CSE 351 Final Exam

Winter 2019

Tuesday, March 19, 2019

Name: Chip D. Signer

UW NetID: abcdef

Instructions

- Show scratch work for partial credit, but put your final answers in the blanks provided.
- Write your UW NetID on the top right corner of **every page**.
- The last sheet is a reference sheet. **Please detach it from the rest of the exam.**
- You have 1 hour and 50 minutes for the exam.
- There are 210 points, plus 10 extra credit points, for a total of 220 points.
- **This is a long exam.** We are giving you many opportunities to show what you know and earn points. If you can't get something, relax. Do not get stuck, **move on to another problem**.

Question	Points	Score
Arrays and Structs	30	
x86-64 Assembly	16	
Procedures	20	
Buffer Overflow	22	
Processes	20	
Caches	32	
Virtual Memory	30	
Allocation	20	
Memory Bugs	20	
Extra Credit	10	
Total:	220	

UW NetID: abcde

Question 1: Arrays and Structs **(30 total points)**

The following questions refer to this definition of `cse_building`.

```
typedef struct {
    int num_donors;          // number of contributors
    char** donor_names;      // pointer to strings representing donors
    char has_soft_seats;     // 1 if has soft furniture, 0 otherwise
    char location_name[10]; // C string for building location
    short height;            // height, in feet
} cse_building;
```

(a) Give the following quantities as numbers of bytes in the blanks on the right.

i. (2 points) `sizeof(cse_building)` 32 bytes

ii. (2 points) Internal fragementation of `cse_building` 5 bytes

iii. (2 points) External fragementation of `cse_building` 2 bytes

(b) Given the following declaration:

```
cse_building building;
```

Rewrite the following expressions using only pointer arithmetic, `&`, `*`, and casting (so no dot or arrow operators).

i. (2 points) `building.num_donors`

Solution: `*(int*)&building`

ii. (2 points) `&building.donor_names`

Solution: `(char**)((char*)&building + 8)`

(c) (8 points) Implement the following function by code below that would go in the designated loop body.

You may assume that at `cse_building` always has non-negative `num_donors`.

```
// params:
//     a -> pointer to the beginning of an array of cse_buildings
//     len -> number of cse_buildings in the array (> 0)
// returns:
//     a pointer to the cse_building with the most donors
//    (if it's a tie, return the first one in the array).
cse_building* has_most_donors(cse_building* a, int len) {
    cse_building* max_donors;
    int max_donor_num = 0;
    for (int i = 0; i < len; i++) {
        // ------------------------ //
        // WRITE CODE THAT GOES HERE //
        // ------------------------ //
    }
    return max_donors;
}
```

Solution:

```
cse_building* has_most_donors(cse_building *a, int len) {
    cse_building *max_donors;
    int max_donor_num = 0;
    for (int i = 0; i < len; i++) {

        cse_building *curr = &a[i];
        int curr_donor_num = curr->num_donors;
        if (curr_donor_num > max_donor_num) {
            max_donors = curr;
            max_donor_num = curr_donor_num;
        }

    }
    return max_donors;
}
```

(d) (4 points) Your friend (who also took CSE 351) claims to have a more space efficient way of storing this information, showing you the following:

```
typedef struct {
    char** donor_names;      // pointer to strings representing donors
    short height;            // height, in feet
    int num_donors;          // number of contributors
    char has_soft_seats;     // 1 if has soft furniture, 0 otherwise
    char location_name[10]; // C string for building location
} cse_building_better;
```

Are they right in saying it's more space efficient? Why or why not?

Solution: No.

We essentially shifted internal fragmentation into external fragmentation.

8 (donor_names) +
2 (height) +
2 (internal frag) +
4 (num_donors) +
1 (has_soft_seats) +
10 (location_name) +
5 (external frag) = 32 bytes in total.

(e) (8 points) Suppose that we knew that there can be at most 3 donors for any cse_building, and that their names would be no longer than 13 characters (plus one null terminator). We can use this information to fill in the following struct definition:

```
typedef struct {
    char donor_names[3][14];
    short height;
    int num_donors;
    char has_soft_seats;
    char location_name[10];
} cse_building_v2;
```

Name 1 benefit of this alternate representation over the one shown in part (d). Name 1 drawback as well.

Solution: Benefits:

- $3 * 14 = 42$ (donor_names) +
  2 (height) +
  4 (num_donors) +
  1 (has_soft_seats) +
  10 (location_name) +
  1 (external frag) = 60 bytes in total; less fragmentation.
- One less memory access when accessing a particular donor (because of contiguity).

Drawbacks:

- Less flexible, since we have to allocate the same amount of space for each donor, and the same number of donors, essentially wasting space when we have names that are smaller than 20 characters, or less than 3 donors.
- Can't re-use the same name in memory for different donors.

UW NetID: abcde

Question 2: x86-64 Assembly **(16 total points)**

(a) `pushq` and `popq` are important instructions for manipulating the stack. However, they can be replaced by combinations of other instructions.

i. (4 points) Write a sequence of assembly instruction(s) without a `pushq` that has the same behavior as `pushq %rax`.

Solution:
```
subq $8, %rsp
movq %rax, (%rsp)
```

ii. (4 points) Write a sequence of assembly instruction(s) without a `popq` that has the same behavior as `popq %rax`.

Solution:
```
movq (%rsp), %rax
addq $8, %rsp
```

(b) (4 points) What is the difference between `movq $1, %rax` and `movq $1, (%rax)`? (1 sentence)

Solution: The first moves 1 into a register, the second moves 1 into memory.

(c) (4 points) Given the following (decimal) values of registers:
`%rax = 20`
`%rbx = 5`

What is the value of `%rsi` after the following instruction:
`leaq 3(%rax, %rbx, 2), %rsi`

Solution: $3 + 20 + 5 * 2 = 33$

UW NetID: abcde

Question 3: Procedures **(20 total points)**

Consider the following C program fragment compiled without and with a common optimization called *inlining*. The left side shows the unoptimized C and corresponding assembly. The right side shows the assembly compiled with inlining, and above that is "manually" inlined C code.

To inline a function, the compiler essentially replaces calls to that function with the whole body of the function. In the below optimized code on the right, `times` has been inlined into the body of `cube`.

```
// the initial C code
int times(int x, int y)  {
    return x * y;
}

int cube(int x) {
    int x2 = times(x, x);
    return times(x, x2);
}

// unoptimized assembly
times:
    movl    %edi, %eax
    imull   %esi, %eax
    ret
cube:
    pushq   %rbx
    movl    %edi, %ebx
    movl    %edi, %esi
    call    times
    movl    %eax, %esi
    movl    %ebx, %edi
    call    times
    popq    %rbx
    ret
```

```
// the manually optimized C
int times(int x, int y)  {
    return x * y;
}

int cube(int x) {
    int x2 = x * x;
    return x * x2;
}

// optimized assembly
times:
    movl    %edi, %eax
    imull   %esi, %eax
    ret
cube:
    movl    %edi, %eax
    imull   %edi, %eax
    imull   %edi, %eax
    ret
```

(a) i. (2 points) Which callee-saved registers (if any) does the unoptimized `cube` use?

Solution: `%rbx`

ii. (2 points) Which callee-saved registers (if any) does the optimized `cube` use?

Solution: none

The optimization makes the code run 4 times faster! You can see that the optimized `cube` has no `call` instructions anymore, instead it has two `imulls`. But it also has far fewer total instructions.

(b) (6 points) Explain briefly why the compiler was able to remove the `pushq` and `popq` instructions and many of the move instructions

> **Solution:** The moves are no longer needed because we don't need to save registers anymore, since we aren't calling anything.
>
> The `pushq` and `popq` are not needed since we don't use `%rbx`.

While it's one of the most important optimizations performed by compilers, inlining is not always effective or even possible. Now we will consider some of the limitations of this technique.

(c) (5 points) Consider recursive functions. Can a recursive function be inlined into itself? If so, are there limitations or drawbacks? If not, explain why. Your response should be 2-3 concise sentences.

> **Solution:** Yes and no are both accepted with sufficient explanation:
>
> Yes, but not "totally inlined", as that would lead to an infinitely large function.
>
> No, you can't inline a function into itself, because it would still call itself. Also, recursive functions require their own stack frame, which cannot be respected if it's inlined (in reality, the compiler could work around this my making register allocation harder, but this answer is acceptable).

(d) (5 points) Consider functions that are very large (in terms of number of instructions) or take a long time to run (have long running loops). These functions *can* be inlined into other functions, but the benefit is much smaller than we saw in the example on the previous page. In some cases, inlining these can make your code slower!

Why is the benefit smaller? Your response should be 2-3 concise sentences.

> **Solution:** When large functions are inlined, they "blow up" the function they are inlined into, resulting in very very large functions.
>
> Long running functions can be inlined, but the pay off is small because they spend more in the function, so the relative cost of calling / returning is smaller.

UW NetID: abcde

Question 4: Buffer Overflow **(22 total points)**

You're Sokka — fighting the good fight against the invading Fire Nation army. To stop a legion of Fire Nation ships, you've decided to hack into them and remotely disable them!

During your mission, you've learned that their control system runs on a system very similar to x86-64 Linux, but with ***two key differences***:

- **Addresses are 32-bits (and thus so are pointers)**
- **All arguments to functions are passed on the stack**

You've also obtained an excerpt on their control system's source code:

```
1  void set_all_ship_controls(char* mode);
2  void check_password(char* password);
3
4  // pass_input_to_function's parameter is a function pointer,
5  // which is the address of a function
6  void pass_input_to_function(void (*f)(char*)) {
7      char input[16];
8      gets(input);
9      f(input); // Calls f based on the argument stored on the stack
10 }
11
12 void login() {
13     printf("Input your password:\n");
14     pass_input_to_function(&check_password);
15     printf("Done!\n");
16 }
```

Recall that `gets()` is a libc function that reads characters from standard input until the newline (`'\n'`) character is encountered. The resulting characters (not including the newline) are stored in the buffer that's given to gets() as a parameter. If any characters are read, gets() appends a null-terminating character (`'\0'`) to the end of the string.

The normal behavior of the above program is as follows. Once `login` is called, it will print "Input your password:". Then it will call `pass_input_to_function` with a pointer to the function `check_password` as the argument. `pass_input_to_function` calls `gets` to get input from stdin, then it calls the given function (in this case, `check_password`) with the input recieved from `gets` as the argument. Then it returns, and then `login` prints "Done!" and returns.

(a) (4 points) Explain why the use of the gets() function introduces a security vulnerability in the program?

> **Solution:** `gets()` introduces a security vulnerability because it does not have a limit on the number of characters read. This can cause a buffer overflow, allowing a user to enter a malicious string that exceeds our buffer.

The table below shows the stack right before line 9 (from the previous page) runs. The stack pointer is `0x7fff8000`.

Address	Value
`0x7fff800C`	`&f`
`0x7fff8008`	return address to line 15
`0x7fff8004`	
`0x7fff8000`	start of buffer `input`

(b) (6 points) You've gained access to their control system, but you don't know the password. Thankfully, you've learned that `set_all_ship_controls` is located at address `0x351`, and that calling `set_all_ship_controls("off!")` will cause all the ships to stop working!

Construct a sequence of bytes that can get sent to the program during a call to `login()` (read by `gets`) and trigger the above vulnerability and cause all the ships to stop working. Set any other overwritten stack values to 0; ignore any problems this may cause for now.

Give your answer as a sequence of bytes, each written in hex.
Put space between each byte (like this: "`13 AE 00 00 4E 00 00 00 00`").

Hint. The ASCII values (in hex) of the characters in "off!" are: `6f 66 66 21`

Solution: The original question said that the buffer is both 8 and 16 bits wide. Therefore, we took two answers:

(for 8 bit buffer)
`6f 66 66 21` (buf[0] - buf[3] — Filled buffer with input string "off!")
`00 00 00 00` (buf[4] - buf[7])
`00 00 00 00` (Where the return address to login would go)
`51 03 00 00` (Replace &f with the address of pass input to function in little endian order)

OR

(for 16 bit buffer)
`6f 66 66 21` (buf[0] - buf[3] —Filled buffer with i nput string "off!")
`00 00 00 00` (buf[4] - buf[7])
`00 00 00 00` (buf[8] - buf[11])
`00 00 00 00` (buf[12] - buf[15])
`00 00 00 00` (Where the return address to login would go)
`51 03 00 00` (Replace &f with the address of pass input to function in little endian order)

(c) (4 points) You notice that, after constructing the string above, there is one more issue: the program will segfault at some point after calling `set_all_ship_controls`! What is the cause of this crash, and how could we change our attack string to prevent it? Explain in 1-2 sentences.

Solution: We overwrite the return address of `pass_input_to_function` to 0, so we get a segmentation fault due to trying to execute invalid instructions.

(d) (4 points) How could `pass_input_to_function` be modified to fix the vulnerability identified in part (a)? Explain in 1 sentence.

Solution: Use `fgets()`.

(e) (4 points) Describe two ways the compiler, hardware, or operating system (on regular x86-64 Linux machines) protect against buffer overflow attacks.

Solution: OS: memory protection
Compiler: stack canaries, compile-time string length checks, stack address randomization

UW NetID: abcde

Question 5: Processes **(20 total points)**

(a) Consider the following function that uses Linux's fork, wait, and exit.

```
1  void fun_with_process() {
2      int status;
3      int x = 2;
4
5      if (fork() == 0) {
6          if (fork() == 0) {
7              x += 3;
8              printf(" %d ", x);
9              exit(0);
10         } else {
11             wait(&status);
12             x--;
13             printf(" %d ", x);
14         }
15     } else {
16         x++;
17     }
18     printf(" %d ", x);
19     exit(0);
20 }
```

i. (2 points) How many different processes will execute line 18?

Solution: 2 processes

ii. (8 points) What are all the different possible outputs (i.e. order of things printed) for this code?

Solution: Four possibilities:
3 5 1 1
5 3 1 1
5 1 3 1
5 1 1 3

(b) Recall that a context switch is when the operating system switches which process is executing. For each of the following items, choose how frequently they would be the same before and after a context switch. In other words, will they be the same in the old running process and the new one? Assume that the operating system will choose a different process from the currently running process upon a context switch, and that the processes do not share any virtual to physical page mappings.

i. (2 points) The running process ID

◯ Always ◯ Sometimes √ Never

ii. (2 points) The valid entries in the TLB

◯ Always ◯ Sometimes √ Never

iii. (2 points) The contents of the page table base register

◯ Always ◯ Sometimes √ Never

iv. (2 points) The contents of the L1 (physical) cache

◯ Always √ Sometimes ◯ Never

v. (2 points) The program executed by the process

◯ Always √ Sometimes ◯ Never

UW NetID: abcde

Question 6: Caches **(32 total points)**

(a) (4 points) What is the difference between spatial and temporal locality? Answer in 1-2 sentences.

> **Solution:** Temporal locality means that if a program accesses a piece of memory, it will soon re-access that same piece.
> Spatial locality means that if a program accesses a piece of memory, it will soon access a nearby piece.

(b) (4 points) How can a write-back cache improve the performance of writes? Explain in 1-2 sentences.

> **Solution:** If the program update locations in the same block over and over again, a write-back cache defers updating memory until a cache block is evicted, saving trips to memory.

(c) For each of the following sequences of memory accesses, determine the best cache parameters to reduce miss rate. Some parameters will be given, so fill in the remaining one.

Assume that the cache is empty at the beginning of each sequence, the cache uses an LRU replacement policy, and that all accesses are valid.

i. (3 points) `0x00`, `0x08`, `0x10`, `0x18`, `0x20`, `0x28`, `0x30`, `0x38`
Total cache size = 32 bytes
Associativity = 1 Block size = 32 bytes

ii. (3 points) `0x00`, `0x20`, `0x40`, `0x80`, `0x01`, `0x21`, `0x41`, `0x81`
Total cache size = 32 bytes
Block size = 8 bytes Associativity = 4 (fully associative)

iii. (3 points) `0x00`, `0x01`, `0x38`, `0x48`, `0x00`, `0x01`
Total cache size = 16 bytes
Block size = 8 bytes Associativity = 1 (direct mapped)

(d) In a typical cache, the most significant bits of an address make up the tag, the next bits make up the index, and the least significant bits make up the offset. Call this the TIO scheme.

Suppose we are considering switching to a cache with a ITO scheme, where the position of the index and tag are swapped. The width of the index and tag stay the same.

Consider the execution of the following function on a direct mapped cache of total size 128 bytes with 16 sets and a block size of 8 bytes. Assume that `i` and `array` (the pointer) are stored in registers, `array` is 8-byte aligned, and no errors will occur during execution.

```
void loopy(int array[32]) {
    int i;
    for (i = 0; i < 32; i++)
        array[i] += 3;
    for (i = 0; i < 32; i++)
        array[i] -= 3;
}
```

The next 2 questions ask about a single execution of `loopy`, starting with an empty cache, under the **normal TIO scheme**.
Phrase your answer as a number of misses followed by a description of the miss kind, in order of occurence. Example: "4 compulsory misses, then 10 conflict misses." *Hint*: there are no capacity misses.

i. (2 points) How many misses did the first loop take? What kinds were they?

Solution: 16 compulsory misses

ii. (2 points) How many misses did the second loop take? What kinds were they?

Solution: No misses

The next 2 questions ask about a single execution of `loopy`, starting with an empty cache, under the **proposed ITO scheme**.

iii. (2 points) How many misses did the first loop take? What kinds were they?

Solution: 16 compulsory miss

iv. (2 points) How many misses did the second loop take? What kinds were they?

Solution: 16 conflict misses

v. (7 points) In general, why might a ITO cache be less effective than a TIO cache? *Hint:* consider adjacent blocks of memory. (1-2 sentences)

Solution: In ITO, adjacent cache blocks will conflict (same index, different tag). Where as in TIO, adjacent cache blocks have adjacent indexes. So ITO means nearby data is going to generate unnecessary conflict misses.

UW NetID: abcde

Question 7: Virtual Memory **(30 total points)**

(a) Virtual memory is an extremely powerful abstraction with many benefits. For each benefit listed below, provide a brief (1-2 sentence) explanation of how VM accomplishes it.

i. (4 points) Protecting processes from one another.

> **Solution:** The OS can map the same virtual address to a different physical address for different processes. OS can also swap pages to disk and load in new pages for current process. (either answer accepted)

ii. (4 points) Allow programs to use more memory than exists on the machine.

> **Solution:** Since the OS controls mappings of virtual addresses to physical addresses, can use memory as a cache for disk. Can swap pages to disk and load more relevant pages into memory.

(b) Last month, I turned off my desktop computer, opened it up, added some more RAM (doubling it from 16GiB to 32GiB), closed it, and turned it back on.

i. (4 points) Did the size of the virtual address space change? Why or why not?

> **Solution:** No. The virtual address space is determined by the operating system and not by the amount of physical memory present.

ii. (4 points) Did the size of the physical address space change? Why or why not?

> **Solution:** Yes. There are now more physical addresses. 16GiB = 2^{34} bytes = 34 bits needed to represent. 32GiB = 2^{35} = 35 bits needed to represent.

iii. (4 points) After the RAM upgrade, I was able to have more programs open (and thus using memory) before performance began to degrade. Why did this happen? What was causing performance to drop after a certain number of programs were running? (1-2 sentences)

> **Solution:** There was more memory available to keep processes pages in memory without having to swap to disk when context switching to the next process. Since the OS is going to disk less, computer feels (and is) faster.

(c) Imagine the following system:

- 16-bit virtual addresses, 10-bit physical addresses.
- A page size of 16 bytes.
- 2-way set associative TLB with 8 total entries.
- All page table entries NOT in the initial TLB start as invalid.

i. (4 points) Compute the following quantities

Page offset bits: 4 PPN bits: 6

VPN bits: 12 TLB index bits: 2

ii. (6 points) The TLB has the following state:

Set	Tag	PPN	Valid	Tag	PPN	Valid
0	0x1B2	–	0	0x283	0x3A	1
1	0x2FB	0x29	1	0x0E8	0x1D	1
2	0x004	–	0	0x346	–	0
3	0x3F4	0x1B	1	0x257	0x36	1

Fill in the associated information for the following accesses to virtual addresses. (enter **n/a** if the answer cannot be determined):

Virtual Address	TLB Hit?	Page Fault?	PPN
0x3A17	Y	N	0x1D
0x0123	N	Y	n/a

UW NetID: abcde

Question 8: Allocation **(20 total points)**

(a) (5 points) In some versions of C, you can make dynamically-sized allocations on the stack. In the listing below, the array `on_the_stack` of `int`s is stored on the stack inside `foo`'s stack frame.

```
int foo(unsigned n) {
    int on_the_stack[n];
    for (int i = 0; i < n; i++) {
        on_the_stack[i] = 0;
    }
    // do something with on_the_stack
}
```

Does this replace the need for `malloc`? Explain in 1-2 sentences.

Solution: Allocations on the stack still have a limited lifetime and cannot outlive the function they are created in.

(b) (5 points) Suppose I write and use the following code in a version of C with a mark-and-sweep garbage collector.

```
long do_it(int* p) {
    return ~(long)p;
}

int* undo_it(long l) {
    return (int*)(~p);
}

void bad_stuff() {
    long stash = do_it(malloc(sizeof(int)));

    // run a bunch of code

    int *p = undo_it(stash);
    *p = 0xC0FFEE;
}
```

In 1-2 sentences, explain how this is potentially dangerous.

Solution: You've hidden the pointer result of `malloc`, so if a GC happens between `do_it` and `undo_it`, the GC won't "see" the pointer as a root, so the allocation will not be marked and will get swept.

(c) i. (4 points) Describe 2 advantages of an implicit free list allocator over an explicit list allocator. (1-2 sentences)

Solution: Other answers may be ok, here are some.
Smaller minimum block size.
Simpler implementation.
Allocation can be faster when heap is empty.

ii. (6 points) Describe the major advantage of an explicit free list allocator over an implicit list allocator. Also describe the situation in which this advantage is very large. (1-2 sentences)

Solution: Allocation will be much faster when the heap is full, because the explicit list stores *only* free blocks. Also, explicit lists allow you to implement segmented lists, which are basically always faster.

UW NetID: abcde

Question 9: Memory Bugs **(20 total points)**

Consider the following C code where `main` shows the intended use of the functions relating to **`struct`** `foo`.

```
1  struct foo {
2      int x;
3      int y;
4      int* p;
5  };
6
7  struct foo* make_foo() {
8      struct foo* foo = malloc(sizeof(struct foo*));
9
10     foo->x = foo->y * 2;
11     foo->y = 2;
12
13     foo->p = malloc(sizeof(int));
14     *(foo->p) = 42;
15
16     return foo;
17 }
18
19 void update_foo(struct foo* foo) {
20     int num = *(foo->p);
21     foo->p = malloc(sizeof(int));
22     *(foo->p) = num + 1;
23 }
24
25 void free_foo(struct foo* foo) {
26     free(foo);
27     free(foo->p);
28 }
29
30 int main() {
31     struct foo* f = make_foo();
32     for(int i = 0; i < 1000000; i++) {
33         update_foo(f);
34     }
35     free_foo(f);
36 }
```

Questions are on the following page.

(a) (10 points) The function `make_foo` has two memory-related bugs. Find them and briefly describe them (no more than 1 sentence each).

Solution: 1. `malloc(sizeof(struct foo*))` should be `malloc(sizeof(struct foo))`. It was allocating the wrong size.
2. Line 10 uses `foo->y` uninitialized.

(b) (5 points) The function `update_foo` also has a memory-related bug. Find it briefly describe it (no more than 1 sentence).

Solution: Line 21 will leak memory; the malloc'ed pointer that was overwritten is never freed.

(c) (5 points) The function `free_foo` also has a memory-related bug. Find it briefly describe it (no more than 1 sentence).

Solution: Line 27 (`free(foo->p)`) accesses freed memory, because line 26 freed it.

UW NetID: abcde

Question 10: *Extra Credit* *(10 total points)*

```
unsigned mystery(unsigned x) {
    x = x - 1;
    x = x | (x >>  1);
    x = x | (x >>  2);
    x = x | (x >>  4);
    x = x | (x >>  8);
    x = x | (x >> 16);
    x = x + 1
    return x;
}
```

(a) (5 points) Show the step-by-step values of `x` for the execution of `mystery(34)`. Fill in the blanks with value of `x` **after** that line runs. Write **in binary**, and only show the **bottom 8 bits**. Note that 34 == `0x22` == `0b 0010 0010`.

`x = x - 1;` 0010 0001

`x = x | (x >> 1);` 0011 0001

`x = x | (x >> 2);` 0011 1101

`x = x | (x >> 4);` 0011 1111

`x = x | (x >> 8);` 0011 1111

`x = x | (x >> 16);` 0011 1111

`x = x + 1;` 0100 0000

(b) (5 points) Describe succinctly what the code above does.

Solution: Rounds up to the next power of 2.

This page intentionally left blank.

CSE 351 Reference Sheet (Final)

Binary	Decimal	Hex
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}
1	2	4	8	16	32	64	128	256	512	1024

SI Size	Prefix	Symbol	IEC Size	Prefix	Symbol
10^3	Kilo-	K	2^{10}	Kibi-	Ki
10^6	Mega-	M	2^{20}	Mebi-	Mi
10^9	Giga-	G	2^{30}	Gibi-	Gi
10^{12}	Tera-	T	2^{40}	Tebi-	Ti
10^{15}	Peta-	P	2^{50}	Pebi-	Pi
10^{18}	Exa-	E	2^{60}	Exbi-	Ei
10^{21}	Zetta-	Z	2^{70}	Zebi-	Zi
10^{24}	Yotta-	Y	2^{80}	Yobi-	Yi

Sizes

C type	Suffix	Size
`char`	`b`	1
`short`	`w`	2
`int`	`l`	4
`long`	`q`	8

IEEE 754 FLOATING-POINT STANDARD

Value: $\pm 1 \times \text{Mantissa} \times 2^{\text{Exponent}}$

Bit fields: $(-1)^S \times 1.M \times 2^{(E-\text{bias})}$

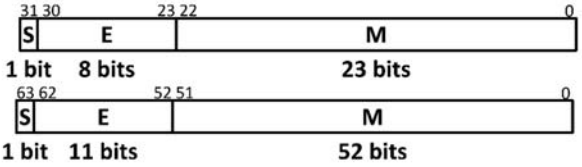
where Single Precision Bias = 127,
Double Precision Bias = 1023.

IEEE 754 Symbols

Exponent	Fraction	Object
0	0	± 0
0	≠0	± Denorm
1 to MAX - 1	anything	± Fl. Pt. Num.
MAX	0	±∞
MAX	≠0	NaN

S.P. MAX = 255, D.P. MAX = 2047

IEEE Single Precision and Double Precision Formats:



Assembly Instructions

`mov a, b`	Copy from `a` to `b`.
`movs a, b`	Copy from `a` to `b` with sign extension. Needs *two* width specifiers.
`movz a, b`	Copy from `a` to `b` with zero extension. Needs *two* width specifiers.
`lea a, b`	Compute address and store in `b`. *Note:* the scaling parameter of memory operands can only be 1, 2, 4, or 8.
`push src`	Push `src` onto the stack and decrement stack pointer.
`pop dst`	Pop from the stack into `dst` and increment stack pointer.
`call <func>`	Push return address onto stack and jump to a procedure.
`ret`	Pop return address and jump there.
`add a, b`	Add from `a` to `b` and store in `b` (and sets flags).
`sub a, b`	Subtract `a` from `b` (compute `b - a`) and store in `b` (and sets flags).
`imul a, b`	Multiply `a` and `b` and store in `b` (and sets flags).
`and a, b`	Bitwise AND of `a` and `b`, store in `b` (and sets flags).
`sar a, b`	Shift value of `b` *right* (*arithmetic*) by `a` bits, store in `b` (and sets flags).
`shr a, b`	Shift value of `b` *right* (*logical*) by `a` bits, store in `b` (and sets flags).
`shl a, b`	Shift value of `b` *left* by `a` bits, store in `b` (and sets flags).
`cmp a, b`	Compare `b` with `a` (compute `b - a` and set condition codes based on result).
`test a, b`	Bitwise AND of `a` and `b` and set condition codes based on result.
`jmp <label>`	Unconditional jump to address.
`j* <label>`	Conditional jump based on condition codes (*more on next page*).
`set* a`	Set byte based on condition codes.

Conditionals

Instruction		(op) s, d	test a, b	cmp a, b
je	"Equal"	d (op) s == 0	b & a == 0	b == a
jne	"Not equal"	d (op) s != 0	b & a != 0	b != a
js	"Sign" (negative)	d (op) s < 0	b & a < 0	b-a < 0
jns	(non-negative)	d (op) s >= 0	b & a >= 0	b-a >= 0
jg	"Greater"	d (op) s > 0	b & a > 0	b > a
jge	"Greater or equal"	d (op) s >= 0	b & a >= 0	b >= a
jl	"Less"	d (op) s < 0	b & a < 0	b < a
jle	"Less or equal"	d (op) s <= 0	b & a <= 0	b <= a
ja	"Above" (unsigned >)	d (op) s > 0U	b & a < 0U	b > a
jb	"Below" (unsigned >)	d (op) s < 0U	b & a > 0U	b < a

Registers

Name	Convention	Name of "virtual" register		
		Lowest 4 bytes	Lowest 2 bytes	Lowest byte
%rax	Return value – Caller saved	%eax	%ax	%al
%rbx	Callee saved	%ebx	%bx	%bl
%rcx	Argument #4 – Caller saved	%ecx	%cx	%cl
%rdx	Argument #3 – Caller saved	%edx	%dx	%dl
%rsi	Argument #2 – Caller saved	%esi	%si	%sil
%rdi	Argument #1 – Caller saved	%edi	%di	%dil
%rsp	Stack Pointer	%esp	%sp	%spl
%rbp	Callee saved	%ebp	%bp	%bpl
%r8	Argument #5 – Caller saved	%r8d	%r8w	%r8b
%r9	Argument #6 – Caller saved	%r9d	%r9w	%r9b
%r10	Caller saved	%r10d	%r10w	%r10b
%r11	Caller saved	%r11d	%r11w	%r11b
%r12	Callee saved	%r12d	%r12w	%r12b
%r13	Callee saved	%r13d	%r13w	%r13b
%r14	Callee saved	%r14d	%r14w	%r14b
%r15	Callee saved	%r15d	%r15w	%r15b

C Functions

void* `malloc(`**size_t** `size)`:
Allocate size bytes from the heap.

void* `calloc(`**size_t** `n,` **size_t** `size)`:
Allocate n*size bytes and initialize to 0.

void `free(`**void*** `ptr)`:
Free the memory space pointed to by ptr.

size_t `sizeof(`**type**`)`:
Returns the size of a given type (in bytes).

char* `gets(`**char*** `s)`:
Reads a line from stdin into the buffer.

pid_t `fork()`:
Create a new child process (duplicates parent).

pid_t `wait(`**int*** `status)`:
Blocks calling process until any child process exits.

int `execv(`**char*** `path,` **char*** `argv[])`:
Replace current process image with new image.

Virtual Memory Acronyms

MMU	Memory Management Unit	**VPO**	Virtual Page Offset	**TLBT**	TLB Tag
VA	Virtual Address	**PPO**	Physical Page Offset	**TLBI**	TLB Index
PA	Physical Address	**PT**	Page Table	**CT**	Cache Tag
VPN	Virtual Page Number	**PTE**	Page Table Entry	**CI**	Cache Index
PPN	Physical Page Number	**PTBR**	Page Table Base Register	**CO**	Cache Offset