Extra Exercise Collection

Disclaimer: These exercises are in no way officially affiliated with the course. There is no guarantee of correctness, although I do my best to fix any mistakes. (If you spot an error, please let me know!)

Current version: October 7, 2025

Exercice 1 :

Using the Shell

(a) Give the bash command to create a new directory with name "foo" in the current directory.

(b) Give the bash command to create a new file called "foo.c" in the parent directory of the current directory.

(c) Give the bash command to print the contents of "foo.txt", which is located in your home directory.

(d) Give the bash command to run the executable "foo", which is located in the sub-directory "build" of the current directory.

(e) Give the bash command to compile "foo.c", located in the current directory, into the executable "foo" using gcc, with optimisation level 3 enabled.

Solution:

(a) `mkdir foo`

(b) `touch ../foo.c`

(c) `less ~/foo.txt`

(d) `./build/foo`

(e) `gcc -o foo -O3 foo.c`

Exercice 2 :

Basics of C: Answer the following questions with true/false.

(a) There is no way for the value of a variable declared within the scope of a function to persist between calls.

(b) An `int` will always be exactly 4 bytes in size.

(c) The standard stipulates that a `long` must always be larger in size than an `int`.

(d) The integer 0 evaluates to False.

(e) The integer 100 evaluates to True.

(f) The integer -1 evaluates to False.

(g) The value of the expression `foo++` is the same as the value of the expression `++foo`.

(h) Suppose we have `int foo[5]`. Then `foo[0]` is guaranteed to be located next to `foo[1]` in memory.

(i) Suppose we have `int foo[5]`. Then `foo[-1]` is a valid expression.

(j) In general, to store a string of length `k`, the smallest possible array we can define to do so is `char str[k]`.

Solution:

(a) **False.** The static keyword may be used.

(b) **False.** The size of an `int` is implementation defined, but it must be at least 2 bytes.

(c) **False.** Both `long` and `int` are 4 bytes in 32 bit x86.

(d) **True.**

(e) **True.**

(f) **False.**

(g) **False.** The prior has the value `foo`, whereas the latter has the value `foo + 1`.

(h) **True.** Arrays are sequentially allocated in memory.

(i) **True.** Array bounds are not checked.

(j) **False.** We require an extra `char` to store the NUL byte.

Exercice 3 :

Operator Precedence: Correctly parenthesise the following expressions according to the rules of operator precedence. Assume `i, j` are some `int`s, `p` is some pointers to `int`s, and `f` is some function returning `int`. *Eg.* `3 * 4 + 2` $\equiv$ `(3 * 4) + 2`.

(a) `i >> j + 1 * i - j`

(b) `i == j & * p + 3`

(c) Challenge: `* p = i >> 3 << j - 2 * ! f () | 1`

Solution:

(a) `i >> (j + (1 * i) - j)`

(b) `(i == j) & ((*p) + 3)` Note that, surprisingly, `&` has a lower precedence than `==`! This is an artifact of ancient C. Back then, no `&&` existed, so it made sense to have it this way, such that one could for instance have `i == j & i == 1` ≡ `(i == j) & (i == 1)`. Dennis Ritchie himself admitted that this should have been changed in retrospect.

(c) `(*p) = (((i >> 3) << (j - (2 * (! (f() ))))) | 1)`

Exercice 4 :

C Integers: Answer the following questions with true/false (T/F) or a short answer (A). For the integer lengths, assume standard values for a x86_64 Linux machine.

(a) Give the hexadecimal representation of the minimum value of an `unsigned int` (A).

(b) Give the hexadecimal representation of the minimum value of an `int` (A).

(c) Give the hexadecimal representation of the maximum value of an `int` (A).

(d) Give a way to compute `i - j` using only bitwise operators and `+` (A).

(e) When signed and unsigned values are mixed in an expression, signed values are implicitly casted to unsigned (T/F).

(f) Suppose we have `short s` defined as some negative number. `(int) s` will be positive, as the extra bits on the left will be filled with 0s (T/F).

(g) Suppose we have `int i` defined as some negative number. `(char) i` will always be negative too (T/F).

(h) `u << 3 + u` ≡ `u * 9` (T/F).

Solution:

(a) `0x00 00 00 00`

(b) `0x80 00 00 00`

(c) `0x7F FF FF FF`

(d) `i + ~j + 1`

(e) **True.**

(f) **False.** It will be sign-extended, so it remains negative.

(g) **False.** It will be truncated, so it might be positive too.

(h) **False.** Operator precedence.

Exercice 5 :

Multiplication puzzles: Match the following series of shifts and adds to a multiplicative factor. Assume `u` is some `unsigned` and no overflow.

(a) `u << 2`

(b) `(u << 5) + (u << 3) + u`

(c) `((u << 10) >> 5) + (u << ~(-11))`

Solution:

(a) `u << 2` $\equiv$ `u * 4`

(b) `(u << 5) + (u << 3) + u` $\equiv$ `u * 32 + u * 8 + u` $\equiv$ `u * 41`

(c) `((u << 10) >> 5) + (u << ~(-11))` $\equiv$ `(u << 5) + (u << 10)`
$\equiv$ `(u * 32) + (u * 1024)` $\equiv$ `u * 1056`

Exercice 6 :

The C Preprocessor

(a) Define a macro `SQUARE(x)` that multiplies its argument with itself.

(b) Define a macro `DEBUG_INT(x)` that takes a variable's name x and prints "DEBUG: x = *value of* x" if `DEBUG` is defined, and otherwise does nothing.

(c) Name two things that belong in a header (.h) and a source (.c) file, respectively.

(d) When are preprocessor macros beneficial? What are their downsides?

Solution:

(a) `#define SQUARE(x) ((x) * (x))` Note that the brackets around `x` are very important. If they weren't there, for example, `SQUARE(1 + 2)` would expand to $1 + 2 * 1 + 2 = 1 + 2 + 2 = 5 \neq 9$.

(b)
```
#ifdef DEBUG
#define DEBUG_INT(x) printf("DEBUG: %s = %d\n", #x, x)
#else
#define DEBUG_INT(x)
#endif
```

Note that the else case is also required.

(c) **Header:** function declarations, global macros/constants, global structs/unions, ...
Source file: function definitions, static definitions/declarations, ...

(d) Macros are useful as a shorthand for repetitive expressions. They should not, and cannot be used as a replacement for functions. They are not recursive and offer no type checking and can thus quickly lead to hard to debug errors.

Exercice 7 :

Memory Layout Basics

(a) Fill in the blanks: The _____ begins at a high address and grows downwards, whereas the _____ begins at a low address and grows upwards.

(b) True/False: Virtual memory gives each process its own address space. Hence it is never possible that two processes access the same physical page.

(c) Fill in the blanks: In a little-endian machine, the least significant byte is stored at the _____ address.

(d) True/False: The stack will always start at the exact same address in memory.

(e) Outline what happens on the stack when a function ("the caller") calls another function ("the callee") and then when the callee returns.

Solution:

(a) The **stack** begins at a high address and grows downwards, whereas the **heap** begins at a low address and grows upwards.

(b) **False.** It is possible, for instance if a page that houses shared libraries.

(c) In a little-endian machine, the least significant byte is stored at the **lowest** address.

(d) **False.** Modern systems use address space layout randomisation, hence this is not true in general.

(e) A stack frame will be allocated for the callee on top of the stack frame of the caller (this means at a lower address!). The stack frame will contain, among other things, the return address, such that the caller can continue executing once the callee has returned. When the callee returns, its stack frame is deallocated and the top of the stack is now at the end of the caller's stack frame.

Exercice 8 :

Pointers

(a) Explain what the unary `&` and `*` operators do.

(b) True/False: It hold that `sizeof (int) == sizeof (int *)`

(c) True/False: Suppose we have:

```
int main(void) {
    int x = 0;
    int *p = malloc(sizeof (int));
    return 0;
}
```

Then it holds that `&x > p`.

(d) True/False: In a byte addressable system, if we have `int a[5]; int *p=a`, then `a[1]` can be accessed via `*(p + sizeof (int))`.

Solution:

(a) The unary `&` returns the memory address of a given variable. The unary `*` operator returns the content of memory at the location of the operand.

(b) **False.** `sizeof (int)` must only be large enough to hold an integer, whereas `sizeof (int *)` must be large enough to hold a memory address.

(c) **True.** `&x` gives an address on the stack, whereas `p` gives an address on the heap.

(d) **False.** Pointers are typed, so `a[1]` can be accessed via `*(p + 1)`.

Exercice 9 :

Cdecl: For the following C declarations, give their natural language counterpart and vice-versa.

(a) `struct foo *x[10]`

(b) `struct foo (*x)[10]`

(c) `int **(*x)(int, int)[10]`

(d) Declare x as an array 10 of pointer to function returning pointer to `int`.

(e) `(char*[]) x`

(f) `(char (*)[10]) x`

(g) `(int (*)(int *, char)[3]) x`

(h) Challenge: Declare x as function taking pointer to `int` and returning array 3 of array 10 of pointer to function taking pointer to pointer to `int` and returning pointer to `union foo`.

(i) Challenge++: `int* (*(*x[10])(int (*)(int *)))(int *)`

Solution:

(a) Declare `x` as an array 10 of pointer to `struct foo`.

(b) Declare `x` as a pointer to array 10 of `struct foo`.

(c) Declare `x` as a pointer to function taking `(int, int)` and returning array 10 of pointer to pointer to `int`.

(d) `int *(*x[10])()`

(e) Cast x into array of pointer to `char`.

(f) Cast x into pointer to array 10 of `char`.

(g) Cast `x` into pointer to function taking pointer to `int` and `char` and returning array 3 of `int`

(h) `union foo*(*x(int*)[3][10])(int**)`

(i) Declare x as array 10 of pointer to function taking (pointer to function taking pointer to int and returning int) and returning pointer to function taking pointer to int and returning pointer to int. Phew!

Exercice 10 :

Malloc Implementations and Metrics

(a) Outline the differences between implicit and explicit free lists.

(b) (True/False): With explicit free lists, boundary tags are no longer required.

(c) Explain the differences between first fit, next fit, and best fit policies.

(d) Consider the following sequence of `(m/c)alloc`/`free` calls. Compute the aggregate payload P_k and the minimum heap size M_k at points 1, 2, and 3.

```c
#include <stdint.h>

int main(void) {
    int *p1 = malloc(1<<10);
    free(p1);
    // Point 1
    uint32_t *p2 = calloc(1<<8, sizeof(uint32_t));
    char *p3 = malloc(1<<10);
    // Point 2
    int64_t p4 = malloc(1<<10);
    free(p2);
    // Point 3
    free(p3);
    free(p4);
    return 0;
}
```

Solution:

(a) With implicit free lists, all blocks, allocated or free, are in an implicit list given by the size attribute in their header. For each block, we can get to the next block by incrementing our pointer by the size of the current block
Explicit free lists on the other hand maintain an explicit (doubly) linked list of all free blocks. All free blocks will contain a pointer to the next and previous free block.

(b) **False.** Boundary tags are still required for coalescing.

(c) **First fit:** Start at the beginning of the free list every time and allocate the first free block of sufficient size.
Next fit: Start where the previous search ended and allocate the first free block of sufficient size.
Best fit: Scan the entire free list for the free block with the fewest superfluous bytes and allocate it.

(d) $P_1 = 0$ bytes, $H_1 = 1024$ bytes
$P_2 = 2048$ bytes, $H_2 = 2048$ bytes
$P_3 = 2048$ bytes, $H_3 = 3072$ bytes

Exercice 11 :

x86_64 Assembly Warm Up: Answer the following questions with true/false.

(a) In x86_64, each instruction is exactly 8 bytes in size.

(b) Generally speaking, compiling a given program for a RISC architecture will result in more instructions than for a CISC architecture.

(c) A long word is 64 bits in size.

(d) `movq (%rax), (%rcx)` is a valid instruction.

(e) `movq 0x9(%rbx, %rcx), %rax` will move whatever is at memory location `%rbx + %rcx + 0x9` into `%rax`.

(f) `testl %eax, %ebx` sets the ZF if `%eax != %ebx`.

Solution:

(a) **False.** Instructions have variable sizes.

(b) **True.** RISC instructions are simpler and thus sometimes multiple need to be combined to achieve what a single CISC instruction can. This is not to say that RISC is slower than CISC, however.

(c) **False.** A long word is 32 bits in size. A quadword is 64 bits in size.

(d) **False.** Memory/memory transfer is not possible in a single instruction.

(e) **True.**

(f) **False.** `testl %eax, %ebx` computes `%ebx & %eax` and sets ZF and SF accordingly. The statement does not hold for `%eax = $1`, `%ebx = $3`, for example.

Exercice 12 :

Condition Codes: Recall the x86_64 condition codes zero flag (ZF), sign flag (SF), carry flag (CF), and overflow flag (OF). For each of the following conditional jumps, derive a predicate that is true if and only if the jump is taken. You may use the operators $\wedge, \vee, \oplus$ (xor), and $\neg$. *Hint:* Assume the previous instruction was `cmpq a, b` (i.e. `subq a, b` where the result is discarded) and recall the mnemonics are from the perspective of `b` relative to `a` (i.e. `jle` is taken if `b` is less than or equal to `a`).

(a) `je`

(b) `jb`

(c) `ja`

(d) `jl`

(e) `jge`

Solution:

(a) ZF. If `a` and `b` are equal, `cmpq a, b` $\triangleq b - a = 0$, so ZF is set.

(b) CF. First, recall that `jb` interprets the value as unsigned. If $b < a$ and we do the subtraction $b - a$ by hand, we can see we will generate a borrow, so CF is set.

(c) $\neg(\text{CF} \vee \text{ZF})$. We have $b > a \iff \neg(b \leq a)$ and $b \leq a \iff b < a \vee b = a$, so we can simply combine our results from the previous two subtasks.

(d) $\text{OF} \oplus \text{SF}$. First, recall that `jl` interprets the value as signed. Furthermore, $\text{OF} \oplus \text{SF} \iff (\neg\text{OF} \wedge \text{SF}) \vee (\text{OF} \wedge \neg\text{SF})$. If $b < a$, $b - a < 0$. However, the subtraction may overflow. The first case holds if the result is negative and no overflow occurred, the second case covers the scenario where the result overflows and is thus positive.

(e) $\neg(\text{OF} \oplus \text{SF})$. We can simply negate the result from the previous subtask.

Exercice 13 :

Reading Assembly Code I: In the following, assume the below C function was compiled with different values of the constant `K` and varying optimisation levels. For each code fragment, state the value of `K`. *Hint:* Per calling convention, the first argument to a function is stored in `%rdi`, the second in `%rsi`, while the return value is stored in `%rax`.

```c
int foo(int a) {
    return K * a;
}
```

(a)

```
foo:
        pushq   %rbp
```

```
        movq    %rsp, %rbp
        movl    %edi, -4(%rbp)
        movl    -4(%rbp), %edx
        movl    %edx, %eax
        sall    $3, %eax
        addl    %edx, %eax
        sall    $2, %eax
        popq    %rbp
        ret
```

(b)

```
foo:
        leal    (%rdi,%rdi,2), %eax
        sall    $5, %eax
        ret
```

(c)

```
foo:
        xorl    %eax, %eax
        ret
```

Solution:

(a) K = 36.

(b) K = 96.

(c) K = 0.

Exercice 14 :

Reading Assembly Code II: Consider the following C function that computes the GCD of a and b using the Euclidean Algorithm. Which of the following x86_64 assembly code fragments correspond to it? *Hint:* More than one answer may be correct. Also recall the calling convention as in Exercise 13.

```
long foo(long a, long b) {
    while (a != b) {
        if (a > b) {
            a = a - b;
        }
        else {
            b = b - a;
        }
```

```
    }
    return a;
}
```

(a)

```
foo:
        movq    %rsi, %rax
        cmpq    %rsi, %rdi
        jne     .L5
.L2:
        ret
.L3:
        subq    %rdi, %rax
.L4:
        cmpq    %rax, %rdi
        je      .L2
.L5:
        cmpq    %rax, %rdi
        jle     .L3
        subq    %rax, %rdi
        jmp     .L4
```

(b)

```
foo:
        movl    $1, %edx
        xorl    %eax, %eax
.L2:
        cmpq    %rsi, %rdx
        jg      .L5
        imulq   %rdi, %rdx
        incq    %rax
        jmp     .L2
.L5:
        decq    %rax
        ret
```

(c)

```
foo:
        movq    %rdi, %rax
        cmpq    %rsi, %rdi
        jge     .L4
.L3:
        movq    %rsi, %rcx
```

```
        addq    %rsi, %rax
        subq    %rdi, %rcx
        leaq    -1(%rcx), %rdx
        imulq   %rsi, %rdx
        movq    %rcx, %rsi
        addq    %rdx, %rax
        cmpq    %rcx, %rdi
        jl      .L3
        ret
.L4:
        ret
```

(d)

```
foo:
        pushq   %rbp
        movq    %rsp, %rbp
        movq    %rdi, -8(%rbp)
        movq    %rsi, -16(%rbp)
        jmp     .L2
.L4:
        movq    -8(%rbp), %rax
        cmpq    -16(%rbp), %rax
        jle     .L3
        movq    -16(%rbp), %rax
        subq    %rax, -8(%rbp)
        jmp     .L2
.L3:
        movq    -8(%rbp), %rax
        subq    %rax, -16(%rbp)
.L2:
        movq    -8(%rbp), %rax
        cmpq    -16(%rbp), %rax
        jne     .L4
        movq    -8(%rbp), %rax
        popq    %rbp
        ret
```

Solution: (a), (d)

Exercice 15 :

Compiling Basics: Answer the following questions with true/false.

(a) There exist functions that do not require a stack frame.

(b) Every function must end in a `ret` instruction.

(c) In C, nested arrays are stored in row-major ordering.

(d) A multi-level array of k dimensions requires the same amount of memory accesses to access as a nested array of k dimensions.

(e) `switch` statements are always implemented as a series of `if`/`else` statements.

Solution:

(a) **True.** Any function that can do its computations solely within registers will not require a stack frame.

(b) **False.** The function may do a *tail-call* optimisation and simply `jmp` to another function instead.

(c) **True.**

(d) **False.** The nested array will only require one memory access, as the array is contiguous in memory. The multi-level array on the other hand will require k memory accesses.

(e) **False.** Compact `switch` statements will be compiled using a jump table.

Exercice 16 :

Struct Alignment: Sketch the memory layout and state the alignment requirement K of the following `struct` declarations

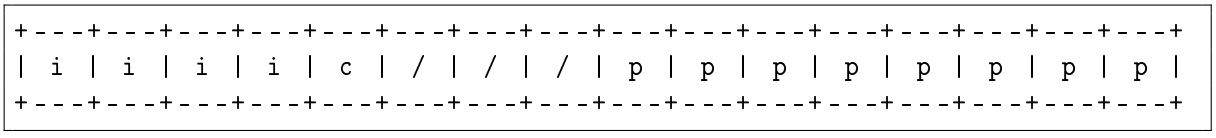
(a) `struct s {int i; char c; void *p;};`

(b) `struct s {char a[3]; char c; double d; int *p[1]};`

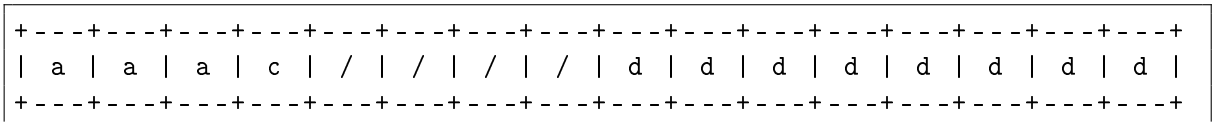
(c) `struct s {short s; struct {int i; char c} t[2]; union {int i; float f} u;};`

Solution:

(a) $K = 8$



(b) $K = 8$



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

(c) $K = 4$

```
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| s | s | / | / |t0i|t0i|t0i|t0i|t0c| / | / | / |t1i|t1i|t1i|t1i|
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
+---+---+---+---+---+---+---+---+
|t1c| / | / | / |uif|uif|uif|uif|
+---+---+---+---+---+---+---+---+
```

Exercice 17 :

Nested Arrays

(a) Let `T a[X]` be an array X of type T. Derive a formula for the address of `a[i]`. *Hint:* You may use the base address of `a` as $\&a$ and the size of `T` as $sizeof(T)$.

(b) Let `T a[X][Y]` be an array X of array Y of type T. Derive a formula for the address of `a[i][j]`. *Hint:* You may also declare `a` as `S a[X]` with `typedef T S[Y]`.

(c) Let `T a[K_1][K_2]...[K_n]` be an array K_1 of array K_2 ...of array K_n of T. Derive a formula for the address of `a[i_1][i_2]...[i_n]`.

Solution:

(a) $\&a + sizeof(T) \cdot i$

(b) $\&a + sizeof(T) \cdot Y \cdot i + sizeof(T) \cdot j = \&a + sizeof(T) \cdot (Y \cdot i + j)$

(c) $\&a + sizeof(T) \cdot \sum_{k=1}^{n} i_k \prod_{l=k+1}^{n} K_l$

Exercice 18 :

Reading `setjmp/longjmp`: What does the following code output?

```c
#include <stdio.h>
#include <setjmp.h>

static jmp_buf buf;

void foo(void) {
    printf("foo1\n");
    longjmp(buf, 2);
```

```
    printf("foo2\n");
}

int main(void) {
    int rv;
    if ((rv = setjmp(buf)) == 0) {
        printf("main1\n");
        foo();
        printf("main2\n");
    }
    else if (rv == 1){
        printf("main3\n");
    }
    else {
        printf("main4\n");
    }
    return 0;
}
```

Solution:

```
main1
foo1
main4
```

Exercice 19 :

Linker Quiz: Answer the following questions with true/false.

(a) Statically linked executables tend to produce larger binaries than dynamically linked executables.

(b) The linker will not link together two symbols of different sizes (assume `-fno-common`).

(c) All the operating system needs to do when starting execution of a statically linked binary is to copy its contents into memory.

(d) By marking the stack as non-executable, buffer overruns are no longer a concern.

Solution:

(a) **True.** Dynamically linked executables need not include all relevant libraries in their binary.

(b) **False.**

(c) **False.** Even though it doesn't need to dynamically link, it still needs to eg. allocate space for symbols in the .bss section.

(d) **False.** Buffer overruns still pose a risk, for instance with return oriented programming. (See attacklab)

Exercice 20 :

Identify the linker symbols: For the following C file, for each name state what kind of linker symbol is generated (global, external, local, none), and whether it is weak or strong where applicable. Assume `-fcommon`. *Challenge:* Additionally, state the exact type of each linker symbol (e.g. an external symbol generates a U in the symbol table). `man nm` provides a succinct overview of the various types.

```c
#include <stddef.h>
#include <stdlib.h>

#define BIAS (10)

extern void add_vec(int *dest, int *src, size_t len);
int sum_vec(int *vec, size_t len);

extern int *vec1;
int arr[25];
int *vec2 = arr;
static const size_t ARR_LEN = 25ul;

int sum() {
    static int iter = 0;

    int *res = calloc(ARR_LEN, sizeof(int));
    add_vec(res, vec1, sizeof(int));
    add_vec(res, vec2, sizeof(int));

    int sum = sum_vec(res, ARR_LEN);
    free(res);

    return sum + iter + BIAS;
}
```

Solution:

(a) **BIAS:** none

(b) **add_vec:** external (U in symbol table)

(c) **sum_vec:** external (U in symbol table)

(d) **vec1:** external (U in symbol table)

(e) **arr:** global, weak (C in symbol table)

(f) **vec2:** global, strong (D in symbol table)

(g) **ARR_LEN:** local (R in symbol table)

(h) **sum:** global, strong (T in symbol table)

(i) **iter:** local (B in symbol table)

(j) **res:** none

(k) **calloc:** external (U in symbol table)

(l) **sum:** none

(m) **free:** external (U in symbol table)

Exercice 21 :

Floating Point Quiz: Answer the following questions with true/false.

(a) Every `int` can be exactly represented as a `double` on an x86_64 Linux machine.

(b) When converting a `float` to a `long`, the error will always be at most 1. (Assuming no special values)

(c) The exponent bias B is computed as $B = 2^e - 1$, where e is the number of exponent bits.

(d) In the case of a denormalised floating point number, the exponent $E = -B + 1$.

(e) Floating point numbers are generally evenly distributed.

(f) Round-to-even rounding is chosen for floating point numbers out of ease of implementation, despite it being statistically biased.

(g) Suppose we have a binary decimal number of the form $1.B \ldots BGRX_1X_2 \ldots$, where the guard bit G is the LSB of our result. Let $S = \bigvee_{i=1}^{\infty} X_i$. We round the result iff $G = R = 1, S = 0$.

Solution:

(a) **True.** `double` is able to represent integers with size ≤ 53 bits exactly.

(b) **False.** The largest possible float is $\sim 3.4 \times 10^{38} >> 9.2 \times 10^{18} \approx 2^{63} - 1$

(c) **False.** $B = 2^{e-1} - 1$. The bias is chosen in such a way that there is a roughly equal positive/negative exponent range.

(d) **True.** This has the benefit that the smallest normalised exponent ($exp = 0 \ldots 01$) is the same as the normalised exponent. ($E = exp - B = 1 - B = -B + 1$)

(e) **False.** As the exponent gets larger, representable values grow further spaced apart.

(f) **False.** Round to even is not statstically biased and arguably harder to implement than other rounding modes.

(g) **False.** This is the case where we round up due to round to even. However, we must also round up if $R = S = 1$.

Exercice 22 :

Converting to FP: Give the bit representation of the following numbers when converted to floating point. The floating point format is half precision IEEE 754, that is 1 sign bit, 5 exponent bits, and 10 fraction bits.

(a) $5000 = 1001110001000_2$

(b) $-\frac{1}{3}$

(c) 1×10^{20}

Solution: We have $B = 2^{e-1} - 1$, so here $B = 2^{5-1} - 1 = 15$. Hence:

(a) $1001110001000_2 = 1.001110001000_2 \times 2^{12}$
We cannot store all fraction bits, so we have to round. We have $G = 0$, $R = 0$, $S = 0$, so we do not need to round up and can simply truncate the result.
This gives us $frac = 0011100010$ with the leading 1 removed.
Clearly $s = 0$. $E = exp - B$, so $exp = E + B = 12 + 15 = 27 = 11011_2$.
Thus the final result is: 0 11011 0011100010

(b) $-\frac{1}{3} = -0.01010101\ldots_2 = -1.010101010101\ldots_2 \times 2^{-2}$
As the fraction is infinite, we clearly need to round. We have $G = 1$, $R = 0$, $S = 1$, so no need to round up.
Hence $frac = 0101010101$. $exp = E + B = -2 + 15 = 13 = 01101_2$.
Thus the final result is: 1 01101 0101010101.

(c) The largest normalised number we can store in our format has $exp = 11110 \implies E = 15$ and $frac = 1111111111$, giving a value of $(2 - 2^{-10}) \times 2^{15} \approx 2^{16} = 2^6 \times 2^{10} \approx 2^6 \times 10^3 << 10^{20}$. Hence we must encode positive infinity.
Thus the final result is: 0 11111 0000000000

Exercice 23 :

Determining key FP values: Give the value of the following descriptions in the format $\frac{a}{b} \times 2^c$ with a, b, c being base 10 integers. Assume the same format as in Exercise 22, i.e. 1 sign bit, 5 exponent bits, and 10 fraction bits.

(a) The largest denormalised number.

(b) The smallest positive normalised number.

(c) The largest normalised number.

Solution: We have $B = 2^{e-1} - 1$, so here $B = 2^{5-1} - 1 = 15$. Hence:

(a) In the denormalised case we have $E = -B + 1 = -15 + 1 = -14$.
Furthermore, we know our fraction has an implicit leading 0. As we want the largest, we want $frac = 1111111111$. This gives $0.1111111111_2 \times 2^{-14} = (1 - 2^{-10}) \times 2^{-14}$.
Hence the final result is $\frac{1023}{1024} \times 2^{-14}$

(b) We want the smallest normalised exponent, so $exp = 00001$, giving $E = 1 - B = -14$.
As our number is normalised, we have an implicit leading 1. As we want the smallest, we get $frac = 0000000000$. This yields $1.0000000000_2 \times 2^{-14} = 1 \times 2^{-14}$. Thus the final result is $\frac{1}{1} \times 2^{-14}$.

(c) We want the largest normalised exponent, so $exp = 11110$, giving $E = exp - B = 30 - 15 = 15$. We have a leading 1 in the fraction, as it is normalised, and since we want the largest, we have $frac = 1111111111$. This gives $1.1111111111_2 \times 2^{15} = (2 - 2^{-10}) \times 2^{15}$.
Thus the final result is $\frac{2047}{1024} \times 2^{15}$

Exercice 24 :

Optimisation Quiz: Answer the following questions with true/false.

(a) With the `-O3` flag enabled, the compiler might perform optimisations that slightly change the semantics of a program, with the benefit of improving performance.

(b) Among RAR, RAW, WAW, and WAR dependencies, RAW is the one that typically carries the greatest performance penalty.

(c) If a procedure is throughput bound, then its operations must execute sequentially.

(d) Much of the manual optimisation that was shown in the lecture is not necessary these days, as modern compilers will simply auto-vectorise code anyway.

(e) If functions didn't have side effects, the compiler could optimise around calls much more.

Solution:

(a) **False.** Generally speaking, the compiler should not change the semantics of a given program. There are however certain flags with which the compiler may perform potentially unsafe optimisations, but `-O3` is not among them.

(b) **True.** Read-after-read dependencies aren't an issue, as a read after a read is always fine. Write-after-write dependencies and write-after-read dependencies can be alleviated with register renaming. Read-after-write dependencies require the read to be postponed until the write has completed.

(c) **False.** This is the case when a procedure is latency bound.

(d) **False.** Compilers are very conservative and still need a lot of assistance from programmers in order to auto-vectorise code, not only by setting the correct flags, but also by preparing the code in such a manner that vectorisation is possible.

(e) **True.** This is a big selling point for more pure/functional languages.

Exercice 25 :

Is the optimisation legal? For the following pairs of C snippets, argue whether they are semantically equivalent in **all** cases, i.e. a compiler would be allowed to rewrite the code in such a manner.

(a)

```
void f_0(int *a, int *b, int *c, size_t len) {
    for (size_t i=0; i<len; i++) {
        b[i] += a[i];
        c[i] += a[i];
    }
}
/* ============================================== */
void f_1(int *a, int *b, int *c, size_t len) {
    size_t i;
    for (i=0; i<len-1; i+=2) {
        b[i] += a[i];
        b[i+1] += a[i+1];
        c[i] += a[i];
        c[i+1] += a[i+1];
    }
    for (; i<len; i++) {
        b[i] += a[i];
        c[i] += a[i];
    }
}
```

(b)

```
int f_0(int *a, size_t j, size_t k) {
    int res = 0;
```

```
    for (size_t i = 0; i < k; i++) {
        res += a[2 * i + j * k];
    }
    return res;
}
/* ============================================ */
int f_1(int *a, size_t j, size_t k) {
    int res = 0;
    size_t i;
    size_t offset = j * k;
    for (i = 0; i < k - 1; i += 2) {
        size_t inner = i << 1;
        res += a[inner + offset];
        res += a[inner + 2 + offset];
    }
    for (; i < k; i++) {
        res += a[(i << 1) + offset];
    }
    return res;
}
```

(c)

```
int f_0(int *a, size_t len) {
    int res = 0;
    for (int i = 0; i < len; i++) {
        res += a[i * 30];
    }
    return res;
}
/* ============================================ */
int f_1(int *a, size_t len) {
    int res = 0;
    for (int i = 0; i < len; i++) {
        res += a[i << 5 - i << 1];
    }
    return res;
}
```

(d)

```
float f_0(float *a, long len) {
    float res = 0;
    for (long i = 0; i < len; i++) {
        res += a[i];
```

```
    }
    for (long i= len - 1; i >= 0; i--) {
        res += a[i];
    }
    return res;
}
/* ============================================== */
float f_1(float *a, long len) {
    float res = 0;
    for (long i = 0; i < len; i++) {
        res += 2.0f * a[i];
    }
    return res;
}
```

Solution:

(a) **Illegal.** This is invalid due to aliasing. Consider the case where

```
int x[] = {1, 2, 0};
int a* = &x[0];
int b* = &x[0];
int c* = &x[1];
```

, then `f_0(a, b, c, 2)` will yield `x[1] == 8`, whereas `f_1(a, b, c, 2)` will yield `x[1] == 6`.

(b) **Legal.** Strength reduction, code motion, and loop unrolling are both used here and have no adverse side effects.

(c) **Illegal.** While this would be a great example for strength reduction, due to C operator precedence `i << 5 - i << 1` $\equiv$ `(i << (5-i)) << 1` $\not\equiv$ `i * 30`.

(d) **Illegal.** While this would be legal for integers, floating point operations are generally not associative due to rounding. Hence they are not semantically equivalent.

Exercice 26 :

Cache Quiz: Answer the following questions with true/false.

(a) A **capacity** miss occurs when a cache is not associative enough to store all of the lines that map to the same set.

(b) A write-allocate policy is usually employed with write-back caches.

(c) A unified cache is accessible by all cores.

Solution:

(a) **False.** This is a **conflict** miss.

(b) **True.** The line is loaded into cache and written to there, that is, it is dirty.

(c) **False.** This is not necessarilly the case. A unified cache is simply one that caches both instructions and data.

Exercice 27 :

Cache formulae: Derive the desired formulae.
Suppose a cache has lines of size 2^b bytes, with associativity 2^e, and 2^s sets in total. Assume a 32-bit address space.

(a) Give a formula for the total number of data bytes that can be stored in the cache.

(b) Give a formula for the number of tag, index, and offset bits.

(c) Give a formula for the set that a given address a will be placed into. *Hint:* You may use the >> operator from C in addition to regular arithmetic operators.

Solution:

(a) 2^b bytes per line, and 2^e lines per set give 2^{b+e} bytes per set. Since there are 2^s sets that gives 2^{b+e+s} bytes in total.

(b) As blocks are 2^b bytes in size, there will be b offset bits. As there are 2^s sets there will be s offset bits. This leaves $32 - b - s$ tag bits.

(c) $set = (a >> b) \bmod 2^s = \lfloor a \div 2^b \rfloor \bmod 2^s$. Explanation: There are 2^s sets in total. Hence we take the index bits modulo 2^s. The shifting or the division and flooring extracts the index bits from the address. Recall that the least significant bits are the offset bits, followed by the index bits, and finally the tag bits.

Exercice 28 :

Exceptions Quiz: Answer the following questions with true/false.

(a) Traps return to the same instruction that triggered them.

(b) Synchronous exceptions are typically handled by the process that triggered them.

(c) There exists exceptions which cannot be masked by the processor.

(d) PICs generally are permitted to reorder exceptions.

Solution:

(a) **False.** Traps return to the next instruction. Consider eg. a system call. Returning back to the system call would lead to an infinite loop.

(b) **False.** Upon an exception, the operating system switches into kernel mode, where the exception is handled.

(c) **True.** There exists so-called non-maskable interrupts. An example would be the hardware watchdog timer, which aids in process switching.

(d) **True.**

Exercice 29 :

Map the exception to its type: For each of the following exception triggering events, state whether it is synchronous or asynchronous, and whether it is a trap, fault, interrupt, or abort.

(a) Calling the `open` system call.

(b) Reading a (valid) memory location that is not paged.

(c) The arrival of data from the network card.

(d) Encountering an `INT 3` (breakpoint) instruction is x86_64.

(e) A memory parity error.

(f) Dereferencing a `NULL` pointer.

Solution:

(a) Trap, and hence synchronous.

(b) (Page) fault, and hence synchronous.

(c) Interrupt, and hence asynchronous.

(d) Trap, and hence synchronous.

(e) Abort, and hence synchronous.

(f) (Segmentation) fault, and hence synchronous. This will lead to the process terminating unless the `SIGSEGV` signal handler has been overwritten.

Exercice 30 :

Virtual Memory Quiz: Answer the following questions with true/false.

(a) The virtual address space must be larger than the physical address space.

(b) Linear page tables are too large to fit in typical main memory sizes.

Solution:

(a) **False.** In fact, in some cases the physical address space may be larger than the virtual address space.

(b) **True.** This is the primary motivation for the use of multi-level page tables.

Exercice 31 :

Cache + Virtual Memory Calculations: Assume a 32-bit virtual address space with a 16-bit physical address space. Assume page size to be 1KiB. Further, assume an 8-way VP cache of size 8KiB with 64 byte blocks.

(a) How many VPN, PPN, VPO, PPO bits are there? Where are they located?

(b) How many CT, CI, CO bits are there? Where are they located?

(c) Will homonyms be an issue here?

(d) How large would a linear page table be? Suppose a 2 byte PTE.

Solution:

(a) Page size is 1KiB = 2^{10}B. Hence there will be 10 VPO and PPO bits. They make up bits [9:0] of the virtual and physical address, respectively.
This leaves $32 - 10 = 22$ VPN, and $16 - 10 = 6$ PPN bits. The VPN makes up bits [31:10] of the virtual address, the PPN makes up bits [15:10] of the physical address.

(b) We know blocks to be of size 64B = 2^6B. Thus there are 6 CO bits. They make up bits [5:0] of the virtual address, as the cache is virtually indexed.
The cache is 8-way associative, and of size 8KiB = 2^{13}B. Applying the formula from Exercise 27, we conclude there are $13 - 5 - 3 = 4$ index bits. They make up bits [6:9] of the virtual address, as the cache is virtually indexed.
Finally, since the cache is physically tagged, there are $16 - 4 - 6 = 6$ tag bits, and they make up bits [15:10] of the physical address.

(c) No, as the index and offset bits fit exactly into the page offset.

(d) The page table needs to store an entry for each VPN. We know there to be 22 VPN bits, hence there must be 2^{22} entries in a linear page table. Each PTE is 2 bytes in size, so the linear page table is of size $2^{22} \cdot 2\text{B} = 2^{23}\text{B} = 8\text{MiB}$

Exercice 32 :

Multiprocessing Quiz: Answer the following questions with true/false.

(a) We say that caches are **consistent**, if their values match.

(b) Snoopy caches require a write-through policy.

(c) In the MESI protocol, if a block is exclusively held, then a *BusRdX* must still be issued upon a write.

(d) x86_64 processors generally implement sequential memory consistency.

(e) False sharing cannot occur in a single processor system, even if multiple threads are running.

(f) The ABA problem cannot occur with pointer tagging.

Solution:

(a) **False.** Then the caches are **coherent**.

(b) **True.** The write-through policy is what enables them to snoop on other processor's writes.

(c) **False.** Being in Exclusive guarantees that no other processor holds the line. Hence, a quiet transition to Modified is legal.

(d) **False.** They typically implement a weaker model called processor consistency.

(e) **True.** The reason why false sharing adversely affects performance is cache coherency. If a single processor system is used, then cache coherency is not of concern.

(f) **False.** The tag can still wrap around.

Exercice 33 :

Devices Quiz: Answer the following questions with true/false.

(a) Each DMA-capable device gets its own virtual address space.

(b) In a producer/consumer ring **underrun** the consumer is consuming faster than the producer can produce.

(c) An advantage of programmed I/O compared to DMA is that caches cannot become inconsistent, as data always flows through the CPU.

(d) When talking about a transmission in the context of devices, we mean a transfer of data from OS to device.

(e) Descriptor rings are typically preferred to buffer rings.

Solution:

(a) **False.** DMA addresses are physical.

(b) **True.** Hence the producer must wait until new resources are available.

(c) **False.** A device register value might change, while the cache continues to store the stale value.

(d) **True.**

(e) **True.** This is because they offer greater flexibility in a number of aspects.