CS210 Computer Systems, Fall-2023
Midterm-II Practice

Instructions

This is a closed book and closed notes exam. NO ELECTRONIC DEVICES. For all multiple choice questions fill in ONE and ONLY ONE circle. Fill the circle in completely.

If you use check marks or other symbols the auto-grader may not be able to process your answer and will assign you a grade of zero and there will be no regrading.

All pages must have your name and id written on it. Unidentified pages will not be graded

This exam has 13 questions, for a total of 61 points.

First Name: ______________________ Last Name: ______________________

BU ID: ____________

PART A: True/False and Multiple Choice

1. (1 point) The following code:

```
        .intel_syntax noprefix
        .section .text
        .global _start
_start:
        xor rdi, rdi
        jmp func
RET1:
        jmp func
RET2:
        mov rax, 60     # LINUX: exit system call 60
        syscall

func:
        inc rdi
        jmp RET1
```

- ◯ Will increment rdi exactly 2 times
- ◯ Will exit with rdi equal to 2
- ◯ Will execute syscall instruction to hand control over to the OS
- ◯ All of the above
- ◯ None of the above

2. (1 point) The following code:

```
        .intel_syntax noprefix
        .section .text
        .global _start
_start:
        xor rdi, rdi
        call func
RET1:
        call func
RET2:
        mov rax, 60      # LINUX: exit system call 60
        syscall

func:
        inc rdi
        ret
```

- ◯ Will exit with rdi equal to 2
- ◯ Will place two values on to the stack
- ◯ Will execute the sycall instruction once
- ◯ Will initally set rdi to zero
- ◯ All of the above
- ◯ None of the above

3. (1 point) The following code:

```
        .intel_syntax noprefix
        .section .text
        .global _start
_start:
        mov rdi, 10
        call func
        mov rax, 60        # LINUX: exit system call 60
        syscall
func:
        dec rdi
        je  done
        call func
done:
        ret
```

- ◯ Will exit with rdi equal to 0
- ◯ Will execute the call instruction 10 times
- ◯ Will execute the ret instruction 10 times
- ◯ Will put and get values from the stack
- ◯ All of the above
- ◯ None of the above

PART B

The following is the gdb dump of 64 bytes of memory in base 2 notation. Using this data please fill in the following tables.

```
0x402000:  01000010  01010101  00100000  01000011  01010011  00100000  00110010  00110001
0x402008:  00110000  00100000  01010010  01110101  01101100  01100101  01010011  00100001
0x402010:  00100000  01010100  01101111  00100000  01000010  01100101  00100000  01001111
0x402018:  01110010  00100000  01101110  01001111  01110100  00100000  00110010  01000010
0x402020:  00101110  00100000  01001111  01101110  01100011  01100101  00100000  01110101
0x402028:  01110000  01101111  01101110  00100000  01100001  00100000  01110100  01101001
0x402030:  01101101  01100101  00100000  01101001  01101110  00100000  01100001  00100000
0x402038:  01100111  01100001  01101100  01100001  01111000  01111001  00101110  00101110
```

4. (4 points) Write the values as single byte values in **hex** notation.

0x402000								
0x402008								
0x402010								
0x402018								
0x402020								
0x402028								
0x402030								
0x402038								

5. (4 points) As 2-byte little endian values in **hex** notation. Hint: you will be expected to know how to do this by hand. Eg not having access to a computer.

0x402000								
0x402010								
0x402020								
0x402030								

6. (4 points) As 4-byte little endian values in **hex** notation. Hint: you will be expected to know how to do this by hand. Eg not having access to a computer.

0x402000				
0x402010				
0x402020				
0x402030				

7. (4 points) As 8-byte little endian values in **hex** notation. Hint: you will be expected to know how to do this by hand. Eg not having access to a computer.

0x402000		
0x402010		
0x402020		
0x402030		

8. (4 points) Finally using the provided ASCII Table please fill in the table below translating each byte into an ascii character.

0x402000								
0x402008								
0x402010								
0x402018								
0x402020								
0x402028								
0x402030								
0x402038								

PART C: Assembly Fragments

Given the code and list of gdb commands below, answer the following questions. Assume the code has been assembled and linked correctly to produce a binary, after which gdb is used with the binary to run the given gdb commands.

Remember to use Little Endian byte ordering for multi-byte values when displayed as single bytes

9. An assembly fragment using the `add` and `mov` instructions. Assembly code for `addA.S`:

```
        .intel_syntax noprefix

        .section .data
result:
        .quad 0x0

        .section .text
        .global _start

_start:
        add rax, rax
        mov QWORD PTR [result], rax
        int3
```

Gdb commands used with the binary `addA` produced from `addA.S`.

```
1  file addA
2  set disassembly-flavor intel
3  x/3i _start
4  b _start
5  run
6  delete 1
7  set $rax = 15
8  si
9  si
10 p /x $rax
11 p /x $pc
12 x/1xg &result
13 x/8xb &result
14 quit
```

Addtionally this is the gdb output for the gdb command at line 3 of the above commands:

```
(gdb) x/3i _start
0x401000 <_start>:    add    rax,rax
0x401003 <_start+3>:  mov    QWORD PTR ds:0x402000,rax
0x40100b <_start+11>:        int3
```

(a) (2 points) Value displayed for rax on line 10 of gdb commands:

____

(b) (2 points) Value displayed for pc on line 11 of gdb commands:

____

(c) (2 points) Value displayed for line 12 of gdb commands (`x/1xg &result` means display one 64bit value at the address in memory of the result symbol in hex notation):

____

(d) (1 point) Values displayed on line 13 of gdb commands: (`x/8xb &result` means display 8 one-byte values starting at the address in memory of the result symbol in hex notation):

____

____

____

____

____

____

____

____

10. A conditional based assembly fragment

Assembly code for `condA.S`.

```
        .intel_syntax noprefix

        .section .data
data0:  .quad 0xdead
data1:  .quad 0xbeef
data2:  .quad 0xfeed
data3:  .quad 0xface

        .section .text
        .global _start

_start:
        xor rcx, rcx
        and rax, 0xf
        mov rbx, QWORD PTR [data0]
        and rbx, 0xf
        cmp rax, rbx
        jne A
        inc rcx
A:      mov rbx, QWORD PTR [data1]
        and rbx, 0xf
        cmp rax, rbx
        jne B
        inc rcx
B:      mov rbx, QWORD PTR [data2]
        and rbx, 0xf
        cmp rax, rbx
        jne C
        inc rcx
C:      mov rbx, QWORD PTR [data3]
        and rbx, 0xf
        test rax, rbx
        jne D
        inc rcx
D:      int3
```

Gdb commands used with the binary `condA` produced from `condA.S`.

```
1  file condA
2  b _start
3  run
4  delete 1
5  set $rax = 0x124d
6  c
7  p /x $rax
8  p /x $rbx
9  p /x $rcx
10 quit
```

Note: the "c" gdb command on line 6 continues execution until the binary stops at the "int3" instruction on line 35 of the source code.

(a) (3 points) Value displayed for rax on line 7 of gdb commands:

____

(b) (2 points) Value displayed for rbx on line 8 of gdb commands:

____

(c) (2 points) Value displayed for rcx on line 9 of gdb commands:

____

First Name: ______________ Last Name: ______________ BU ID: ____________

11. A loop based assembly fragment.

Assembly code for `loopA.S`:

```
        .intel_syntax noprefix

        .section .data
data:
        .quad 0x1, 0x3, 0x5, 0x7, 0x9, 0xb, 0xd, 0xf

        .section .text
        .global _start

_start:
        xor rax, rax
        mov rbx, OFFSET [data]
        xor rcx, rcx
A:
        add rax, QWORD PTR [rbx + 8*rcx]
        inc rcx
        cmp rcx, 4
        jl  A
        int3
```

Gdb commands used with the binary `loopA` produced from `loopA.S`.

```
1 file loopA
2 b _start
3 run
4 delete 1
5 c
6 p /x $rax
7 p /x $rbx
8 p /x $rcx
9 quit
```

Assume the address of symbol `data` is 0x402000.

(a) (3 points) Value displayed for rax on line 6 of gdb commands:

____

(b) (2 points) Value displayed for rbx on line 7 of gdb commands:

____

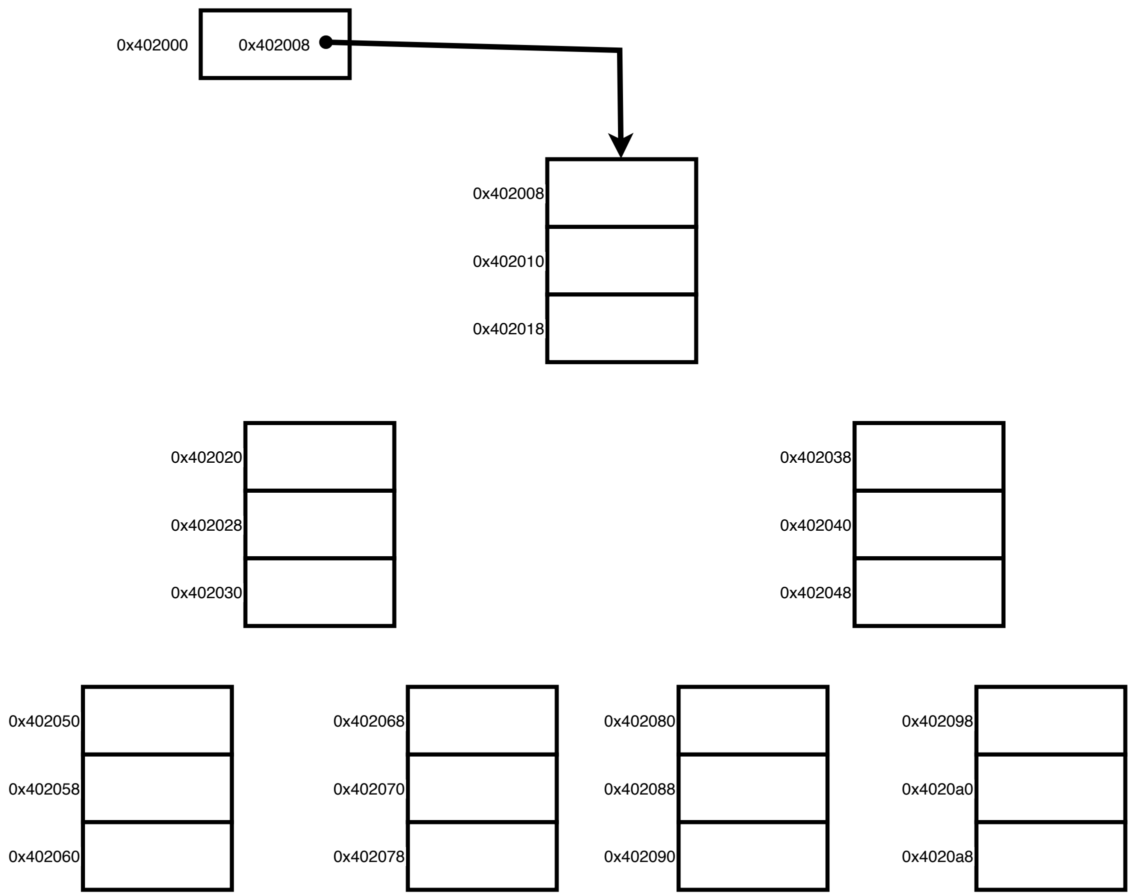
(c) (2 points) Value displayed for rcx on line 8 of gdb commands:

____

12. Given the following memory contents gathered with gdb please answer the following questions:

```
(gdb) x/12i _start
   0x401000 <_start>:     xor    rdi,rdi
   0x401003 <_start+3>:   mov    rax,QWORD PTR ds:0x402000
   0x40100b <A>:          add    rdi,QWORD PTR [rax]
   0x40100e <A+3>:        mov    rbx,QWORD PTR [rax+0x8]
   0x401012 <A+7>:        mov    rdx,QWORD PTR [rax+0x10]
   0x401016 <A+11>:       cmp    QWORD PTR [rax],0x6
   0x40101a <A+15>:       cmovl  rax,rbx
   0x40101e <A+19>:       cmovge rax,rdx
   0x401022 <A+23>:       cmp    rax,0x0
   0x401026 <A+27>:       jne    0x40100b <A>
   0x401028 <A+29>:       mov    rax,0x3c
   0x40102f <A+36>:       syscall
(gdb) x/1gx 0x402000
        0x402000:      0x0000000000402008
(gdb) x/21gx 0x402008
0x402008:       0x0000000000000003      0x0000000000402020
0x402018:       0x0000000000402038      0x0000000000000005
0x402028:       0x0000000000402050      0x0000000000402068
0x402038:       0x0000000000000007      0x0000000000402080
0x402048:       0x0000000000402098      0x0000000000000001
0x402058:       0x0000000000402008      0x0000000000000000
0x402068:       0x0000000000000002      0x0000000000402020
0x402078:       0x0000000000000000      0x0000000000000009
0x402088:       0x0000000000000000      0x0000000000000000
0x402098:       0x000000000000000b      0x0000000000000000
0x4020a8:       0x0000000000000000
```

(a) (4 points) Please update the following diagram. Be sure to fill all boxes **and include arrows to indicate address relationships**.



(b) (2 points) What will be the value of rdi when the program exits (the syscall on line 13 is executed)? If the program does not exit then your answer should be “NONE”.

rdi=_____________________________________

13. Assembly code for `pgmv1.s`:

```
        .intel_syntax noprefix
        .section .data
POS_SUM: .quad 0x0
NEG_SUM: .quad 0x0
data:   .quad -1, -1, 0, 2, 3, 1, 1, -100, 1, -4

        .section .text
        .global _start
_start:
        xor rax, rax
        xor rdi, rdi
A:
        mov rcx, QWORD PTR [data + rdi * 8]
        cmp rcx, 0
        jl B
        add QWORD PTR [POS_SUM], rcx
        jmp C
B:
        add QWORD PTR [NEG_SUM], rcx
C:
        add rax, rcx
        inc rdi
        cmp rax, 0
        je D
        jmp A
D:
        int3
```

Gdb commands used with the binary `pgmv1` produced from `pgmv1.s`.

```
file pgmv1
b _start
run
delete 1
c
p /x $rax
p /x $rcx
p /x $rdi
x /1gd &POS_SUM
x /1gd &NEG_SUM
quit
```

Assume the address of symbol `data` is 0x402000.

First Name: ____________ Last Name: ____________ BU ID: ____________

Please fill in the following:

(a) (3 points) Value displayed for rax on line 6 of gdb commands: ____________

(b) (2 points) Value displayed for rcx on line 7 of gdb commands: ____________

(c) (2 points) Value displayed for rdi on line 8 of gdb commands: ____________

(d) (2 points) Value displayed for `POS_SUM` on line 9 of gdb commands: ____________
(`x /1gd &POS_SUM` will print the 8 byte value at `POS_SUM` as a signed decimal integer). Your answer shoud be written as a normal decimal signed number.

(e) (2 points) Value displayed for `NEG_SUM` on line 10 of gdb commands: ____________
(`x /1gd &NEG_SUM` will print the 8 byte value at `NEG_SUM` as a decmial signed integer). Your answer shoud be written as a normal decimal signed number.

14. You are debugging a recursive program that is being compiled into x86 assembly. This program uses a stack to implement function calls. Given the following C function:

```c
long long func(long long x, long long y){
  long long a = x - y;

  if (a < 0) return x;
  else return func(a, y);
}
```

The translation into x86 assembly code:

```asm
func:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 24
        mov     QWORD PTR [rbp-16], rdi
        mov     QWORD PTR [rbp-24], rsi
        mov     rax, rdi
        sub     rax, rsi
        mov     QWORD PTR [rbp-8], rax
        cmp     QWORD PTR [rbp-8], 0
        jge     A
        mov     rax, QWORD PTR [rbp-16]
        jmp     B
A:
        mov     rdi, QWORD PTR [rbp-8]
        call    func
B:
        add     rsp, 24
        mov     rsp, rbp
        pop     rbp
        ret
```

And the addresses and contents of the stack region of memory in hex:

Address	Data	
rsp → 0x7fffffffdd38	0x0000000000000004	
0x7fffffffdd40	0x0000000000000013	
0x7fffffffdd48	0x000000000000000f	
rbp → 0x7fffffffdd50	0x00007fffffffdd78	
0x7fffffffdd58	0x000000000040105b	
0x7fffffffdd60	0x0000000000000004	
0x7fffffffdd68	0x0000000000000017	
0x7fffffffdd70	???	
0x7fffffffdd78	0x00007fffffffdda0	
0x7fffffffdd80	0x000000000040105b	
0x7fffffffdd88	0x0000000000000004	
0x7fffffffdd90	0x000000000000001b	
0x7fffffffdd98	0x0000000000000017	
0x7fffffffdda0	0x0000000000000000	
0x7fffffffdda8	0x0000000000401013	
0x7fffffffddb0	0x0000000000000001	
0x7fffffffddb8	0x00007fffffffe0f6	
0x7fffffffddc0	0x0000000000000000	
0x7fffffffddc8	0x00007fffffffe11c	
0x7fffffffddd0	0x00007fffffffe12c	

Answer the following questions given that func is called from an external function and **execution is stopped just prior to the execution of the instruction labeled 'B:'**. When execution is stopped, **rsp contains the value 0x7fffffffdd38 and rbp contains the value 0x7fffffffdd50.** Write your answers in hex, and you may ignore any leading 0s.

(a) (2 points) What were the arguments to the current active call to func?

0x_____________ and 0x_____________

(b) (2 points) What were the arguments to the initial first call to func?

0x_____________ and 0x_____________

(c) (2 points) What is the missing value in location 0x7fffffffdd70?

0x_____________

(d) (2 points) What is the return address to the external function that initially called func?

0x_____________

(e) (2 points) What is the address of the call instruction that called func originally, given that a call instruction takes of 5 bytes of memory?

0x_____________

(f) (2 points) What is the final return value if func were to complete execution?

0x_____________

INTEL Registers: Names, sizes and bit positions

63	31	16	7 0
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cx	cl
rdx	edx	dx	dl
rsi	esi	si	sil
rdi	edi	di	dil
rbp	ebp	bp	bpl
rsp	esp	sp	spl
r8	r8d	r8w	r8b/r8l
r9	r9d	r9w	r9b/r9l
r10	r10d	r10w	r10b/r10l
r11	r11d	r11w	r11b/r11l
r12	r12d	r12w	r12b/r12l
r13	r13d	r13w	r13b/r13l
r14	r14d	r14w	r14b/r14l
r15	r15d	r15w	r15b/r15l

INTEL EFLAGS Single bit flags that we are concerned with

- ZF Zero Flag set if result was zero
- SF Sign Flag set if result was negative (most significant bit set)
- CF Carry Flag set if operation generates a carry or borrow
- OF Overflow Flag set if overflow on signed operation

INTEL Address modes: The ways you can specify the source or destinations for an instruction : Allowed combinations are:

Immediate: the source value is stored in the instruction
eg. ADD EAX, 14 # Add 14 into the 32 bit EAX register
MOV RAX, 0xdeadbeef # set RAX

Register to register:
eg. ADD R8B, AL # add 8 bit AL value to R8B register
MOV RAX, R8 # copy the value from R8 into RAX

Memory operands:
[BaseReg + scale * IndexReg + Displacement]
Where BaseReg and IndexReg can be any general purpose register
scale is a numeric value of 1,2,4,8
Displacement is 8, 16 or 32 bit value. Often this will be a symbolic label
MOV RAX, QWORD PTR [RBX + 8*RDI + XARRAY]
Notes: In general you can omit various terms to meet your needs
When doing moves only one operand can be a memory operand.

Labels: Mark code or data with a name, linker updates references with address
1) Reference in addressing mode as displacement: Eg. MOV RAX, BYTE PTR [X]
2) Reference as a target for a call or jump: Eg. CALL myfunc or JMP loop
3) Reference the actual address as a value: Eg. MOV RAX, OFFSET X

Typical Intel GNU Assembly Instruction formats:
mnemonic : mnemonic with no operands
eg. int3
mnemonic <dst> : mnemonic with one destination operand:
eg. inc RAX. # RAX=RAX+1
mnemonic <dst>, <src> : mnemonic with one destination and one src
eg. add RAX, RBX # RAX = RAX + RBX
mnemonic <dst>, <srcA>, <srcB> : mnemonic with one destination and two srcs
eg. imul rax, rbx, 42 # rax = rbx * 42

Common Intel mnemonics (instructions):

Data Transfer:
MOV : move to/from locations : mov dst, src : dst = src;
MOVZX/MOVSX: move smaller src to larger dst zero/sign extending respectively
CMOVcc: conditional mov : cmov<cc> dst, src: if cc then dst=src else do nothing : cmovg dst, src : dst = src if greater. See flow control instructions for example conditions

ALU:
ADD <dst>, <src>: add integers (signed or unsigned) : dst = dst + src;
SUB <dst>, <src> : subtract integers (signed or unsigned) : dst = dst - src;
IMUL <dst>,<srcA>,<srcB>: multiplies integers (signed): dst = srcA * srcB;
INC. : Increment : inc dst : dst = dst + 1
DEC. : Decrement : dec dst : dst = dst - 1
AND <dst>, <src> : Bitwise boolean and : dst = dst & src
OR <dst>, <src> : Bitwise boolean or : dst = dst | src
XOR <dst>, <src> : Bitwise boolean xor : dst = dst ^ src
NOT <dst>. : Bitwise boolean not : dst = ~dst
SHR <dst>, imm. : logical shift right : dst = dst >> imm (zero extends)
SAR <dst>, imm : arithmetic shift right : dst = dst >> imm (sign extends)
SHL <dst>, imm : logical shift left: dst = dst << imm (zero fill)
SAL <dst>, imm : arithmetic shift left: dst = dst << imm (zero fill same as SHL)
CMP <dst>, <src> : set eflags based on subtraction: dst - src
TEST <dst>, <src> : set eflags based on bitwise and of dst & src

Control flow:
JMP <dst> : Unconditional jump : jmp <dst> : pc = dst : dst is usually a label
eg. jmp loop_begin
but can also be indirect:
1) a register (which contains the address to jump to)
jmp rax
2) ptr to a memory location which contains the address to jump to
jmp qword ptr [myjmptable]
JE <dst> : jump if equal : jmp if zero eflag is set (1) same as JZ
JNE <dst> : jump if not equal: jmp if zero eflag is NOT set (0)
JZ <dst> : jump if zero : same as JE
JNZ <dst> : jump not zero: same as JNE
JG <dst> : jump if greater (signed)
JGE <dst> : jump if greater or equal (signed)
JL <dst> : jump if less (signed)
JLE <dst> : jump if less or equal (signed)
JA <dst> : jump if above (unsigned)
JAE <dst> : jump if above or equal (unsigned)
JB <dst> : jump if below (unsigned)
JBE <dst> : jump if below or equal (unsigned)

Stack:
PUSH : stack push : push src : rsp=rsp - len(src); M[rsp] = src;
POP : stack pop : pop dst : dst = M[rsp]; rsp = rsp + len(src);
CALL/RET : call and return from subroutine : call pushes address of the following instruction on the stack and then sets pc to the specified target address. ret pops the top value from the stack and sets the pc to this address

Misc:
NOP : no operation
INT3 : hand control back to debugger
SYSCALL : request operating system call routine

Byte Vector Sizes and Names
1 Byte : INTEL BYTE : GAS directive .byte. : C unsigned char and char (signed)
2 Bytes : INTEL WORD : GAS directive .short : C unsigned short and short (signed)
4 Bytes : INTEL DWORD : GAS directive .long : C unsigned int and int (signed)
8 Bytes : INTEL QWORD : GAS directive .quad : C unsigned long long and long long (signed)
NOTE: On INTEL 64 bit machines all pointer types (char *, short *, int *, long long * and void *) are 8 bytes in size

Two's Complement facts:
Value of bit vector $X_w = [b_{w-1}\ldots b_0]$ is $-2^{w-1}b_{w-1} + \sum_{i=0}^{w-2} 2^i b^i$
Negation of a value: $-x = \ x + 1$
$-1 = [1\ldots 1]$
min $= -2^{w-1} = [10\ldots 0]$, max$=2^{w-1} - 1 = [01\ldots 1]$

```
GDB Commands:

file <binary>     : opens a new binary replacing the current one eg. file empty
run               : creates a process from the current open binary and initiates the cpu's
                    execution within it
b <symbol>        : sets a breakpoint to stop execution when the PC equals the address of
                    symbol eg. b _start
c                 : continue execution from current PC address until execution terminates
                    or a break point is hit
si                : single step a cpu instruction eg. unfreeze the cpu so it can do one
                    execution loop
p /x $<REG>       : print the current value of the specified register in hex
x/<n>bx <address> : print/examine n memory bytes sized values start at the specified address
                    in hex notation
x/<n>hx <address> : same as above but n memory 2-byte sized values
x/<n>wx <address> : same as above but n memory 4-byte sized values
x/<n>gx <address> : same as above but n memory 8-byte sized values
set $<REG>=<value>: sets the value of the specified registers.  Value can be specified in
                    notations by using the right prefix eg. 0x for hex, 0b for binary.  The
                    default is signed two-complement integers.
set {CType}(address)=<value>: set in memory at the address specified.  CType is one of the
                              C programming type names for bytes sized quantities.
                              See notes below for a list

NOTES:
      1. When using x to display multi-bytes sized (eg. x/1hx <addr>) gdb will reorder
      to account for endianness of the computer.  For example if the bytes at address _start,
      on a little endian computer, are 0xFA 0x10 and we use the command x/1hx & _start gdb
      will display something like
      (gdb) x/1xh &_start
      0x401000 <_start>:      0x10FA
      This is true for all the other multi-bytes sizes (h,w,g)

      2. For the p and x command the following Format letters can be used
      o(octal), x(hex), d(decimal), u(unsigned decimal), t(binary), f(float), a(address),
      i(instruction), c(char), s(string) and z(hex, zero padded on the left).

      3. CType names: "unsigned char" : 1 byte, "unsigned short" : 2 byte,
                      "unsigned int" : 4 byte, "unsigned long long": 8 byte
```

INTEL C Linux Calling Conventions :

Defines how registers should be used by caller and callee code. It also defines how arguments and the return value for a C function should be assigned to registers and the stack. The First 6 integer arguments are passed in registers as follows

Argument 0 : rdi
Argument 1 : rsi
Argument 2 : rdx
Argument 3 : rcx
Argument 4 : r8
Argument 5 : r9
Return value : rax

If more than 6 arguments are required the remainder are pushed on the stack in reverse order (last pushed first). A functions return value must be place in rax.

The function code (callee) is free to overwrite any of the 7 above registers along with r10 and r11. Calling code (caller) needs to save and restore these registers if it wants to rely on their values. Thus they are called volatile and caller saved. The values of the remaining general purpose registers (rbx, rsp, rbp, r12-r15) must not be affected by a function as such they are called non-volatile and callee saved. Eg. if a function writes them it must restore their value before returning to the caller

```
ASCII Hex Table (Hex Character)

    00 nul   01 soh   02 stx   03 etx   04 eot   05 enq   06 ack   07 bel
    08 bs    09 ht    0a nl    0b vt    0c np    0d cr    0e so    0f si
    10 dle   11 dc1   12 dc2   13 dc3   14 dc4   15 nak   16 syn   17 etb
    18 can   19 em    1a sub   1b esc   1c fs    1d gs    1e rs    1f us
    20 sp    21  !    22  "    23  #    24  $    25  %    26  &    27  '
    28  (    29  )    2a  *    2b  +    2c  ,    2d  -    2e  .    2f  /
    30  0    31  1    32  2    33  3    34  4    35  5    36  6    37  7
    38  8    39  9    3a  :    3b  ;    3c  <    3d  =    3e  >    3f  ?
    40  @    41  A    42  B    43  C    44  D    45  E    46  F    47  G
    48  H    49  I    4a  J    4b  K    4c  L    4d  M    4e  N    4f  O
    50  P    51  Q    52  R    53  S    54  T    55  U    56  V    57  W
    58  X    59  Y    5a  Z    5b  [    5c  \    5d  ]    5e  ^    5f  _
    60  `    61  a    62  b    63  c    64  d    65  e    66  f    67  g
    68  h    69  i    6a  j    6b  k    6c  l    6d  m    6e  n    6f  o
    70  p    71  q    72  r    73  s    74  t    75  u    76  v    77  w
    78  x    79  y    7a  z    7b  {    7c  |    7d  }    7e  ~    7f del
```

Linux X86 64 Bit Alignment Rules: (type - alignment in bytes)
char - none, short - 2 , int - 4, long long - 8, Same for unsigned integers. Pointers - 8. long double - 16. Arrays aligned to alignment of element type. Structures aligned to max alignment of its fields, padding added in between fields as need and at end to ensure field and overall alignment.