SUM

`sum:`

This line labels the following code. Allows for a jump command to jump to the beginning of this procedure.

```
    pushl   %ebp
```

This line pushes the 32 bits in register %ebp onto the stack. The contents of register %ebp is the pointer to the previous frame in the stack (the frame in which this function was called).

```
    movl    %esp, %ebp
```

This line then replaces the frame pointer with the current top of the stack (the new frame pointer).

```
    movl    12(%ebp), %eax
```

This line moves the data in address 12(%ebp) to %eax. To be more precise, address 12(%eax) is the location of the parameter int y and %eax is just a general register.

```
    addl    8(%ebp), %eax
```

This line then adds the data in address 8(%ebp) to %eax and stores the result in %eax. Address 8(%ebp) points to parameter int x and %eax contained the value of int y. After this instruction, %eax contains the value x + y.

```
    popl    %ebp
```

This line pops the top of the stack off and replaces the frame pointer with this value. This frame pointer points to the prior frame that called this procedure.

```
    ret
```

This line pops the top of the stack again and replaces the instruction pointer with it. This pointer points to the instruction that called this function initially. At the end of all this, we are back at the instruction that called the sum procedure initially, except now the result of the sum procedure is stored in the %eax register which can be used by the program. NOTE: All commands are in long form meaning that they act on 32 bits

SUB

`sub:`

This line labels the following code. Allows for a jump command to jump to the beginning of this procedure.

```
    pushl   %ebp
```

This line pushes the 32 bits in register %ebp onto the stack. The contents of register %ebp is the pointer to the previous frame in the stack (the frame in which this function was called).

```
    movl    %esp, %ebp
```

This line then replaces the frame pointer with the current top of the stack (the new frame pointer).

```
    movl    8(%ebp), %eax
```

This line moves the data in address 8(%ebp) to %eax. To be more precise, address 8(%eax) is the location of the parameter int x and %eax is just a general register.

```
    subl    12(%ebp), %eax
```

This line then subtracts the data in address 12(%ebp) from %eax and stores the result in %eax. Address 12(%ebp) points to parameter int y and %eax contained the value of int x. After this instruction, %eax contains the value x - y.

```
    popl    %ebp
```

This line pops the top of the stack off and replaces the frame pointer with this value. This frame pointer points to the prior frame that called this procedure.

```
    ret
```

This line pops the top of the stack again and replaces the instruction pointer with it. This pointer points to the instruction that called this function initially. At the end of all this, we are back at the instruction that called the sub procedure initially, except now the result of the sub procedure is stored in the %eax register which can be used by the program. NOTE: All commands are in long form meaning that they act on 32 bits

MUL

```
mul:
```

This line labels the following code. Allows for a jump command to jump to the beginning of this procedure.

```
    pushl   %ebp
```

This line pushes the 32 bits in register %ebp onto the stack. The contents of register %ebp is the pointer to the previous frame in the stack (the frame in which this function was called).

```
    movl    %esp, %ebp
```

This line then replaces the frame pointer with the current top of the stack (the new frame pointer).

```
    movl    12(%ebp), %eax
```

This line moves the data in address 12(%ebp) to %eax. To be more precise, address 12(%eax) is the location of the parameter int y and %eax is just a general register.

```
    imull   8(%ebp), %eax
```

This line then multiplies the data in address 8(%ebp) with %eax and stores the result in %eax. Address 8(%ebp) points to parameter int x and %eax contained the value of int y. After this instruction, %eax contains the value x - y. Because of the i prefixing mull, this multiplication is also signed (acting on signed integers).

```
    popl    %ebp
```

This line pops the top of the stack off and replaces the frame pointer with this value. This frame pointer points to the prior frame that called this procedure.

```
    ret
```

This line pops the top of the stack off and replaces the frame pointer with this value. This frame pointer points to the prior frame that called this procedure. At the end of all this, we are back at the instruction that called the mul procedure initially, except now the result of the mul procedure is stored in the %eax register which can be used by the program. NOTE: All commands are in long form meaning that they act on 32 bits

DIV

```
div:
```

This line labels the following code. Allows for a jump command to jump to the beginning of this procedure.

```
    pushl   %ebp
```

This line pushes the 32 bits in register %ebp onto the stack. The contents of register %ebp is the pointer to the previous frame in the stack (the frame in which this function was called).

```
    movl    %esp, %ebp
```

This line then replaces the frame pointer with the current top of the stack (the new frame pointer).

```
    movl    8(%ebp), %edx
```

This line moves the data in address 8(%ebp) to %edx. To be more precise, address 8(%eax) is the location of the parameter int x and %edx is just a general register.

```
    movl    %edx, %eax
```

This line moves the data in %edx to %eax. Now both %edx and %eax contain int x.

```
    sarl    $31, %edx
```

This line performs an arithmetic shift to the right 31 bits. This fills the entire register with either 0s (if positive) and 1s (if negative). The idivl command takes the divident to be the concatenation of the %edx register with the %eax register, in that order (written EDX:EAX). This just prepares the %edx register to be empty because our dividend is no larger than 32 bits (because it is an int)

```
    idivl   12(%ebp)
```

This line divides EDX:EAX by the data in address 12(%ebp) which containts int y. The quotient is stored in %eax and the remainder in %edx

```
    popl    %ebp
```

This line pops the top of the stack off and replaces the frame pointer with this value. This frame pointer points to the prior frame that called this procedure.

```
    ret
```

This line pops the top of the stack off and replaces the frame pointer with this value. This frame pointer points to the prior frame that called this procedure. At the end of all this, we are back at the instruction that called the div procedure initially, except now the result of the div procedure (quotient of the division) is stored in the %eax register which can be used by the program. NOTE: All commands are in long form meaning that they act on 32 bits