

Practice Final

Prof. Tak Auyeung

May 12, 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!

Use the following assumptions unless otherwise instructed.

Integer types. `int32` is a signed 32-bit integer, `int16` is a signed 16-bit integer. Unsigned integers are `uintXX`, replace `XX` with the number of bits. `char` is a character, considered unsigned in this test.

Parameters and local variables. Labels with the same names as parameters and local variables (in C) are defined as the displacement from the frame pointer to the first byte of the construct.

Return values. `eax` is used to return an `int32` result, `ax` is used to return an `int16` result.

Structures. Given a structure named `S1`, `S1_SIZE` is the size of the structure in bytes. Given a field called `X` in structure `S1`, `S1_X` is the displacement of field `X` from the first byte of a structure `S1` in bytes. `S1_X_SIZE` is the size of field `X` in structure `S1`.

Non-scalar return value. Space reserved for the return value is allocated on the stack immediately *higher than* the last parameter for subroutines returning non-scalar (arrays, structures) values.

1 What is the effective address of the following operand?

```
label1(,%edx,2)
```

- A `label1+%edx+2`
- B `(label1+%edx)*2`
- C `label1+(%edx*2)`
- D this operand is invalid

E the operand is valid, but the effective address is none of the above

2 Which of the following instructions does not access memory (other than fetching the instruction itself)?

- A `popl %eax`
- B `popl (%eax)`
- C `addl $2,%esp`
- D `addl $2,2`
- E `pushl %eax`

3 Which flag or flags are interpreted by `jnz`?

- A only Z
- B only Z and C
- C all flags except Z
- D only O
- E only O and S

4 Which pseudocode fits the following code? Assume X and Y are 32-bit signed integer local variables/parameters.

```
label0:
    movl X(%ebp),%eax
    cmpl Y(%ebp),%eax
    jnl  label1
    # block of code here
label1:
```

- A **if** `X < Y` **then**
 block of code here
end if
- B **if** `X > Y` **then**
 block of code here
end if
- C **while** `X < Y` **do**
 block of code here
end while
- D **while** `X > Y` **do**
 block of code here
end while
- E **repeat**
 block of code here
until `X > Y`

5 Which of the following instructions cannot possibly set the overflow 0 flag, regardless of the value of %eax?

- A `addl $1,%eax`
- B `addl %eax,%eax`
- C `subl $1,%eax`
- D `cmpl $1,%eax`
- E all of the above instructions may set the 0 flag

6 What is the value of %dx after the following instructions?

```
movw $16,%ax
movw $16,%bx
mulw %bx
```

- A 16
- B 1
- C 0
- D cannot be determined because %dx is not initialized
- E %dx is not changed by the instructions

7 Which of the following is equivalent to `pushl $0x12345678`?

- A `pushw $0x1234`
`pushw $0x5678`
- B `pushw $0x5678`
`pushw $0x1234`
- C `pushw $0x8765`
`pushw $0x4321`
- D `pushw $0x4321`
`pushw $0x8765`
- E None of the above because we don't know the value at location 0x12345678

8 Which C code is consistent with the following subroutine definition?

```
sub1:
    oldBP = 0
    retAddr = oldBP + 4
    p1 = retAddr + 4
    p2 = p1 + 2
    pushl %ebp
    movl %esp,%ebp

    movl $0,%eax
    movw p2(%ebp),%ax
    addw p1(%ebp),%ax

    movl %ebp,%esp
    popl %ebp
    ret
```

```
A void sub1(int32 p1, int32 p2)
{
    p1 = p1 + p2;
}
```

```
B int32 sub1(int32 p1, int32 p2)
{
    return p1 + p2;
}
```

```
C int16 sub1(int32 p1, int16 p2)
{
    return p1 + p2;
}
```

```
D int16 sub1(int16 p1, int16 p2)
{
    return p1 + p2;
}
```

```
E int32 sub1(int16 p1, int16 p2)
{
    return p1 + p2;
}
```

9 Assume that the following subroutine is already implemented:

```
void memcpy(void *dest, void *src, int32 n);
```

In which `dest` is a destination pointer, `src` is a source pointer, and `n` specifies the number of bytes to copy from where `src` points at to where `dest` points at. The special type `void *` means we don't care *what* the pointers are pointing at.

How many bytes must be available on the stack for a successful call to `memcpy`?

- A 3
- B 4
- C 8
- D 12
- E 16

10 The following code is for adding two complex numbers:

```
struct Complex
{
    int32 i;
    int32 j;
};

struct Complex Complex_add(
    struct Complex a,
    struct Complex b)
{
    a.i = a.i + b.i;
    a.j = a.j + b.j;
    return a;
}
```

How do we implement the following statement?

```
x = Complex_add(p, q);
```

Assume x, p and q are all of type `struct Complex` and they are all parameters or local variables. Also, assume `memcpy` is implemented as described in question 9. The following code is used to implement the call:

```
subl $V1,%esp
movl %esp,%eax
pushl $Complex_Size
pushl $q
INSTR1
pushl %eax
call memcpy
addl $12
subl $V1,%esp
movl %esp,%eax
pushl $Complex_Size
pushl $p
INSTR1
pushl %eax
call memcpy
addl $12
call Complex_add
addl $V1*2, %esp
movl %esp,%eax
pushl $V1
pushl %eax
pushl $x
INSTR1
call memcpy
addl $12
addl $V1
```

How should we substitute `INSTR1` and `V1`?

- A `INSTR1` is `nop`, and `V1` is 4
- B `INSTR1` is `addl %ebp,%esp`, and `V1` is `Complex_SIZE`
- C `INSTR1` is `pushl Complex_SIZE`, and `V1` is 4
- D `INSTR1` is `addl %ebp,(%esp)`, and `V1` is `Complex_SIZE`
- E `INSTR1` is `addl %ebp,(%esp)`, and `V1` is 4

11 Given that `arr` is an array of `int32` as a local variable, how do we add elements of the array from index 0 until we encounter an element that has a value of zero? Let us use `%edx` to store the sum of elements in `arr`.

```
A  movl $0,%edx
   movl $arr,%eax
   addl %ebp,%eax
L1:
   cmpl $0,(%eax)
```

```
   jz  L2
   addl (%eax),%edx
   addl $4,%eax
   jmp  L1
L2:
B  movl $0,%edx
   movl $0,%eax
L1:
   cmpl $0,arr(%ebp,%eax,4)
   jz  L2
   addl arr(%ebp,%eax,4)
   addl $1,%eax
   jmp  L1
L2:
C  movl $0,%edx
   movl %ebp,%eax
L1:
   cmpl $0,arr(%eax)
   jz  L2
   addl arr(%eax),%edx
   addl $4,%eax
   jmp  L1
L2:
D 11a and 11c
E 11a, 11b and 11c
```

12 Assume `arr` is an array of `int16`, and it is a local variable. In addition, the current values are 45, 6, 0, 12, 5 and 20 (from element 0 to element 5). We define `arr_NITEMS` as 6 (number of items). What are the elements values in the array, from element 0 to element 5, after the following code executes?

```
movl $0,%eax
movl $1,%ebx
L1:
  cmpl $arr_NITEMS,%ebx
  jnb  L2
  movw arr(%ebp,%eax,2),%dx
  cmpw arr(%ebp,%ebx,2),%dx
  jng  L1_0
  pushw arr(%ebp,%eax,2)
  pushw arr(%ebp,%ebx,2)
  popw arr(%ebp,%eax,2)
  popw arr(%ebp,%ebx,2)
L1_0:
  addl $1,%ebx
  jmp  L1
L2:
```

- A 45, 6, 0, 12, 5, 20
- B 0, 6, 12, 5, 20, 45
- C 0, 5, 6, 12, 20, 45
- D 0, 45, 6, 12, 5, 20
- E 45, 20, 12, 6, 5, 0