Monday, 6 January 2025 8:19 AM

linkers

loaders

program $\xrightarrow[\text{initialise}]{\text{load in MM}}$ process

stack, PC etc

DLL ⇝ linked libraries (not always)

↙ STATIC ↘ DYNAMIC ↗

Loaders → not explicitly defined

1) Input should be syntactically & semantically correct

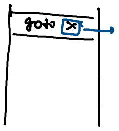
2) Input must be parsed correctly

Why is GO-TO /JUMP/BRANCH not recommended ?

↳ goto $\boxed{x}$ ← PC

i) cache miss is possible

ii) execution becomes SLOW



need to goto x and find it's address and replace in executable

Q How many bytes is a cache ?

need to optimise loops (same reason)

macro

pre-processing directories ?

function

How are macro and function are different wrt COMPILER ?

ARM processor manufacturer ?

When did C language come into EXISTENCE ??? Hint - 1989

assembler converts assembly code to object code

linker generates .exe

Difference b/w CPU cycle and machine cycle and instruction cycle ?

64 bits (nowadays) in 1 machine cycle

↳ how much in 1 cycle ?

64 /128 /256 /512 ?

(data size = 64)

02/01/25

Q 1. If you run an executable program where

i) stack is not defined

which stack will be used to initialize PC?

Ans → user program runs at root if kernel stack is to be used → DANGEROUS

SCRATCHPAD (Read-write op's → required (Execute x))

Learn MAKEFILE →

Architecture → How many ALUs? Direct Access / Indirect Access ?

Organisation → Implementation of architecture

Preprocessor → #import statements

→ removes comments and extra spaces and everything required (modules) is stored in .i file

objdump → disassemble .s file

ld → dynamic linking → shared among multiple programs.

gcc → static linking → all import modules are used by one program

elf 32 → type of ISA (32-bit)

dynamic linker & absolute loader ⇒ assembly lang. is NOT PORTABLE.

why is dynamic linking is used in present OS ?

add w_1 w_2 $w_1 \leftarrow w_1 + w_2$ (2-address)

add w_1 w_2 w_3 $w_1 \leftarrow w_2 + w_3$ (3-address)

} why 2 and 3 ?

NOP operation ?

↓

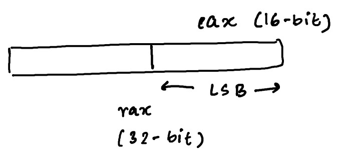
no operation → why is it used?

emulation → firmware ISA ($ISA_1 \rightarrow ISA_2$)

arm intel

cross-assembly options

rax → 64-bit accumulator



Q → why do we use general purpose registers ?

RBP → base addressing mode

Q → why does stack grow downwards? (easier to check if stack is full)

Stack section ↑

BSS → resb → reserved bytes

TEXT → dynamic / static linker → will program start from main?

interrupts → interrupt service route

↓

used by printf, scanf, echo

i) software interrupts — OS defined

ii) hardware interrupts — ROM available

Linux → only 1 interrupt routine is used

multiple parameter parsing → CPU registers

ecx → used as counter

Debugging Assembly

gdb <filename>

layout asm

break _start

run

stepi → run one at a time

info registers eax

system calls

- eax = 0 → default value used by OS → NOT ALLOWED
- eax = 1 → exit with no error
- eax = -1 → exit with error
- eax = 3 → read
- eax = 4 → write

By default, 32 bit registers (eax, ebx, ...) are initialised as 0.

If we use a PART of them, rest of the part takes garbage value.

_start or _main ?

global specifies that codespaces are different.

can define global c programs → .inc → use extern

lea → load effective address ⇒ dereferencing (*var)

mov

/temp files – ?

swap space ? (paging space on hard disk)

how is swap space different from filesystem ?

SCRATCHPAD

#include <stdio.h> → linking external programs containing printf / scanf

call puts → put a stream to display

xor is faster than mov

assembler directives (do not have opcodes)

What are constants ? (macro or function)

Before compiling, macros are replaced

object code has RELATIVE ADDRESSES

when loader loads the program in memory absolute addresses are assigned.

Header
Text Section
Data Section
Relocation Info
Symbol Table
Debugging Info

OBJECT CODE

Symbols and literals (constants) are mapped into corr. object codes by assembler.

Loader is part of the OS.

Why do we need 2-pass assembler

↪ In 1-pass, we need to backtrack after finding location of label each time

Assembler always assume its a macro assembler

↓

substituted before computing PASS 1.

LTORG → assign address to literals only when you encounter this

1. Symbol Table
2. Literal Table
3. Pool Table

} pass I

4. Opcode Table → ISA (already available)

H/W

```
START  200
LOOP   MOVER  AREG, NUM1
       ADD    AREG , NUM2
       SUB    AREG ,  =2
       MOVEM  AREG , RESULT
       JMP    LOOP
NUM1   DC   5
NUM2   DC   10
RESULT DS   1
  END
```

DLLs are in the OS → commonly shared across all users

Scratchpad → stack defined (by default) by OS.

How can we differentiate b/w

i) hardware interrupt
ii) software interrupt
iii) functions

] How do these take place?

EXEC → different level of returning?

1) Mnemonic Opcode Table (MOT)
2) Pseudo Opcode Table (POT) → only needed by assembler

PROG

Assembly directive (POT)

↓ ↗ opcode

START 100 : (AD,01) — (C,100)

↓ constant

MOVER AREG, A 100 : (IS,01) 01 (S,01)

★ I.S → Imperative statement (How)

DL → Declaritive statement (what)

LOOP : PRINT B : (IS,09) — (S,03)

↓

SYM - TABLE

Sym. No.	Symbol	Address
01	A	107
02	LOOP	101
03	B	
04	D	
05	LABEL	101

LIT TABLE

Lit no.	Literal	Address
01	='9'	105

02	='23'	106

label
↓
fill entry in symbol table
and skip

ADD BREG, = '9'
↓
(IS,03) 02 (L,01)

Note → Assuming all instructions are of the same size

SUB BREG, D
↓
(IS,04) 02 (S,04)

COMP CREG, ='23'

(IS,08) 03 (L,02)

LTORG → assign addresses to literals

(107) A DS 3 → reserves 3 memory locations for A starting at 107
↓
(S,01) (DL,01) _ 03

LABEL : EQU LOOP → skip (already addressed LOOP)

ORIGIN 500 → no need to write object code for ORIGIN

assembler directives are NOT REQUIRED to be converted into machine code

ldd /bin/ls → path to ls command
↓
lists dynamic dependencies

regular expressions are used for searching for particular files in directory using ls.