Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Department of Computer Science

Last Name : _____

First Name : _____

Leginr. : _____

Computer Architecture and Systems Programming

252-0061-00

Tuesday 20th January 2015, 9:00-12:00

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. 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 9 questions. The maximum number of points that can be achieved is 160.
- This exam paper consists of 23 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.

For examiners' use only:

1	2	3	4	5	6	7	8	9

Total: ____

Name: _____ Leginr: _____

Question 1 [12 points]

In the following question assume:

- `a` and `b` are declared as `int` in C.
- The machine uses two's complement format for signed numbers.
- `MAX_INT` and `MIN_INT` are the maximum and minimum representable signed integer values respectively
- `W` is one less than the number of bits needed to represent an `int` (i.e. `W == 31` on a machine with 32-bit `int`s).
- Right-shifts in C are arithmetic, not logical.

Match each of the descriptions on the left with a line of code on the right (write in the letter).

1. `a`.

_____

2. `a * 7`.

_____

3. One's complement of `a`

_____

4. `a / 4` .

_____

5. `a & b`.

_____

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

_____

a. `~(~a | (b ^ (MIN_INT + MAX_INT)))`

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 ^ (MIN_INT + MAX_INT)`

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

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

i. `a >> 2`

Question 2 [10 points]

Consider the following 5-bit floating point representation based on the IEEE floating point format:

- There is a sign bit in the most significant bit.
- The next two bits are the exponent. The exponent bias is 1.
- The last two bits are the significand.

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

The table below enumerates the entire non-negative range for this 5-bit floating point representation. Fill in the blank table entries using the following directions:

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

m**:** The fractional value of the mantissa. **Your answer must be expressed as a fraction of the form** $x/4$**.**

Value: The numeric value represented. **Your answer must be expressed as a fraction of the form** $x/4$.

You need not fill in entries marked "—".

(10 points)

Bits	E	m	Value
0 00 00	—	—	0
0 00 01			
0 00 10			
0 00 11			
0 01 00			
0 01 01			
0 01 10			
0 01 11			
0 10 00	1	$4/4$	$8/4$
0 10 01			
0 10 10			
0 10 11			

Name: _____ Leginr: _____

Question 3 [20 points]

Explain the concept and operation of Direct Memory Access or DMA, and contrast it with Programmed I/O. What problems are DMA trying to solve?

(6 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.

(4 points)

[Question continues on the next page]

[continued]

Modern high-speed I/O devices (such as networking adaptors or disk interfaces) use DMA in conjunction with rings of buffer descriptors. Explain briefly how buffer descriptor rings work.

(7 points)

What are the main advantages of buffer descriptor rings over the simple use of DMA for data transfer to and from a device?

(3 points)

Question 4

[12 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;
}
```

Without the optimizer enabled, this yields the following machine code:

```
silly:
        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    silly
        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
        movq    %rbp, %rsp
        popq    %rbp
        ret
```

[Question continues on the next page]

[continued]

Draw the stack frame used in this function, indicating where the program values are stored and where the`%rsp` and `%rbp` registers point just before the recursive call to `silly()`.

(8 points)

How much of this stack frame is, strictly speaking, necessary? In other words, how much could be optimized away by the compiler without changing the behavior of the program?

(4 points)

Name: _____ Leginr: _____

Question 5 [25 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
- 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 table.

(14 points)

Cache	m	C	B	E	S	t	s	b
1.	32	1kB	32			24	3	
2.	48	32kB	64	8	64	36		
3.	48	64kB	64	2		33		6
4.	48			4	64	36	6	
5.		48kB	32		512	36	9	5
6.	50	512kB	32		1024		10	5

[Question continues on the next page]

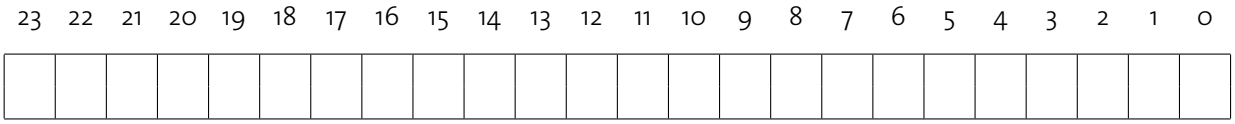
[continued]

Consider a machine with 24-bit physical addresses, and an 16-kilobyte, 2-way, physically-tagged, physically-addressed cache with a line (block) size of 64 bytes.

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

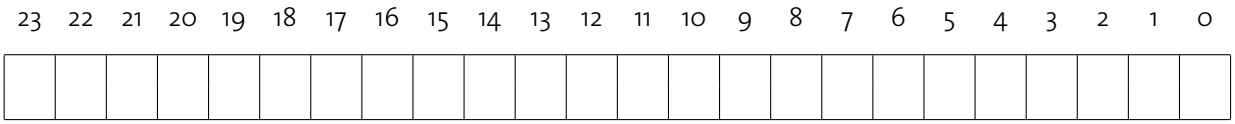
(3 points)



Now suppose the machine also has 24-bit virtual addresses, and a page size of 1 kilobyte. In the box below, now indicate the fields that would be used to determine the following:

PPO The physical page offset
PPN The physical page number

(2 points)



[Question continues on the next page]

[continued]

A program which makes heavy use of the cache can slow down other programs running on the same machine by constantly evicting their data from the cache. Suppose an operating system for the machine described above wished to restrict how much of the cache a particular user program was able to use (this technique is sometimes called "cache colouring").

Explain how, by careful use of virtual-to-physical page mappings in the page table and MMU, the Operating System could stop a particular program from using more than $1/8$ of the total cache capacity.

(6 points)

Question 6 [18 points]

The following assembler code is the result of compiling a function to multiply its `int` argument by a constant. What is the constant?

```
mul_x:
        movl    %edi, %eax
        sall    $14, %eax
        ret
```

(2 points)

What constant is the following function multiplying its argument by?

```
mul_y:
        movl    %edi, %eax
        sall    $4, %eax
        addl    %edi, %eax
        ret
```

(2 points)

What constant is the following function multiplying its argument by?

```
mul_z:
        leal    (%rdi,%rdi,2), %eax
        leal    (%rax,%rax,8), %eax
        ret
```

(2 points)

[Question continues on the next page]

[continued]

Now consider the following C function:

```
int div3(int i)
{
    return (i/3);
}
```

When compiled on a recent C compiler, the resulting assembly code was as follows:

```
div3:
      movl    %edi, %eax
      movl    $1431655766, %edx
      sarl    $31, %edi
      imull   %edx
      subl    %edi, %edx
      movl    %edx, %eax
      ret
```

Explain in detail how and why this assembly function works.

Note: the "`imull %edx`" instruction:

- multiplies `%eax` and `%edx`
- puts the upper 32 bits of the result into `%edx`
- puts the lower 32 bits of the result into `%eax`

(10 points)

[Question continues on the next page]

[continued]

Why do you think the compiler is generating code this complex in this case?

(2 points)

Question 7

[22 points] ☐

Describe the operation of the Compare-And-Swap (CAS) instructions found on some CISC processors like the x86 architecture (where they are known as `CMPXCHG8B` and `CMPXCHG16B`), and also explain why CAS can be difficult to implement efficiently in hardware.

(4 points)

Some uses of CAS suffer from what is known as the "ABA" problem. Explain what this problem is, and give a typical solution to the problem. (4 points)

[Question continues on the next page]

[continued]

RISC processors (such as the MIPS architecture) provide a different form of synchronization facility known as *Load-locked* (or *Load-linked*) and *Store-conditional*. It consists of two machine-code instructions:

LL Loads a value from a memory location memory into a register (as in `mov`) and "remembers" the address the value was loaded from.

SC Stores a value to a memory location **if** the location was previously loaded using **LL** on this core, **and** no other core has written to that location in the meantime. Otherwise, does nothing. If the store happened, return 1, otherwise return 0.

Suppose these instructions are provided to C code in the following functions:

```
uint64_t LL(uint64_t *location);
uint64_t SC(uint64_t *location, uint64_t value);
```

Using these functions, give the rough pseudo code for the following function which performs a Compare-And-Swap operation and returns the result on a 64-bit computer. You may assume the machine implements sequential memory consistency, and you can omit memory barriers and fences.

(3 points)

```
typedef uint64_t word;
word CAS(word cmp, word *location, word value)
{









}
```

[Question continues on the next page]

[continued]

Does your new CAS function suffer from the ABA problem that affects hardware CAS instructions? Explain why, or why not.

(2 points)

LL/SC is often implemented by building on the machine's cache coherency protocol. Consider a machine with a writeback cache and MESI (Modified, Exclusive, Shared, Invalid) cache coherence. Show a simple way in which LL/SC can be implemented by describing how the `LL` and `SC` instructions interact with the processor cache.

(6 points)

[Question continues on the next page]

[continued]

Almost all real implementations of `SC` provide a so-called "weak" form which is much easier to implement in hardware. In weak `SC`, the store can occasionally fail even if nothing else has modified the target location since the corresponding `LL` instruction. List some reasons why such "spurious failures" would occur.

(3 points)

Question 8 [16 points]

Most CPU architectures are classified as either *Big-endian* or *Little-endian* (though some can be configured as either).

Explain what *endianness* means when talking about computer architecture.

(2 points)

You are given a 64-bit machine (and a C compiler), but are not told whether the machine is Big-endian or Little-endian.

Write a short C function that uses casts and pointers (but no unions) to return 0 if the machine it runs on is Big-endian, and 1 if it is Little-endian.

(5 points)

[Question continues on the next page]

[continued]

Now write a different C function that uses `unions` (but no casts or pointers) to return 0 if the machine it runs on is Big-endian, and 1 if it is Little-endian.

(5 points)

[Question continues on the next page]

[continued]

Would your program work on a machine with a 32-bit word size?

(2 points)

Which of your two functions would you expect to be faster, if any, and why?

(2 points)

.

Question 9

[25 points] ☐

You are designing the Memory Mangement Unit (MMU) for a new processor. The processor has a word size of 64 bits and pointers are also 64 bits in size.

The following parts of the design have been decided already:

- The MMU will support a page size of 32kB.
- It will perform address translation using a 3-level hierarchical page table.
- As with x86 machines, each individual table at any level of the page table occupies a single page of physical memory. Level 3 is the root table, and contains entries which hold the physical addresses of Level 2 tables. Level 2 table entries similarly point to Level 1 tables, and Level 1 table entries point to physical pages or frames.
- All page table entries will be 64 bits in size.
- The Physical Address Space is limited to 52 bits (4096 terabytes) in size, which should be enough for the time being.

The rest of this question is about the consequences of these design decisions.

How many entries are there in each individual page table page? Show your working.

(2 points)

How many bits are required in the page table entry to represent the address of the next level of translation? Show your working.

(2 points)

How many bits of the virtual address are translated by each level in the page table? Why?

(2 points)

[Question continues on the next page]

[continued]

You have probably realized by now that the number of different virtual addresses that can be translated by this MMU is less than 2^{64}. What is the effective size of the Virtual Address Space of this processor? Show your working.

(2 points)

.

Assuming that this MMU deals with virtual addresses the same way as the x86 MMUs studied in the course, explain the mapping between full 64-bit pointer values and actual virtual addresses used by the MMU.

(2 points)

Why would the designers implement a smaller **physical** address space rather than the full 64 bits?

(2 points)

[Question continues on the next page]

[continued]

Why would the designers implement a smaller **virtual** address space rather than the full 64 bits?

(4 points)

Explain what "large pages" are in virtual address translation. Why might large pages be a good thing?

(5 points)

[Question continues on the next page]

[continued]

What sizes of large pages can be supported by your MMU, and why? Show your working.

(4 points)