Text in ASCII coded to Hexadecimal:

second character \ first character	2	3	4	5	6	7
0	␣	0	@	P	`	p
1	!	1	A	Q	a	q
2	"	2	B	R	b	r
3	#	3	C	S	c	s
4	$	4	D	T	d	t
5	%	5	E	U	e	u
6	&	6	F	V	f	v
7	'	7	G	W	g	w
8	(	8	H	X	h	x
9	)	9	I	Y	i	y
A	*	:	J	Z	j	z
B	+	;	K	[	k	{
C	,	<	L	\	l	|
D	-	=	M	]	m	}
E	.	>	N	^	n	~
F	/	?	O	_	o	DEL

(*) Data is stored in stack and registers in reversed format:

0x7501ca75 will be stored as

75	ca	01	c0

8bit → al; ax = 16bits; 16bits; eax = 32bits; rax = 64bits

registers used:

8086: ax, cx, dx, bx, sp, bp, si, di

x86: eax, ecx, edx, ebx, esp, ebp, esi,

amd 64: rax, rbx, rcx, rdx, rsp, rbp, rsi, rdi, r8, r9, r10, r11, r12, r13, r14, r15.

x86 x64 processor

all three used for instruction pointer ← Special purpose reg: eip (x86), rip (amd 64), r15 (arm).

Teacher's Sign. : _____

→ Set registers:

```
.intel_syntax noprefix
.global _start
_start:
 mov rdi, 0x1337
 mov rax, 60
 syscall
```

Save it and run as arg after performing all steps from before via /challenge/run.

Here we set rdi to '0x1337'.

→ Set Multiple Registers:

```
.intel_syntax noprefix
.global _start
 _start:
 mov rax, 0x1337
 mov r12, 0xCAFEB00D1337BEEF
 mov rsp, 0x31337
 ~~mov rax, 60~~
 ~~syscall~~
```

Save as assembly and run as arg.

Did not perform exit as it was not intended, we just wanted to set values.

→ Add to Register:

- add rdi, 0x1337 # similar to ~~rdi~~ rdi = rdi + 0x1337.

Now,

```
.intel_syntax noprefix
.global _start
_start:
 add rdi, 0x331337
```

Save and run as arg.

→ Linear equation registers:

we are given linear equation:
rax = rdi x rsi + rdx

```
.intel-syntax noprefix
.global _start
_start:
 imul rdi, rsi
 add rdi, rdx
 mov rax, rdi
```

save and run

→ Integer-Division:

~~1~~ mov rax, reg-1
div reg-2

'div' is a special instruction that can divide a 128-bit dividend by a 64-bit divisor while storing both 'quotient' and 'remainder' using only one register as an operand.

This complex 'div' takes place as follows:

div reg
↓ interprets.

rax = rdx : rax / reg

rdx = remainder.

'rdx:rax' means rdx will be upper 64-bits of 128 bits divider and rax will be lower 64-bits of 128-bits dividend.

We must be careful about what is in rdx and rax, before we call div.

Now;

Save, compile and execute

```
.intel_syntax noprefix
.global _start
_start:
mov rax, rdi        → done to make sure it is dividend as rax is default dividend.
mov rdx, 0          → done to set highest 64-bits saved as '0' as they mentioned it is 64-bits in size.
div rsi             ← Done to divide 'rax' with rsi as divisor and store results in 'rax'.
```

→ Modulo Operation:

Modulo is 'Remainder' in 'C' we represent it as '%' and it gives remainder of any division operation.

Now,

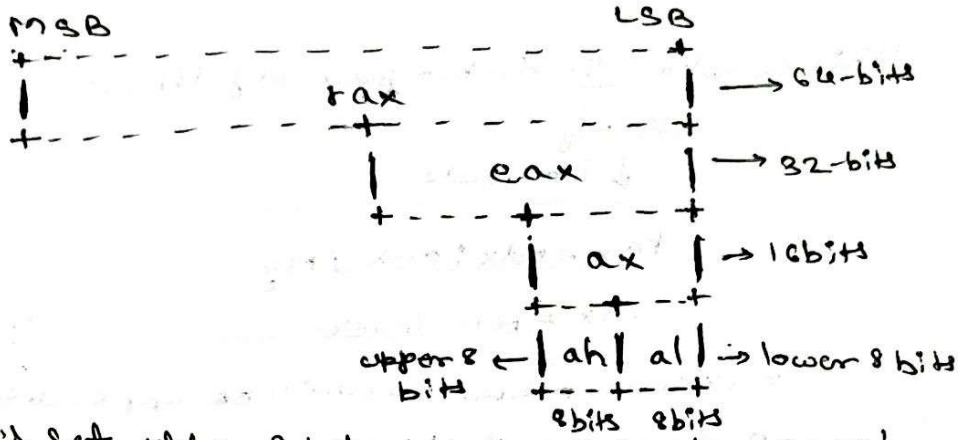
Save and run to

```
.intel_syntax noprefix
.global _start
_start:
mov rax, rdi        ← here we move rdi into rax for default dividend
mov rdx, 0          ← we set upper 64 bits of 'rdx' 0
div rsi             ← divide 'rax' with 'rsi' as divisor
mov rax, rdx        ← quotient is stored in 'rax' and remainder in 'rdx'; so we mov 'rdx' into 'rax'. for getting remainder.
```

Teacher's Sign :

→ Set upper Byte:-

We have till now, set 64-bit registers 'rax, rdi, etc.', we can also access 32-bit registers of same registers using lower 32-bits of 'rax' as 'eax' lower 16-bits of 'rax' as 'ax' and lower 8 bits as 'al'



Now lets set upper 8 bits of 'ax register to '0x42'

```
.intel_syntax noprefix
.global _start
_start:
 mov ah, 0x42
```

} save run

→ Efficient modulo:-

we can perform modulo operation without using 'div' if the number is divisible by '2', just by using mov.

eg:- '77' in binary is '0b1001101'

8 in binary is 2^3 i.e. $= 2^{\wedge}3 = 2^{\wedge}n = n = 3$

∴ 77 % 8 = 5 also last 'n' bits of '77' i.e. last '3' bits of '77' is '101' = 5

Now, rdi %. 256 = rdi %. $2^{\wedge}8$ → last 8bits or lower 8bits stored in rdi

i.e. 'dil'

Now, rsi % 65536 = rsi %. $2^{\wedge}16$ → last 16bits or lower 16bits stored in rsi

i.e. 'si'

Now;

```
.intel_syntax noprefix
.global _start
_start:
 mov al, dil
 mov bx, si
```

} save and run

we move lower 8 bits of rdi i.e 'dil' to compatible 'al' which is lower 8 bits for 'rax'.

similarly 'lower 16 bits of rsi i.e. 'si' to compatible 'bx' which is lower 16 bit for 'rbx'

EXPERIMENT : No.

Page No.
Date

→ Byte-Extraction:

'shl' shifts value in register by specified no. of bits in left direction i.e. most → less significant bit position.

'shr' shifts values in register by specified no. of bits in right direction i.e. least → most significant bit position.

eg: shl al, 1 → shifts value in 'al' by '1' bit to left.

al = 10001010 → al = 00010100

Now;

```
.intel_syntax noprefix
.global _start
_start:
mov rax, rdi
shr rax, 32
shl rax, 56
shr rax, 56
```

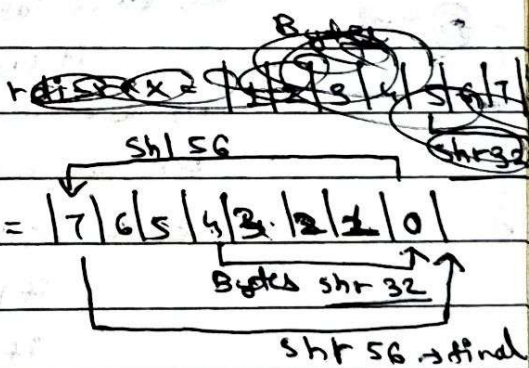
Save and run

keep a copy of rdi in 'rax' so that original value can be found later

shift right by '32 bits' to bring 5th least significant byte to the 0th or starting position

shift left by '56 bits' to bring the 5th least significant byte to the 7th position and cancel out all other bits to '0'

shift right by '56 bits' to bring the byte to 0th position, to be read only that directly.



→ Bit-wise AND:

denoted by 'and', similarly, there exist 'or', 'xor' etc. and does bitwise 'and' operation of every bit in both operands and stored result in the first or destination operand.

Here, 'rax' have all bits stored to '1' for simpler analysis. If you know, you know, we have to do 'rdi AND rsi' and set rax or store result in 'rax' without using 'mov'.

```
.intel_syntax noprefix
.global _start
_start:
 and rdi,rsi
 and rax,rdi
```
save and run

check even? ⟶ Most Hard till now!!

Now, we want to check,

if rdi is even then
 rax = 1
else
 rax = 0

using only 'and, or & xor'

we will first need to set 'rax' to '0' for which we will use 'xor' which set '0' if both bits are same and '1' if different.

`xor rax, rax`

then, we have to make 'rax' equivalent to 'rdi' using only 'or' as '0 or any' is that thing

`or rax, rdi`

then, we know if lowest bit of any no is '0', it is even no. i.e. divisible by '2' eg:- '10010010'

and if last bit of any no. is '1', it is odd no. i.e. not divisible by '2' eg '1001011'

Now, we have to keep only lowest or last bit containing even/odd info.

so we will perform 'and'

`and rax, 1`

as, if last bit of rax is '0' (even) and 1 → 0

if last bit of rax is '1' (odd) and 1 → 1.

Now we have only last bit info i.e. if no. is even '0' or if odd '1'

but, we want rax '1' if even & '0' if odd i.e. flipped.

so we will use 'xor', as, if last bit of rax is '0' (even) xor 1 → 1 (set to)

if last bit of rax is '1' (odd) xor 1 → 0 (set to)

`xor rax, 1`

So final code:-

```
.intel_syntax noprefix
.global _start
_start:
 xor rax, rax
 or rax, rdi
 and rax, 1
 xor rax, 1
```
save and run to

EXPERIMENT : No.

Page No.
Date

→ Memory Read:

In x86 x64, we can access things at a memory location, called dereferencing, such as:

mov rax, [0xdeadbeef] → Move value stored at address into 'rax'.

mov rax, [rdi] → Move value stored at address of what rdi holds into 'rax'

mov [rax], rdi → Move value stored in rdi to address of what 'rax' holds.

Note: x86-x64 has memory which goes from '0' to '0xffffffffffffffff (very huge).

Now, we have to place, value stored at '0x404000' into rax'. Make sure 'rax' is original value stored at '0x404000'

Now:

```
.intel_syntax noprefix
.global _start
_start:
mov rax, [0x404000]
```

} Save and run

→ Memory Write:

Now, we want to write value stored at 'rax' into memory address of location '0x404000'.

Now:

```
.intel_syntax noprefix
.global _start
_start:
mov [0x404000], rax
```

} Save and run

Teacher's Sign. :

→ Memory Increment:

Now, we have to store value at address '0x404000' into 'rax' and then increment value stored at address '0x404000' by '0x1337'.

We can do this using two ways:

1st: (Complex)

```
.intel_syntax noprefix
.global _start
_start:
mov rax, [0x404000]
add QWORD PTR [0x404000], 0x1337.
```

} save and run

Here, we move value stored at '0x404000' into 'rax'. and then add '0x1337' directly into value stored at '[0x404000]' using 'QWORD' for Quadword, which denotes type of word or no. of bytes and 'PTR' or pointer which points to addr [0x404000] → in x64 : 1 = 64 bits i.e. 8 bytes

2nd: (Simple)

```
.intel_syntax noprefix
.global _start
_start:
mov rax, [0x404000]
mov rbx, rax
add rbx, 0x1337
mov [0x404000], rbx
```

} Save & run

→ Byte Access:

As we know, x86-64 have every memory address storing '64-bits' or '8 bytes' or 'Quadword'.

Some name references:

Quad word = 8 bytes = 64 bits
Double word = 4 bytes = 32 bits
words = 2 byte = 16 bits
Byte = 1 byte = 8 bits.

So,

mov al, [address] → moves least significant 'byte' from addr to 'rax'

mov ax, [address] → moves least significant 'word' from addres to 'rax'

mov eax, [address] → moves least significant 'double' from addr to 'rax'

mov rax, [address] → moves full 'Quadword' from addr to 'rax'

Now, we have to set 'rax' to 'byte' at '0x404000'

```
.intel_syntax noprefix
.global _start
_start:
mov al, [0x404000]
```

} save and run

EXPERIMENT : No.

Page No.
Date

→ Memory Size Access

Now, we have to set
~~rax to byte at '0x404000'~~
~~rax to word at '0x404000'.~~
rcx to double at '0x404000'
rdx to quadword at '0x404000'.

So;

```
.intel_syntax noprefix
.global _start
_start:
mov al, [0x404000]
mov bx, [0x404000]
mov ecx, [0x404000]
mov rdx, [0x404000]
```

Save and run

→ Little Endian Write:

As we know, values are stored in reverse order in registers or memory location.

eg: [0x1330] = 0x00000000deadcode

This will stored into memory as:

~~[0x1330] = 0xde~~
~~[0x1331] = 0xc0~~

[0x1330] = 0xde
[0x1331] = 0xc0
[0x1332] = 0xad
[0x1333] = 0xde
[0x1334] = 0x00
[0x1335] = 0x00
[0x1336] = 0x00
[0x1337] = 0x00

stored in reversed order to make sure that ~~operational performance~~ is maximized eg: consider it in stack. stored every value in chunk as shown here. This is called as 'Little Endian' in x86.

Sundaram

Teacher's Sign. :

Now, we have to set,

[rdi] = 0xdeadbeef00001337
[rsi] = 0xcoffee0000

we can do this using the previously used QuadWord PTR
mov QWORD PTR [ ], value;
but it wont work since exact memory-size is not QWORD and it's hard to guess just by looking at hex-encoded values.
So, we will set the values in general purpose registers and mov those register's values to the address at which the '[rdi] and [rsi]' points; This will ensure that registers know the exact memory requirements.
So;

```
.intel_syntax noprefix
.global _start
_start:
 mov rax, 0xdeadbeef00001337
 mov rbx, 0xcoffee0000
 mov [rdi], rax
 mov [rsi], rbx
```

save and run

monday
12/01/2026

→ Memory Swap

as we know, memory is stored in linear manner, 'Little endian' as mentioned in previous challenge & concept

Due to this pattern of storage, we can access only required values or things from memory using offsets.
lets say 5th byte:
mov al, [address+4];
Now, we have to;
- Load two consecutive quad words from addresses stored in rdi.
- Calculate sum of previous steps quad words.
- Store sum at address in rsi.

Save and run

```
.intel_syntax noprefix
.global _start
_start:
 mov rax, [rdi] → first quad word at '[rdi]'
 mov rbx, [rdi+8] → second quad word at '[rdi]'
 add rax, rbx
 mov[rsi], rax
```

→ Stack Subtraction

Stack is region of memory that can store values for later.
Push → to store a value in stack.
pop → to retrieve a value from stack.
Follows LIFO.
On x86, pop instruction will take value from top of stack and put it into register.
Similarly, push instruction will take value from register and put it onto top of stack.

EXPERIMENT : No.

Page No.
Date

Now, take top value of stack, subtract rdi from it, push it back.

save and run

```
.intel-syntax noprefix
.global start
_start:
    pop rax  ⟶ pops value at top and store in 'rax'
    sub rax, rdi  → subs and stores in 'rax'
    push rax  ⟶ push value back into stack.
```

⟶ Swap stack Values:

Now; swap values in rdi & rsi

i.e. rdi = 2 & rsi = 5 ⟹ rdi = 5 & rsi = 2

save and run

```
.intel-syntax noprefix
.global _start
_start:
    push rdi
    push rsi
    pop rdi
    pop rsi
```

⟶ Average stack Values:

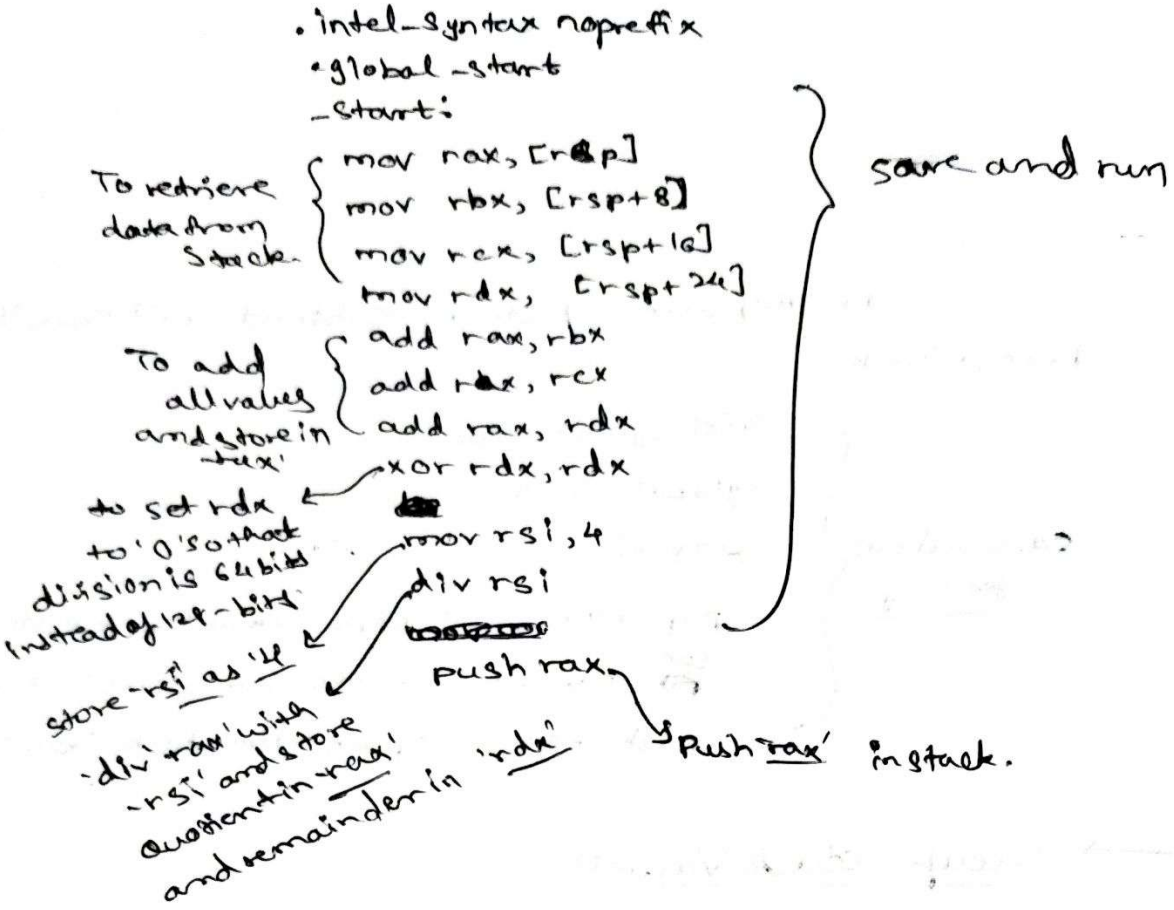
'rsp' is stack pointer which always points at top of stack i.e. memory address of last value pushed.

we can use [rsp] to access value at memory address of rsp.

Now;

Without using pop, calculate average of 4 consecutive quad words stored on stack.

Teacher's Sign. : _____



→ Absolute Jumps

Now, we will be working with control flow manipulations. This involves using instructions to both directly and indirectly control special register 'rip' (register Instruction pointer). we will use 'Jmp, call, cmp' and their alternatives.

earlier we learnt to control data in pseudo-control ways but X86-64 gives us actual instructions to manipulate Control flow directly.

Two major ways to Control flow:

① Through a Jump.

② Through a Call.

Here we will learn about Jumps

Two Types of Jumps:

↳ Conditional Jumps: Depends on results of earlier instruction

↳ Unconditional Jumps: Does not depend on results of earlier instruction.

For all Jumps there are 3 types:

① Relative Jump: jump + or − the next instruction.

② Absolute Jumps: jump to a specific address.

③ Indirect Jumps: jump to memory address specified in a register.

Tuesday 13/01/2026

EXPERIMENT : No.

Page No.
Date

Now, let's Jump to absolute address: 0x403000

```
.intel_syntax noprefix
.global _start
_start:
mov rax, 0x40300      <- store address in general purpose register before jump (mandatory)
jmp rax
```

Save & run

→ Relative Jumps:

In relative jumps, we tell CPU to "Jump forward a certain number of bytes from where you are currently executing". This is useful because our code can move in memory and jump will still reach correct target.

Tools used for relative jumps:

→ Labels: Instead of calculating addresses manually, you can use labels as placeholders. Assembler will automatically calculate off-set from jump instruction to label.

→ nop: No Operation → A single byte instruction that does nothing. It is predictable in size and can be used as fillers to create an exact distance for your jumps.

→ .rept: Repeat Directive → A directive that tells assembler to repeat given instruction multiple times.

Now:

① Make first instruction jump

② make that jmp relative jmp of exactly 0x51 bytes from current instruction.

Sundaram

Teacher's Sign. :

② Fill space between jump and destination with nop instruction using .rept.
① At label where jmp lands, set rax to 0x1.

→ Now,

```
.intel_syntax noprefix
.global _start
_start:
 jmp after
.rept 0x51
 nop
.endr
after:
 mov rax, 0x1
```

repeat for 51 bytes

end rept

save & run

→ Jump-Trampoliner

Now, create Two Jump trampoliner

→ make first instruction jmp.
→ Make jmp at relative jump to 0x51 bytes from its current position.
→ At 0x51, write full-code:
  → Place top value on stack into register rdi.
  → jmp to absolute address 0x403000.

Now,

```
.intel_syntax noprefix
.global _start
 _start:
 jmp tottheplace
      .rept .0x51
        nop
      .endr
 tottheplace:
        pop rdi
        mov rax, 0x403000
        jmp rax.
```

save & run

→ Conditional Jumps:

let's use conditions to jumps

```
if [x] is 0x7f454c46:
    y = [x+4] + [x+8] + [x+12]
else if [x] is 0x00008A4D:
    y = [x+4] - [x+8] - [x+12]
else:
    y = [x+4] * [x+8] * [x+12]
```

EXPERIMENT : No.

Page No.
Date

Now;

Save & run

```
.intel_syntax noprefix
.global _start
_start:
mov eax, DWORD PTR [rdi]
cmp eax, 0x7F454C46
je forward
cmp eax, 0x0000SA4D
je forwardto
    mov eax, DWORD PTR [rdi+4]
    imul eax, DWORD PTR [rdi+8]
    imul eax, DWORD PTR [rdi+12]
    jmp done
forward:
    mov eax, DWORD PTR [rdi+4]
    add eax, DWORD PTR [rdi+8]
    add eax, DWORD PTR [rdi+12]
    jmp done
forwardto:
    mov eax, DWORD PTR [rdi+4]
    sub eax, DWORD PTR [rdi+8]
    sub eax, DWORD PTR [rdi+12]
done:
```

Wed 14/06/2024

Indirect Jumps

Indirect jumps are used to do switch operations most of the times

eg:

```
switch(number):
0: jmp do_thing_0
1: jmp do_thing_1
2: jmp do_thing_2
default: do_default_thing.
```

If we know the range of the numbers, a switch statement works well for instance, let's take Jump table. A Jump table is contiguous section of memory that holds addresses of places to jump.

In above eg: Jump table would look like:

```
[0x1337] = address of do_thing_0
[0x1337+0x8] = address of do_thing_1
[0x1337+0x10] = address of do_thing_2.
[0x1337+0x18] = address of do_default_thing.
```

Using jump tables, we can greatly reduce amount of 'cmps' we use.
Now all we have to do is check if number is greater than 2.
if yes, then always do:

```
jmp [0x1337+0x18].
```

otherwise:

```
jmp [jump_table_address + number * 8]
```

Using above logic implement:

```
if rdi is 0:
    jmp 0x4030le.
else if rdi is 1:
    jmp 0x4030da
else if rdi is 2:
    jmp 0x4031d5
else if rdi is 3:
    jmp 0x403268
else:
    jmp 0x40332c.
```

Creation of jump_table:

```
.section .data
jump_table:
.quad case 0 - #index 0.
.quad case 1 - # index 1
.quad case 2 #index 2
.quad case 3 #index 3
.quad case 4 ## index 4.
.quad default # default index.
```

Now;

```
.intel_syntax noprefix
.global _start
_start:
cmp rdi, 3          <- Compare rdi with 3
ja default          <- Jump to default if case rdi, 3 is true.
jmp [rsi+rdi*8]     <- else jump to rsi+rdi*8 since '1 byte = 8 bits'
default:
jmp [rsi+32]        <- index '4' since 8x4=32
```

Save and run

→ Average Loop:

Now,

let's compute average on 'n' consecutive quad words, where

rdi = memory address of 1st quad word.

rsi = n (amount to loop for)

rax = average computed

```
.intel_syntax noprefix
.global _start
_start:
mov rax, 0          ← set rax = 0
mov, rbx, rsi       ← copy rsi 'n' in 'rbx' for preserving 'n' value
start_loop:
cmp rbx, 0          ← compare for '0'
je final_output     ← value at rdi.
add rax, [rdi]      ← add rax = rax +
add rdi, 8          ← add rdi = rdi + next 8 bits.
dec rbx             ← decrement 'rbx' by '1'
jmp start_loop      ← repeat all the steps.
final_output:
mov rdx, 0          ← set rdx to '0' to avoid 128bits calculation and allow 64bits calculation.
div rsi
```

default, RDX and RAX → RDX:RAX / rsi → may differ as register

RDX stores remainder

RAX stores quotient

Thursday
15/01/26

→ Count Non-zero :

Here we will implement while loop structure just like we did for loop in previous level.

Now;

Count consecutive non-zero bytes in a contigous region of memory

where: rdi = mem addr of 1st byte

rax = number of consecutive non-zero bytes

If rdi = 0, set rax = 0.

eg: rdi = 0x1000

[0x1000] = 0x41

[0x1001] = 0x42

[0x1002] = 0x43

[0x1003] = 0x44

[0x1004] = 0x00 → stop

∴ rax = 4

Now;

```
.intel_syntax noprefix
.global _start
_start:
    xor rax, rax        ← set rax to '0'
    test rdi, rdi       ← check if rdi is '0'
    je done
loop:
    cmp byte PTR [rdi], 0   ← compare value in rdi with '0'
    je done
    inc rax
    inc rdi
    jmp loop
done:
    mov rax, rax        ← just to end and avoid infinite looping
```

save & run

→ String Lowest

Now, let's hop onto functions: A function is a callable section of code that does not destroy the control flow.

"Call" instruction pushes memory address of next instruction onto the stack and then jumps to the value stored in first argument.

eg:

0x1021 mov rax, 0x400000

0x1028 call rax

0x102a mov [rsi], rax.

① Call pushes 0x102a, address of next instruction, onto stack.

② call jumps to 0x400000, value stored in rax.

'ret' instruction is opposite of call, 'ret' pops top value off of stack and jump to it.

eg:

	stack addr	value
0x1034 mov rax, rdx	RSP+0x8	0xdeadbeef
0x1042 ret	RSP+0x0	0x0000102a

'ret' will jump to '0x102a'.

Now;

```
Str_lower (src_addr):
    i=0
    If src_adr != 0:
        while [src_addr] != 0x00:
            if [src_addr] <= 0x5a:
                [src_addr] = foo ([src_addr])
                i=+1
                src_addr+=1
    return i;
```

Now,

```
.intel_syntax noprefix
.global _start
.global str_lower
_start:
str_lower:
    xor ecx, ecx
    test rdi, rdi
    je done
    mov rsi, rdi
loop:
    mov al, byte ptr [rsi]
```

Teacher's Sign. : _____

```
    cmp al, 0
    je done

    cmp al, 0x5a
    jg skip_foo
    movzx edi, al
    mov rax, 0x403000
    call rax
    mov byte ptr [rsi], al
    inc ecx
skip_foo:
    inc rsi
    jmp loop
done:
  mov eax, ecx
   ret
```

→ Most Common Byte

In previous level, we learned how to make our first function and how to call other functions. Now, we will work with functions that have a function stack frame.

A function stack frame is set of pointers and values pushed onto stack to save things for later use and allocate space on stack for function variables.

'rbp' is a special register, used to tell where our stack frame first started.

Let's say we have to construct some list (a contiguous space of memory) that is only used in our function. list is 5 elements long & each element is a 'dword'. A list of 5 elements would already take 5 registers. So instead we can make space on stack.

```
mov rbp, rsp  → setup base of stack as top.
# move stack 0x14 bytes (5*4) down.
# acts as an allocation.
  sub rsp, 0x14
# assign list[2] = 1337
  mov eax, 1337
  mov [rbp-0xc], eax
# do more ops on list
# restore allocated space.
  mov rsp, rbp
   ret.
```

'rbp' is used to always restore stack where it originally was. if we do not reset or restore stack after use, we will eventually run out. also we 'sub' from 'rsp' because stack grows down.

Now,

```
most_common_byte (src_addr, size):
    i=0
    while i <= size-1:
     curr_byte = [src_addr+i]
     [stack_base - curr_byte*2] += 1
     i+=1
```

b = 0

max_freq = 0

max_freq_byte = 0

while b <= 0xff:

if [stack_base - b*2] > max_freq:

max_freq = [stack_base - b*2]

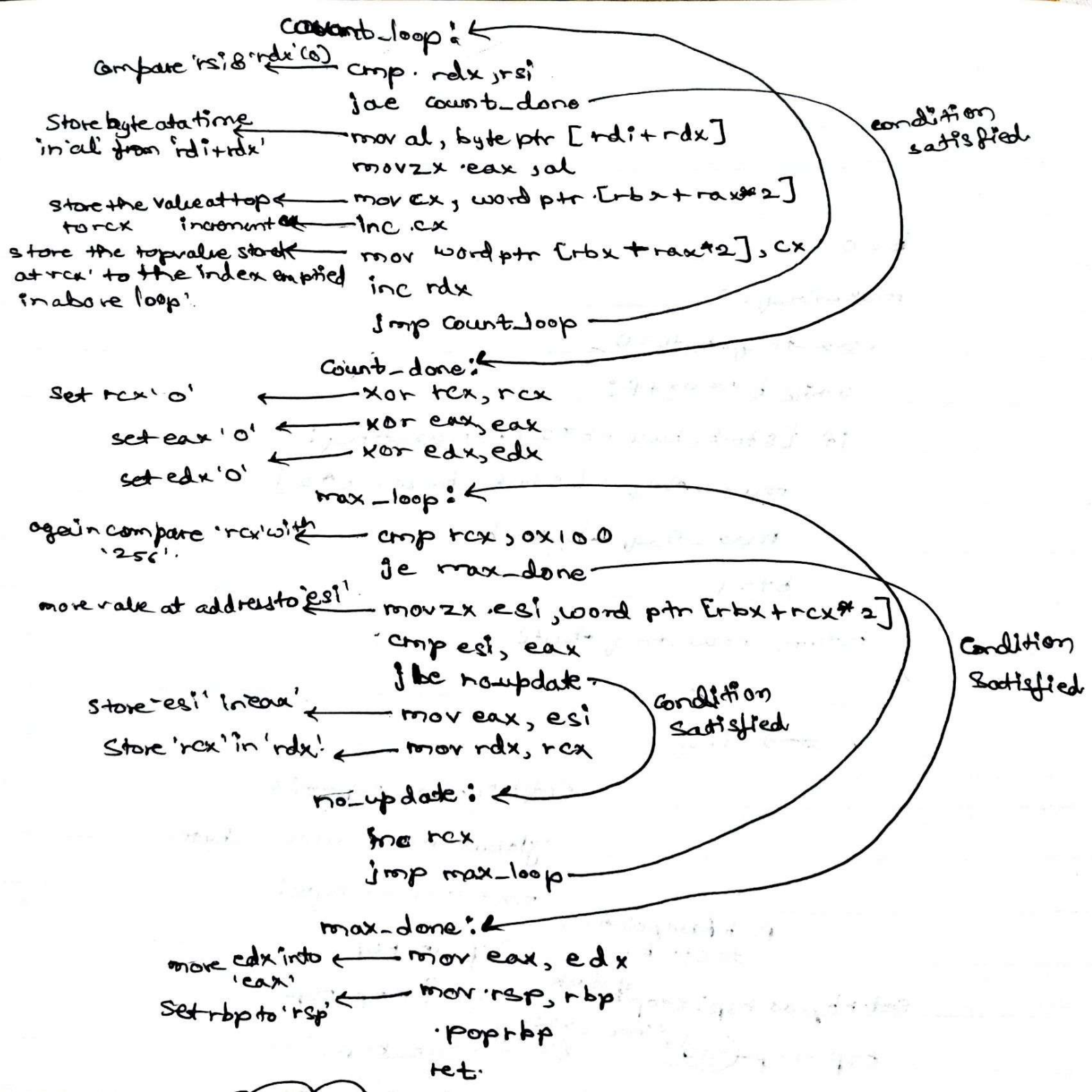
max_freq_byte = b

b += 1

return max_freq_byte

→ Now,

```
.intel_syntax noprefix
.global most_common_byte
most_common_byte:
push rbp                    ← Push base pointer to stack.
mov rbp, rsp                ← set rbp as rsp's i.e top of stack
sub rsp, 512                ← rsp = rsp - (512) → since 2x256 = 512 bytes
mov rbx, rsp                ← set or copy rsp to rbx
xor rcx, rcx                ← make rcx '0'
zero_loop:                  ← compare 'rcx with '256' i.e. 0x100 in hex.
cmp rcx, 0x100              ← if satisfied
je zero_done
mov word ptr [rbx + rcx*2], 0   ← i.e. empty the value
                                  set stack top 'rbx' to rbx + rcx*2
                                  'rcx*2' because total 512 bytes i.e. 2x of 256.
inc rcx                     ← increment rcx by 1.
jmp zero_loop
zero_done:
xor rdx, rdx                ← set rdx '0'
```


count_loop:
Compare 'rsi' & 'rdx'(0)
cmp rdx, rsi
jae count_done
Store byte data in 'al' from 'rdi+rdx'
mov al, byte ptr [rdi+rdx]
movzx eax, al
store the value at top to rcx
mov cx, word ptr [rbx+rax*2]
increment cx
inc cx
store the top value stored at rcx' to the index emptied in above loop'.
mov word ptr [rbx+rax*2], cx
inc rdx
jmp count_loop
condition satisfied
count_done:
Set rcx' 0'
xor rcx, rcx
set eax '0'
xor eax, eax
set edx '0'
xor edx, edx
max_loop:
again compare 'rcx' with '256'.
cmp rcx, 0x100
je max_done
move value at address to 'esi'
movzx esi, word ptr [rbx+rcx*2]
cmp esi, eax
jbe no_update
Store 'esi' in 'eax'
mov eax, esi
Store 'rcx' in 'rdx'
mov rdx, rcx
condition satisfied
no_update:
inc rcx
jmp max_loop
Condition Satisfied
max_done:
move edx' into 'eax'
mov eax, edx
Set rbp to 'rsp'
mov rsp, rbp
pop rbp
ret.