Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Department of Computer Science

Last Name : ____________________

First Name : ____________________

Leginr. : ____________________

Systems Programming and Computer Architecture

Thursday 30th January 2020, 8:30

Rules

- 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 have 180 minutes for the exam.
- The exam consists of 15 questions.
- The maximum number of points that can be achieved is 160.
- 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.
- You are allowed no electronic or written aids, except for a German-English dictionary and the x86 reference sheet that should be on your desk. If this sheet is missing, please raise your hand.

For examiners' use only:

Question:	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Max pts:	6	7	11	10	20	9	11	7
Points:								

Question:	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Max pts:	8	8	16	6	13	20	8
Points:							

Total:

Name: ______________________ Leginr: ______________

Question 1 [6 points]

Consider the following C function:

```
int decide(int k, int t)
{
    int x = 1;
    do {
        switch(k) {
        case 1:
            x *= 3;
            break;
        case 6:
            k =- 5;
            break;
        case 2:
            x *= 5;
        case 3:
            x *= 7;
            break;
        case 4:
            x += 2;
        case 5:
            x /= 2;
        default:
            x <<= 1;
            continue;
        }
    } while ( x < t );
    return x;
}
```

For each of the following sets of arguments to `decide`, say what the return value of the function will be.

When `k = 3, t = 15`:

(1 point)

When `k = 2, t = 35`:

(1 point)

[Question continues on the next page]

[continued]

When `k = 0, t = 10`:

(1 point)

When `k = 5, t = 10`:

(1 point)

When `k = 4, t = 5`:

(1 point)

When `k = 6, t = 10`:

(1 point)

Question 2 [7 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.

(1 point)

```
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 04                shl    $0x4,%eax
   d: 5d                      pop    %rbp
   e: c3                      retq
```

Answer:

(1 point)

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

Answer:

[Question continues on the next page]

[continued]

(1 point)

```
0000000000000000 <multiply>:
   0: 55                    push   %rbp
   1: 48 89 e5              mov    %rsp,%rbp
   4: 89 7d fc              mov    %edi,-0x4(%rbp)
   7: 8b 55 fc              mov    -0x4(%rbp),%edx
   a: 89 d0                 mov    %edx,%eax
   c: 01 c0                 add    %eax,%eax
   e: 01 d0                 add    %edx,%eax
  10: c1 e0 02              shl    $0x2,%eax
  13: 01 d0                 add    %edx,%eax
  15: 5d                    pop    %rbp
  16: c3                    retq
```

Answer:

(1 point)

```
0000000000000000 <multiply>:
   0: 89 f8                 mov    %edi,%eax
   2: f7 d8                 neg    %eax
   4: c1 e0 04              shl    $0x4,%eax
   7: c3                    retq
```

Answer:

[Question continues on the next page]

[continued]

(1 point)

```
0000000000000000 <multiply>:
   0: 55                    push   %rbp
   1: 48 89 e5              mov    %rsp,%rbp
   4: 89 7d fc              mov    %edi,-0x4(%rbp)
   7: b8 00 00 00 00        mov    $0x0,%eax
   c: 5d                    pop    %rbp
   d: c3                    retq
```

Answer:

(1 point)

```
0000000000000000 <multiply>:
   0: 8d 04 bf              lea    (%rdi,%rdi,4),%eax
   3: 8d 04 c7              lea    (%rdi,%rax,8),%eax
   6: c3                    retq
```

Answer:

(1 point)

```
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: 6b c0 eb              imul   $0xffffffeb,%eax,%eax
   d: 5d                    pop    %rbp
   e: c3                    retq
```

Answer:

Question 3 [11 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:

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

(1 point)

“declare x as array 5 of array 10 of int”:

(1 point)

“declare x as array 10 of array 5 of int”:

(1 point)

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

(1 point)

“declare x as array 10 of int”:

(1 point)

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

(1 point)

[Question continues on the next page]

[continued]

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

(1 point)

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

(1 point)

"declare x as pointer to int":

(1 point)

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

(1 point)

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

(1 point)

Question 4

[10 points]

Consider the following `main` C function:

```
#include <stdio.h>
#include <stdint.h>

struct s1 {
    char     f1;
    uint32_t f2;
};
union u1 {
    uint32_t   f1;
    char       f2;
    union u1 *f3;
};
int main(int argc, char *argv[])
{
    char *p1 = (char *)(0x8000);
    printf("p1 = %p\n", p1);
    return 0;
}
```

When compiled and run, this program produces the following output:

```
p1 = 0x8000
```

For each of the following code fragments, say what the output will be when the `printf` statement in the body of the `main` function is replaced with the fragment.

- If a definite value is printed, say what it is.
- If an unpredictable value is printed, write *UNPREDICTABLE*
- If the program crashes, write *CRASH*
- If the program fails to compile, write *COMPILE-TIME ERROR*

(1 point)

```
printf("p1 + 1 = %p\n", p1 + 1);
```

(1 point)

```
uint32_t *p2 = ((uint32_t *)p1) + 4;
printf("p2 = %p\n", p2);
```

(1 point)

```
uint32_t *p3 = ((uint32_t *)p1) + 4;
printf("p3++ = %p\n", p3++);
```

[Question continues on the next page]

[continued]

(1 point)

```
uint16_t *p4 = ((uint16_t *)p1) - 4;
printf("p4 = %p\n", p4);
```

(1 point)

```
uint16_t *p5 = ((uint16_t *)p1) - 4;
printf("--p5 = %p\n", --p5);
```

(1 point)

```
struct s1 *p6 = ((struct s1 *)p1) + 5;
printf("p6 = %p\n", p6);
```

(1 point)

```
union u1 *p7 = ((union u1 *)p1) + 2;
printf("p7 = %p\n", p7);
```

(1 point)

```
struct s1 *a1 = (struct s1 *)p1;
printf("&a1[1] = %p\n", &a1[1]);
```

(1 point)

```
struct s1 *a2 = (struct s1 *)p1;
printf("&(a2->f2) = %p\n", &(a2->f2));
```

(1 point)

```
struct s1 a3[5] = (struct s1 *)p1;
printf("&a3[1] = %p\n", &a3[1]);
```

Question 5 [20 points]

Consider a simple I/O device which only receives variable-sized packets (at most 1024 bytes in size) from a network and transfers them to memory using a DMA-based descriptor ring set up by the device driver as part of the operating system. The descriptors are laid out as a contiguous array in memory, and so each descriptor has an index and an address in memory where it is stored. Descriptors in the ring have three fields:

owned: true if and only if the descriptor and its associated buffer are owned by the hardware device,

buffer address: the address in memory of the buffer corresponding to this descriptor,

size: either how large the buffer is (for empty buffers), or how much data was transferred into the buffer (for buffers containing a received packet).

Recall that both devices and device drivers can be in states where they are "Running" (there is processing they can perform right now, and are aware of this) or "Stopped" (they have stopped processing until notified that they can continue.

The OS initializes the descriptor queue and driver as follows:

Descriptors:

Descriptor number	Descriptor address	Buffer address	Size in bytes	Owned by device?
0	0x100000	0x20000	1024	True
1	0x100010	0x21000	1024	True
2	0x100020	0x22000	1024	True
3	0x100030	0x23000	1024	True

Device:

Next	0
State	Stopped
Overrun	False

Driver:

Next	0
State	Stopped

[Question continues on the next page]

[continued]

Now suppose the device receives a packet of size 100 bytes, followed quickly by another of size 200 bytes, and a third of 300 bytes size. The driver does not run in the meantime. Show the resulting state of the system in the following tables.

(8 points)

Descriptors:

Descriptor number	Descriptor address	Buffer address	Size in bytes	Owned by device?
0	0x100000			
1	0x100010			
2	0x100020			
3	0x100030			

Device:

Next	
State	
Overrun	

Driver:

Next	
State	

[Question continues on the next page]

[continued]

The driver now starts to execute, polling the queue to see if packets are waiting, and processing them in the order in which they were received before returning the buffer to the device. Fill in the fields to reflect the state of the system just after the driver proceses the **first** packet.

(6 points)

Descriptors:

Descriptor number	Descriptor address	Buffer address	Size in bytes	Owned by device?
0	0x100000			
1	0x100010			
2	0x100020			
3	0x100030			

Device:

Next	
State	
Overrun	

Driver:

Next	
State	

[Question continues on the next page]

[continued]

Before the driver can process another packet, the driver now receives 3 more packets, each of size 400 bytes. Fill in the following fields to reflect the state of the system now.

(6 points)

Descriptors:

Descriptor number	Descriptor address	Buffer address	Size in bytes	Owned by device?
0	0x100000			
1	0x100010			
2	0x100020			
3	0x100030			

Device:

Next	
State	
Overrun	

Driver:

Next	
State	

Question 6

[9 points]

Consider the following `main` C function:

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
    char s1[20] = "hello, world";
    char *s2    = "hi, enviromnent";

    // ...

    return 0;
}
```

For each of the following code fragments, say what new output is produced when we **add each new fragment successively** into the `main` function body just before the `return` statement.

```
    printf("sizeof(s1) = %lu\n", sizeof(s1));
```

(1 point)

```
    printf("sizeof(s2) = %lu\n", sizeof(s2));
```

(1 point)

```
    printf("strlen(s1) = %lu\n", strlen(s1));
```

(1 point)

```
    printf("strlen(s2) = %lu\n", strlen(s2));
```

(1 point)

[Question continues on the next page]

[continued]

```
s2 += 5;
printf("s2 = '%s'\n", s2);
```

(1 point)

```
printf("strlen(s2) = %lu\n", strlen(s2));
```

(1 point)

```
printf("s1[strlen(s1)] = %d\n", s1[strlen(s1)]);
```

(1 point)

```
*(s1 + 3) = '\0';
printf("s1 = '%s'\n", s1);
```

(1 point)

```
printf("s2 - 4 = '%s'\n", s2 - 4);
```

(1 point)

Question 7

[11 points]

The `alloca()` function has the following declaration:

```
#include <alloca.h>

void *alloca(size_t size);
```

The function allocates `size` bytes of memory in the stack frame of the caller, and returns a pointer to the start of this allocated space.

Does the space allocated by `alloca()` need to be freed explicitly? Explain why.

(2 points)

Why can `alloca()` *not* be implemented entirely as a function?

(6 points)

What happens to the space allocated by `alloca()` if `longjmp()` is used to jump out of the calling function? Explain why.

(3 points)

Question 8 [7 points]

What is the output of the following C program?

```
#include <stdio.h>
#define SWITCH_1 1
#define SWITCH_2 0

#if SWITCH_1
#define NAME    "Federer"
#else
#define NAME     "Wawrinka"
#endif

#ifdef SWITCH_2
#define VORNAME "Roger"
#undef NAME
#else
#define VORNAME "Stan"
#endif

#ifndef NAME
#define NAME  "Wattenhofer"
#endif

int main(int argc, char *argv[])
{
    printf("%s %s\n", VORNAME, NAME );
}
```

(2 points)

[Question continues on the next page]

[continued]

Give the definition of a C preprocessor macro `PRINT` which prints an arbitrary C expression together with its value at run time.

More precisely, the `PRINT` takes two arguments. The first is a format specifier as used by the format strings in standard C library `printf()` function. The second is an arbitrary C expression which should evaluate at compile time to a value compatible with the format specifier. Thus, for example, the fragment:

```
char s[10] = "hello, world";
int  x = 255;
PRINT(c,s[2]);
PRINT(x,x);
```

– will compile to code which, when executed, will result in the output:

```
s[2] = l
x = ff
```

You may find it helpful to use the C preprocessor `#str` syntax, which translates `str` into itself but quoted (`"str"`).

(5 points)

Question 9 [8 points]

Suppose the file `buffer.h` contains **only** the following declaration:

```
struct buffer {
    void    *data;
    size_t   length;
    char     flags[4];
};
```

On a 64-bit x86 Linux computer, what is `sizeof(struct buffer)`?

(2 points)

Now suppose that two other header files, `llist.h` and `hashtable.h`, both use the declaration of `struct buffer` and so include the line `#include "buffer.h"`. All is well until the programmer tries to compile the following file `.c` file:

```
#include "buffer.h"
#include "llist.h"
#include "hashtable.h"

int main(int argc, char *argv[])
{
    return 0;
}
```

How does the compilation fail, and why?

(2 points)

Show a change to **only** the file `buffer.h` which will fix this problem.

(4 points)

Question 10 [8 points]

For each listed identifier in the following C object file, say whether it is a **strong linker symbol**, a **weak linker symbol**, a **macro**, a symbol **local** to the compilation unit, or **none** of these.

```
#define string "hello, world"
static int count;
extern char *otherstring;
struct element {
    void           *data;
    struct element *next;
};
struct element *head = NULL;
struct element *tail;

void push( struct element *e )
{
    e->next = head;
    head    = e;
}
struct element *pull();
```

(8 points)

count:
element:
head:
otherstring:
pull:
push:
string:
tail:

Name: ______________________ Leginr: ______________

Question 11 [16 points]

Consider a floating point format which uses 10 bits but otherwise follows IEEE standard format. 5 bits are used for the fractional part, and 4 bits to represent the exponent.

Sketch the format of this number as bits, with the most significant bit on the left. Mark each bit as **S** for sign, **M** for mantissa, or **E** for exponent.

(2 points)

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|     |     |     |     |     |     |     |     |     |     |
|     |     |     |     |     |     |     |     |     |     |
|     |     |     |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

The **bias** for this format is 7. Explain why.

(2 points)

How is the real number 1/2 represented in binary in this system? Show your working.

(4 points)

[Question continues on the next page]

[continued]

What real number is represented by the binary value 1111000000 in this system? Show your working

(4 points)

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

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

(4 points)

Question 12

[6 points]

Suppose we have a 64-bit processor with the following memory system characteristics:

- 32-kilobyte, 8-way set-associative cache with 128 bytes per line/block
- 64-entry, 8-way set-associative TLB with a page size of 8192 bytes

Give the following values (in hexadecimal notation) corresponding to a virtual address of `0x12345678` in this system.

(6 points)

`Cache Index:`

`Cache Offset:`

`TLB Index:`

`TLB Tag:`

`Virtual Page Number (VPN):`

`Virtual Page Offset (VPO):`

Question 13 [13 points]

Suppose you are running the following function on a machine with a single 64-bit x86 core, with a 1-kilobyte, direct-mapped cache with 32 bytes per line/block.

```
#define ROWS 256
float sum_vector( float f[] )
{
  float acc = 0.0;

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

Assume a cold (empty) cache to start, and ignore misses due to instruction fetches or any other non-array accesses.

How many cache misses will this function incur when it is called with `f = 0x10000`? Show your working.

(2 points)

How many of these misses were compulsory?

(1 point)

How many of these misses were conflict misses?

(1 point)

[Question continues on the next page]

[continued]

We now run the following function on the same machine as before:

```
#define ROWS 256
void copy_vector( float s[], float d[] )
{
  for( int r=0; r < ROWS; r++) {
      d[r] = s[r];
  }
}
```

Again assuming a cold cache, how many cache misses will this function incur when it is called with `s = 0x10000` and `d = 0x20000`?

(2 points)

How many of these misses were compulsory?

(3 points)

How many of these misses were conflict misses?

(2 points)

Again assuming a cold cache, how many cache misses will this function incur when it is called with `s = 0x10000` and `d = 0x20080`?

(2 points)

Question 14

[20 points]

Suppose we have a cache-coherent computer with 3 processors, each of which has its own **write-back** data cache. The cache coherence protocol is MSI, and the machine provides **sequential consistency**.

Assume we start with empty caches, and main memory initialized to zero. At each time step, one of the cores executes an operation on the **same location in memory**, and no other interaction with the memory system occurs.

In the following table, fill in the blank entries for the cache line state, its contents as a result of the operation (or write `None` it is unknown or undefined), and the contents of the location in main memory.

(6 points)

Operations			Caches						Memory
time	core	operation	core 0 state	core 0 value	core 1 state	core 1 value	core 2 state	core 2 value	
0	-	-	I	None	I	None	I	None	0
1	0	store 42			I	None	I	None	
2	1	load			S	42	I	None	
3	2	store 52					M	52	
4	1	store 52							

[Question continues on the next page]

[continued]

Now consider the same machine before, but using a MESI protocol instead of MSI.

In the following table, fill in the blank entries as before, but note that you also need to fill in some of the "operations" fields as well.

(6 points)

Operations			Caches						Memory
time	core	operation	core 0 state	core 0 value	core 1 state	core 1 value	core 2 state	core 2 value	
0	-	-	I	None	I	None	I	None	0
1	2	load			I	None			
2	0	store 42			I	None			
3			S	42	S	42			
4	0	invalidate	I	None					

[Question continues on the next page]

[continued]

Finally, now suppose the machine uses a MESIF protocol instead of MESI or MSI. In the following table, fill in the blank entries as in the previous part of the question.

(6 points)

Operations			Caches						Memory
			core 0		core 1		core 2		
time	core	operation	state	value	state	value	state	value	
0	-	-	I	None	I	None	I	None	0
1	1	load	I	None			I	None	
2	0	store 42					I	None	
3	1	load					I	None	
4	2	store 52							
5	2	invalidate					I	None	

How would your answers change, if at all, if the machine implemented weak consistency instead of sequential consistency?

(2 points)

Name: ______________________ Leginr: ______________

Question 15 [8 points]

Consider the following, rather bad, C code:

```
#include <string.h>

/* copy string x to buf */
void foo(char *x)
{
    int buf[1];
    strcpy((char *)buf, x);
}

void callfoo()
{
    foo("abcdefghaaaabbbbccccdddd");
}
```

Here is the machine code this code assembles to on a 64-bit x86 Linux machine (with stack protection disabled):

```
000000000000064a <foo>:
64a: 55                    pushq  %rbp
64b: 48 89 e5              movq   %rsp,%rbp
64e: 48 83 ec 20           subq   $0x20,%rsp
652: 48 89 7d e8           movq   %rdi,-0x18(%rbp)
656: 48 8b 55 e8           movq   -0x18(%rbp),%rdx
65a: 48 8d 45 fc           leaq   -0x4(%rbp),%rax
65e: 48 89 d6              movq   %rdx,%rsi
661: 48 89 c7              movq   %rax,%rdi
664: e8 b7 fe ff ff        callq  520 <strcpy@plt>
669: 90                    nop
66a: c9                    leaveq
66b: c3                    retq

000000000000066c <callfoo>:
66c: 55                    pushq  %rbp
66d: 48 89 e5              movq   %rsp,%rbp
670: 48 8d 3d ad 00 00 00  leaq   0xad(%rip),%rdi        # 724 <_IO_stdin_used+0x4>
677: e8 ce ff ff ff        callq  64a <foo>
67c: 90                    nop
67d: 5d                    popq   %rbp
67e: c3                    retq
```

Recall that:

- `strcpy(char *dst, char *src)` copies the string at address `src` (including the terminating null character) to address `dst`, and does not check the size of the destination buffer.
- 64-bit Linux/x86 machines are Little Endian.
- The ASCII character code for 'a' is `0x61`.
- The `leaveq` instruction copies `rbp` to `rsp`, and then pops `rbp` off the stack.

Now consider what happens when `callfoo` calls `foo` with the string given in the code.

List the contents of the following memory locations immediately after `strcpy` returns to `foo`. Each answer should be an unsigned 4-byte integer expressed as 8 hex digits.

(2 points)

`*buf`

(2 points)

`*(buf+1)`

Immediately **before** the `retq` (return) instruction at the end of function `foo` executes, what is the value of the base pointer register `rbp`?

(2 points)

Immediately **after** the `retq` (return) instruction at the end of function `foo` executes, what is the value of the program counter register `rip`?

(2 points)