CMSC 131 Introduction to Computer Organization and Machine-Level Programming
Handout 2: Introduction to GNU Debugger (gdb)

Learning Outcomes

At the end of this session, the student should be able to:

1. describe the purpose of the GNU Debugger;
2. understand data movement in Assembly; and
3. perform debugger commands.

Content

I. GNU Debugger
II. Data Movement in Assembly

GNU Debugger

A debugger allows the user to control the execution of a program, examine variables, other memory (i.e., stack space), and display program output (if any). GNU Debugger, which is also called gdb, is the most popular debugger for UNIX systems.

```
1   global _start
2
3   section .data
4         SYS_EXIT equ 60
5         bnum db 5
6         wnum dw 2000
7         dnum dd 100000
8         qnum dq 1234567890
9         message db "Hello!",10
10
11  section .text
12
13  _start:
14        mov ax, word[wnum]     ; ax = wnum
15        mov al, byte[bnum]     ; al = bnum
16
17        mov ebx, dword[dnum]   ; ebx = dnum
18        mov bx, word[wnum]     ; bx = wnum
19
20        ;exit code
21        mov rax, SYS_EXIT
22        xor rdi, rdi
23        syscall
```

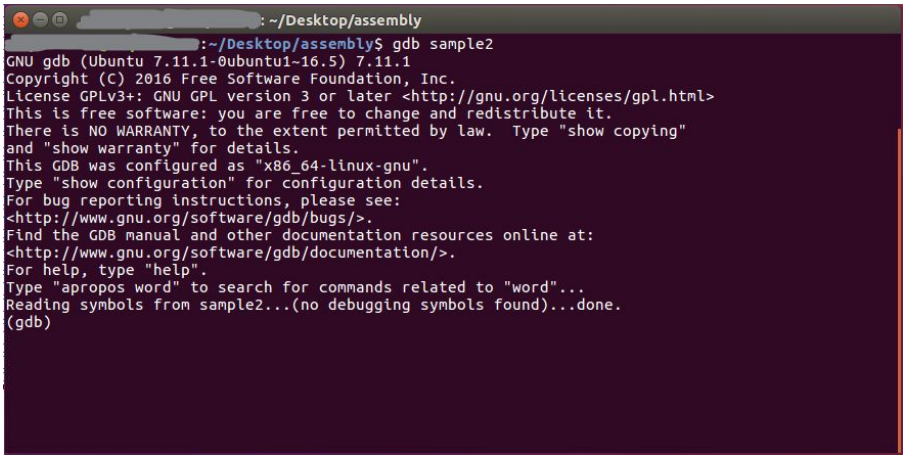
* *Save as **sample2.asm**. Assemble and link.*

Starting gdb

gdb is started with the executable file:

```
gdb <execFilename>
```

In the given example code above, `gdb sample` should display something like:



To view the list of functions of the executable:

```
info functions
```

Among the displayed functions, you can choose which one to disassemble:

```
disas <functionName>
```

In the given example, `_start` is the only function available for disassembly:

```
disas _start
```

This will dump the assembler code for the function: the memory address (left column) and the line of code (right column).

The default disassembly syntax in AT&T. To switch to Intel:

```
set disassembly-flavor intel
```

Try `disas _start` again to see the difference.

To run the executable in gdb:

```
r [<commandlineArgs>]
```

This will execute the program entirely but no debugging is done.

In order to effectively use the debugger, it is necessary to set a breakpoint (execution pause location) to pause the program at a selected location. To set a breakpoint:

```
b*<functionName>[+<offset>]
```

When the program is executed, it will execute *up to*, but *not including* the statement with a breakpoint. An arrow/smiley (=>) symbol will also appear during disassembly pointing to the next statement to be executed.

Multiple breakpoints can be set. The list of existing breakpoints can be viewed:

```
info breakpoints / i b
```

At this point, while the program is paused, we can inspect the registers:

```
info registers / i r
```

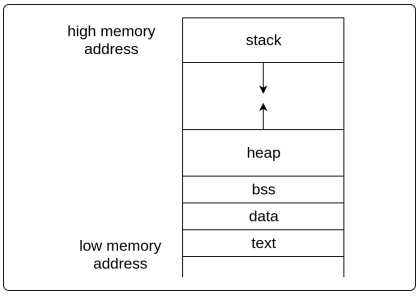
This will display the register contents by register name (left column), in both hex (middle column) and signed decimal (right column). It can also be used with a specific register:

```
info registers [<registerName>] / i r [<registerName>]
```

Remember the memory layout in the previous session? It can be viewed on gdb using:

```
info proc mappings
```

Depending on the contents of the program, this will display the text section, the data+bss sections, the stack, and the heap according to the layout we have discussed:




```
(gdb) info proc mappings
process 4953
Mapped address spaces:

          Start Addr           End Addr       Size     Offset objfile
            0x400000           0x401000     0x1000        0x0 /home/[redacted]
            0x600000           0x601000     0x1000        0x0 /home/[redacted]
      0x7ffff7ffa000     0x7ffff7ffd000     0x3000        0x0 [vvar]
      0x7ffff7ffd000     0x7ffff7fff000     0x2000        0x0 [vdso]
      0x7ffffffde000     0x7fffffff000    0x21000        0x0 [stack]
  0xffffffffff600000 0xffffffffff601000     0x1000        0x0 [vsyscall]
(gdb)
```

In the debug screen above:
`0x400000` to `0x401000` is allocated for the text section,
`0x600000` to `0x601000` is for data and bss sections, and
`0x7fffffde000` to `0x7fffffff000` is for the stack.

To execute the next instruction:

```
ni
```

Notice that the arrow symbol moved to the next instruction.

To *examine* memory location:

`x/<n><f><u> &<variable>`	<n> number of locations to display, 1 is default. <f> format: d – decimal (signed) x – hex u – decimal (unsigned) c – character s – string f – floating point <u> unit size: b – byte (8-bits) h – halfword (16-bits) w – word (32-bits) g – giant (64-bits)

If an inappropriate memory dump command is used (i.e., incorrect size), there is no error message and the debugger will display what was requested (even if it does not make sense). Examining variables will require use of the appropriate memory dump command based on the data declarations.

For the given example, assuming signed data, examine memory commands would be as follows:

```
x/db &bnum
x/dh &wnum
x/dw &dnum
x/dg &qnum
x/s &message
```

The output is the variable address(left column) and the value (right column). The examine memory commands can also be used with a memory address directly (as opposed to a variable name):

```
x/db 0x6000d8
```

To *print* the contents of a register:

```
print [/<f>] $<register> / p[/<f>] $<register>
```

The <f> has the same options as in examine memory locations for variables.
Examples:

```
p $rax        ;prints 0
p/x $rax      ;prints 0x0
```

Commands Cheat sheet

Command	Description
`set disassembly-flavor intel`	Set syntax to Intel
`info registers / i r`	View registers' information
`info functions`	View list of functions
`info proc mappings`	View memory mapping
`disas <functionName>`	Disassemble function
`r`	Run from the start
`b*<functionName>[+offset]`	Set a breakpoint
`info breakpoints / i b`	View list of breakpoints
`clear*<functionName>[+offset]`	Delete breakpoint
`delete break <breakpointNum>`	Delete breakpoint with num from the list of breakpoints

`x/<n><f><u> &<variable>`	Examine memory location
`p[/<f>] $<register>`	Print contents of a register
`ni`	Execute next instruction
`si`	Same as ni but executes function calls
`cont / c`	Continue until next breakpoint
`quit / q`	Quit the debugger

Data Movement

Typically, data must be moved into a CPU register from RAM in order to be operated upon. Once the calculations are completed, the result may be copied from the register and placed into a variable. The general form of the move instruction is:

```
mov <dest>, <src>
```

The source operand is copied from the source operand into the destination operand.
Operand can be:

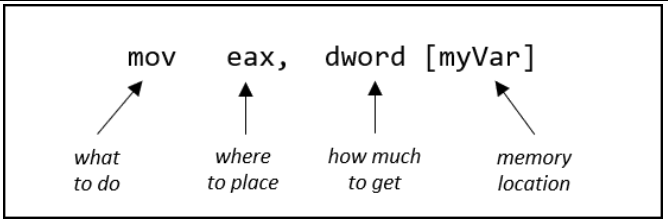
- Register (`rax, ebx, cx, dh`)
- Immediate value (`2, 10, 0x1A`)
- Memory / variable (square brackets denote the value)
  - `[myVar]` - value of `myVar`
  - `myVar` - address of `myVar`
- Destination: Register or variable in brackets
- Source: Register, Immediate value, variable with/without brackets

Rules:

- The destination and source operand must be of the **same size** (both bytes, both words, etc.).
- The destination operand can not be an immediate.
- Operands can not be both memory. If a memory to memory operation is required, two instructions must be used.

Therefore, to perform:

```
eax = myVar
```



Other examples:

```
mov   ax, 42              ; ax = 42
mov   cl, byte [bVar]     ; cl = bVar
```

```
mov   dword [dVar], eax     ; dVar = eax
mov   qword [qVar], rdx     ; qVar = rdx

mov   al, byte [bVar]
mov   byte [bAns], al       ; bAns = bVar
mov   rsi, bVar             ; rsi = &bVar
```

References

Jorgensen, Ed. 2019. x86-64 Assembly Language Programming with Ubuntu. Version 1.1.40.

Hermocilla, JAC, Queliste, MD, and de Robles, MBB. 2018. Buffer Overflow Exploitation on 64-bit Linux Systems Slides.

Bellenthin, W. yet another gdb cheatsheet. https://gist.github.com/williballenthin/8bd6e29ad8504b9cb308039f675ee889