Namn:_____

Personnummer:_____

Datorarkitektur, 2006

Tentamen 2006-06-02

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.
- The aproximate limits for grades on this exam are:
  - To pass (G or 3): 30 points.
  - For grade 4: 43 points.
  - For grade VG: 50 points.
  - For grade 5: 55 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. (12 points):

Consider the following 5-bit floating point representation based on the IEEE floating point format.
There is a sign bit in the most significant bit.
The next three bits are the exponent, with an exponent bias 3.
The last bit is the fraction.
The rules are like those in the IEEE standard (normalized, denormalized, representation of 0, infinity, and NAN).
The floating point format encode numbers in a form:

$$V \;=\; (-1)^s \times M \times 2^E$$

where M is the *significand* and E is the *exponent*.
Fill in missing entries in the table below with the following instructions for each column:

Description: Some unique property of this number, such as, "The largest denormalized value."

Binary: The 5 bit representation.

M**:** The value of the Mantissa written in decimal format.

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

Value: The numeric value represented, written in decimal format.

You need not fill in entries marked "—". For the arithmetic expressions, recall that the rule with IEEE format is to round to the number nearest the exact result. Use "round-to-even" rounding.

Description	Binary	M	E	Value
Minus Zero				-0.0
Positive Infinity		—	—	$+\infty$
	`01101`			
Smallest number > 0				
One				1.0
$4.0 - 0.75$				
$2.0 + 3.0$				

Problem 2. (9 points):

Consider the following C declarations:

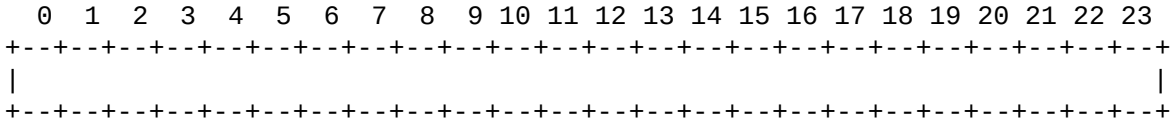
```
typedef struct {
    short code;
    long start;
    char raw[3];
    double data;
} OldSensorData;

typedef struct {
    short code;
    short start;
    char raw[5];
    short sense;
    short ext;
    double data;
} NewSensorData;
```

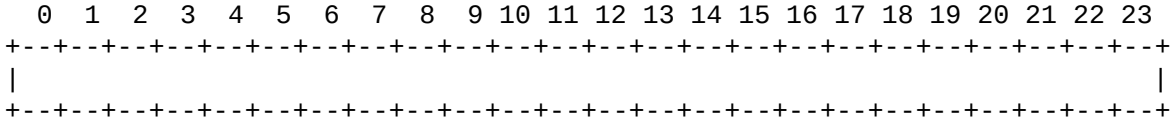
A. Using the templates below (allowing a maximum of 24 bytes), indicate the allocation of data for structs of type `OldSensorData` `NewSensorData`. Mark off and label the areas for each individual element (arrays may be labeled as a single element). **Cross hatch the parts that are allocated, but not used (to satisfy alignment).**

Assume the Linux alignment rules described in book and discussed in lectures. **Clearly indicate the right hand boundary of the data structure with a vertical line.**

`OldSensorData:`



`NewSensorData:`



B. Now consider the following C code fragment:

```
void foo(OldSensorData *oldData)
{
    NewSensorData *newData;

    /* this zeros out all the space allocated for oldData */
    bzero((void *)oldData, sizeof(oldData));

    oldData->code = 0x104f;
    oldData->start = 0x80501ab8;
    oldData->raw[0] = 0xe1;
    oldData->raw[1] = 0xe2;
    oldData->raw[2] = 0x8f;
    oldData->raw[-5] = 0xff;
    oldData->data = 1.5;

    newData = (NewSensorData *) oldData;

    ...
```

Once this code has run, we begin to access the elements of newData. Below, give the value of each element of `newData` that is listed. Assume that this code is run on a Little-Endian machine such as a Linux/x86 machine. You must give your answer in hexadecimal format. **Be careful about byte ordering!**.

(a) `newData->start  = 0x`_____

(b) `newData->raw[0] = 0x`_____

(c) `newData->raw[2] = 0x`_____

(d) `newData->raw[4] = 0x`_____

(e) `newData->sense  = 0x`_____

Problem 3. (8 points):

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

```
int array1[M][N];
int array2[N][M];

void copy(int i, int j)
{
    array1[i][j] = array2[j][i];
}
```

Suppose the above code generates the following assembly code:

```
copy:
  pushl %ebp
  movl %esp,%ebp
  pushl %ebx
  movl 8(%ebp),%ecx
  movl 12(%ebp),%eax
  leal 0(,%eax,4),%ebx
  leal 0(,%ecx,8),%edx
  subl %ecx,%edx
  addl %ebx,%eax
  sall $2,%eax
  movl array2(%eax,%ecx,4),%eax
  movl %eax,array1(%ebx,%edx,4)
  popl %ebx
  movl %ebp,%esp
  popl %ebp
  ret
```

What are the values of `M` and `N`?

`M` =

`N` =

Problem 4. (12 points):

In this problem, you are given the task of reconstructing C code based on some declarations of C structures and unions, and the IA32 assembly code generated when compiling the C code.

Below are the data structure declarations. (Note that these declarations are shown horizontally rather than vertically simply so that they fit on one page.)

```
struct s1 {            struct s2 {           union u1 {
  char a[3];             struct s1 *d;         struct s1 *h;
  union u1 b;            char e;               struct s2 *i;
  int c;                 int f[4];             char j;
};                       struct s2 *g;       };
                       };
```

You may find it helpful to diagram these data structures.

For each IA32 assembly code sequence below on the left, fill in the missing portion of corresponding C source line on the right.

A.
```
proc1:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    movl 12(%eax),%eax
    movl %ebp,%esp
    popl %ebp
    ret
```
```
int proc1(struct s2 *x)
{
  return x->____________________ ;
}
```

B.
```
proc2:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    movl 4(%eax),%eax
    movl 20(%eax),%eax
    movl %ebp,%esp
    popl %ebp
    ret
```
```
int proc2(struct s1 *x)
{
  return x->____________________ ;
}
```

C.
```
proc3:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    movl (%eax),%eax
    movsbl 4(%eax),%eax
    movl %ebp,%esp
    popl %ebp
    ret
```
```
char proc3(union u1 *x)
{
  return x->____________________ ;
}
```

D.
```
proc4:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    movl (%eax),%eax
    movl 24(%eax),%eax
    movl (%eax),%eax
    movsbl 1(%eax),%eax
    movl %ebp,%esp
    popl %ebp
    ret
```
```
char proc4(union u1 *x)
{
  return x->____________________ ;
}
```

Problem 5. (7 points):

Match each of the assembler routines on the left with the equivalent C function on the right.

```
foo1:
     pushl %ebp
     movl %esp,%ebp
     movl 8(%ebp),%eax
     sall $4,%eax
     subl 8(%ebp),%eax
     movl %ebp,%esp
     popl %ebp
     ret

foo2:
     pushl %ebp
     movl %esp,%ebp
     movl 8(%ebp),%eax
     testl %eax,%eax
     jge .L4
     addl $15,%eax
.L4:
     sarl $4,%eax
     movl %ebp,%esp
     popl %ebp
     ret

foo3:
     pushl %ebp
     movl %esp,%ebp
     movl 8(%ebp),%eax
     shrl $31,%eax
     movl %ebp,%esp
     popl %ebp
     ret
```

```
int choice1(int x)
{
    return (x < 0);
}

int choice2(int x)
{
    return (x << 31) & 1;
}

int choice3(int x)
{
    return 15 * x;
}

int choice4(int x)
{
    return (x + 15) /4
}

int choice5(int x)
{
    return x / 16;
}

int choice6(int x)
{
    return (x >> 31);
}
```

Fill in your answers here:

foo1 corresponds to choice _____.

foo2 corresponds to choice _____.

foo3 corresponds to choice _____.

Problem 6. (3 points):

Consider the following C functions and assembly code:

```
int fun7(int a)
{
    return a * 30;
}

int fun8(int a)
{
    return a * 34;
}

int fun9(int a)
{
    return a * 18;
}
```

```
pushl %ebp
movl %esp,%ebp
movl 8(%ebp),%eax
sall $4,%eax
addl 8(%ebp),%eax
addl %eax,%eax
movl %ebp,%esp
popl %ebp
ret
```

Which of the functions compiled into the assembly code shown?

Problem 7. (4 points):

Consider the following fragment of IA32 code from the C standard library:

```
0x400446e3 <malloc+7>: call   0x400446e8 <malloc+12>
0x400446e8 <malloc+12>: popl   %eax
```

After the `popl` instruction completes, what hex value does register `%eax` contain?

Problem 8. (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 12 bits wide.
- The cache is 4-way set associative, with a 2-byte block size and 32 total lines.

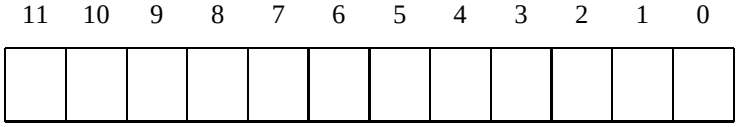
In the following tables, **all numbers are given in hexadecimal**. The contents of the cache are as follows:

4-way Set Associative Cache																
Index	Tag	Valid	Byte 0	Byte 1	Tag	Valid	Byte 0	Byte 1	Tag	Valid	Byte 0	Byte 1	Tag	Valid	Byte 0	Byte 1
0	29	0	34	29	87	0	39	AE	7D	1	68	F2	8B	1	64	38
1	F3	1	0D	8F	3D	1	0C	3A	4A	1	A4	DB	D9	1	A5	3C
2	A7	1	E2	04	AB	1	D2	04	E3	0	3C	A4	01	0	EE	05
3	3B	0	AC	1F	E0	0	B5	70	3B	1	66	95	37	1	49	F3
4	80	1	60	35	2B	0	19	57	49	1	8D	0E	00	0	70	AB
5	EA	1	B4	17	CC	1	67	DB	8A	0	DE	AA	18	1	2C	D3
6	1C	0	3F	A4	01	0	3A	C1	F0	0	20	13	7F	1	DF	05
7	0F	0	00	FF	AF	1	B1	5F	99	0	AC	96	3A	1	22	79

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



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: `3B6`

A. Physical address format (one bit per box)

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

B. Physical memory reference

Parameter	Value
Cache Offset (CO)	0x
Cache Index (CI)	0x
Cache Tag (CT)	0x
Cache Hit? (Y/N)	
Cache Byte returned	0x