CSED211 Lab 03

Instruction & GDB debugger

2018. 09. 19.

Announcement

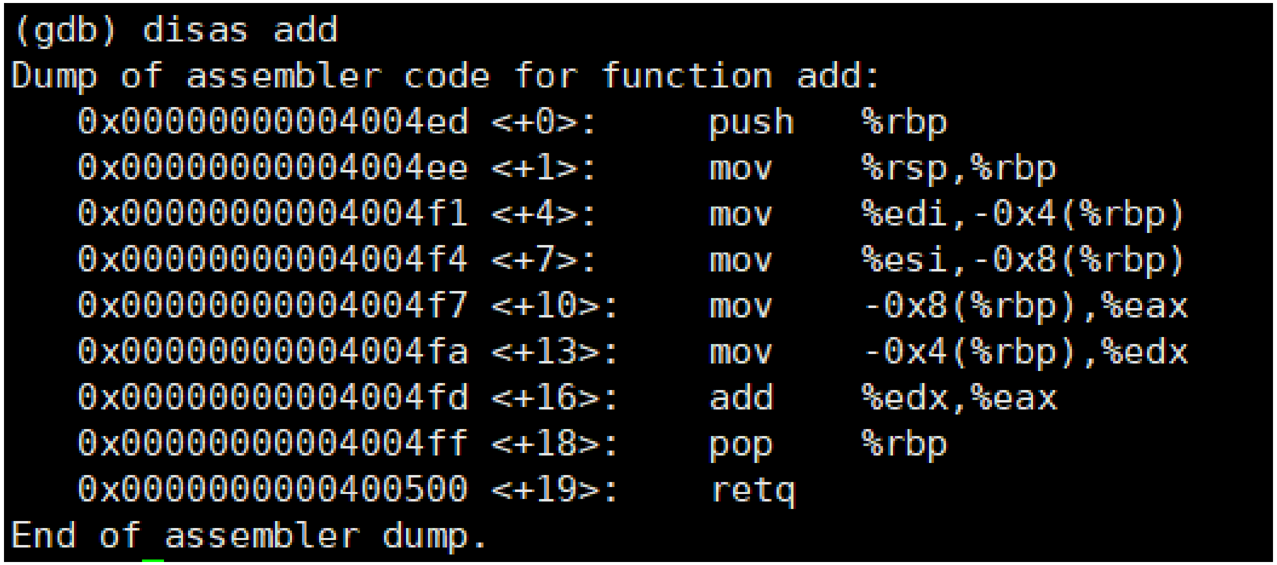
- 9/26(Wed): No class
    - Thanks giving day : Have a nice holiday!

- However, we have a class in 10/3(Wed)...

Assembly Language: Instruction

Assembly Language

- Machine-friendly language
- Result of 'compile'
- What machine really sees and runs



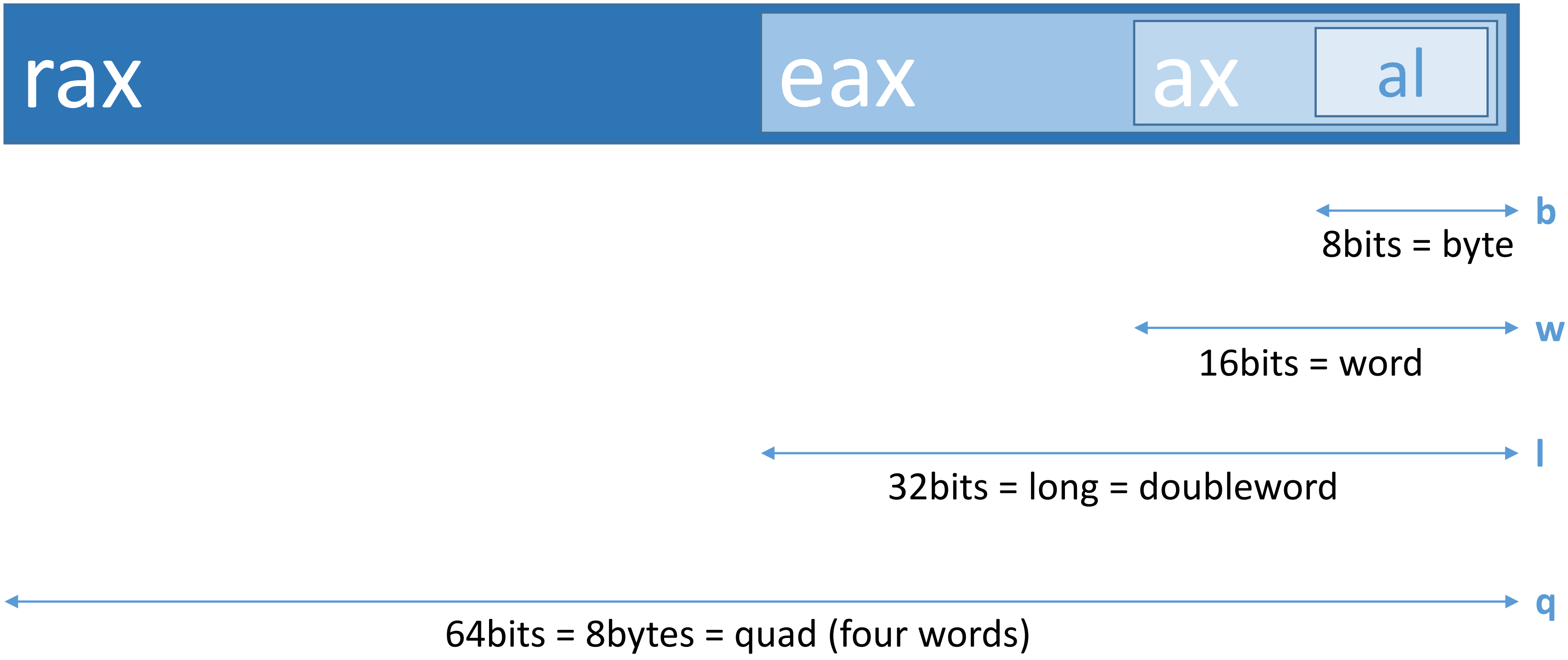
```
(gdb) disas add
Dump of assembler code for function add:
   0x00000000004004ed <+0>:     push   %rbp
   0x00000000004004ee <+1>:     mov    %rsp,%rbp
   0x00000000004004f1 <+4>:     mov    %edi,-0x4(%rbp)
   0x00000000004004f4 <+7>:     mov    %esi,-0x8(%rbp)
   0x00000000004004f7 <+10>:    mov    -0x8(%rbp),%eax
   0x00000000004004fa <+13>:    mov    -0x4(%rbp),%edx
   0x00000000004004fd <+16>:    add    %edx,%eax
   0x00000000004004ff <+18>:    pop    %rbp
   0x0000000000400500 <+19>:    retq
End of assembler dump.
```

Registers

- A quickly accessible storage available to CPU
- General purpose registers:
  - rax (return value) → eax in 32bit
  - rbx (calle saved) → ebx in 32 bit
  - rcx (4th argument) → …
  - rdx (3rd argument)
  - rsi (2nd argument)
  - rdi (1st argument)
- Special purpose registers:
  - rsp (stack pointer)

Reference ***Figure 3.2***

Size of Registers



rax

eax

ax

al

b

8bits = byte

w

16bits = word

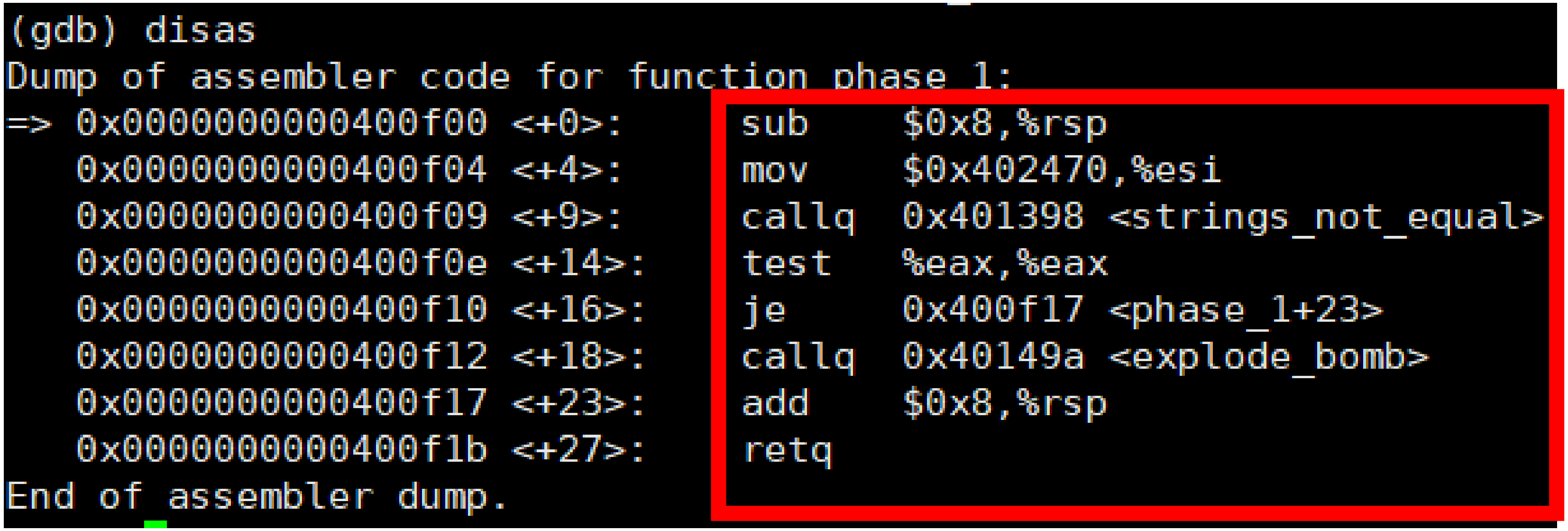
l

32bits = long = doubleword

q

64bits = 8bytes = quad (four words)

Assembly Instruction



```
(gdb) disas
Dump of assembler code for function phase_1:
=> 0x0000000000400f00 <+0>:     sub    $0x8,%rsp
   0x0000000000400f04 <+4>:     mov    $0x402470,%esi
   0x0000000000400f09 <+9>:     callq  0x401398 <strings_not_equal>
   0x0000000000400f0e <+14>:    test   %eax,%eax
   0x0000000000400f10 <+16>:    je     0x400f17 <phase_1+23>
   0x0000000000400f12 <+18>:    callq  0x40149a <explode_bomb>
   0x0000000000400f17 <+23>:    add    $0x8,%rsp
   0x0000000000400f1b <+27>:    retq
End of assembler dump.
```

Assembly Op Code

- Unit of operation
- Calculate and save the result to register/memory
- Binary operators:

  OP SRC, DST ⇔ DST = ***OP***(SRC, DST)

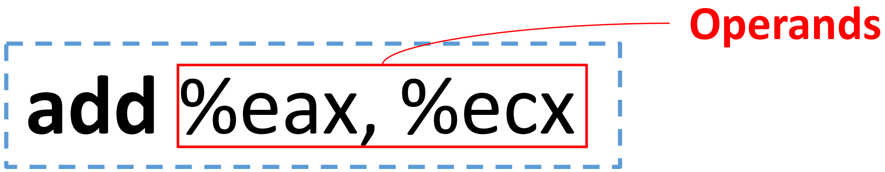
  - mov, sub, add, cmp, …
  - e.g., **add** %eax, %ecx → ecx = eax **+** ecx

- Unary operators:

  OP DST ⇔ DST = ***OP***(DST)

  - inc, dec, neg, not, …

Assembly Operands



add %eax, %ecx

Operands

- Immediate
    - $0x8, $5, $-1
- Register
    - %rsp, %esi, %eax, %r14
- Memory
    - D(Rb, Ri, S)
    - 8(%ebx), 12(%ebx, %ecx, 4)

Reference ***Figure 3.3***

Assembly Basic – 1

- Data movement instructions

```
MOV           a1, a2   ( a2 = a1 )

%rbp          → p
(%rbp)        → *p
20(%rbp)      → *(p + 20)
```

Assembly Basic – 2

- Arithmetic instructions

  ADD a1, a2
    - a2 = a2 + a1

  SUB a1, a2
    - a2 = a2 - a1

  IMUL a1, a2
    - a2 = a2 * a1

Special Operations

- CLTQ (may see it in bomblab):
  - Convert long to quad : %rax = SignExtend(%eax)
  - eax = 0x05 → rax = 0x05
  - eax = 0x80000000 → rax = 0xffffffff 80000000

- LEA: load effective address
  - Does not dereference memory
  - Used as an arithmetic operator to perform memory address calculations
    - Commonly used for calculating offsets into an array in a loop
  - lea 7(%rdx, %rdx, 4), %rax ⇔ rax = 5 * rdx + 7

Control: Compare and Flags

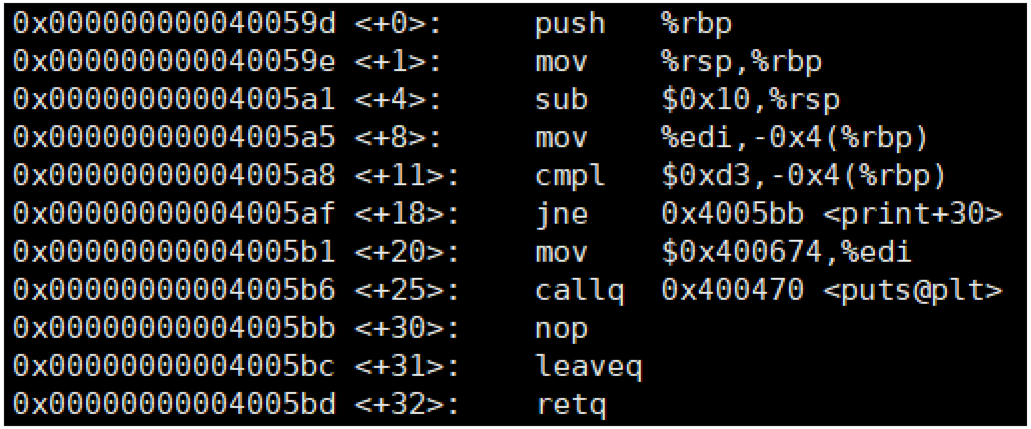
- Condition codes:
  - CMP S1, S2 → executes R = S2 – S1, and set ***Flags***
  - TEST S1, S2 → executes R = S1 & S2, and set ***Flags***

- Flags (about R = S1 + S2)
  - **CF** (Carry Flag) is set if **(unsigned) R < (unsigned) S1**
  - **ZF** (Zero Flag) is set if **R == 0**
  - **SF** (Sign Flag) is set if **R < 0**
  - **OF** (Overflow flag) is set if **(S1 < 0 == S2 < 0) && (R < 0 != S1 < 0)**

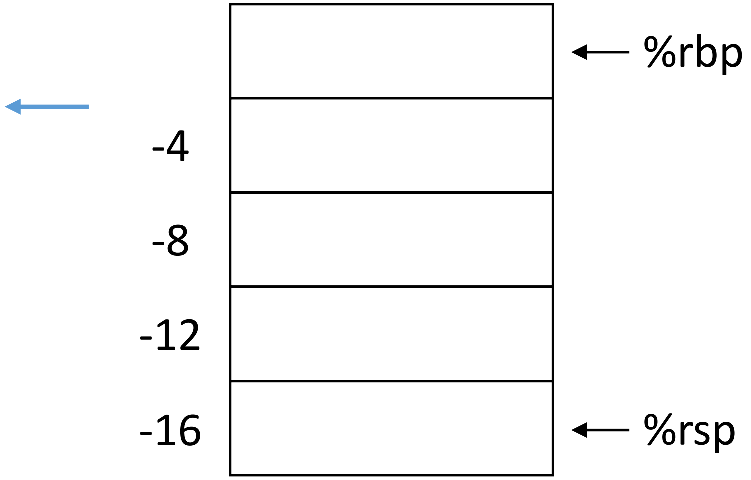
Control: Jumps

Instruction	**Jump condition**	**Description**
jmp _Label_	1	Direct jump
jmp _*Operand_	1	Indirect jump
je _Label_	ZF	Equal / zero
jne _Label_	~ZF	Not equal / not zero
jg _Label_	~(SF^OF)&~ZF	Greater (>)
jge _Label_	~(SF^OF)	Greater or equal(>=)
jl _Label_	SF^OF	Less (<)
jle _Label_	(SF^OF)|ZF	Less or equal (<=)

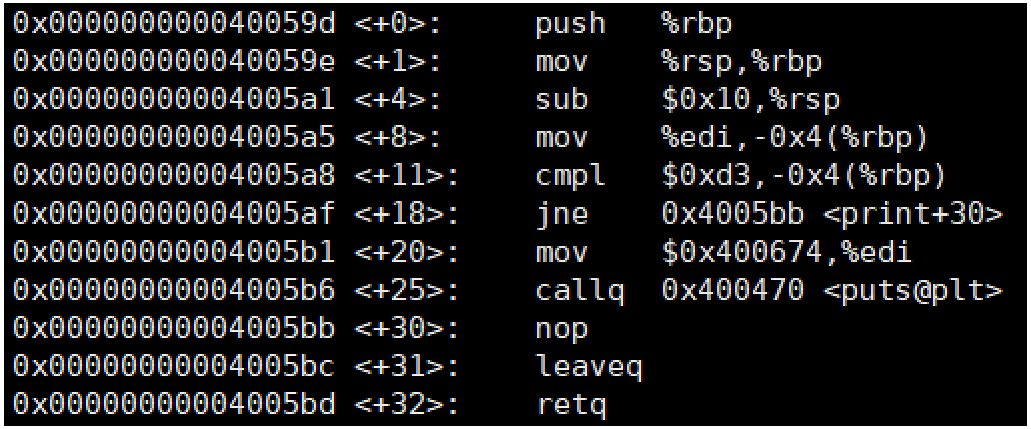
Example of Control



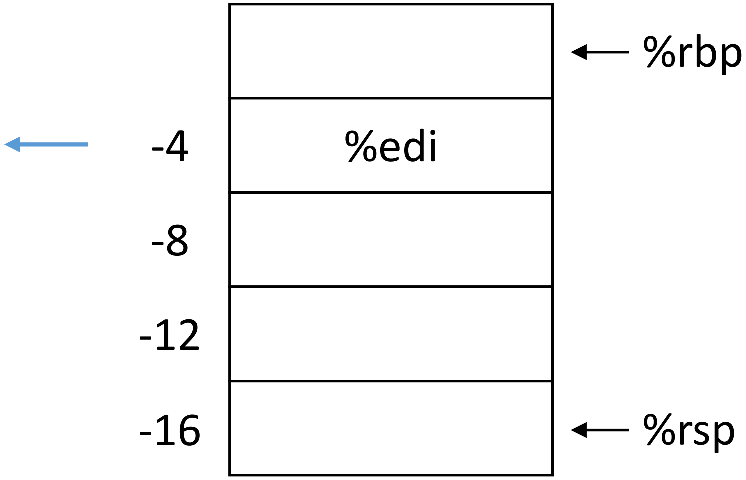
```
0x000000000040059d <+0>:     push   %rbp
0x000000000040059e <+1>:     mov    %rsp,%rbp
0x00000000004005a1 <+4>:     sub    $0x10,%rsp
0x00000000004005a5 <+8>:     mov    %edi,-0x4(%rbp)
0x00000000004005a8 <+11>:    cmpl   $0xd3,-0x4(%rbp)
0x00000000004005af <+18>:    jne    0x4005bb <print+30>
0x00000000004005b1 <+20>:    mov    $0x400674,%edi
0x00000000004005b6 <+25>:    callq  0x400470 <puts@plt>
0x00000000004005bb <+30>:    nop
0x00000000004005bc <+31>:    leaveq
0x00000000004005bd <+32>:    retq
```



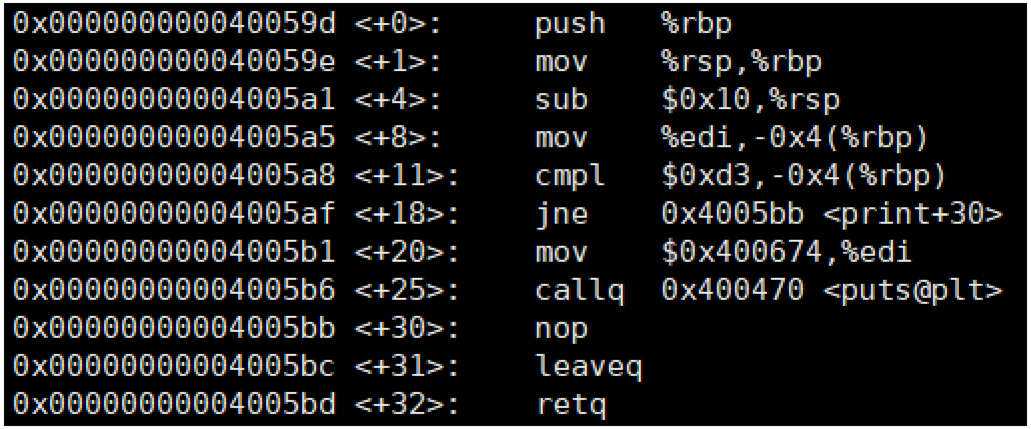
Example of Control



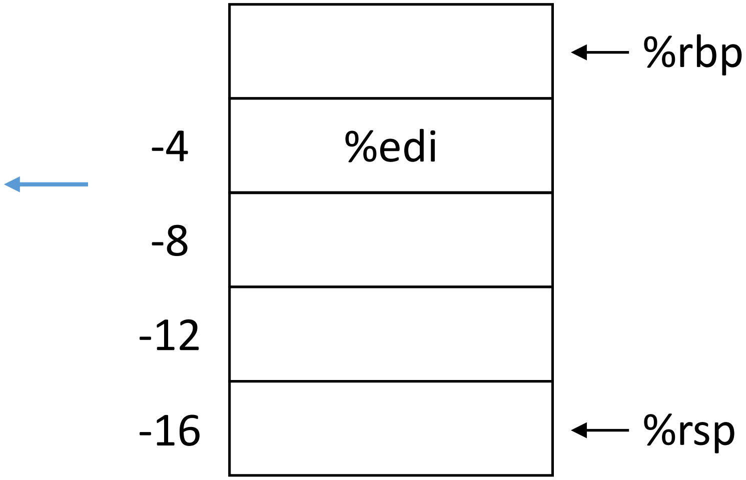
```
0x000000000040059d <+0>:     push   %rbp
0x000000000040059e <+1>:     mov    %rsp,%rbp
0x00000000004005a1 <+4>:     sub    $0x10,%rsp
0x00000000004005a5 <+8>:     mov    %edi,-0x4(%rbp)
0x00000000004005a8 <+11>:    cmpl   $0xd3,-0x4(%rbp)
0x00000000004005af <+18>:    jne    0x4005bb <print+30>
0x00000000004005b1 <+20>:    mov    $0x400674,%edi
0x00000000004005b6 <+25>:    callq  0x400470 <puts@plt>
0x00000000004005bb <+30>:    nop
0x00000000004005bc <+31>:    leaveq
0x00000000004005bd <+32>:    retq
```



Example of Control

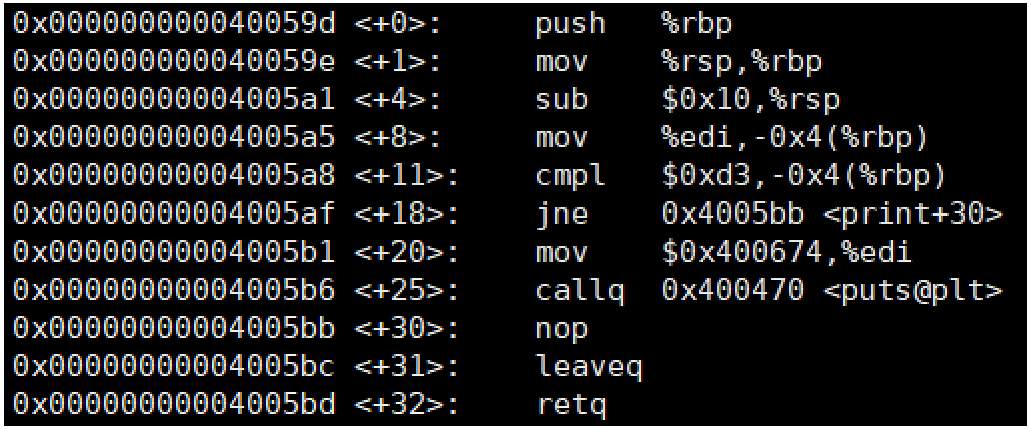


```
0x000000000040059d <+0>:     push   %rbp
0x000000000040059e <+1>:     mov    %rsp,%rbp
0x00000000004005a1 <+4>:     sub    $0x10,%rsp
0x00000000004005a5 <+8>:     mov    %edi,-0x4(%rbp)
0x00000000004005a8 <+11>:    cmpl   $0xd3,-0x4(%rbp)
0x00000000004005af <+18>:    jne    0x4005bb <print+30>
0x00000000004005b1 <+20>:    mov    $0x400674,%edi
0x00000000004005b6 <+25>:    callq  0x400470 <puts@plt>
0x00000000004005bb <+30>:    nop
0x00000000004005bc <+31>:    leaveq
0x00000000004005bd <+32>:    retq
```

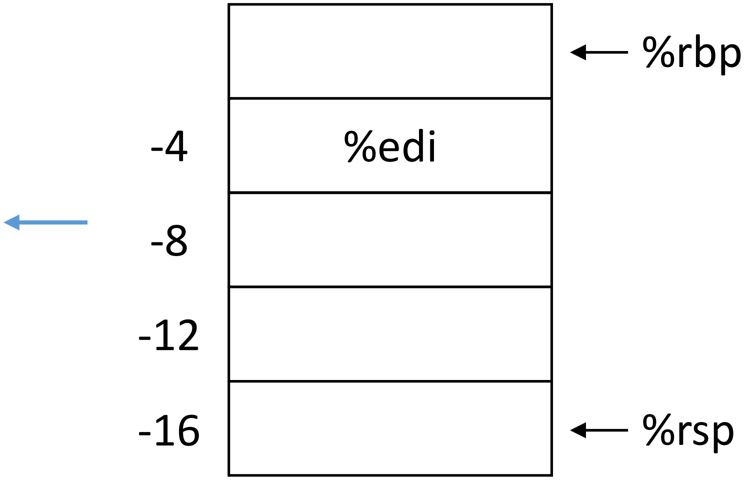


Compare 0xd3 and %edi

Example of Control



```
0x000000000040059d <+0>:     push   %rbp
0x000000000040059e <+1>:     mov    %rsp,%rbp
0x00000000004005a1 <+4>:     sub    $0x10,%rsp
0x00000000004005a5 <+8>:     mov    %edi,-0x4(%rbp)
0x00000000004005a8 <+11>:    cmpl   $0xd3,-0x4(%rbp)
0x00000000004005af <+18>:    jne    0x4005bb <print+30>
0x00000000004005b1 <+20>:    mov    $0x400674,%edi
0x00000000004005b6 <+25>:    callq  0x400470 <puts@plt>
0x00000000004005bb <+30>:    nop
0x00000000004005bc <+31>:    leaveq
0x00000000004005bd <+32>:    retq
```



Compare 0xd3 and %edi

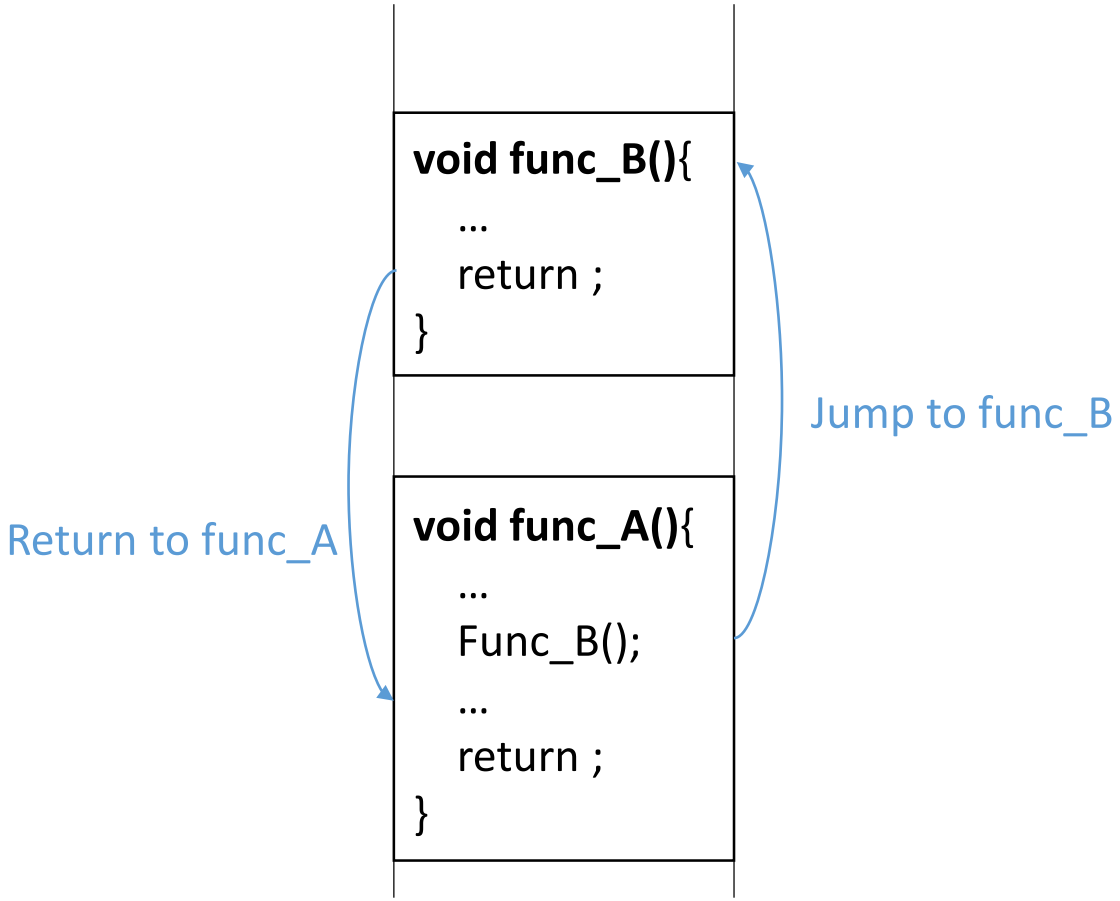
→ if not equal, jump to <print+30>

Stack

- Data structure to manage execution control flow
  - Located in memory
  - Variables can be stored in stack
  - Last in, First out (LIFO)
  - Stack grows downward
    - %rbp: previous stack pointer
    - %rsp: current stack pointer
  - **PUSH**, **POP** instruction
    - PUSH: push new data on the top
    - POP: pop (take out) the top data of stack

Function Call Procedure (1/2)

```
void fun_A(){
     …
     fun_B();
     …
}
```



```
void func_B(){
   …
   return ;
}
```

```
void func_A(){
   …
   Func_B();
   …
   return ;
}
```

Function Call Procedure (2/2)

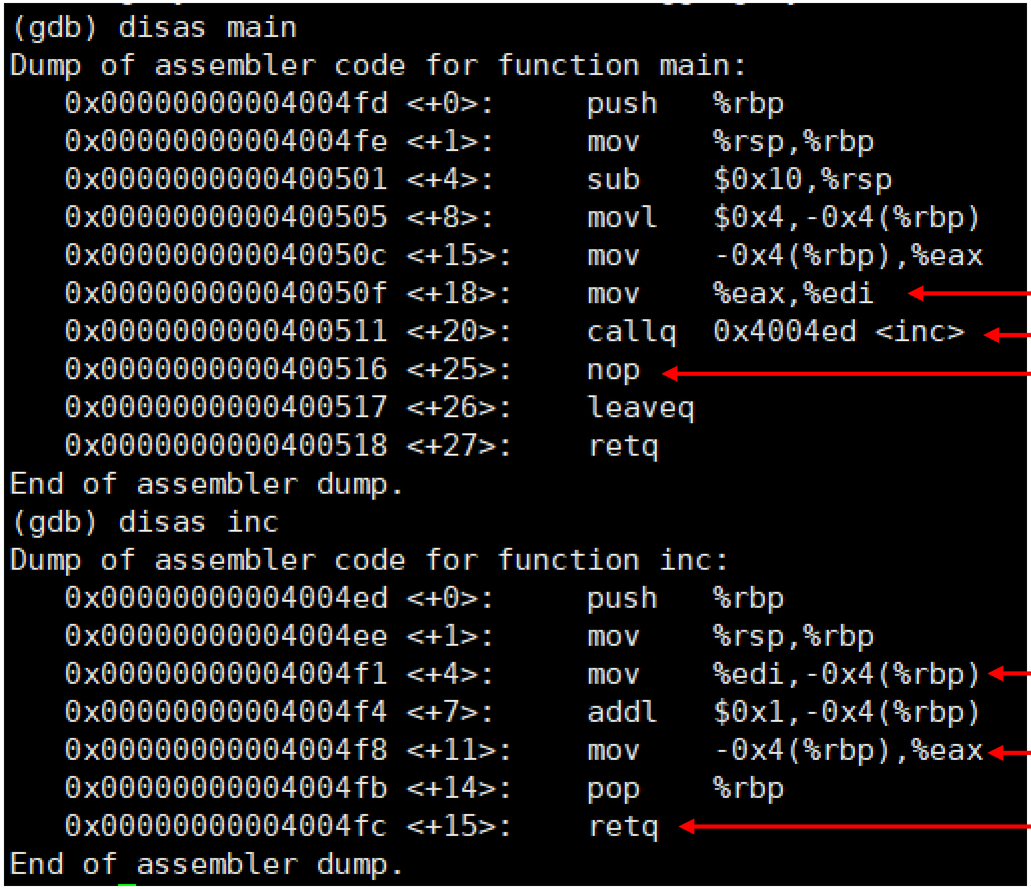
- Caller function
  - Function which calls function
- Callee function
  - Function which is called by caller function
- CALL and RET Instruction
  - CALL: saves return address on the stack and jump to the callee function
  - RET: transfers program control to a return address located on the top of the stack

Return Address & Function Parameter

- Return address
    1. Caller function calls Callee function and Callee function is executed
        - Call instruction stores the return address (next instruction address from call)
    2. Return to proper address in caller function with the return address

- Function parameter
    1. Store the parameter in register or stack
    2. Caller function calls Callee function.
    3. Callee function uses the register or stack data as parameter.

Example of Procedure



```
(gdb) disas main
Dump of assembler code for function main:
   0x00000000004004fd <+0>:     push   %rbp
   0x00000000004004fe <+1>:     mov    %rsp,%rbp
   0x0000000000400501 <+4>:     sub    $0x10,%rsp
   0x0000000000400505 <+8>:     movl   $0x4,-0x4(%rbp)
   0x000000000040050c <+15>:    mov    -0x4(%rbp),%eax
   0x000000000040050f <+18>:    mov    %eax,%edi
   0x0000000000400511 <+20>:    callq  0x4004ed <inc>
   0x0000000000400516 <+25>:    nop
   0x0000000000400517 <+26>:    leaveq
   0x0000000000400518 <+27>:    retq
End of assembler dump.
(gdb) disas inc
Dump of assembler code for function inc:
   0x00000000004004ed <+0>:     push   %rbp
   0x00000000004004ee <+1>:     mov    %rsp,%rbp
   0x00000000004004f1 <+4>:     mov    %edi,-0x4(%rbp)
   0x00000000004004f4 <+7>:     addl   $0x1,-0x4(%rbp)
   0x00000000004004f8 <+11>:    mov    -0x4(%rbp),%eax
   0x00000000004004fb <+14>:    pop    %rbp
   0x00000000004004fc <+15>:    retq
End of assembler dump.
```

1 Save *parameter*

2 Call *inc function*

6 *Return address* from *inc function*

3 *Use parameter*

4 Save *return value*

5 Return to *main*

GDB

GDB

- The GNU Project debugger
- It runs on Linux systems
- It works for C, C++, Objective-C, Pascal, …

- To debug a program with GDB, include “-g” option when compile
  - gcc example.c –g –o example
- Execute GDB with an execution file
  - gdb example

GDB commands

- disas (Disassemble)
  - shows machine codes
  - disas *func* : shows the machine code of the function ‘*func*’
  - disas *10* : shows the machine code at the line ’*10*’
  - disas **0x300* : shows the machine code at the address ‘*0x300*’

- l (List)
  - shows source codes
  - l *10* : shows the source code at the line ‘*10*’
  - l *func* : show source codes of the function ‘*func*’

GDB commands

- b (Break)
  - makes a breakpoint
  - b *func* : break before executing the function ‘*func*’
  - b *10* : break at the line ‘*10*’
  - b **0x300* : break at the address ‘*0x300*’
- cl (Clear)
  - deletes a breakpoint
  - same as above
- d (Delete)
  - deletes all breakpoints

GDB commands

- r (Run)
  - runs the program
  - r *arg1 arg2* : run the program with arguments ‘*arg1*’ and ‘*arg2*’
- k (Kill)
  - kills currently running program
- n (Next)
  - executes the next instruction
  - n *6* : executes next 6 instructions
- c (Continue)
  - executes the instructions before the next breakpoint

GDB commands

- info (Information)
  - shows available symbols in the program
  - info functions : lists available functions at the current line
  - info variables : lists available global variables at the current line
  - info locals : lists available local variables at the current line

- p (Print)
  - prints the value of a variable
  - p *var* : prints the value of the variable ‘*var*’
  - p *func* : prints the address of the function ‘*func*’
  - p **0x300* : prints the value at the address ‘*0x300*’
  - p $*eax* : prints the value of the register ‘*eax*’
  - p/<format><expression>: print expression in format
    - e.g., p/c 0x61 → print ‘a’ which is character expression of 0x61

GDB commands

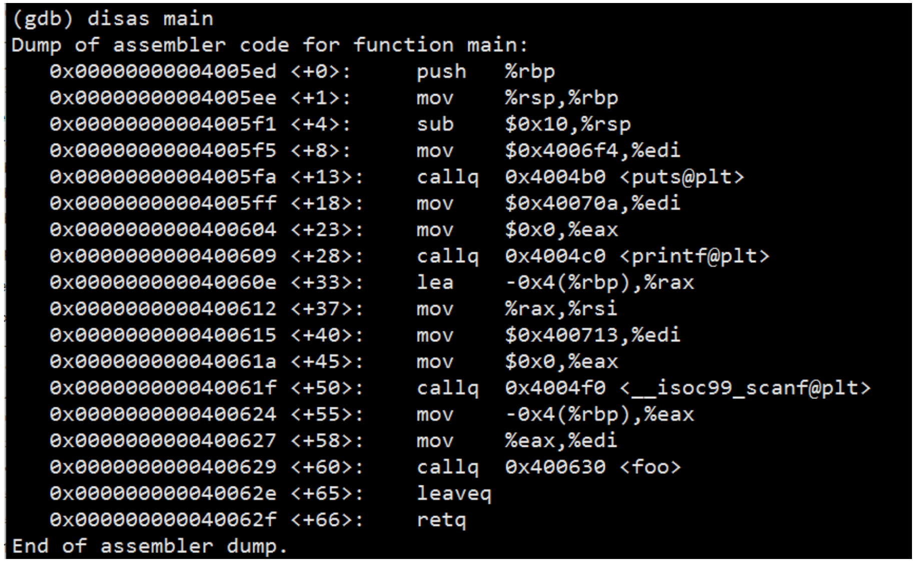
- X(check the memory)
  - x/<format> <address>: print contents of memory in format
    - e.g., x/s 0x4005a57 → print string stored in 0x4005a57

- More Information
  - GDB online manuals
    - https://sourceware.org/gdb/current/onlinedocs/gdb/
  - Feel free to ask Google

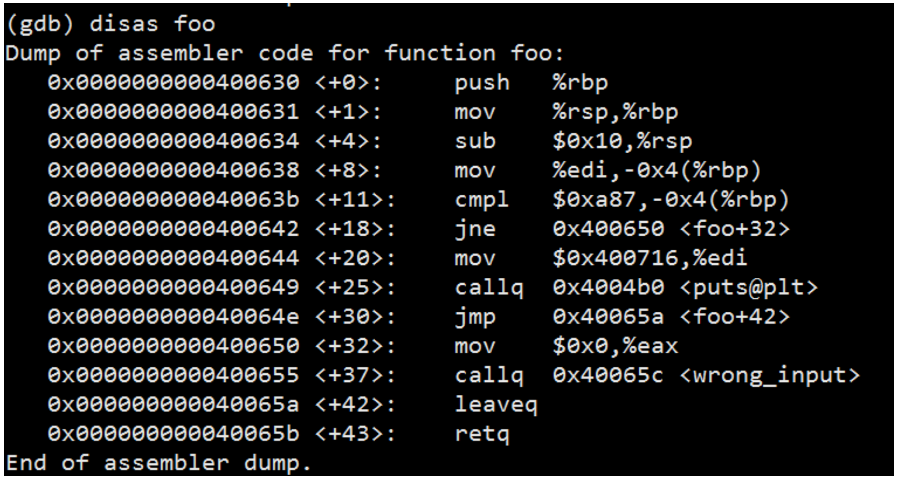
Practice

- Analyze the assembly code using GDB

- Let’s print something by satisfying the control in *foo* function, avoiding *wrong_input* function



```
(gdb) disas main
Dump of assembler code for function main:
   0x00000000004005ed <+0>:     push   %rbp
   0x00000000004005ee <+1>:     mov    %rsp,%rbp
   0x00000000004005f1 <+4>:     sub    $0x10,%rsp
   0x00000000004005f5 <+8>:     mov    $0x4006f4,%edi
   0x00000000004005fa <+13>:    callq  0x4004b0 <puts@plt>
   0x00000000004005ff <+18>:    mov    $0x40070a,%edi
   0x0000000000400604 <+23>:    mov    $0x0,%eax
   0x0000000000400609 <+28>:    callq  0x4004c0 <printf@plt>
   0x000000000040060e <+33>:    lea    -0x4(%rbp),%rax
   0x0000000000400612 <+37>:    mov    %rax,%rsi
   0x0000000000400615 <+40>:    mov    $0x400713,%edi
   0x000000000040061a <+45>:    mov    $0x0,%eax
   0x000000000040061f <+50>:    callq  0x4004f0 <__isoc99_scanf@plt>
   0x0000000000400624 <+55>:    mov    -0x4(%rbp),%eax
   0x0000000000400627 <+58>:    mov    %eax,%edi
   0x0000000000400629 <+60>:    callq  0x400630 <foo>
   0x000000000040062e <+65>:    leaveq
   0x000000000040062f <+66>:    retq
End of assembler dump.
```



```
(gdb) disas foo
Dump of assembler code for function foo:
   0x0000000000400630 <+0>:     push   %rbp
   0x0000000000400631 <+1>:     mov    %rsp,%rbp
   0x0000000000400634 <+4>:     sub    $0x10,%rsp
   0x0000000000400638 <+8>:     mov    %edi,-0x4(%rbp)
   0x000000000040063b <+11>:    cmpl   $0xa87,-0x4(%rbp)
   0x0000000000400642 <+18>:    jne    0x400650 <foo+32>
   0x0000000000400644 <+20>:    mov    $0x400716,%edi
   0x0000000000400649 <+25>:    callq  0x4004b0 <puts@plt>
   0x000000000040064e <+30>:    jmp    0x40065a <foo+42>
   0x0000000000400650 <+32>:    mov    $0x0,%eax
   0x0000000000400655 <+37>:    callq  0x40065c <wrong_input>
   0x000000000040065a <+42>:    leaveq
   0x000000000040065b <+43>:    retq
End of assembler dump.
```

Step

1. >> tar –xvf practice_gdb.tar
2. >> gdb practice_gdb
3. (gdb) disas main
4. (gdb) disas foo
5. Analyze the control in *foo* function in order to call *printf()* function
6. Reversely trace which variable affect the control
7. Analyze which type of *scanf()* input

Assignment: Bomblab

- Disassemble the binary ./bomb and defuse the bomb!
- Solve 6 phase (*phase_1, phase_2, …, phase _6*) avoiding *explode_bomb*
- Find hidden phase
- Refer to the additional pdf file

- You will have to write the report in full detail for your next lab assignment!
- **Due date: 10/2(Tue) 23:59**

What to Do for *Bomblab*

- This Lab assignment may be difficult!, so …

- Read and understand *Chapter 3* in your textbook
  - *A practice problem* is really helpful for your understanding.

- Use and understand GDB
  - GDB is a useful debugger and will be used in the OS class (CSED312).

- Search what you have to know
  - Lab and class resources are not enough to refuse the binary bomb. If you want to know more, search it (in Google)!