

Exam 1

Prof. Tak Auyeung

April 19, 2005

Instructions: You may bring any material that is hand-written or printed *prior* to the examination to help you. You can also bring a calculator if you think it may help you. However, you can only use the calculator for numerical computations only.

You, as an individual, are expected to do your own work. This means you cannot seek, receive or otherwise acquire any assistance except clarifications from the professor during an examination. Any communication involving the contents of the subject matter or the examination is considered cheating. Do not initiate or accept such communication, or the result of your examination is automatically voided.

Each correct answer is worth one point, each wrong answer is worth -0.25 point, and each unanswered question is worth zero point.

Make sure you write down your name on the upper right corner *first*, otherwise I cannot give points to anonymous students!

Notation: The name of a local variable assumes different roles in different parts of this test. In pseudocode or C code, the name of a local variable refers to the variable itself. In assembly code, unless otherwise noted, the name of a local variable is the displacement from `ebp` in number of bytes to the local variable. The same applies to parameters.

Types: `int16` is a 16-bit signed integer, `int32` is a 32-bit signed integer, `char` is an 8-bit character. We assume all addresses and pointers are 32-bit.

Conventions: Return values are returned by registers. 16-bit and 8-bit values are returned by `ax`, 32-bit values are returned by `eax`.

- 1 The following is a partial subroutine. Most of the prologue and epilogue is included. One instruction is missing. Fill in the blank using one of the choices.

```
sub1:
---
movl  %esp,%ebp
addl  $lastLocalVar,%esp
...
movl  %ebp,%esp
popl  %ebp
ret
```

- A `pushl %ebp`
- B `pushl %esp`
- C `popl %ebp`

- D `movl %ebp,%esp`
- E `subl $4,%esp`

- 2 It is a convention to allocate space for local variables in reversed order. In other words, if a subroutine lists `v1`, `v2` and `v3` as local variables, `v1` is allocated space on the stack last. From a different perspective, `v1` has the lowest address when compared to all other local variables. Which of the following definitions is consistent with this convention? (`v1_size` is the size of `v1` in bytes, and etc.)

In addition, the label `lastLocalVar` should be defined to the displacement from `ebp` to the local variable that has the lowest address.

- A `oldBP = 0`
`v1 = oldBP - v1_size`
`v2 = oldBP - v2_size`
`v3 = oldBP - v3_size`
`lastLocalVar = v3`
- B `oldBP = 0`
`v1 = oldBP - v1_size`
`v2 = v1 - v2_size`
`v3 = v2 - v3_size`
`lastLocalVar = v3`
- C `oldBP = 0`
`v3 = oldBP - v3_size`
`v2 = oldBP - v2_size`
`v1 = oldBP - v1_size`
`lastLocalVar = v1`
- D `oldBP = 0`
`v3 = oldBP - v3_size`
`v2 = v3 - v2_size`
`v1 = v2 - v1_size`
`lastLocalVar = v1`
- E `oldBP = 0`
`v3 = oldBP + v3_size`
`v2 = v1 + v2_size`
`v1 = v2 + v1_size`
`lastLocalVar = v1`

- 3 The following subroutine adds `int32`, and returns the sum in `eax`. Its C prototype is as follows:

```
int32 add2(int32 x, int32 y);
```

It is implemented by the following assembly code. Note that the prologue and epilogue code are omitted but assumed done correctly, as are definitions of the displacements:

```
add2:
    # displacement definiteions
    # prologue
    pushl %eax
    movl  x(%ebp),%eax
    addl  y(%ebp),%eax
    popl  %eax
    # epilogue
    ret
```

Is there a problem with this subroutine? If so, what is it?

- A There is no problem.
- B ax should be used instead of eax.
- C eax should not be pushed and popped.
- D x(%ebp) should be just x, and similarly for y(%ebp).
- E The movl and addl instructions should be swapped.

4 The following code is used by a caller to invoke the subroutine sub2. How should we clean up the stack? Assume the parameters are useless after the called subroutine returns. Select one of the choices to substitute

```
----
    pushl $52
    pushw myVar(%ebp)
    call  sub2
    ---
```

- A subl \$52,%esp
- B subl \$52,%ebp
- C addl \$6,%esp
- D popl \$52
 - popw myVar(%ebp)
- E popw myVar(%ebp)
 - popl \$52

5 Assume the following C prototypes:

```
int32 f1(int32 x);
int32 f2(int32 x);
int32 f3(int32 x);
```

Assuming that v1 is a local variable of the caller, how do we implement the following code?

```
v1 = f1(f2(f3(1234)));
```

```
A pushl $1234
   call f3
   call f2
   call f1
   # clean up parameter
   movl %eax,v1(%ebp)
```

```
B pushl $1234
   call f1
   call f2
   call f3
   # clean up parameter
   movl %eax,v1(%ebp)
```

```
C pushl $1234
   call f1
   # clean up parameter
   pushl %eax
   call f2
   # clean up parameter
   pushl %eax
   call f3
   # clean up parameter
   movl %eax,v1(%ebp)
```

```
D pushl $1234
   call f3
   popl %eax
   pushl %eax
   call f2
   popl %eax
   pushl %eax
   call f1
   popl %eax
   movl %eax,v1(%ebp)
```

```
E pushl $1234
   call f3
   movl %eax,(%esp)
   call f2
   movl %eax,(%esp)
   call f1
   # clean up parameter
   movl %eax,v1(%ebp)
```

6 I need to preserve the value of ebx and ecx. However, the subroutine sub3 can potentially overwrite these two registers. How do I preserve these registers? Assume the original code to call sub3 is as follows:

```
pushl $123 # 2nd parameter
pushl $94 # 1st parameter
call  sub3
# parameter clean up code
```

```
A  pushl $123 # 2nd parameter
   pushl $94 # 1st parameter
   pushl %ebx
   pushl %ecx
   call  sub3
```

```

    popl %ecx
    popl %ebx
    # parameter clean up code

```

```

B   pushl $123 # 2nd parameter
    pushl $94 # 1st parameter
    pushl %ebx
    pushl %ecx
    call  sub3
    popl  %ebx
    popl  %ecx
    # parameter clean up code

```

```

C   pushl %ebx
    pushl %ecx
    pushl $123 # 2nd parameter
    pushl $94 # 1st parameter
    call  sub3
    # parameter clean up code
    popl  %ecx
    popl  %ebx

```

```

D   pushl %ebx
    pushl %ecx
    pushl $123 # 2nd parameter
    pushl $94 # 1st parameter
    call  sub3
    popl  %ebx
    popl  %ecx
    # parameter clean up code

```

E There is no way to preserve the values of `ebx` and `ecx` by the same stack used for parameter passing.

7 Assume a subroutine has a C prototype as follows:

```
void *f4(void);
```

This means it returns a pointer (address), and there is no parameter. The following is the implementation in assembly language:

```

f4:
    popl %eax
    jmp  (%eax)

```

The following is how this subroutine is used, assume `mystery` is a local variable of the caller:

```

call  f4
movl  %eax,mystery(%ebp)

```

Choose the correct explanation.

- A The subroutine crashes (segmentation fault) when it attempts to execute the `jmp` instruction.
- B The subroutine crashes (segmentation fault) when it attempts to execute the `popl` instruction.
- C The subroutine does not crash, but the stack is not balanced.

D `mystery` (as a variable) stores the address of the `movl` instruction in the caller's code.

E `mystery` (as a variable) stores some unspecifiable value.

8 In C notation, "`int32 *p;`" means `p` is a pointer to a 32-bit integer. "`int32 i;`" means `i` is a 32-bit integer. "`&i`" means the address of `i`. How do we store the address of `i` in `p`? In other words, how do we implement "`p = &i;`"? Assume `p` and `i` are either local variables or parameters.

```

A   movl i(%ebp),%eax
    movl %eax,p(%ebp)

```

```

B   movl $i,%eax
    addl %ebp,%eax
    movl %eax,p(%ebp)

```

```

C   movl $i,%eax
    addl %ebp,%eax
    movl p,%ebx
    addl %ebp,%ebx
    movl %eax,%ebx

```

```

D   movl $i,%eax
    addl %ebp,%eax
    movl p,%ebx
    addl %ebp,%ebx
    movl %ebx,%eax

```

E 8a and 8c

9 A subroutine call should use the following code:

```

pushl $1234
pushl myVar(%ebp)
call  sub5
# parameter clean up

```

However, the programmer makes a mistake, and it becomes the following:

```

pushw $1234
pushl myVar(%ebp)
call  sub5
# parameter clean up

```

Assuming that the subroutine does not attempt to change the parameters, what is the result of the erroneous code? Assume the "parameter clean up code" is correct with respect to the parameters actually passed.

- A The program always crashes in the subroutine `sub5`.
- B The program always crashes in the caller, after the `call` instruction.
- C The program always crashes as the caller returns to its own caller.

- D The program may not crash, but `sub5` may not perform its calculations correctly.
- E The program works fine as is, since 1234 can be represented in 16 bits.

10 Subroutine `sub6` expects no parameters. What happens when the following code is used to call `sub6`?

```
pushl $1234
pushl myVar(%ebp)
call sub6
# parameter clean up
```

- A The program always crashes in the subroutine `sub5`.
- B The program always crashes in the caller, after the `call` instruction.
- C The program always crashes as the caller returns to its own caller.
- D The program may not crash, but the behavior of `sub6` may not be correct.
- E The program works fine as is.

- 11 A
- B
- C
- D
- E

- 12 A
- B
- C
- D
- E