

CS 143A

Name (Print):

Fall 2019

Seat:

SEAT

Midterm

Left person:

11/13/2019

Right person:

Time Limit: 11:00pm - 11:50pm

-
- Don't forget to write your name on this exam.
 - This is an open book, open notes exam. But no online or in-class chatting.
 - Ask us if something is confusing in the questions.
 - **Organize your work**, in a reasonably neat and coherent way, in the space provided. Work scattered all over the page without a clear ordering will receive very little credit.
 - **Mysterious or unsupported answers will not receive full credit.** A correct answer, unsupported by explanation will receive no credit; an incorrect answer supported by substantially correct explanations might still receive partial credit.
 - If you need more space, use the back of the pages; clearly indicate when you have done this.

Problem	Points	Score
1	15	
2	10	
3	10	
4	10	
Total:	45	

1. Calling conventions

(a) (5 points) Below is the source code and a disassembly of a simple C program:

```

int foo(int a) {
    a++;
    return a;
}

int main(int ac, char **av)
{
    return foo(1);
}

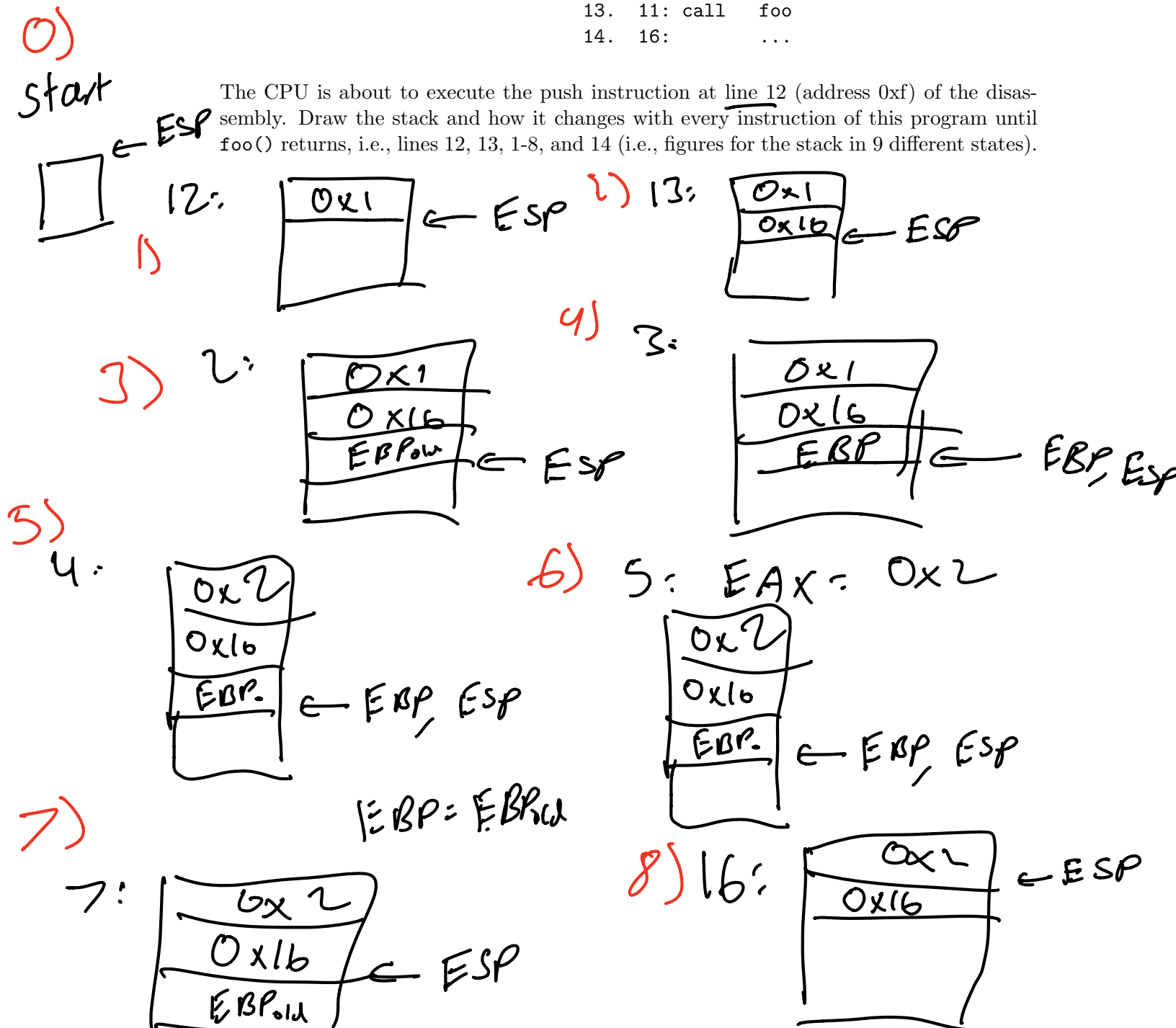
```

```

1. 00000000 <foo>:
2. 0: push    ebp
3. 1: mov     ebp, esp
4. 3: add     DWORD PTR [ebp+0x8], 0x1
5. 7: mov     eax, DWORD PTR [ebp+0x8]
6. a: pop     ebp
7. b: ret
8.
9.
10. 0000000c <main>:
11. ...
12. f: push    0x1    <---- eip
13. 11: call    foo
14. 16: ...

```

The CPU is about to execute the push instruction at line 12 (address 0xf) of the disassembly. Draw the stack and how it changes with every instruction of this program until foo() returns, i.e., lines 12, 13, 1-8, and 14 (i.e., figures for the stack in 9 different states).



- (b) (10 points) Imagine your CPU is identical to x86 but does not have `call` and `ret` instructions. How will you implement the assembly code above (i.e., your code should support recursive invocation of functions)?

`call (label) :`
`push ret-addr`
`jump label`

`ret =`
`pop EBX`
`jump EBX`

??
,,

2. Basic page tables.

(a) (5 points) Which physical address is accessed by the following mov instruction?

```
mov    eax, DWORD PTR [ebx+0x8]
```

Here the `ebx` register contains `0x1000`, and the data segment is configured to have the base of `0x1000`.

Page Directory Page:

PDE 0: PPN=0x1, PTE_P, PTE_U, PTE_W

PDE 1: PPN=0x2, PTE_P, PTE_U, PTE_W

PDE 2: PPN=0x3, PTE_P, PTE_U, PTE_W

... all other PDEs are zero

The Page Table Page at the physical address `0x1000`:

PTE 0: PPN=0x0, PTE_P, PTE_U, PTE_W

PTE 1: PPN=0x1, PTE_P, PTE_U, PTE_W

PTE 2: PPN=0x2, PTE_P, PTE_U, PTE_W

The Page Table Page at the physical address `0x2000`:

PTE 0: PPN=0x4, PTE_P, PTE_U, PTE_W

PTE 1: PPN=0x5, PTE_P, PTE_U, PTE_W

PTE 2: PPN=0x6, PTE_P, PTE_U, PTE_W

The Page Table Page at the physical address `0x3000`:

PTE 0: PPN=0x7, PTE_P, PTE_U, PTE_W

PTE 1: PPN=0x8, PTE_P, PTE_U, PTE_W

PTE 2: PPN=0x9, PTE_P, PTE_U, PTE_W

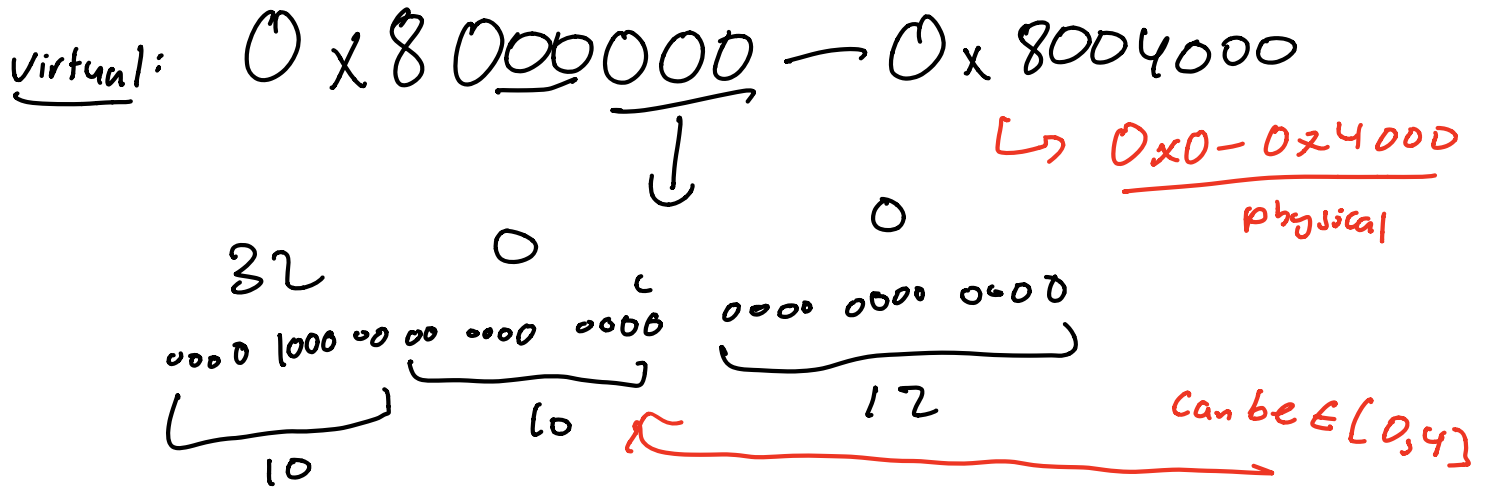
0x1008
 ↳ logical: 0x2008
 4 bits

0... 0010 0000 0000 1000
 10 10

0 2 8
 ↑

0x2008?

- (b) (5 points) Draw a page table (you can use the format similar to the previous question) that maps virtual (or if you want to be more specific, linear) addresses 2GB (0x8000000) to 2GB + 16KB (0x8004000) to physical addresses 0 to 16KB (0x4000).



Page Table Directory (PDT) page:

PDE 32: PPN = 0x1 PTE_P, PTE_U, PTE_W

all other PDEs = 0

in physical

Page Table Page: @ 0x1000

PTE 0: PPN = 0x0

PTE 1: PPN = 0x1

PTE 2: PPN = 0x2

PTE 3: PPN = 0x3

PTE 4: PPN = 0x4

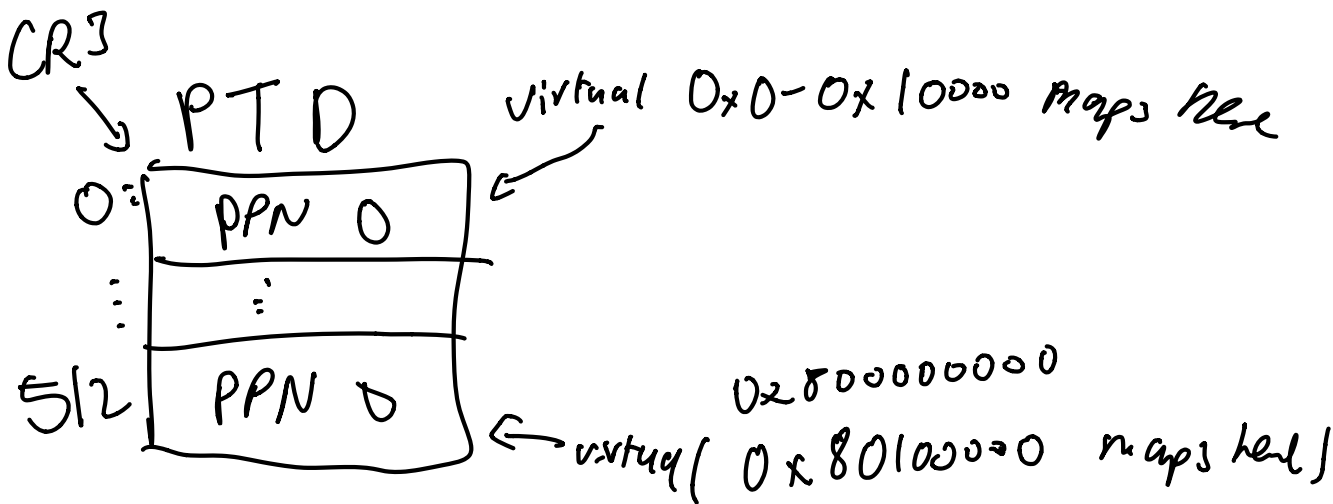
3. Xv6 boot

- (a) (10 points) When xv6 boots, the boot loader loads the xv6 kernel at the physical address 0x10000 (1MB). The boot loader jumps to the entry point of the kernel ELF file which is linked to be at 0x10000c. The first thing the xv6 kernel does is it sets up the boot-time page table that maps two regions of virtual memory (0-4MB and 2GB-(2GB+4MB)) to the physical range 0 to 4MB, and then jumps to the C `main()` function that runs in the 2GB-(2GB+4MB) range. This seems counter-intuitive: the kernel seems to be able to run at two virtual address ranges: 0-4MB and 2GB-(2GB+4MB)—this seems to be against the rules of linking and loading (i.e., the object file is linked to run at a specific address, either 0x10000 or 0x80100000, but the kernel manages to run at these two location).

Can you explain how this works, i.e., what was done to allow the kernel to run in two different address ranges?

The boot time page table allows this to happen.

PPN 0 = 4MB page
[0-4MB) in physical memory



Both map to same physical address!

4. Relocation

Alice compiles the following C file.

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

```
int add (int a, int b) {  
    printf("Numbers are added together\n");  
    return a+b;  
}
```

```
int main() {  
    int a,b;  
    a = 3;  
    b = 4;  
  
    int ret = add(a,b);  
  
    printf("Result: %u\n", ret);  
    return 0;  
}
```

(a) (5 points) Which symbols are undefined and need to be resolved (explain your answer)?

add() undefined b/c not static func
printf() undefined too because imported function

Are the strings undefined too??

← TBM my weakest section so IDK
abt my
answers.

← program starts here

--

- (b) (5 points) Which symbols need to be relocated? Note, that C treats string constants as globals allocated in the read-only data section. Explain your answer.

--