Lab Manual

1. Assumed Prior Knowledge

1.1 Background Knowledge

opcodes, instructions, subroutines, stacks, registers, program counters
bits, nibbles, bytes
endianness

x86-64 — little-endian machine
14 64-bit general purpose registers
(RAX, RBX, RCX, RDX, RDI, RSI and R8-R15)
64-bit stack pointer register (RSP)
↳ contains the memory address of the top of the current program stack
base pointer register (RBP)
↳ used during subroutine execution

Von Neumann architecture

3 subsystems
- → CPU
- → random access main memory
- → I/O subsystem

CPU → capable of executing instructions, residing in the main memory, which are binary codes

1.2 Essential Concepts

Programs are stored in the computer memory as sequences of instructions and data in binary format.

binary → machine language representation of a program

machine language

0	01001000
1	11000111
2	11000000
3	00000001
4	00000000
5	00000000
6	00000000
7	01001000
8	11000111
9	11000001
10	00000001
11	00000000
12	00000000
13	00000000
14	01001000
15	00000001
16	11000001

binary

```
48 c7 c0 01 00 00 00    # move the number 1 into the RAX register
48 c7 c1 01 00 00 00    # move the number 1 into the RCX register
48 01 c1                # add the contents of RAX to RCX
```

hex

punch cards — zeros and ones

assemblers (1950s) — computer programs

↳ translates text from symbolic assembly language to machine code

instruction code — mnemonic, represented in decimal or hex

each architecture → own machine code

own assembly language ←

```
movq    $1, %rax    # Move the number 1 into the RAX register
movq    $1, %rcx    # Move the number 1 into the RCX register
addq    %rax, %rcx  # Add the contents of RAX to RCX
```

assembly

2. Designing a Program

description → pseudocode → assembly code

2.1. Description

input ≥ 0

even → +1

input	output
0	1
1	1
2	3
3	3
10	11
21	21
42	43
1041	1041

2.2. Specification

```
main() {
        // print the welcome string
        print("Welcome to our program!")

        // call the inout subroutine
        inout()
}

inout() {
        // ask for the input
        int NUMBER = read(keyboard_input)

        // check whether the number is even
        if (NUMBER % 2 == 0) { // use modulo to determine divisibility by 2
                NUMBER = NUMBER + 1 // increment by 1
        }

        // print the outcome
        print(NUMBER)
}
```

2.3. Implementation

```
# ***********************************************************************
# * Program: Oddifier                                                   *
# * Description: This program prints the closest >= odd number to the input *
# ***********************************************************************

.text
welcome:        .asciz "\nWelcome to our program!\n"
prompt:         .asciz "\nPlease enter a positive number:\n"
input:          .asciz "%ld"
output:         .asciz "The result is: %ld.\n\n"

.global main

main:
        # prologue
        pushq   %rbp                    # push the base pointer
        movq    %rsp, %rbp              # copy stack pointer value to base pointer

        movq    $0, %rax                # no vector registers in use for printf
        movq    $welcome, %rdi          # first parameter: welcome string
        call    printf                  # call printf to print welcome
        call    inout                   # call the subroutine inout

        # epilogue
        movq    %rbp, %rsp              # clear local variables from stack
        popq    %rbp                    # restore base pointer location

end:    # this loads the program exit code and exits the program

        movq    $0, %rdi
        call    exit

# ******************************************************************
# * Subroutine: inout                                              *
# * Description: this subroutine takes an integer as input from a user, *
# * increments it by 1 if it is even, and prints it out           *
# * Parameters: there are no parameters and no return value       *
# ******************************************************************
inout:
        # prologue
        pushq   %rbp                    # push the base pointer
        movq    %rsp , %rbp             # copy stack pointer value to base pointer

        movq    $0, %rax                # no vector registers in use for printf
        movq    $prompt, %rdi           # param1: prompt string
        call    printf                  # call printf to print prompt

        subq    $16, %rsp               # reserve space in stack for the input
        movq    $0, %rax                # no vector registers in use for scanf
        movq    $input, %rdi            # param1: input format string
        leaq    -16(%rbp), %rsi         # parameter2: address of the reserved space
        call    scanf                   # call scanf to scan the users input

        movq    -16(%rbp), %rsi         # load the input value into RSI
                                        # (RSI is the second parameter register)

        movq    %rsi, %rax              # copy the input to RAX
        movq    $2, %rcx                # move the value 2 to RCX
        movq    $0, %rdx                # clear the contents of RDX
        divq    %rcx                    # divide the content of RDX:RAX by the
                                        # content of RCX (result stored
                                        # in RAX and remainder in RDX)
        cmpq    $0, %rdx                # compare RDX to 0
        jne     odd                     # if they are not equal (input is odd),
                                        # don't increment
even:
        incq    %rsi                    # increment the input value

odd:
        movq    $0, %rax                # no vector registers in use for printf
        movq    $output, %rdi           # param1: output string
                                        # param2: number (in RSI)
        call    printf                  # call printf to print the output

        # epilogue

 movq   %rbp , %rsp                     # clear local variables from stack
 popq   %rbp                            # restore base pointer location

 ret                                    # return from subroutine
```

3. Assembler Directives (.bss, .text, .global, .skip, .asciz, etc.)

↳ special functions

.text: tells the assembler to put all subsequent code in a specific section

.global: makes certain labels visible

3.1. Section Directives: .text, .data, .bss

memory space of a program → 3 different sections

```
.text
.data
.bss
```

- .text → hold all instructions, read-only
- .data → initialized variables
- .bss → uninitialized variables

3.2. Defining constants: .equ

```
.equ NAME, EXPRESSION
```

defines symbolic names for expressions

```
.equ FOO, 1024

pushq $FOO  # push 1024
```

3.3. Declaring variables: .byte, .word, .long, .quad

```
.byte VALUE
.word VALUE
.long VALUE
.quad VALUE
```

reserves and initialises memory for variables

```
FOO: .byte 0xAA, 0xBB, 0xCC      # three bytes starting at address FOO
BAR: .word 2718, 2818            # a couple of words
BAZ: .long 0xDEADBEEF            # a single long
BAK: .quad 0xDEADBEEFBAADF00D    # a single quadword
```

```
FOO: .byte 0x0D, 0xF0, 0xAD, 0xBA, 0xEF, 0xBE, 0xAD, 0xDE
FOO: .word 0xF00D, 0xBAAD, 0xBEEF, 0xDEAD
FOO: .long 0xBADF00D, 0xDEADBEEF
FOO: .quad 0xDEADBEEFBAADF00D
```

3.4. Reserving memory: .skip

```
.skip AMOUNT
```

```
BUFFER: .skip 1024 # reserve 1024 bytes of memory
```

3.5. Strings variables: .ascii, .asciz

```
STRING:   .ascii string
STRINGZ:  .asciz string
```

reserves and initialises blocks of ASCII encoded characters

```
WELCOME:  .ascii "Hello!!"   # A string..
          .byte 0x00         # ..followed by a 0-byte.

WELCOME:  .asciz "Hello!!"   # A string followed by a 0-byte.
```

3.6. Global symbols: .global

```
.global label
```

enters a label into the symbol table

table of contents, contained in binary assembled file

labels → useful for access by other programs

visible in debugger

other programs using subroutines

export main label → very important

↳ shows the operating system where to start running the program

```
.global main
```

4. x86-64 Assembly Language

4.1. Instructions and Operands

Instructions ↓ what should happen

Operands ↓ the data to act with

4.1.1. Operand Prefixes: Registers and Literal Values

register names → %

literal values → $

4.1.2. Instruction Postfixes (Specifying Operand Size)

b, w, l, q modifiers

↳ byte, word (2b), long (4b), quadword (8b)

specifies the size of the operand

```
pushb $3 # Push one byte onto the stack (0x03) (NOTE: See below)
pushw $3 # Push two bytes onto the stack (0x0003)
pushl $3 # Push four bytes onto the stack (0x00000003)
pushq $3 # Push eight bytes onto the stack (0x0000000000000003)
```

4.1.3. Partial Registers

addresses smaller parts of the 64-bits registers

```
movl %eax, %ebx # Copy 32 bits values between registers
movq %rax, %rbx # Copy 64 bits values between registers
movw %ax, %bx   # Copy only the lowest order 16 bits
movb %al, %bl   # Copy only the lowest order 8 bits
movb %ah, %al   # Copy 8 bits within a single register
```

8-byte	4-byte	2-byte	1-byte
%rax	%eax	%ax	%al
%rcx	%ecx	%cx	%cl
%rdx	%edx	%dx	%dl
%rbx	%ebx	%bx	%bl
%rsi	%esi	%si	%sil
%rdi	%edi	%di	%dil
%rsp	%esp	%sp	%spl
%rbp	%ebp	%bp	%bpl
%r8	%r8d	%r8w	%r8b
%r9	%r9d	%r9w	%r9b
%r10	%r10d	%r10w	%r10b
%r11	%r11d	%r11w	%r11b
%r12	%r12d	%r12w	%r12b
%r13	%r13d	%r13w	%r13b
%r14	%r14d	%r14w	%r14b
%r15	%r15d	%r15w	%r15b

4.1.4. Addressing Memory

- immediate
  - label → the value at the address of the label
  - $label → the address
- indirect
  - (%RAX) → value at the memory address
  - −8(%RBP) → value at the memory address stored in register + displacement

table(%RDI, %RCX, 8) ≡

displacement(base, index, scale)

displacement ↳ constant expression; base, index → registers; scale → 1, 2, 4 or 8

= displacement + base + index × scale

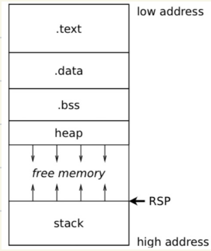
4.2. Instruction Set

Mnemonic	Operands	Action	Description
			Data Transfer
`mov.`	SRC, DST	DST = SRC	Copy.
`pushq`	SRC	`%RSP` -= 8, (`%RSP`) = SRC	Push a value onto the stack.
`popq`	DST	DST = (`%RSP`) , `%RSP` += 8	Pop a value from the stack.
`xchg.`	A, B	TMP = A, A = B, B = TMP	Exchange two values.
`movzb.`	SRC, DST	DST = SRC (one byte only)	Move byte, zero extended.
`movzw.`	SRC, DST	DST = SRC (one word only)	Move word, zero extended.
			Arithmetic
`add.`	SRC, DST	DST = DST + SRC	Addition.
`sub.`	SRC, DST	DST = DST - SRC	Subtraction.
`inc.`	DST	DST = DST + 1	Increment by one.
`dec.`	DST	DST = DST - 1	Decrement by one.
`mul.`	SRC	`%RDX:%RAX` = `%RAX` * SRC	Unsigned multiplication.
`imul.`	SRC	`%RDX:%RAX` = `%RAX` * SRC	Signed multiplication.
`div.`	SRC	`%RAX` = `%RDX:%RAX` / SRC `%RDX` = `%RDX:%RAX` % SRC	Unsigned division.
`idiv.`	SRC	`%RAX` = `%RDX:%RAX` / SRC `%RDX` = `%RDX:%RAX` % SRC	Signed division.
			Branching
`jmp`	ADDRESS		Jump to address (or label).
`je`	ADDRESS		Jump if equal.
`jne`	ADDRESS		Jump if not equal.
`jg`	ADDRESS		Jump if greater than.
`jge`	ADDRESS		Jump if greater or equal.
`jl`	ADDRESS		Jump if less than.
`jle`	ADDRESS		Jump if less or equal.
`call`	ADDRESS		Jump and push return address.
`ret`			Pop address and jump to it.
`loop`	ADDRESS		`decq %RCX`, jump if not zero.
			Logic and Shifting
`cmp.`	A, B	`sub` A B (Only set flags)	Compare and set condition flags.
`xor.`	SRC, DST	DST = SRC ^ DST	Bitwise exclusive or.
`or.`	SRC, DST	DST = SRC | DST	Bitwise inclusive or.
`and.`	SRC, DST	DST = SRC & DST	Bitwise and.
`shl.`	A, DST	DST = DST << A	Shift left by one bit.
`shr.`	A, DST	DST = DST >> A	Shift right by one bit
			Other
lea.	A, DST	DST = &A	Load effective address.
int	INT_NR		Software interrupt.

4.3. Registers & Variables

variables
- registers → fast and easy access
  - → caller saved → subroutines overwrite registers
- stacks
- main memory

4.4. The Stack



4.4.1. The Stack Pointer

RSP — stack pointer register
- → initialized by operating system at start
- → contains the address of the first byte after program's memory space
- → "grows" downward into program's memory space
- → push instruction → RSP value is decremented + pushed value stored at the new location at which the stack pointer then points

4.4.2. Cleaning up the Stack

stack overflow — overwrite program's code and data

```
pushq $42               # Push a magic number, the seventh argument
movq ..., %...          # Move arguments 2 through 6 to their registers
movq $formatstr, %rdi   # First argument: the format string
movq $0, %rax           # no vector arguments for printf
call printf             # Print the numbers
addq $8, %rsp           # Clean the stack (pop the magic number)
```

4.4.3. The Base Pointer

RBP — base pointer register

entry of subroutine
- → push value of RBP into the stack
- → copy current stack pointer value to RBP

end of subroutine → pop old RBP value off the stack

during subroutine → RBP points at the base of subroutine's stack area — find local variables and subroutine arguments

4.4.4. Subroutine Prologue and Epilogue

prologue – storing old base pointer and copying the stack pointer to be a new base pointer

epilogue – restoring the old base pointer

stack frame – space between RBP and RSP

RBP points opposite of RSP

4.5. Subroutines

block of instructions starting at memory address (label)