

```

section .data
    fmt_add: db "Addition result: %d", 10, 0
    fmt_sub: db "Subtraction result: %d", 10, 0
    fmt_mul: db "Multiplication result: %d", 10, 0
    fmt_div: db "Division result: %d", 10, 0
    fmt_neg: db "Logical Negation result: %d", 10, 0

section .text
    extern printf
    global main

main:
    push rbp
    mov rbp, rsp

    mov rcx, 8          ; arg1 = 8
    mov rdx, 2          ; arg2 = 2

    call additionCode
    mov rcx, rax

    call subtractionCode
    mov rcx, rax

    call multiplicationCode
    mov rcx, rax

    call divisionCode
    mov rcx, rax

    call logicalNegationCode

    mov rax, 60          ; Exit syscall number
    xor rdi, rdi          ; Exit code 0
    syscall

additionCode:
    ; Set up stack frame
    push rbp
    mov rbp, rsp

    ; Reserve 16 bytes of stack space manually
    push 0              ; Reserve 8 bytes for local variable 1 (arg1)
    push 0              ; Reserve 8 bytes for local variable 2 (arg2)

    ; Store arg1 (rcx) and arg2 (rdx) in local variable space
    mov [rbp-8], rcx      ; Store arg1 at rbp-8
    mov [rbp-16], rdx      ; Store arg2 at rbp-16

    ; Perform arithmetic function (addition)
    mov rcx, [rbp-8]      ; Load arg1 into rcx
    mov rdx, [rbp-16]      ; Load arg2 into rdx

add_loop:
    cmp rcx, 0            ; Check if arg1 (rcx) is zero
    je add_done           ; If zero, addition is complete
    inc rdx               ; Increment rdx (partial result)
    dec rcx               ; Decrement rcx
    jmp add_loop          ; Repeat

```

```

add_done:
    mov [rbp-8], rdx      ; Store answer at rbp-8

    ; Print the first answer to the console using printf
    mov rsi, rdx          ; Move result into rsi (printf argument)
    lea rdi, [fmt_add]    ; Load format string
    xor rax, rax          ; Clear rax for variadic function
    call printf            ; Call printf to print the result

    ; store answer in rax / clean up stack frame
    pop rdx
    pop rax

    ; Restore stack frame
    mov rsp, rbp
    pop rbp
    ret                  ; Return to caller

subtractionCode:
    ; Set up stack frame
    push rbp
    mov rbp, rsp

    ; Reserve 16 bytes of stack space manually
    push 0                ; Reserve 8 bytes for local variable 1 (arg1)
    push 0                ; Reserve 8 bytes for local variable 2 (arg2)

    ; Store arg1 (rcx) and arg2 (rdx) in local variable space
    mov [rbp-8], rcx       ; Store arg1 at rbp-8
    mov [rbp-16], rdx      ; Store arg2 at rbp-16

    ; Perform arithmetic function (subtraction)
    mov rcx, [rbp-8]        ; Load arg1 into rcx
    mov rdx, [rbp-16]        ; Load arg2 into rdx

sub_loop:
    cmp rdx, 0              ; Check if arg2 (rdx) is zero
    je sub_done             ; If zero, subtraction is complete
    dec rcx                ; Decrement rcx (arg1)
    dec rdx                ; Decrement rdx (arg2)
    jmp sub_loop            ; Repeat

sub_done:
    mov [rbp-8], rcx       ; Store arg1 at rbp-8

    ; Print the answer to the console using printf
    mov rsi, rcx            ; Move result into rsi (printf argument)
    lea rdi, [fmt_sub]      ; Load format string
    xor rax, rax            ; Clear rax for variadic function
    call printf              ; Call printf to print the result

    ; store answer in rax / clean up stack frame
    pop rdx
    pop rax

    ; Restore stack frame
    mov rsp, rbp
    pop rbp

```

```

ret           ; Return to caller

multiplicationCode:
; Set up stack frame
push rbp
mov rbp, rsp

; Reserve 16 bytes of stack space manually
push 0          ; Reserve 8 bytes for local variable 1 (arg1)
push 0          ; Reserve 8 bytes for local variable 2 (arg2)

; Store arg1 (rcx) and arg2 (rdx) in local variable space
mov [rbp-8], rcx    ; Store arg1 at rbp-8
mov [rbp-16], rdx   ; Store arg2 at rbp-16

; Perform arithmetic function (multiplication)
mov rcx, [rbp-8]    ; Load arg1 into rcx
mov rdx, [rbp-16]    ; Load arg2 into rdx
xor rbx, rbx        ; Clear rax to accumulate result

mul_loop:
cmp rdx, 0          ; Check if arg2 (rdx) is zero
je mul_done         ; If zero, multiplication is complete

    mov rcx, [rbp-8]    ; Load arg1 into rcx
mul_addition:
    cmp rcx, 0          ; Check if rbx (remaining value) is zero
    je mul_addition_done ; If zero, end addition
    inc rbx             ; Increment result in rax
    dec rcx             ; Decrement rbx
    jmp mul_addition   ; Repeat until rbx is zero

mul_addition_done:
    dec rdx              ; Decrement arg2
    jmp mul_loop         ; Repeat multiplication loop

mul_done:
    mov [rbp-8], rbx     ; Store arg1 at rbp-8

    ; Store the final result in rax
    mov rsi, rbx          ; Move result into rsi (printf argument)
    lea rdi, [fmt_mul]    ; Load format string
    xor rax, rax          ; Clear rax for variadic function
    call printf            ; Print the result

    ; store answer in rax / clean up stack frame
    pop rdx
    pop rax

    ; Restore stack frame
    mov rsp, rbp
    pop rbp
    ret                  ; Return to caller

divisionCode:
; Set up stack frame
push rbp
mov rbp, rsp

```

```

; Reserve 16 bytes of stack space manually
push 0          ; Reserve 8 bytes for local variable 1 (arg1)
push 0          ; Reserve 8 bytes for local variable 2 (arg2)

; Store arg1 (dividend) and arg2 (divisor) in local variable space
mov [rbp-8], rcx    ; Store arg1 at rbp-8
mov [rbp-16], rdx   ; Store arg2 at rbp-16

; Initialize rbx (quotient) to 0
xor rbx, rbx      ; Clear rbx to store the quotient

; Load dividend and divisor
mov rcx, [rbp-8]    ; Load dividend into rcx
mov rdx, [rbp-16]   ; Load divisor into rdx

cmp rdx, 0          ; Check if divisor is zero
je div_by_zero      ; Handle division by zero

div_outer_loop:
    cmp rcx, 0          ; Check if dividend is zero
    jle div_done         ; If zero or negative, division is complete

    mov rsi, rdx         ; Load divisor into rsi for inner loop

div_inner_loop:
    cmp rsi, 0          ; Check if divisor (rsi) is zero
    je inner_loop_done  ; Exit inner loop if zero

    dec rcx              ; Decrement dividend
    dec rsi              ; Decrement divisor
    jmp div_inner_loop   ; Repeat until divisor reaches zero

inner_loop_done:
    inc rbx              ; Increment quotient
    jmp div_outer_loop   ; Continue outer loop

div_done:
    mov [rbp-8], rbx      ; Store the quotient in rbp-8 for consistency

    ; Store the final result in rbx (quotient)
    mov rsi, rbx          ; Move result into rsi (printf argument)
    lea rdi, [fmt_div]    ; Load format string
    xor rax, rax          ; Clear rax for variadic function
    call printf            ; Print the result

    mov rax, [rbp-8]       ; Reload the original value into rax
    mov rdx, [rbp-16]

    jmp div_exit          ; Skip division by zero error handling

div_by_zero:
    mov rax, -1            ; Set error value for division by zero
    mov rsi, rax            ; Move error value into rsi (printf argument)
    lea rdi, [fmt_div]    ; Load format string
    xor rax, rax            ; Clear rax for variadic function
    call printf            ; Print error message

div_exit:
    ; store answer in rax / clean up stack frame

```

```

pop rax
pop rcx

; Restore stack frame
mov rsp, rbp
pop rbp
ret           ; Return to caller

logicalNegationCode:
; Set up stack frame
push rbp
mov rbp, rsp

; Manually reserve space on the stack
push 0          ; Reserve 16 bytes of stack space for local variables
push 0

; Store arg1 (dividend) and arg2 (divisor) in local variable space
mov [rbp-8], rcx    ; Store arg1 at rbp-8
mov [rbp-16], rdx   ; Store arg2 at rbp-16

; Initialize rbx with 0 (representing false)
xor rbx, rbx

; Load the input value from the reserved space
mov rcx, [rbp-8]

cmp rcx, 0          ; Compare input with 0
je logical1         ; If input is 0, go to logical1 (set result to 1)

logical0:
; If input is non-zero, rax stays 0 (false)
mov [rbp-8], rbx    ; store 0

mov rsi, rbx        ; Move result (rax) into rsi (for printf)
lea rdi, [fmt_neg]  ; Load the format string
xor rax, rax        ; Clear rax for variadic function
call printf          ; Print the result (0 or 1)
jmp neg_exit         ; Exit the function

logical1:
; If input is 0, set rax to 1 (true)
mov rbx, 1           ; Set rax to 1 (true)
mov [rbp-8], rbx    ; store 1

mov rsi, rbx        ; Move result (rax) into rsi (for printf)
lea rdi, [fmt_neg]  ; Load the format string
xor rax, rax        ; Clear rax for variadic function
call printf          ; Print the result (0 or 1)

neg_exit:
; store answer in rax / clean up stack frame
pop rdx
pop rax

; Restore stack frame
mov rsp, rbp
pop rbp

```

```
ret          ; Return to caller
```