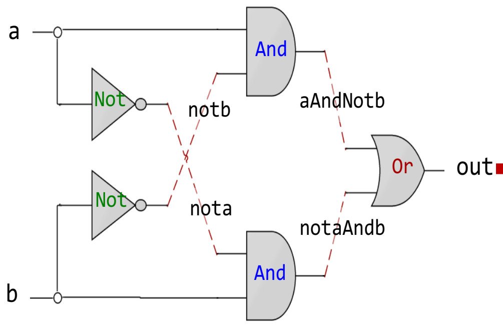
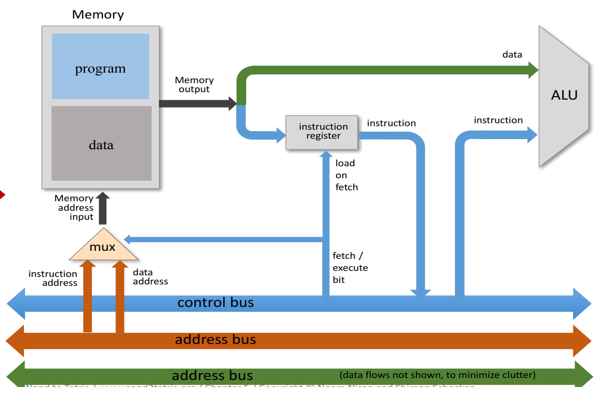
Computer Organization & Assembly Language Programming



```
CHIP Xor {
  IN a, b;
  OUT out;
  PARTS:
  Not(in=a, out=nota);
  Not(in=b, out=notb);
  And(a=nota, b=b, out=w1);
  And(a=a, b=notb, out=w2);
  Or(a=w1, b=w2, out=out);
}
```



Lecture # 31

Debugging C Programs with GDB

```
#include<stdio.h>
#include<stdlib.h>
int main(){
  printf("Learning is fun with Arif\n");
  exit(0);
}
```

```
global main
SECTION .data
   msg: db "Learning is fun with Arif", 0Ah, 0h
   len_msg: equ $ - msg
SECTION .text
   main:
      mov rax,1
      mov rdi,1
      mov rsi,msg
      mov rdx,len_msg
      syscall
      mov rax,60
      mov rdi,0
      syscall
```

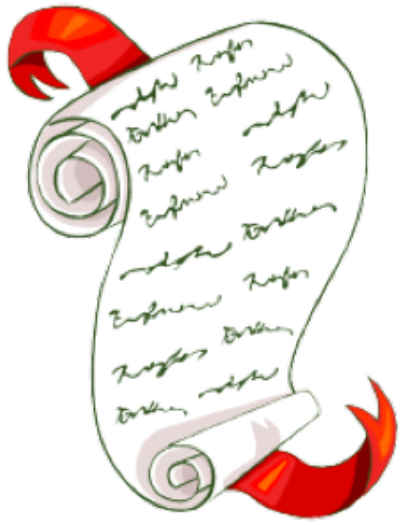
```
0:  b8 01 00 00 00
5:  bf 01 00 00 00
a:  48 be 00 00 00 00 00
11: 00 00 00
14: ba 1b 00 00 00
19: 0f 05
1b: b8 3c 00 00 00
20: bf 00 00 00 00
25: 0f 05
```

For resources visit my personal website:
https://www.arifbutt.me
and course bitbucket repository:
https://bitbucket.org/arifpucit/coal-repo

Instructor: Muhammad Arif Butt, Ph.D.

Today's Agenda

- Review of C compilation process
- What is debugging and what is a debugger?
- Compiling, loading & running a program in GDB
- Getting information about running process
- **Demo**
- Getting help and listing your source
- Setting breakpoints / watchpoints
- Stepping through your code
- Examining / Modifying variables
- Convenience variables
- Setting conditional breakpoints
- **Demo**
- gdb Text User Interface
- **Demo**



https://sourceware.org/gdb/onlinedocs/gdb/index.html#SEC_Contents

Review of C-Compilation Process

Source Code File(s)
`hello.c`

```
$ gcc –E hello.c 1> hello.i
```

Preprocessor (cpp)
Interpret preprocessor directives, Include header files, Expand macros, Remove comments

Processed Code File(s)
`hello.i`

```
$ gcc –S hello.i
```

Compiler (cc)
Checks for syntax errors, Converts source code to assembly code of underlying processor

Assembly Code File(s)
`hello.s`

```
$ gcc –c hello.s
```

Assembler (as/yasm)
Generates relocatable object files to be used by linker, Contains symbol table

Object Code File(s)
`hello.o`

Static Library (.a)

```
$gcc --static –c hello.o -lc
```

Linker (ld)
Static vs Dynamic linking, Contains code and data for all functions defined in src files, Contains global symbol table

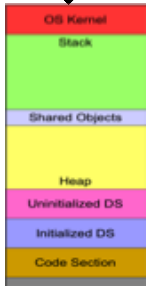
Executable File
`a.out`

Stored on hard disk in ELF format

Dynamic Library (.so)

```
$ ./a.out
```

Loader



Logical Process Address Space in memory

```
$ gcc –save-temps hello.c
```

What is Debugging?

- Debugging is the science and art of finding and eliminating bugs in a computer program
- A debugger is a program running another program
- Using a debugger, a programmer can
  - Start a program, specifying anything that might affect its behavior
  - Make a program stop on specified conditions
  - Examine what has happened, when a program has stopped
  - Change things in a program, so you can experiment with correcting the effects of one bug and go on to learn about another
- Some Famous Debuggers:
  - GDB (PEDA plug-in)
  - Radare2
  - Intel Debugger
  - SoftIce
  - Immunity Debugger
  - Strace

What is GDB?

- GDB, the GNU Project debugger, allows you to see what is going on inside another program while it executes -- or what another program was doing at the moment it crashed
- The programs being debugged might be executing on the same machine as GDB (native), on another machine (remote), or on a simulator
- GDB is a portable debugger that can run on most popular UNIX and Microsoft Windows variants, as well as on Mac OS X
- The target processors include IA-32, x86-64, alpha, arm, mips, powerpc, sparc and many others
- GDB works for many programming languages including Assembly, C/C++, Objective C, OpenCL, Go, Modula-2, Fortran, Pascal and Ada
- Programmers use GDB for following tasks:
  - Run time analysis of binaries
  - Manipulating program flow
  - Disassembly
  - Reverse Engineering and cracking binaries

Note: GDB is too good a debugger, however, it lacks intuitive interface, do not have a smart context display, do not have commands for exploit development, and has weak scripting support. To use GDB for exploit development one can use PEDA plugin (Python Exploit Development Assistance)

GUI Interfaces of GDB

- Although for learning purpose we will be using command line interface of gdb, however there are many GUI interfaces of GDB like:
  - Data Display Debugger
  - Nemiver
- Following IDEs also use gdb at their back end:
  - Xcode
  - Visual Studio
  - Code::Blocks
  - Dev C++
  - Eclipse
  - NetBeans

Compiling, Loading and Running Program with GDB

- In order to load and properly analyze a program in `gdb` you need to compile it with `–g` or `–ggdb` option. This is done to instruct the compiler to keep debugging symbols in the object files

```
$ gcc --ggdb  -c myadd.c mysub.c driver.c
$ gcc myadd.o mysub.o driver.o –o myexe
```

- There are two ways to load a program `myexe` in `gdb` :

```
$ gdb
(gdb) file myexe        }  $ gdb myexe
```

- Once `gdb` is running and has a program loaded, you can set command line arguments:

```
(gdb)set args arg1 arg2 …
```

- Once `gdb` is running and has a program loaded, you can run the program:

```
(gdb)run [arg1 arg2 …]
```

- To load an already running program in `gdb` (need to be `sudo`) :

```
$ sudo gdb attach <pid>      OR        (gdb)attach <pid>
```

Getting Info about the Running Process

- Once a program *loaded* inside `gdb`, you can use the following command to display the name of all the source files from which symbols have been read in

```
(gdb)info sources
```

- Once a program *loaded* inside `gdb`, you can use the following command to display the name of all the functions (user defined & library functions)

```
(gdb)info functions
```

- Once a program *loaded* inside `gdb`, you can use the following command to display the name of all the global variables

```
(gdb)info variables
```

- Once a program *loaded* inside `gdb`, you can use the following command to display the name of all the local variables inside a specific function

```
(gdb)info scope function-name
```

- Once a program *running* inside `gdb`, you can use the following command to display the name of all the local variables inside the active frame

```
(gdb)info locals
```

Demo

A Hello World with GDB

31/ex1/

Instructor: Muhammad Arif Butt, Ph.D.

Getting Help in GDB

- To get the listing of twelve different classes in which `gdb` commands are categorized, give the following command

```
(gdb) help
```

```
List of classes of commands:
aliases -- Aliases of other commands.
breakpoints -- Making program stop at certain points.
data -- Examining data.
files -- Specifying and examining files.
internals -- Maintenance commands.
obscure -- Obscure features.
running -- Running the program.
stack -- Examining the stack.
status -- Status inquiries.
support -- Support facilities.
tracepoints -- Tracing of program execution without stopping the program.
user-defined -- User-defined commands.
```

- To get list of commands inside a class: `(gdb)help <classname>`
- To get detailed help about a specific command: `(gdb)help <command>`

Listing Your Source Code inside GDB

- The `list` command of `gdb` is used to display the source code (provided if the source file is there in the `pwd`)
- Following command will display 10 lines after or around previous listing
  - `(gdb) list`
- Following command will display 10 lines around the given line number
  - `(gdb) list file:<line#>`
- Following command will display lines between the two line numbers (both inclusive)
  - `(gdb) list file:<line#> , file:<line#>`
- Following command will display lines around the given function name
  - `(gdb) list file:<function_name>`
- To change the default display of 10 lines, use following command
  - `(gdb) set listsize <newssize>`

Setting Break Points

- Breakpoint is the place (LOC) in your program where you want to stop the execution of your program. Once a breakpoint is hit during execution of a program, you can inspect/modify contents of variables, CPU registers as well as different memory addresses. You can set as many break points as you feel like
- You can set a break point mentioning line#, function name, or by virtual address
  - `(gdb) break prog1.c:line#`
  - `(gdb) break prog1.c:function_name`
  - `(gdb) break prog1.c:*0x2ff5bbcc2100`
- You can get information about existing break points, already set in your program
  - `(gdb) info break`

```
Num     Type           Disp  Enb   Address               What
1       breakpoint     keep  y     0x0000000000001154    in main at factorial.c:5
2       breakpoint     keep  y     0x0000000000001184    in main at factorial.c:8
```

- You can enable/disable/delete/clear break points, already set in your program
  - `(gdb) disable <breakpoint#>`
  - `(gdb) enable  <breakpoint#>`
  - `(gdb) delete  <breakpoint#>`
  - `(gdb) clear   <breakpoint#>`

Setting Watch Points

- A breakpoint stops the execution of a program at a specific location; watchpoint acts on variables
- Following gdb command will set a watch on variable named var-name; whenever the value of this variable will change, gdb will interrupt the program and print out the old and the new value
  - `(gdb) watch <var-name>`
- You can get information about existing break points, already set in your program
  - `(gdb) info watch`

```
Num     Type            Disp  Enb  Address          What
2       hw watchpoint   keep  y                     var-name
```

- You can enable/disable/delete/clear watchpoints, as you can do with breakpoints
  - `(gdb) disable <watchpoint#>`
  - `(gdb) disable <watchpoint#>`
  - `(gdb) enable  <watchpoint#>`
  - `(gdb) delete  <watchpoint#>`
  - `(gdb) clear   <watchpoint#>`

Stepping Through Your Code

Once a breakpoint is hit, you can do either of the following:

- Continue execution till next breakpoint or end of program
  - ➢ `(gdb) continue / c / ci`
- Execute and move to next instruction, but don't dive into functions
  - ➢ `(gdb) next / n / ni`
- Execute and move to next instruction by diving into functions
  - ➢ `(gdb) step / s / si`
- Continue until the current function returns
  - ➢ `(gdb) finish`

Displaying and Modifying Values of Variables

Once a breakpoint is hit during execution of a program, you can inspect/modify contents of variables, CPU registers as well as different memory addresses

- The **`print`** command is the most common command to check the contents of variables in the specified format. (Table shows the format characters)
  - `(gdb) print /format-char <var-name>`
- The **`whatis`** command is used to check the type of a variable
  - `(gdb) whatis <var-name>`
- The **`set`** command is used to modify the value of a variable
  - `(gdb) set variable <var-name> = <value>`
- Unlike **`print`** the **`display`** command is used to display the value of a variable each time the program stops
  - `(gdb) display <var-name>`
- You can get information about existing displayed variables
  - `(gdb) info display`

```
Num     Enb     Expression
2       y       var-name
```

- You can enable/disable/undisplay an already displayed variable
  - `(gdb) enable/disable display <display#>`
  - `(gdb) undisplay <display#>`

d	Integer, signed decimal
u	Integer, un-signed decimal
x	Integer, print as hex
o	Integer, print as octal
t	Integer, print as binary
f	Floating point number
c	Read as int, print as char
s	Treat as C-string
a	Address

Convenience Variables in GDB

- GDB provides convenience variables that you can use within GDB to hold on to a value and refer to it later. These variables exist entirely within GDB; they are not part of your program, and setting a convenience variable has no direct effect on further execution of your program
  - ➢ `(gdb) set $i = 15`
  - ➢ `(gdb) print $i`

    ```
    $1 = 15
    ```
  - ➢ `(gdb) set $name = “Arif Butt”`
  - ➢ `(gdb) print $name`

    ```
    $2 = “Arif Butt”
    ```
- You can call library functions that are linked with the running process inside gdb
  - ➢ `(gdb) call strcpy($msg, “Hello World”)`
  - ➢ `(gdb) print $msg`

    ```
    $2 = “Hello World”
    ```
- You can call user defined functions in a similar fashion, (once the process is running)
  - ➢ `(gdb) call myadd(25, 34)`

    ```
    $3 = 59
    ```
  - ➢ `(gdb) set $i = 100`
  - ➢ `(gdb) set $j = 25`
  - ➢ `(gdb) call myadd($i, $j)`

    ```
    $4 = 125
    ```

Setting Conditional Break Points

- A conditional break point is similar to a breakpoint that is set if the condition is met

```
4. int main(int argc, char* argv[]){
5.  for(int ctr=0; ctr<=10; ctr++)
6.    printf("The value of ctr is %d \n", ctr);
7.  return 0;
8. }
```

- **(gdb) break <line#> if <condition>**
- Consider the code snippet, in which we want to put a break when the value of the variable `ctr` value reaches 5. Place a breakpoint at the `printf` instruction
  - **(gdb) break 6 if ctr == 5**
- Get information about existing break point
  - **(gdb) info break**

```
Num      Type            Disp   Enb   Address              What
1      breakpoint      keep    y   0x000000000000114d      in main at cond_bp.c:6
       stop only if ctr == 5
```

- Now run the program, it will stop only when the condition is met
  - **(gdb) run**
  - **(gdb) print ctr**

```
$1 = 5
```

Demo

Use of

Breakpoints, watchpoints, stepping, displaying and modifying

31/ex2/fact.c

Text User Interface of gdb

- The gdb Text User Interface, TUI in short, is a terminal interface which uses the curses library to show the source file, the assembly output, the program registers and gdb commands in separate text windows
- To run gdb in tui mode give following command

```
$ gdb –q -tui
```

- If gdb is already running, you can switch from simple command mode to tui mode by giving following command
  - ➢ `(gdb) tui enable/disable`
- In TUI mode, gdb can display several text windows:
  - Command Window: Displays the gdb prompt to get the commands and also displays the GDB output
  - Source Window: Displays the source file of the program along with current line and active breakpoints. The current line is shown with > marker and active breakpoints are shown with * marker
  - Assembly Window: Displays the disassembly of the program
  - Register Window: Displays the processor registers. Registers are highlighted when their values change

Layouts of Text User Interface of gdb

- Layout means which TUI windows are displayed. There are different layout, some are mentioned below:
  - Source Only
    - `(gdb) layout src`
  - Assembly Only
    - `(gdb) layout asm`
  - Source and Assembly
    - `(gdb) layout split`
  - Registers
    - `(gdb) layout regs`
- To change the register group displayed in the tui register window, you can use the following command
  - `(gdb) tui reg general/float/vector/all`
- To move the focus to any window for scrolling, use the following command
  - `(gdb) focus src/asm/regs/cmd`

Demo

https://sourceware.org/gdb/onlinedocs/gdb/TUI.html

Bonus

Installing GDB and other ToolChain

- The set of programming tools used to create a binary executable is referred to as the tool chain (Compiler, Assembler, Linker, Loader, Debugger). Some GNU Tool Chains are gcc, glibc, binutils, bison, m4, gdb, make, cmake, autoconf, automake, …
- In this part of the course there are number of different tools that are required to be installed. If you are using Debian based Linux you can confirm the availability of following tools or packages by giving the following commands on your terminal:

```
$ sudo apt-get install binutils
$ sudo apt-get install build-essential
$ sudo apt-get install gcc make cmake gdb nasm
$ sudo apt-get install libglib2.0-dev
$ sudo apt-get install valgrind electric-fence
$ sudo apt-get install bison
```

Loading Multiple Processes in GDB

- GDB let you load multiple programs in a single session and switch focus between them
- GDB represents the state of each program executing with an object called inferior
- You can load multiple binaries in `gdb` using following command:

```
$ gdb
(gdb) add-inferior –exec myexe1
(gdb) add-inferior –exec myexe2
```

- To get information about different loaded processes inside gdb:

```
(gdb)info inferiors
```

- To switch focus from one program to another:

```
(gdb)inferior <inferior#>
```

<u>ToDo</u>:

Use `gdb` to execute/debug multiple processes simultaneously, which are communicating with each other via different IPC mechanisms

Static Analysis of a Program

- With poorly coded programs a hacker can use tools like **`strings`** and **`nm`** to reveal private/secret information and crack it

```
$ strings  <binary_file>
$ nm <binary_file>
```

- So a smart developer never ships his program with debugging symbols as it is a security loop hole and let a hacker analyze and exploit your program and perform reverse engineering
  - Make a copy of debugging symbols in another file

```
$ objcopy  --only-keep-debug  <binary_file>  <symbol_file>
```

  - Now strip all symbols from the binary file

```
$ strip  --strip-debug  --strip-unneeded  <binary_file>
```

  - If you have a binary_file w/o debug symbols and the symbol_file containing debug symbols you can merge them with following command:

```
$ objcopy  --add-gnu-debuglink=<symbol_file>  <binary_file>
```

Postmortem of a Dead Process

- Whenever a process abnormally terminates on receipt of a signal (`SIGFPE, SIGSEGV, SIGQUIT`)from operating system, Linux creates a core file which is a binary image of the process at the time of its death
- In most OS configurations the generation of core file is disabled. To enable it on Linux, give the following command:

```
$ ulimit –c unlimited
```

- The program code once executed will give a Floating point exception with core dumped
- You can load the core dump file along with the program in gdb to investigate the reason of process death

```
$ gdb myexe corefile
```

```
int main()
{
  int i = 7,j = 0;
  float f = i/j;
  printf("%d/%d=%.2f\n",i,j,f);
  return 0;
}
```

Things To Do

O.k., and now you'll do exactly what I'm telling you !
Access denied