

# Practice Final

Prof. Tak Auyeung

December 13, 2004

**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 zero point, and each unanswered question is also worth zero point. This means you *should* guess and leave no question unanswered.

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

**A word on symbols.** For a global and static variable, the assembly symbol corresponding to a C variable name is the absolute address of the base of the variable. For a C auto local variable or parameter, the assembly symbol of the same name is the offset (in number of bytes) from the frame pointer to the base of the parameter or auto local variable.

For any field *F* of **struct** *S*, the symbol `offset_S_F` is the offset in number of bytes from the base of the structure (record) *S* to field *F*.

For any **struct** *S*, the symbol `sizeof_S` is defined to be the size of a **struct** *S* in number of bytes.

- 1 The following is the C code representation of a record (structure):

```
struct X
{
    int i;
    int j;
};

struct Y
{
    int i;
    struct X a;
};
```

Assume that an `int` is a 32-bit integer. How many bytes should be allocated for a variable of type `struct Y`?

- A 2
- B 4
- C 8
- D 12
- E 20

**The answer is 1d.** **struct X** has two integers, each integer has 4 bytes. As a result, a **struct X** has 8 bytes. A **struct Y** has a **struct X** (as field `a`) and an integer (as field `i`), so it has 8+4 bytes, which is 12.

- 2 Refer to question 1, we add an auto local variable definition as follows:

```
struct Y *p;
```

In a program, we need to perform the following:

```
p->a.i = 0;
```

How do we implement this in assembly?

- A `movl p(%ebp),%eax`  
`movl $0,offset_Y_a+offset_X_i(%eax)`
- B `movl p+offset_Y_a(%ebp),%eax`  
`movl $0,offset_X_i(%eax)`
- C `movl p+offset_Y_a+offset_X_i(%ebp),%eax`  
`movl $0,(%eax)`
- D `movl $0,p+offset_Y_a+offset_X_i(%ebp)`  
`movl $0,(%eax)`
- E 2b and 2d

**The answer is 2a.** Because `p` is a pointer, its value must be first loaded into a register before it can be dereferenced (indirectly accessed). This pretty much rules out all but 2a. Since `eax` points to the base of a **struct Y**, we need to get to field `a` by adding `offset_Y_a` as an offset to field `a`. Once we get field `a`, we need to add `offset_X_i` to get to field `i` of the **struct X** (field `a`).

- 3 Refer to question 1. Now we add two auto local variable definitions in C:

```
struct Y *a[10];
int i;
```

How do we implement the following statement in assembly:

```
a[i]->i = 23;
```

- A `movl a(%ebp,i,4),%ebx`  
`movl $23,offset_Y_i(%ebx)`
- B `movl a+i*sizeof_Y(%ebp),%ebx`  
`movl $23,offset_Y_i(%ebx)`
- C `movl i(%ebp),%eax`  
`movl $23,a+sizeof_Y+offset_Y_i(%eax)`
- D `movl i(%ebp),%eax`  
`movl a(%ebp,%eax,4),%ebx`  
`movl $23,offset_Y_i(%ebx)`
- E All of the above implement the C statement

**The answer is 3d.** `i` is an indexing variable that changes at run time (cannot be determined at assemble time), it must be loaded into a register before we can use it as an index! This leaves two choices remaining (C and D). Next, we need to use it as an index. Luckily, array `a` is an array of pointers. This means we can use the built-in indexing addressing mode. The operand `a(%ebp,%eax,4)` gets to the pointer. In this case, `ebp` is a “base register”, `eax` is an indexing register, and 4 is the scaling factor. Once the address of a `struct Y` is loaded into `ebx`, we get to field `i` using the offset `offset_Y_i`.

- 4 Refer to question 1. Assume we allocate a local variable `y` of type `struct Y` from the stack. Symbols are defined as follows:

```
oldEbp = 0
y = oldEbp - sizeof_Y
```

The matching code to perform the allocation at run time is as follows:

```
pushl %ebp
movl %esp,%ebp
addl $y,%esp
```

Given these assumptions, how do we perform the following statement in assembly?

```
y.a.j = 12;
```

- A `movl y(%ebp),%eax`  
`movl offset_Y_a(%eax),%ebx`  
`movl $12,offset_X_j(%ebx)`
- B `movl $12,y+offset_Y_a+offset_X_j(%ebp)`
- C `movl y+offset_Y_a(%ebp),%eax`  
`movl $12,offset_X_j(%eax)`
- D 4a and 4b
- E all of the above accomplishes the C statement

**The answer is 4b.** This one is kind of simple. Because `y` is the offset from `ebp` to the base of a `struct Y`, we just lump all the offsets on it directly. To be verbose, we add `offset_Y_a` to get to field `a`, which turns out to be a `struct X`. Next, we add `offset_X_j` to get to field `j` of field `a` of `y`. Note that the other choices do not work because whenever we load something into a register, then use that register indirectly, we are dereferencing a pointer. There is no pointer to be dereferenced.

- 5 Refer to question 1. Assume we allocate a global (static) variable `y` of type `struct Y`. A Symbol is defined as follows:

```
.data
y: .fill 1, sizeof_Y, 0
```

Now, we need to call a subroutine expecting a single parameter that is of `struct X *` type. How do we pass *the address* of `y.a` as a parameter?

- A `pushl $y+offset_Y_a`
- B `pushl y+offset_Y_a`
- C `subl $offset_Y_a,%esp`  
`movl %esp,%ebx`  
`movl $y,%eax`  
L0:  
`cmpl $y+sizeof_Y, %eax`  
`jnb L1`  
`movl (%eax),%ecx`  
`movl %ecx,(%ebx)`  
`addl $4,%eax`  
`addl $4,%ebx`  
`jmp L0`  
L1:
- D `pushl $y`  
`addl $offset_Y_a,(%esp)`
- E 5a and 5d

**The answer is 5e.** In this case, `y` is an absolute address to the base of a `struct Y`. We want to push the *address* of field `a` on the stack. The expression `y+offset_Y_a` is the address. We want to use immediate mode because we are not pushing the first four bytes of field `a` of `y`. Choice D is just a clumsier way to computer the sum of `y` and `offset_Y_a`.

- 6 Refer to 1. `ptr` is an auto local variable pointer to a `struct Y`.

```
struct Y *ptr;
```

How do we implement the C code `++(ptr->i)` so that field `i` of what `ptr` points to is incremented?

- A `addl $sizeof_Y,ptr+offset_Y_i(%ebp)`

```

B movl ptr(%ebp),%eax
  movl offset_Y_i(%eax),%ebx
  addl $1,%ebx
C movl ptr(%ebp),%eax
  movl offset_Y_i(%eax),%ebx
  addl $1,(%ebx)
D addl $1,ptr+offset_Y_i(%ebp)
E movl ptr(%ebp),%eax
  addl $1,offset_Y_i(%eax)

```

The answer is 6e. ptr is a pointer, which means we need to first load it into a register, then access whatever it points to. This rules out the first choice. Because eax points to the base of a struct Y, the memory operand offset\_Y\_i(%eax) is field i of the struct Y pointed to by ptr. We can then perform the incrementing by adding one to it.

7 In a subroutine, we need to allocate an array of 12 8-bit characters as the only (auto) local variable. This is done by the following code:

```

oldEbp = 0
localArray = oldEbp - 12
sub1:
pushl %ebp
movl %esp,%ebp
addl $localArray,%esp
...

```

Which C expression will corrupt the saved ebp?

- A localArray[0] = 0;
- B localArray[11] = 0;
- C localArray[12] = 0;
- D localArray[16] = 0;
- E localArray[17] = 0;

The answer is 7c. localArray[0] to localArray[11] are fine because the array has 12 characters. localArray[12] is the least significant byte of the saved ebp. FYI, localArray[16] is the least significant byte of the return address.

8 Describe the result of executing the following code.

```

pushl %eax
pushl %ebx
pushl %ecx
pushl %edx
movl $4,%eax
movl $1,%ebx
pushl $'\n'
movl %esp,%ecx
movl $1,%edx
addl $4,%esp

```

```

popl %edx
popl %ecx
popl %ebx
popl %eax

```

- A Nothing
- B It writes a newline character to the standard output file. There is no problem.
- C It reads a character from the standard input file. There is no problem.
- D It crashes.
- E It does something useful, and it does not crash immediately. But, the stack is left unbalanced.

The answer is 8a. Note that int \$0x80 is missing. So we saved all the registers, load them up with something, then restore all the registers, all without actually requesting the OS to do something.

9 Using our standard method to allocate parameters and local variables from the stack, how many bytes are required every time we call the following subroutine? Assume each int is a 32-bit integer.

```

void f(void *x, int *y)
{
  int i;
  int *j;

  // statements
}

```

- A 4
- B 8
- C 16
- D 24
- E Cannot be determined because we don't know the size of the array that x points to.

The answer is 9d. A pointer to anything requires 4 bytes. An integer requires 4 bytes. We need 4 × 4 = 16 bytes just for the parameters and local variables. Then we need 4 bytes for the return address, and another four bytes for the saved ebp. The total adds up to 24. However, since I did not mention explicitly to include the hidden stuff, I'll accept 16 as correct as well.

10 j and k are both auto local variables of int type. What assembly code passes j as a parameter to subroutine s1, and store the returned value to k? Assume eax is used to return an int value. In other words, how do we implement the following C code?

```
k = s1(j);
```

- A `pushl $j`  
`call s1`  
`popl %eax`  
`movl %eax,k`
- B `pushl j`  
`call s1`  
`movl %eax,k`
- C `pushl j(%ebp)`  
`call s1`  
`addl $4,%esp`  
`movl %eax,k(%ebp)`
- D `pushl j(%ebp)`  
`call s1`  
`popl k(%ebp)`
- E 10c and 10d

**The answer is 10c.** Since `j` is an auto local variable, we need to use `ebp` to get to it. This leaves C and D as possible answers. However, the return value is specified in `eax` (not on the stack), only C makes sense.

- 11 Some architectures cannot `push` immediate operands. How else can we implement the following instruction? Assume that we can use another register as a temporary register, and that the status flag can be modified. Also, assume all other instructions and all addressing modes are supported.

- ```
pushl $3
```
- A `subl $4,%esp`  
`movl $3,(%esp)`
  - B `movl $3,%eax`  
`pushl %eax`
  - C `pushl 3`
  - D `movl $3,(%esp)`
  - E 11a and 11b

**The answer is 11e.** Option A is just another way to push something (copied almost exactly from my notes). Option B is fairly obvious.

- 12 Consider the following instruction. In which case will both the `0` flag and `Z` flag be set (don't worry about the other flags)?

- ```
addl %eax,%ebx
```
- A `%eax = 0x80000000` and `%ebx = 0x80000000`
  - B `%eax = 0x7fffffff` and `%ebx = 0x7fffffff`
  - C `%eax = 0x00000001` and `%ebx = 0x7fffffff`
  - D None of the above, but there is such a case
  - E None of the above, but such a case does not exist

**The answer is 12a.** The `Z` flag is set because 0 is the result, and it will be stored in `ebx`. However, 0 is also set because the sign of the result makes no sense. Adding two negative values should result in a negative value. The bit pattern of `0x00000000` indicates that it is non-negative. As a result, the `0` flag is also set. Note that none of the other choices will set the `Z` flag because the results are not zero.

- 13 The instruction `jna` often does not exist on most other architectures. How can we implement it using other instructions? For example, how do we implement `jna L1` for all cases?

- A `jc L1`
- B `jc L1`  
`jz L1`
- C `jng L1`
- D `jnz L0`  
`jc L1`  
`L0:`
- E `jnc L1`  
`jz L1`

**The answer is 13b.** `jna` means “jump if not above”. This translates to “jump if less than or equal to”. `jc` takes care of “less than”, and `jz` takes care of “equal to”. As a result, between those two, we get the same logic.

- 14 Refer to question 1. Assume the following:

- `struct Y y1` is a parameter
- `struct Y y2` is a parameter
- `int num` is an auto local variable

How do we implement the following statement?

```
y1.i = y2.a.j + num;
```

- A `movl y2+offset_Y_a+offset_X_j(%ebp),%eax`  
`addl num(%ebp),%eax`  
`movl %eax,y2+offset_Y_i(%ebp)`
- B `movl y2+offset_Y_a+offset_X_j(%ebp),y1+offset_Y_i`  
`addl num(%ebp),y1+offset_Y_i(%ebp)`
- C `movl num(%ebp),%eax`  
`addl y2+offset_Y_a+offset_X_j(%ebp),%eax`  
`movl %eax,y1+offset_Y_i(%ebp)`
- D `addl $y2+offset_Y_a+offset_X_j,num(%ebp)`  
`movl $offset_Y_a+offset_X_j,y1+offset_Y_i(%ebp)`
- E 14a and 14c

**The answer is 14e.** Since addition is commutative, either both A and C are both the answer, or they both are not the answer. B can be ruled out because `movl` cannot have two memory operands. D can be ruled out because adding the displacement to `num` does not do us any good. I mean, D is way, way, *way* off.

- 15 If the C flag is set after the following instruction, what do we know about the registers?

```
cmpl %eax,%ebx
```

- A %eax is (unsigned) greater than %ebx
- B %eax is (signed) greater than %ebx
- C %ebx is (unsigned) greater than %eax
- D %ebx is (signed) greater than %eax
- E not one of the above can be confirmed

**The answer is 15a.** The C flag is set iff the second operand is less than the first (unsigned). This is the same to say that the first operand is greater than the second (unsigned).

- 16 We suspect that a subroutine `bart` does not balance the stack. In order to confirm this, we try to write some code so that we jump to label `yikes` if and only if the stack is not balanced. Which of the following invocation code does this?

We assume subroutine `bart` does not have any parameters, and it does not return any value.

- A 

```
pushl %esp
call bart
cmpl %esp,(%esp)
jnz yikes
addl $4,%esp
```
- B 

```
movl %esp,%eax
call bart
cmpl %eax,%esp
jnz yikes
```
- C 

```
pushl %esp
call bart
cmpl %esp,(%esp)
jnz yikes
addl $4,%esp
```
- D 

```
movl %esp,%eax
pushl %eax
call bart
cmpl %eax,(%esp)
jnz yikes
addl $4,%esp
```
- E none of the above code will do it

**The answer is 16e.** This one is a little tricky. If the stack is unbalanced, the `ret` instruction of the called subroutine cannot guarantee to get the proper return address, which means our checking code never gets control in the first place.

- 17 An operand of which of the following addressing modes does not access memory for the operand itself?

- A immediate

- B register
- C direct
- D indirect
- E indexed

**The answer is 17a.** Okay, this one is a freebie (or at least it was intended to be one).

- 18 Which of the following is not a valid instruction and will cause assemble time error?

- A `pushl %eax`
- B `pushl 1`
- C `pushl $1`
- D `pushl (%eax)`
- E all of the above are valid instructions

**The answer is 18e.** Okay, another freebie.

- 19 Which conditional jump is not supported by the 386 architecture?

- A `jpg`
- B `jpl`
- C `js`
- D `jb`
- E all of the above are supported

**The answer is 19e.** Okay, *yet* another freebie.

- 20 If the following instruction sequence is repeated 32 times, what will be the value of register `%eax`? The answer has to work for all possible initial values of `%eax`.

```
addl %eax,%eax
addl $1,%eax
```

- A 0
- B the most positive value of a 32-bit signed number
- C the most positive value of a 32-bit unsigned number
- D the most negative value of a 32-bit signed number
- E 32

**The answer is 20c.** Not all questions are freebies. `addl %eax,%eax` is multiply by two, which is a left shift in binary numbers. Adding one after a left shift guarantees that the least significant bit is a 1. Doing this 32 times guarantees that all original bits are shifted out, and replaced by all 1's.

- 21 Assume that `%eax` has a value of 23 (in decimal) initially. What is the value of `%eax` after the following instructions?

```
pushl %eax
subl  %eax, (%esp)
addl  %eax, %eax
addl  %eax, %eax
addl  %eax, %eax
popl  %eax
```

- A 23
- B 230
- C 46
- D 184
- E 0

**The answer is 21e.** The second instruction zeros out whatever the stack pointer points to. It doesn't matter what we do to `eax` after that, because we just pop the value of 0 back into `eax` at the very end.

- 22 Instead of using the `jz` instruction, which other replacement instruction can be used in all cases for the following situation?

```
cmpl $0, %eax
jz   L0
```

- A `ja`
- B `jna`
- C `j1`
- D `jb`
- E None of the above can replace `jz` in this context

**The answer is 22b.** Let's think about this. `cmpl $0, %eax` compares `eax` to 0. `jz` means we jump iff `eax` is zero. When a number is interpreted unsigned, the only value that is not above 0 is 0 itself! This is why we can use `jna` in place of `jz` in this specific situation.

- 23 After the following instruction, one and only one of the `O` and `S` flags is set. What can we know for sure?

```
cmpl %ebx, %eax
```

- A `%eax` is less than `%ebx` unsigned
- B `%ebx` is less than `%eax` signed
- C `%eax` is not less than `%ebx` unsigned
- D `%ebx` is no less than `%eax` signed
- E `%eax` is less than `%ebx` signed

**The answer is 23e.** This is kind of a freebie. The imaginary `L` flag is the exclusive-or (`xor`) of the `O` flag and the `S` flag. This means the `L` flag (`L` for less than) is set iff one and only one of the `O` or `S` flag is set.

- 24 Which of the following instructions does not access (read or write) memory (other than the instructions)?

- A `pushl $1`
- B `jmp L1`
- C `movl $0, (%eax)`
- D `cmpb $0, someChar`
- E `addl 4, %esp`

**The answer is 24b.** What can I say? I ran out of tricky questions. E is not an answer because 4 is a direct operand that accesses memory location 4.