3_2 Program Encodings

- Suppose we write a C program as two files p1.c and p2.c. We can then compile this code using a Unix command line:

```
linux> gcc -Og -o p p1.c p2.c
```

  - gcc: indicates the gcc C compiler. It's the default compiler on Linux.
  - -Og: instructs the compiler to apply a level of optimization that yields machine code that follows the overall structure of the original C code.
    - higher levels of optimization can generate code that is so heavily transformed that the relationship between the generated machine code and the original source code is difficult to understand.
    - -Og optimization as a learning tool and then see what happens as we increase the level of optimization.
    - In practice, higher levels of optimization are considered a better choice in terms of the resulting program performance.
  - -o: output.

- The gcc command invokes an entire sequence of programs to turn the source code into executable code:
  1. C preprocessor expands the source code to include any files specified with `#include` commands and to expand any macros, specified with `#define` declarations.
  2. The compiler generates assembly code versions of the two source files having names p1.s and p2.s.
  3. The assembler converts the assembly code into binary object-code files p1.o and p2.o.
  4. The linker merges these two object-code files along with code implementing library functions and generates the final executable code file p.

- Executable code is the second form of machine code we will consider—it is the exact form of code that is executed by the processor.

3.2.1 Machine-Level Code

- Computer systems employ several different forms of abstraction, hiding details of an implementation through the use of a simpler abstract model. Two of these are especially important for machine-level programming:
  - The format and behavior of a machine-level program is defined by the instruction set architecture, or ISA, defining the processor state, the format of the instructions, and the effect each of these instructions will have on the state.
  - The memory addresses used by a machine-level program are virtual addresses, providing a memory model that appears to be a very large byte array.

Machine Instructions

- The compiler does most of the work in the overall compilation sequence, transforming programs expressed in the relatively abstract execution model provided by C into the very elementary instructions that the processor executes.
- The assembly-code representation is very close to machine code.
- The machine code for x86-64 differs greatly from the original C code. Parts of the processor state are visible that normally are hidden from the C programmer:

    - program counter: PC is called %rip in x86-64, which indicates the address in memory of the **next** instruction to be executed.
    - integer register file: contains 16 named locations storing 64-bit values.
        - The registers can hold addresses (corresponding to C pointers) or integer data.
    - condition code registers: hold status information about the most recently executed arithmetic or logical instruction.
        - used to implement conditional changes in the control or data flow, such as is required to implement if and while statements.
    - vector registers: each hold one or more integer or floating-point values.
- A single machine instruction performs only a very elementary operation.

Memory

- Whereas C provides a model in which objects of different data types can be declared and allocated in memory, machine code views the memory as simply a large byte-addressable array.
    - Aggregate data types in C such as arrays and structures are represented in machine code as contiguous collections of bytes.
    - Even for scalar data types, assembly code makes no distinctions between signed or unsigned integers, between different types of pointers, or even between pointers and integers.
- The program memory contains:
    - the executable machine code for the program,
    - some information required by the operating system,
    - a run-time stack for managing procedure calls and returns,
    - blocks of memory allocated by the user.
- The program memory is addressed using virtual addresses. At any given time, only limited subranges of virtual addresses are considered valid.
    - Example: x86-64 virtual addresses are represented by 64-bit words. In current implementations of these machines, the upper 16 bits must be set to zero, and so an address can potentially specify a byte over a range of 2^{48}, or 64 terabytes.
- The operating system manages this virtual address space, **translating virtual addresses into the physical addresses** of values in the actual processor memory.

3.2.2 Code Examples

- Suppose a C code file mstore.c containing the following function definition:

```
long mult2(long, long);

void multstore(long x, long y, long *dest) {
    long t = mult2(x, y);
    *dest = t;
}
```

- Input the command line:

```
linux> gcc -Og -S mstore.c
```

```
parallels@ubuntu-linux-22-04-desktop:/media/psf/csapp/3_2 Program Encodings.assets$ gcc -Og -S P
200.c
```

- It generate P200.s file and open it (because my pc is macos, I have to install cross compiler based on arm cpu.):

```
parallels@ubuntu-linux-22-04-desktop:/media/psf/csapp/3_2 Program Encodings.assets$ x86_64-linux
-gnu-gcc -Og -S P200.c
parallels@ubuntu-linux-22-04-desktop:/media/psf/csapp/3_2 Program Encodings.assets$ ls
P200.c  P200.s
parallels@ubuntu-linux-22-04-desktop:/media/psf/csapp/3_2 Program Encodings.assets$ vim P200.s
```

```
        .file   "P200.c"
        .text
        .globl  multstore
        .type   multstore, @function
multstore:
.LFB0:
        .cfi_startproc
        endbr64
        pushq   %rbx
        .cfi_def_cfa_offset 16
        .cfi_offset 3, -16
        movq    %rdx, %rbx
        call    mult2@PLT
        movq    %rax, (%rbx)
        popq    %rbx
        .cfi_def_cfa_offset 8
        ret
        .cfi_endproc
.LFE0:
        .size   multstore, .-multstore
        .ident  "GCC: (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0"
        .section        .note.GNU-stack,"",@progbits
        .section        .note.gnu.property,"a"
        .align 8
        .long   1f - 0f
        .long   4f - 1f
        .long   5
0:
        .string "GNU"
1:
        .align 8
        .long   0xc0000002
        .long   3f - 2f
2:
        .long   0x3
3:
        .align 8
4:
```

- To extract the core code:

```
        .cfi_startproc
        endbr64
        pushq   %rbx
        .cfi_def_cfa_offset 16
        .cfi_offset 3, -16
        movq    %rdx, %rbx
        call    mult2@PLT
        movq    %rax, (%rbx)
        popq    %rbx
        .cfi_def_cfa_offset 8
        ret
        .cfi_endproc
```

- CFI – Call Frame Information. Based on that name, all the assembler directives begin with `.cfi_`.
  - `.cfi_startproc` – the start of every annotated function. It causes a new CFI table for this function to be initialized.

- `.cfi_def_cfa_offset 16` - offset 16 bytes to CFA. `.cfi_def_cfa_offset` modifies a rule for computing CFA. Register remains the same, but offset is new. Note that it is the absolute offset that will be added to a defined register to compute CFA address.
- `.cfi_offset 3, -16` - `.cfi_offset register, offset` Previous value of register is saved at offset offset from CFA.
- `.cfi_endproc` - trigger the CFI table to be emitted.
- Canonical Frame Address (CFA). This is the value of the stack pointer just before the CALL instruction in the parent function (RSP value before CALL).

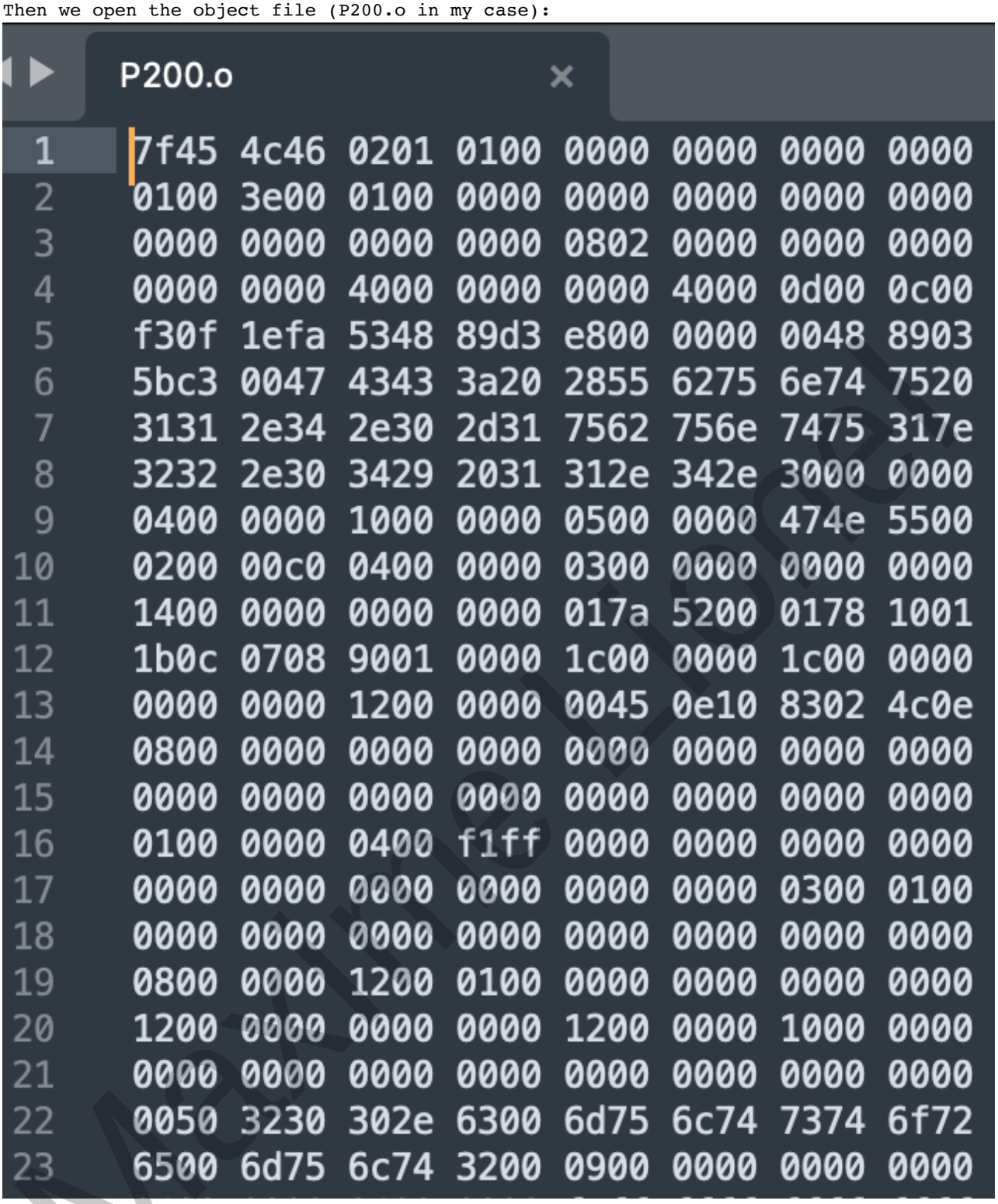
```
        endbr64
        pushq   %rbx
        movq    %rdx, %rbx
        call    mult2@PLT
        movq    %rax, (%rbx)
        popq    %rbx
        ret
```

- We just ignore the CFI information first. Go through the instructions one by one:
  - `pushq %rbx` - the contents of register %rbx is pushed onto the program stack.
  - `movq %rdx, %rbx` - move the content of register %rdx to %rbx%.
  - `call mult2@PLT` - call function mult2.
  - `movq %rax, (%rbx)` - move the content of register %rdx to the address pointed by content of %rbx%.
  - `popq %rbx` - pop the top of the stack into register %rbx
- If we use the -c command-line option, gcc will both compile and assemble the code:

```
        gcc -Og -c mstore.c
For me, the command is like:
        x86_64-linux-gnu-gcc -Og -c P200.c
```

```
![[image-20240307172148993.png]]
```

- Then we open the object file (P200.o in my case):



- To inspect the contents of machine-code files, a class of programs known as disassemblers (objdump):

```
        objdump -d mstore.o
For myself, the command is like:
        x86_64-linux-gnu-objdmp -d P200.o
```

```
![[image-20240308150438282.png]]

```

-gnu-objdump -d P200.o

P200.o: file format elf64-x86-64
```
```

```
Disassembly of section .text:

0000000000000000 <multstore>:
   0:   f3 0f 1e fa             endbr64
   4:   53                      push   %rbx
   5:   48 89 d3                mov    %rdx,%rbx
   8:   e8 00 00 00 00          call   d <multstore+0xd>
   d:   48 89 03                mov    %rax,(%rbx)
  10:   5b                      pop    %rbx
  11:   c3                      ret
```

- endbr64 - End Branch 64 Bit. For details, please refer to chapter 17 of intel sw manual. It's the Control-flow Enforcement Technology for security.

```
IF EndbranchEnabled(CPL) & EFER.LMA = 1 & CS.L = 1
  IF CPL = 3
  THEN
    IA32_U_CET.TRACKER = IDLE
    IA32_U_CET.SUPPRESS = 0
  ELSE
    IA32_S_CET.TRACKER = IDLE
    IA32_S_CET.SUPPRESS = 0
  FI
FI;
```

CHAPTER 17
CONTROL-FLOW ENFORCEMENT TECHNOLOGY (CET)

17.1 INTRODUCTION

Return-oriented programming (ROP), and similarly CALL/JMP-oriented programming (COP/JOP), have been the prevalent attack methodologies for stealth exploit writers targeting vulnerabilities in programs. These attack methodologies have the common elements:

- A code module with execution privilege and contain small snippets of code sequence with the characteristic: at least one instruction in the sequence being a control transfer instruction that depends on data either in the return stack or in a register for the target address.
- Diverting the control flow instruction (e.g., RET, CALL, JMP) from its original target address to a new target (via modification in the data stack or in the register).

Control-Flow Enforcement Technology (CET) provides the following capabilities to defend against ROP/COP/JOP style control-flow subversion attacks:

- Shadow stack: Return address protection to defend against ROP.
- Indirect branch tracking: Free branch protection to defend against COP/JOP.

Both capabilities introduce new instruction set extensions, and are described in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C, & 2D.

Control-Flow Enforcement Technology introduces a new exception (#CP) with interrupt vector 21.

- x86-64 instructions can range in length from 1 to 15 bytes.
- there is a unique decoding of the bytes into machine instructions.
- For example, only the instruction pushq %rbx can start with byte value 53.
- The disassembler determines the assembly code based purely on the byte sequences in the machine-code file.
- The disassembler uses a slightly different naming convention for the instruc- tions than does the assembly code generated by gcc.

- Now let's compile another c file (main function):

```c
#include <stdio.h>
void multstore(long, long, long *);
int main() {
    long d;
    multstore(2, 3, &d);
    printf("2 * 3 --> %ld\n", d);
    return 0;
    }
```

```c
long mult2(long a, long b) {
    long s = a * b;
    return s;
}
```

```
parallels@ubuntu-linux-22-04-desktop:/media/psf/csapp/3_2 Program Encodings.assets$ x86_64-linux-gnu-objdump -d prog

prog:     file format elf64-x86-64


Disassembly of section .init:

0000000000001000 <_init>:
    1000:       f3 0f 1e fa             endbr64
    1004:       48 83 ec 08             sub    $0x8,%rsp
    1008:       48 8b 05 d9 2f 00 00    mov    0x2fd9(%rip),%rax        # 3fe8 <__gmon_start__@Base>
    100f:       48 85 c0                test   %rax,%rax
    1012:       74 02                   je     1016 <_init+0x16>
    1014:       ff d0                   call   *%rax
    1016:       48 83 c4 08             add    $0x8,%rsp
    101a:       c3                      ret

Disassembly of section .plt:
0000000000001020 <.plt>:
    1020:       ff 35 92 2f 00 00       push   0x2f92(%rip)        # 3fb8 <_GLOBAL_OFFSET_TABLE_+0x8>
    1026:       f2 ff 25 93 2f 00 00    bnd jmp *0x2f93(%rip)        # 3fc0 <_GLOBAL_OFFSET_TABLE_+0x10>
    102d:       0f 1f 00                nopl   (%rax)
    1030:       f3 0f 1e fa             endbr64
    1034:       68 00 00 00 00          push   $0x0
    1039:       f2 e9 e1 ff ff ff       bnd jmp 1020 <_init+0x20>
    103f:       90                      nop
    1040:       f3 0f 1e fa             endbr64
    1044:       68 01 00 00 00          push   $0x1
    1049:       f2 e9 d1 ff ff ff       bnd jmp 1020 <_init+0x20>
    104f:       90                      nop

Disassembly of section .plt.got:
0000000000001050 <__cxa_finalize@plt>:
    1050:       f3 0f 1e fa             endbr64
    1054:       f2 ff 25 9d 2f 00 00    bnd jmp *0x2f9d(%rip)        # 3ff8 <__cxa_finalize@GLIBC_2.2.5>
    105b:       0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)
```

```
Disassembly of section .plt.sec:

0000000000001060 <__stack_chk_fail@plt>:
    1060:       f3 0f 1e fa             endbr64
    1064:       f2 ff 25 5d 2f 00 00    bnd jmp *0x2f5d(%rip)        # 3fc8
<__stack_chk_fail@GLIBC_2.4>
    106b:       0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)

0000000000001070 <__printf_chk@plt>:
    1070:       f3 0f 1e fa             endbr64
    1074:       f2 ff 25 55 2f 00 00    bnd jmp *0x2f55(%rip)        # 3fd0
<__printf_chk@GLIBC_2.3.4>
    107b:       0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)

Disassembly of section .text:
0000000000001080 <_start>:
    1080:       f3 0f 1e fa             endbr64
    1084:       31 ed                   xor    %ebp,%ebp
    1086:       49 89 d1                mov    %rdx,%r9
    1089:       5e                      pop    %rsi
    108a:       48 89 e2                mov    %rsp,%rdx
    108d:       48 83 e4 f0             and    $0xfffffffffffffff0,%rsp
    1091:       50                      push   %rax
    1092:       54                      push   %rsp
    1093:       45 31 c0                xor    %r8d,%r8d
    1096:       31 c9                   xor    %ecx,%ecx
    1098:       48 8d 3d dc 00 00 00    lea    0xdc(%rip),%rdi        # 117b <main>
    109f:       ff 15 33 2f 00 00       call   *0x2f33(%rip)        # 3fd8
<__libc_start_main@GLIBC_2.34>
    10a5:       f4                      hlt
    10a6:       66 2e 0f 1f 84 00 00    cs nopw 0x0(%rax,%rax,1)
    10ad:       00 00 00

00000000000010b0 <deregister_tm_clones>:
    10b0:       48 8d 3d 59 2f 00 00    lea    0x2f59(%rip),%rdi        # 4010 <__TMC_END__>
    10b7:       48 8d 05 52 2f 00 00    lea    0x2f52(%rip),%rax        # 4010 <__TMC_END__>
    10be:       48 39 f8                cmp    %rdi,%rax
    10c1:       74 15                   je     10d8 <deregister_tm_clones+0x28>
    10c3:       48 8b 05 16 2f 00 00    mov    0x2f16(%rip),%rax        # 3fe0
<_ITM_deregisterTMCloneTable@Base>
    10ca:       48 85 c0                test   %rax,%rax
    10cd:       74 09                   je     10d8 <deregister_tm_clones+0x28>
    10cf:       ff e0                   jmp    *%rax
    10d1:       0f 1f 80 00 00 00 00    nopl   0x0(%rax)
    10d8:       c3                      ret
    10d9:       0f 1f 80 00 00 00 00    nopl   0x0(%rax)
00000000000010e0 <register_tm_clones>:
    10e0:       48 8d 3d 29 2f 00 00    lea    0x2f29(%rip),%rdi        # 4010 <__TMC_END__>
    10e7:       48 8d 35 22 2f 00 00    lea    0x2f22(%rip),%rsi        # 4010 <__TMC_END__>
    10ee:       48 29 fe                sub    %rdi,%rsi
    10f1:       48 89 f0                mov    %rsi,%rax
    10f4:       48 c1 ee 3f             shr    $0x3f,%rsi
    10f8:       48 c1 f8 03             sar    $0x3,%rax
    10fc:       48 01 c6                add    %rax,%rsi
    10ff:       48 d1 fe                sar    %rsi
    1102:       74 14                   je     1118 <register_tm_clones+0x38>
    1104:       48 8b 05 e5 2e 00 00    mov    0x2ee5(%rip),%rax        # 3ff0
<_ITM_registerTMCloneTable@Base>
```

```
    110b:       48 85 c0                test   %rax,%rax
    110e:       74 08                   je     1118 <register_tm_clones+0x38>
    1110:       ff e0                   jmp    *%rax
    1112:       66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)
    1118:       c3                      ret
    1119:       0f 1f 80 00 00 00 00    nopl   0x0(%rax)
0000000000001120 <__do_global_dtors_aux>:
    1120:       f3 0f 1e fa             endbr64
    1124:       80 3d e5 2e 00 00 00    cmpb   $0x0,0x2ee5(%rip)        # 4010 <__TMC_END__>
    112b:       75 2b                   jne    1158 <__do_global_dtors_aux+0x38>
    112d:       55                      push   %rbp
    112e:       48 83 3d c2 2e 00 00    cmpq   $0x0,0x2ec2(%rip)        # 3ff8
<__cxa_finalize@GLIBC_2.2.5>
    1135:       00
    1136:       48 89 e5                mov    %rsp,%rbp
    1139:       74 0c                   je     1147 <__do_global_dtors_aux+0x27>
    113b:       48 8b 3d c6 2e 00 00    mov    0x2ec6(%rip),%rdi        # 4008 <__dso_handle>
    1142:       e8 09 ff ff ff          call   1050 <__cxa_finalize@plt>
    1147:       e8 64 ff ff ff          call   10b0 <deregister_tm_clones>
    114c:       c6 05 bd 2e 00 00 01    movb   $0x1,0x2ebd(%rip)        # 4010 <__TMC_END__>
    1153:       5d                      pop    %rbp
    1154:       c3                      ret
    1155:       0f 1f 00                nopl   (%rax)
    1158:       c3                      ret
    1159:       0f 1f 80 00 00 00 00    nopl   0x0(%rax)
0000000000001160 <frame_dummy>:
    1160:       f3 0f 1e fa             endbr64
    1164:       e9 77 ff ff ff          jmp    10e0 <register_tm_clones>

0000000000001169 <multstore>:
    1169:       f3 0f 1e fa             endbr64
    116d:       53                      push   %rbx
    116e:       48 89 d3                mov    %rdx,%rbx
    1171:       e8 68 00 00 00          call   11de <mult2>
    1176:       48 89 03                mov    %rax,(%rbx)
    1179:       5b                      pop    %rbx
    117a:       c3                      ret

000000000000117b <main>:
    117b:       f3 0f 1e fa             endbr64
    117f:       48 83 ec 18             sub    $0x18,%rsp
    1183:       64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
    118a:       00 00
    118c:       48 89 44 24 08          mov    %rax,0x8(%rsp)
    1191:       31 c0                   xor    %eax,%eax
    1193:       48 89 e2                mov    %rsp,%rdx
    1196:       be 03 00 00 00          mov    $0x3,%esi
    119b:       bf 02 00 00 00          mov    $0x2,%edi
    11a0:       e8 c4 ff ff ff          call   1169 <multstore>
    11a5:       48 8b 14 24             mov    (%rsp),%rdx
    11a9:       48 8d 35 54 0e 00 00    lea    0xe54(%rip),%rsi        # 2004 <_IO_stdin_used+0x4>
    11b0:       bf 01 00 00 00          mov    $0x1,%edi
    11b5:       b8 00 00 00 00          mov    $0x0,%eax
    11ba:       e8 b1 fe ff ff          call   1070 <__printf_chk@plt>
    11bf:       48 8b 44 24 08          mov    0x8(%rsp),%rax
    11c4:       64 48 2b 04 25 28 00    sub    %fs:0x28,%rax
    11cb:       00 00
    11cd:       75 0a                   jne    11d9 <main+0x5e>
```

```
    11cf:       b8 00 00 00 00          mov    $0x0,%eax
    11d4:       48 83 c4 18             add    $0x18,%rsp
    11d8:       c3                      ret
    11d9:       e8 82 fe ff ff          call   1060 <__stack_chk_fail@plt>

00000000000011de <mult2>:
    11de:       f3 0f 1e fa             endbr64
    11e2:       48 89 f8                mov    %rdi,%rax
    11e5:       48 0f af c6             imul   %rsi,%rax
    11e9:       c3                      ret

Disassembly of section .fini:

00000000000011ec <_fini>:
    11ec:       f3 0f 1e fa             endbr64
    11f0:       48 83 ec 08             sub    $0x8,%rsp
    11f4:       48 83 c4 08             add    $0x8,%rsp
    11f8:       c3                      ret
```

- Now we focus on multstore function:

```
// the new version of compiling with main.c
0000000000001169 <multstore>:
    1169:       f3 0f 1e fa             endbr64
    116d:       53                      push   %rbx
    116e:       48 89 d3                mov    %rdx,%rbx
    1171:       e8 68 00 00 00          call   11de <mult2>
    1176:       48 89 03                mov    %rax,(%rbx)
    1179:       5b                      pop    %rbx
    117a:       c3                      ret

// the original version of separated compiling
0000000000000000 <multstore>:
           0:   f3 0f 1e fa             endbr64
           4:   53                      push   %rbx
           5:   48 89 d3                mov    %rdx,%rbx
           8:   e8 00 00 00 00          call   d <multstore+0xd>
           d:   48 89 03                mov    %rax,(%rbx)
          10:   5b                      pop    %rbx
          11:   c3                      ret
```

- Compare with the original one:
  - The addresses listed along the left are different — the linker has shifted the location of this code to a different range of addresses.
  - the linker has filled in the address that the callq instruction should use in calling the function mult2 (line 4 of the disassembly).
    - `e8 68 00 00 00 call 11de <mult2>` and `e8 00 00 00 00 call d <multstore+0xd>`
    - One task for the linker is to match function calls with the locations of the executable code for those functions.

3.2.3 Notes on Formatting

- The assembly code generated by gcc is difficult for a human to read.

- P200.s

```
        .file   "P200.c"
        .text
        .globl  multstore
        .type   multstore, @function
multstore:
.LFB0:
        .cfi_startproc
        endbr64
        pushq   %rbx
        .cfi_def_cfa_offset 16
        .cfi_offset 3, -16
        movq    %rdx, %rbx
        call    mult2@PLT
        movq    %rax, (%rbx)
        popq    %rbx
        .cfi_def_cfa_offset 8
        ret
        .cfi_endproc
.LFE0:
        .size   multstore, .-multstore
        .ident  "GCC: (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0"
        .section        .note.GNU-stack,"",@progbits
        .section        .note.gnu.property,"a"
        .align 8
        .long   1f - 0f
        .long   4f - 1f
        .long   5
0:
        .string "GNU"
1:
        .align 8
        .long   0xc0000002
        .long   3f - 2f
2:
        .long   0x3
3:
        .align 8
4:
```

- All of the lines beginning with '.' are directives to guide the assembler and linker, which we can ignore directly.
- To provide a clearer presentation of assembly code, we will show it in a form that omits most of the directives, while including line numbers and explanatory annotations. Then the code will be:

```
;  void multstore(long x, long y, long *dest)
;   x in %rdi, y in %rsi, dest in %rdx
multstore:
        endbr64    ; CET for security
        pushq   %rbx. ; save rbx to stack
        movq    %rdx, %rbx ; copy the content of rdx to rbx
        call    mult2@PLT ; call function multi2(x.y)
        movq    %rax, (%rbx) ; store the result to the address pointed by rbx
```

```
        popq    %rbx ; restore rbx
        ret ; return
```

- For some applications, the programmer must drop down to assembly code to access low-level features of the machine.
  - One approach is to write entire functions in assembly code and combine them with C functions during the linking stage.
  - A second is to use gcc's support for embedding assembly code directly within C programs.

ATT & Intel assembly-code formats

- ATT format - the default format for gcc, objdump, and the other tools we will consider.
- Intel format - Microsoft, Intel documents.
- If we use the command below:

```
        x86_64-linux-gnu-gcc -Og -S -masm=intel P200.c
```

```
parallels@ubuntu-linux-22-04-desktop:/media/psf/csapp/3_2 Program Encodings.assets$ x86_64-linux
-gnu-gcc -Og -S -masm=intel P200.c
```

- The P200.s result will be like:

```
multstore:
        endbr64
        push    rbx
        mov     rbx, rdx
        call    mult2@PLT
        mov     QWORD PTR [rbx], rax
        pop     rbx
        ret
```

- Differences between ATT format and Intel format:
  - Intel code omits the size designation suffixes.
    - `mov` in intel format while `movq` in ATT format.
  - Intel code omits the `%` character in front of register names, using `rbx` instead of `%rbx`.
  - Intel code has a different way of describing locations in memory—for example, `QWORD PTR [rbx]` rather than `(%rbx)`.
  - Instructions with multiple operands list them in the reverse order.
    - Example: if we want move the content in rdx to rbx,
      - in intel format: `mov rbx, rdx`
      - in ATT format: `movq %rdx, %rbx`