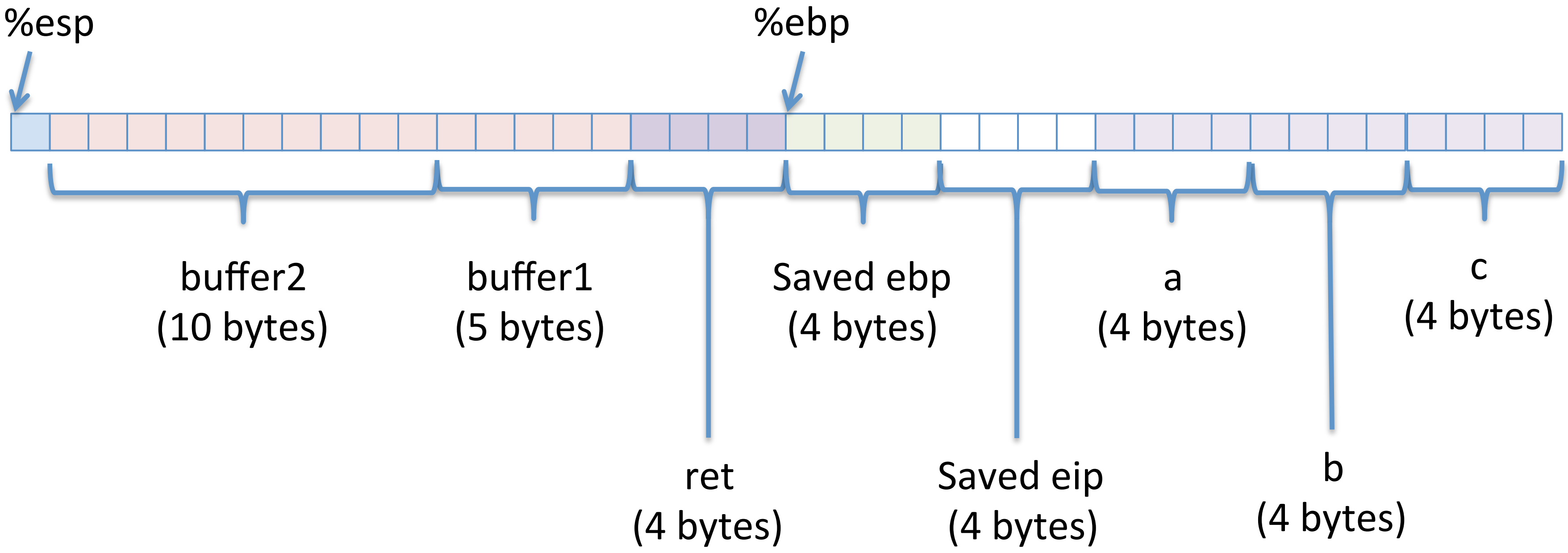


```
#include <stdio.h>

void function(int a, int b, int c) {
  char buffer1[5];
  char buffer2[10];
  int *ret;

  ret = NULL;
}

void main() {

  int x;
  x = 0;
  function(1,2,3);
  x = 1;
  printf("  x = %d\n", x);
}
```

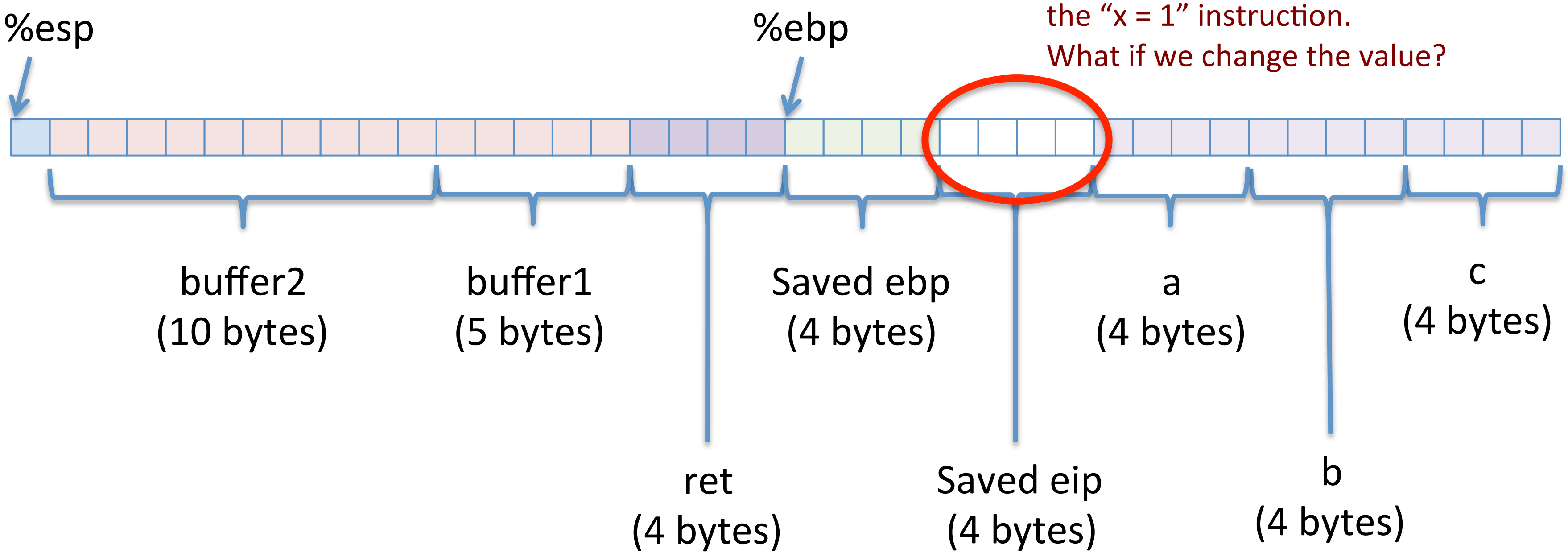
Goal: Modify function() so that the program prints “x = 0” instead of “x = 1”

Idea: If we can skip this line,
then we’ll get the output that we want

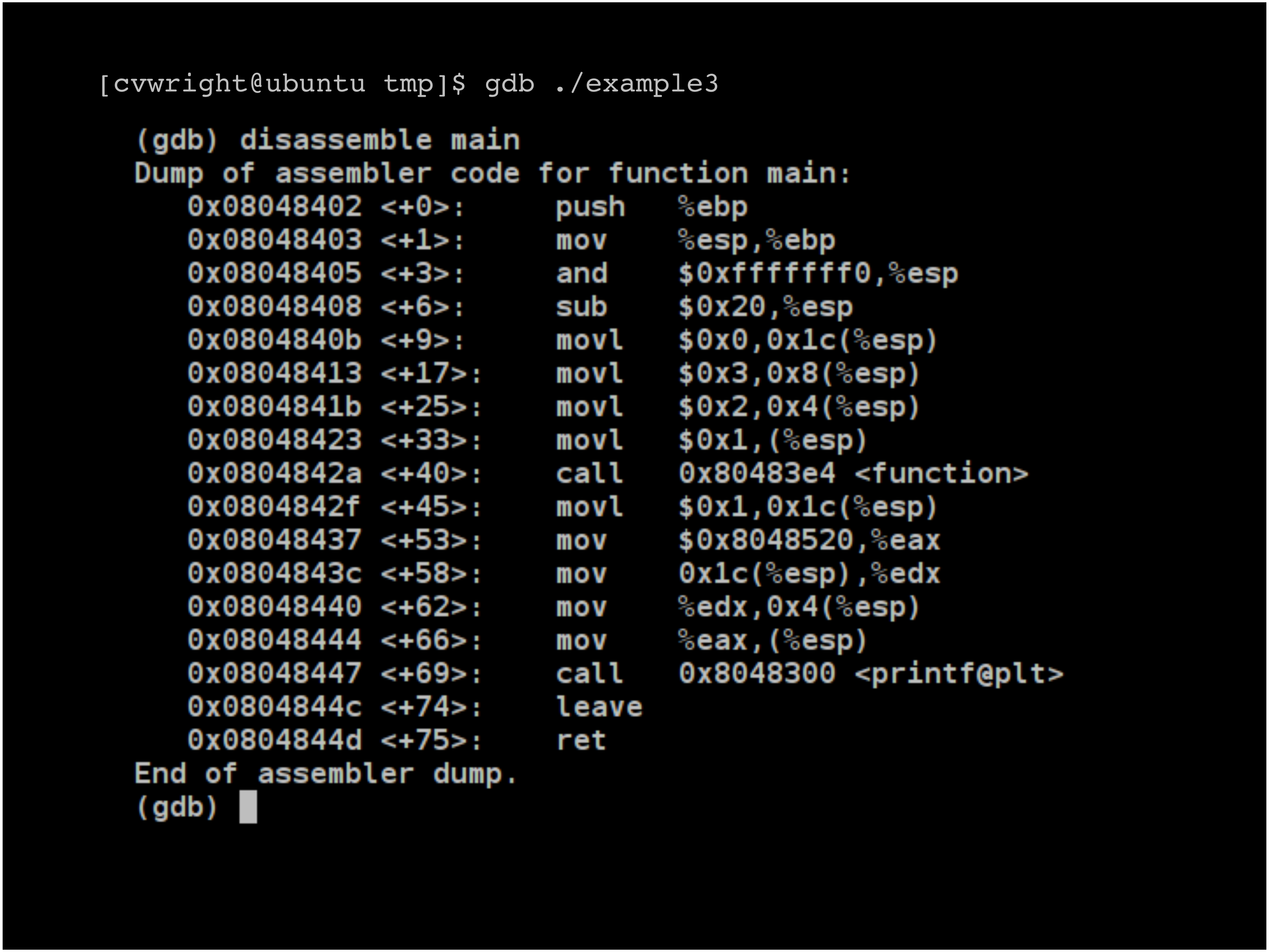
Example 3 Stack View: function()



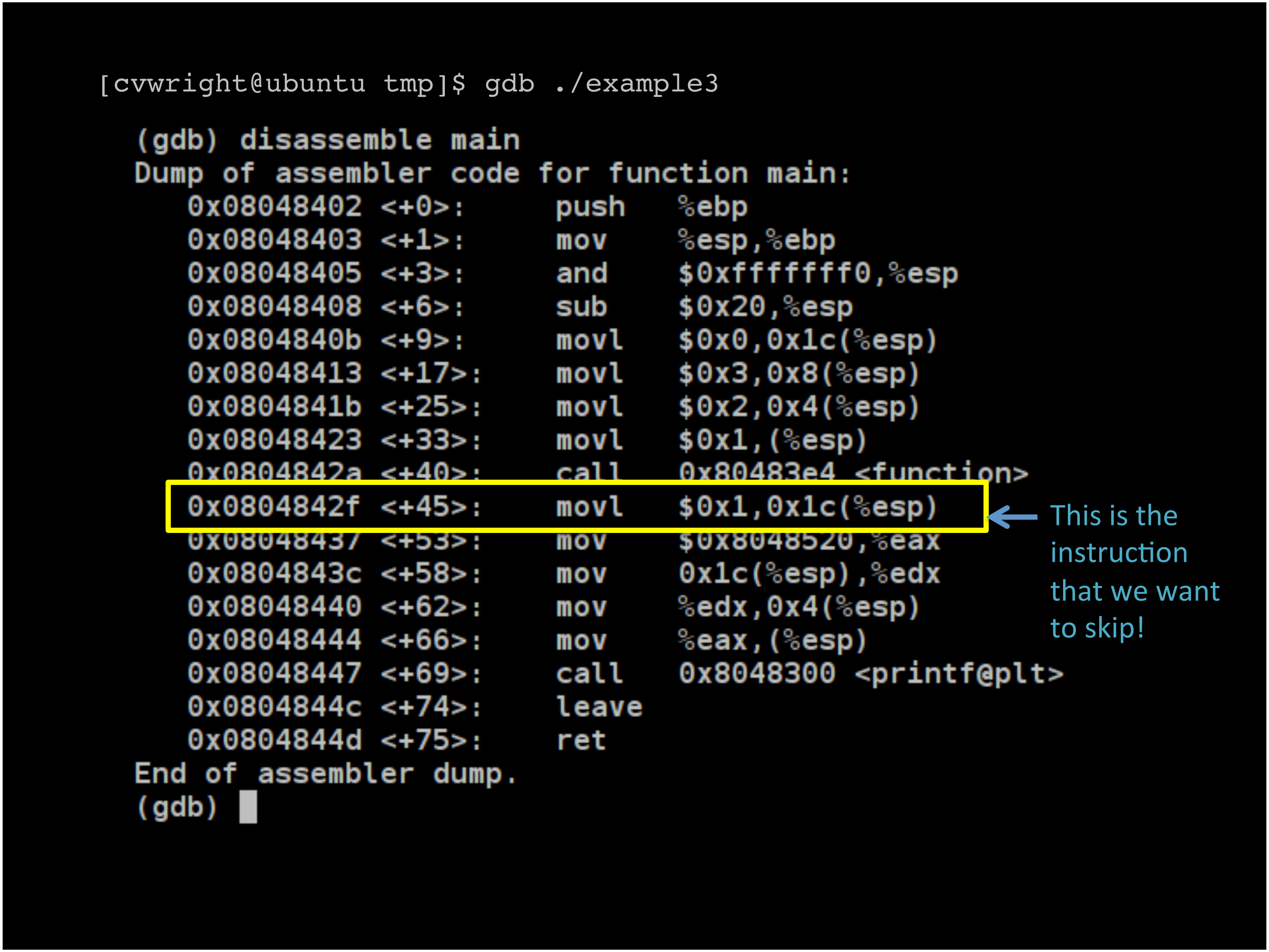
Example 3 Stack View: function()



This contains the address of the “x = 1” instruction.
What if we change the value?

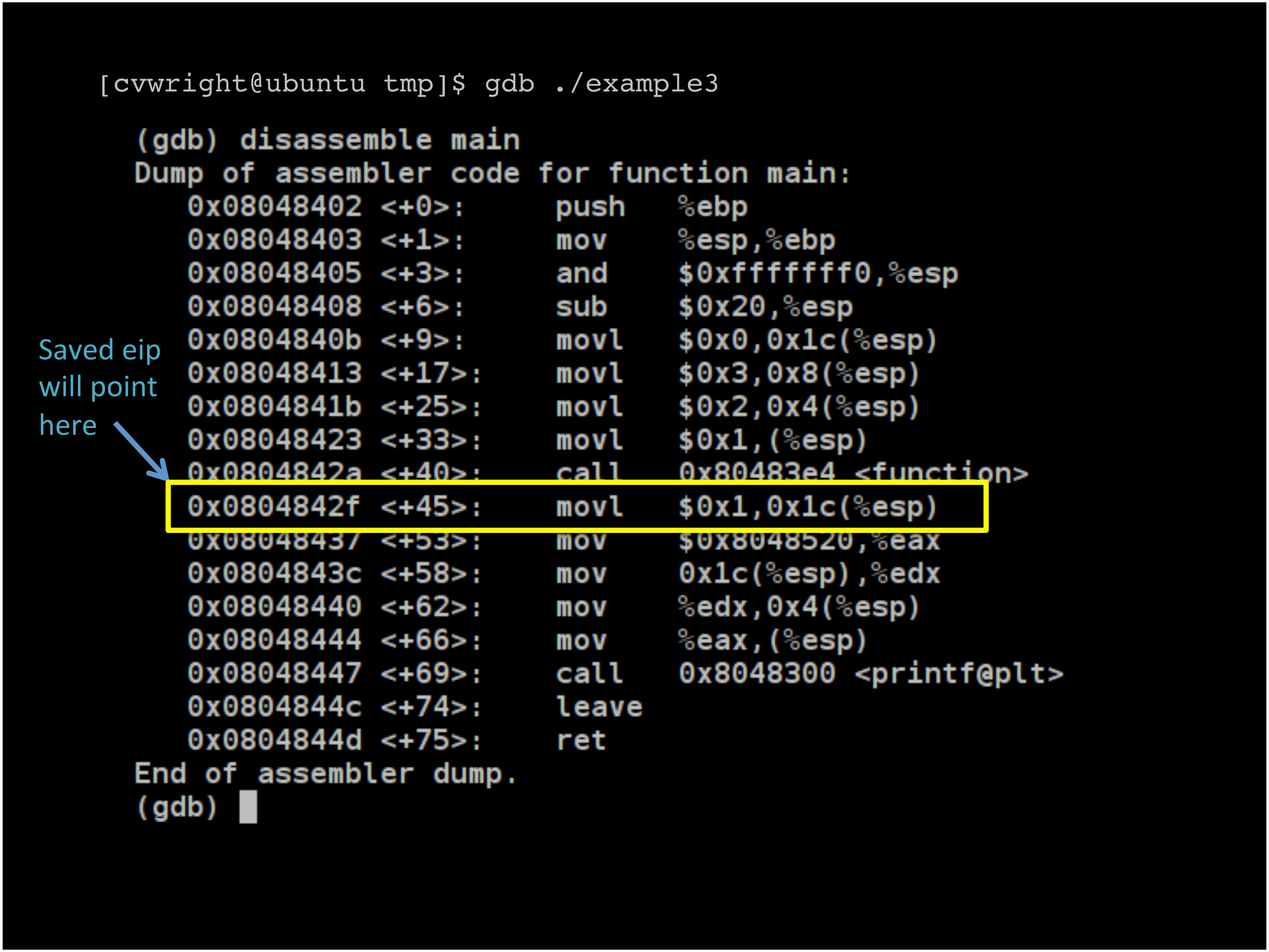


```
[cvwright@ubuntu tmp]$ gdb ./example3

  (gdb) disassemble main
  Dump of assembler code for function main:
     0x08048402 <+0>:     push   %ebp
     0x08048403 <+1>:     mov    %esp,%ebp
     0x08048405 <+3>:     and    $0xfffffff0,%esp
     0x08048408 <+6>:     sub    $0x20,%esp
     0x0804840b <+9>:     movl   $0x0,0x1c(%esp)
     0x08048413 <+17>:    movl   $0x3,0x8(%esp)
     0x0804841b <+25>:    movl   $0x2,0x4(%esp)
     0x08048423 <+33>:    movl   $0x1,(%esp)
     0x0804842a <+40>:    call   0x80483e4 <function>
     0x0804842f <+45>:    movl   $0x1,0x1c(%esp)
     0x08048437 <+53>:    mov    $0x8048520,%eax
     0x0804843c <+58>:    mov    0x1c(%esp),%edx
     0x08048440 <+62>:    mov    %edx,0x4(%esp)
     0x08048444 <+66>:    mov    %eax,(%esp)
     0x08048447 <+69>:    call   0x8048300 <printf@plt>
     0x0804844c <+74>:    leave
     0x0804844d <+75>:    ret
  End of assembler dump.
  (gdb) █
```



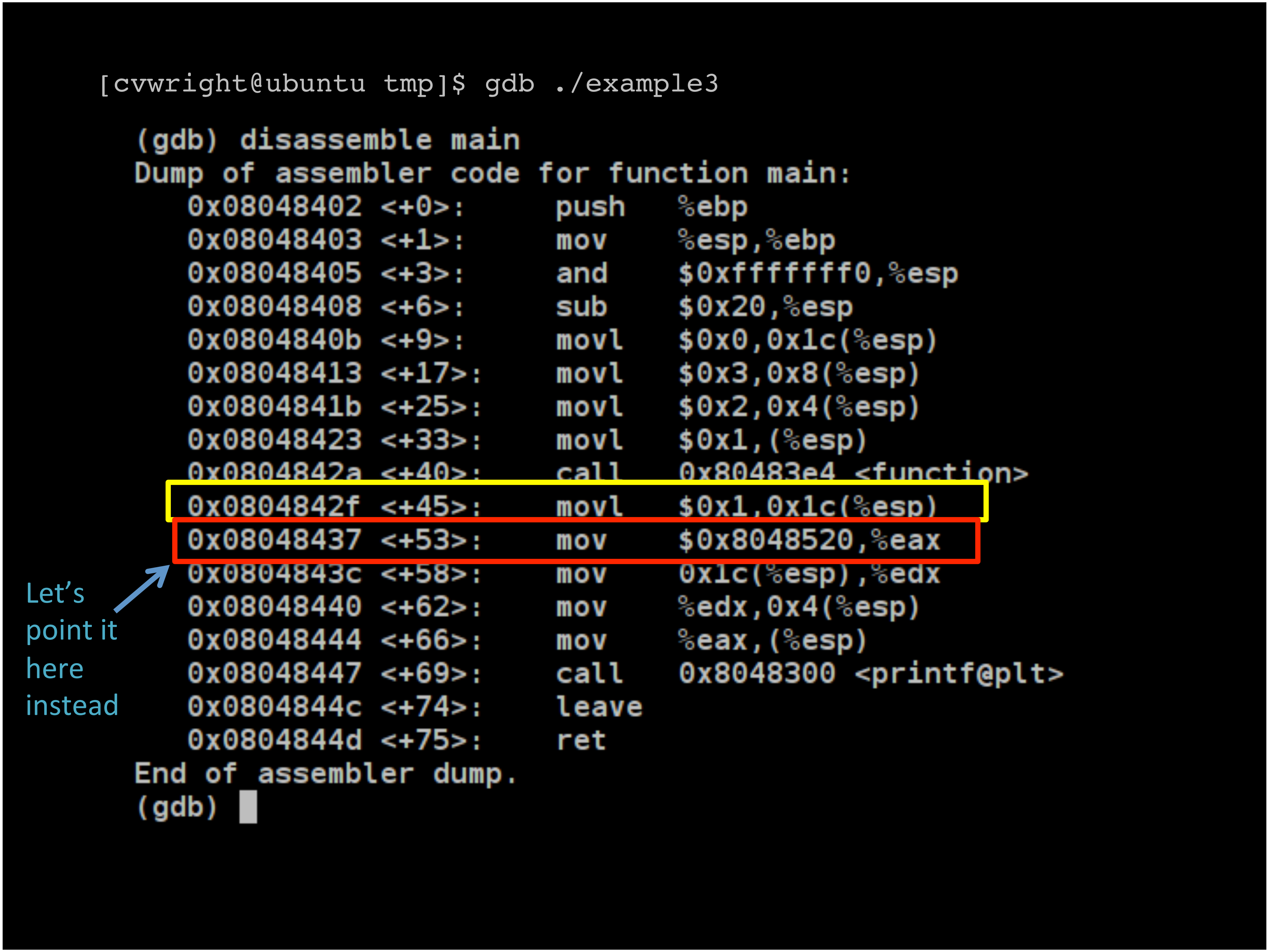
```
[cvwright@ubuntu tmp]$ gdb ./example3

  (gdb) disassemble main
  Dump of assembler code for function main:
     0x08048402 <+0>:     push   %ebp
     0x08048403 <+1>:     mov    %esp,%ebp
     0x08048405 <+3>:     and    $0xfffffff0,%esp
     0x08048408 <+6>:     sub    $0x20,%esp
     0x0804840b <+9>:     movl   $0x0,0x1c(%esp)
     0x08048413 <+17>:    movl   $0x3,0x8(%esp)
     0x0804841b <+25>:    movl   $0x2,0x4(%esp)
     0x08048423 <+33>:    movl   $0x1,(%esp)
     0x0804842a <+40>:    call   0x80483e4 <function>
     0x0804842f <+45>:    movl   $0x1,0x1c(%esp)
     0x08048437 <+53>:    mov    $0x8048520,%eax
     0x0804843c <+58>:    mov    0x1c(%esp),%edx
     0x08048440 <+62>:    mov    %edx,0x4(%esp)
     0x08048444 <+66>:    mov    %eax,(%esp)
     0x08048447 <+69>:    call   0x8048300 <printf@plt>
     0x0804844c <+74>:    leave
     0x0804844d <+75>:    ret
  End of assembler dump.
  (gdb) 
```

This is the instruction that we want to skip!



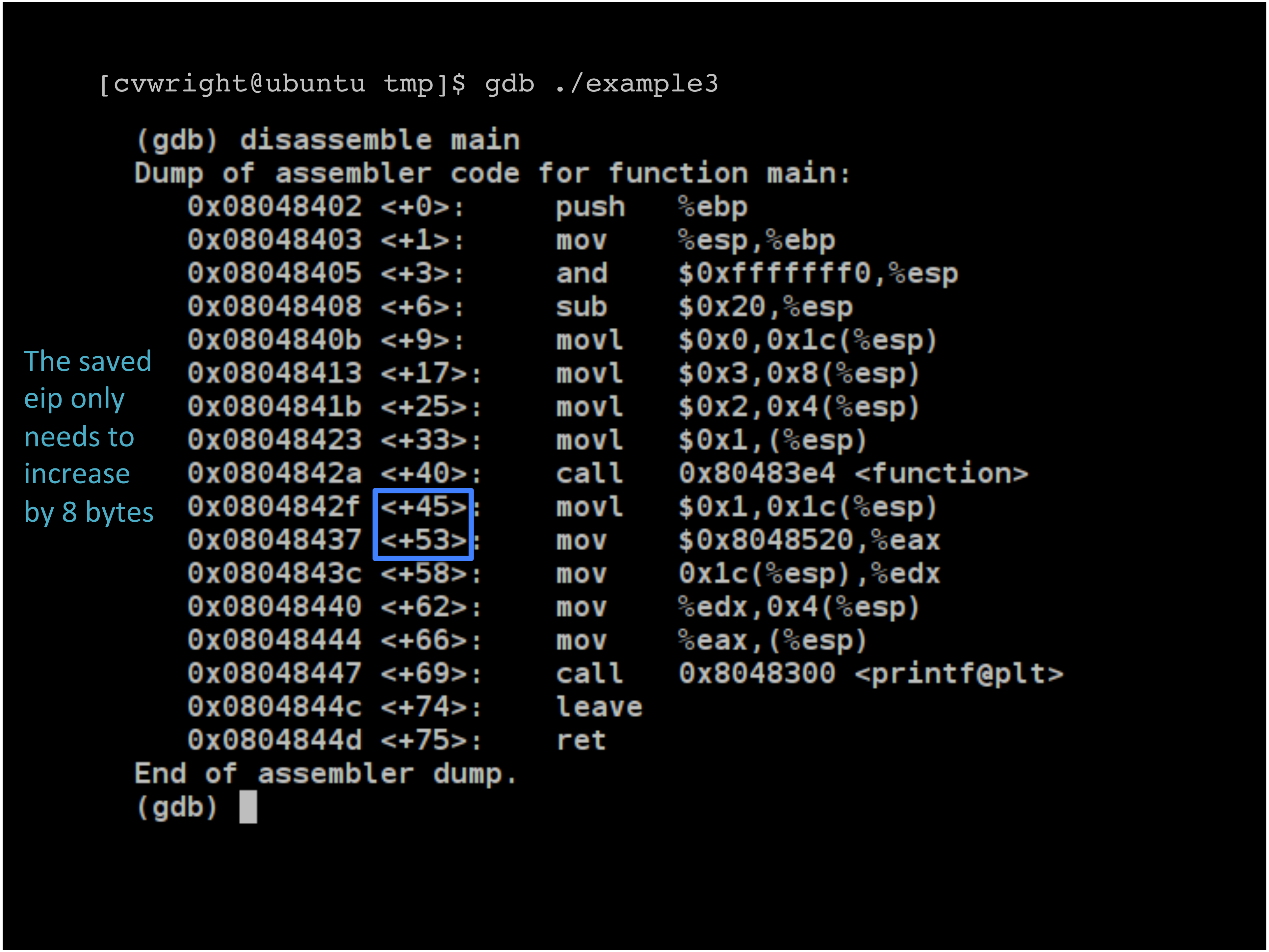
```
[cvwright@ubuntu tmp]$ gdb ./example3

  (gdb) disassemble main
  Dump of assembler code for function main:
     0x08048402 <+0>:     push   %ebp
     0x08048403 <+1>:     mov    %esp,%ebp
     0x08048405 <+3>:     and    $0xfffffff0,%esp
     0x08048408 <+6>:     sub    $0x20,%esp
     0x0804840b <+9>:     movl   $0x0,0x1c(%esp)
     0x08048413 <+17>:    movl   $0x3,0x8(%esp)
     0x0804841b <+25>:    movl   $0x2,0x4(%esp)
     0x08048423 <+33>:    movl   $0x1,(%esp)
     0x0804842a <+40>:    call   0x80483e4 <function>
     0x0804842f <+45>:    movl   $0x1,0x1c(%esp)
     0x08048437 <+53>:    mov    $0x8048520,%eax
     0x0804843c <+58>:    mov    0x1c(%esp),%edx
     0x08048440 <+62>:    mov    %edx,0x4(%esp)
     0x08048444 <+66>:    mov    %eax,(%esp)
     0x08048447 <+69>:    call   0x8048300 <printf@plt>
     0x0804844c <+74>:    leave
     0x0804844d <+75>:    ret
  End of assembler dump.
  (gdb)
```

Saved eip will point here



```
[cvwright@ubuntu tmp]$ gdb ./example3

  (gdb) disassemble main
  Dump of assembler code for function main:
     0x08048402 <+0>:     push   %ebp
     0x08048403 <+1>:     mov    %esp,%ebp
     0x08048405 <+3>:     and    $0xfffffff0,%esp
     0x08048408 <+6>:     sub    $0x20,%esp
     0x0804840b <+9>:     movl   $0x0,0x1c(%esp)
     0x08048413 <+17>:    movl   $0x3,0x8(%esp)
     0x0804841b <+25>:    movl   $0x2,0x4(%esp)
     0x08048423 <+33>:    movl   $0x1,(%esp)
     0x0804842a <+40>:    call   0x80483e4 <function>
     0x0804842f <+45>:    movl   $0x1,0x1c(%esp)
     0x08048437 <+53>:    mov    $0x8048520,%eax
     0x0804843c <+58>:    mov    0x1c(%esp),%edx
     0x08048440 <+62>:    mov    %edx,0x4(%esp)
     0x08048444 <+66>:    mov    %eax,(%esp)
     0x08048447 <+69>:    call   0x8048300 <printf@plt>
     0x0804844c <+74>:    leave
     0x0804844d <+75>:    ret
  End of assembler dump.
  (gdb) █
```

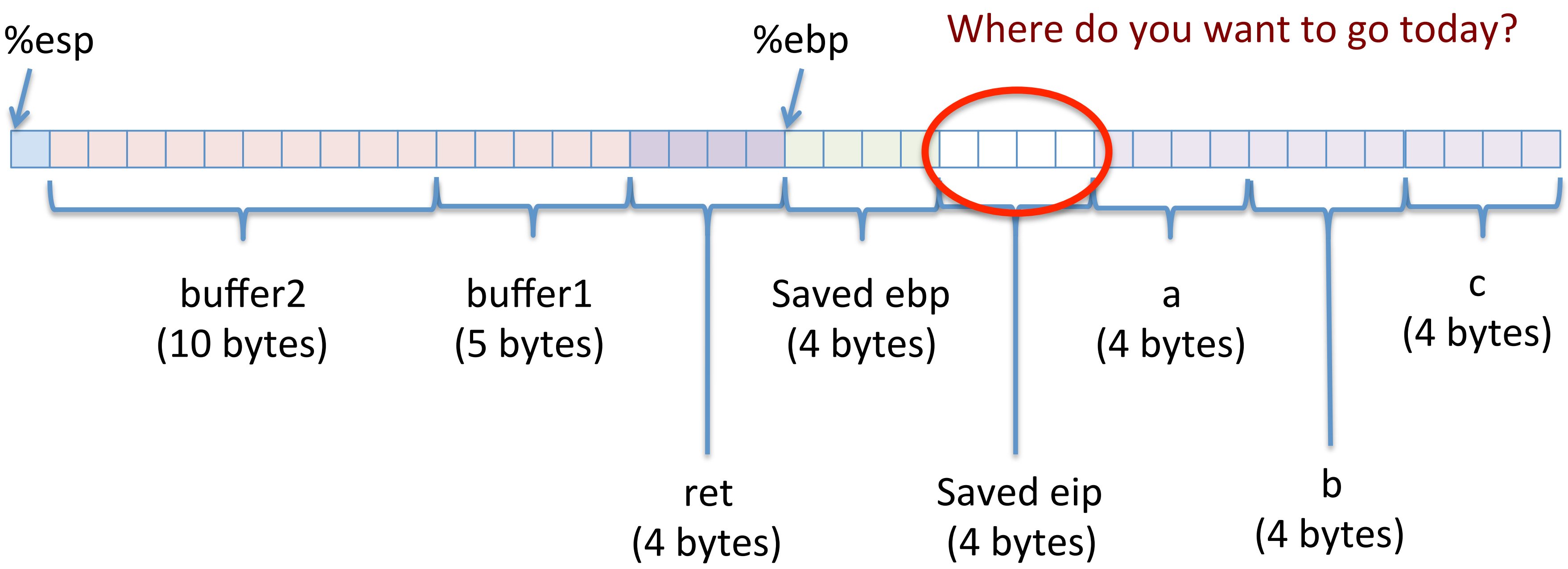
Let’s point it here instead



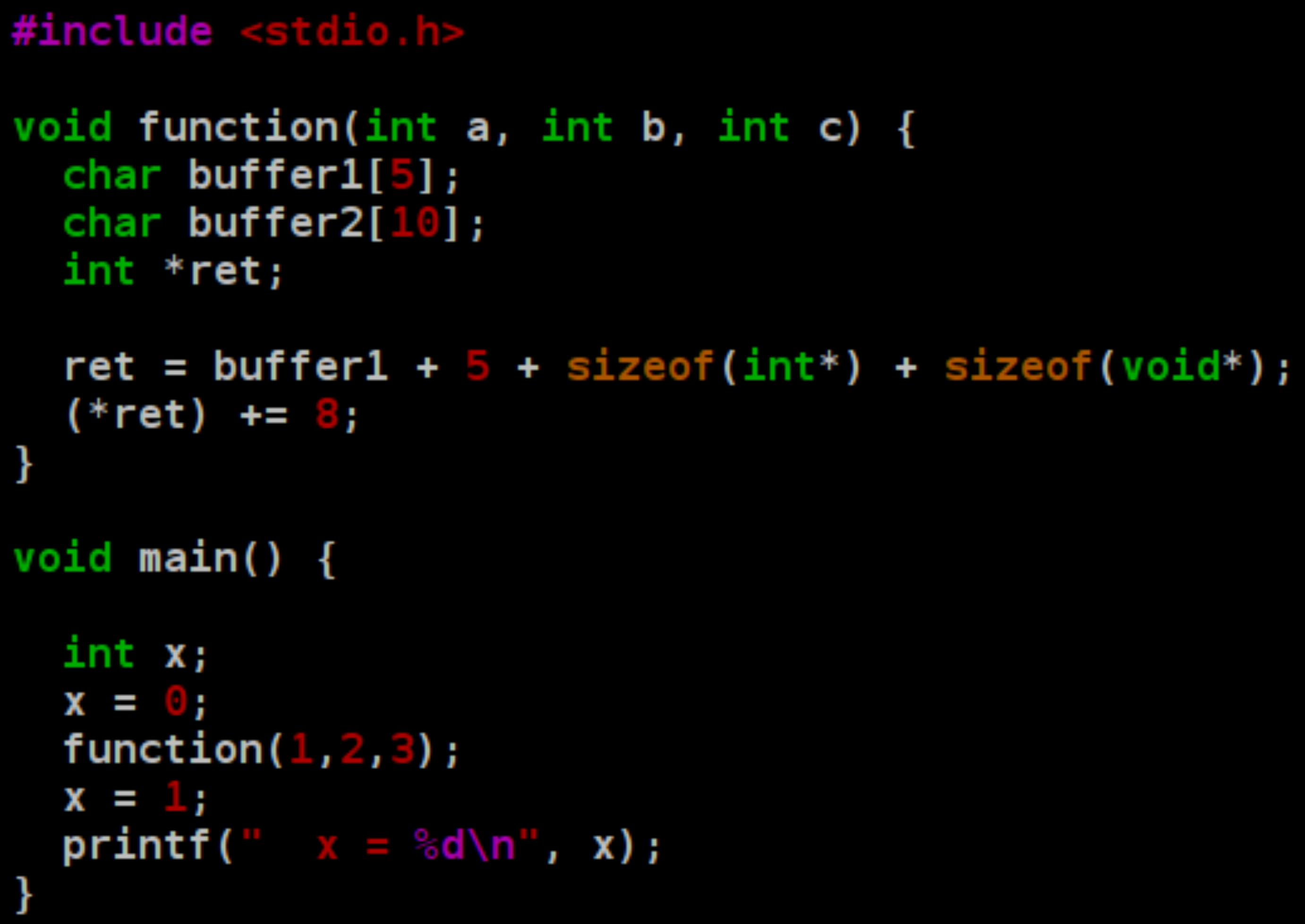
```
[cvwright@ubuntu tmp]$ gdb ./example3

  (gdb) disassemble main
  Dump of assembler code for function main:
     0x08048402 <+0>:     push   %ebp
     0x08048403 <+1>:     mov    %esp,%ebp
     0x08048405 <+3>:     and    $0xfffffff0,%esp
     0x08048408 <+6>:     sub    $0x20,%esp
     0x0804840b <+9>:     movl   $0x0,0x1c(%esp)
     0x08048413 <+17>:    movl   $0x3,0x8(%esp)
     0x0804841b <+25>:    movl   $0x2,0x4(%esp)
     0x08048423 <+33>:    movl   $0x1,(%esp)
     0x0804842a <+40>:    call   0x80483e4 <function>
     0x0804842f <+45>:    movl   $0x1,0x1c(%esp)
     0x08048437 <+53>:    mov    $0x8048520,%eax
     0x0804843c <+58>:    mov    0x1c(%esp),%edx
     0x08048440 <+62>:    mov    %edx,0x4(%esp)
     0x08048444 <+66>:    mov    %eax,(%esp)
     0x08048447 <+69>:    call   0x8048300 <printf@plt>
     0x0804844c <+74>:    leave
     0x0804844d <+75>:    ret
  End of assembler dump.
  (gdb) █
```

The saved eip only needs to increase by 8 bytes

Example 3 Stack View: function()

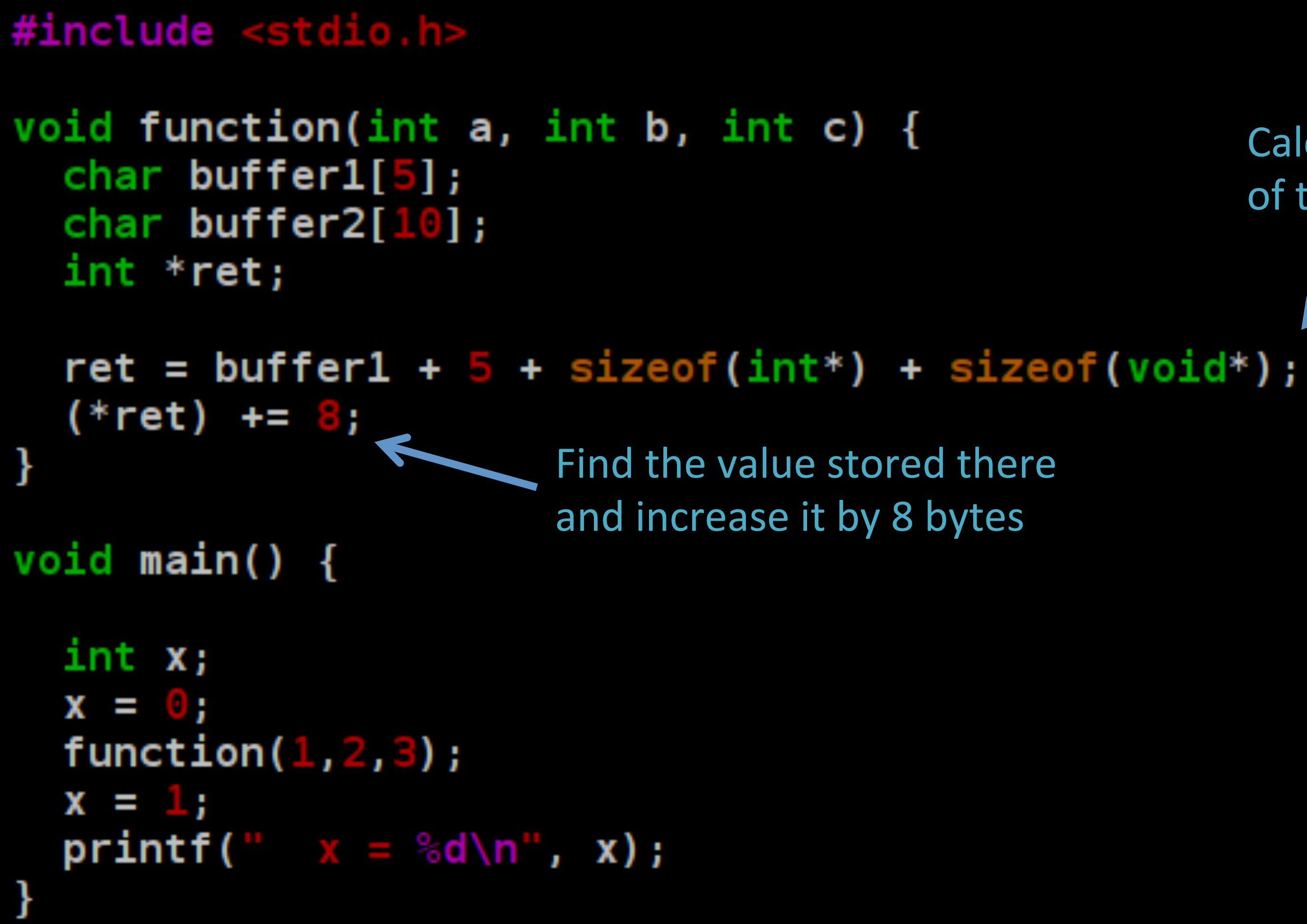


Example 3



```
#include <stdio.h>

void function(int a, int b, int c) {
  char buffer1[5];
  char buffer2[10];
  int *ret;

  ret = buffer1 + 5 + sizeof(int*) + sizeof(void*);
  (*ret) += 8;
}

void main() {

  int x;
  x = 0;
  function(1,2,3);
  x = 1;
  printf("  x = %d\n", x);
}
```

Example 3

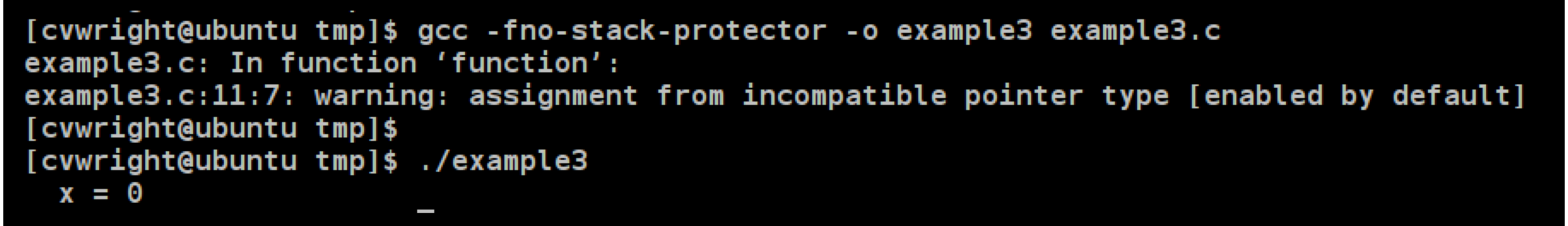


```
#include <stdio.h>

void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
   int *ret;

   ret = buffer1 + 5 + sizeof(int*) + sizeof(void*);
   (*ret) += 8;
}

void main() {

   int x;
   x = 0;
   function(1,2,3);
   x = 1;
   printf("  x = %d\n", x);
}
```

Calculate address of the saved eip

Find the value stored there and increase it by 8 bytes

Example3 Success!



```
[cvwright@ubuntu tmp]$ gcc -fno-stack-protector -o example3 example3.c
example3.c: In function ‘function’:
example3.c:11:7: warning: assignment from incompatible pointer type [enabled by default]
[cvwright@ubuntu tmp]$
[cvwright@ubuntu tmp]$ ./example3
  x = 0
```

Status for Today

- We’ve seen how programs can misbehave
  - Example 2: Stack buffer overflow
    → Saved %eip got overwritten
    → Segmentation fault
  - Example 3: Carefully modified stack
    → Changed control flow

Status for Today

- Haven't seen any real attacks yet
  - Programs just caused trouble for themselves

- Next up:
  - Stack overflow vulnerabilities
  - Code injection exploits
    - Shellcode
    - Payload