Namn:____

Personnummer:____

Datorarkitektur, 2009

Tentamen 2009-03-13

Instructions:

- Make sure that your exam is not missing any sheets, then write your full name on the front.
- Write your answers in the space provided below the problem. If you make a mess, clearly indicate your final answer.
- The exam has a maximum score of 60 points plus 3 possible bonus points.
- The aproximate limits for grades on this exam are:
  - To pass (grade E): 30 points.
  - For grade D: 37 points.
  - For grade C: 45 points.
  - For grade B: 52 points.
  - For grade A: 59 points.
- The problems are of varying difficulty. The point value of each problem is indicated. Pile up the easy points quickly and then come back to the harder problems.
- This exam is OPEN BOOK. You may use any books or notes you like but no computer, calulator, telephone etc. Good luck!

Problem 1. (9 points):

Assume we are running code on a 10-bit machine using two's complement arithmetic for signed integers. A "short" integer is encoded using 5 bits. Fill in the empty boxes in the table below. The following definitions are used in the table:

```
short sy = -6;
int y = sy;
int x = -23;
unsigned ux = x;
```

Note: You need not fill in entries marked with "–".

Expression	Decimal Representation	Binary Representation
Zero	0	
–	-10	
–	29	
–		`01 1010 0010`
ux		
y		
$x >> 3$		
TMax		
$-$TMin		
TMax $+$ TMax		
TMin $+$ TMin		

Problem 2. (14 points):

Consider the following 12-bit floating point representation based on the IEEE floating point format:

- There is a sign bit in the most significant bit.
- The next five bits are the exponent. The exponent bias is 15.
- The last six bits are the significand.

The rules are like those in the IEEE standard (normalized, denormalized, representation of 0, infinity, and NAN).

We consider the floating point format to encode numbers in a form:

$$(-1)^s \times m \times 2^E$$

where m is the *mantissa* and E is the exponent.

Fill in the table below for the following numbers, with the following instructions for each column:

Hex: The 3 hexadecimal digits describing the encoded form.

m**:** The fractional value of the mantissa. This should be a number of the form x or x/y, where x is an integer, and y is an integral power of 2. Examples include: 0, 23/16, and 1/64.

E**:** The integer value of the exponent.

Value: The numeric value represented. Use the notation x or $x \times 2^z$, where x and z are integers.

As an example, to represent the number $7/2$, we would have $s = 0$, $m = 7/4$, and $E = 1$. Our number would therefore have an exponent field of `0x10` (decimal value $15 + 1 = 16$) and a significand field `0x30` (binary 110000_2), giving a hex representation `430`.

You need not fill in entries marked "—".

Description	Hex	m	E	Value
-0				-0
Smallest value > 1				
256				—
Largest Denormalized				
$-\infty$		—	—	$-\infty$
Number with hex representation `3A0`	`3A0`			

Problem 3. (8 points):

Consider the source code below, where `M` and `N` are constants declared with `#define`.

```
int mat1[M][N];
int mat2[N][M];

int copy_element(int i, int j)
{
    mat1[i][j] = mat2[j][i];
}
```

This generates the following assembly code:

```
copy_element:
        pushl   %ebp
        movl    %esp, %ebp
        movl    8(%ebp), %ecx
        leal    (%ecx,%ecx,2), %edx
        sall    $3, %edx
        movl    12(%ebp), %eax
        subl    %ecx, %edx
        addl    %eax, %edx
        leal    (%eax,%eax,4), %eax
        addl    %ecx, %eax
        movl    mat2(,%eax,4), %eax
        movl    %eax, mat1(,%edx,4)
        leave
        ret
```

A. What is the value of `M`:

B. What is the value of `N`:

Problem 4. (10 points):

Buffer overflow

This problem concerns the following C code, excerpted from Dr. Evil’s best-selling autobiography, “World Domination My Way”. He calls the program *NukeJr*, his baby nuclear bomb phase.

```
/*
 * NukeJr - Dr. Evil’s baby nuke
 */
#include <stdio.h>
#define EOF -1

int overflow(void);
int one = 1;

/* main - NukeJr’s main routine */
int main() {
  int val = overflow();

  val += one;
  if (val != 15213)
    printf("Boom!\n");
  else
    printf("Curses! You’ve defused NukeJr!\n");
  _exit(0); /* syscall version of exit that doesn’t need %ebp */
}

/* overflow - writes to stack buffer and returns 15213 */
int overflow() {
  char buf[4];
  int val, i=0;

  while(scanf("%x", &val) != EOF)
    buf[i++] = (char)val;
  return 15213;
}
```

Buffer overflow (cont)

Here is the corresponding machine code for NukeJr when compiled and linked on a Linux/x86 machine:

```
08048560 <main>:
 8048560:       55                pushl  %ebp
 8048561:       89 e5             movl   %esp,%ebp
 8048563:       83 ec 08          subl   $0x8,%esp
 8048566:       e8 31 00 00 00    call   804859c <overflow>
 804856b:       03 05 90 96 04    addl   0x8049690,%eax        # val += one;
 8048570:       08
 8048571:       3d 6d 3b 00 00    cmpl   $0x3b6d,%eax          # val == 15213?
 8048576:       74 0a             je     8048582 <main+0x22>
 8048578:       83 c4 f4          addl   $0xfffffff4,%esp
 804857b:       68 40 86 04 08    pushl  $0x8048640
 8048580:       eb 08             jmp    804858a <main+0x2a>
 8048582:       83 c4 f4          addl   $0xfffffff4,%esp
 8048585:       68 60 86 04 08    pushl  $0x8048660
 804858a:       e8 75 fe ff ff    call   8048404 <_init+0x44>  # call printf
 804858f:       83 c4 10          addl   $0x10,%esp
 8048592:       83 c4 f4          addl   $0xfffffff4,%esp

0804859c <overflow>:
 804859c:       55                pushl  %ebp
 804859d:       89 e5             movl   %esp,%ebp
 804859f:       83 ec 10          subl   $0x10,%esp
 80485a2:       56                pushl  %esi
 80485a3:       53                pushl  %ebx
 80485a4:       31 f6             xorl   %esi,%esi
 80485a6:       8d 5d f8          leal   0xfffffff8(%ebp),%ebx
 80485a9:       eb 0d             jmp    80485b8 <overflow+0x1c>
 80485ab:       90                nop
 80485ac:       8d 74 26 00       leal   0x0(%esi,1),%esi
 80485b0:       8a 45 f8          movb   0xfffffff8(%ebp),%al    # L1: loop start
 80485b3:       88 44 2e fc       movb   %al,0xfffffffc(%esi,%ebp,1)
 80485b7:       46                incl   %esi
 80485b8:       83 c4 f8          addl   $0xfffffff8,%esp
 80485bb:       53                pushl  %ebx
 80485bc:       68 80 86 04 08    pushl  $0x8048680
 80485c1:       e8 6e fe ff ff    call   8048434 <_init+0x74>    # call scanf
 80485c6:       83 c4 10          addl   $0x10,%esp
 80485c9:       83 f8 ff          cmpl   $0xffffffff,%eax
 80485cc:       75 e2             jne    80485b0 <overflow+0x14> # goto L1
 80485ce:       b8 6d 3b 00 00    movl   $0x3b6d,%eax
 80485d3:       8d 65 e8          leal   0xffffffe8(%ebp),%esp
 80485d6:       5b                popl   %ebx
 80485d7:       5e                popl   %esi
 80485d8:       89 ec             movl   %ebp,%esp
 80485da:       5d                popl   %ebp
 80485db:       c3                ret
```

Buffer overflow (cont)

This problem tests your understanding of the stack discipline and byte ordering. Here are some notes to help you work the problem:

- Recall that Linux/x86 machines are Little Endian.
- The `scanf("%x", &val)` function reads a whitespace-delimited sequence of characters from `stdin` that represents a hex integer, converts the sequence to a 32-bit `int`, and assigns the result to `val`. The call to `scanf` returns either 1 (if it converted a sequence) or EOF (if no more sequences on `stdin`).

  For example, calling `scanf` four time on the input string `"0 a ff"` would have the following result:

  - 1st call to `scanf`: `val=0x0` and `scanf` returns 1.
  - 2nd call to `scanf`: `val=0xa` and `scanf` returns 1.
  - 3rd call to `scanf`: `val=0xff` and `scanf` returns 1.
  - 4th call to `scanf`: `val=?` and `scanf` returns EOF.

Buffer overflow (questions):

A. After the `subl` instruction at address `0x804859f` in function `overflow` completes, the stack contains a number of objects which are shown in the table below. Determine the address of each object as a byte offset from `buf[0]`.

Stack object	Address of stack object
return address	`&buf[0] + ________`
old %ebp	`&buf[0] + ________`
buf[3]	`&buf[0] + ________`
buf[2]	`&buf[0] + ________`
buf[1]	`&buf[0] + 1`
buf[0]	`&buf[0] + 0`

B. What input string would defuse NukeJr by causing the call to `overflow` to return to address `0x8048571` instead of `804856b`? *Notes: (i) Your solution is allowed to trash the contents of the* `%ebp` *register. (ii) Each underscore is a one or two digit hex number.*

`Answer: "0  0  0  0  ___  ___  ___  ___  ___  ___  ___  ___ "`

Problem 5. (8 points):

In this problem you will specify how to implement some new instructions for the Y86 machine.
The actions of an instruction is decribed inte the coursebook by a table that shows what is done in each step of the machine. Here are three examples:

Stage	`OPl` rA, rB	`irmovl` V, rB	`pushl` rA
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_4[PC+2] valP ← PC+6	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2
Decode	valA ← R[rA] valB ← R[rB]		valA ← R[rA] valB ← R[%esp]
Execute	valE ← valB OP valA Set CC	valE ← 0 + valC	valE ← valB + (-4)
Memory			M_4[valE] ← valA
Write back	R[rB] ← valE	R[rB] ← valE	R[%esp] ← valE
PC update	PC ← valP	PC ← valP	PC ← valP

You shall describe the three instructions `incr`, `decr` and `not`, that implements the following C-operations:

instruction	C-operation
`incr` x	x = x + 1
`decr` x	x = x - 1
`not` x	x = ∼x

All three instructions have the format:

icode	ifun	8	rB

All three instructions set the condition codes similar to `OPl`.

Fill in the operations done in each stage:

Stage	`incr` rB	`decr` rB	`not` rB
Fetch	icode:ifun $\leftarrow M_1[PC]$	icode:ifun $\leftarrow M_1[PC]$	icode:ifun $\leftarrow M_1[PC]$
Decode			
Execute			
Memory			
Write back			
PC update			

Problem 6. (8 points):

You are writing a new 3D game that you hope will earn you fame and fortune. You are currently working on a function to blank the screen buffer before drawing the next frame. The screen you are working with is a 640x480 array of pixels. The machine you are working on has a 64 KB direct mapped cache with 4 byte lines. The C structures you are using are:

```
struct pixel {
    char r;
    char g;
    char b;
    char a;
};

struct pixel buffer[480][640];
register int i, j;
register char *cptr;
register int *iptr;
```

Assume:

- `sizeof(char) = 1`
- `sizeof(int) = 4`
- `buffer` begins at memory address 0
- The cache is initially empty.
- The only memory accesses are to the entries of the array `buffer`. Variables `i`, `j`, `cptr`, and `iptr` are stored in registers.

A. What percentage of the writes in the following code will miss in the cache?

```
for (j=0; j < 640; j++) {
    for (i=0; i < 480; i++){
        buffer[i][j].r = 0;
        buffer[i][j].g = 0;
        buffer[i][j].b = 0;
        buffer[i][j].a = 0;
    }
}
```

Miss rate for writes to `buffer`: _________ %

B. What percentage of the writes in the following code will miss in the cache?

```
char *cptr;
cptr = (char *) buffer;
for (; cptr < (((char *) buffer) + 640 * 480 * 4); cptr++)
    *cptr = 0;
```

Miss rate for writes to `buffer`: _________ %

C. What percentage of the writes in the following code will miss in the cache?

```
int *iptr;
iptr = (int *) buffer;
for (; iptr < (buffer + 640 * 480); iptr++)
    *iptr = 0;
```

Miss rate for writes to `buffer`: _________ %

D. Which code (A, B, or C) should be the fastest? _________

Problem 7. (3 points):

Consider the following C functions and assembly code:

```
int fun1(int a, int b)
{
    unsigned ua = (unsigned) a;
    if (ua < b)
        return b;
    else
        return ua;
}

int fun2(int a, int b)
{
    if (b < a)
        return b;
    else
        return a;
}

int fun3(int a, int b)
{
    if (a < b)
        return a;
    else
        return b;
}
```

```
funX:
        pushl   %ebp
        movl    %esp, %ebp
        movl    12(%ebp), %eax
        cmpl    8(%ebp), %eax
        jl      .L5
        movl    8(%ebp), %eax
.L5:
        leave
        ret
```

Which of the functions compiled into the assembly code shown?