Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Department of Computer Science

Last Name : ______________________

First Name : ______________________

Leginr. : ______________________

Systems Programming and Computer Architecture

Monday 4th February 2019

Rules

- You have 180 minutes for the exam.
- Please write your name and Legi-ID number on all sheets of paper.
- Please write in a blue or black pen. Do not use pencil.
- Please write your answers on the exam sheet. 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. Answers which cannot be read cannot be awarded points.
- The exam consists of 11 questions.
- The maximum number of points that can be achieved is 155.
- This exam paper consists of 31 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 electronic or 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.
- Justify all your answers, unless the question explicitly says “no justification needed”
- Mark all drawings precisely.

For examiners’ use only:

1	2	3	4	5	6	7	8	9	10	11

Total:

Name: ____________________ Leginr: ____________

Question 1 [15 points]

The following table gives the parameters for a number of different caches, where:

- m is the number of physical address bits
- C is the total cache size (number of data bytes)
- B is the block (line) size in bytes
- E is the number of blocks or lines per set (associativity)
- S is the number of sets in the cache.
- t is the number of tag bits
- s is the number of set index bits
- b is the number of block offset bits

Fill in the blank values in the following table:

(15 points)

Cache	m	C	B	E	S	t	s	b
Intel Skylake L3				16		29	13	6
Motorola MC68030 D-cache	32	256 bytes	16				4	
Apple A9 L2	48	3072kB	64		4096			
ARM Cortex-A8	32	32kB		4				5
Cavium ThunderX-1 L1D		32kB		32	8	38		

Question 2

[12 points]

What is meant by a **processor exception**?

(2 points)

Processor exceptions are generally divided into **synchronous** and **asynchronous** exceptions. Explain the difference.

(2 points)

The result of a **synchronous** exception can be one of four different outcomes, listed below.

For each one, explain why such an outcome might be appropriate, and give an example of an exception that would cause such a result.

1. The operating system can **resume** the process that was running by jumping to the machine instruction immediately **after** the one that was executing when the exception occurred (possibly running other processes in the meantime).

(2 points)

[Question continues on the next page]

[continued]

2. The operating system can **resume** the process that was running by **restarting** the machine instruction that was executing when the exception occurred (possibly running other processes in the meantime).

(2 points)

3. The operating system can **kill** or terminate the process causing the exception.

(2 points)

4. The hardware or the operating system can cause the entire machine to halt execution.

(2 points)

Question 3 [19 points]

Explain the difference between **strong** and **weak** linker symbols.

(4 points)

Now assume we have a 64-bit x86 machine, which is little-endian and therefore stores the least-significant byte of a word in the byte with the lowest address.

Assuming standard alignment rules for a 64-bit x86 Linux C compiler, sketch the layout in memory of an instance of the following struct definition:

```
struct membuf {
    char     valid;
    size_t   length;
    void    *buffer;
    unsigned int flags;
};
```

(4 points)

What is the value of `sizeof(struct membuf)`?

(2 points)

[Question continues on the next page]

[continued]

Consider the following two C files:

```
#include <stdio.h>

struct cs {
   int count;
   unsigned short flags;
};
struct cs gcount;
extern void add_counter( int n );

int main(int c, char *argv[])
{
    gcount.flags = 0xe700;
    gcount.count = 1;
    add_counter( 42 );
    printf("count = %d\n", gcount.count);
    return 0;
}
```

– and –

```
struct cs {
   unsigned short flags;
   int count;
};
struct cs gcount = { 0, 0 };

void add_counter( int n )
{
    gcount.count += n;
}
```

[Question continues on the next page]

[continued]

These two files compile and link successfully, without errors. However, the program does not work as the programmer originally expected. What is the output when the program is run? Explain why, in detail.

(5 points)

Why did the C compiler not report an error in this case?

(2 points)

Why did the linker not report an error in this case?

(2 points)

Name: ______________________ Leginr: ______________

Question 4 [10 points]

Consider the following recursive C function:

```
int shift_and_add( int n, int *p )
{
    int x;
    int y;

    if (n > 0) {
     y = shift_and_add( n << 1, &x);
    } else {
        x = y = 0;
    }
    *p = x + y + n;
    return x + y;
}
```

When compiled with the optimizer off, this results in the following assembly language:

```
shift_and_add:
    pushq %rbp
    movq %rsp, %rbp
    subq $32, %rsp
    movl %edi, -20(%rbp)
    movq %rsi, -32(%rbp)
    cmpl $0, -20(%rbp)
    jle .L2
    movl -20(%rbp), %eax
    leal (%rax,%rax), %edx
    leaq -8(%rbp), %rax
    movq %rax, %rsi
    movl %edx, %edi
    call shift_and_add
    movl %eax, -4(%rbp)
    jmp .L3
.L2:
    movl $0, -4(%rbp)
    movl -4(%rbp), %eax
    movl %eax, -8(%rbp)
.L3:
    movl -8(%rbp), %edx
    movl -4(%rbp), %eax
    addl %eax, %edx
    movl -20(%rbp), %eax
    addl %eax, %edx
    movq -32(%rbp), %rax
    movl %edx, (%rax)
    movl -8(%rbp), %edx
    movl -4(%rbp), %eax
    addl %edx, %eax
    leave
    ret
```

[Question continues on the next page]

[continued]

Note that, in this unoptimized code, the stack pointer is first copied to the base pointer (`%rbp`) before the stack frame is allocated.

Also, the `leave` mnemonic is equivalent to:

```
movq %rbp, %rsp
popq %rbp
```

At what byte offset relative to **the new value of the stack pointer** `%rsp` is the variable `x` stored?

(1 point)

Does it, in principle, need to be stored on the stack? Explain why, or why not.

(2 points)

At what byte offset relative to **the new value of the stack pointer** `%rsp` is the variable `y` stored?

(1 point)

Does it, in principle, need to be stored on the stack? Explain why, or why not.

(2 points)

[Question continues on the next page]

[continued]

What is being stored at the address pointed to by the new stack pointer?

(1 point)

Does it, in principle, need to be stored on the stack? Explain why, or why not.

(2 points)

What, if anything, is being stored at an offset of 8 from the new stack pointer?

(1 point)

Question 5 [13 points]

Consider a simple Ethernet network adaptor, which sends and receives packets (stored in memory) over a network.

The sending interface to the device consists of memory-mapped device registers and a single buffer descriptor ring.

The ring holds 256 descriptors in a contigous array in main memory. The format of each descriptor is as follows:

```
struct descriptor {
    uint64_t    buffer_addr; // Address of a packet in memory
    uint64_t    flags;       // Buffer and descriptor metadata
};
```

The `flags` field of the descriptor is formatted as follows:

- bit 8: 1 if descriptor is owned by the device, 0 otherwise.
- bit 9: 1 if the buffer has an error associated with it, 0 otherwise.
- bits 16-28: the length of received the data in the buffer.

All other bits are reserved: they should be set by software to zero, but the hardware might set them to any value

Sketch the layout of this descriptor in memory. Assume conventional 64-bit x86 Linux alignment rules for structures.

(2 points)

[Question continues on the next page]

[continued]

What is the value of `sizeof(struct descriptor)` ?

(1 point)

Fill in the body of the following function, which takes a pointer to a descriptor as an argument and returns a boolean value which is true if the descriptor is “free”, in other words it is owned by software and not by the device:

(1 point)

```
bool desc_is_free( struct descriptor *d)
{

}
```

Now fill in the body of the following function, which takes a pointer to a descriptor as an argument and returns the address in memory of the buffer that the descriptor refers to:

(2 points)

```
void *desc_buffer_addr( struct descriptor *d)
{

}
```

[Question continues on the next page]

[continued]

Finally, fill in the body of the following function, which takes a pointer to a descriptor as an argument, initializes it to refer to a buffer in memory at address `buffer` containing a packet of length `length`, and marks it as owned by the device.

The function should be as *robust* as possible; in other words, it should be a strict as possible in formatting the descriptor, but make no assumptions about the contents of the descriptor beforethe function is called.

(7 points)

```
void desc_send_buffer( struct descriptor *d, void *buffer, size_t length )
{

}
```

Question 6 [15 points]

A cache coherency protocol can be modelled as a state machine for each cache line (block) in main memory, where state transitions are signalled by messages arriving from the local core (or previous-level cache) and from remote caches.

For each of the following messages used by the MESI protocol, explain what operation on a cache it corresponds to.

PrRd (processor read):

(1 point)

PrWr (processor write):

(1 point)

BusRd (remote read):

(1 point)

BusRdX (remote read exclusive):

(1 point)

HIT:

(1 point)

[Question continues on the next page]

[continued]

Now fill in the following table. For each combination of starting state and message, fill in the corresponding cell with the new state that the cache line should transition to.

Note that in some cases it may be possible to transition to one of several states (depending on the state of other caches). Include all the possible states in your answer.

(10 points)

	PrRd	PrWr	BusRd	BusRdX	
M (Modified)					
E (Exclusive)					
S (Shared)					
I (Invalid)					

Question 7 [10 points]

Consider the following list of C syntax fragments:

1. `int *x;`
2. `int x[10];`
3. `(int *)x;`
4. `int *x[10];`
5. `int **x[10];`
6. `int (*x [10])[10];`
7. `int *x[10][10];`
8. `int x[10][5];`
9. `int x[5][10];`
10. `int (*x)[](int *);`
11. `int *x(int []);`
12. `int (*x[])(int *);`
13. `int *x(int *);`
14. `int *x[](int *);`
15. `(int (*)(int *, int))x;`
16. `int *x(int *, int);`
17. `(int (*)(int[], int))x;`
18. `int *x(int *, int)[];`
19. `(int *(*)(int *, int))x;`
20. `int (*x)(int *, int);`
21. `int *x[](int *, int);`

For each of the descriptions on the following page, give the number of the C syntax fragment that corresponds precisely to it.

[Question continues on the next page]

[continued]

Descriptions:

cast x into pointer to int:

(1 point)

declare x as array 10 of pointer to int:

(1 point)

declare x as array 10 of pointer to array 10 of int:

(1 point)

declare x as array 10 of array 5 of int:

(1 point)

declare x as pointer to array of function (pointer to int) returning int:

(1 point)

declare x as array of pointer to function (pointer to int) returning int:

(1 point)

[Question continues on the next page]

[continued]

declare x as array of function (pointer to int) returning pointer to int:

(1 point)

cast x into pointer to function (array of int, int) returning int:

(1 point)

cast x into pointer to function (pointer to int, int) returning pointer to int:

(1 point)

declare x as pointer to function (pointer to int, int) returning int:

(1 point)

Name: ______________________ Leginr: ______________

Question 8 [18 points]

Consider the following code to add up the elements of a matrix of floating point numbers

```
#define ROWS 256
#define COLS 64
float sum_matrix( float *f )
{
  float acc = 0.0;

  for( int c=0; c < COLS; c++) {
    for( int r=0; r < ROWS; r++) {
      acc += f[c + r*COLS];
    }
  }
  return acc;
}
```

This function compiles to the following machine code:

```
0000000000000000 <sum_matrix>:
   0: be 00 00 00 00        mov    $0x0,%esi
   5: 66 0f ef c0           pxor   %xmm0,%xmm0
   9: 89 f0                 mov    %esi,%eax
   b: 8d 8e 00 40 00 00     lea    0x4000(%rsi),%ecx
  11: 48 63 d0              movslq %eax,%rdx
  14: f3 0f 58 04 97        addss  (%rdi,%rdx,4),%xmm0
  19: 83 c0 40              add    $0x40,%eax
  1c: 39 c8                 cmp    %ecx,%eax
  1e: 75 f1                 jne    11 <sum_matrix+0x11>
  20: 83 c6 01              add    $0x1,%esi
  23: 83 fe 40              cmp    $0x40,%esi
  26: 75 e1                 jne    9 <sum_matrix+0x9>
  28: f3 c3                 repz retq
```

Suppose this code executes on a 64-bit x86 processor with a 1-kilobyte, direct-mapped data cache with 64 bytes per line/block.

Also assume that the function starts with a cold (i.e. empty) cache.

[Question continues on the next page]

[continued]

After the first 8 iterations through the **inner** loop (i.e. the first 8 rows of the first column of the matrix), how many cache misses have occurred? Show your working.

(2 points)

How many elements of the matrix are held in the cache at this point?

Show your working.

(2 points)

After the first full interation through the outer loop (i.e. one column of the matrix), how many cache misses have occurred? Show your working.

(2 points)

[Question continues on the next page]

[continued]

After the first **two** iterations through the outer loop, how many cache misses have occurred? Show your working.

(2 points)

How many of these are **compulsory** cache misses? Show your working.

(2 points)

[Question continues on the next page]

[continued]

Now suppose we instead traverse the matrix row-by-row, as follows:

```
#define ROWS 256
#define COLS 64
float sum_matrix( float *f )
{
  float acc = 0.0;

  for( int r=0; r < ROWS; r++) {
    for( int c=0; c < COLS; c++) {
      acc += f[c + r*COLS];
    }
  }
  return acc;
}
```

After the first 8 iterations through the **inner** loop (i.e. the first 8 columns of the first row of the matrix), how many cache misses have occurred? Show your working.

(2 points)

How many elements of the matrix are held in the cache at this point?

Show your working.

(2 points)

[Question continues on the next page]

[continued]

After the first **two** iterations through the outer loop, how many cache misses have occurred? Show your working.

(2 points)

How many of these are **compulsory** cache misses? Show your working.

(2 points)

Question 9

[9 points]

You have been asked to design the format of a page table for a 32-bit machine. The machine has 32-bit word size, so the Page Table Entry (PTE) size should be 32 bits and the virtual address space is 31 bits wide.

The page size is set at 512 bytes.

The page table has 2 levels, and decodes the 31-bit Virtual Address Space into a 32-bit wide Physical Address Space.

Assuming that the level 1 and level 2 page tables are the same size (and so decode the same number of bits of the virtual address), how many bits does each level decode? Explain why.

(2 points)

How large (in bytes) is a single page table at each level as a result?

(3 points)

[Question continues on the next page]

[continued]

Assuming that each page table is stored in memory at a "naturally aligned" physical address (i.e. an address which is an integer multiple of the size of the table), what is the **minimum** number of physical address bits that must be stored in the first (top) level Page Table Entry?

(2 points)

What is the **minimum** number of physical address bits that must be stored in the second-level Page Table Entry?

(2 points)

Question 10 [16 points]

Consider a floating point format which uses 8 bits but otherwise follows IEEE standard format. 4 bits are used for the fractional part, and 3 bits to represent the exponent.

The **bias** for this format is 3. Explain why.

(2 points)

What real number is represented by the binary value 10000000 in this system? Show your working

(2 points)

[Question continues on the next page]

[continued]

How is the real number 8 represented in binary in this system? Show your working.

(4 points)

What real number is represented by the binary value 11100000 in this system? Show your working

(4 points)

[Question continues on the next page]

[continued]

What number in this system is represented by the smallest positive denormalized value?

Give your answer as a decimal number, and show your working.

(4 points)

Question 11 [18 points]

Consider the following C function, where ARG is a decimal integer constant defined elsewhere:

```
#include <stdint.h>
int32_t multiply( int32_t a )
{
    return a * ARG;
}
```

The following disassemblies are of versions of the `multiply` function, compiled with different values for `ARG` and different optimization settings in the compiler.

Recall that the first (in this case, only) argument to a function is passed in the `rdi` register, and the result returned in the `rax` register.

For each one, say what the value of `ARG` was, and explain why.

```
0000000000000000 <multiply>:
   0: 55                    push   %rbp
   1: 48 89 e5              mov    %rsp,%rbp
   4: 89 7d fc              mov    %edi,-0x4(%rbp)
   7: 8b 45 fc              mov    -0x4(%rbp),%eax
   a: c1 e0 03              shl    $0x3,%eax
   d: 5d                    pop    %rbp
   e: c3                    retq
```

(3 points)

Answer:

[Question continues on the next page]

[continued]

```
0000000000000000 <multiply>:
   0: 89 f8                mov    %edi,%eax
   2: c1 e0 04             shl    $0x4,%eax
   5: 01 f8                add    %edi,%eax
   7: c3                   retq
```

(3 points)

Answer:

```
0000000000000000 <multiply>:
   0: 6b c7 f3             imul   $0xfffffff3,%edi,%eax
   3: c3                   retq
```

(3 points)

Answer:

[Question continues on the next page]

[continued]

```
0000000000000000 <multiply>:
   0: 89 f8                 mov    %edi,%eax
   2: c1 e0 07              shl    $0x7,%eax
   5: 29 f8                 sub    %edi,%eax
   7: c3                    retq
```

(3 points)

Answer:

```
0000000000000000 <multiply>:
   0: 89 f8                 mov    %edi,%eax
   2: f7 d8                 neg    %eax
   4: c3                    retq
```

(3 points)

Answer:

[Question continues on the next page]

[continued]

```
0000000000000000 <multiply>:
   0: 8d 04 bf            lea    (%rdi,%rdi,4),%eax
   3: 8d 04 47            lea    (%rdi,%rax,2),%eax
   6: c3                  retq
```

(3 points)

Answer: