Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Department of Computer Science

Computer Architecture and Systems Programming

252-0061-00

Thursday 3rd February 2011

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 15 questions. The maximum number of points that can be achieved is 140.
- This exam paper consists of 27 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 the x86 cheat sheet that should be on your desk. If this is missing, please raise your hand.

Statement

- If you wish us to publish your results (grade) on a web page, then please sign the following statement.

Signature : _____

Name: Leginr:_____

Question 1 [10 points]

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

Fill in the empty boxes in the following table. `TMax` is the highest representable integer; `TMin` is the lowest representable number.

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		

Question 2 [13 points]

This question is about exceptional control flow in processors.

(a) [6 points] Explain in detail the sequence of events that occurs when a processor handles an exception.

(b) [3 points] Explain the difference between *asynchronous* and *synchronous* exceptions. Give an example of a class of *asynchronous exception*.

[Question continues on the next page]

(c) [4 points] Synchronous exceptions are generally classified into *traps*, which are intentional, and *faults*, which are not. Give one example of each type, and explain its purpose.

Question 3 [10 points]

In this question assume the variables `a` and `b` are signed integers and that the machine uses two's complement representation. Also assume that `TMax` is the maximum integer, `TMin` is the minimum integer, and `W` is one less than the word length (e.g., `W` = 31 for 32-bit integers).

Match each of the descriptions on the left with a line of code on the right: write the correct letter (a-i) next to each description (1-6).

1. The one's complement of `a`: ____

2. `a`. ____

3. `a & b` ____

4. `a * 7` ____

5. `a / 4` ____

6. `(a < 0) ? 1 : -1` ____

a. `~(~a | (b ^ (TMin + TMax)))`

b. `((a ^ b) & ~b) | (~(a ^ b) & b)`

c. `1 + (a << 3) + ~a`

d. `(a << 4) + (a << 2) + (a << 1)`

e. `((a < 0) ? (a + 3) : a) >> 2`

f. `a ^ (TMin + TMax)`

g. `~((a | (~a + 1)) >> W) & 1`

h. `~((a >> W) << 1)`

i. `a >> 2`

Question 4 [10 points]

Consider 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 in the corresponding C source code below.

You may only use the symbolic variables `h`, `t` and `len` in your expressions below — *do not use register names*.

Note: The opcode `jae` is "jump above or equal", in other words the unsigned version of `jge`.

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

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

      _______________;

      _______________;

      _______________;
   }

   return;
}
```

Question 5 [5 points]

This question is about cache lookups. Assume:

- 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.

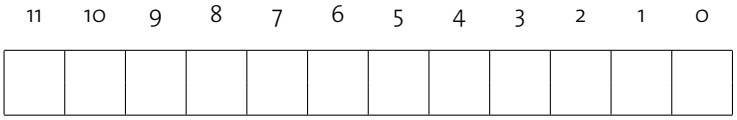
In the following table, **all numbers are given in hexadecimal**.

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

(a) [2 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:

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



[Question continues on the next page]

(b) [3 points] The processor tries to access the following physical address:

`0x3B6`

Show the cache entry accessed and the cache byte value returned **in hex**. Also, indicate whether a cache miss occurs.

You should show your calculation. You may find it helpful to fill in the following tables:

Physical address format (one bit per box):

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

Physical memory references:

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

Question 6 [10 points]

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>

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]

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]

This question uses the NukeJr program to test 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 times on the input string `"0 a ff"` would produce the following sequence of results:

  1. `val=0x0` and `scanf` returns 1.
  2. `val=0xa` and `scanf` returns 1.
  3. `val=0xff` and `scanf` returns 1.
  4. `val` is undefined and `scanf` returns EOF.

(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`

[Question continues on the next page]

(b) Fill in the empty boxes with an input string that would defuse NukeJr by causing the call to `overflow` to return to address `0x8048571` instead of `804856b`.

Notes:

- Your solution is allowed to trash the contents of the `%ebp` register.
- Each box should contain a one or two digit hex number (i.e. one character or byte).

0x00	0x00	0x00	0x00								

Question 7 [10 points]

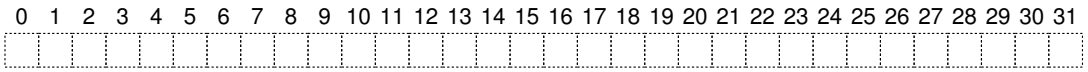
Consider the following C declaration:

```
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];
```

(a) [6 points] Using the template below (allowing a maximum of 32 bytes), indicate the allocation of data for a `Node` struct. Mark off and label the areas for each individual element (there are 6 of them). Cross hatch (i.e. mark with 'x') the parts that are allocated, but not used (to satisfy alignment).

Assume the standard 32-bit x86 Linux size and alignment rules: words and pointers are 32 bits, and elements are naturally aligned except that `double`s are aligned to 4-byte boundaries.

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



[Question continues on the next page]

(b) [4 points] Consider the following six Linux/ia32 assembly language sequences:

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

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

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

```
D:  sall $2,%edx
    leal (%eax,%edx),%eax
    movl (%eax),%eax
    movl 16(%eax),%eax
```

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

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

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

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

For each of the four C references below, please indicate which 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.

i. `NodeTree[i]->flag`

ii. `NodeTree[i]->left->left->c`

iii. `NodeTree[i]->next->next->flag`

iv. `NodeTree[i]->right->left->left`

Question 8 [10 points]

This question is about Input/Output devices.

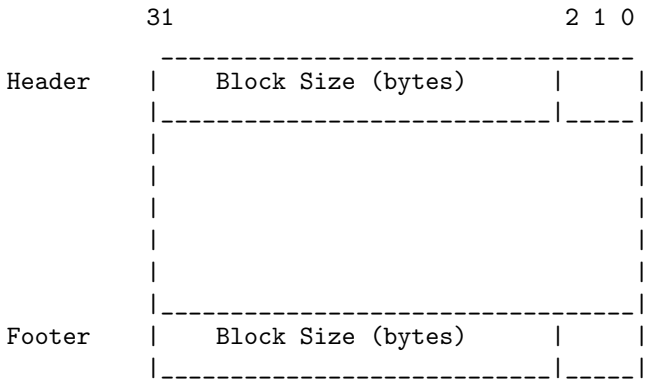
(a) [6 points] Explain the concept and operation of Direct Memory Access or DMA. What are the main advantages of DMA over Programmed I/O?

(b) [4 points] Explain how processor caches complicate the use of DMA, and what must be done to make DMA work in the presence of processor caches.

Question 9 [8 points]

This question is about dynamic storage allocation.

Consider a memory allocator that uses an implicit free list. The layout of each allocated and free memory block is as follows:



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.

The remaining 3 lower order bits are used 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.

[Question continues on the next page]

Given the contents of the heap shown on the left, show the new contents of the heap (in the right table) after a call to `free(0x400b010)` is executed.

Your answers should be given as hex values. Note that the address grows from bottom up. Assume that the allocator uses immediate coalescing, that is, adjacent free blocks are merged immediately each time a block is freed.

Address	
0x400b028	0x00000012
0x400b024	0x400b611c
0x400b020	0x400b512c
0x400b01c	0x00000012
0x400b018	0x00000013
0x400b014	0x400b511c
0x400b010	0x400b601c
0x400b00c	0x00000013
0x400b008	0x00000013
0x400b004	0x400b601c
0x400b000	0x400b511c
0x400affc	0x00000013

Address	
0x400b028	
0x400b024	0x400b611c
0x400b020	0x400b512c
0x400b01c	
0x400b018	
0x400b014	0x400b511c
0x400b010	0x400b601c
0x400b00c	
0x400b008	
0x400b004	0x400b601c
0x400b000	0x400b511c
0x400affc	

Question 10 [12 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 otherwise the same as in the IEEE standard, with regard to normalized and denormalized values, and representations of 0 (zero), infinity, and NAN.

As in IEEE standard format, 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				
Largest Denormalized				
$-\infty$		—	—	—
Number with hex representation `3AA0`	—			

Question 11 [8 points]

This question is about synchronization in multiprocessor systems.

(a) [2 points] Describe the operation of the Test-And-Set instruction for multiprocessor synchronization commonly found on modern processors.

(b) [2 points] Give C functions `acquire()` and `release()` to implement the simplest mutex spinlock using a Test-And-Set primitive. You may assume the existence of a function `int TAS(int *p)` which implements Test-And-Set.

[Question continues on the next page]

(c) [4 points] Explain why such a spinlock may result in poor performance in practice on real hardware, and describe an optimization technique which can solve most of this problem.

Question 12 [8 points]

This question is about optimizing a procedure for maximum performance on an Intel Pentium III. This processor has (or had) functional units with the following performance characteristics:

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

You've just joined a programming team that is trying to develop the world's fastest factorial routine. Starting with recursive factorial, they've converted the code to use iteration:

```
int fact(int n)
{
  int i;
  int result = 1;

  for (i = n; i > 0; i--)
    result = result * i;

  return result;
}
```

By doing so, they have reduced the number of cycles per element (CPE) for the function from approximately 63 to approximately 4 (really!). Still, they would like to do better.

One of the programmers heard about loop unrolling. He generated the following code:

```
int fact_u2(int n)
{
  int i;
  int result = 1;

  for (i = n; i > 0; i-=2) {
    result = (result * i) * (i-1);
  }

  return result;
}
```

Unfortunately, the team has discovered that this code returns 0 (zero) for some values of argument `n`.

[Question continues on the next page]

(a) [2 points] For what values of `n` will `fact_u2` and `fact` return different values?

(b) [2 points] Show how to fix `fact_u2` so that its behavior is identical to `fact`. [Hint: there is a special trick for this procedure that involves modifying just a single character.]

(c) [2 points] Benchmarking `fact_u2` shows no improvement in performance. How would you explain that?

(d) [2 points] You modify the line inside the loop to read:

```
result = result * (i * (i-1));
```

To everyone's astonishment, the measured CPE is now 2.5. How do you explain this performance improvement?

Question 13 [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
        pushl %ebx
        movl 8(%ebp),%ecx
        movl 12(%ebp),%ebx
        movl %ecx,%edx
        leal (%ebx,%ebx,8),%eax
        sall $4,%edx
        sall $2,%eax
        subl %ecx,%edx
        movl mat2(%eax,%ecx,4),%eax
        sall $2,%edx
        movl %eax,mat1(%edx,%ebx,4)
        movl -4(%ebp),%ebx
        movl %ebp,%esp
        popl %ebp
        ret
```

(a) What is the value of `M`?

(b) What is the value of `N`?

Question 14 [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 the following:

- `sizeof(char) = 1`
- `sizeof(int) = 4`
- `buffer` begins at memory address 0x00
- 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) [2 points] 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;
    }
}
```

[Question continues on the next page]

(b) [2 points] 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;
```

(c) [2 points] 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;
```

(d) [2 points] Which code (A, B, or C) should be the fastest?

Question 15 [10 points]

This problem concerns the way virtual addresses are translated into physical addresses. Assume:

- The memory is byte addressable.
- Memory accesses are to 4-byte words.
- Virtual addresses are 20 bits wide.
- Physical addresses are 16 bits wide.
- The page size is 4096 bytes.
- The TLB is 4-way set associative with 16 total entries.

In the following tables, **all numbers are given in hexadecimal**. The contents of the TLB and the page table for the first 32 pages are as follows:

TLB			
Index	Tag	PPN	Valid
0	03	B	1
	07	6	0
	28	3	1
	01	F	0
1	31	0	1
	12	3	0
	07	E	1
	0B	1	1
2	2A	A	0
	11	1	0
	1F	8	1
	07	5	1
3	07	3	1
	3F	F	0
	10	D	0
	32	0	0

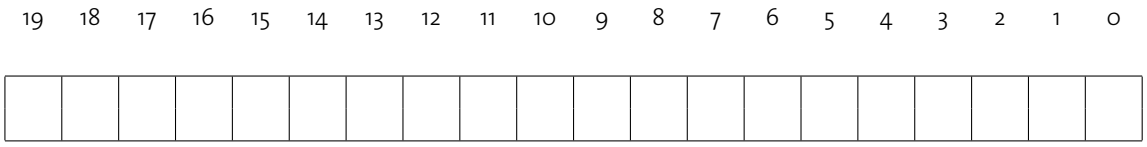
Page Table					
VPN	PPN	Valid	VPN	PPN	Valid
00	7	1	10	6	0
01	8	1	11	7	0
02	9	1	12	8	0
03	A	1	13	3	0
04	6	0	14	D	0
05	3	0	15	B	0
06	1	0	16	9	0
07	8	0	17	6	0
08	2	0	18	C	1
09	3	0	19	4	1
0A	1	1	1A	F	0
0B	6	1	1B	2	1
0C	A	1	1C	0	0
0D	D	0	1D	E	1
0E	E	0	1E	5	1
0F	D	1	1F	3	1

[Question continues on the next page]

(a) [5 points]

i. 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.)

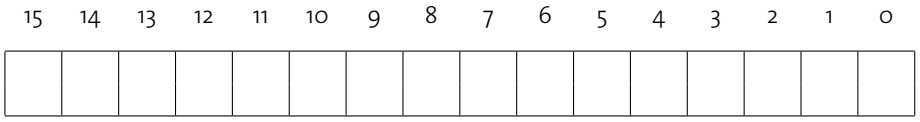
VPO The virtual page offset
VPN The virtual page number
TLBI The TLB index
TLBT The TLB tag



ii. 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



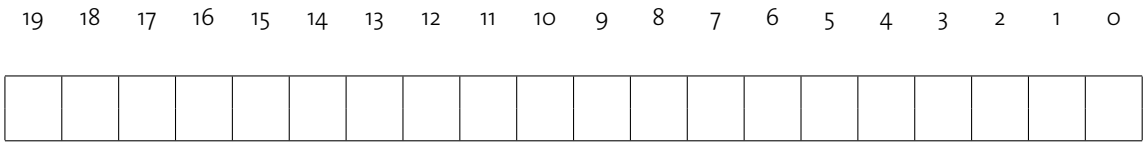
[Question continues on the next page]

(b) [5 points] For each of the given virtual addresses, indicate the TLB entry accessed and the physical address. Indicate whether the TLB misses and whether a page fault occurs.

If there is a page fault, enter "-" for "PPN" and leave the physical address blank.

i. **Virtual address**: `7E37C`

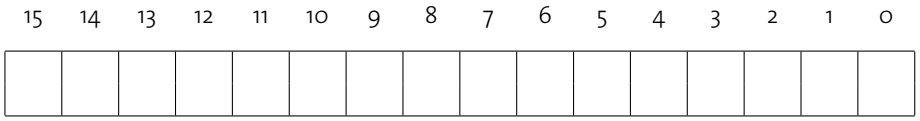
Virtual address format (one bit per box):



Address translation:

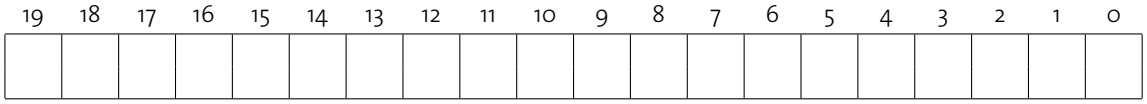
Parameter	Value
VPN	
TLB Index	
TLB Tag	
TLB Hit? (Y/N)	
Page Fault? (Y/N)	
PPN	

Physical address format (one bit per box):



ii. **Virtual address**: `16A48`

Virtual address format (one bit per box):



Address translation:

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

Physical address format (one bit per box):