Introduction to 8086 Assembly

Lecture 12

Interfacing Assembly with C

Why interfacing?

- Reasons for
  -

Why interfacing?

- Reasons for
  - Efficiency
  - Low-level programming (accessing hardware, etc.)
  - Using specific CPU instructions
- Reasons against
  -

Why interfacing?

- Reasons for
  - Efficiency
  - Low-level programming (accessing hardware, etc.)
  - Using specific CPU instructions
- Reasons against
  - Compilers are good (and will get better) at optimizing code
  -

Why interfacing?

- Reasons for
  - Efficiency
  - Low-level programming (accessing hardware, etc.)
  - Using specific CPU instructions
- Reasons against
  - Compilers are good (and will get better) at optimizing code
  - Portability

Interfacing with C

- Inline assembly
  - Compiler-dependent; No standard syntax
- Calling assembly functions in C

Interfacing with C

- Inline assembly
  - Compiler-dependent; No standard syntax
- Calling assembly functions in C

Remember: C Calling Conventions

callfunc.c

```c
#include <stdio.h>

int sum(int,int,int,int);

int main() {
  int c;

  c = sum(2,4,8,10);

  return  0;
}

int sum(int a, int b, int c, int d) {
  return a+b+c+d;
}
```

callfunc.asm

```
    .file   "callfunc.c"
    .intel_syntax noprefix
    .text
    .globl  main
    .type   main, @function
main:
    lea     ecx, [esp+4]
    and     esp, -16
    push    DWORD PTR [ecx-4]
    push    ebp
    mov     ebp, esp
    push    ecx
    sub     esp, 20

    push    10
    push    8
    push    4
    push    2
    call    sum
    add     esp, 16

    mov     DWORD PTR [ebp-12], eax
    mov     eax, 0
    mov     ecx, DWORD PTR [ebp-4]
    leave
```

callfunc.asm (cont.)

```
    lea     esp, [ecx-4]
    ret
    .size   main, .-main
    .globl  sum
    .type   sum, @function

sum:
    push    ebp
    mov     ebp, esp
    mov     edx, DWORD PTR [ebp+8]
    mov     eax, DWORD PTR [ebp+12]
    add     edx, eax
    mov     eax, DWORD PTR [ebp+16]
    add     edx, eax
    mov     eax, DWORD PTR [ebp+20]
    add     eax, edx
    pop     ebp
    ret

    .size   sum, .-sum
    .ident  "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.9) 5.4.0
20160609"
    .section    .note.GNU-stack,"",@progbits
```

Toosi
f Technology

Remember: C Calling Conventions

callfunc.c

```c
#include <stdio.h>

int sum(int,int,int,int);

int main() {
  int c;

  c = sum(2,4,8,10);

  return  0;
}

int sum(int a, int b, int c, int d) {
  return a+b+c+d;
}
```

callfunc.asm

```
    .file   "callfunc.c"
    .intel_syntax noprefix
    .text
    .globl  main
    .type   main, @function
main:
    lea     ecx, [esp+4]
    and     esp, -16
    push    DWORD PTR [ecx-4]
    push    ebp
    mov     ebp, esp
    push    ecx
    sub     esp, 20

    push    10
    push    8
    push    4
    push    2
    call    sum
    add     esp, 16

    mov     DWORD PTR [ebp-12], eax
    mov     eax, 0
    mov     ecx, DWORD PTR [ebp-4]
    leave
```

parameters pushed in reverse order

callfunc.asm (cont.)

```
    lea     esp, [ecx-4]
    ret
    .size   main, .-main
    .globl  sum
    .type   sum, @function

sum:
    push    ebp
    mov     ebp, esp
    mov     edx, DWORD PTR [ebp+8]
    mov     eax, DWORD PTR [ebp+12]
    add     edx, eax
    mov     eax, DWORD PTR [ebp+16]
    add     edx, eax
    mov     eax, DWORD PTR [ebp+20]
    add     eax, edx
    pop     ebp
    ret

    .size   sum, .-sum
    .ident  "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.9) 5.4.0
20160609"
    .section    .note.GNU-stack,"",@progbits
```

Toosi
f Technology

Remember: C Calling Conventions

callfunc.c

```
#include <stdio.h>

int sum(int,int,int,int);

int main() {
  int c;

  c = sum(2,4,8,10);

  return  0;
}

int sum(int a, int b, int c, int d) {
  return a+b+c+d;
}
```

callfunc.asm

```
.file   "callfunc.c"
.intel_syntax noprefix
.text
.globl  main
.type   main, @function
main:                          caller
lea     ecx, [esp+4]
and     esp, -16
push    DWORD PTR [ecx-4]
push    ebp
mov     ebp, esp
push    ecx
sub     esp, 20

push    10
push    8
push    4
push    2
call    sum
add     esp, 16

mov     DWORD PTR [ebp-12], eax
mov     eax, 0
mov     ecx, DWORD PTR [ebp-4]
leave
```

Caller clears the parameters from stack

callfunc.asm (cont.)

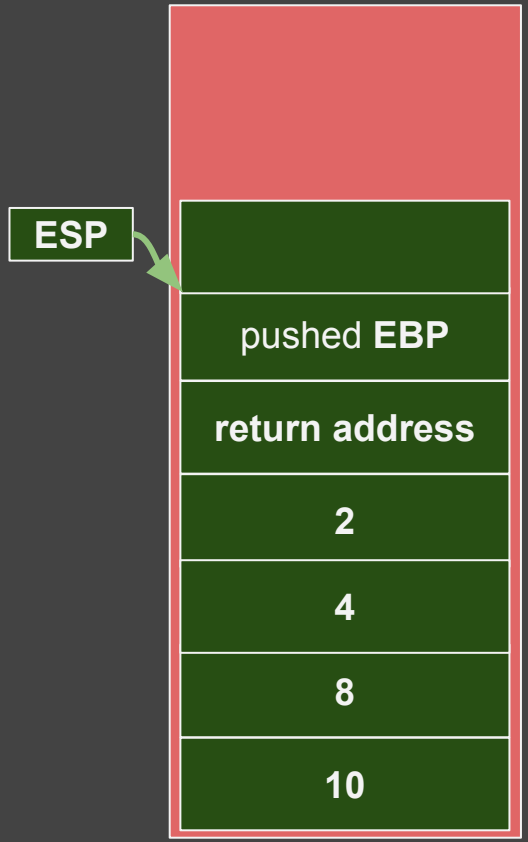
```
lea     esp, [ecx-4]
ret
.size   main, .-main
.globl  sum
.type   sum, @function

sum:                           callee
        push    ebp
        mov     ebp, esp
        mov     edx, DWORD PTR [ebp+8]
        mov     eax, DWORD PTR [ebp+12]
        add     edx, eax
        mov     eax, DWORD PTR [ebp+16]
        add     edx, eax
        mov     eax, DWORD PTR [ebp+20]
        add     eax, edx
        pop     ebp
        ret
        .size   sum, .-sum
        .ident  "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.9) 5.4.0
20160609"
        .section        .note.GNU-stack,"",@progbits
```

Toosi of Technology

Remember: C Calling Conventions

callfunc.c

```
#include <stdio.h>

int sum(int,int,int,int);

int main() {
  int c;

  c = sum(2,4,8,10);

  return  0;
}

int sum(int a, int b, int c, int d) {
  return a+b+c+d;
}
```



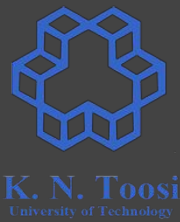
callfunc.asm (cont.)

```
        lea     esp, [ecx-4]
        ret
        .size   main, .-main
        .globl  sum
        .type   sum, @function

sum:                                    callee
        push    ebp
        mov     ebp, esp
        mov     edx, DWORD PTR [ebp+8]          a
        mov     eax, DWORD PTR [ebp+12]         b
        add     edx, eax
        mov     eax, DWORD PTR [ebp+16]         c
        add     edx, eax
        mov     eax, DWORD PTR [ebp+20]         d
        add     eax, edx
        pop     ebp                     return value
        ret                             stored in EAX

        .size   sum, .-sum
        .ident  "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.9) 5.4.0
20160609"
        .section        .note.GNU-stack,"",@progbits
```

C calling conventions (32-bit)

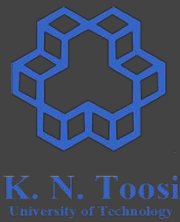
- Parameters are push on stack in reverse order
- The caller removes parameters from stack
- Return value stored in EAX (not in all cases, see next page)
- **C assumes the following registers are preserved**
  - **EBX, ESI, EDI, EBP, CS, DS, SS, ES**
- **labels (putting an underscore before labels)**
  - **Not needed for linux gcc**
- **CALLING CONVENTIONS ARE DIFFERENT in 64-BIT programming**
  - https://en.wikipedia.org/wiki/X86_calling_conventions#x86-64_calling_conventions
  - https://aaronbloomfield.github.io/pdr/book/x86-64bit-ccc-chapter.pdf

Return values (32-bit)



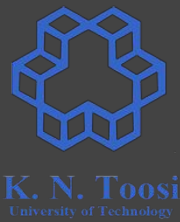
- void functions return nothing
- 8-bit, 16-bit and 32-bit integer values are stored in **`EAX`**
  - bit extension depends on signed/unsigned
- 64 bit integers are stored in **`EDX:EAX`**
- Addresses (pointers) are stored in **`EAX`**
- Floating point values are stored in **`ST0`**
- What else?

Return values (32-bit)



- void functions return nothing
- 8-bit, 16-bit and 32-bit integer values are stored in **`EAX`**
  - bit extension depends on signed/unsigned
- 64 bit integers are stored in **`EDX:EAX`**
- Addresses (pointers) are stored in **`EAX`**
- Floating point values are stored in **`ST0`**
- What else?
  - Structures,

Return values (32-bit)



- void functions return nothing
- 8-bit, 16-bit and 32-bit integer values are stored in **`EAX`**
  - bit extension depends on signed/unsigned
- 64 bit integers are stored in **`EDX:EAX`**
- Addresses (pointers) are stored in **`EAX`**
- Floating point values are stored in **`ST0`**
- What else?
  - Structures,
  - C++ Objects

C calling conventions (64-bit)

- First 6 parameters are (in order) put in
  - Integer, pointer: RDI, RSI, RDX, RCX, R8, R9
  - Floating point: XMM0, XMM1, XMM2, XMM3, XMM4, XMM5
- Additional parameters are pushed on stack in reverse order
- Return value stored in
  - 8, 16, 32, 64 bit integers, pointers: RAX
  - 128 bit integers: RDX:RAX
  - floating points: XMM0 (, XMM1)

C calling conventions (64-bit)

- C assumes the following registers are preserved
  - RBX, RBP, R12, R13, R14, R15
- Microsoft uses a different convention
- Look at
  - https://en.wikipedia.org/wiki/X86_calling_conventions#x86-64_calling_conventions
  - https://aaronbloomfield.github.io/pdr/book/x86-64bit-ccc-chapter.pdf

Remember: Modular Programming

test.c

```c
#include <stdio.h>

extern int fact(int);

extern int maxval;

int main() {
  int x = 8;
  printf("x!=%d\n", fact(x));

  return  0;
}
```

fact.c

```c
int maxval = 2;
static int flag = 1;

int fact(int n) {
  return n==0 ? 1 : n*fact(n-1);
}

static int condmax(int a, int b) {
  return (a > b && flag) ? a : b;
}
```

Remember: Modular Programming

first.asm

```
extern fact, var1

segment .text

    mov eax, [var1]

    push 6
    call fact
    add esp, 4
```

second.asm

```
global fact, var1

segment .data

var1:    dd   22

segment .text

fact:
     ;; factorial function
```

Example1: Calling an assembly routine in C (32-bit)

K. N. Toosi
University of Technology

printsum.c

```c
#include <stdio.h>

int sum(int,int);

int main() {
  int a,b,c;

  scanf("%d %d", &a, &b);

  c = sum(a,b);

  printf("%d\n",c);

  return 0;
}
```

calcsum.asm

```
segment .text
        global sum
sum:
        push ebp
        mov  ebp, esp

        push ebx

        mov  eax, [ebp+8]
        mov  ebx, [ebp+12]
        add  eax, ebx

        pop  ebx

        pop  ebp
        ret
```

How to compile, link, and run (32-bit)

1. Compile the C file to object file
   - `gcc -c -m32 printsum.c`
   - creates `printsum.o`
2. Compile the assembly file to object file
   - `nasm -f elf calcsum.asm`
   - creates `calcsum.o`
3. Linking the object files (and C libraries)
   - `gcc -m32 printsum.o calcsum.o  -o printsum`
4. Running the executable
   - `./printsum`

How to compile, link, and run (32-bit)

1. Compile the C file to object file
   - `gcc -c -m32 printsum.c`
   - creates `printsum.o`
2. Compile the assembly file to object file
   - `nasm -f elf calcsum.asm`
   - creates `calcsum.o`
3. Linking the object files (and C libraries)
   - `gcc -m32 printsum.o cal`
4. Running the executable
   - `./printsum`

```
gcc -c -m32 printsum.c
nasm -f elf calcsum.asm
gcc -m32 printsum.o calcsum.o -o printsum

./printsum
```

How to compile, link, and run (32-bit)

1. Compile the C file to object file
   - `gcc -c -m32 printsum.c`
   - creates `printsum.o`
2. Compile the assembly file to object file
   - `nasm -f elf calcsum.asm`
   - creates `calcsum.o`
3. Linking the object files (and C libraries)
   - `gcc -m32 printsum.o calcsum.o  -o printsum`
4. Running the executable

```
$ gcc -c -m32 printsum.c && nasm -f elf calcsum.asm && gcc -m32
printsum.o calcsum.o  -o printsum && ./printsum
```

Using Makefile

Makefile

```
GCC_OPTIONS= -m32
NASM_OPTIONS= -f elf

printsum: printsum.o calcsum.o
        gcc $(GCC_OPTIONS) -o printsum printsum.o calcsum.o

printsum.o: printsum.c
        gcc -c $(GCC_OPTIONS) printsum.c

calcsum.o: calcsum.asm
        nasm $(NASM_OPTIONS) calcsum.asm
```

Using Makefile

K. N. Toosi University of Technology

printsum.c

```c
#include <stdio.h>

int sum(int,int);

int main() {
  int a,b,c;

  scanf("%d %d", &a, &b);

  c = sum(a,b);

  printf("%d\n

  return 0;
}
```

calcsum.asm

```
segment .text
        global sum
sum:
        push ebp
        mov  ebp, esp

        push ebx

        mov  eax, [ebp+8]
        mov  ebx, [ebp+12]
```

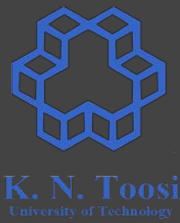
```
b.nasihatkon@kntu:example1$ make
gcc -c -m32 printsum.c
nasm -f elf calcsum.asm
gcc -m32 -o printsum printsum.o calcsum.o
b.nasihatkon@kntu:example1$ ./printsum
-4
12
8
```

Optimizing the code

calcsum.asm

```
segment .text
        global sum
sum:
        push ebp
        mov  ebp, esp

        push ebx

        mov  eax, [ebp+8]
        mov  ebx, [ebp+12]
        add  eax, ebx

        pop  ebx

        pop  ebp
        ret
```

Optimizing the code



calcsum.asm

```
segment .text
        global sum
sum:
        push ebp
        mov  ebp, esp

        push ebx

        mov  eax, [ebp+8]
        mov  ebx, [ebp+12]
        add  eax, ebx

        pop  ebx

        pop  ebp
        ret
```

calcsum.asm

```
segment .text
        global sum
sum:




        mov  eax, [esp+4]
        add  eax, [esp+8]




        ret
```

Example1: 64-bit version

printsum.c

```c
#include <stdio.h>

int sum(int,int);

int main() {
  int a,b,c;

  scanf("%d %d", &a, &b);

  c = sum(a,b);

  printf("%d\n",c);

  return 0;
}
```

calcsum.asm

```
segment .text

        global sum
sum:
        mov rax, rdi
        add rax, rsi

        ret
```

Makefile (64-bit)

Makefile

```
GCC_OPTIONS= -m64
NASM_OPTIONS= -f elf64

printsum: printsum.o calcsum.o
	gcc $(GCC_OPTIONS) -o printsum printsum.o calcsum.o

printsum.o: printsum.c
	gcc -c $(GCC_OPTIONS) printsum.c

calcsum.o: calcsum.asm
	nasm $(NASM_OPTIONS) calcsum.asm
```

Example2: Calling a C routine in assembly

main.asm

```
segment .text
    extern sum, print_sint, print_uint, print_hex
    global main

main:
      push 1
      push -2
      call sum
      add esp, 8

      push eax
      call print_sint
      call print_uint
      call print_hex
      add esp, 4

      mov ebx, 0
      mov eax, 1
      int 0x80
```

mytools.c

```
#include <stdio.h>

int sum(int a, int b) {
  return a+b;
}

void print_sint(int a) {
  printf("%d\n", a);
}

void print_uint(int a) {
  printf("%u\n", a);
}

void print_hex(int a) {
  printf("%x\n", a);
}
```

Example2: Calling a C routine in assembly

main.asm

```
segment .text
    extern sum, print_sint, print_uint, print_hex
    global main

main:                          ← because we use GCC to link
    push 1
    push -2
    call sum
    add esp, 8

    push eax
    call print_sint
    call print_uint
    call print_hex
    add esp, 4

    mov ebx, 0
    mov eax, 1                 ← Exit system call (32-bit, linux)
    int 0x80
```

mytools.c

```
#include <stdio.h>

int sum(int a, int b) {
  return a+b;
}

void print_sint(int a) {
  printf("%d\n", a);
}

void print_uint(int a) {
  printf("%u\n", a);
}

void print_hex(int a) {
  printf("%x\n", a);
}
```

Example2: Calling a C routine in assembly

main.asm

```
segment .text
    extern sum, print_sint, print_uint, print_hex
    global main

main:
      push 1
      push -2
      call sum
      add esp, 8

      push eax
      call print_sint
      call print_uint
      call print_hex
      add esp, 4

      mov ebx, 0
      mov eax, 1
      int 0x80
```

```
nasm -f elf main.asm

gcc -c -m32 mytools.c

gcc -m32 main.o mytools.o -o main

./main
```

mytools.c

```
#include <stdio.h>

int sum(int a, int b) {
  return a+b;
}

void print_sint(int a) {
    printf("%d\n", a);

void print_uint(int a) {
    printf("%u\n", a);

void print_hex(int a) {
    printf("%x\n", a);
```

Example2: Calling a C routine in assembly

main.asm

```
segment .text
    extern sum, print_sint, print_uint, print_hex
    global main

main:
        push 1
        push 2
        push eax
        call print_sint
        call print_uint
        call print_hex
        add esp, 4

        mov ebx, 0
        mov eax, 1
        int 0x80
```

mytools.c

```
#include <stdio.h>

int sum(int a, int b) {
  return a+b;
}

void print_uint(int a) {
  printf("%u\n", a);
}

void print_hex(int a) {
  printf("%x\n", a);
}
```

```
nasm -f elf main.asm && gcc -c -m32 mytools.c && gcc -m32 main.o mytools.o -o main && ./main
```

Compile using Makefile

Makefile

```
GCC_OPTIONS= -m32
NASM_OPTIONS= -f elf

main: mytools.o main.o
    gcc $(GCC_OPTIONS) -o main mytools.o main.o

mytools.o: mytools.c
    gcc -c $(GCC_OPTIONS) mytools.c

main.o: main.asm
    nasm $(NASM_OPTIONS) main.asm
```

Compile using Makefile

Makefile

```
GCC_OPTIONS= -m32
NASM_OPTIONS= -f elf

main: mytools.o main.o
    gcc $(GCC_OPTIONS) -o main mytools.o main.o

mytools.o: mytools.c
    gcc -c $(GCC_OPTIONS) mytools.c

main.o: main.asm
    nasm $(NASM_OPTIONS) main.asm
```

```
b.nasihatkon@kntu:example2$ ls
main.asm  Makefile  mytools.c
b.nasihatkon@kntu:example2$
b.nasihatkon@kntu:example2$ make
gcc -c -m32 mytools.c
nasm -f elf main.asm
gcc -m32 -o main mytools.o main.o
b.nasihatkon@kntu:example2$
b.nasihatkon@kntu:example2$ ./main
-1
4294967295
ffffffff
```

Example2: 64-bit version

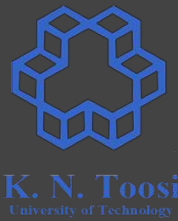
main.asm

```
segment .text
   extern sum, print_sint, print_uint, print_hex
   global main

main:
        mov rdi, 1
        mov rsi,-2
        call sum

        mov rbx, rax

        mov rdi, rbx
        call print_sint

        mov rdi, rbx
        call print_uint

        mov rdi, rbx
        call print_hex

        mov rdi, 0
        mov rax, 60
        syscall
```

mytools.c

```
#include <stdio.h>

int sum(int a, int b) {
  return a+b;
}

void print_sint(int a) {
  printf("%d\n", a);
}

void print_uint(int a) {
  printf("%u\n", a);
}

void print_hex(int a) {
  printf("%x\n", a);
}
```

Makefile (64-bit)



Makefile

```
GCC_OPTIONS= -m64
NASM_OPTIONS= -f elf64

main: mytools.o main.o
   gcc $(GCC_OPTIONS) -o main mytools.o main.o

mytools.o: mytools.c
   gcc -c $(GCC_OPTIONS) mytools.c

main.o: main.asm
   nasm $(NASM_OPTIONS) main.asm
```

Example3: Calling C Standard Library functions

Write an assembly program equivalent to the following C program. Call functions scanf, abs and printf from the C standard library.

callstdlib.c

```c
#include <stdio.h>
#include <stdlib.h>

int a;

int main() {

  scanf("%d", &a);

  printf("|%d| = %d\n", a, abs(a));

  return 0;
}
```

```c
int scanf(const char *format, ...);

int printf(const char *format, ...);

int abs(int j);
```

Example3: Calling C stdlib functions

callstdlib.c

```c
#include <stdio.h>
#include <stdlib.h>

int a;

int main() {

  scanf("%d", &a);

  printf("|%d| = %d\n", a, abs(a));

  return 0;
}
```

callstdlib.asm

```
segment .data
  a:       dd    0
  format1:  db   "%d", 0
  format2:  db   "|%d| = %d", 10, 0

segment .text
  extern abs, scanf, printf
  global main

main:
      push a
      push format1
      call scanf
      add esp, 8

      push dword [a]
      call $abs
      add esp, 4
```

callstdlib.asm (cont.)

```
      push eax
      push dword [a]
      push format2
      call printf
      add  esp, 12

      mov eax, 1
      int 0x80
```

Example3: Calling C stdlib functions

callstdlib.c

```c
#include <stdio.h>
#include <stdlib.h>

int a;

int main() {

  scanf("%d", &a);

  printf("|%d| = %d\n", a, abs(a));

  return 0;
}
```

callstdlib.asm

```
segment .data
  a:       dd    0
  format1:  db   "%d", 0
  format2:  db   "|%d| = %d", 10, 0

segment .text
  extern abs, scanf, printf
  global main

main:
      push a
      push format1
      call scanf
      add esp, 8

      push dword [a]
      call $abs
      add esp, 4
```

"$" because "abs" is an NASM keyword

callstdlib.asm (cont.)

```
      push eax
      push dword [a]
      push format2
      call printf
      add  esp, 12

      mov eax, 1
      int 0x80
```

Example3: Calling C stdlib functions

callstdlib.c

```c
#include <stdio.h>
#include <stdlib.h>

int a;

int main() {

  scanf("%d", &a);

  printf("|%d| = %d\n", a, abs(a));

  return 0;
}
```

callstdlib.asm

```
segment .data
  a:       dd    0
  format1:  db   "%d", 0
  format2:  db   "|%d| = %d", 10, 0

segment .text
  extern abs, scanf, printf
  global main

main:
      push a
      push format1
      call scanf
      add esp, 8

      push dword [a]
      call $abs
      add esp, 4
```

callstdlib.asm (cont.)

```
      push eax
      push dword [a]
      push format2
      call printf
      add  esp, 12

      mov eax, 1
      int 0x80
```

why not include stdio, stdlib?

Example3: Calling C stdlib functions

callstdlib.asm

```
segment .data
  a:       dd    0
  format1:  db   "%d", 0
  format2:  db   "|%d| = %d", 10, 0

segment .text
  extern abs, scanf, printf
  global main

main:
      push a
```

callstdlib.asm (cont.)

```
      push eax
      push dword [a]
      push format2
      call printf
      add  esp, 12

      mov eax, 1
      int 0x80
```

```
nasm -f elf callstdlib.asm    # compile assembly -> callstdlib.o

gcc -m32 callstdlib.o  -o callstdlib    # link (with C libraries) -> callstdlib

./callstdlib  # execute
```

Example 3: Compile using Makefile

Makefile

```
GCC_OPTIONS= -m32
NASM_OPTIONS= -f elf

callstdlib: callstdlib.o
        gcc $(GCC_OPTIONS) -o callstdlib callstdlib.o

callstdlib.o: callstdlib.asm
        nasm $(NASM_OPTIONS) callstdlib.asm
```

Look at asm_io.asm

asm_io.asm

```
int_format:     db  "%i", 0
string_format:  db  "%s", 0
```

```
global  read_int, print_int, print_uint, print_string, read_char
global  print_char, print_nl, sub_dump_regs, sub_dump_mem
global  sub_dump_math, sub_dump_stack
extern  _scanf, _printf, _getchar, _putchar
```

```
read_int:
        enter   4,0
        pusha
        pushf

        lea     eax, [ebp-4]
        push    eax
        push    dword int_format
        call    _scanf
        pop     ecx
        pop     ecx

        popf
        popa
        mov     eax, [ebp-4]
        leave
        ret
```

```
print_int:
        enter   0,0
        pusha
        pushf

        push    eax
        push    dword int_format
        call    _printf
        pop     ecx
        pop     ecx

        popf
        popa
        leave
        ret
```

```
print_string:
        enter   0,0
        pusha
        pushf

        push    eax
        push    dword string_format
        call    _printf
        pop     ecx
        pop     ecx

        popf
        popa
        leave
        ret
```

Look at asm_io.asm

asm_io.asm

```
int_format:     db  "%i", 0
string_format:  db  "%s", 0
```

```
global  read_int, print_int, print_uint, print_string, read_char
global  print_char, print_nl, sub_dump_regs, sub_dump_mem
        ack
        ar
```

why the underscores?

```
read_int:
        enter   4,0
        pusha
        pushf

        lea     eax, [ebp-4]
        push    eax
        push    dword int_format
        call    _scanf
        pop     ecx
        pop     ecx

        popf
        popa
        mov     eax, [ebp-4]
        leave
        ret
```

```
        pusha
        pushf

        push    eax
        push    dword int_format
        call    _printf
        pop     ecx
        pop     ecx

        popf
        popa
        leave
        ret
```

```
print_string:
        enter   0,0
        pusha
        pushf

        push    eax
        push    dword string_format
        call    _printf
        pop     ecx
        pop     ecx

        popf
        popa
        leave
        ret
```

Look at asm_io.asm

asm_io.asm

```
int_format:     db  "%i", 0
string_format:  db  "%s", 0
```

```
global  read_int, print_int, print_uint, print_string, read_char
global  print_char, print_nl, sub_dump_regs, sub_dump_mem
```

```
%ifdef ELF_TYPE
  %define _scanf    scanf
  %define _printf   printf
  %define _getchar getchar
  %define _putchar putchar
%endif
```

```
read_int:
    enter   4,0
    pusha
    pushf

    lea     eax, [ebp-4]
    push    eax
    push    dword int_format
    call    _scanf
    pop     ecx
    pop     ecx

    popf
    popa
    mov     eax, [ebp-4]
    leave
    ret
```

```
    push    eax
    push    dword int_format
    call    _printf
    pop     ecx
    pop     ecx

    popf
    popa
    leave
    ret
```

```
print_string:
    enter   0,0
    pusha
    pushf

    push    eax
    push    dword string_format
    call    _printf
    pop     ecx
    pop     ecx

    popf
    popa
    leave
    ret
```