Question F5: Caching [10 pts]

We have 16 KiB of RAM and two options for our cache. Both are two-way set associative with 256 B blocks, LRU replacement, and write-back policies. **Cache A** is size 1 KiB and **Cache B** is size 2 KiB.

(A) Calculate the TIO address breakdown for **Cache B**: [1.5 pt]

Tag bits	Index bits	Offset bits
4	2	8

14 address bits. $\log_2 256 = 8$ offset bits. 2 KiB cache = 8 blocks. 2 blocks/set → 4 sets.

(B) The code snippet below accesses an integer array. Calculate the **Miss Rate** for **Cache A** if it starts *cold*. [3 pt]

```
#define LEAP 4
#define ARRAY_SIZE 512
int nums[ARRAY_SIZE];          // &nums = 0x0100 (physical addr)
for (i = 0; i < ARRAY_SIZE; i+=LEAP)
    nums[i] = i*i;
```

$\boxed{1/16}$

Access pattern is a single write to `nums[i]`. Stride = LEAP = 4 ints = 16 bytes. 256/16 = 16 strides per block. First access is a compulsory miss and the next 15 are hits. Since we never revisit indices, this pattern continues for all cache blocks. You can also verify that the offset of &nums is 0x00, so we start at the beginning of a cache block.

(C) For each of the proposed (independent) changes, write **MM** for "higher miss rate", **NC** for "no change", or **MH** for "higher hit rate" to indicate the effect on **Cache A** for the code above:[3.5 pt]

Direct-mapped	_NC_	Increase block size	_MH_
Double LEAP	_MM_	Write-through policy	_NC_

Since we never revisit blocks, associativity doesn't matter. Larger block size means more strides/block. Doubling LEAP means fewer strides/block. Write hit policy has no effect.

(D) Assume it takes 200 ns to get a block of data from main memory. Assume **Cache A** has a hit time of 4 ns and a miss rate of 4% while **Cache B**, being larger, has a hit time of 6 ns. What is the worst miss rate Cache B can have in order to perform as well as Cache A? [2 pt]

$\boxed{\textbf{0.03 or 3\%}}$

$AMAT_A = HT_A + MR_A \times MP = 4 + 0.04*200 = 12$ ns.

$AMAT_B = HT_B + MR_B \times MP \leq 12 \;\rightarrow\; 200\, MR_B \leq 6 \;\rightarrow\; MR_B \leq 0.03$

Name:____

4. *Processes* (**12** points) In this problem, assume Linux.

   (a) Can the same program be executing in more than one process simultaneously?

   (b) Can a single process change what program it is executing?

   (c) When the operating system performs a context switch, what information does *NOT* need to be saved/maintained in order to resume the process being stopped later (circle all that apply):

   - The page-table base register
   - The value of the stack pointer
   - The time of day (i.e., value of the clock)
   - The contents of the TLB
   - The process-id
   - The values of the process' global variables

   (d) Give an example of an exception (asynchronous control flow) in which it makes sense to later re-execute the instruction that caused the exception.

   (e) Give an example of an exception (asynchronous control flow) in which it makes sense to abort the process.

   Solution:

   (a) Yes (the question is ambiguous as to what "simultaneous" means. We clarified during the exam, "Assume it *is* the case that multiple processes execute simultaneously. Then the question is whether more than one of these processes can be executing the same program." Under this interpretation, only "yes" is plausibly correct.)

   (b) Yes

   (c) The time of day and the contents of the TLB

   (d) Page fault for memory on disk (other answers possible; full credit given just for page-fault even though that's ambiguous)

   (e) Division by zero (other answers possible)

Question F7: Processes [9 pts]

(A) The following function prints out four numbers. In the following blanks, list three possible outcomes: [3 pt]

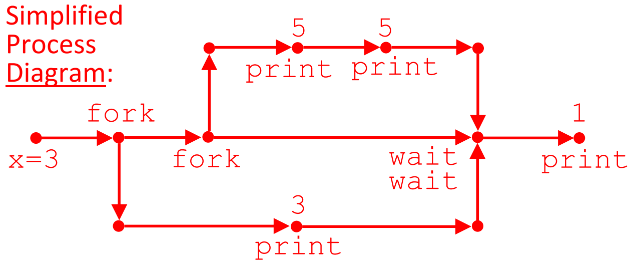
```
void concurrent(void) {
   int x = 3, status;
   if (fork()) {
      if (fork() == 0) {
         x += 2;
         printf("%d",x);
      } else {
         wait(&status);
         wait(&status);
         x -= 2;
      }
   }
   printf("%d",x);
   exit(0);
}
```

(1) _3, 5, 5, 1______

(2) _5, 3, 5, 1______

(3) _5, 5, 3, 1______

Simplified Process Diagram:



(B) For the following examples of exception causes, write "**N**" for intentional or "**U**" for unintentional from the perspective of the user process. [2 pt]

System call	__N__	Hardware failure	__U__
Segmentation fault	__U__	Mouse clicked	__U__

Syscalls are part of code you are executing. The others are external to the process.

(C) Briefly define a **zombie** process. Name a process that can *reap* a zombie process. [2 pt]

Zombie process: A process that has ended/exited but is still consuming system resources.
Reaping process: The parent process or `init/systemd` (PID 1).

(D) In the following blanks, write "**Y**" for yes or "**N**" for no if the following need to be updated when **execv** is run on a process. [2 pt]

Page table __Y__ PTBR __N__ Stack __Y__ Code __Y__

The process already has its own page table, so while we will need to invalidate PTEs from the old process image, we don't need to create another page table, so the PTBR can remain the same. We replace/update the old process image's virtual address space, including Stack and Code.

Question M1: Number Representation [8 pts]

(A) Take the 32-bit numeral **`0xC0800000`**. Circle the number representation below that has the *negative* value for this numeral. [2 pt]

Floating Point [✓ Sign & Magnitude] Two's Complement Unsigned

Unsigned:	Can only represent positive numbers.
Floating Point:	S = 1 and E = 10000001_2 → Exp = 2, so a small negative number.
Sign & Mag:	Negative number with magnitude 100 0000 $10...0_2$.
Two's:	Negative number with magnitude 011 1111 $10...0_2$ (flip bits + 1).

(B) Let `float f` hold the value $\mathbf{2^{20}}$. What is the *largest power of 2* that gets rounded off when added to `f`? Answer in exponential form, not just the exponent. [2 pt]

23 bits in M, so need 24th power less than 2^{20} to get rounded off.

$\boxed{\mathbf{2^{-4}}}$

Traffic lights display three basic colors: red (R), yellow (Y), and green (G), so we can use them to encode base 3! We decide to use the encoding 0↔R, 1↔Y, 2↔G. For example, $5 = 1\times3^1+2\times3^0$ would be encoded as **YG**. Assume each traffic light can only display one color at a time.

(C) What is the *unsigned* decimal value of the traffic lights displaying **RGYY**? [2 pt]

$0 \times 3^3 + 2 \times 3^2 + 1 \times 3^1 + 1 \times 3^0 = 18 + 3 + 1 = 22.$

$\boxed{\mathbf{22}}$

(D) If we have **9 bits** of binary data that we want to store, how many *traffic lights* would it take to store that same data? [2 pt]

9 bits represents 512 things. Powers of 3: 1, 3, 9, 27, 81, 243, 729.

$\boxed{\textbf{6 traffic lights}}$

Question M2: Design Question [2 pts]

(A) The machine code for x86-64 instructions are variable length. Name one advantage and one disadvantage of this design decision. [2 pt]

Advantage:	Machine code/Code section of memory is more compact (don't need to pad). No limit on number of instructions in ISA.
Disadvantage:	Harder to tell/find where to read next instruction. Need more complex hardware to fetch and/or decode instructions.

3. Virtual Memory (9 points)

Assume we have a virtual memory detailed as follows:

- 256 MiB Physical Address Space
- 4 GiB Virtual Address Space
- 1 KiB page size
- A TLB with 4 sets that is 8-way associative with LRU replacement

For the following questions it is fine to leave your answers as powers of 2.

a) How many bits will be used for:

Page offset? _____**10**______

Virtual Page Number (VPN)? ____**22**_____ Physical Page Number (PPN)? ___**18**______

TLB index? _______**2**_________ TLB tag? _______**20**__________

b) How many entries in this page table?

$$2^{22}$$

c) We run the following code with an empty TLB. Calculate the TLB miss rate for data (ignore instruction fetches). Assume **`i`** and **`sum`** are stored in registers and **`cool`** is page-aligned.

```
#define LEAP 8
int cool[512];
... // Some code that assigns values into the array cool
... // Now flush the TLB. Start counting TLB miss rate from here.
int sum;
for (int i = 0; i < 512; i += LEAP) {
  sum += cool[i];
}
```

TLB Miss Rate: (fine to leave you answer as a fraction) ____ $\frac{1}{32}$ __________

2. Buffer Overflow (15 points)

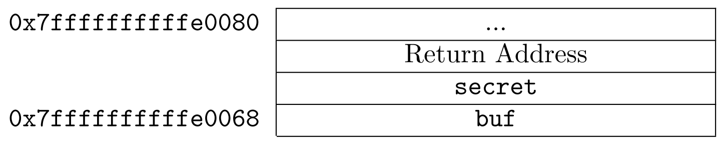
The following code runs on a 64-bit x86 Linux machine. The figure below depicts the stack at point `A` before the function `gatekeeper()` returns. The stack grows downwards towards lower addresses.

```
void get_secret(char*);
void unlock(void);
void backdoor(void);

void gatekeeper() {
    char secret[8];
    char buf[8];

    fill_secret(secret);
    gets(buf);

    if (strcmp(buf, secret) == 0)
        unlock();
A:
    return 0;
}
```



You are joining a legion of elite hackers, and your final test before induction into the group is gaining access to the CIA mainframe. `gatekeeper()` is a function on the mainframe that compares a password you provide with the system's password. If you try to brute force the password, you will be locked out, and your hacker reputation will be tarnished forever.

Assume that `fill_secret()` is a function that places the mainframe's password into the `secret` buffer so that it can be compared with the user-provided password stored in `buf`.

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.

`strcmp()` is a function that returns 0 if two (null-terminated) strings are the same.

(a) Explain why the use of the `gets()` function introduces a security vulnerability in the program.

`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.

(b) You think it may be possible to unlock the mainframe, even without the correct password. Provide a hexadecimal representation of an attack string that causes the `strcmp()` call to return `0`, such that `unlock()` is then called. `gatekeeper()` should return normally, as to avoid raising any suspicion.

Essentially the buffers `secret` and `buf` need to be the same. Thus, any identical null-terminated 8-byte strings will work, as long as they don't contain premature 00's, for this counts as a null-terminator. For example, `0xffffffffffffff00ffffffffffffff00` works.

(c) The function `backdoor()` is located at address `0x0000000000000351`. Construct a string that can be given to this program that will cause the function `gatekeeper()` to unconditionally transfer control to the function `backdoor()`. Provide a hexadecimal representation of your attack string.

Fill up buf, fill up secret, replace return address with `backdoor`'s address. For example:

`0x 1234567812345678 1234567812345678 5103000000000000`

(d) How should the program be modified in order to eliminate the vulnerabilities the function `gets()` introduces?

Various answers accepted here, such as using `fgets()` instead, which has the string length as a parameter.

(e) Describe two types of protection operating systems and compilers can provide against buffer overflow attacks. Briefly explain how each protection mechanism works.

OS:

- memory protection

Compiler:

- stack canaries
- compile-time string length checks
- stack address randomization

4. Processes (10 points)

(a) After a context switch, the VPN to PPN mappings in the TLB from the previous running process no longer apply. A simple solution to this problem is to "shoot down" the TLB, by invalidating all the entries in the TLB, but this can often cause inefficiency if there is frequent context switching.

What additional information can be added to the TLB that can be utilized to reduce this inefficiency in the TLB on a context switch? *Hint: consider how processes can be uniquely identified by the MMU.*

Tagging TLB entries with the unique process ID

(b) Suppose you are in control of the CPU and operating system, and you realize that you have a process A that requires a large uninterrupted chunk of CPU time to perform its important work. What would you adjust to ensure that this can happen?

Any of these answers were accepted:

- Extend the context switch timer
- Avoid context switch
- Increase the priority of process A
- Ignore interrupts

One important consideration that a lot of answers did not take into account is that we are in control of the CPU and not another user process, so solutions like using `waitpid()` or other system calls are not correct.

(c) Consider an OS running a process A which incurs a timer interrupt at time t_1. The OS context switches to some other processes which do some work. Later, the OS context switches back to process A at time t_2. Note that process A was not run between t_1 and t_2.

Circle the items which are guaranteed to be the same at time t_1 and t_2.

Underlined items are guaranteed to be the same.

- **Register `%rbx`**: At time t_1, the OS will save all of the register values for process A, including `%rbx`. At time t_2 when the OS context switches back, it restores the value of `%rbx`. Thus we are guaranteed to have the same saved value.
- **Process A's Page Table**: Between t_1 and t_2, another process could have evicted one of process A's pages that was in physical memory, thus altering its value.
- **Instruction pointer**: Similar to register `%rbx`, this register is also explicitly saved. One point of clarification is that a context switch will only occur when a particular instruction is complete. So, at t_1 `%rip` will point to the next instruction that has **not been executed**. At t_2 when the OS context switches back, this value will still be the same so that it will in fact execute that exact instruction.
- **L1 Cache**: The L1 cache can have been changed by other processes' accesses to memory.
- **Page fault handler code**: The page fault handler code is part of the kernel (OS code) so this will not change.
- **Page Table Base Register**: The PTBR contains the location in physical memory of the page table for the currently executing process, which will not change from t_1 to t_2, so this is guaranteed to be the same.

(d) Consider the following C program, running on an x86-64 Linux machine. The program starts running at function `main`. Assume that `printf` flushes immediately.

```
int main() {
    int* x = (int*) malloc(sizeof(int));
    *x = 1;
    if (fork() == 0) {
        spoon(x);
    } else {
        *x = 8 * *x;
        printf("%d\n", *x);
    }
}

void spoon(int* x) {
    printf("%d\n", *x);
    if (fork() == 0) {
        *x = 2 * *x;
    } else {
        *x = 4 * *x;
    }
    printf("%d\n", *x);
}
```

Provide **two** possible outputs of running this code.

Output 1: Output 2:

The main sources of error were:

- Each process has its own copy of `x`, so the largest possible output is 8.
- The first `printf` in `spoon` will always print the `1` before the `2` or `4`.

All possible outputs:

- 8 1 2 4
- 8 1 4 2
- 1 8 2 4
- 1 8 4 2
- 1 2 8 4
- 1 4 8 2
- 1 2 4 8
- 1 4 2 8

Question 4: Procedures & The Stack [24 pts.]

Consider the following x86-64 assembly and C code for the recursive function `rfun`.

```
// Recursive function rfun
long rfun(char *s) {
  if (*s) {
    long temp = (long)*s;
    s++;
    return temp + rfun(s);
  }
  return 0;
}

// Main Function - program entry
int main(int argc, char **argv) {
  char *s = "CSE351";
  long r = rfun(s);
  printf("r: %ld\n", r);
}
```

```
00000000004005e6 <rfun>:
  4005e6: 0f b6 07               movzbl (%rdi),%eax
  4005e9: 84 c0                  test   %al,%al
  4005eb: 74 13                  je     400600 <rfun+0x1a>
  4005ed: 53                     push   %rbx
  4005ee: 48 0f be d8            movsbq %al,%rbx
  4005f2: 48 83 c7 01            add    $0x1,%rdi
  4005f6: e8 eb ff ff ff         callq  4005e6 <rfun>
  4005fb: 48 01 d8               add    %rbx,%rax
  4005fe: eb 06                  jmp    400606 <rfun+0x20>
  400600: b8 00 00 00 00         mov    $0x0,%eax
  400605: c3                     retq
  400606: 5b                     pop    %rbx
  400607: c3                     retq
```

(A) How much space (in bytes) does this function take up in our final executable? [2 pts.]

34 Bytes

Count all bytes (middle column) or subtract address of first instruction (0x4005e6) from last instruction (0x400607), then add 1 byte for the retq instruction.

(B) The compiler automatically creates labels it needs in assembly code. How many labels are used in `rfun` (including the procedure itself)? [2 pts.]

3

The addresses 0x4005e6, 0x400600 (Base Case), 0x400606 (Exit)

(C) In terms of the C function, what value is being saved on the stack? [2 pts.]

`*s`

The movsbq instruction at 0x4005ee puts *s into %rbx, which is pushed onto the stack by the pushq instruction at 0x4005ed.

(D) What is the return address to `rfun` that gets stored on the stack during the recursive calls (in hex)? [2 pts.]

`0x4005fb`

(E) Assume main calls `rfun` with `char *s = "CSE351"` and then prints the result using the `printf` function, as shown in the C code above. Assume `printf` does not call any other procedure. Starting with (and including) `main`, how many total stack frames are created, and what is the maximum depth of the stack? [2 pts.]

Total Frames: **8**	Max Depth: **7**

```
main -> rfun(s) -> rfun(s+1) -> rfun(s+2) -> rfun(s+3) -> rfun(s+4) -> rfun(s+5)
     -> printf()
```

```
The recursive call to rfun(s+6), which handles the null-terminator in the string does not
create a stack frame since we consider the return address pushed to the stack during a
procedure call to be part of the caller's stack frame. When handling the null character, the
byte is discovered to be null and the je instruction at 0x4005eb is taken, transferring
control to the mov instruction at 0x400600, which stores 0 in %rax and returns. Thus, the base
case of the recursive function does not create a stack frame.
```

(F) Assume main calls `rfun` with `char *s = "CSE351"`, as shown in the C code. After `main` calls `rfun`, we find that the return address to `main` is stored on the stack at address `0x7fffffffdb38`. On the first call to `rfun`, the register `%rdi` holds the address `0x4006d0`, which is the address of the input string "CSE351" (i.e. `char *s == 0x4006d0`). Assume we stop execution prior to executing the `movsbq` instruction (address `0x4005ee`) during the **fourth** call to `rfun`. [14 pts.]

*For each address in the stack diagram below, fill in both the **value** and a **description** of the entry.*

*The **value** field should be a hex value, an expression involving the C code listed above (e.g., a variable name such as `s` or `r`, or an expression involving one of these), a literal value (integer constant, a string, a character, etc.), "unknown" if the value cannot be determined, or "unused" if the location is unused.*

*The **description** field should be one of the following: "Return address", "Saved %reg" (where reg is the name of a register), a short and descriptive comment, "unused" if the location is unused, or "unknown" if the value is unknown.*

Memory Address	Value	Description
`0x7fffffffdb48`	`unknown`	`%rsp when main is entered`
`0x7fffffffdb38`	`0x400616`	`Return address to main`
`0x7fffffffdb30`	`unknown`	`original %rbx`
`0x7fffffffdb28`	`0x4005fb`	`Return address`
`0x7fffffffdb20`	`*s, "C", 0x43`	`Saved %rbx`
`0x7fffffffdb18`	`0x4005fb`	`Return address`
`0x7fffffffdb10`	`*s, *(s+1), "S", 0x53`	`Saved %rbx`
`0x7fffffffdb08`	`0x4005fb`	`Return address`
`0x7fffffffdb00`	`*s, *(s+2), "E", 0x45`	`Saved %rbx`

CSE 351 Reference Sheet (Midterm)

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

IEEE 754 FLOATING-POINT STANDARD

Value: ±1 x Mantissa x 2^{Exponent}

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

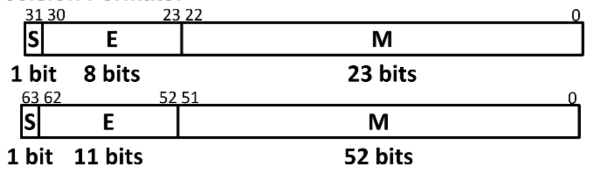
where Single Precision Bias = 127,
Double Precision Bias = 1023.

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.
`leaq 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 `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	Condition Codes	(op) s, d	test a, b	cmp a, b
je “Equal”	ZF	d (op) s == 0	b & a == 0	b == a
jne “Not equal”	~ZF	d (op) s != 0	b & a != 0	b != a
js “Sign” (negative)	SF	d (op) s < 0	b & a < 0	b-a < 0
jns (non-negative)	~SF	d (op) s >= 0	b & a >= 0	b-a >= 0
jg “Greater”	~(SF^OF) & ~ZF	d (op) s > 0	b & a > 0	b > a
jge “Greater or equal”	~(SF^OF)	d (op) s >= 0	b & a >= 0	b >= a
jl “Less”	(SF^OF)	d (op) s < 0	b & a < 0	b < a
jle “Less or equal”	(SF^OF) | ZF	d (op) s <= 0	b & a <= 0	b <= a
ja “Above” (unsigned >)	~CF & ~ZF	d (op) s > 0U	b & a < 0U	b > a
jb “Below” (unsigned <)	CF	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

Sizes

C type	x86-64 suffix	Size (bytes)
char	b	1
short	w	2
int	l	4
long	q	8