Mo Tu We Th Fr Sa Su

Memo No.
Date 13 / 11 / 2024

Assembly Crash Course

Assembly

→ Made in late 1940's early 50's

Sentence → instruction
Verb → operation
Noun → Operand

Architectures: x86 → on most personal computers
arm → phones
ppc
mips
risc v
pdp-11

Registers

→ Typically same size as the CPU architecture word width of

movsx → move sign extend, preserving 2's complement

Mo Tu We Th Fr Sa Su

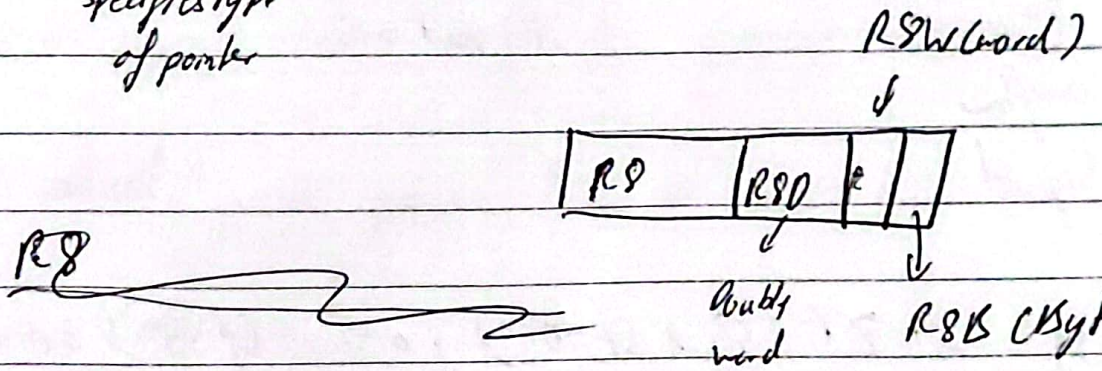
Memo No.
Date

- When you push onto the stack the value of the top pointer decreases

- You can get calculated address with leq
(Load Effective Address)

mov DWORD PTR [rax], 0x1337

DWORD → specifies type of pointer



Warning:
Oper^n on 32bit regs in 64-bit architecture causes the top 64bit register to be zeroes. (wipes the top)
Does it happen with AX or AL

Control flow
-> jmp -> helps skip instruc^n, (jumps X bytes)
- There are cond^n al jumps
je -> jump if equal, jnz -> if not zero

Mo Tu We Th Fr Sa Su

Memo No.
Date / /

eb 04 — jump 4 bytes

eb fe ? — infinite loop

→ Conditional jumps checks conditions stored in "flags" register

Main conditional flags:

CF Carry

ZF Zero

OF Overflow

SF Signed

ex jnz checks if last operation is not zero

jg → ZF=0 & SF=0 | ja → CF=0 & ZF=0

jg — jump greater

ja — jump above

Function Calls

call — saves next instruction after call

ret — returns to that instruction

Mo Tu We Th Fr Sa Su

Memo No.
Date / /

Callee & Caller func^n's

- Registers are shared between func^n's, so calling conventions should agree on what registers are protected

In Linux amd64, when you call a func^n, it will give back rbx, rbp, r12, r13, r14, r15 in the same way it found it. (callee saved).

Caller saved → you push it and save it

System Calls

is instruction that makes a call into the OS

'syscall' triggers the system call specified by the value in rax

eg:

```
:
move rax, 0
syscall
```

system call for read

or

```
:
mov rax, 1
syscall
```

for write

60 → exit

PTO →→→

Mo Tu We Th Fr Sa Su

RSA

Memo No.
Date 09 / 12 / 2024

$n = pq$, n -> the public key, available to anyone

p, q -> two primes

-> We use two primes because from n, factorizing the number (primes in this case) is difficult

Now we have an 'e' which is co-prime to $(p-1)(q-1)$

- The public key (N, e) is known to everyone and they can send me an encrypted message 'c'.

where $\boxed{c = m^e \pmod n}$, where m is the message

Note:
-> Primes are coprime to each other

To decrypt this we use a private key 'd'.

$$\boxed{m = c^d \pmod n}$$

where d is a no. such that

$$de = 1 \pmod{(p-1)(q-1)}$$

$$d = e^{-1} \pmod{(p-1)(q-1)}$$

- You can only decrypt the message with d, to get that

Mo Tu We Th Fr Sa Su

Memo No.
Date / /

we need to know p & q, which is really hard when using large primes.

Euler's Totient [$\phi(n)$]

$\phi(n)$: no. of coprimes on $[1, n]$

Eg: $\phi(6) = 2$

$6 = \{1, 2, 3, 4, 5, 6\}$

✓ ✗ ✗ ✗ ✓ (2, 3, 2)

$6 \rightarrow 2, 3$

- For a prime no. 'p', $\phi(p) = p - 1$ as $\{\underbrace{1, 2, \ldots}_{\text{coprime to } p}, p\}$

- $\phi(p^a) = p^{a-1}(p-1)$

∵ p is prime, prime factors of $p^a = p \cdot p \cdot p \ldots$

all the nos that share a factor with p^a are the multiples of $p \rightarrow$ i.e. $p, 2p, 3p, \ldots, p^a$

$\Rightarrow p^{a-1}(p)$

So coprimes will be $= p^a - p^{a-1} = p^{a-1}(p-1)$

↑ nos that share factor with p^a

Mo Tu We Th Fr Sa Su

Memo No. _____
Date / /

Assembly continued

Debugging

is done with debuggers such as GDB

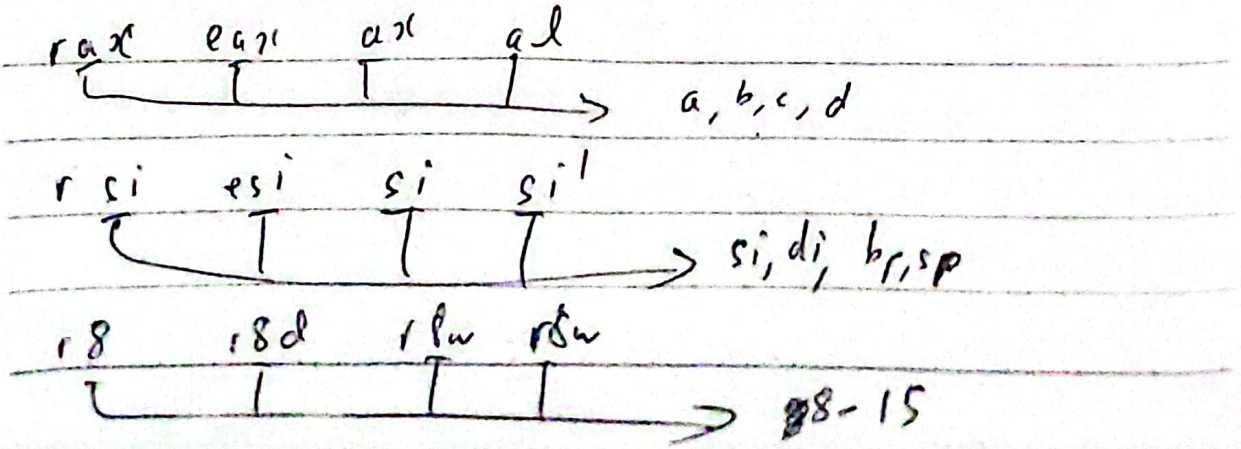
- gdb helps you add a breakpoint, interrupt a program & let you look around

'int 3' → breakpoint instruction

Resources

GDB → go to debugging
strace → helps figure out how the program is interacting with OS
Rappel → helps explore effects of instructions

General Purpose Registers (GPRs)



Mo Tu We Th Fr Sa Su

Memo No.
Date

rsp → used to point to current top of stack
rbp → base pointer during func^{n} calls
rip → points to next instruc^{n} to be executed

Flag Register
rflag → CF, PF, AF, -- (given earlier)

• BSS holds uninitialized data

; → comments anything after it

Defining Consts
<name> equ <value>

Data Section
initialized data must be declared in the "section .data"
→ all initialized variables & constants are placed here.
→ Variable names must start with letter

Supported datatypes:	db	dw	dd	dq	ddq	dt
No. of bits:	8	16	32	64	128 (int)	128 (float)

Mo Tu We Th Fr Sa Su

Memo No. _____
Date / /

BSS Section

→ Uninitialized data is declared in "section .bss" section.

<variable name> <resType> <count>

Supported data types: resb resw resd resq resdq
No. of bits: 8 16 32 64 128

Text Section

→ The code is placed in the "section .text"
→ one instruction per line
→ Contains some headers / labels that define initial program entry point.
eg: ~~global start~~ global _start
_start:

Tool Chain

Consists of: Assembler
Linker
Loader
Debugger

Human readable → (Assembler) → object file → (Linker) → Executable file → (Loader) → load memory