Assembly Language in UNIX

Advanced Programming Environment in the UNIX Environment

Chun-Ying Huang <chuang@cs.nctu.edu.tw>

Outline

Assembly Basics

Common Mnemonics

Function Call and Calling convention

Stack Frame

Integrate C and Assembly

- Assembly `hello world!'
- Library implemented in Assembly

Before We Start …

Materials used in this slide is based on x86_64 assembly, which is very similar to x86 instructions

This is not a complete assembly course, so we will not have a complete introduction to the instructions

- Try to read the online references or books, for example
- One possible assembly tutorial: https://www.tutorialspoint.com/assembly_programming/
- One textbook I used before: Assembly Language for x86 Processors, by Kip Irvine
- I personally prefer Intel assembly syntax (compared to AT&T), just a personal preference

Official Guide:

- Intel 64 and IA-32 Architectures Software Developer's Manual
- https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf

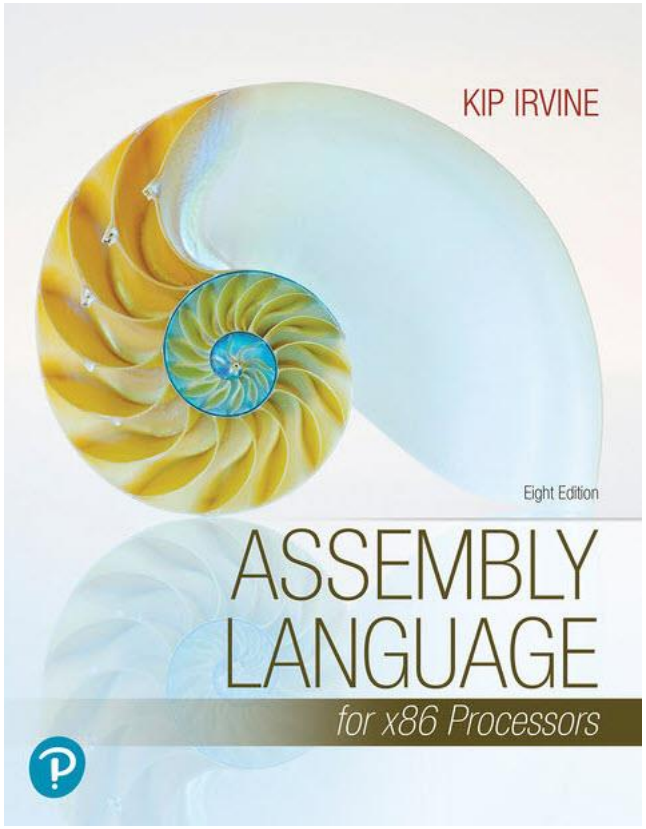
Reference Book

Assembly Language for x86 Processors, 8/e (2019)

- By Kip Irvine
- Prentice-Hall (Pearson Education)

組合語言(第七版)

- 譯者：白能勝, 王國華, 張子庭
- 全華圖書



Tools for Understanding Assembly in UNIX Environment

Additional compiler (gcc) options

- **-m32**: output Intel x86 object codes
- **-masm=intel**: Use Intel syntax instead of AT&T syntax
- **-fno-stack-protector**: disable stack protector ... might be easier for understanding stack frame

Assembler (yasm)

- $ **yasm -f elf32**: Output x86 object codes
- $ **yasm -f elf64**: Output x86_64 object codes

Linker (ld)

- $ **ld -m elf_i386**: Link with x86 object codes
- $ **ld -m elf_x86_64**: Link with x86_64 object codes

Tools for Understanding Assembly in UNIX Environment

Debugger (gdb-peda)

- PEDA - Python Exploit Development Assistance for GDB
- https://github.com/longld/peda

Installation (in $HOME/peda)

- $ git clone https://github.com/longld/peda.git ~/peda
- $ echo "source ~/peda/peda.py" >> ~/.gdbinit

Assembly Language Characteristics

Low level language

One-to-one mapping from mnemonics to machine codes

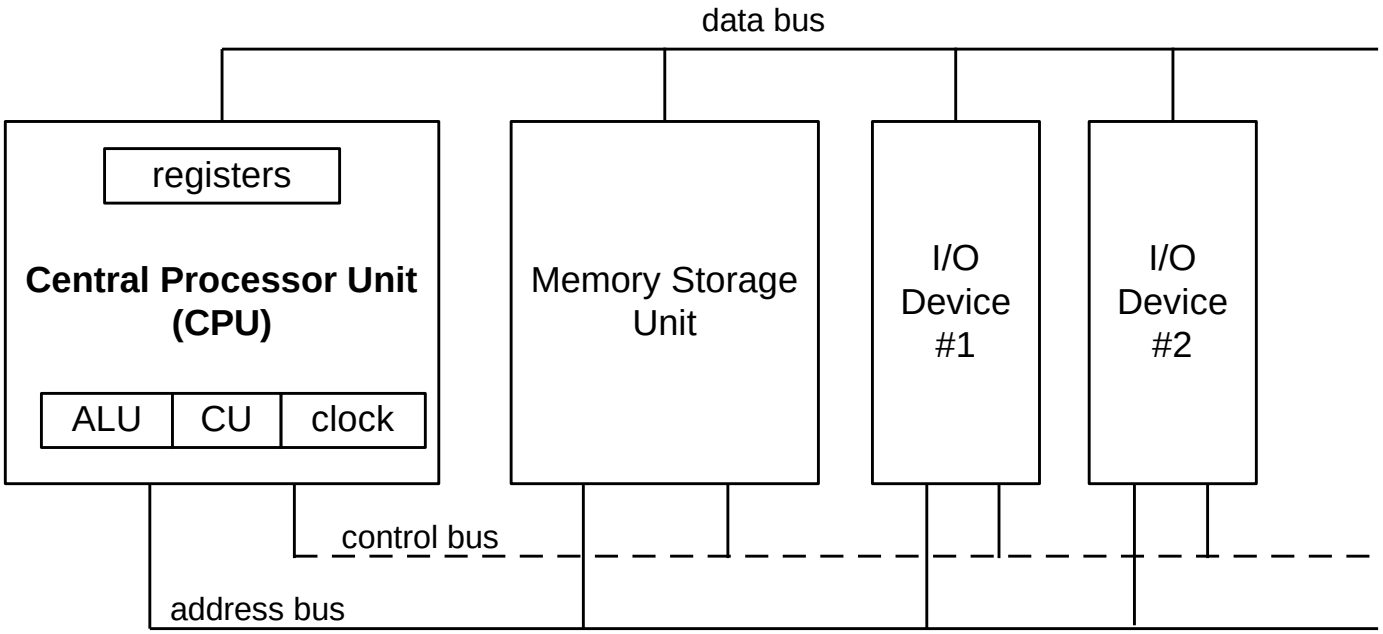
Assembler: Turn assembly codes to machine codes

Machine and platform-dependent

- Different machines/platforms have different assembler
- Even assemblers on the same machine/platform could be different

This course focuses on Intel x86_64 assembly on Linux platform

Basic Microcomputer Design



Instruction Execution Cycle

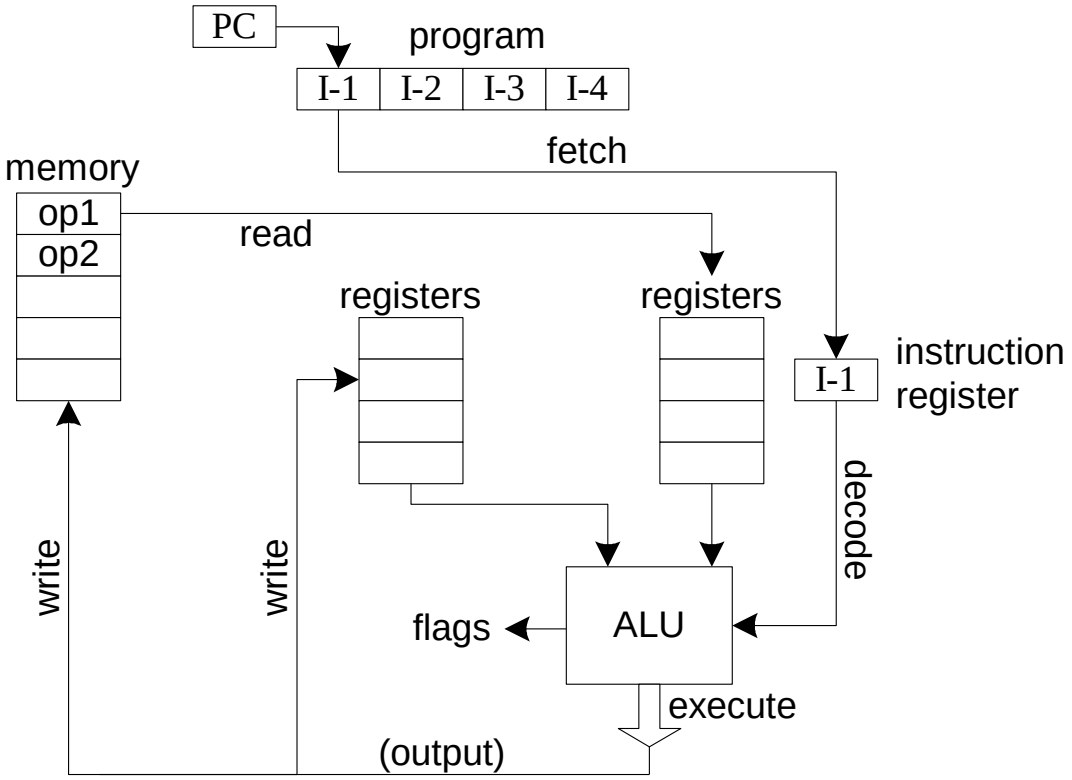
Fetch

Decode

Fetch operands

Execute

Store output



Basic Execution Environment

Addressable memory

General-purpose registers

Index and base registers

Specialized register uses

Status flags

Floating-point, MMX, XMM registers

Addressable Memory

Long mode (x86_64)

- 256 TB
- 48-bit virtual address

Protected mode (x86)

- 4 GB
- 32-bit virtual address

Real-address and Virtual-8086 modes

- 1 MB space
- 20-bit address

General Purpose Registers

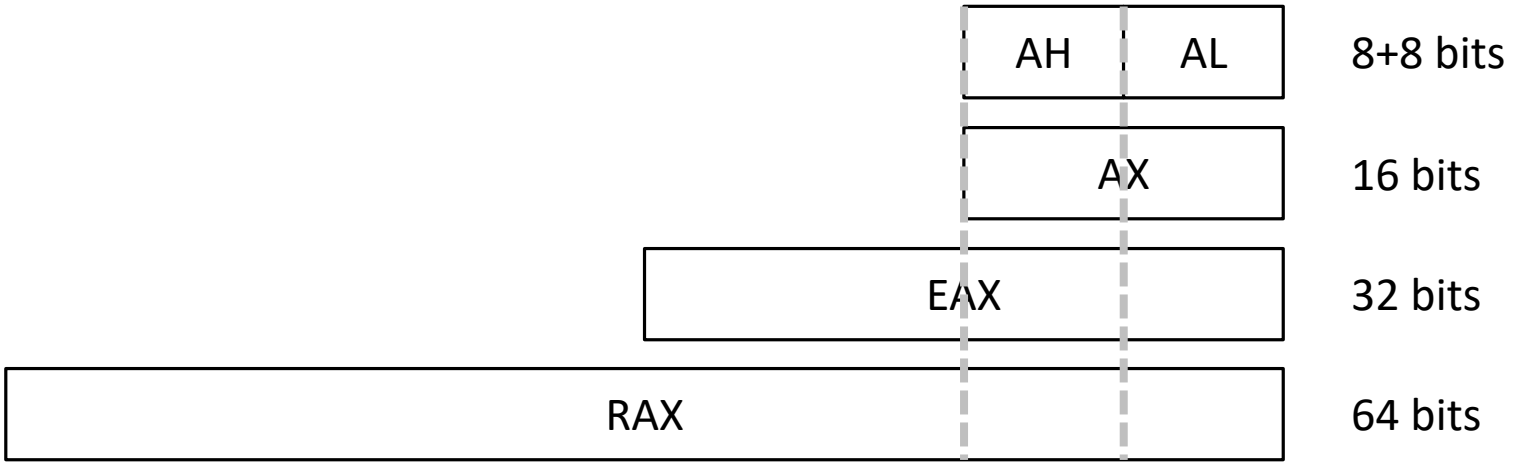
RAX	RDI	R8	R12
RBX	RSI	R9	R13
RCX	RBP	R10	R14
RDX	RSP	R11	R15

EFLAGS
RIP

Segment Registers

CS	ES
DS	FS
SS	GS

Access Parts of Registers



64 bit	32 bit (full)	16 bit	8 bit (high)	8 bit (low)
RAX	EAX	AX	AH	AL
RBX	EBX	BX	BH	BL
RCX	ECX	CX	CH	CL
RDX	EDX	DX	DH	DL

Index and Base Registers

Registers with alternative names

- Will replace the whole value

64-bit name	32-bit name	16-bit name
RDI	EDI	DI
RSI	ESI	SI
RBP	EBP	BP
RSP	ESP	SP

Register Uses

General purpose
- RAX – accumulator
- RCX – loop counter
- RSP – stack pointer
- RDI, RSI – index registers
- RBP – frame pointer

Segment
- CS – code segment
- DS – data segment
- SS – stack segment
- ES, FS, GS – additional

RIP
- Instruction counter

EFLASGS
- CPU flags (bit-by-bit)
- carry, overflow, sign, zero
- aux carry, parity, …

Common Format

Basic format (in a single line)

- Label (optional)
- Mnemonic (mandatory)
- Operands (optional: zero or more …)
- Commands (optional)

Mnemonics: mov, add, jmp, …

Operands: constant, constant expression, register, memory

- Constants are often called immediate values

Instruction Format Examples

No operand

- stc ; set carry flag

One operand

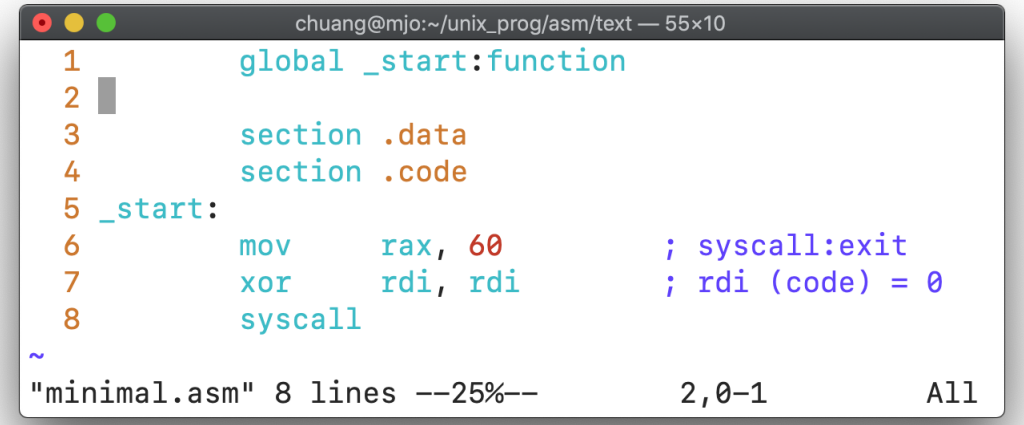
- inc eax ; eax = eax + 1
- inc BYTE **PTR** [a] ; a = a + 1 - PTR is required on MASM/gas but cannot be used on yasm/nasm

Two operands

- add ebx, ecx ; ebx = ebx + ecx
- sub BYTE [a], 25 ; a = a - 25
- add eax, 36*25 ; eax = eax + 36*25

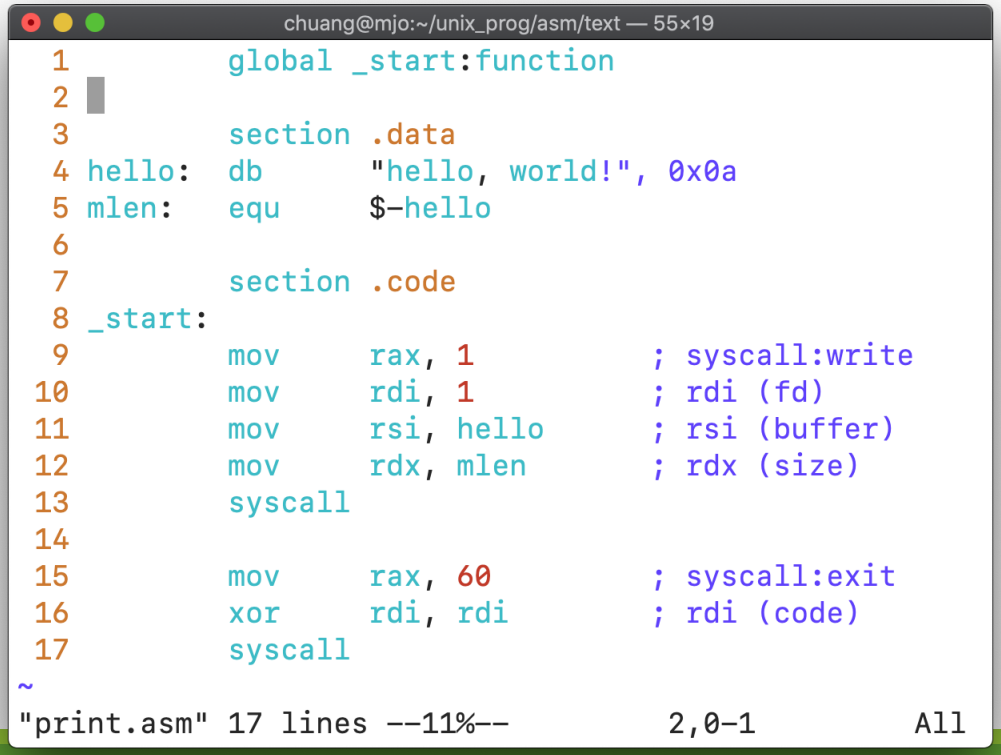
A Minimal Example

```
$ yasm -f elf64 -DYASM -D__x86_64__ -DPIC minimal.asm -o minimal.o
$ ld -m elf_x86_64 -o minimal minimal.o
```



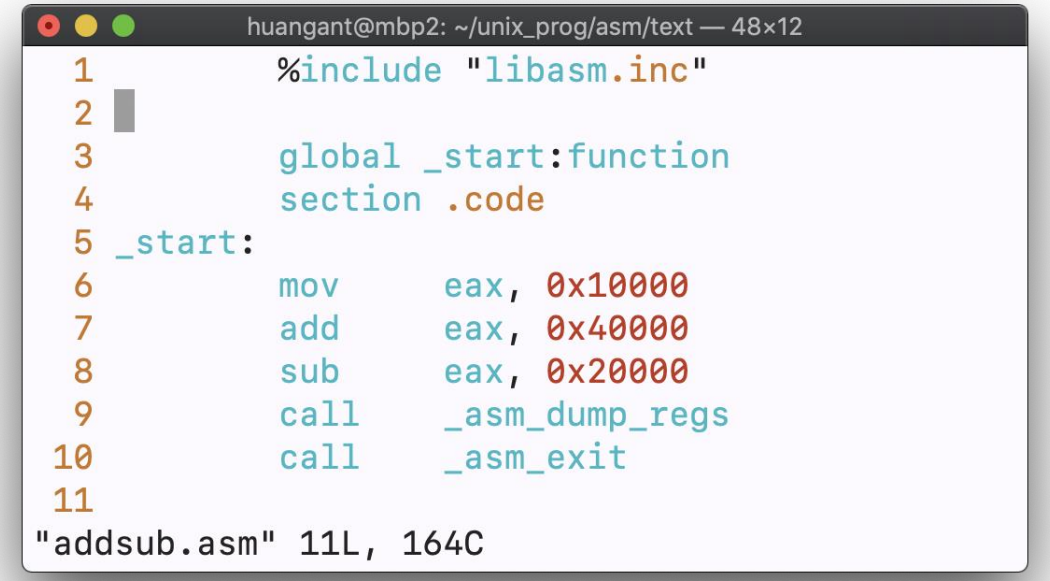
A Hello, World Example

```
$ yasm -f elf64 -DYASM -D__x86_64__ -DPIC print.asm -o print.o
$ ld -m elf_x86_64 -o print print.o
```



The "addsub.asm" Example

```
$ yasm -f elf64 -DYASM -D__x86_64__ -DPIC libasm64.asm -o libasm64.o
$ ar rc libasm64.a libasm64.o
$ yasm -f elf64 -DYASM -D__x86_64__ -DPIC addsub.asm -o addsub.o
$ ld -m elf_x86_64 -o addsub addsub.o libasm64.a
```



Initialized Data (Data Section)

INTEGER

- db – BYTE
- dw – WORD (2-byte)
- dd – DWORD (4-byte)
- dq – QWORD (8-byte)
- dt – tbyte (10-byte)
- ddq – dqword (16-byte)
- do – the same as ddq

FLOATING POINT

- dd – single-precision (4-byte)
- dq – double-precision (8-byte)
- dt – extended-precision (10-byte)

Initialized Data (Data Section) (Cont'd)

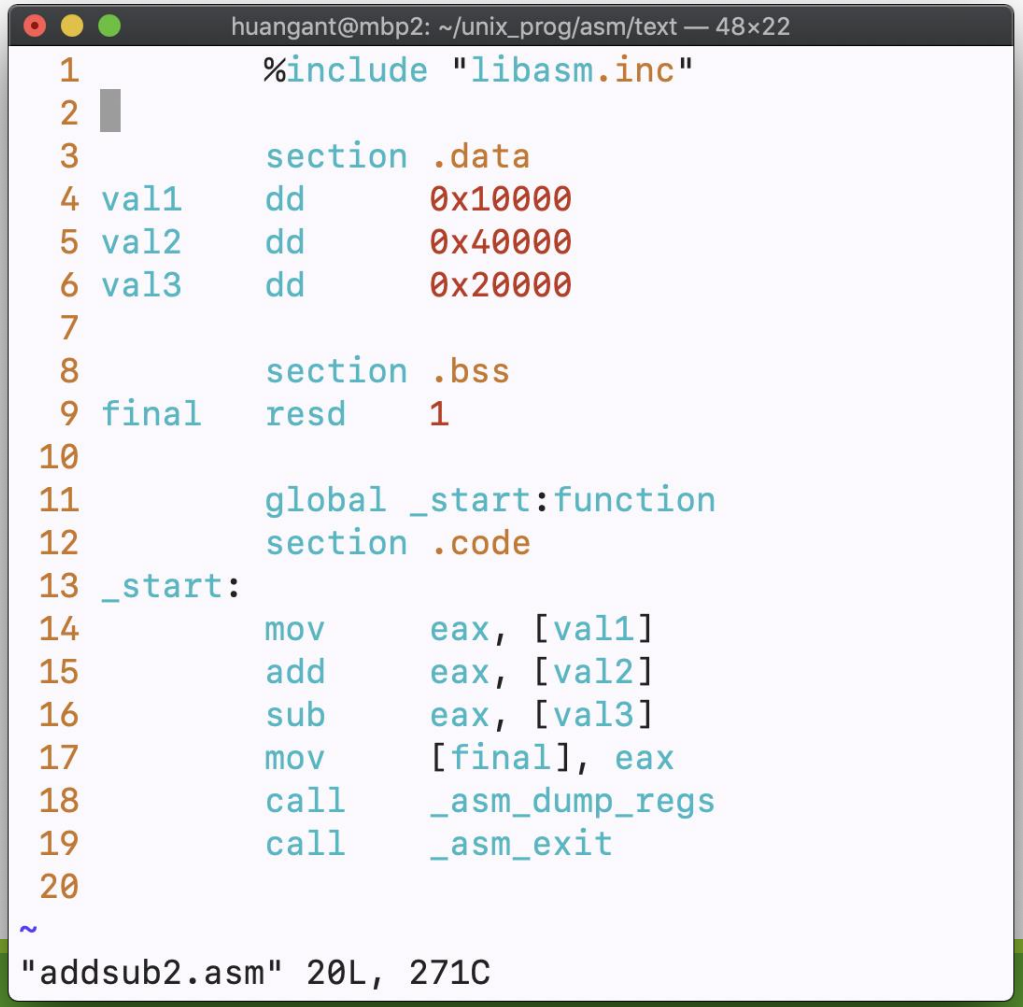
```
a       db      0x55                ; just the byte 0x55
b       db      0x55,0x56,0x57      ; three bytes in succession
c       db      'a',0x55            ; character constants are OK
d       db      'hello',13,10,'$'   ; so are string constants
e       dw      0x1234              ; 0x34 0x12
f       dw      'a'                 ; 0x41 0x00 (it's just a number)
g       dw      'ab'                ; 0x41 0x42 (character constant)
h       dw      'abc'               ; 0x41 0x42 0x43 0x00 (string)
i       dd      0x12345678          ; 0x78 0x56 0x34 0x12
j       dq      0x1122334455667788  ; 0x88 0x77 0x66 0x55 0x44 0x33 0x22 0x11
k       ddq     0x112233445566778899aabbccddeeff00
        ; 0x00 0xff 0xee 0xdd 0xcc 0xbb 0xaa 0x99
        ; 0x88 0x77 0x66 0x55 0x44 0x33 0x22 0x11
l       do      0x112233445566778899aabbccddeeff00 ; same as previous
m       dd      1.234567e20         ; floating-point constant
n       dq      1.234567e20         ; double-precision float
o       dt      1.234567e20         ; extended-precision float
```

Uninitialized Data (BSS Section)

- resb – 1-byte
- resw – 2-byte
- resd – 4-byte
- resq – 8-byte
- rest – 10-byte
- resdq – 16-byte
- reso – the same as resdq

```
buffer:         resb    64      ; reserve 64 bytes
wordvar:        resw    1       ; reserve a word
realarray       resq    10      ; array of ten reals
```

The "addsub2.asm" Example



Practice

addsub and addsub2 in gdb

addsub and addsub2 on Quiz Server

Constant & Repetition

The "equ" keyword

```
message db 'hello, world'
msglen  equ $-message
```

Repetition

```
zerobuf:        times 64 db 0
buffer:         db 'hello, world'
                times 64-$+buffer db ' '
```

Assembly Instructions

Data movement instructions

Simple arithmetic instructions

Shift and rotate instructions

Multiplication and division instructions

Data Movement Instructions

Syntax

- **MOV dst, src**
- dst = src
- **XCHG dst, src**
- dst, src = src, dst
- **Mnemonics** are case in-sensitive
- **Operands** can be register (**reg**), memory (**mem**), or immediate (constant) (**imm**) in units of 8-, 16-, 32-, 64-bit

Operands' guideline

- No more than one memory operand
- CS, RIP/EIP cannot be the destination
- No immediate (constant) to segment register moves
- Data sizes are usually equivalent and can be determined

Memory Operand

Access a memory address

- mov eax, [0x600000]
- mov eax, [0x600000+4]
- mov eax, [ebx + 4]
- mov eax, [ebx + ecx***4** + 4]
- mov [0x600004], ebx
- inc [0x600008] **; this is invalid**
- inc DWORD PTR [0x600008]
- mov [0x600000], [0x600004] **; this is invalid**

Can be 1, 2, 4, or 8

LEA: Load Effective Address

Move the address into the target operand

- lea eax, [0x600000] ; eax = 0x600000
- lea eax, [0x600000+4] ; eax = 0x600000 + 4
- lea eax, [ebx + 17] ; eax = ebx + 17
- lea eax, [ebx + ecx*4 + 4] ; eax = ebx + ecx*4 + 4

Some special usage

- Add a constant to a register
- Quick multiplication of 2, 3, 5, 9

Simple Arithmetic Instructions

Arithmetic operations affect the states of EFLAGS

Each FLAG is a single bit in EFLAGS

- ZF (zero): zero
- CF (carry): too large/small
- SF (sign): negative
- OF (overflow): invalid signed result
- PF (parity):
  even # of bit-1 in the lowest byte

Syntax	Operation
`inc dst`	dst = dst + 1
`dec dst`	dst = dst - 1
`add dst, src`	dst = dst + src
`sub dst, src`	dst = dst - src
`neg dst`	dst = -dst

Syntax	Operation
`and dst, src`	dst = dst & src
`or dst, src`	dst = dst | src
`xor dst`	dst = dst ^ src
`not dst`	dst = ~dst
`test dst, src`	dst & src => eflags
`cmp dst, src`	dst – src => flags

Practice

swapreg

swapmem

leax

eval1

Bitwise Operations

Truth table for a single bit

- Just like Boolean operations

AND

x	y	x ∧ y
0	0	0
0	1	0
1	0	0
1	1	1

OR

x	y	x ∨ y
0	0	0
0	1	1
1	0	1
1	1	1

XOR

x	y	x ⊕ y
0	0	0
0	1	1
1	0	1
1	1	0

NOT

X	¬X
F	T
T	F

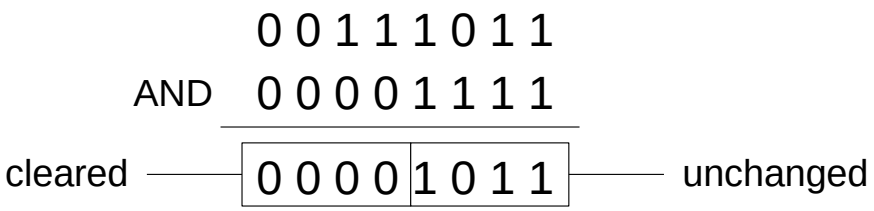
AND Instruction

Performs a Boolean AND operation between each pair of matching bits in two operands

Syntax:

AND *destination, source*

(same operand types as MOV)



AND

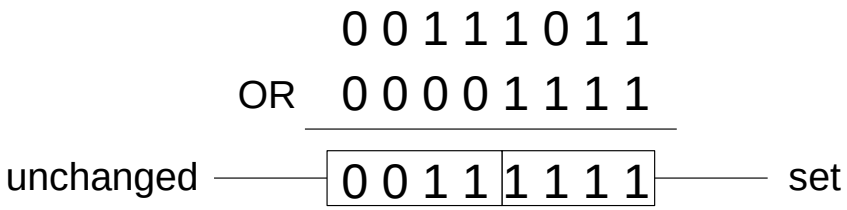
x	y	x ∧ y
0	0	0
0	1	0
1	0	0
1	1	1

OR Instruction

Performs a Boolean OR operation between each pair of matching bits in two operands

Syntax:

OR *destination, source*



OR

x	y	x ∨ y
0	0	0
0	1	1
1	0	1
1	1	1

XOR Instruction

Performs a Boolean exclusive-OR operation between each pair of matching bits in two operands

Syntax:

XOR *destination, source*

```
  0 0 1 1 1 0 1 1
XOR 0 0 0 0 1 1 1 1
  -----------------
  0 0 1 1 | 0 1 0 0
unchanged     inverted
```

XOR

x	y	x ⊕ y
0	0	0
0	1	1
1	0	1
1	1	0

XOR is a useful way to toggle (invert) the bits in an operand.

NOT Instruction

Performs a Boolean NOT operation on a single destination operand

Syntax:

NOT *destination*

NOT 0 0 1 1 1 0 1 1

1 1 0 0 0 1 0 0 ——— inverted

NOT

X	¬X
F	T
T	F

Practice

tolower

clear17

dec2ascii

ul+lu

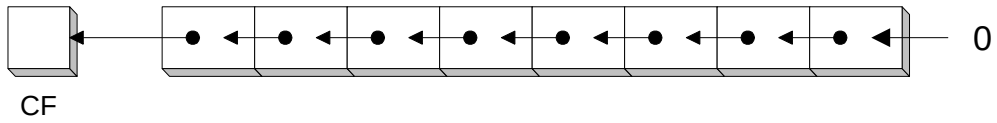
Shift Instructions

Shift left ➔ value = value * 2

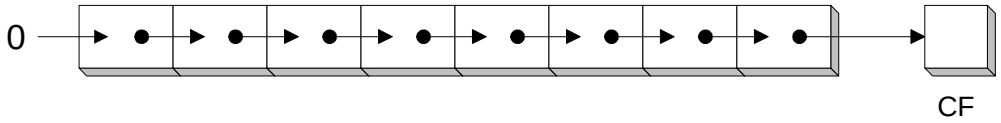
Shift right ➔ value = value / 2

Logical shift

- SHL reg/mem, imm8/cl

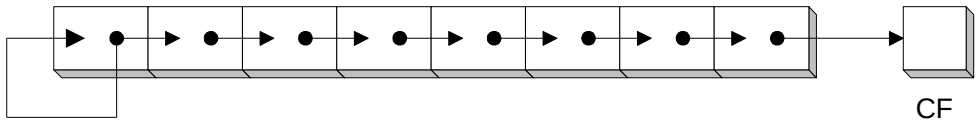


- SHR reg/mem, imm8/cl



Arithmetic shift right

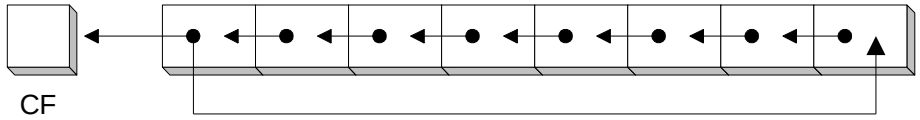
- SAL == SHL
- SAR reg/mem, imm8/cl



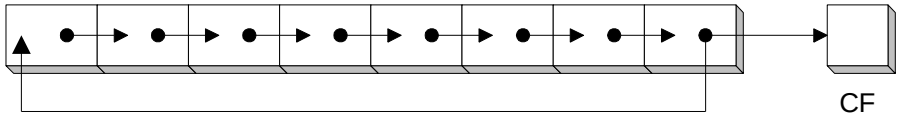
Rotate Instructions

Rotate w/o the carry flag

- ROL reg/mem, imm8/cl

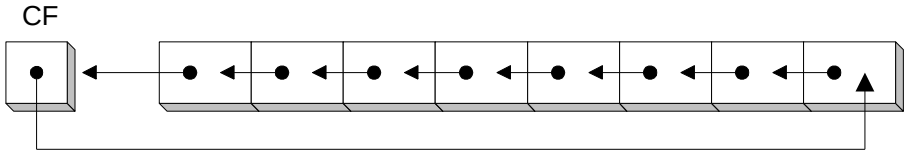


- ROR reg/mem, imm8/cl

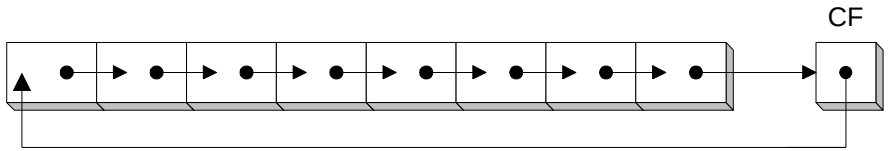


Rotate with the carry flag

- RCL reg/mem, imm8/cl



- RCR reg/mem, imm8/cl



Practice

mulbyshift

isolatebit

Multiplication and Division

The design of multiplication and division instructions is ***totally different from*** other arithmetic instructions!!!

Multiplication

- Put multiplicand (被乘數) in AL/AX/EAX/RAX
- **MUL** multiplier (乘數) (or use **IMUL** for signed integer)
- Result is stored in AX/DX:AX/EDX:EAX/RDX:RAX

Division

- Put dividend (被除數) in AX/DX:AX/EDX:EAX/RDX:RAX
- **DIV** divisor (除數) (or use **IDIV** for signed integer)
- Result: Quotient (商) in AL/AX/EAX/RAX;
  Remainder (餘數) in AH/DX/EDX/RDX

Multiplication

Syntax

- **MUL** multiplier (reg/mem)
- reg/mem can be 8-, 16-, 32-, or 64-bit
- Size of multiplicand is ***implicitly*** determined by the multiplier

Multiplicand	Multiplier	Product
AL	reg/mem8	AX
AX	reg/mem16	DX:AX
EAX	reg/mem32	EDX:EAX
RAX	reg/mem64	RDX:RAX

Division

Syntax

- **DIV** divisor (reg/mem)
- reg/mem can be 8-, 16-, 32-, or 64-bit
- Size of dividend is ***implicitly*** determined by the divisor

Dividend	Divisor	Quotient	Remainder
AX	reg/mem8	AL	AH
DX:AX	reg/mem16	AX	DX
EDX:EAX	reg/mem32	EAX	EDX
RAX:RDX	reg/mem64	RAX	RDX

Division (Cont'd)

Suppose we plan to implement an equation A / B

- Suppose both A and B are N-bit numbers, N can be 8, 16, 32, or 64
- We have to convert A to be a 2*N-bit number

Relevant instructions (with sign extension)

- CBW: extend AL ==> AX
- CWD: extend AX ==> DX:AX
- CDQ: extend EAX ==> EDX:EAX
- CQO: extend RAX ==> RDX:RAX

No instruction is required for extending unsigned numbers

- Simply fill AH/DX/EDX/RDX with zero

Practice

Implement mathematic equations

- var4 = (var1 + var2) * var3
- eax = (-var1 * var2) + var3
- var4 = (var1 * 5) / (var2 – 3)
- var4 = (var1 * -5) / (-var2 % var3)
- var3 = (var1 * -var2) / (var3 – ebx)

Notice

- You cannot change the values of variables in RHS

Control Flow Instructions (1/3)

Review: **TEST** and **CMP** instruction => Change EFLAGS

Jumps based on flags or equality

Mnemonic	Description	Flags
JZ	Jump if zero	ZF = 1
JNZ	Jump if not zero	ZF = 0
JC	Jump if carry	CF = 1
JNC	Jump if not carry	CF = 0
JO	Jump if overflow	OF = 1
JNO	Jump if not overflow	OF = 0
JS	Jump if signed	SF = 1
JNS	Jump if not signed	SF = 0
JP	Jump if parity (even)	PF = 1
JNP	Jump if not parity (odd)	PF = 0

Mnemonic	Description
JE	Jump if equal (*leftOp* = *rightOp*)
JNE	Jump if not equal (*leftOp* ≠ *rightOp*)
JCXZ	Jump if CX = 0
JECXZ	Jump if ECX = 0

Control Flow Instructions (2/3)

Jumps based on unsigned comparisons

Mnemonic	Description
JA	Jump if above (if *leftOp* > *rightOp*)
JNBE	Jump if not below or equal (same as JA)
JAE	Jump if above or equal (if *leftOp* >= *rightOp*)
JNB	Jump if not below (same as JAE)
JB	Jump if below (if *leftOp* < *rightOp*)
JNAE	Jump if not above or equal (same as JB)
JBE	Jump if below or equal (if *leftOp* <= *rightOp*)
JNA	Jump if not above (same as JBE)

Control Flow Instructions (3/3)

Jumps based on signed comparisons

Mnemonic	Description
JG	Jump if greater (if *leftOp* > *rightOp*)
JNLE	Jump if not less than or equal (same as JG)
JGE	Jump if greater than or equal (if *leftOp* >= *rightOp*)
JNL	Jump if not less (same as JGE)
JL	Jump if less (if *leftOp* < *rightOp*)
JNGE	Jump if not greater than or equal (same as JL)
JLE	Jump if less than or equal (if *leftOp* <= *rightOp*)
JNG	Jump if not greater (same as JLE)

Implement Conditional Statements

Human Readable Codes

```
if (condition) {
    Block_A
}
```

Straightforward Implementation

```
    cmp or test
    Jcond L1
    JMP L2
L1:
    Block_A
L2:
```

Fall-through Implementation

```
    cmp or test
    J<INV>cond L1
    Block_A
L1:
```

Implement Conditional Statements

Human Readable Codes

```
if (condition) {
    Block_A
} else {
    Block_B
}
```

Straightforward Implementation

```
    cmp or test
    Jcond L1
    JMP L2
L1:
    Block_A
    JMP L3
L2:
    Block_B
L3:
```

Fall-through Implementation

```
    cmp or test
    Jcond L1
    Block_B
    JMP L2
L1:
    Block_A
L2:
```

Implement *Compound* Conditional Statements – AND

Human Readable Codes

```
if (cond1 and cond2) {
    Block_A
} else {
    Block_B
}
```

Fall-through Implementation

```
    cmp1 or test1
    J<INV>cond1 L1
    cmp2 or test2
    J<INV>cond2 L1
    Block_A
    JMP L2
L1:
    Block_B
L2:
```

Implement *Compound* Conditional Statements – OR

Human Readable Codes

```
if (cond1 or cond2) {
    Block_A
} else {
    Block_B
}
```

Fall-through Implementation

```
    cmp1 or test1
    Jcond1 L1
    cmp2 or test2
    Jcond2 L1
    Block_B
    JMP L2
L1:
    Block_A
L2:
```

Implement Control Flows

FOR loop

WHILE loop

FOR Loop

CPU built-in loops

- Use CX/ECX/RCX as the counter

Repeat the loop for 10 times

```
        mov    ecx, 10
L1:
        ; do something here
        loop   L1
```

FOR Loop (Cont'd)

C-style FOR loop

- Suppose we already have the variable ***i***
- `for (i = 0; i < 10; i++) { /* … */ }`

```
        mov     DWORD [i], 0
L1:
        cmp     DWORD [i], 10
        jge     L2

        ; do something here

        inc     DWORD [i]
        jmp     L1
L2:
```

WHILE Loop

C-style FOR loop

- Suppose we already have the variable ***i***
- `while (cond) { /* … */ }`

```
L1:
        cmp or test
        J<INV>cond L2

        ; do something here

        jmp    L1
L2:
```

Practice

posneg

loop15

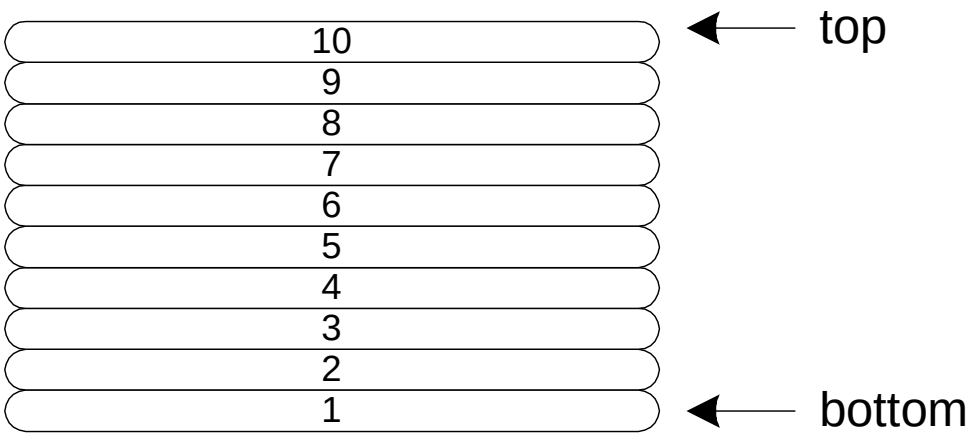
dispbin

bubble

Stack

A stack with plates

- Plates are only added to the top
- Plates are only removed from the top
- LIFO structure: last-in first-out



Stack (Cont'd)

The CPU stack

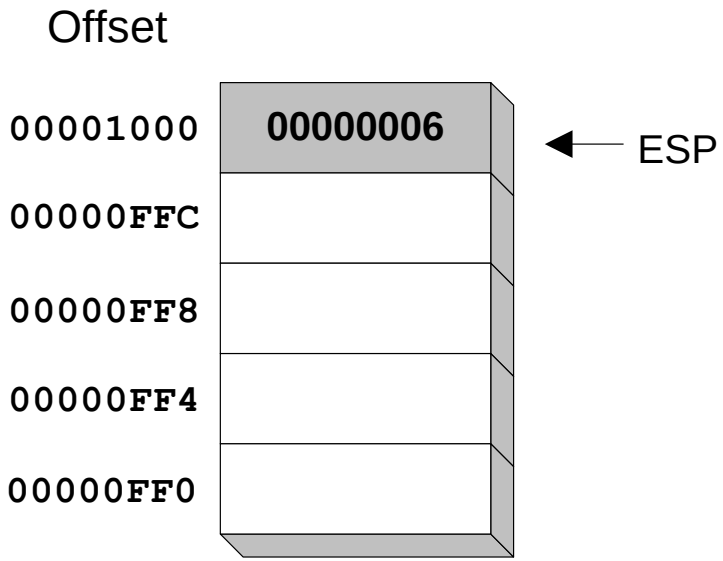
- A temporary storage
- Used to maintain program states
- RSP/ESP points to the top of the stack
- The stack grows from higher addresses to lower addresses

PUSH

- Put a number into the stack
- ESP = ESP - sizeof(object)
- [ESP] = object

POP

- Remove a number from the stack
- ESP = ESP + sizeof(top object)

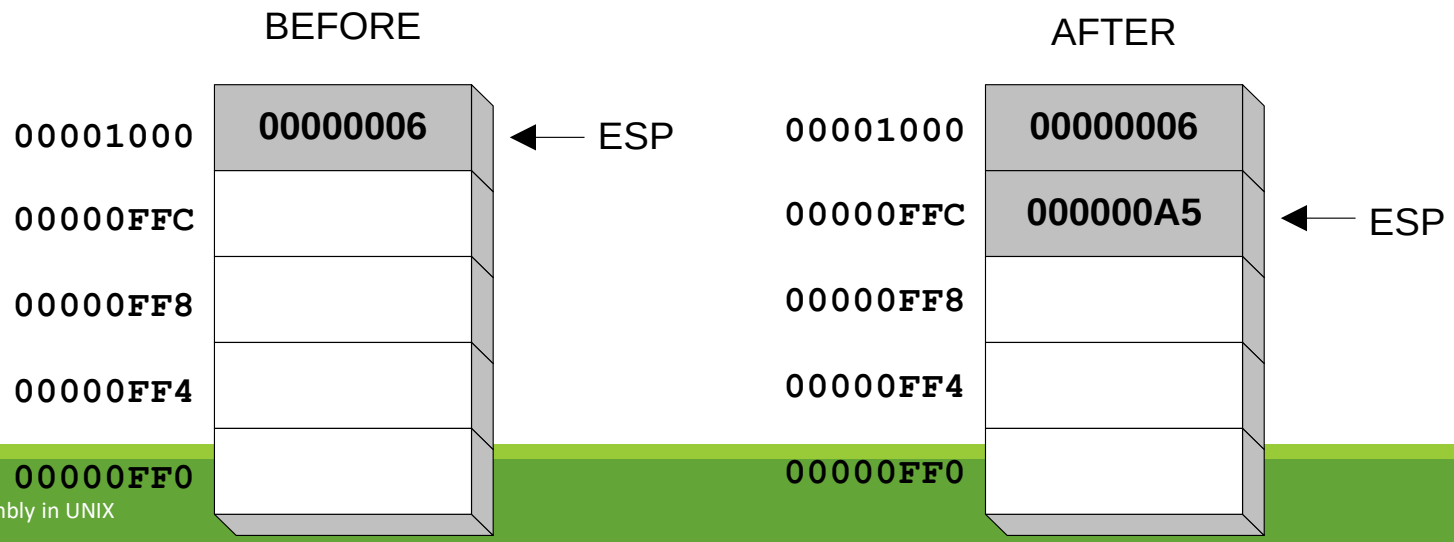


PUSH Operation

Syntax

- push r/m16
- push r/m32
- push r/m64
- push imm32

A 32-bit example

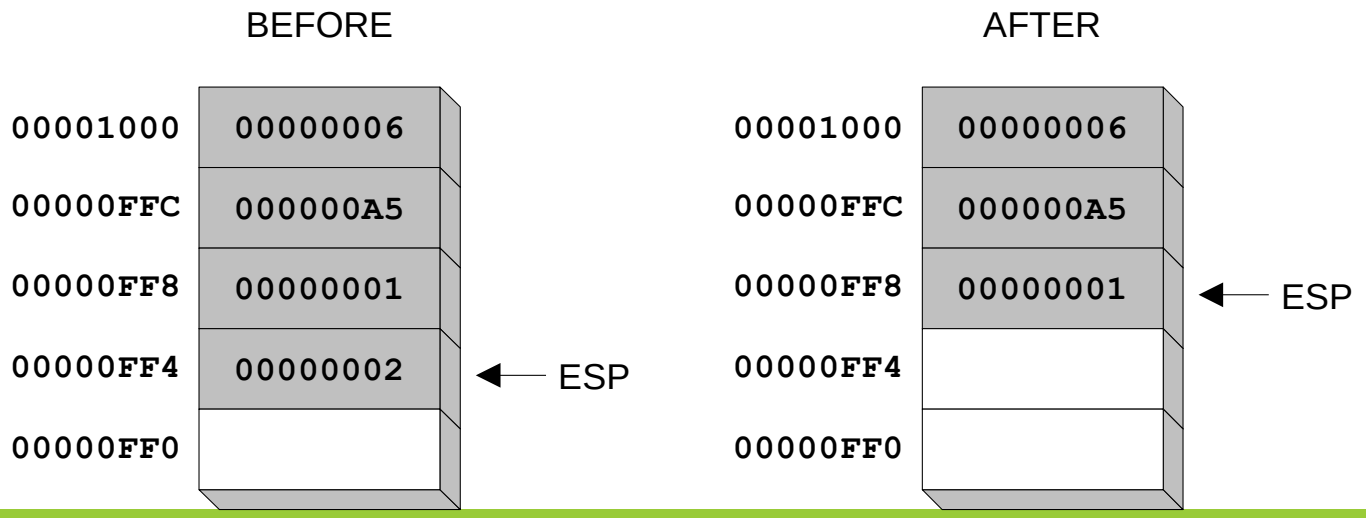


POP Operation

Syntax

- pop r/m16
- pop r/m32
- pop r/m64

A 32-bit example



Function Calls

A program always has to make a number of function (or API) calls

Functions, APIs, or libraries can be implemented in different languages. However, …

- Codes must be compiled into object files (machine codes)
- Object files can be aggregated into a single library
- Developers and compilers must know how a function is called at the instruction level

Therefore, we need to define "calling convention" for interoperable object (machine) codes

CALL and RET Instruction

CALL: change program control flow

- Call a function
- PUSH return address into the stack
- Set RIP to be the entry point of the target function
- Parameters can be passed by using registers or the stack

RET: return to the caller

- pop rip
- … but you cannot do this by yourself – dst cannot be RIP/EIP
- Return value is passed by using RAX/EAX register

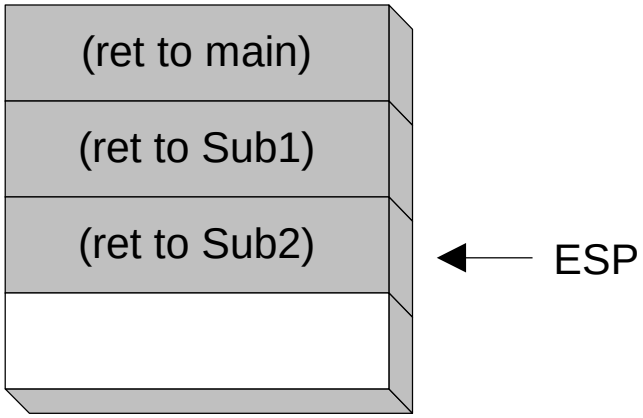
```
main:
     00000020 call MySub
     00000025 mov eax,ebx
     .
     .

MySub:
     00000040 mov eax,edx
     .
     .
     ret
```

Nested Function Call

Suppose we have called Sub3

The return addresses are stored in the stack




```
main PROC
    .
    .
    call Sub1
    exit
main ENDP

Sub1 PROC
    .
    .
    call Sub2
    ret
Sub1 ENDP

Sub2 PROC
    .
    .
    call Sub3
    ret
Sub2 ENDP

Sub3 PROC
    .
    .
    ret
Sub3 ENDP
```

Pass Parameters to a Function

Use registers

- Easier and faster
- Limited number of stacks ==> Limited number of parameters

Use stack

- Basically no limitation on the number of parameters (only depends on the available stack space)

Calling Convention

Calling convention is language / architecture dependent

- Different languages / machines / architectures could have different calling conventions

How about system calls? – That's another story!

- System calls have their own "calling convention"

What do we have to know about a calling convention?

- Where is the return address?
- How parameters are passed (before making a call)
- How parameters are cleared (after returning from a call)
- How return values are received

Simplified Calling Convention: Intel x86

cdecl: C/C++

- Caller pushes parameters to the stack, from right to left
- Caller is responsible to remove parameters from the stack

stdcall: Windows API

- Caller pushes parameters to the stack, from right to left
- Callee is responsible to remove parameters from the stack

Return value is stored in EAX register

Simplified Calling Convention: Intel x86_64

Registers can be used to pass function call parameters

System V AMD64

- The first 6 are passed by using registers RDI, RSI, RDX, RCX, R8, R9
- The rest are pushed onto stack from right to left

Microsoft x64

- The first 4 are passed by using registers RCX, RDX, R8, R9
- The rest are pushed onto the stack from right to left

Return value is stored in RAX

Stack parameters are always removed by the caller

Function Call in C (x86_64)

Compile the program

- gcc -o fun1 -fno-stack-protector fun1.c

Several options to dump assembly codes

Option #1: compiler options

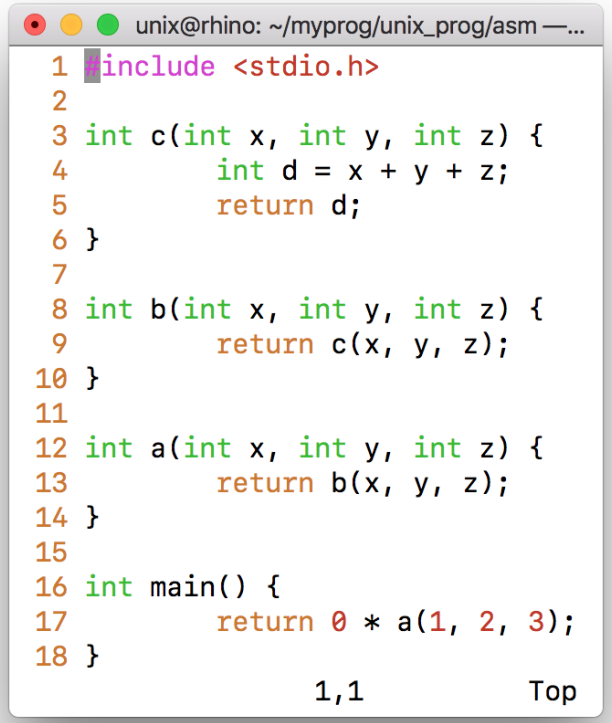
- with **-S** and **-masm=intel**

Option #2: use objdump

- $ **objdump -d fun > /tmp/fun.out**

Option #3: use gdb (or gdb-peda)

- Use the examine command (x)



fun1.c - Dumped Assembly Codes

Hint: x86_64 calling convention

- RDI, RSI, RDX, RCX, R8, R9

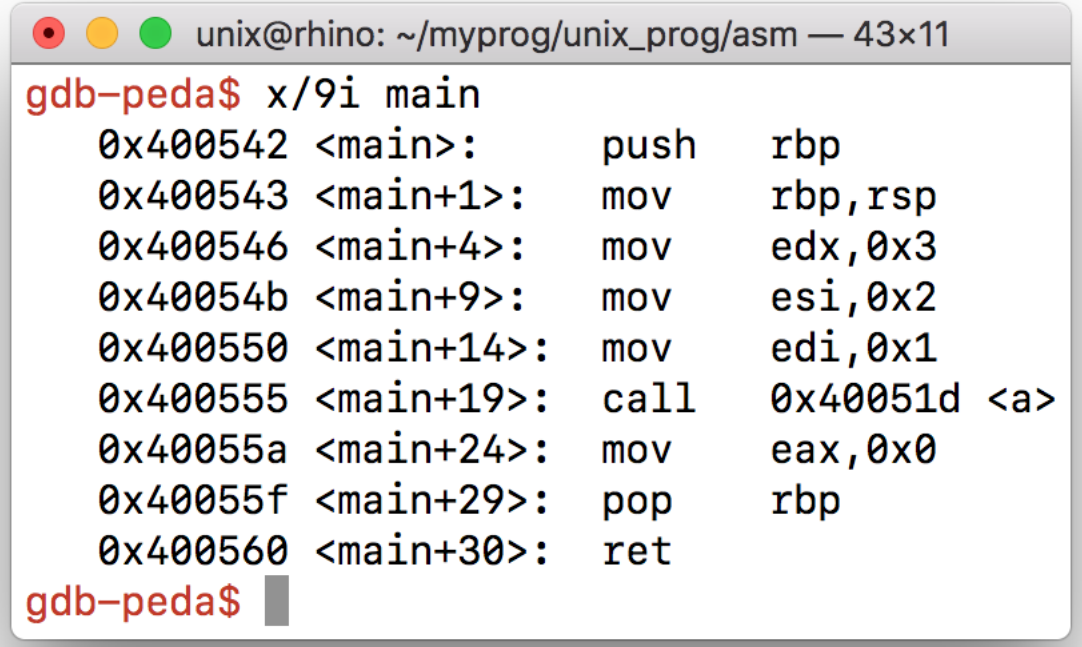
main() calls **a(1, 2, 3)**

The ***call*** instruction

- Push return address (0x40055a in this case)
- Set EIP = callee

The ***ret*** instruction

- ***ret*** == ***pop eip***



fun1.c - Dumped Assembly Codes

unix@rhi

```
gdb-peda$ x/14i a
   0x40051d <a>:
   0x40051e <a+1>:
   0x400521 <a+4>:
   0x400525 <a+8>:
   0x400528 <a+11>:
   0x40052b <a+14>:
   0x40052e <a+17>:
   0x400531 <a+20>:
   0x400534 <a+23>:
   0x400537 <a+26>:
   0x400539 <a+28>:
   0x40053b <a+30>:
   0x400540 <a+35>:
   0x400541 <a+36>:
gdb-peda$
```

unix@rhino: ~/myprog/unix_prog/asm — 56×16

```
gdb-peda$ x/14i c
   0x4004d6 <c>:        push   rbp
   0x4004d7 <c+1>:      mov    rbp,rsp
   0x4004da <c+4>:      mov    DWORD PTR [rbp-0x14],edi
   0x4004dd <c+7>:      mov    DWORD PTR [rbp-0x18],esi
   0x4004e0 <c+10>:     mov    DWORD PTR [rbp-0x1c],edx
   0x4004e3 <c+13>:     mov    edx,DWORD PTR [rbp-0x14]
   0x4004e6 <c+16>:     mov    eax,DWORD PTR [rbp-0x18]
   0x4004e9 <c+19>:     add    edx,eax
   0x4004eb <c+21>:     mov    eax,DWORD PTR [rbp-0x1c]
   0x4004ee <c+24>:     add    eax,edx
   0x4004f0 <c+26>:     mov    DWORD PTR [rbp-0x4],eax
   0x4004f3 <c+29>:     mov    eax,DWORD PTR [rbp-0x4]
   0x4004f6 <c+32>:     pop    rbp
   0x4004f7 <c+33>:     ret
gdb-peda$
```

Stack Frame

Look at the first few and last few lines of a function …

Entering a function

- `push rbp`
- `mov rbp, rsp`
- `sub rsp, 0x10 ; locals`

Leaving a function

- `leave ; mov rsp, rbp`
  `; pop rbp`
- `ret`

main() → a() → b() → c()

Addr	Value
0000	
...	
	0x7fffffffe490 [RBP of b()]
	0x40051b [ret to b()]
	... 16 bytes local storage ...
	0x7fffffffe4b0 [RBP of a()]
	0x400540 [ret to a()]
	... 16 bytes local storage ...
	0x7fffffffe4c0 [RBP of main()]
...	0x40055a [ret to main()]
...	
FFFF	

Stack Frame – RBP/EBP

RBP (EBP) is the ***frame pointer*** of the current function call

- Push RBP (EBP): preserve the frame pointer for the caller
- Leave: restore the frame pointer for the caller

It is the baseline to access ***parameters*** and ***local variables***

- For ***x86***, all the parameters are stored in the stack
- For ***x86-64***, parameters after the 7th are stored in the stack (SysV)

unix@rhino: ~/myprog/unix_prog/asm — 49×11

```
1  #include <stdio.h>
2
3  int a(int x, int y, int z) {
4          int p = 0x111, q = 0x222, r = 0xccc;
5          return x*p + y*q + z*r;
6  }
7
8  int main() {
9          return a(1, 2, 3);
10 }
                              1,1           All
```

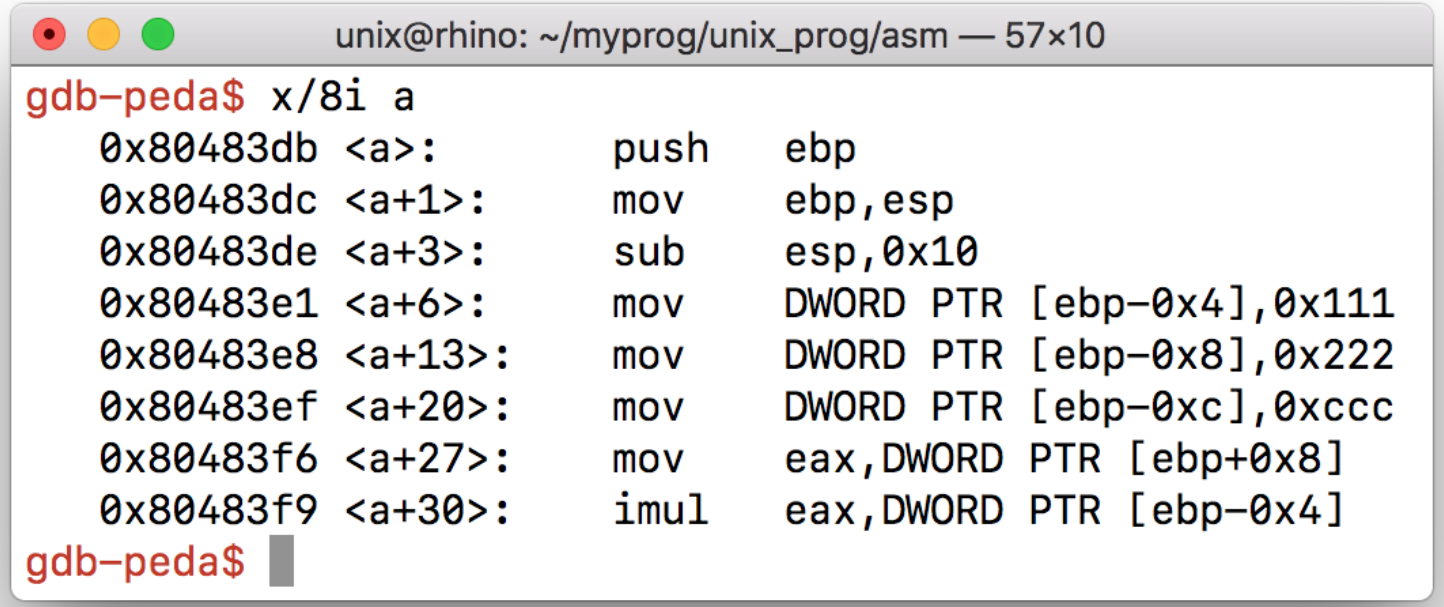
unix@rhino: ~/myprog/unix_prog/asm — 45×11

```
gdb-peda$ x/9i main
   0x8048413 <main>:    push   ebp
   0x8048414 <main+1>:  mov    ebp,esp
   0x8048416 <main+3>:  push   0x3
   0x8048418 <main+5>:  push   0x2
   0x804841a <main+7>:  push   0x1
   0x804841c <main+9>:  call   0x80483db <a>
   0x8048421 <main+14>: add    esp,0xc
   0x8048424 <main+17>: leave
   0x8048425 <main+18>: ret
gdb-peda$
```

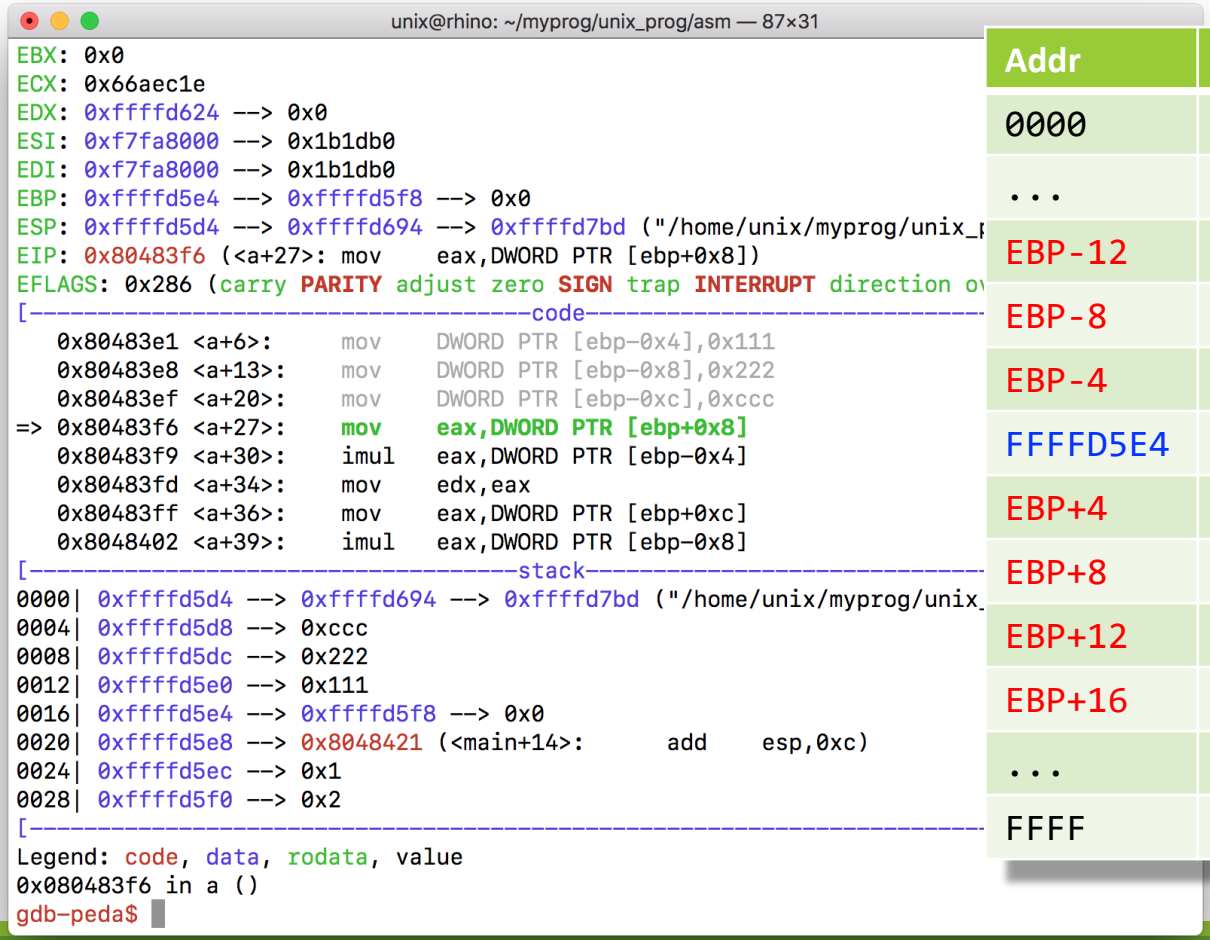
Stack Frame – x86 Function

Let's see the stack frame for function **a()**

fun2.c compiled for x86



Stack Frame – x86 Function (Cont'd)



Addr	Value
0000	
...	
EBP-12	0xccc [local]
EBP-8	0x222 [local]
EBP-4	0x111 [local]
FFFFD5E4	0xffffd5f8 [EBP of main()]
EBP+4	0x8048421 [return to main()]
EBP+8	0x01 [parameter #1]
EBP+12	0x02 [parameter #2]
EBP+16	0x03 [parameter #3]
...	
FFFF	

Practice

minicall

recur

Integrate C and Assembly

Now we knows ***calling convention*** and ***stack frames***

We can start working with Assembly!

Case Study

- Assembly `Hello, World!'
- Library implemented in Assembly

Additional notes …

- Work with a C library: You can call functions in C library
- Work without a C library: Work with system calls

Assembly `Hello, World!' (1/3)

The C implementation

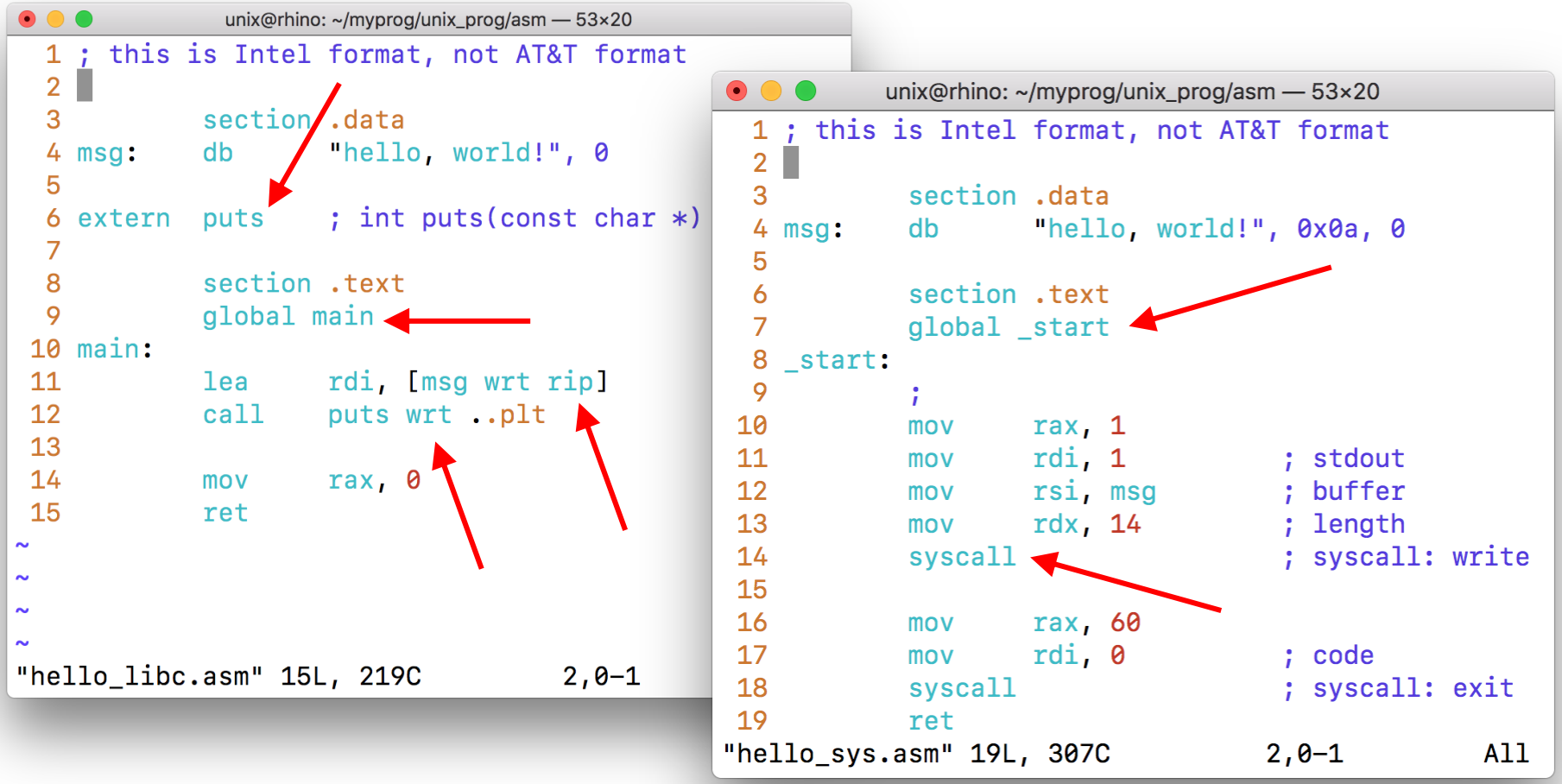
unix@rhino: ~/myprog — 36×7

```
  1 #include <stdio.h>
  2
  3 int main() {
  4         puts("hello, world!");
  5         return 0;
  6 }
"hello.c" 6L, 70C 2,0-1          All
```

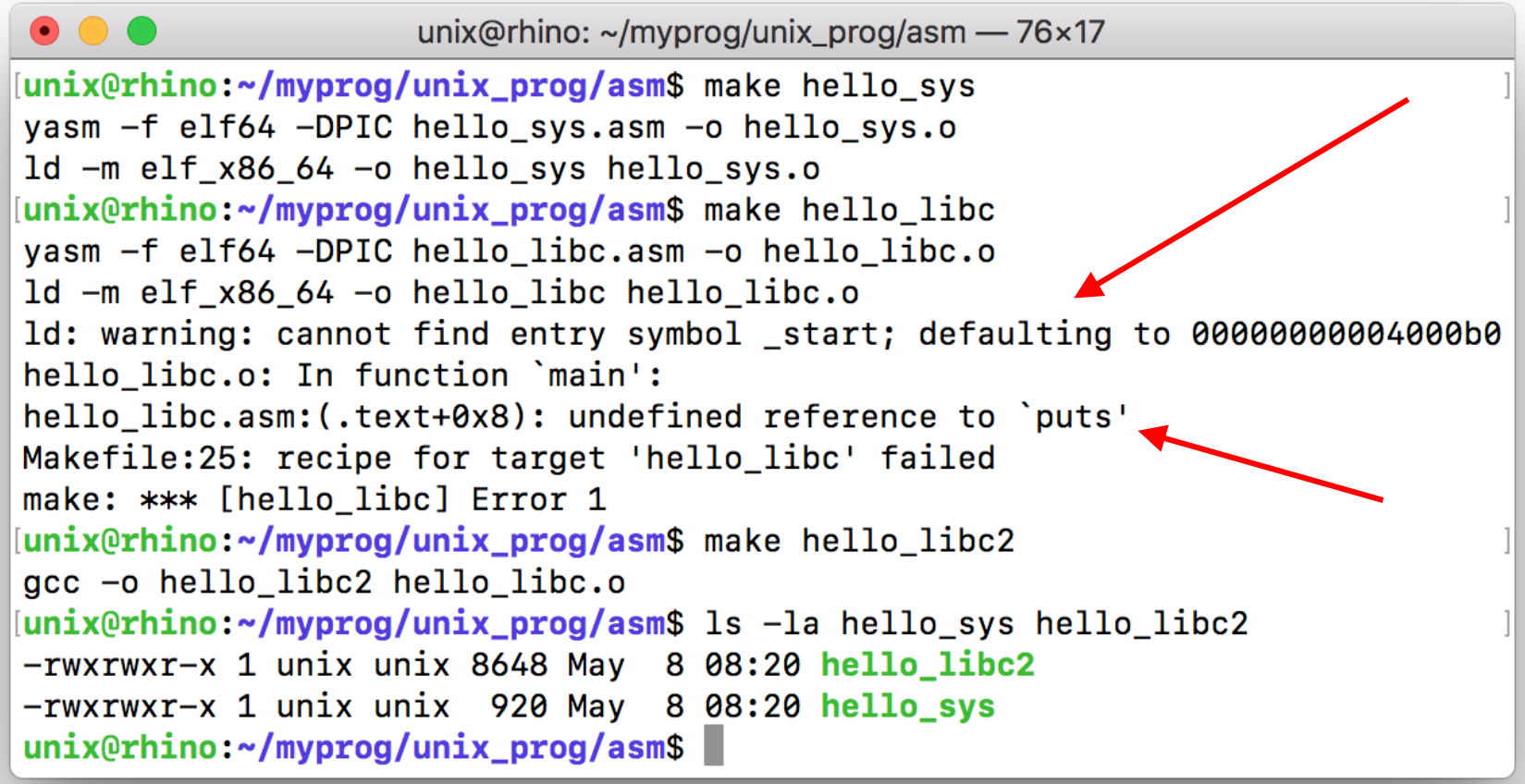
unix@rhino: ~/myprog — 61×5

```
[unix@rhino:~/myprog$ gcc hello.c -o hello
[unix@rhino:~/myprog$ strip hello
[unix@rhino:~/myprog$ ls -la hello
-rwxrwxr-x 1 unix unix 6312 May  6 10:17 hello
unix@rhino:~/myprog$
```

Assembly `Hello, World!' (2/3)



Assembly `Hello, World!' (3/3)



Library Implemented in Assembly

Work without a C library

What do you need?

- A self-made header file (**libmini.h**)
- A self-made ``start.o" that implements ***_start*** function
- A self-made mini C library
- Implement in Assembly

Examples

- CAT program
- sleep program

chuang@mjo:~/unix_prog/asm — 58×22

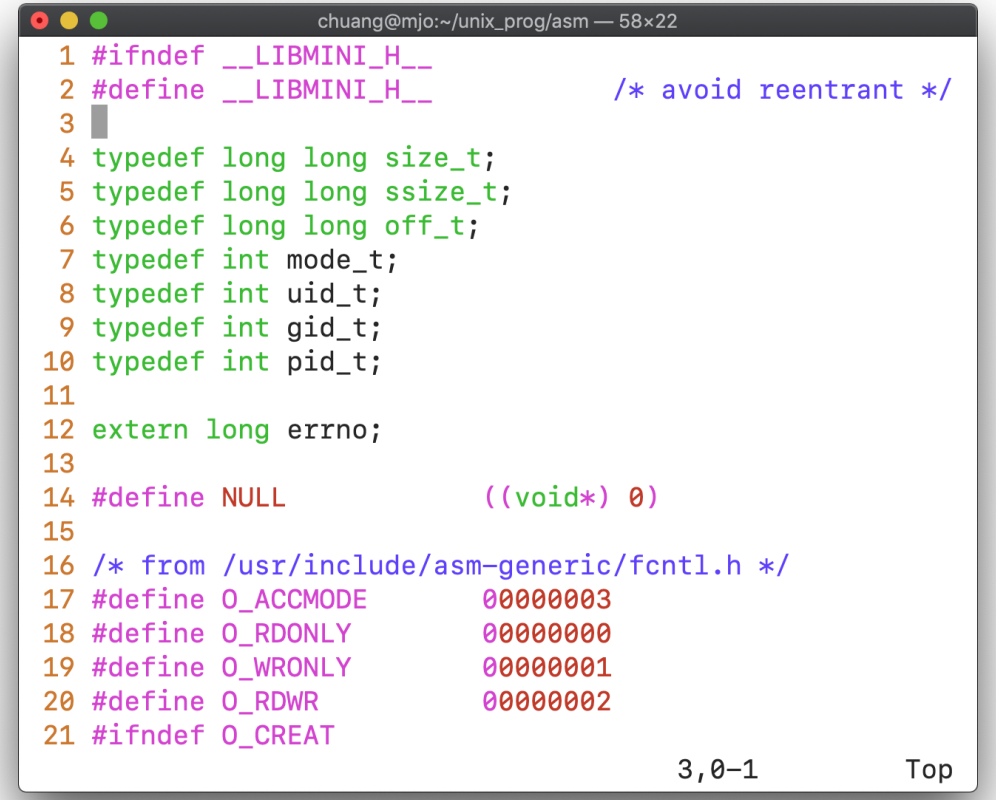
```
 1 #include "libmini.h"
 2
 3 int main(int argc, char *argv[]) {
 4     int fd = 0, rlen;
 5     char buf[4096];
 6     if(argc > 1) {
 7         if((fd = open(argv[1], O_RDONLY)) < 0) {
 8             perror("open");
 9             return -1;
10         }
11     }
12     while((rlen = read(fd, buf, sizeof(buf))) > 0) {
13         if(write(1, buf, rlen) < 0) {
14             perror("write");
15             return -2;
16         }
17     }
18     close(fd);
19     return 0;
20 }
~
                                              2,0-1         All
```

The Header File – libmini.h

It contains …

- Data types
- Constants
- Native system calls
- Syscall wrappers
- Additional functions

See the demo!



The Entry Point – ``start.o"

Reminder: The entry point of a Linux program is the _start function

The universal ***_start*** function

- Prepare argc, argv, and envp
- Call ``main" function
- Read return value from ``main"
- Call ``exit" function

unix@rhino: ~/myprog/unix_prog/asm — 48×13

```
 1 extern main
 2 extern sys_exit
 3
 4         section .text
 5         global _start
 6 _start:
 7         mov     rdi, [rsp]      ; argc
 8         lea     rsi, [rsp+8]    ; argv
 9         call    main
10         mov     rdi, rax        ; exit code
11         call    sys_exit
12         ret
"start.asm" 12L, 172C          3,0-1          All
```

The Entry Point – ``start.o'' (Cont'd)

Ref: System V AMD 64 ABI

- https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf

Figure 3.11: Initial Process Stack

Purpose	Start Address	Length
Unspecified	High Addresses	
Information block, including argument strings, environment strings, auxiliary information ...		varies
Unspecified		
Null auxiliary vector entry		1 eightbyte
Auxiliary vector entries ...		2 eightbytes each
0		eightbyte
Environment pointers ...		1 eightbyte each
0	`8+8*argc+%rsp`	eightbyte
Argument pointers	`8+%rsp`	argc eightbytes
Argument count	`%rsp`	eightbyte
Undefined	Low Addresses	

The Mini C Library

In the CAT program: We need ***open***, ***read***, ***write***, and ***close*** functions

Read the Linux x86_64 System Call Table

Use ***similar*** calling convention to pass function call parameters!

%rax	Syscall	%rdi	%rsi	%rdx	%r10	%r8	%r9
0	sys_read	fd	*buf	count			
1	sys_write	fd	*buf	count			
2	sys_open	*filename	flags	mode			
3	sys_close	fd					

The Mini C Library (Cont'd)

Implement a single syscall

```
        section .text
        global sys_read:function
sys_read:
        push    r10         ; for the 4th parameter
        mov     r10, rcx
        mov     rax, 0
        syscall
        pop     r10
        ret
```

Hints

- The 4th parameter: RCX vs R10
- Create shared object

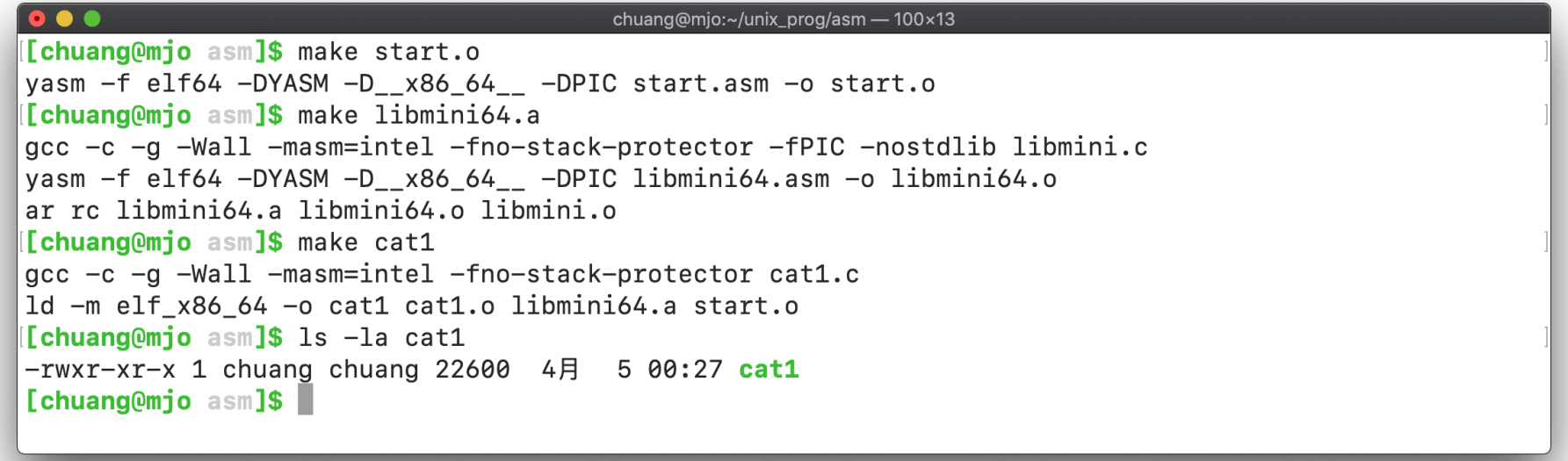
Leverage MACRO feature

chuang@mjo:~/unix_prog/asm — 38×30

```
 1
 2 %macro gensys 2
 3         global sys_%2:function
 4 sys_%2:
 5         push    r10
 6         mov     r10, rcx
 7         mov     rax, %1
 8         syscall
 9         pop     r10
10         ret
11 %endmacro
12
13 ; RDI, RSI, RDX, RCX, R8, R9
14
15 extern  errno
16
17         section .data
18
19         section .text
20
21         gensys   0, read
22         gensys   1, write
23         gensys   2, open
24         gensys   3, close
25         gensys   9, mmap
26         gensys  10, mprotect
27         gensys  11, munmap
28         gensys  22, pipe
29         gensys  32, dup
                   1,0-1          Top
```

Put Everything Together – Link Statically

Everything in libmini64.a are included in the executable …



Put Everything Together – Link Dynamically

Only required functions are linked into the executable …

```
chuang@mjo:~/unix_prog/asm — 100×13
[chuang@mjo asm]$ make start.o
yasm -f elf64 -DYASM -D__x86_64__ -DPIC start.asm -o start.o
[chuang@mjo asm]$ make libmini64.so
gcc -c -g -Wall -masm=intel -fno-stack-protector -fPIC -nostdlib libmini.c
yasm -f elf64 -DYASM -D__x86_64__ -DPIC libmini64.asm -o libmini64.o
ar rc libmini64.a libmini64.o libmini.o
ld -shared libmini64.o libmini.o -o libmini64.so
[chuang@mjo asm]$ make cat1s
gcc -c -g -Wall -masm=intel -fno-stack-protector cat1.c
ld -m elf_x86_64 --dynamic-linker /lib64/ld-linux-x86-64.so.2 -o cat1s start.o cat1.o -L. -lmini64
[chuang@mjo asm]$ ls -la cat1s
-rwxr-xr-x 1 chuang chuang 15704  4月   5 00:29 cat1s
[chuang@mjo asm]$
```

```
chuang@mjo:~/unix_prog/asm — 100×12
[chuang@mjo asm]$ ./cat1s
./cat1s: error while loading shared libraries: libmini64.so: cannot open shared object file: No such
 file or directory
[chuang@mjo asm]$ ldd ./cat1s
        linux-vdso.so.1 (0x00007ffd9a770000)
        libmini64.so => not found
[chuang@mjo asm]$ LD_LIBRARY_PATH=. ./cat1s
lksdjfa;ldj;adjkf
lksdjfa;ldj;adjkf
^C
[chuang@mjo asm]$
```

The Self-Made Mini C Library: Non-System-Calls

Some popular functions are implemented based on system calls

Example

- printf – based on sys_write
- signal – based on sys_rt_sigaction
- **sleep** – based on sys_nanosleep
- **open** – based on sys_open

Case #1 – sleep(): It's Easy!

Prototype for sleep

- **unsigned int sleep(unsigned int s);**

Prototype for nanosleep:

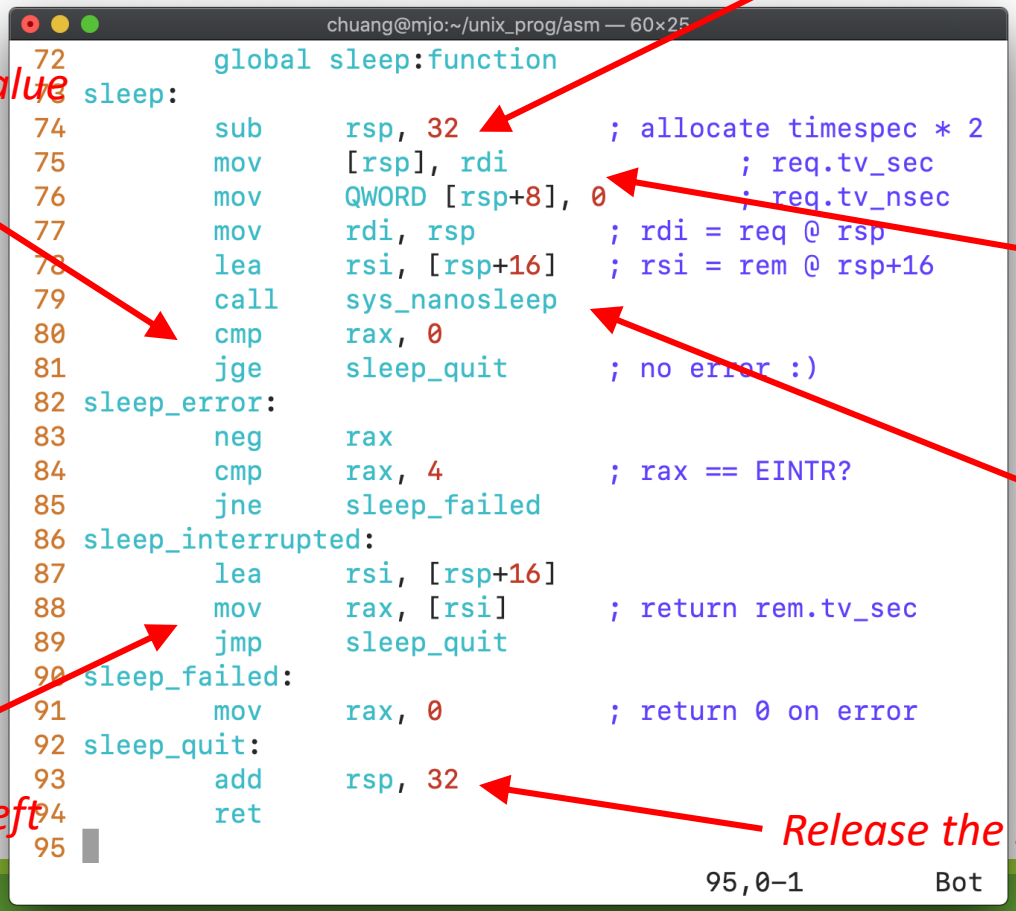
- **long sys_nanosleep(struct timespec *req, struct timespec *rem);**

Implement **sleep()** function in C

```
unsigned int sleep(unsigned int s) {
        long ret;
        struct timespec req = { s, 0 }, rem;
        ret = sys_nanosleep(&req, &rem);
        if(ret >= 0) return ret;
        if(ret == -EINTR) return rem.tv_sec;
        return 0;
}
```

Case #1 – sleep(): Assembly

Implement the same function in Assembly



*Allocate 32 bytes for timespec * 2*

Check return value

req = { s, 0 }

Call sys_nanosleep

Return time left

Release the spaces allocated on the stack

sleep() – Running Example

Sleep & show a message …

chuang@mjo:~/unix_prog/asm — 49×11

```
 1 #include "libmini.h"
 2
 3 int main() {
 4         char s[] = "sleeping for 5s ...\n";
 5         char m[] = "hello, world!\n";
 6         write(1, s, sizeof(s));
 7         sleep(5);
 8         write(1, m, sizeof(m));
 9         return 0;
10 }
                                2,0-1         Top
```

chuang@mjo:~

```
[chuang@mjo asm]$ make testmini_x64
gcc -c -g -Wall -masm=intel -fno-stack-protector testmini.c
yasm -f elf64 -DYASM -D__x86_64__ -DPIC start.asm -o start.o
ld -m elf_x86_64 --dynamic-linker /lib64/ld-linux-x86-64.so.2 -o testmini_x64 testmini.o start.o -L. -lmini64
[chuang@mjo asm]$ ls -la testmini_x64
-rwxr-xr-x 1 chuang chuang 15360  4月   5 00:45 testmini_x64
[chuang@mjo asm]$ LD_LIBRARY_PATH=. ./testmini_x64
sleeping for 5s ...
hello, world!
[chuang@mjo asm]$
```

Case #2 – open()

It's a little bit complicated compared to sleep() …

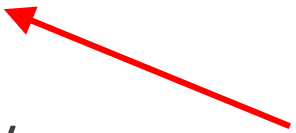
- Variable length parameters
  int open(const char *pathname, int flags);
  int open(const char *pathname, int flags, mode_t mode);
- Handle **errno** properly

It might be easier to implement in Assembly

- Don't care how many parameters are passed
  – Parameters are read from registers/stacks

PIC (position independent code) is required

- **errno** is declared in a C file
  – Need a special trick to locate the variable



3rd argument: Required for O_CREAT and O_TMPFILE

Case #2 – open(): Assembly

chuang@mjo:~/unix_prog/asm — 52×18

```
55          global open:function
56  open:
57          call    sys_open
58          cmp     rax, 0
59          jge     open_success    ; no error :)
60  open_error:
61          neg     rax
62          mov     rdi, [rel errno wrt ..gotpcrel]
63          mov     [rdi], rax      ; errno = -rax
64          mov     rax, -1
65          jmp     open_quit
66  open_success:
67          mov     rdi, [rel errno wrt ..gotpcrel]
68          mov     QWORD [rdi], 0  ; errno = 0
69  open_quit:
70          ret
71
                                    71,0-1        69%
```

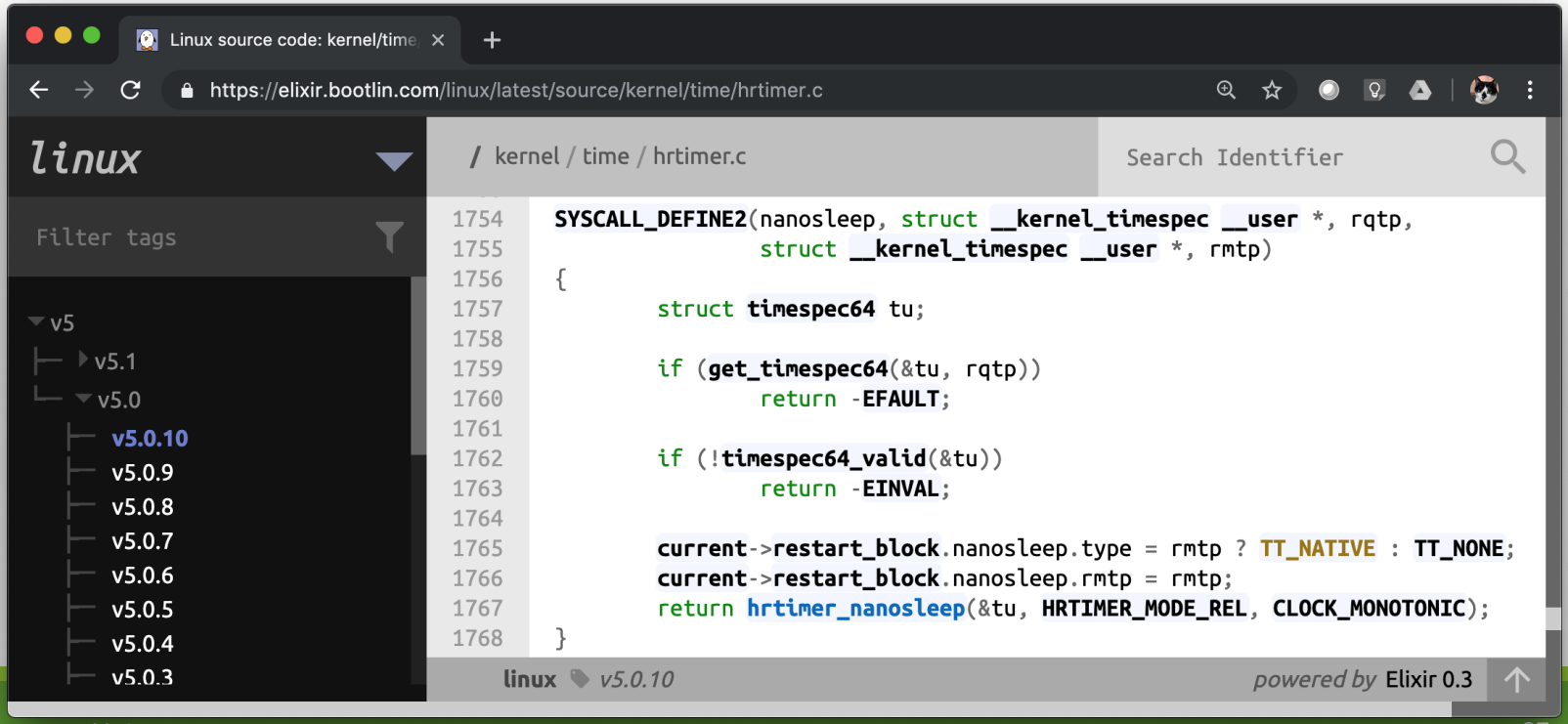
Two or Three parameters, call sys_open directly!

Set errno value RAX is negated, will explain later …

System Call Return Value

System calls always return a long integer (x86_64)

- It is the returned result (usually casted to the correct data type)
- Or an errno code (if it is less than zero)



Q & A