Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Department of Computer Science

Last Name : _____

First Name : _____

Leginr. : _____

# Systems Programming and Computer Architecture

Tuesday 24th January 2023, 08:30

**Rules**

- This exam paper consists of 29 pages in addition to this title page. Please read through this paper to ensure that you have all the pages, and if not, raise your hand.
- You have 180 minutes for the exam.
- This paper consists of 15 questions, and the maximum number of points that can be achieved on this paper is 169.
- Write your answers on the exam sheet. If you need more paper, 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 your name and Legi-ID number on the first sheet of the paper, and on **all loose** sheets of paper that you hand in. Write in a blue or black pen. Do not use pencil.
- You are allowed no electronic or written aids, except for a German-English dictionary.

**For examiners' use; do not write in this space:**

| Question: | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |
|---|---|---|---|---|---|---|---|---|
| Max pts: | 12 | 11 | 14 | 12 | 8 | 10 | 6 | 10 |
| Points: | | | | | | | | |

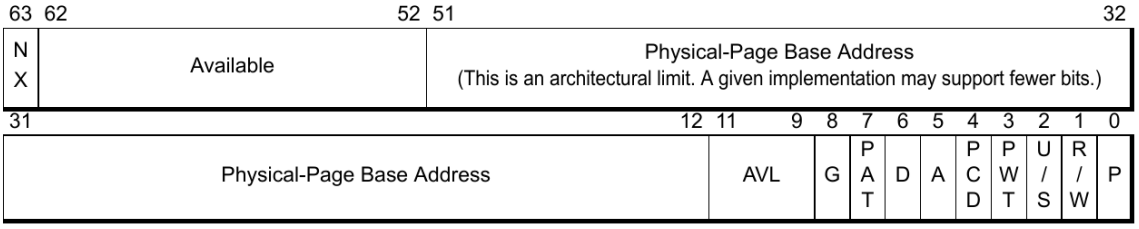
| Question: | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
|---|---|---|---|---|---|---|---|
| Max pts: | 12 | 12 | 12 | 18 | 10 | 10 | 12 |
| Points: | | | | | | | |

Total:

Name: _____ Leginr: _____

## Question 1 [12 points]

The following diagram shows the bitwise format of a Page Table Entry (PTE) for a 64-bit x86 processor:




Recall the following bit-field assignments:

**NX:** 1 ⇒ do not execute

**D:** 1 ⇒ page is dirty (reset by software)

**A:** 1 ⇒ page was accessed (reset by software)

**R/W:** 0 ⇒ read-only

**P:** 1 ⇒ page is present (PTE is valid)

On the following page is a list of C expressions in values `P`, `X`, and `Y`, all of which are of type `uint64_t`. `P` holds a *valid* (present) PTE formatted as shown above, while `X` and `Y` hold **small** ($<$ 100) unsigned integers.

For each of the following scenarios, give the letter corresponding to the correct expression, and the correct values of `X` and `Y`, to achieve the desired result.

(12 points)

| Scenario | Expression num. | `X` | `Y` |
|---|---|---|---|
| Return the Physical Frame Number (PFN) | | | |
| An operation which changes the PTE bits, but is guaranteed not to change the PTE meaning | | | |
| Return a True value if and only if the page has been read but not written. | | | |
| Ensure that data in the page can be read or written, but no code can be executed from it. | | | |

*[ Question continues on the next page ]*

*[continued]*

The possible expressions are as follows. Assume that any behavior specified as "implementation-defined" by the C standard follows the natural x86 machine instruction semantics.

A. `P & ( (1<<X)-1 ) >> Y`

B. `P << Y | X`

C. `P |= (1 << X) | Y`

D. `(P & X) == Y`

E. `P = (P & X) + Y`

F. `P = Y | (P << X)`

G. `(P & ( (1<<X)-1 )) | Y`

## Question 2 [11 points]

Consider the following C function, where the body of the `switch` statement has been omitted:

```
int swtch_1( int x, int y )
{
    switch(x) {
        ...
    }
}
```

Now suppose that every `case` clause in the `switch` statement, and the `default` clause, simply contains a `return` statement – for example:

```
case 42:  return(1789);
```

When compiled, it produces the following assembly language:

```
swtch_1:
    endbr64
    cmpl $5, %edi
    ja .L2
    movl %edi, %edi
    leaq .L4(%rip), %rdx
    movslq (%rdx,%rdi,4), %rax
    addq %rdx, %rax
    notrack jmp *%rax
.L4:
    .long .L2-.L4
    .long .L9-.L4
    .long .L7-.L4
    .long .L6-.L4
    .long .L5-.L4
    .long .L3-.L4
.L7:
    movl $384, %eax
    ret
.L6:
    movl $42, %eax
    ret
.L5:
    movl %esi, %eax
    sall $4, %eax
    addl %esi, %eax
    ret
.L3:
    leal 5(%rsi), %eax
    ret
.L2:
    movl $1973, %eax
    ret
.L9:
    movl $256, %eax
    ret
```

*[ Question continues on the next page ]*

*[continued]*

What is the value returned by the `default` clause?

(1 point)

In the following table, fill in a row for each `case` label in the code, giving the label and the return value. Assume that no `case` clause returns the same value as the default.

(10 points)

| `case` label | `return` expression |
|---|---|
| | |

Name: _____ Leginr: _____

## Question 3 [14 points]

Put a check mark in the box corresponding to the correct answer(s).

a) In the following code, what order of loops exhibits the best locality?

```
 // int a[X][Y][Z] is declared earlier int i, j, k,
sum = 0;
for (i = 0; i < Y; i++)
    for (j = 0; j < Z; j++)
       for (k = 0; k < X; k++)
          sum += a[k][i][j];
```

(2 points)

- ☐ i on the outside, j in the middle, k on the inside (as is).
- ☐ j on the outside, k in the middle, i on the inside.
- ☐ k on the outside, i in the middle, j on the inside.
- ☐ The order does not matter.

b) Consider the C declaration `int array[8] = {0, 1, 2, 3, 4, 5, 6, 7};`
Suppose that the compiler has placed the variable `array` in the `%ecx` register. How do you move the value at `array[6]` into the `%eax` register? Assume that `%ebx` is 6.

(2 points)

- ☐ `leal 24(%ecx), %eax`
- ☐ `leal (%ecx, %ebx, 4), %eax`
- ☐ `movl (%ecx, %ebx, 4), %eax`
- ☐ `movl 8(%ecx, %ebx, 1),%eax`
- ☐ `leal 4(%ecx, %ebx, 1),%eax`

c) Which of the following is a good reason to use dynamic memory in a C program?

(2 points)

- ☐ Allocating dynamic memory is significantly faster than pushing/popping the stack.
- ☐ Dynamic memory is more secure; the stack is prone to corruption from buffer overflow attacks.
- ☐ Dynamic memory has automatic garbage collection.
- ☐ None of the above.

*[ Question continues on the next page ]*

*[continued]*

d) Which of the following techniques can help protect against buffer overflow vulnerabilities? Select all that apply.

(2 points)

- ☐ Randomize stack offsets by allocating a random amount of space on the stack at the start of a program.
- ☐ Enable Address Space Layout Randomization (ASLR).
- ☐ When copying strings in C, use the `strcpy` library function (which copies the full string) instead of `strncpy` (which copies the first n bytes of the string).
- ☐ Mark memory regions that do not contain the original program code as non-executable.

e) Which of the following are benefits of using dynamic vs. static libraries?

(2 points)

- ☐ Changes to the library do not require recompiling all programs that use the library.
- ☐ Library code can be shared by multiple processes; the OS can map different program virtual addresses to the same physical address to share pages with library code.
- ☐ Self-contained program executables.
- ☐ Smaller size of program executables.
- ☐ Most programs will run noticeably faster.

f) Virtual memory can be viewed as a mechanism for using DRAM as a cache for disk storage. How would you describe this DRAM cache?

(2 points)

- ☐ Fully associative, shared, write-through
- ☐ Fully associative, shared, write-back
- ☐ 2-way set associative, exclusive, write-back
- ☐ Direct-mapped, inclusive, write-through

g) Using larger memory pages has which of the following effects compared to using smaller pages:

(2 points)

- ☐ Reduces the size of the page table
- ☐ Increases the address translation latency
- ☐ Reduces internal fragmentation
- ☐ Increases the coverage of memory addresses for a given TLB size

Name: _____ Leginr: _____

## Question 4 [12 points]

Consider the generic `swap` function below. It is used to make two values of a particular size trade places in memory.

```
void swap(void *a, void *b, size_t sz)
{
    char tmp[sz];
    memcpy(tmp, a, sz);
    memcpy(a, b, sz);
    memcpy(b, tmp, sz);
}
```

Note the following when using this function in this question:

- The third argument to swap is the return value of sizeof. Complete it with the name of a standard type.
- Do not move/change any memory outside the boxes shown in the diagram.
- Casting pointers is allowed.
- The declaration of memcpy is:

```
void *memcpy(void *dest, const void *src, size_t n)
```

Assume `int* ptr` points to element 0 of the following integer array in memory:

| 1 | -1 | 0 | 16 |
|---|---|---|---|

Complete the `mixup1` function below such that the contents of the integer array will be the following after the function returns. Assume the code executes on a little-endian machine.

| 255 | -255 | 0 | 16 |
|---|---|---|---|

(6 points)

```
void mixup1(int *ptr)
{

    swap(_________________________________________________,

         _________________________________________________,

         sizeof(_____________________________________________));
}
```

*[ Question continues on the next page ]*

*[continued]*

Complete the `mixup2` function below such that the contents of the integer array will be the following after the function returns. Assume the code executes on a little-endian machine and `mixup2` executes after a single call to `mixup1`. Assume `int* ptr` still points to element 0 of the integer array.

| 255 | -255 | 0 | 4096 |
|---|---|---|---|

(6 points)

```
void mixup2(int *ptr)
{

    swap(_________________________________________________,

         _________________________________________________,

         sizeof(_____________________________________________));
}
```

Name: _____ Leginr: _____

## Question 5 [8 points]

Consider the following code for matrix multiplication with a blocking factor *B*. Throughout this question, assume `n` is greater than 512.

```
/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
  int i, j, k, i1, j1, k1;
    for (i= 0; i< n; i+=B)
      for (j = 0; j < n; j+=B)
        for (k = 0; k < n; k+=B)
          for (i1 = i; i1 < i+B; i1++)
            for (j1 = j; j1 < j+B; j1++)
              for (k1 = k; k1 < k+B; k1++)
                c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
```

a) Assuming the L2 data cache size is 64MB ($2^{26}$ bytes), what is the optimal blocking factor *B*? Show your work. It is sufficient to provide an expression (e.g., $B = 226/\sqrt{7}$); you do not need to calculate the exact value of *B*.

(4 points)

*[ Question continues on the next page ]*

*[continued]*

b) Now assume the TLB fits 128 page table entries and the operating system uses 4KB ($2^{12}$ bytes) pages. What is the maximum value for *B* that would avoid TLB thrashing within a blocking iteration? A 'blocking iteration' is defined as a completion of the inner loops i1, j1, and k1. It is sufficient to provide an expression, you do not need to calculate the exact value.

(4 points)

## Question 6 [10 points]

a) How many bits are required per entry for a TLB that has the following characteristics:

- Virtual addresses are 32 bits wide
- Physical addresses are 42 bits wide
- The page size is 4K bytes
- The TLB contains 32 entries of the page table
- The TLB is 4-way associative.
- For simplicity, assume the metadata per entry consists of 1 valid bit, 1 dirty bit, and a 4-bit address space identifier.

(5 points)

*[ Question continues on the next page ]*

*[continued]*

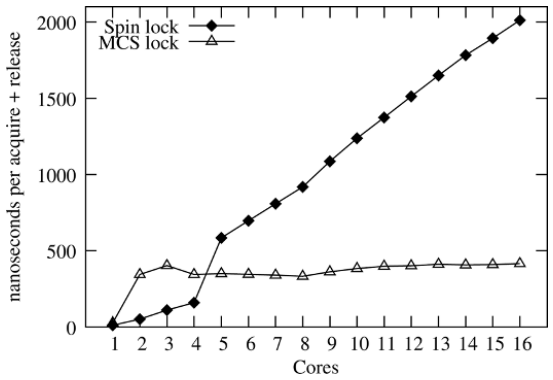
b) Consider a system with 32-bit virtual addresses, 42-bit physical addresses, 4KB page sizes, and four levels of page tables. Each level of page tables has 32 entries. How many physical memory *pages* are used for page tables on the machine if 10 processes are concurrently running on the machine and *each process* is accessing exactly 64KB of contiguous memory in its virtual address space? Assume that each of the processes access different physical memory. Assume each page table is on a separate physical page.

(5 points)

Name: _____ Leginr: _____

## Question 7 [6 points]

Recall two different lock implementations we discussed for multicore processors: the spin lock and the Mellor-Crummey and Scott (MCS) lock. The following graph shows the latency for acquiring and releasing each type of lock as a function of the number of contending cores:




For reference, below is the code for acquiring and releasing a spin lock:

```
void acquire_spinlock( int *lock) {
    while (TAS(lock) == 1)
      ;
}

void release_spinlock(int *lock) {
    *lock = 0;
}
```

For reference, below is the code for acquiring and releasing a MCS lock:

```
struct qnode {
    struct qnode *next;
    int locked;
};
typedef struct qnode *lock_t;

void acquire_mcs(lock_t *lock, struct qnode *local) {
    local->next = NULL;
    struct qnode *prev = XCHG(lock, local);
    if (prev) {  // queue was non-empty
        local->locked = 1;
        prev->next = local;
        while (local->locked)
            ;  // spin
    }
}
```

*[ Question continues on the next page ]*

*[continued]*

```
void release_mcs(lock_t *lock, struct qnode *local) {
    if (local->next == NULL) {
        if (CAS(lock, local, NULL)) {
            return;
        }
        while (local->next == NULL)
            ; // spin
    }
    local->next->locked = 0;
}
```

a) Explain why MCS locks do not suffer from the performance collapse that spin locks do when many cores are contending for the same lock.

(4 points)

b) Explain why acquiring and releasing an MCS lock takes more time than acquiring and releasing a spinlock when there are only a few cores contending for the lock.

(2 points)

Name: _____ Leginr: _____

## Question 8 [10 points]

Consider a 64-bit processor architecture is not x86, but which uses the same C data type sizes and alignment rules as 64-bit x86 Linux. The procedure calling conventions enforced by the C compiler for this machine are as follows:

- The stack pointer always points to the last element pushed onto the stack.
- No data is stored beyond the stack top, i.e. there is no “red zone”
- All function arguments are pushed onto the stack in the order in which they are declared.
- All local variables are allocated space on the stack, in the order in which they are declared.
- There is no padding in the stack frame apart from that needed for alignment.

Now consider the following code running on this machine:

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

struct list {
    uint16_t ar[4];
    struct list *next;
};

int64_t link( struct list sp, int64_t v )
{
    uint16_t *p = &sp.ar[3];
    int64_t i, t = v;

    for( i=0; i<4; i++)
        t += sp.ar[i];
    sp.next = &sp;
    return t;
}

int main(int argc, char *argv[])
{
    struct list el = { { 0, 0, 0, 0, }, NULL };

    link( el, 42 );
    return 0;
}
```

*[ Question continues on the next page ]*

*[continued]*

Each row of the following diagram represents the top seven 64-bit fields of the stack, just **before** the function `spfunc` returns, i.e. just after `sp.next` is assigned. Two fields are already filled in.

- For each field with a C expression stored in it, write this expression in the "identifier" column.
- If the field holds a **return address**, write `return address` in the "identifier" column.
- If the value of the field is a **pointer to another location in the stack**, fill in the "value" column with its value *relative to the stack pointer* (note that since the stack pointer always points to the top of the stack, this will always be positive).
- If the field is an **integer** whose value is known, write this value in the "value" column.
- If the value of the field is **not known**, write `unknown` in the "value" field.

(10 points)

| Address (increases upwards) | Identifier | Value |
|---|---|---|
| stack pointer + 48 → | | |
| stack pointer + 40 → | | |
| stack pointer + 32 → | | |
| stack pointer + 24 → | | |
| stack pointer + 16 → | | |
| stack pointer + 8 → | | |
| stack pointer → | `i` | 4 |

## Question 9 [12 points]

Consider the following `main` C function, compiled and executed on a contemporary 64-bit Linux machine:

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

int main(int argc, char *argv[])
{
    char s1[100] = "Simonetta Sommaruga";
    char s2[100] = "Guy Parmelin";
    char *s3     = "Ignazio Cassis";

    // fragment goes here

    return 0;
}
```

For each of the following code fragments, say what output is produced when the comment is replaced by the given fragment:

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

(1 point)

```
printf("sizeof(s2) = %lu\n", sizeof(s1));
```

(1 point)

```
printf("sizeof(s3) = %lu\n", sizeof(s3));
```

(1 point)

```
printf("strlen(s3) = %lu\n", strlen(s3));
```

(1 point)

*[ Question continues on the next page ]*

*[continued]*

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

(1 point)

```
s3 += 4;
printf("s3 = '%s'\n", s3);
```

(1 point)

```
*(s1 + 4) = '\0';
printf("s1 = '%s'\n", s1);
```

(1 point)

```
*(s2 + 6) = 0;
printf("strlen(s2) = %lu\n", strlen(s2));
```

(1 point)

What happens if the comment is replaced by the following fragment? Explain why in detail.

```
*(s3 + 13) = 0;
printf("strlen(s3) = %lu\n", strlen(s3));
```

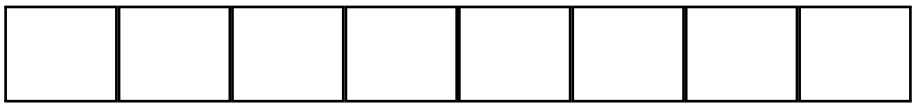
(4 points)

## Question 10 [12 points]

Consider a floating point format which uses 8 bits but otherwise follows IEEE standard format. 3 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.

(1 point)



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

(1 point)

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

(4 points)

*[ Question continues on the next page ]*

*[continued]*

Now consider the problem of rounding the integer 763 to this format. In binary, the number 763 is 1011111011.

In the following diagram, label each of the bits of the number 763 as follows:

**1** if the bit is an implicit leading 1

**G** if the bit is a guard bit

**B** if the bit is not a guard and is unchanged after rounding

**R** if the bit is a round bit

**S** if the bit is a part of the sticky bit

(6 points)

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |

Name: _____ Leginr: _____

## Question 11 [12 points]

Consider the following program:

```
int main(int argc, char *argv[])
{
    void *p = malloc( 100 );
    void *q = malloc( 200 );
    free( p );
    void *r = malloc( 50 );
    free( r );
    return 0;
}
```

In the rest of the question, assume the following:

- The program executes correctly
- No statements are optimized out or reordered by the compiler
- The standard C library is used
- The initial heap size is zero, and it grows monotonically on demand in increments of 1024 bytes for each `sbrk`.
- All heap operations complete successfully, and memory is not exahusted.

You should also ignore internal fragmentation and overhead in your calculations.

What is the **aggregate payload** at the end of the program (just before it exits)?

(2 points)

What is the minimum **heap size** at the end of the program (just before it exits)?

(2 points)

*[ Question continues on the next page ]*

*[continued]*

Now consider the following program, under the same conditions and assumptions as before:

```
int main(int argc, char *argv[])
{
    void *p = malloc( 100 );
    void *q = calloc( 32, 8 );
    p = realloc( p, 256 );
    free( q );
    void *r = malloc( 128 );
    return 0;
}
```

What is the aggregate payload at the end of the program (just before it exits)?

(2 points)

What is the **peak utilization** of the heap in this program?

(4 points)

What happens to the **virtual memory** used by the heap when the program exists?

(2 points)

Name: _____ Leginr: _____

## Question 12 [18 points]

For each statement, state whether it is TRUE of FALSE. If the statement is FALSE, explain why and correct the statement. Note: no partial points will be given if the explanation provided is not correct.

a) Multi-level page tables are an address translation latency optimization.

(2 points)

b) A fence/barrier executed on one processor only affects the order in which reads/writes on that processor are executed and committed to memory; it does not affect the ordering of reads/writes on other processors.

(2 points)

c) The compiler ensures that procedure arguments and local variables are always stored in memory at addresses that correspond to the procedure's stack frame.

(2 points)

*[ Question continues on the next page ]*

*[continued]*

d) The only way the compiler can add or remove data on the stack on x86 processors is through `push` and `pop` instructions.

(2 points)

e) When libc's malloc implementation needs to request more memory from the operating system, the system call it issues is an example of a synchronous exception on the processor.

(2 points)

f) Device registers are areas of memory that only a device can read or write to, not the processor.

(2 points)

*[ Question continues on the next page ]*

*[continued]*

g) False sharing of a cache line is good for performance when accesses to the cache line are from different processors and bad for performance when accesses to the cache line are on the same processor.

(2 points)

h) The operating system maintains one page table per NUMA node for virtual memory translation.

(2 points)

i) The MESI cache coherence protocol improves on the MSI protocol by adding the Exclusive state, which reduces bus traffic when the only processor that is updating a particular cache block wants to write the block back to main memory.

(2 points)

## Question 13 [10 points]

Which of the following code optimizations can reduce the number of instructions that the processor needs to execute for a program? For each, write YES or NO and **explain your reasoning**. A correct answer with no explanation will receive no points.

(10 points)

i) Code inlining.

ii) Code motion.

iii) Loop unrolling.

iv) Reassociation.

v) Strength reduction.

Name: _____ Leginr: _____

## Question 14 [10 points]

Consider the following code running on three processors sharing main memory:

| **Processor 1** | **Processor 2** | **Processor 3** |
|---|---|---|
| *x = 1 | *y = 2 | *z = 3 |
| a = *z | b = *x | *x = 3 |
| *y = 1 | *z = 2 | c = *y |

Assume that the values in memory at addresses x, y, and z are initially 0 and `int` a = b = c = 0 before any of the processors begin execution of the above code.

a) Which of the following results are possible under Sequential Consistency? Select all that apply.

(5 points)

- ☐ `a=0, b=0, c=0`
- ☐ `a=0, b=1, c=2`
- ☐ `a=3, b=1, c=1`
- ☐ `a=3, b=1, c=2`
- ☐ `a=0, b=3, c=0`

b) Which of the following results are possible under Total Store Order? Select all that apply.

(5 points)

- ☐ `a=0, b=0, c=0`
- ☐ `a=0, b=1, c=2`
- ☐ `a=3, b=1, c=1`
- ☐ `a=3, b=1, c=2`
- ☐ `a=0, b=3, c=0`

Name: _____ Leginr: _____

## Question 15 [12 points] ☐

In the following question assume:

- `a` and `b` are declared as `int` in C.
- The machine uses 32-bit two's complement format for signed `int`s.
- `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`).

For each of the descriptions in the left column, write in the letter of the corresponding expression in the right column.

(12 points)

1. `a` ☐
2. `a * 7` ☐
3. One's complement of `a` ☐
4. `a / 4` ☐
5. `a & b` ☐
6. `(a < 0) ? 1 : -3` ☐

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`

# x86-64 Reference Sheet (GNU assembler format)

## Instructions

### Data movement

| | |
|---|---|
| `movq` Src, Dest | Dest ← Src |
| `movsbq` Src,Dest | Dest (quad) ← Src (byte), sign-extend |
| `movzbq` Src,Dest | Dest (quad) ← Src (byte), zero-extend |

### Conditional move

| | |
|---|---|
| `cmove` Src, Dest | Equal / zero |
| `cmovne` Src, Dest | Not equal / not zero |
| `cmovs` Src, Dest | Negative |
| `cmovns` Src, Dest | Nonnegative |
| `cmovg` Src, Dest | Greater (signed >) |
| `cmovge` Src, Dest | Greater or equal (signed ≥) |
| `cmovl` Src, Dest | Less (signed <) |
| `cmovle` Src, Dest | Less or equal (signed ≤) |
| `cmova` Src, Dest | Above (unsigned >) |
| `cmovae` Src, Dest | Above or equal (unsigned ≥) |
| `cmovb` Src, Dest | Below (unsigned <) |
| `cmovbe` Src, Dest | Below or equal (unsigned ≤) |

### Control transfer

| | |
|---|---|
| `jmp` Label | jump |
| `je` Label | jump equal |
| `jne` Label | jump not equal |
| `js` Label | jump negative |
| `jns` Label | jump non-negative |
| `jg` Label | jump greater (signed >) |
| `jge` Label | jump greater or equal (signed ≥) |
| `jl` Label | jump less (signed <) |
| `jle` Label | jump less or equal (signed ≤) |
| `ja` Label | jump above (unsigned >) |
| `jb` Label | jump below (unsigned <) |
| `jmp` *Src | jump indirect (register) |
| `pushq` Src | %rsp ← %rsp – 8, Mem[%rsp] ← Src |
| `popq` Dest | Dest ← Mem[%rsp], %rsp ← %rsp + 8 |
| `call` Label | push addr next instruction, `jmp label` |
| `ret` | %rip ← Mem[%rsp], %rsp ← %rsp + 8 |
| `endbr64` | End branch target (no-op) |
| `leave` | %rsp ← %rbp ; `popq` %rbp |

### Arithmetic operations

| | |
|---|---|
| `leaq` Src, Dest | Dest ← address of Src |
| `incq` Dest | Dest ← Dest + 1 |
| `decq` Dest | Dest ← Dest − 1 |
| `addq` Src, Dest | Dest ← Dest + Src |
| `subq` Src, Dest | Dest ← Dest − Src |
| `imulq` Src, Dest | Dest ← Dest ∗ Src |
| `xorq` Src, Dest | Dest ← Dest ^ Src |
| `orq` Src, Dest | Dest ← Dest \| Src |
| `andq` Src, Dest | Dest ← Dest & Src |
| `negq` Dest | Dest ← − Dest |
| `notq` Dest | Dest ← ∼ Dest |
| `salq` $k$, Dest | Dest ← Dest ≪ $k$ (also `shlq`) |
| `sarq` $k$, Dest | Dest ← Dest ≫ $k$ (arithmetic) |
| `shrq` $k$, Dest | Dest ← Dest ≫ $k$ (logical) |
| `cmpq` Src2, Src1 | Set CCs Src1 – Src2 |
| `testq` Src2, Src1 | Set CCs Src1 & Src2 |

## Integer registers

| | |
|---|---|
| `%rax` | Return value |
| `%rbx` | Callee saved |
| `%rcx` | 4th argument |
| `%rdx` | 3rd argument |
| `%rsi` | 2nd argument |
| `%rdi` | 1st argument |
| `%rbp` | Callee saved |
| `%rsp` | Stack pointer |
| `%r8` | 5th argument |
| `%r9` | 6th argument |
| `%r10` | Scratch register |
| `%r11` | Scratch register |
| `%r12` | Callee saved |
| `%r13` | Callee saved |
| `%r14` | Callee saved |
| `%r15` | Callee saved |

## Addressing modes

- **Immediate**
  $val Val
  val: constant integer value
  `movq $7, %rax`
- **Normal**
  (R) Mem[Reg[R]]
  R: register R specifies memory address
  `movq (%rcx), %rax`
- **Displacement**
  D(R) Mem[Reg[R]+D]
  R: register specifies start of memory region
  D: constant displacement D specifies offset
  `movq 8(%rdi), %rdx`
- **Indexed**
  D(Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]+D]
  D: displacement: 1, 2, or 4 byte constant
  Rb: base register: any integer register
  Ri: index register: any, except %esp
  S: scale: 1, 2, 4, or 8
  `movq 0x100(%rcx,%rax,4), %rdx`

## Instruction suffixes

| | |
|---|---|
| `b` | byte |
| `w` | word (2 bytes) |
| `l` | long (4 bytes) |
| `q` | quad (8 bytes) |

## Condition codes

| | |
|---|---|
| **CF** | Carry Flag |
| **ZF** | Zero Flag |
| **SF** | Sign Flag |
| **OF** | Overflow Flag |