

Tool Chain

A set of system programs used to create a program

- Editor
- Compiler / assembler
- Linker
- Loader
- Debugger

Assembler

- Reads assembly source file (.s)
- Processes directives and macros
- Translates instructions to machine code stored in object file (.obj) [binary format]
- Optionally generates listing file (.lst)

Linker

- Combines several object files and produces single executable file

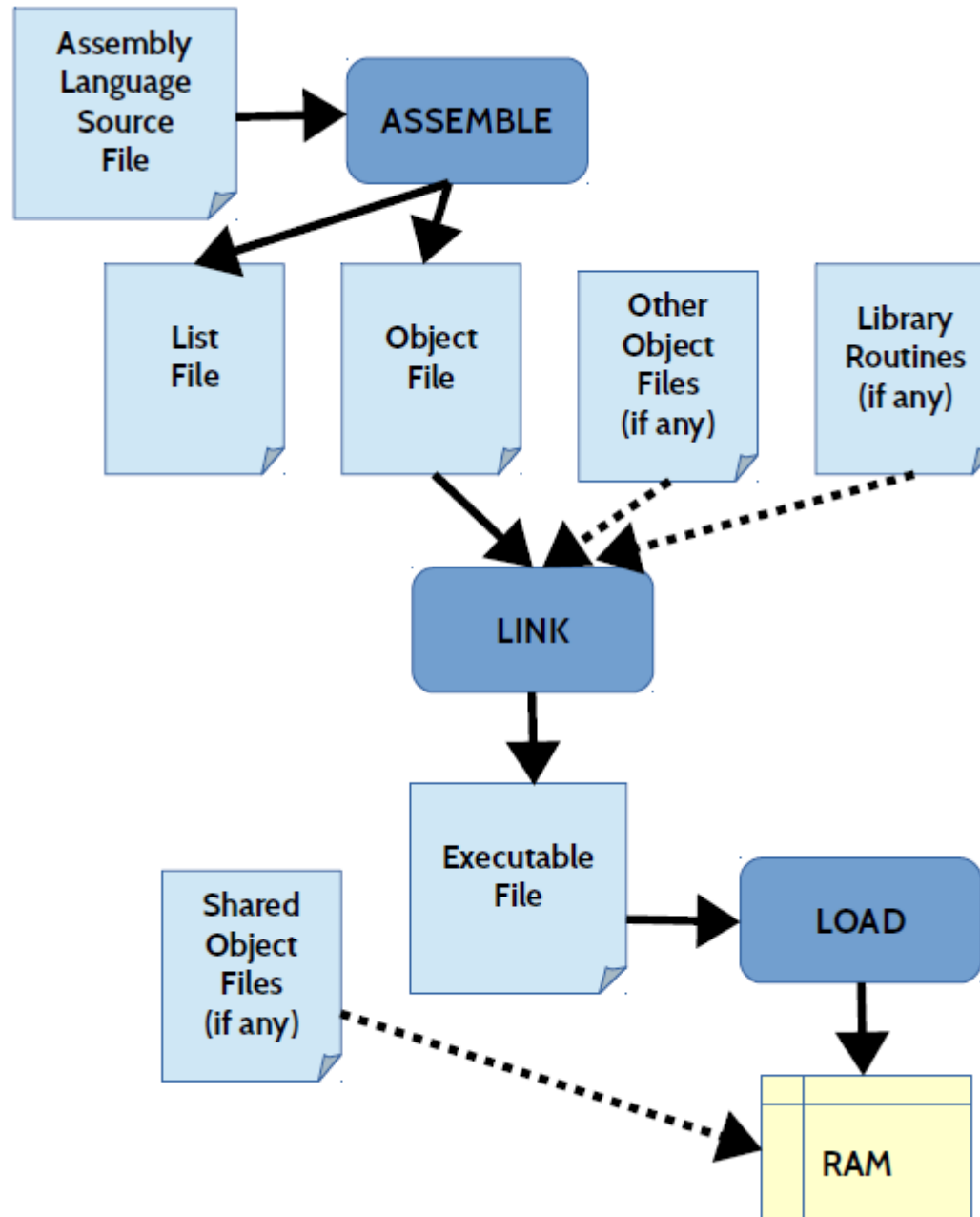
Loader

- A part of OS that loads executable file into memory and run it

Debugger

- A program that used to monitor/control execution of programs

Tool Chain



Tool Chain

In this course, we will be using ***nasm*** and ***yasm*** as an assembler and ***DDD*** as a debugger

NASM: Net-wide assembler, widely used assembler for Linux

YASM: newer and claimed to be better over nasm

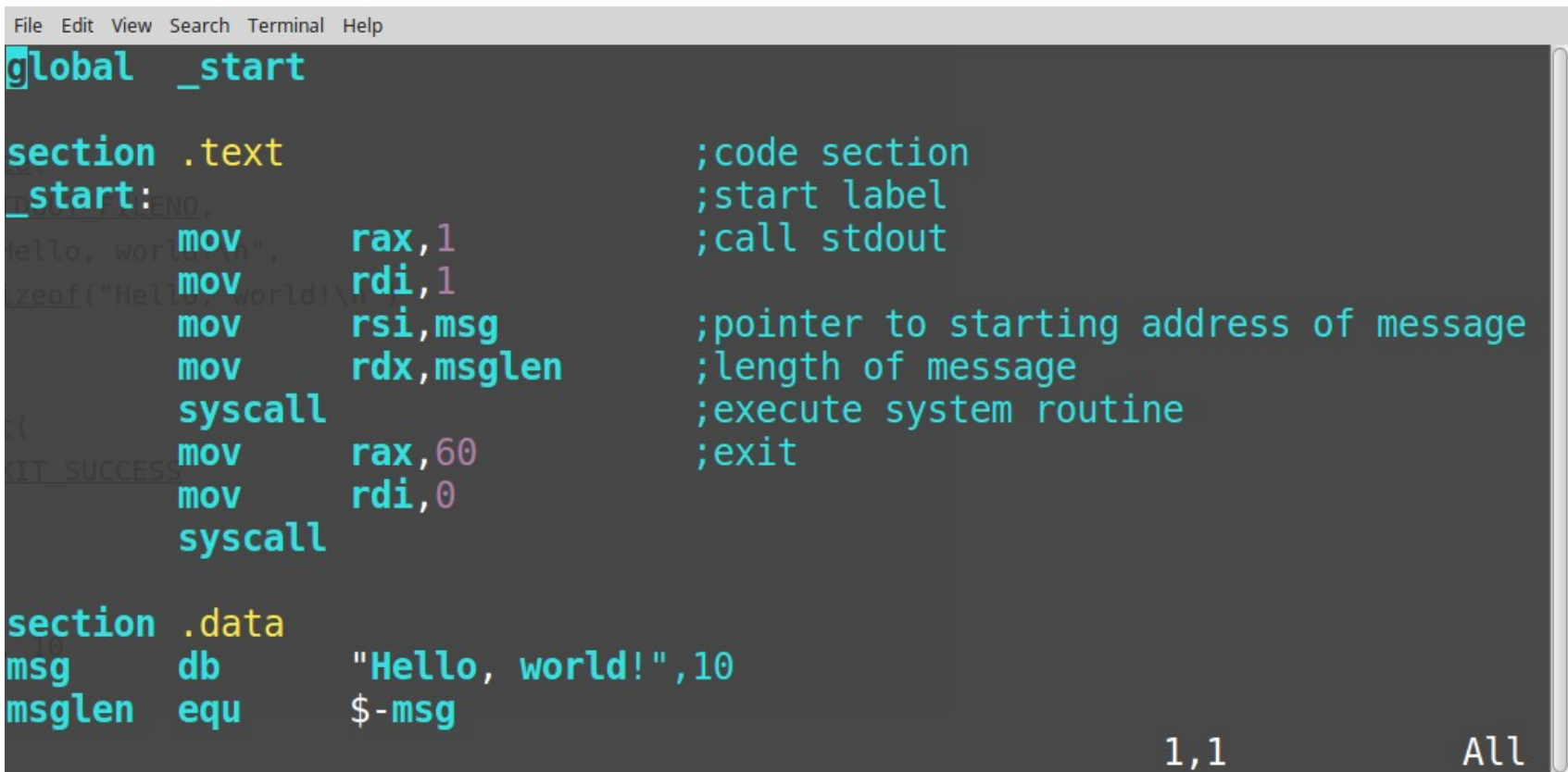
DDD: graphical user-interface frontend for ***gdb***

Assemble Command

Steps:

- Edit and save source file (.s)
- Use the following command

```
yasm -f elf64 -o hello.o hello.s  
ld -o hello hello.o  
./hello
```



```
File Edit View Search Terminal Help  
global _start  
  
section .text ;code section  
_start: ;start label  
    mov rax,1 ;call stdout  
    mov rdi,1  
    mov rsi,msg ;pointer to starting address of message  
    mov rdx,msglen ;length of message  
    syscall ;execute system routine  
    mov rax,60 ;exit  
    mov rdi,0  
    syscall  
  
section .data  
msg db "Hello, world!",10  
msglen equ $-msg
```

1,1 All

Assemble Command

Steps:

- **-f** specifies format : **elf64** = linux 64-bit binary format
- **-o** specifies object file output
- **man yasm** to view help

Generate listing file:

- -l specifies listing file name

```
yasm -f elf64 -o hello.o -I hello.lst hello.s
```

```

File Edit View Search Terminal Help
4 [section .text]
5 _start:
6 00000000 48C7C001000000 mov rax,1
7 00000007 48C7C701000000 mov rdi,1
8 0000000E 48C7C6[00000000] mov rsi,msg
9 00000015 48BA0E000000000000- mov rdx,msglen
10 00000015 00
11 0000001F 0F05 syscall
12 00000021 48C7C03C000000 mov rax,60
13 00000028 48C7C700000000 mov rdi,0
14 0000002F 0F05 syscall
15
16 [section .data]
17 00000000 48656C6C6F2C20776F- msg db "Hello, world!",10
18 00000000 726C64210A
19 msglen equ $-msg
(END)
```

Assembly for Debugging

Steps:

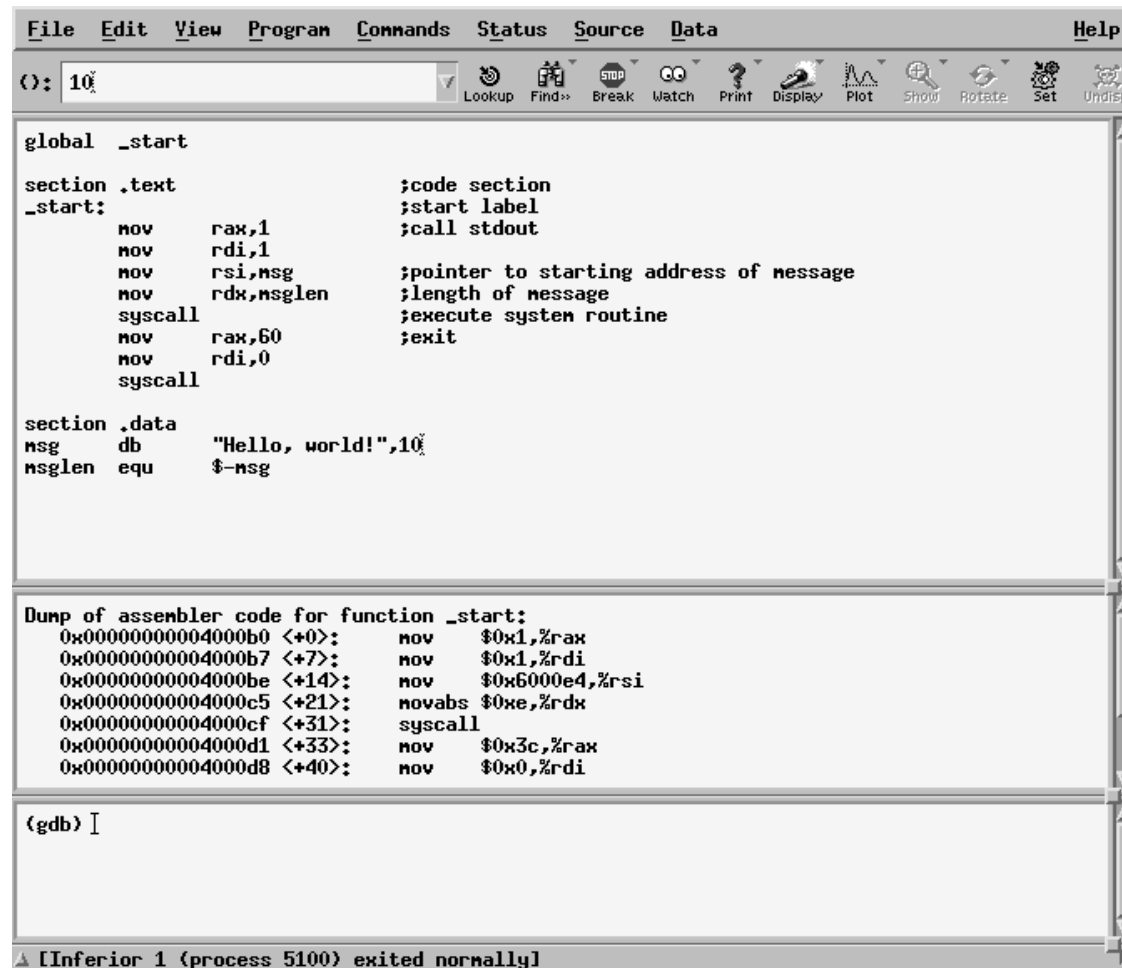
- `-g dwarf2` specifies debugging format to be added to object file

```
yasm -f elf64 -g dwarf2 -o hello.o hello.s
```

```
ld -o hello hello.o
```

```
./hello
```

```
ddd ./hello &
```



The screenshot shows a debugger window with a menu bar (File, Edit, View, Program, Commands, Status, Source, Data, Help) and a toolbar with icons for Lookup, Find, Break, Watch, Print, Display, Plot, Show, Rotate, Set, and Undis. The main window displays assembly code for a function named `_start`. The code is organized into sections: `.text` (code section) and `.data` (data section). The `.text` section starts with `_start:` and contains instructions for setting up registers, calling `stdout`, and executing a system routine. The `.data` section contains a message string `msg` and its length `msglen`. Below the assembly code, a dump of the assembler code for the function `_start` is shown, listing memory addresses and the corresponding instructions. The debugger prompt `(gdb) [` is visible at the bottom.

```
global _start

section .text                ;code section
_start:                      ;start label
    mov     rax,1             ;call stdout
    mov     rdi,1
    mov     rsi,msg           ;pointer to starting address of message
    mov     rdx,msglen        ;length of message
    syscall                  ;execute system routine
    mov     rax,60            ;exit
    mov     rdi,0
    syscall

section .data
msg     db      "Hello, world!",10
msglen  equ     $-msg

Dump of assembler code for function _start:
0x0000000004000b0 <+0>:      mov     $0x1,%rax
0x0000000004000b7 <+7>:      mov     $0x1,%rdi
0x0000000004000be <+14>:     mov     $0x6000e4,%rsi
0x0000000004000c5 <+21>:     movabs  $0xe,%rdx
0x0000000004000cf <+31>:     syscall
0x0000000004000d1 <+33>:     mov     $0x3c,%rax
0x0000000004000d8 <+40>:     mov     $0x0,%rdi

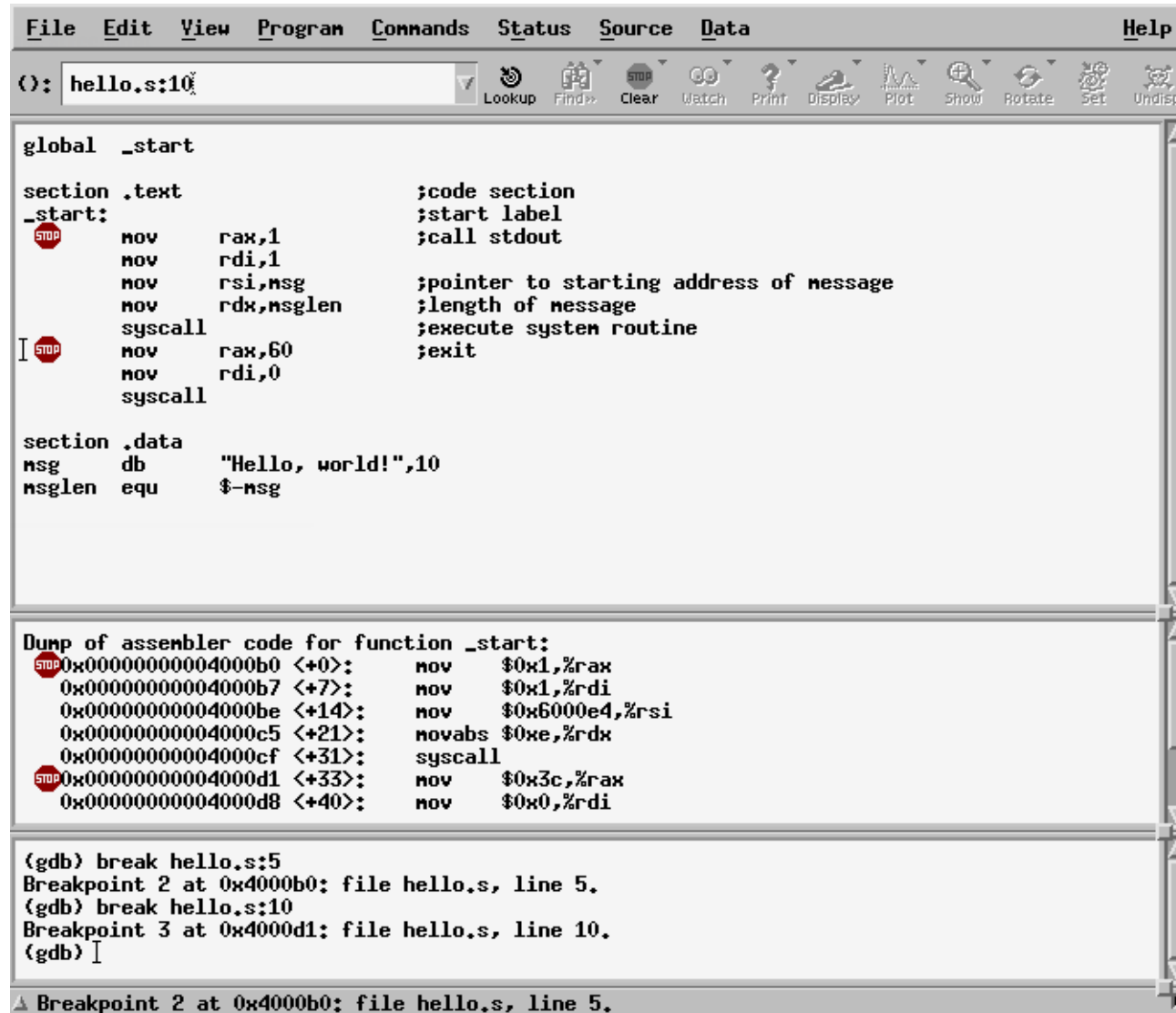
(gdb) [
```

▲ [Inferior 1 (process 5100) exited normally]

Debugging

Adding break points:

- Break points allow program execution to be stopped at specified instructions
- Right click on a line you want program to be stopped
- Run a program
- Step execution can be used



The screenshot shows a debugger window with a menu bar (File, Edit, View, Program, Commands, Status, Source, Data, Help) and a toolbar with icons for Lookup, Find, Clear, Watch, Print, Display, Plot, Show, Rotate, Set, and Undis. The main pane displays assembly code for a program named 'hello.s'. The code is organized into sections: .text (code) and .data (data). The .text section contains the _start function, which initializes registers, calls stdout, prints a message, and exits. The .data section contains a message string. The code is as follows:

```
global _start

section .text
_start:
    mov     rax,1
    mov     rdi,1
    mov     rsi,msg
    mov     rdx,msglen
    syscall
    mov     rax,60
    mov     rdi,0
    syscall

section .data
msg     db     "Hello, world!",10
msglen  equ     $-msg
```

Below the assembly code, a dump of the assembler code for the _start function is shown, with memory addresses and instructions:

```
Dump of assembler code for function _start:
0x0000000004000b0 <+0>:    mov     $0x1,%rax
0x0000000004000b7 <+7>:    mov     $0x1,%rdi
0x0000000004000be <+14>:   mov     $0x6000e4,%rsi
0x0000000004000c5 <+21>:   movabs  $0xe,%rdx
0x0000000004000cf <+31>:   syscall
0x0000000004000d1 <+33>:   mov     $0x3c,%rax
0x0000000004000d8 <+40>:   mov     $0x0,%rdi
```

At the bottom, the GDB command prompt shows the following commands and their output:

```
(gdb) break hello.s:5
Breakpoint 2 at 0x4000b0: file hello.s, line 5.
(gdb) break hello.s:10
Breakpoint 3 at 0x4000d1: file hello.s, line 10.
(gdb) [
```

The status bar at the bottom indicates: Breakpoint 2 at 0x4000b0: file hello.s, line 5.

Viewing registers

Registers can be viewed during program execution

- Program must be in step execution mode
- Set break point(s)
- Run the program
- Go to menu Status → Registers

