Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Department of Computer Science

Computer Architecture and Systems Programming

252-0061-00

Monday 21st January 2013, 9:00-12:00

Last Name : ______________________________

First Name : ______________________________

Leginr. : ______________________________

Rules

- You have 180 minutes for the exam.
- Please write your name and Legi-ID number on all sheets of paper.
- Please write your answers on the exam sheet. Please also use the reverse sides of the exam sheets. If you need more paper, please raise your hand so that we can provide you with additional paper. Write your name and Legi-ID number on those extra sheets of paper.
- Write as clearly as possible and cross out everything that you do not consider to be part of your solution. You must give your answers in either English or German.
- The exam consists of 14 questions. The maximum number of points that can be achieved is 170.
- This exam paper consists of 30 pages in addition to this title page. Please read through the exam paper to ensure that you have all the pages, and if not, please raise your hand.
- You are not allowed to use any written aids in this exam, except for a German-English dictionary and the x86 reference sheet that should be on your desk. If the reference sheet is missing, please raise your hand.

Name: ______________________________ Leginr: ____________________

Question 1 [10 points]

The next question 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>

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;
}
```

[Question continues on the next page]

[continued]

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
 8048595:       6a 00           pushl  $0x0
 8048597:       e8 b8 fe ff ff  call   8048454 <_init+0x94>  # call _exit

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
```

[Question continues on the next page]

[continued]

This question uses the NukeJr program to test your understanding of stack discipline and byte ordering. Here are some notes to help you answer the question:

- 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.

(3 points)

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. Fill in the three blank values by determining the address of each object as a byte offset from `buf[0]`:

Stack object	Address of stack object
return address	`&buf[0]` + 8
old %ebp	`&buf[0]` + _______
buf[3]	`&buf[0]` + _______
buf[2]	`&buf[0]` + _______
buf[1]	`&buf[0]` + 1
buf[0]	`&buf[0]` + 0

[Question continues on the next page]

[continued]

What input string would defuse NukeJr by causing the call to `overflow` to return to address `0x8048571` instead of `804856b`?

(7 points)

Notes:

- Your solution is allowed to trash (change) the contents of the `%ebp` register.
- Write your answer below as a sequence of one- or two-digit hex numbers, one for each character in the input string.
- Show your working.

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

Question 2 [16 points]

Consider a **6-bit** two's complement representation.

Fill in the empty boxes in the following table:

Number	Decimal Representation	Binary Representation
Zero	0	
n/a	-1	
n/a	5	
n/a	-10	
n/a		01 1010
n/a		10 0110
TMax		
TMin		
TMax+TMax		
TMin+TMin		
TMin+1		
TMin−1		
TMax+1		
−TMax		
−TMin		

Name: ______________________________ Leginr: ____________________

Question 3 [18 points]

Consider the following 16-bit floating point representation based on the IEEE floating point format:

- There is a sign bit in the most significant bit.
- The next seven bits are the exponent. The exponent bias is 63.
- The last eight bits are the significand.

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

As described in the course, 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 4 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, 67/64, and 1/256.

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 `0x40` (decimal value $63 + 1 = 64$) and a significand field `0xC0` (binary 11000000_2), giving a hex representation `40C0`.

You need not fill in entries marked "—".

Description	Hex	m	E	Value
-0				—
Smallest value > 1				
256				—-
Largest Denormalized				
$-\infty$		—	—	—
Number with hex representation `3AA0`	—			

Question 4 [13 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;
```

Using the templates below (allowing a maximum of 24 bytes), indicate the allocation of data for structs of type `OldSensorData` and `NewSensorData`. Mark off and label the areas for each individual element (arrays may be labeled as a single element).

Cross out the parts that are allocated to satisfy alignment, but not used for data.

Assume the Linux alignment rules discussed in class (in particular, `doubles` are aligned to 4-byte boundaries).

Clearly indicate the right hand boundary of the data structure with a vertical line.

(8 points)

`OldSensorData:`

```
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                                                       |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

`NewSensorData:`

```
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                                                       |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

[Question continues on the next page]

[continued]

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!**.

(5 points)

- `newData->start  = 0x_______________`
- `newData->raw[0] = 0x_______________`
- `newData->raw[2] = 0x_______________`
- `newData->raw[4] = 0x_______________`
- `newData->sense  = 0x_______________`

Question 5 [8 points]

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

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

int 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),%ebx
  leal (%ecx,%ecx,8),%edx
  sall $2,%edx
  movl %ebx,%eax
  sall $4,%eax
  subl %ebx,%eax
  sall $2,%eax
  movl array2(%eax,%ecx,4),%eax
  movl %eax,array1(%edx,%ebx,4)
  popl %ebx
  movl %ebp,%esp
  popl %ebp
  ret
```

What are the values of M and N? Show your working.

M =

N =

Name: ______________________________ Leginr: ________________

Question 6 [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;
}
```

Question 7 [8 points]

This next problem will test your understanding of stack frames. It is based on the following recursive C function:

```
int silly(int n, int *p)
{
    int val, val2;

    if (n > 0) {
        val2 = silly(n << 1, &val);
    } else {
        val = val2 = 0;
    }

    *p = val + val2 + n;

    return val + val2;
}
```

This yields the following machine code:

```
silly:
        pushl %ebp
        movl %esp,%ebp
        subl $20,%esp
        pushl %ebx
        movl 8(%ebp),%ebx
        testl %ebx,%ebx
        jle .L3
        addl $-8,%esp
        leal -4(%ebp),%eax
        pushl %eax
        leal (%ebx,%ebx),%eax
        pushl %eax
        call silly
        jmp .L4
        .p2align 4,,7
.L3:
        xorl %eax,%eax
        movl %eax,-4(%ebp)
.L4:
        movl -4(%ebp),%edx
        addl %eax,%edx
        movl 12(%ebp),%eax
        addl %edx,%ebx
        movl %ebx,(%eax)
        movl -24(%ebp),%ebx
        movl %edx,%eax
        movl %ebp,%esp
        popl %ebp
        ret
```

[Question continues on the next page]

[continued]

Is the variable `val` stored on the stack? If so, at what byte offset (relative to `%ebp`) is it stored, and why is it necessary to store it on the stack?

(2 points)

Is the variable `val2` stored on the stack? If so, at what byte offset (relative to `%ebp`) is it stored, and why is it necessary to store it on the stack?

(2 points)

What (if anything) is stored at `-24(%ebp)`? If something is stored there, why is it necessary to store it?

(2 points)

What (if anything) is stored at `-8(%ebp)`? If something is stored there, why is it necessary to store it?

(2 points)

Question 8 [7 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
```

[Question continues on the next page]

[continued]

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

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

(3 points)

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.

(2 points)

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

(2 points)

Name: ______________________ Leginr: ______________

Question 9 [9 points]

This question is about how to program devices such a UARTs and Ethernet controllers from system software (such as the operating system).

In the course we discussed how the *hardware device* and the software *device driver* can be thought of as finite state machines (FSMs). In this view, *data* is transferred between the two state machines, and *events* are used by one FSM (either the driver or the device) to signal a state transition in the other FSM.

Give two ways in which data is transferred **from** the driver to the device:

(2 points)

Give two ways in which data is transferred **to** the driver from the device:

(2 points)

How can the driver signal events to the device?

(2 points)

[Question continues on the next page]

[continued]

The device signals events to the device driver using *interrupts*. Explain the function of the Programmable Interrupt Controller (PIC) in a computer system, and list the problems it solves.

(3 points)

Question 10 [16 points]

Consider a direct mapped cache of size 64K with block size of 16 bytes. Furthermore, the cache is write-back and write-allocate.

You will calculate the miss rate for the following code using this cache. Remember that `sizeof(int) == 4`. Assume that the cache starts cold (i.e. empty) and that local variables and computations take place completely within the registers and do not spill onto the stack.

Express all cache miss rates as percentages.

First consider the following code to copy one matrix to another.

Assume that the `src` matrix starts at address 0 and that the `dest` matrix follows immediately follows it.

```
void copy_matrix(int dest[ROWS][COLS], int src[ROWS][COLS])
{
    int i, j;

    for (i=0; i<ROWS; i++) {
        for (j=0; j<COLS; j++) {
            dest[i][j] = src[i][j];
        }
    }
}
```

1. What is the cache miss rate if `ROWS = 128` and `COLS = 128`?

2. What is the cache miss rate if `ROWS = 128` and `COLS = 192`?

3. What is the cache miss rate if `ROWS = 128` and `COLS = 256`?

(8 points)

[Question continues on the next page]

[continued]

Now consider the following two implementations of a horizontal flip and copy of the matrix. Again assume that the `src` matrix starts at address 0 and that the `dest` matrix follows immediately follows it.

The first implementation:

```
void copy_n_flip_matrix1(int dest[ROWS][COLS], int src[ROWS][COLS])
{
    int i, j;

    for (i=0; i<ROWS; i++) {
        for (j=0; j<COLS; j++) {
            dest[i][COLS - 1 - j] = src[i][j];
        }
    }
}
```

1. What is the cache miss rate if `ROWS = 128` and `COLS = 128`?

2. What is the cache miss rate if `ROWS = 128` and `COLS = 192`?

The second implementation:

```
void copy_n_flip_matrix2(int dest[ROWS][COLS], int src[ROWS][COLS])
{
    int i, j;

    for (j=0; j<COLS; j++) {
        for (i=0; i<ROWS; i++) {
            dest[i][COLS - 1 - j] = src[i][j];
        }
    }
}
```

1. What is the cache miss rate if `ROWS = 128` and `COLS = 128`?

2. What is the cache miss rate if `ROWS = 192` and `COLS = 128`?

(8 points)

Question 11 [8 points]

Consider the following C declaration:

```
struct Node{
   char c;
   double value;
   struct Node* next;
   int flag;
   struct Node* left;
   struct Node* right;
};

typedef struct Node* pNode;

/* NodeTree is an array of N pointers to Node structs */
pNode NodeTree[N];
```

For each of the four C references below, please indicate which Linux/IA32 assembly code section (labeled A – F) places the value of that C reference into register `%eax`. If no match is found, please write “NONE” next to the C reference.

The initial register-to-variable mapping for each assembly code section is:

```
%eax = starting address of the NodeTree array
%edx = i
```

The assembly code sections are:

```
A.      sall $2, %edx               B.  sall $2,%edx
        leal (%eax,%edx),%eax           leal (%eax,%edx),%eax
        movl 16(%eax),%eax              movl (%eax),%eax
                                        movl 24(%eax),%eax
                                        movl 20(%eax),%eax
                                        movl 20(%eax),%eax

C:      sall $2,%edx                D:  sall $2,%edx
        leal (%eax,%edx),%eax           leal (%eax,%edx),%eax
        movl 20(%eax),%eax              movl (%eax),%eax
        movl 20(%eax),%eax              movl 16(%eax),%eax
        movsbl (%eax),%eax

E:      sall $2, %edx               F:  sall $2, %edx
        leal (%eax,%edx),%eax           leal (%eax,%edx),%eax
        movl (%eax),%eax                movl (%eax),%eax
        movl 16(%eax),%eax              movl 12(%eax),%eax
        movl 16(%eax),%eax              movl 12(%eax),%eax
        movl 20(%eax),%eax              movl 16(%eax),%eax
```

[Question continues on the next page]

[continued]

The C references:

`NodeTree[i]->flag` (2 points)

`NodeTree[i]->left->left->c` (2 points)

`NodeTree[i]->next->next->flag` (2 points)

`NodeTree[i]->right->left->left` (2 points)

Question 12 [17 points]

The following problem concerns the way virtual addresses are translated into physical addresses.

Make the following assumptions:

- The memory is byte addressable.
- Memory accesses are to **1-byte words** (*not* 4-byte words).
- Virtual addresses are 16 bits wide.
- Physical addresses are 13 bits wide.
- The page size is 512 bytes.
- The TLB is 8-way set associative with 16 total entries.
- 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 TLB, the page table for the first 32 pages, and the cache are as follows:

TLB			
Index	Tag	PPN	Valid
0	09	4	1
	12	2	1
	10	0	1
	08	5	1
	05	7	1
	13	1	0
	10	3	0
	18	3	0
1	04	1	0
	0C	1	0
	12	0	0
	08	1	0
	06	7	0
	03	1	0
	07	5	0
	02	2	0

Page Table					
VPN	PPN	Valid	VPN	PPN	Valid
00	6	1	10	0	1
01	5	0	11	5	0
02	3	1	12	2	1
03	4	1	13	4	0
04	2	0	14	6	0
05	7	1	15	2	0
06	1	0	16	4	0
07	3	0	17	6	0
08	5	1	18	1	1
09	4	0	19	2	0
0A	3	0	1A	5	0
0B	2	0	1B	7	0
0C	5	0	1C	6	0
0D	6	0	1D	2	0
0E	1	1	1E	3	0
0F	0	0	1F	1	0

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	19	1	99	11	23	11	00	0	99	11	23	11
1	15	0	4F	22	EC	11	2F	1	55	59	0B	41
2	1B	1	00	02	04	08	0B	1	01	03	05	07
3	06	0	84	06	B2	9C	12	0	84	06	B2	9C
4	07	0	43	6D	8F	09	05	0	43	6D	8F	09
5	0D	1	36	32	00	78	1E	1	A1	B2	C4	DE
6	11	0	A2	37	68	31	00	1	BB	77	33	00
7	16	1	11	C2	11	33	1E	1	00	C0	0F	00

[Question continues on the next page]

Name: ______________________________ Leginr: ____________________

[continued]

(3 points)

The box below shows the format of a virtual address. Indicate (by labeling the diagram) the fields (if they exist) that would be used to determine the following: (If a field doesn't exist, don't draw it on the diagram.)

VPO	The virtual page offset
VPN	The virtual page number
TLBI	The TLB index
TLBT	The TLB tag

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

(3 points)

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:

PPO	The physical page offset
PPN	The physical page number
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

[Question continues on the next page]

[continued]

For the given virtual address, indicate the TLB entry accessed, the physical address, and the cache byte value returned **in hex**.

Indicate whether the TLB misses, whether a page fault occurs, and whether a cache miss occurs.

If there is a cache miss, enter “-” for “Cache Byte returned”. If there is a page fault, enter “-” for “PPN” and leave parts C and D blank.

Virtual address: `1DDE`

(6 points)

Virtual address format (one bit per box):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Address translation:

Parameter	Value
VPN	0x
TLB Index	0x
TLB Tag	0x
TLB Hit? (Y/N)	
Page Fault? (Y/N)	
PPN	0x

[Question continues on the next page]

[continued]

(5 points)

Physical address format (one bit per box):

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

Physical memory reference:

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

Question 13 [10 points]

Consider a dynamic memory allocator that uses an implicit free list. Each memory block, either allocated or free, has a size that is a multiple of eight bytes. Thus, only the 29 higher order bits in the header and footer are needed to record block size, which includes the header and footer and is represented in units of bytes.

The usage of the remaining 3 lower order bits is as follows:

- `bit 0` indicates the use of the current block: 1 for allocated, 0 for free.
- `bit 1` indicates the use of the previous adjacent block: 1 for allocated, 0 for free.
- `bit 2` is unused and is always set to be 0.

Five helper routines are defined to facilitate the implementation of `free(void *p)`, and they are given below. For each routine, what it does is explained in the comment above the function definition.

For each routine, fill in the body of the helper routines the code section label that implement the corresponding functionality correctly. There are three choices for each routine.

```
/* given a pointer p to an allocated block, i.e., p is a
   pointer returned by some previous malloc()/realloc() call;
   returns the pointer to the header of the block */
void * header(void* p)
{
  void *ptr;

  ________________;
  return ptr;
}
```

Choices:

1. `ptr=p-1`
2. `ptr=(void *)((int *)p-1)`
3. `ptr=(void *)((int *)p-4)`

```
/* given a pointer to a valid block header or footer,
   returns the size of the block */
int size(void *hp)
{
  int result;

  ________________;
  return result;
}
```

Choices:

1. `result=(*hp)&(~7)`
2. `result=((*(char *)hp)&(~5))<<2`
3. `result=(*(int *)hp)&(~7)`

[Question continues on the next page]

[continued]

```
/* given a pointer p to an allocated block, i.e. p is
   a pointer returned by some previous malloc()/realloc() call;
   returns the pointer to the footer of the block */
void * footer(void *p)
{
  void *ptr;

  ________________;
  return ptr;
}
```

Choices:

1. `ptr=p+size(header(p))-8`
2. `ptr=p+size(header(p))-4`
3. `ptr=(int *)p+size(header(p))-2`

```
/* given a pointer to a valid block header or footer,
   returns the usage of the currect block,
   1 for allocated, 0 for free */
int allocated(void *hp)
{
  int result;

  ________________;
  return result;
}
```

Choices:

1. `result=(*(int *)hp)&1`
2. `result=(*(int *hp)&0`
3. `result=(*(int *)hp)|1`

```
/* given a pointer to a valid block header,
   returns the pointer to the header of previous block in memory */
void * prev(void *hp)
{
  void *ptr;

  ________________;

  return ptr;
}
```

Choices:

1. `ptr = hp - size(hp)`
2. `ptr = hp - size(hp-4)`
3. `ptr = hp - size(hp-4) + 4`

Question 14 [20 points]

This question is about hardware primitives for multiprocessor synchronization.

In the course we discussed the TAS (Test and Set) instruction, which atomically reads a word of memory, writes a '1' to the word, and returns the previous value stored at that location. TAS can be wrapped in a C function as follows:

```
int TAS(int *loc);
```

And can be used to implement mutual exclusion with a spinlock as follows:

```
void acquire(int *loc)
{
    while( TAS(loc) == 1);
}

void release(int *loc)
{
    *loc = 0;
}
```

For each of the synchronization primitives on the following pages, give the corresponding C prototype, description, and sketch functions for `acquire` and `release`.

You may assume:

- The memory system provides sequential consistency
- There is a function:

  ```
  int getpid(void);
  ```

  – which returns the unique identifier of the calling thread.
- This thread identifier will never be zero.
- For this question, you don't need to worry about contention for the memory system.

[Question continues on the next page]

[continued]

(7 points)

Compare-and-Swap (CAS):

C prototype:

Description:

Acquire the lock:

```
void acquire(                         )
{

}
```

Release the lock:

```
void release(                         )
{

}
```

[Question continues on the next page]

[continued]

(7 points)

Load-Linked and Store-Conditional (LL / SC):

C prototypes:

Description:

Acquire the spinlock:

```
void acquire(                    )
{




}
```

Release the spinlock:

```
void release(                    )
{




}
```

[Question continues on the next page]

[continued]

(6 points)

Now consider an "atomic increment" instruction, `ATOMICINC`, which atomically reads a value from a memory location, adds one to this value, stores the result back to the memory location, and returns the *new* value:

```
int ATOMICINC(int *loc)
```

Explain how to implement a spinlock with `acquire()` and `release()` functions as before, but using this synchronization instruction instead of the ones you used above.

Acquire the spinlock:

```
void acquire(                                          )
{

}
```

Release the spinlock:

```
void release(                         )
{

}
```

If implemented correctly, this new spinlock has a useful runtime property that the previous ones did not. What is it?