Namn:____

Personnummer:____________________________________

Datorarkitektur, 2008

Tentamen 2008-03-14

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. Good luck!

Problem 1. (8 points):

Consider a **7-bit** two's complement representation. Fill in the empty boxes in the following table. Addition and subtraction should be performed based on the rules for 7-bit, two's complement arithmetic

Number	Decimal Representation	Binary Representation
Zero	0	
n/a	-3	
n/a	11	
n/a	-17	
n/a		`0 110011`
n/a		`1 010010`
TMax		
TMin		
TMin+TMin		
TMin+1		
TMax+1		
$-$TMax		
$-$TMin		

Problem 2. (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    12(%ebp), %edx
        leal    0(,%edx,8), %eax
        movl    8(%ebp), %ecx
        subl    %edx, %eax
        pushl   %ebx
        leal    (%ecx,%eax,2), %eax
        leal    (%ecx,%ecx,8), %ebx
        movl    mat2(,%eax,4), %eax
        addl    %edx, %ebx
        movl    %eax, mat1(,%ebx,4)
        popl    %ebx
        leave
        ret
```

A. What is the value of `M`:

B. What is the value of `N`:

Problem 3. (14 points):

Consider the source code below, used to keep track of the rooms currently reserved in a family-run hotel. Each entry in the residents array stores a name of the customer reserving the room. FLOORS represents the number of floors in the hotel. ROOMS represents the number of rooms per floor. Both are constants declared with #define. LEN, the maximum number of bytes allocated for a name, is defined to be 12.

```
char residents[FLOORS][ROOMS][LEN];

void
reserve_room(int floor, int room, char *custname)
{
        strcpy(residents[floor][room], custname);
}
```

The assembly code for the function reserve_room looks like this:

```
reserve_room:
        pushl %ebp
        movl %esp,%ebp
        movl 12(%ebp),%eax
        movl 16(%ebp),%edx
        pushl %edx
        movl 8(%ebp),%edx
        sall $4,%edx
        subl 8(%ebp),%edx
        leal (%eax,%eax,2),%eax
        leal residents(,%eax,4),%eax
        leal (%eax,%edx,4),%edx
        pushl %edx
        call strcpy
        movl %ebp,%esp
        popl %ebp
        ret
```

A. What is the value of ROOMS?

B. Due to a strange bug, the program accesses residents[0][1][-2]. What value is actually being accessed? (Express your answer as residents[x][y][z] where x, y and z are all non-negative such that residents[x][y][z] access the same value. You may assume that FLOORS and ROOMS are both greater than 1)

C. The programmer realizes that this implementation is wasteful of memory. Successive fires in several memory chip factories in Taiwan drive up memory prices and finally convince him to improve the memory efficiency of his implementation to maintain the competitiveness of the family hotel.

The declaration of `residents` is changed to be a two dimensional array of pointers to character strings (names). The new code allocates memory for customer names only for those rooms that are actually reserved. Otherwise, `residents[f][r]` stores a NULL pointer. **For simplicity, assume there is no storage overhead due to `malloc`.**

The new declaration looks like this:

```
char *residents[FLOORS][ROOMS];

void
reserve_room(int floor, int room, char *custname)
{
    residents[floor][room] = malloc(LEN);
    strcpy(residents[floor][room], custname);
}
```

After a few months. The programmer goes back to review the memory savings of his improved scheme. During that period, the hotel was 20% reserved. The programmer is delighted because the savings are found to be 168 bytes! How many floors does this hotel have? (that is, what is the value of FLOORS?)

Problem 4. (10 points):

Condider the following assembly code for a C `for` loop:

```
loop:
        pushl %ebp
        movl %esp,%ebp
        movl 0x8(%ebp),%edx
        movl %edx,%eax
        addl 0xc(%ebp),%eax
        leal 0xffffffff(%eax),%ecx
        cmpl %ecx,%edx
        jae .L4
.L6:
        movb (%edx),%al
        xorb (%ecx),%al
        movb %al,(%edx)
        xorb (%ecx),%al
        movb %al,(%ecx)
        xorb %al,(%edx)
        incl %edx
        decl %ecx
        cmpl %ecx,%edx
        jb .L6
.L4:
        movl %ebp,%esp
        popl %ebp
        ret
```

Based on the assembly code above, fill in the blanks below in its corresponding C source code. (Note: you may only use the symbolic variables `h`, `t` and `len` in your expressions below — *do not use register names.*)

```
void loop(char *h, int len)
{
   char *t;

   for (_____________; ___________; h++, t--) {

      _______________;

      _______________;

      _______________;
   }

   return;
}
```

Problem 5. (6 points):

This problem concerns the following, low-quality code:

```
void foo(int x)
{
  int a[3];
  char buf[4];
  a[0] = 0xF0F1F2F3;
  a[1] = x;
  gets(buf);
  printf("a[0] = 0x%x, a[1] = 0x%x, buf = %s\n", a[0], a[1], buf);
}
```

In a program containing this code, procedure `foo` has the following disassembled form on an IA32 machine:

```
080485d0 <foo>:
80485d0:       55               pushl  %ebp
80485d1:       89 e5            movl   %esp,%ebp
80485d3:       83 ec 10         subl   $0x10,%esp
80485d6:       53               pushl  %ebx
80485d7:       8b 45 08         movl   0x8(%ebp),%eax
80485da:       c7 45 f4 f3 f2   movl   $0xf0f1f2f3,0xfffffff4(%ebp)
80485df:       f1 f0
80485e1:       89 45 f8         movl   %eax,0xfffffff8(%ebp)
80485e4:       8d 5d f0         leal   0xfffffff0(%ebp),%ebx
80485e7:       53               pushl  %ebx
80485e8:       e8 b7 fe ff ff   call   80484a4 <_init+0x54>  # gets
80485ed:       53               pushl  %ebx
80485ee:       8b 45 f8         movl   0xfffffff8(%ebp),%eax
80485f1:       50               pushl  %eax
80485f2:       8b 45 f4         movl   0xfffffff4(%ebp),%eax
80485f5:       50               pushl  %eax
80485f6:       68 ec 90 04 08   pushl  $0x80490ec
80485fb:       e8 94 fe ff ff   call   8048494 <_init+0x44>  # printf
8048600:       8b 5d ec         movl   0xffffffec(%ebp),%ebx
8048603:       89 ec            movl   %ebp,%esp
8048605:       5d               popl   %ebp
8048606:       c3               ret
8048607:       90               nop
```

For the following questions, recall that:

- `gets` is a standard C library routine.
- IA32 machines are little-endian.
- C strings are null-terminated (i.e., terminated by a character with value 0x00).
- Characters '`0`' through '`9`' have ASCII codes `0x30` through `0x39`.

Fill in the following table indicating where on the stack the following program values are located. Express these as decimal offsets (positive or negative) relative to register `%ebp`:

Program Value	Decimal Offset
`a`	
`a[2]`	
`x`	
`buf`	
`buf[3]`	
Saved value of register `%ebx`	

Problem 6. (9 points):

The following problem concerns optimizing a procedure for maximum performance on an Intel Pentium III. Recall the following performance characteristics of the functional units for this machine:

Operation	Latency	Issue Time
Integer Add	1	1
Integer Multiply	4	1
Integer Divide	36	36
Floating Point Add	3	1
Floating Point Multiply	5	2
Floating Point Divide	38	38
Load or Store (Cache Hit)	1	1

Consider the following two procedures:

Loop 1

```
int loop1(int *a, int x, int n)
{
  int y = x*x;
  int i;
  for (i = 0; i < n; i++)
    x = y * a[i];
  return x*y;
}
```

Loop 2

```
int loop2(int *a, int x, int n)
{
  int y = x*x;
  int i;
  for (i = 0; i < n; i++)
    x = x * a[i];
  return x*y;
}
```

When compiled with GCC, we obtain the following assembly code for the inner loop:

Loop 1

```
.L21:
   movl %ecx,%eax
   imull (%esi,%edx,4),%eax
   incl %edx
   cmpl %ebx,%edx
   jl .L21
```

Loop 2

```
.L27:
   imull (%esi,%edx,4),%eax
   incl %edx
   cmpl %ebx,%edx
   jl .L27
```

Running on a Intel Pentium III machine, we find that Loop 1 requires 3.0 clock cycles per iteration, while Loop 2 requires 4.0.

A. Explain how it is that Loop 1 is faster than Loop 2, even though it has one more instruction

B. By using the compiler flag `-funroll-loops`, we can compile the code to use 4-way loop unrolling. This speeds up Loop 1. Explain why.

C. Even with loop unrolling, we find the performance of Loop 2 remains the same. Explain why.

Problem 7. (5 points):

The following problem concerns basic cache lookups.

- The memory is byte addressable.
- Memory accesses are to **1-byte words** (not 4-byte words).
- Physical addresses are 13 bits wide.
- The cache is 2-way set associative, with a 4 byte line size and 16 total lines.

In the following tables, **all numbers are given in hexadecimal**. The contents of the cache are as follows:

2-way Set Associative Cache												
Index	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3
0	09	1	86	30	3F	10	00	0	99	04	03	48
1	45	1	60	4F	E0	23	38	1	00	BC	0B	37
2	EB	0	2F	81	FD	09	0B	0	8F	E2	05	BD
3	06	0	3D	94	9B	F7	32	1	12	08	7B	AD
4	C7	1	06	78	07	C5	05	1	40	67	C2	3B
5	71	1	0B	DE	18	4B	6E	0	B0	39	D3	F7
6	91	1	A0	B7	26	2D	F0	0	0C	71	40	10
7	46	0	B1	0A	32	0F	DE	1	12	C0	88	37

Part 1

The box below shows the format of a physical address. Indicate (by labeling the diagram) the fields that would be used to determine the following:

CO	The block offset within the cache line
CI	The cache index
CT	The cache tag

12	11	10	9	8	7	6	5	4	3	2	1	0

Part 2

For the given physical address, indicate the cache entry accessed and the cache byte value returned **in hex**. Indicate whether a cache miss occurs.

If there is a cache miss, enter "**-**" for "Cache Byte returned".

Physical address: `0E34`

A. Physical address format (one bit per box)

12	11	10	9	8	7	6	5	4	3	2	1	0

B. Physical memory reference

Parameter	Value
Byte offset	0x
Cache Index	0x
Cache Tag	0x
Cache Hit? (Y/N)	
Cache Byte returned	0x