Jared Dyreson
CPSC-240 09
TR at 11:30 to 13:20
February 19, 2019

Short Answer

1. Assembler code representation for the C equivalent
    a. sum = number + other
        i. mov r10, 10
        ii. mov r11, 11
        iii. add r10, r11
        iv. ; add <dst> <src>
    b. subtrahend = number - other
        i. mov r10, 11
        ii. mov r11, 10
        iii. sub r11, r10
    c. quotient = dividend / divisor
        i. mov al, 10 ; dividend
        ii. mov ah, 0 ; remainder
        iii. mov bl, 2 ; divisor
        iv. div bl ; dividend / divisor
        v. mov byte[rdi], bl
        vi. mov rax, SYS_WRITE
        vii. mov rdx, 1
        viii. syscall
    d. modulo = dividend % divisor
        i. mov al, 10 ; dividend
        ii. mov ah, 0; remainder
        iii. mov bl, 2 ; divisor
        iv. div bl ; dividend / divisor
        v. mov byte[rdi], ah
        vi. mov rax, SYS_WRITE
        vii. mov rdx, 1

      viii. syscall

2. Assembler flags and their meaning
    a. **CF: Carry Flag** → It indicates when an arithmetic carry or borrow has been generated out of the four most significant bits
        i. 101 + 101 = will result in a carry flag when you add the last elements together
    b. **AF: Auxiliary Flag** → It indicates when an arithmetic carry or borrow has been generated out of the four least significant bits
    c. **ZF: Zero Flag** → to check the result of an arithmetic operation, including bitwise logical instructions. It is set to 1, or true, if an arithmetic result is zero, and reset otherwise
    d. **OF: Overflow Flag** → a single bit in a system status register used to indicate when an arithmetic overflow has occurred in an operation, indicating that the signed two's-complement result would not fit in the number of bits used for the operation
    e. **SF: Sign Flag** → a single bit in a system status (flag) register used to indicate whether the result of the last mathematical operation resulted in a value in which the most significant bit was set
    f. These registers are stored in the **eflags** register
3. The general purpose registers in x86-64 ASM
    a. r{8..15} : allow for the use of putting anything in them and not possibly messing up program flow (generally)
    b. rax : place syscalls here
    c. Rsi : content of SYS_WRITE
    d. Rdi : STDOUT Flag
    e. Rdx : strlen(message)
4. Four instructions that use the stack pointer and their purpose
    a. Push : push a value onto the stack
    b. Pop : take a value off the stack
    c. Pushf : save the status register to the stack
    d. Popf : restore the status register from the stack
5. Difference between **sar** and **shr**
    a. **shr** : Performs a logical shift right operation on destination operand

   b. **sar** : Performs an arithmetic shift right operation on destination operand

6. This operation "xor rdi, rdi" will clear the rdi register without having to do a mov instruction
7. The connection to these registers is that they are on the same register, rax being the full 64 bit lane, eax 32, ax 16, al 8. Each are special in their own right.
8. The **rip** register points to the next instruction to be executed
9. A label is a ghetto version of a function that can be treated as a section of code to be either executed in sequence or on a conditional basis, depending on program flow. It does not appear to have a size.
10. **je** is "jump if the compare was equal to a constant other than 0". **jz** is "jump if zero"
11. **jb** "jump below" and **jl** "jump if less"
12. Nasm will spit out an unlinked executable and is useless unless linked by **ld**, which will then render a coherent ELF executable (depending on architecture)
13. Symbols are used to debug and are used by GDB to move through program flow. They are a roadmap for GDB.
14. Linker (ld) will produce an ELF executable or more verbosely put:
    a. ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked, not stripped
15. Consider the following code
    a. mov rax, SYS_READ ; rax holds our syscall for reading in file
    b. mov rdx, BUFFER_SIZE; we tell rdx the size of the buffer
    c. mov rdi, 1; indicates how big our data type is (sizeof(char))
    d. syscall
16. Those values provided in the above code snippet are then issued to the kernel, requesting permission to read the file
17. The stack frame is a data structure and more specifically the place where all programs live. Values that are loaded in first are let out last. This is especially helpful when we want the base pointer to be buried at the bottom so it does not get accidentally popped off. The reason why recursion can exist
18. The **rbp** is the base pointer of the stack, best kept at the bottom.
19. The values go as follows
    a. -255, -65535, 1, 1

20. Dissecting the r9 register
    a. r9 → 64
    b. r9d → 32
    c. r9w → 16
    d. r9b → 8
21. You can test rdx by running the **test** operand
22. Subroutine to translate numbers to hex
    a. See last page

```
; Worksheet 3 Subroutine
; Written by Jared Dyreson

%define INGEST_NUMBER 15
section .data
variable: db INGEST_NUMBER, 10
hextable: db "0123456789ABCDEF"
section .text
global _start

_start:
jmp cast_int_to_hex
cast_int_to_hex:
cmp byte[variable], 16
jge exit_exceeds_bounds_exception
mov rdi, 1
mov al, byte[variable]
lea rbx, [hextable]
xlat
mov byte[variable], al
mov rsi, variable
mov rax, 1
mov rdx, 2
syscall

jmp exit_succesfully

exit_exceeds_bounds_exception:
mov rax, 60
mov rdi, 1
syscall

exit_succesfully:
mov rax, 60
mov rdi, 0
syscall
```