Secrets: Outlandish, Polaris
Red = not taught this year

## Table of contents



## Byte Sizes

| Type | Size |
|---|---|
| bool, char, unsigned char, signed char, __int8 | 1 byte |
| __int16, short, unsigned short, wchar_t, __wchar_t | 2 bytes |
| float, __int32, int, unsigned int, long, unsigned long | 4 bytes |
| double, __int64, long double, long long | 8 bytes |

PTRs ARE 8 BYTES

Secrets: Outlandish, Polaris
Red = not taught this year

## Converting Number Formats

**Hex to Dec**

int(insert Hex or bin [via 0b#######] here)

```
In [4]: int(0x2d)
Out[4]: 45
```

bin(insert int/hex [via 0x#####] here)

```
In [5]: bin(0x2d)
Out[5]: '0b101101'
```

hex(insert int or bin [via 0b####] here)

```
In [7]: hex(0b101)
Out[7]: '0x5'

In [ ]:
```

## DATAREP

1.
    a.
    b.
    c. Think about padding for alignment. They come out the same
    d.
    e. Think about padding for alignment. To find the MAX size, its 4*ints. If you had it ordered in the worst way, then you'd also have 4*chars, so 4*chars+4*ints
    f. The alignment of the struct must be the alignment of the largest type in the struct. Also since there is no padding, the struct must already be a multiple of the alignment, so its 0
    g.
2.
3. Use python to convert between hex and ints and ints and binary
    a.
    b.
    c.
4.
    a.
    b.
5.
    a.

    b.
    c. First of all we need to recognize that it is only changing every other byte in the array. We cast the array to a char *, so by changing every other byte we are actually only changing the left half of the image.

       Now see if you can figure out how that gives you the image in J.
    d.
    e.
    f.
    g.
    h.
    i.
    j.
6.
    a.
    b.
    c.
    d.
    e.
7.
    a.
    b.
    c.
8.
    a.
    b.
9.
    a.
10.
11.
12. Why does the this function subtract 1 from the ptr passed in?
    a. `meta* m = (meta*) ptr - 1;`
    b. The pointer is cast to a pointer to the metadata, so the -1 shift will move the pointer a number of bytes equal to the size of the metadata.
13.
    a. internal fragmentation: free memory inside allocation
        i. malloc allocates with alignment 16, creates internal fragmentation
        ii. external fragmentation: free memory outside allocation
    b. Question: Why is this helping make the average allocation more consistent? aren't we still allocating a big new bucket every 1024th allocation?
        i. Answer: Loops can be deceptively expensive.
14.
15.

16.
17.
18.
19.
    a. why is UINT_MAX + 1 defined and INT_MAX + 1 undefined?
        i. C/C++ standard specifies that unsigned arithmetic is modulo arithmetic, but signed integer overflow is simply not defined.
        ii. As for the motivation, it may have something to do signed integers having different representations for negative numbers. Also it doesn't really make sense from a mathematical point of view to have INT_MAX + 1 suddenly become INT_MIN.
20.
21.
22.
23.
24.
25.

## ASM

1.
    a.
    b.
    c.
    d.

Secrets: Outlandish, Polaris
Red = not taught this year

I would suggest writing down what each block is doing in the most general term possible. So for example:

```
.LBB0_1:
        movl    %r8d, %ecx
        imull   %ecx, %ecx
        movl    $1, %edx
```

We know that after this block executes, we will have $rcx = r8d^2$ and $rdx = 1$.

So using that analysis I come to the following sequence of actions:

**f:**

$r8d \leftarrow 1$
$goto\ LBB0\_1$

**.LBB0_6:**

$r8d \leftarrow r8d + 1$

**.LBB0_1:**

$rcd \leftarrow r8d^2$
$rdx \leftarrow 1$

**.LBB0_2:**

$rdi \leftarrow rdx^2$
$rsi \leftarrow 1$

**.LBB0_3:**

$rax \leftarrow rsi^2$
$if\ rax + rdi = rcx\ then\ goto\ LBB0\_7$
$rsi \leftarrow rsi + 1$
$if\ rsi \leq rdx\ then\ goto\ LBB0\_3$
$rdx \leftarrow rdx + 1$
$if\ rdx \leq r8d\ then\ goto\ LBB0\_2$
$goto\ LBB0\_6$

I omit the rest because it's not as critical for now. From this code one should see that we want the following equality to hold to get out of this "loop": $rdx^2 + rsi^2 = r8d^2$. Some of the interesting comparisons happen here:

```
        cmpl    %edx, %esi
        leal    1(%rsi), %eax
        movl    %eax, %esi
        jl      .LBB0_3
```

The reason I transformed this into:

$rsi \leftarrow rsi + 1$
$if\ rsi \leq rdx\ then\ goto\ LBB0\_3$

is because if we look at it, the operation that affects `jl` is `cmpl` (neither `leal` nor `movl` affect the flags). Because right after "comparing" $rsi$ to $rdx$ we increase $rsi$ by one, the condition for the jump is checked with the old value of $rsi$ which is $rsi - 1$. So the jump happens if $rsi - 1 < rdx$ or in other words $rsi \leq rdx$ (this holds for integers). After this, one should see that $rsi$ will be increase until it hits the value of $rdx$ at which point $rdx$ is increased by one and $rsi$ is set back to 1. At the same time if $rdx$ ever exceeds $r8d$ then $r8d$ will be increased by one while $rdx$ is reset to 1.

So to generalize we have an equation $rdx^2 + rsi^2 = r8d^2$ where $rsi \leq rdx$ and $rdx \leq r8d$. This is triple is otherwise known as Pythagorean triple, and the smallest one in terms of $r8d$ would be the triple (3, 4, 5) where $r8d = 5, rdx = 4, rsi = 3$ (one should understand why this order).

And then the last part just copies `"%d %d\n"` into $rdi$ and because $rsi$ and $rdx$ already contain the necessary values, when `printf` is called, it will just print those two numbers starting with $rsi$. Therefore the answer is #3.

2. 255 = 256 - 1 = 2^7 - 1 = 1111 1111 in binary

Secrets: Outlandish, Polaris

Red = not taught this year

   a. b & 255 = b & 1111 1111 which gives the lower 8 bits of b
      i. This is the same as b % 256, as anything above the 8th bit is cut off

3.
   a. unsigned char. %al is Byte 8 of %rax. movzbl means zero-extended byte to long. zero extended means unsigned, byte is 1 byte, which is char
   b. int or unsigned. movl means move long, which is 4 bytes. int is 4 bytes.
   c. signed char. movsbl means sign-extended byte to long.
   d. unsigned short. movzwl means zero-extended word, which is 2 bytes.
   e. a short in array of 8-byte structs. w is 2 bytes, which is a short. 8 is the scale, so we move by 8 bytes within this array, indicating 8-byte structs
   f. array of ints. long is 4 bytes, which is an int. 4 is the scale, so it's an array of 4-byte ints.
   g. char from a structure, or 4th char of a string. Only 1 byte is being moved, so we must be dealing with a char inside a larger data structure
   h. Always. jge condition: !(SF ^ OF).
      i. a^b (XOR) is essentially a test of equality - it returns 0 when a and b match. 0^0 = 0, 1^1 = 0, 0^1 = 1.
      ii. OF is always 0, since you can't overflow with XOR.
      iii. SF is always 0, since a number XOR'd with itself is always 0, since everything matches.
   i. Any odd number. jne condition: !ZF
      i. test S2 S1 sets flags according to S1 & S2.
      ii. 1 & lowest bit of (odd number) = 1. ZF = 0, so !ZF = 1. jne met.
   j. eax < edx, considered as signed.

4.
   a. for loop is f2
      i. you increment loop index: addl $1, %ebx
      ii. you compare to end of loop: cmpl $10, %ebx
   b. switch statement is f3
      i. lots of small functions with jumps
   c. if/else is f1
      i. testb $1, %dl
      ii. jne .L3
      iii. test one condition and jump if it's met, no looping
   d. while loop is f4
      i. looping, but without a loop index

5. How do we know that register rx needs to be caller saved? It is never referenced after the call, so isn't it ambiguous
   a. Function f does not save its data before overwriting it, so it must be caller saved.
   b. Basically every register is either caller saved or callee saved. If the callee doesn't explicitly save a register before overwriting its data, it means the register is caller saved, and vice versa.

Secrets: Outlandish, Polaris
Red = not taught this year

   c. There is no overwriting `%rx` after the call to `g`, but remember `f` is also a function, which means somebody called `f`. `f` did not save `%rx` for that function before overwriting it on line 3 of the function.
6.
7.
   a. See register names. 3rd argument is waldo!
   b. 2nd argument is waldo
   c. %rdi is the matrix, which is made of int*s.  These are 8 bytes long, so we multiply the column by 8 and then add it to %rdi to get waldo!
   d. Ya gotta jump to where waldo is
   e. Look at the arguments bruh
   f. gave up on the rest
8.
9.
10. %rcx is supposed to be %rdi
    a. set rax to 0 first, and then return
    b. Seems to be a mistake? There is no “xchgq %rdi, %rax”
    c.
11.
    a.
    b.
    c.
    d.
    e.
    f.
    g.
    h. Why is the string being passed to strtoul on the heap? Where was it malloc'ed (does it need to be malloc'ed)?
       i. It is passing the return of a library function, which seems to duplicate the string passed to it (i asked this in another post here..). If you think about duplication, it would definitely need to dynamically allocate space to make this happen - since it wouldn't know what string it was being passed. So that string would be on the heap.
       ii. It would be hard to answer this question without knowing what strdup does. So we would expect you to man strdup.
12.
13.
14.
    a.
    b. If c is a signed char*, why are we using movzbl? Shouldn't we be using movsbl? Also, since a char is stored in memory as 2 bytes, would it be acceptable to do use "w" instead of "b"?

i. Great question! I agree that `movsbl` is better. If the argument were an `unsigned char`, then `movzwl` would be equivalent to `movzbl`; but for a signed char, we need `movsbl`, because the sign bit is located in a different place.

15.
16.
   a.
   b. What is the difference between 16B-16C? Also I thought that even tail calls and recursive jumps back to the start of the function would change the stack pointer?

      If you look at tail call elmination lecture, the compilier can actually "do away with" return functions, instead jumping to another function (NOT calling it) and using its return value to return.

      For instance, take the example of

      f(int x)
      {
         return g(x);
      }

      Instead of the normal assembly where you might have

      f:
      movl into EDI
      call g
      mov rax rax // Wouldnt actually happen - rax is already properly set!?
      ret

      You can see that a lot of that is redundant. Tail call elimination could instead (i'm free handing this but the point is there):

      f:
      jmp g_address

      g:
      movl RDI RAX
      ret

      Does that help for C? So there is no adjustment to the stack pointer since there is no call - instead it is modeled as a jump
   c.
   d.
   e.

Secrets: Outlandish, Polaris
Red = not taught this year

17. Question: I am still a bit confused how we can tell what the argument type and return type from assembly code. Does anyone have a strategy they tend to use or ideas on how to approach problems like these?

    Answer: I tend to identify the number of arguments by looking for the register's specific calling convention. You can use the cheatsheet to assist: https://cs61.seas.harvard.edu/site/2018/Asm1/

    For example, if you see %rsi or %esi, then there is probably at least 2 arguments referenced within the function. How these registers are used will bring light to whether these are arguments of the caller or for the callee.

    As for determining the appropriate data type, the suffix of the assembly instruction will help identify this. For example, `movb` is moving a byte, which has a size of 1, so it would be a `char`. If you notice a register being dereferenced with the use of parentheses, then you're dealing with a pointer.

    Question: How do we know how many arguments a function takes?
    The confusion is as follows: what will the assembly do if there is an unused argument? For example, could f1 have one or two arguments (%rsi and %rdx) that are just unused? (In f1, it looks like %rdx is being used as a general-purpose register. Can this happen if the third parameter is just unused?)

    Answer: "Most likely" is because there is no way to tell for sure (a function could even "use" a third register by passing it to a child function without ever doing anything to the register, although that doesn't happen here)

    (ARthur) - You basically tell by the ASM using the registers associated with inputs. Your question about EDX in f1 - its being used as 'scratch space'. The first time we see it its value is being SET, meaning we're not using anything that could have been input on it, meaning nothing was. The only input register being 'used' (i.e., as a SOURCE not a destination) is RDI.

    a.
    b.
    c.
    d.
    e.
    f. Question: why is it strlen?
        i. For sure, the loop starts at the address in “rax”.

Each iteration through the loop it dereferences “rax” and compares the character to 0. It’s looking for the null termination of a string. If the character is not zero, it adds 1 to the address of “rax” (i.e. the register is now pointing to the next character). Then it loops again.

The original address was in “rdi”, so “rax” - “rdi” is going to give you the length of the string.

# IO

1.
   a. In a direct-mapped cache (see bottom of page), each block can fit in exactly one slot. For example, a cache might require that the block with address A go into slot number (A mod S). Because find_slot can only return exactly one slot, and readc uses find_slot to put the block into the slot, it’s a DM Cache

   b. So, as is, the combination of find_slot and readc make this a DM Cache. If we change the inners of find_slot we still only get 1 return value, so we can’t just change that.  But if we edited both readc and find_slot, we could get sets going. Also, if we just changed readc, we could have it completely ignore the find_slot function.
   c.
   d. Each slot is 4096 bytes, but we’re only putting 1 char into each slot at the moment. 2 puts bytes 0-4095 into the same slot. 4 makes a single slot (so 4096 bytes per slot). 5 does the same thing as 2.

      To reduce the number of system calls, we want to avoid entering inside the if statement and setting off the read function, so we want our f->pos to be in the bounds of our slot (i.e. between s->pos and s->pos + s->sz). Every time io61_readc is called, we are prefetching 4096 bytes (SLOTSIZ) from position f->pos to position f->pos + 4096. A good cache will make use of this prefetching when reading sequentially by assigning positions that are close to each other to the same slot. For example, when reading in the first byte of our file, a slot is loaded with file data at positions 0 to 4096. The next time we call io61_readc for the second byte (in this problem we're reading sequentially), we want find_slot to assign us to the same slot, since it has already fetched and saved the data at position 1.

Fix #4 (return 0;) will assign us to the very first slot no matter where our file position is, but it's still more effective than our old find_slot. After our first read() for position 0, we won't have to call read() again until we reach position 4097, at which point f->pos will be out of the bounds of the data that slot 0 covers. The other fixes are similar - the important part is assigning positions close to each other to the same slot (e.g. while #2 will return slot 0 for both position 5 and 6, #1 will return slot 53 and slot 38 for position 5 and 6, respectively, which is a waste since we'll already have position 6 saved in slot 53).

e.

2.

3.

**a.** Question: Why do we inspect N/2 on average?On an intuitive level, think of this as if I were to tell you that one of the N elements is the one you are looking for and you can only turn the elements over one-by-one, you will find that as we run this experiment multiple times with me changing the location of the element you are looking for to random locations uniformly (without preference for any location), then on average you will have to turn N/2 elements to find what you need (there might be times when you have to turn every element over, and sometimes the element you are looking for is the first one, however, on average you still turn N/2 elements over).

**b.** The difference between 3A and 3B is how we store the commits.

commit_info* commits vs commit_info** commits

The first variable is a pointer to commit_info while the second is a pointer to a pointer to commit_info. I *believe* this would be analogous to an array of commit_info structs vs an array of pointers to commit_info structs.

Since we are storing pointers in 3B, we have to go through each pointer before we can get to the struct. We can store 8 pointers per cache line. So that's N/8 additional cache lines we need to consider. Since on average we examine 1/2 of the objects, you end up with N/8/2 which is N/16 additional cache lines. , Thus N/16 + N/2 = 9N/16

**c.**

**d.** The initial bucket is based on the first 8 bytes of hash, which is an arbitrary value. Furthermore, since each bucket has type commit_info*, it is 8 bytes in size, and 8 such buckets fit into a single 64-byte cache line. This means that after every 8 buckets, we'd need 1 cache line. Note also, though, that we can conceivably start on the last bucket in a cache line (in the example given, it's reading 2 buckets starting at bucket 15, but bucket 7 is equally bad, as is bucket 23) -- and this would incur the cost of fetching the cache line that bucket is on (e.g. cache line #0 for bucket 7), and the next bucket is on a different cache line, so it immediately incurs the cost of fetching the next cache line itself (e.g. cache line #1 for bucket 8, which immediately follows bucket 7). Basically, there is a "really bad" bucket to start at.

e.

4.
5.
6.
7.
8. Bélády's optimal algorithm (include accents)

C. Recall that Hit rate=# hits# hits+# misses,# hits=# accesses−# misses

There is only 1 miss for a cache line (the compulsory miss) in this sequential array access scenario. So the hit rate is maximized by maximizing the number of hits, or equivalently maximizing the number of accesses. Unit sizer of 1 gives you the most accesses.

9.
   a.
   b.
   c. why does unit size of 1 maximize hit rate?
      i. Hit rate=# hits/(# hits+# misses),# hits=# accesses−# misses
         There is only 1 miss for a cache line (the compulsory miss) in this sequential array access scenario. So the hit rate is maximized by maximizing the number of hits, or equivalently maximizing the number of accesses. Unit size of 1 gives you the most accesses.
10.
   a. Undefined: B. it causes memcpy to read beyond the end of cache buffer
   b. loop forever without causing undefined behavior: A, D.
11. A. O_SYNC Write operations on the file will complete according to the requirements of synchronized I/O file integrity completion (by contrast with the synchronized I/O data integrity completion provided by O_DSYNC.)

    By the time write(2) (or similar) returns, the output data and associated file metadata have been transferred to the underly-ing hardware (i.e., as though each write(2) was followed by a call to fsync(2)). See NOTES below.
12.
13.

a. Question: Why is Solange's code faster on smaller sizes and Donald's code faster on larger sizes? Isn't Solange's code always slower because Donald only makes one system call?

   Donald's code does NOT use the cache. He is calling the read() system call directly.

   Solange uses the cache, meaning she is only making a single system call if the size is below the cache size.

   Take the instance where 3 READS are called in each instance, each of size 1024.

   Donald will make 3 system calls.

   Solange will make 1 system call (filling the buffer with 4096), and then the following two will be cached reads.

b.

c.

d. Donald's code does NOT use the cache. He is calling the read() system call directly.

   Solange uses the cache, meaning she is only making a single system call if the size is below the cache size.

   Take the instance where 3 READS are called in each instance, each of size 1024.

   Donald will make 3 system calls.

   Solange will make 1 system call (filling the buffer with 4096), and then the following two will be cached reads.

14. E. question: Why does the buffer cache speed up reverse sequential access to a disk file? I'm not 100% here, but i would guess its because of the hardware of a disk. Thinking about how a disk spins, if you needed REVERSE sequential, then it would need to readjust the head each time, having an extremely high access cost.

    Using a buffer cache, entire segments or entire files can be pre-loaded from disk, removing the high-cost of the hardware readjustment

    Streams cannot be memory mapped: memory mapping only works for random-access files (and not even for all random-access files). Memory mapping is a little more dangerous; if your program has an error and modifies memory inappropriately, that might now corrupt a disk file. (If your program has such an error, it suffers from undefined behavior and could corrupt the file anyway, but memory mapping does make corruption slightly more likely.)

Secrets: Outlandish, Polaris
Red = not taught this year

# MISC

1. Key here is it says "INTO" his repo, so the chain would be:
    a. Git add, Git commit, Git push (pull is only used to grab stuff)
    b. See order above
    c. Here, we're mainly looking at the spot that starts with "At noon on October 11..." We see that Snowden pushed a commit first. We then see a log of Norton followed by a merge, so Norton must have tried to push his code (which would fail due to Snowden's push), then pulled the new changes. Following a pull, Norton would commit and then push. We see that the next log shows Snowden made a change, WITHOUT a merge, so he must have pulled first, then committed, then pushed.
    d. Following the logic above, conflicts could occur when Norton pulled prior to the merge, and when Snowden pulled after the 1st merge. A conflict can occur whenever changes are made to the local and shared repo (e.g. Norton edited github repo but Snowden made local changes)
2.

Running code with sanitizers:
clang++ -fsanitize=address,undefined test.cpp && ./a.out

Pointer syntax explained in depth: http://www.cplusplus.com/doc/tutorial/pointers/

Units:
1 GB = 10^9 bytes
1 TB = 10^12 bytes

ACII:

Secrets: Outlandish, Polaris

Red = not taught this year

| ASCII control characters | | |
|---|---|---|
| 00 | NULL | (Null character) |
| 01 | SOH | (Start of Header) |
| 02 | STX | (Start of Text) |
| 03 | ETX | (End of Text) |
| 04 | EOT | (End of Trans.) |
| 05 | ENQ | (Enquiry) |
| 06 | ACK | (Acknowledgement) |
| 07 | BEL | (Bell) |
| 08 | BS | (Backspace) |
| 09 | HT | (Horizontal Tab) |
| 10 | LF | (Line feed) |
| 11 | VT | (Vertical Tab) |
| 12 | FF | (Form feed) |
| 13 | CR | (Carriage return) |
| 14 | SO | (Shift Out) |
| 15 | SI | (Shift In) |
| 16 | DLE | (Data link escape) |
| 17 | DC1 | (Device control 1) |
| 18 | DC2 | (Device control 2) |
| 19 | DC3 | (Device control 3) |
| 20 | DC4 | (Device control 4) |
| 21 | NAK | (Negative acknowl.) |
| 22 | SYN | (Synchronous idle) |
| 23 | ETB | (End of trans. block) |
| 24 | CAN | (Cancel) |
| 25 | EM | (End of medium) |
| 26 | SUB | (Substitute) |
| 27 | ESC | (Escape) |
| 28 | FS | (File separator) |
| 29 | GS | (Group separator) |
| 30 | RS | (Record separator) |
| 31 | US | (Unit separator) |
| 127 | DEL | (Delete) |

| ASCII printable characters | | | | | |
|---|---|---|---|---|---|
| 32 | space | 64 | @ | 96 | ` |
| 33 | ! | 65 | A | 97 | a |
| 34 | " | 66 | B | 98 | b |
| 35 | # | 67 | C | 99 | c |
| 36 | $ | 68 | D | 100 | d |
| 37 | % | 69 | E | 101 | e |
| 38 | & | 70 | F | 102 | f |
| 39 | ' | 71 | G | 103 | g |
| 40 | ( | 72 | H | 104 | h |
| 41 | ) | 73 | I | 105 | i |
| 42 | * | 74 | J | 106 | j |
| 43 | + | 75 | K | 107 | k |
| 44 | , | 76 | L | 108 | l |
| 45 | - | 77 | M | 109 | m |
| 46 | . | 78 | N | 110 | n |
| 47 | / | 79 | O | 111 | o |
| 48 | 0 | 80 | P | 112 | p |
| 49 | 1 | 81 | Q | 113 | q |
| 50 | 2 | 82 | R | 114 | r |
| 51 | 3 | 83 | S | 115 | s |
| 52 | 4 | 84 | T | 116 | t |
| 53 | 5 | 85 | U | 117 | u |
| 54 | 6 | 86 | V | 118 | v |
| 55 | 7 | 87 | W | 119 | w |
| 56 | 8 | 88 | X | 120 | x |
| 57 | 9 | 89 | Y | 121 | y |
| 58 | : | 90 | Z | 122 | z |
| 59 | ; | 91 | [ | 123 | { |
| 60 | < | 92 | \ | 124 | \| |
| 61 | = | 93 | ] | 125 | } |
| 62 | > | 94 | ^ | 126 | ~ |
| 63 | ? | 95 | _ | | |

| Extended ASCII characters | | | | | | | |
|---|---|---|---|---|---|---|---|
| 128 | Ç | 160 | á | 192 | └ | 224 | Ó |
| 129 | ü | 161 | í | 193 | ┴ | 225 | ß |
| 130 | é | 162 | ó | 194 | ┬ | 226 | Ô |
| 131 | â | 163 | ú | 195 | ├ | 227 | Ò |
| 132 | ä | 164 | ñ | 196 | ─ | 228 | õ |
| 133 | à | 165 | Ñ | 197 | ┼ | 229 | Õ |
| 134 | å | 166 | ª | 198 | ã | 230 | µ |
| 135 | ç | 167 | º | 199 | Ã | 231 | þ |
| 136 | ê | 168 | ¿ | 200 | ╚ | 232 | Þ |
| 137 | ë | 169 | ® | 201 | ╔ | 233 | Ú |
| 138 | è | 170 | ¬ | 202 | ╩ | 234 | Û |
| 139 | ï | 171 | ½ | 203 | ╦ | 235 | Ù |
| 140 | î | 172 | ¼ | 204 | ╠ | 236 | ý |
| 141 | ì | 173 | ¡ | 205 | ═ | 237 | Ý |
| 142 | Ä | 174 | « | 206 | ╬ | 238 | ¯ |
| 143 | Å | 175 | » | 207 | ¤ | 239 | ´ |
| 144 | É | 176 | ░ | 208 | ð | 240 | ≡ |
| 145 | æ | 177 | ▒ | 209 | Ð | 241 | ± |
| 146 | Æ | 178 | ▓ | 210 | Ê | 242 | ‗ |
| 147 | ô | 179 | │ | 211 | Ë | 243 | ¾ |
| 148 | ö | 180 | ┤ | 212 | È | 244 | ¶ |
| 149 | ò | 181 | Á | 213 | ı | 245 | § |
| 150 | û | 182 | Â | 214 | Í | 246 | ÷ |
| 151 | ù | 183 | À | 215 | Î | 247 | ¸ |
| 152 | ÿ | 184 | © | 216 | Ï | 248 | ° |
| 153 | Ö | 185 | ╣ | 217 | ┘ | 249 | ¨ |
| 154 | Ü | 186 | ║ | 218 | ┌ | 250 | · |
| 155 | ø | 187 | ╗ | 219 | █ | 251 | ¹ |
| 156 | £ | 188 | ╝ | 220 | ▄ | 252 | ³ |
| 157 | Ø | 189 | ¢ | 221 | ¦ | 253 | ² |
| 158 | × | 190 | ¥ | 222 | Ì | 254 | ■ |
| 159 | ƒ | 191 | ┐ | 223 | ▀ | 255 | nbsp |

x64 Cheat Sheet

Fall 2018

1. x64 Registers

x64 assembly code uses sixteen 64-bit registers. Additionally, the lower bytes of some of these registers may be accessed independently as 32-, 16- or 8-bit registers. The register names are as follows:

| 8-byte register | Bytes 5-8 | Bytes 7-8 | Byte 8 |
|---|---|---|---|
| %rax | %eax | %ax | %al |
| %rcx | %ecx | %cx | %cl |
| %rdx | %edx | %dx | %dl |
| %rbx | %ebx | %bx | %bl |
| %rsi | %esi | %si | %sil |
| %rdi | %edi | %di | %dil |

Secrets: Outlandish, Polaris
Red = not taught this year

| | | | |
|---|---|---|---|
| %rsp | %esp | %sp | %spl |
| %rbp | %ebp | %bp | %bpl |
| %r8 | %r8d | %r8w | %r8b |
| %r9 | %r9d | %r9w | %r9b |
| %r10 | %r10d | %r10w | %r10b |
| %r11 | %r11d | %r11w | %r11b |
| %r12 | %r12d | %r12w | %r12b |
| %r13 | %r13d | %r13w | %r13b |
| %r14 | %r14d | %r14w | %r14b |
| %r15 | %r15d | %r15w | %r15b |

For more details of register usage, see Register Usage, below.

2. Operand Specifiers

The basic types of operand specifiers are below. In the following table,

- Imm refers to a constant value, e.g. 0x8048d8e or 48,
- Ex refers to a register, e.g. %rax,
- R[Ex] refers to the value stored in register Ex, and
- M[x] refers to the value stored at memory address x.

| Type | From | Operand Value | Name |
|---|---|---|---|
| Immediate | $Imm | Imm | Immediate |
| Register | Ea | R[Ea] | Register |
| Memory | Imm | M[Imm] | Absolute |
| Memory | (Ea) | M[R[Eb]] | Absolute |
| Memory | Imm(Eb, Ei, s) | M[Imm + R[Eb] + (R[Ei] x s)] | Scaled indexed |

More information about operand specifiers can be found on pages 169-170 of the textbook.

3. x64 Instructions

In the following tables,

- “byte” refers to a one-byte integer (suffix b),
- “word” refers to a two-byte integer (suffix w),
- “doubleword” refers to a four-byte integer (suffix l), and
- “quadword” refers to an eight-byte value (suffix q).

Most instructions, like mov, use a suffix to show how large the operands are going to be. For example, moving a quadword from %rax to %rbx results in the instruction movq %rax, %rbx. Some instructions, like ret, do not use suffixes because there is no need. Others, such as movs

Secrets: Outlandish, Polaris
Red = not taught this year

and movz will use two suffixes, as they convert operands of the type of the first suffix to that of the second. Thus, assembly to convert the byte in %al to a doubleword in %ebx with zero-extension would be movzbl %al, %ebx.

In the tables below, instructions have one suffix unless otherwise stated.

3.1 Data Movement

cwtl
Convert word in %ax to doubleword in %eax (sign-extended)
182

| | | |
|---|---|---|
| cltq | Convert doubleword in %eax to quadword in %rax (sign-extended) | 182 |
| cqto | Convert quadword in %rax to octoword in %rdx:%rax | 182 |

3.2 Arithmetic Operations
Unless otherwise specified, all arithmetic operation instructions have one suffix.

3.2.1 Unary Operations

| Instruction | | Description | Page # |
|---|---|---|---|
| inc | D | Increment by 1 | 178 |
| dec | D | Decrement by 1 | 178 |
| neg | D | Arithmetic negation | 178 |
| not | D | Bitwise complement | 178 |

3.2.2 Binary Operations

| Instruction | | Description | Page # |
|---|---|---|---|
| leaq | S, D | Load effective address of source into destination | 178 |

Secrets: Outlandish, Polaris
Red = not taught this year

add S, D Add source to destination 178
sub S, D Subtract source from destination 178
imul S, D Multiply destination by source 178
xor S, D Bitwise XOR destination by source 178
or S, D Bitwise OR destination by source 178
and S, D Bitwise AND destination by source 178

3.2.3 Shift Operations

Instruction Description Page #
sal / shl k, D Left shift destination by k bits 179
sar k, D Arithmetic right shift destination by k bits 179
shr k, D Logical right shift destination by k bits 179

3.2.4 Special Arithmetic Operations

Instruction Description Page #
imulq S Signed full multiply of %rax by S
Result stored in %rdx:%rax 182

mulq S Unsigned full multiply of %rax by S
Result stored in %rdx:%rax 182

idivq
S Signed divide %rdx:%rax by S Quotient stored in %rax
Remainder stored in %rdx
182

divq
S Unsigned divide %rdx:%rax by S Quotient stored in %rax
Remainder stored in %rdx
182

3.3 Comparison and Test Instructions
Comparison instructions also have one suffix.

Instruction Description Page #

Secrets: Outlandish, Polaris

Red = not taught this year

| | | | |
|---|---|---|---|
| cmp | S2, S1 | Set condition codes according to S1 - S2 | 185 |
| test | S2, S1 | Set condition codes according to S1 & S2 | 185 |

## 3.4 Accessing Condition Codes

None of the following instructions have any suffixes.

### 3.4.1 Conditional Set Instructions

| Instruction | | Description | Condition Code | Page # |
|---|---|---|---|---|
| sete / setz | D | Set if equal/zero | ZF | 187 |
| setne / setnz | D | Set if not equal/nonzero | ~ZF | 187 |
| sets | D | Set if negative | SF | 187 |
| setns | D | Set if nonnegative | ~SF | 187 |
| setg / setnle | D | Set if greater (signed) | ~(SF^0F)&~ZF | 187 |
| setge / setnl | D | Set if greater or equal (signed) | ~(SF^0F) | 187 |
| setl / setnge | D | Set if less (signed) | SF^0F | 187 |
| setle / setng | D | Set if less or equal | (SF^OF)\|ZF | 187 |
| seta / setnbe | D | Set if above (unsigned) | ~CF&~ZF | 187 |
| setae / setnb | D | Set if above or equal (unsigned) | ~CF | 187 |
| setb / setnae | D | Set if below (unsigned) | CF | 187 |
| setbe / setna | D | Set if below or equal (unsigned) | CF\|ZF | 187 |

### 3.4.2 Jump Instructions

| Instruction | | Description | Condition Code | Page # |
|---|---|---|---|---|
| jmp | Label | Jump to label | | 189 |
| jmp | *Operand | Jump to specified location | | 189 |
| je / jz | Label | Jump if equal/zero | ZF | 189 |
| jne / jnz | Label | Jump if not equal/nonzero | ~ZF | 189 |
| js | Label | Jump if negative | SF | 189 |
| jns | Label | Jump if nonnegative | ~SF | 189 |
| jg / jnle | Label | Jump if greater (signed) | ~(SF^0F)&~ZF | 189 |
| jge / jnl | Label | Jump if greater or equal (signed) | ~(SF^0F) | 189 |
| jl / jnge | Label | Jump if less (signed) | SF^0F | 189 |
| jle / jng | Label | Jump if less or equal | (SF^OF)\|ZF | 189 |
| ja / jnbe | Label | Jump if above (unsigned) | ~CF&~ZF | 189 |
| jae / jnb | Label | Jump if above or equal (unsigned) | ~CF | 189 |
| jb / jnae | Label | Jump if below (unsigned) | CF | 189 |
| jbe / jna | Label | Jump if below or equal (unsigned) | CF\|ZF | 189 |

### 3.4.3 Conditional Move Instructions

Secrets: Outlandish, Polaris
Red = not taught this year

Conditional move instructions do not have any suffixes, but their source and destination operands must have the same size.

| | | | | |
|---|---|---|---|---|
| cmovne / cmovnz | S, D | Move if not equal/nonzero | ~ZF | 206 |
| cmovs | S, D | Move if negative | SF | 206 |
| cmovns | S, D | Move if nonnegative | ~SF | 206 |
| cmovg / cmovnle | S, D | Move if greater (signed) | ~(SF^0F)&~ZF | 206 |
| cmovge / cmovnl | S, D | Move if greater or equal (signed) | ~(SF^0F) | 206 |
| cmovl / cmovnge | S, D | Move if less (signed) | SF^0F | 206 |
| cmovle / cmovng | S, D | Move if less or equal | (SF^OF)\|ZF | 206 |
| cmova / cmovnbe | S, D | Move if above (unsigned) | ~CF&~ZF | 206 |
| cmovae / cmovnb | S, D | Move if above or equal (unsigned) | ~CF | 206 |
| cmovb / cmovnae | S, D | Move if below (unsigned) | CF | 206 |
| cmovbe / cmovna | S, D | Move if below or equal (unsigned) | CF\|ZF | 206 |

## 3.5 Procedure Call Instruction

Procedure call instructions do not have any suffixes.

| Instruction | | Description | Page # |
|---|---|---|---|
| call | Label | Push return address and jump to label | 221 |
| call | *Operand | Push return address and jump to specified location | 221 |
| leave | | Set %rsp to %rbp, then pop top of stack into %rbp | 221 |
| ret | | Pop return address from stack and jump there | 221 |

# 4. Coding Practices

## 4.1 Commenting

Each function you write should have a comment at the beginning describing what the function does and any arguments it accepts. In addition, we strongly recommend putting comments alongside your assembly code stating what each set of instructions does in pseudocode or some higher level language. Line breaks are also helpful to group statements into logical blocks for improved readability.

## 4.2 Arrays

Arrays are stored in memory as contiguous blocks of data. Typically an array variable acts as a pointer to the first element of the array in memory. To access a given array element, the index value is multiplied by the element size and added to the array pointer. For instance, if arr is an array of ints, the statement:
arr[i] = 3;

can be expressed in x86-64 as follows (assuming the address of arr is stored in %rax and the index i is stored in %rcx):
movq $3, (%rax, %rcx, 8)

More information about arrays can be found on pages 232-241 of the textbook.

### 4.3 Register Usage

There are sixteen 64-bit registers in x86-64: %rax, %rbx, %rcx, %rdx, %rdi, %rsi, %rbp, %rsp, and %r8-r15. Of these, %rax, %rcx, %rdx, %rdi, %rsi, %rsp, and %r8-r11 are considered caller-save registers, meaning that they are not necessarily saved across function calls. By convention, %rax is used to store a function's return value, if it exists and is no more than 64 bits long. (Larger return types like structs are returned using the stack.) Registers %rbx, %rbp, and %r12-r15 are callee-save registers, meaning that they are saved across function calls. Register %rsp is used as the stack pointer, a pointer to the topmost element in the stack. Additionally, %rdi, %rsi, %rdx, %rcx, %r8, and %r9 are used to pass the first six integer or pointer parameters to called functions. Additional parameters (or large parameters such as structs passed by value) are passed on the stack.
In 32-bit x86, the base pointer (formerly %ebp, now %rbp) was used to keep track of the base of the current stack frame, and a called function would save the base pointer of its caller prior to updating the base pointer to its own stack frame. With the advent of the 64-bit architecture, this has been mostly eliminated, save for a few special cases when the compiler cannot determine ahead of time how much stack space needs to be allocated for a particular function (see Dynamic stack allocation).

### 4.4 Stack Organization and Function Calls

#### 4.4.1 Calling a Function

To call a function, the program should place the first six integer or pointer parameters in the registers %rdi, %rsi, %rdx, %rcx, %r8, and %r9; subsequent parameters (or parameters larger than 64 bits) should be pushed onto the stack, with the first argument topmost. The program should then execute the call instruction, which will push the return address onto the stack and jump to the start of the specified function.
Example:

```
# Call foo(1, 15)
movq   $1, %rdi         # Move 1 into %rdi Movq        $15, %rsi        # Move 15 into %rsi
call      foo       # Push return address and jump to label foo
```

If the function has a return value, it will be stored in %rax after the function call.

#### 4.4.2 Writing a Function

Secrets: Outlandish, Polaris
Red = not taught this year

An x64 program uses a region of memory called the stack to support function calls. As the name suggests, this region is organized as a stack data structure with the “top” of the stack growing towards lower memory addresses. For each function call, new space is created on the stack to store local variables and other data. This is known as a stack frame. To accomplish this, you will need to write some code at the beginning and end of each function to create and destroy the stack frame.

Setting Up: When a call instruction is executed, the address of the following instruction is pushed onto the stack as the return address and control passes to the specified function.
If the function is going to use any of the callee-save registers (%rbx, %rbp, or %r12-r15), the current value of each should be pushed onto the stack to be restored at the end. For example:

```
Pushq  %rbx
pushq  %r12
pushq  %r13
```

Finally, additional space may be allocated on the stack for local variables. While it is possible to make space on the stack as needed in a function body, it is generally more efficient to allocate this space all at once at the beginning of the function. This can be accomplished using the call subq $N, %rsp where N is the size of the callee's stack frame. For example:

```
subq    $0x18, %rsp    # Allocate 24 bytes of space on the stack
```

This set-up is called the function prologue.

Using the Stack Frame: Once you have set up the stack frame, you can use it to store and access local variables:

- Arguments which cannot fit in registers (e.g. structs) will be pushed onto the stack before the call instruction, and can be accessed relative to %rsp. Keep in mind that you will need to take the size of the stack frame into account when referencing arguments in this manner.
- If the function has more than six integer or pointer arguments, these will be pushed onto the stack as well.
- For any stack arguments, the lower-numbered arguments will be closer to the stack pointer. That is, arguments are pushed on in right-to-left order when applicable.
- Local variables will be stored in the space allocated in the function prologue, when some amount is subtracted from %rsp. The organization of these is up to the programmer.

Cleaning Up: After the body of the function is finished and the return value (if any) is placed in %rax, the function must return control to the caller, putting the stack back in the state in which it

was called with. First, the callee frees the stack space it allocated by adding the same amount to the stack pointer:

```
addq    $0x18, %rsp    # Give back 24 bytes of stack space
```

Secrets: Outlandish, Polaris
Red = not taught this year

Then, it pops off the registers it saved earlier

```
popq   %r13   # Remember that the stack is FILO! popq     %r12
popq   %rbx
```

Finally, the program should return to the call site, using the ret instruction:

```
ret
```

Summary: Putting it together, the code for a function should look like this:

```
foo:

pushq  %rbx    #        Save registers, if needed
pushq  %r12
pushq  %r13
subq   $0x18, %rsp   #        Allocate stack space

# Function body

addq   $0x18, %rsp   #        Deallocate stack space
popq   %r13    #        Restore registers
popq   %r12
popq   %rbx ret      #        Pop return address and return control
               #        to caller
```

4.4.3 Dynamic stack allocation

You may find that having a static amount of stack space for your function does not quite cut it. In this case, we will need to borrow a tradition from 32-bit x86 and save the base of the stack frame into the base pointer register. Since %rbp is a callee-save register, it needs to be saved before you change it. Therefore, the function prologue will now be prefixed with:

```
pushq  %rbp

movq   %rsp, %rbp
```

Consequently, the epilogue will contain this right before the ret:

```
movq   %rbp, %rsp

popq   %rbp
```

Secrets: Outlandish, Polaris
Red = not taught this year

This can also be done with a single instruction, called leave. The epilogue makes sure that no matter what you do to the stack pointer in the function body, you will always return it to the right place when you return. Note that this means you no longer need to add to the stack pointer in the epilogue.

This is an example of a function which allocates between 8-248 bytes of random stack space during its execution:

pushq
movq pushq pushq subq
... %rbp
%rsp,
%rbx
%r12
$0x18,
%rbp

%rsp #

# # Use base pointer

Save registers

Allocate some stack space
call andq rand
$0xF8,

%rax # # # Get random number
Make sure the value is 8-248 bytes and aligned on 8 bytes
subq %rax, %rsp # Allocate space
…
movq movq movq (%rbp), %r12 0x8(%rbp), %rbx
%rbp, %rsp #

# Restore registers from base of frame

Reset stack pointer and restore base

popq
%rbp ret # pointer

This sort of behavior can be accessed from C code by calling pseudo-functions like alloca,

Secrets: Outlandish, Polaris
Red = not taught this year

which allocates stack space according to its argument.
More information about the stack frame and function calls can be found on pages 219-232 of the textbook.

Function Call Order:

| Argv: | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| Recall: | edi | esi | edx | ecx | r8d | r9d |

Things to include at the top of C++ file:
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>