NTNU | Norwegian University of Science and Technology

Compiler Construction

Lecture 15: x86-64 and real world procedures

Michael Engel

Overview

- The real world: x86-64
  - Structure of ELF files
  - Storage types
  - Loading and running programs
  - A quick x86-64 assembler intro
  - Procedures and parameters on x86-64

Structure of executable files

- Early executable files were mostly just the program's instructions and data dumped to disk
  - Example: MS-DOS and CP/M `.COM` files

- Modern executable files are no longer simple images of the program's memory
  - Instead, they have structure and meta information
- Different executable formats exist
  - Unix/Linux: **ELF** ("Executable and Linking Format") [1]
  - Windows: **PE** ("Portable Executable") [2]
  - MacOS X: **Mach-O** ("Mach-Object") [3]

ELF file structure

ELF header

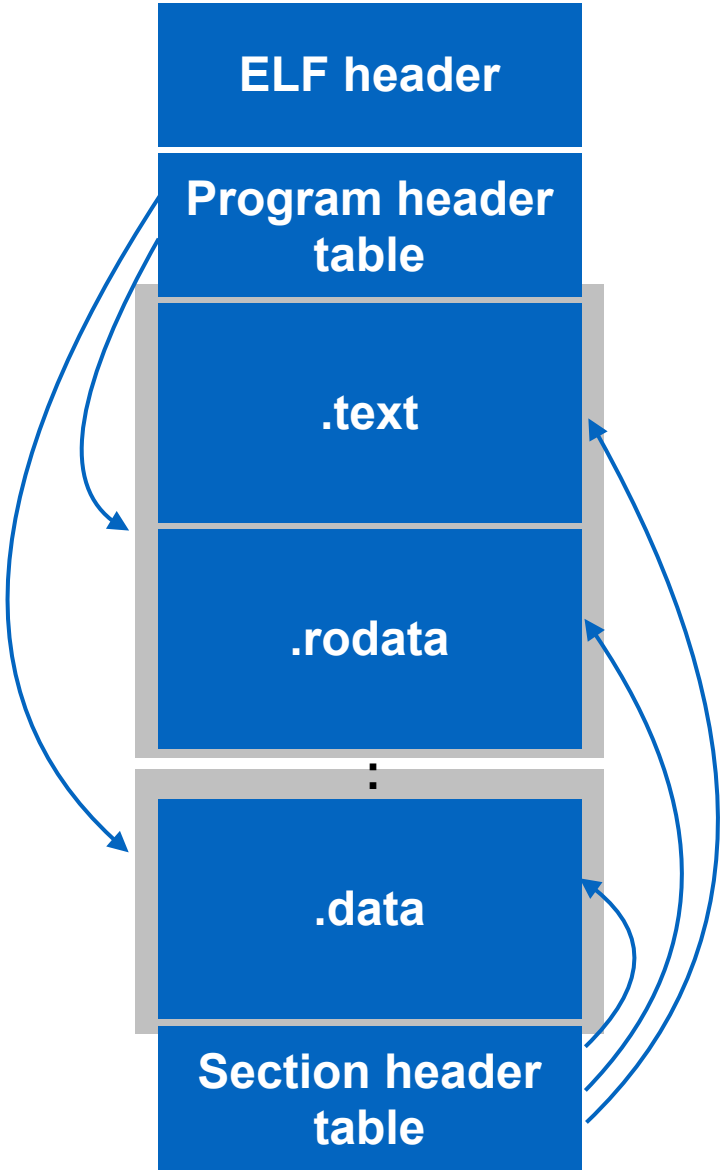
- ELF identifier
- target architecture and size
- pointers to start of program and section header table in the file

Program header table

- tells the system how to create a process image

Section header table

- contains information about and pointers to the different code and data sections of the ELF file



ELF header structure

Defined (along with the rest of ELF file structure) in **`/usr/include/elf.h`**

```
typedef struct
{
  unsigned char e_ident[EI_NIDENT];  /* Magic number and other info */
  Elf64_Half    e_type;       /* Object file type */
  Elf64_Half    e_machine;    /* Architecture */
  Elf64_Word    e_version;    /* Object file version */
  Elf64_Addr    e_entry;      /* Entry point virtual address */
  Elf64_Off     e_phoff;      /* Program header table file offset */
  Elf64_Off     e_shoff;      /* Section header table file offset */
  Elf64_Word    e_flags;      /* Processor-specific flags */
  Elf64_Half    e_ehsize;     /* ELF header size in bytes */
  Elf64_Half    e_phentsize;  /* Program header table entry size */
  Elf64_Half    e_phnum;      /* Program header table entry count */
  Elf64_Half    e_shentsize;  /* Section header table entry size */
  Elf64_Half    e_shnum;      /* Section header table entry count */
  Elf64_Half    e_shstrndx;   /* Section header string table index */
} Elf64_Ehdr;
```

ELF header structure in real life

```
typedef struct
{
  unsigned char e_ident[EI_NIDENT];
  Elf64_Half    e_type;
  Elf64_Half    e_machine;
  Elf64_Word    e_version;
  Elf64_Addr    e_entry;
  Elf64_Off     e_phoff;
  Elf64_Off     e_shoff;
  Elf64_Word    e_flags;
  Elf64_Half    e_ehsize;
  Elf64_Half    e_phentsize;
  Elf64_Half    e_phnum;
  Elf64_Half    e_shentsize;
  Elf64_Half    e_shnum;
  Elf64_Half    e_shstrndx;
} Elf64_Ehdr;
```

Let's take a look inside the `ls` program:

```
$ xxd -c8 /bin/ls
00000000: 7f45 4c46 0201 0100  .ELF....
00000008: 0000 0000 0000 0000  ........
00000010: 0300 3e00 0100 0000  ..>.....
00000018: 3061 0000 0000 0000  0a......
00000020: 4000 0000 0000 0000  @.......
00000028: 2817 0200 0000 0000  (......
00000030: 0000 0000 4000 3800  ....@.8.
00000038: 0b00 4000 1d00 1c00  ..@.....
```

Ugh! There has to be a better way to look inside ELF files!

ELF header structure – readable

```
typedef struct
{
  unsigned char e_ident[EI_NIDENT];
  Elf64_Half    e_type;
  Elf64_Half    e_machine;
  Elf64_Word    e_version;
  Elf64_Addr    e_entry;
  Elf64_Off     e_phoff;
  Elf64_Off     e_shoff;
  Elf64_Word    e_flags;
  Elf64_Half    e_ehsize;
  Elf64_Half    e_phentsize;
  Elf64_Half    e_phnum;
  Elf64_Half    e_shentsize;
  Elf64_Half    e_shnum;
  Elf64_Half    e_shstrndx;
} Elf64_Ehdr;
```

```
$ readelf -h /bin/ls
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x6130
  Start of program headers:          64 (bytes into file)
  Start of section headers:          137000 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         11
  Size of section headers:           64 (bytes)
  Number of section headers:         29
  Section header string table index: 28
```

ELF sections and symbols

Section	Function
Text (.text)	Machine code, entry point
Read-only data (.rodata)	Pre-initialized constants
Read-write data (.data)	Pre-initialized variables
Base storage segment (.bss)	Uninitialized data
Symbols (.symtab)	Addresses for symbolic names
others	Debug information, C++ constructors, …

ELF sections and symbols

```
$ readelf -W -S /bin/ls
There are 29 section headers, starting at offset 0x21728:
Section Headers:
  [Nr] Name              Type            Address           Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            0000000000000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        00000000000002a8 0002a8 00001c 00   A  0   0  1
  [ 5] .dynsym           DYNSYM          00000000000003c0 0003c0 000c60 18   A  6   1  8
  [ 6] .dynstr           STRTAB          0000000000001020 001020 0005cc 00   A  0   0  1
    :
  [14] .text             PROGBITS        00000000000046f0 0046f0 0125be 00  AX  0   0 16
    :
  [16] .rodata           PROGBITS        0000000000017000 017000 005129 00   A  0   0 32
    :
  [25] .data             PROGBITS        0000000000022380 021380 000268 00  WA  0   0 32
  [26] .bss              NOBITS          0000000000022600 0215e8 0012d8 00  WA  0   0 32
    :
  [28] .shstrtab         STRTAB          0000000000000000 02161c 00010a 00      0   0  1
```

From C code to ELF sections

foo.c

```c
const int a = 42;
int b = 23;
int c;

int main(void) {
  c = a + b;
  return c;
}
```

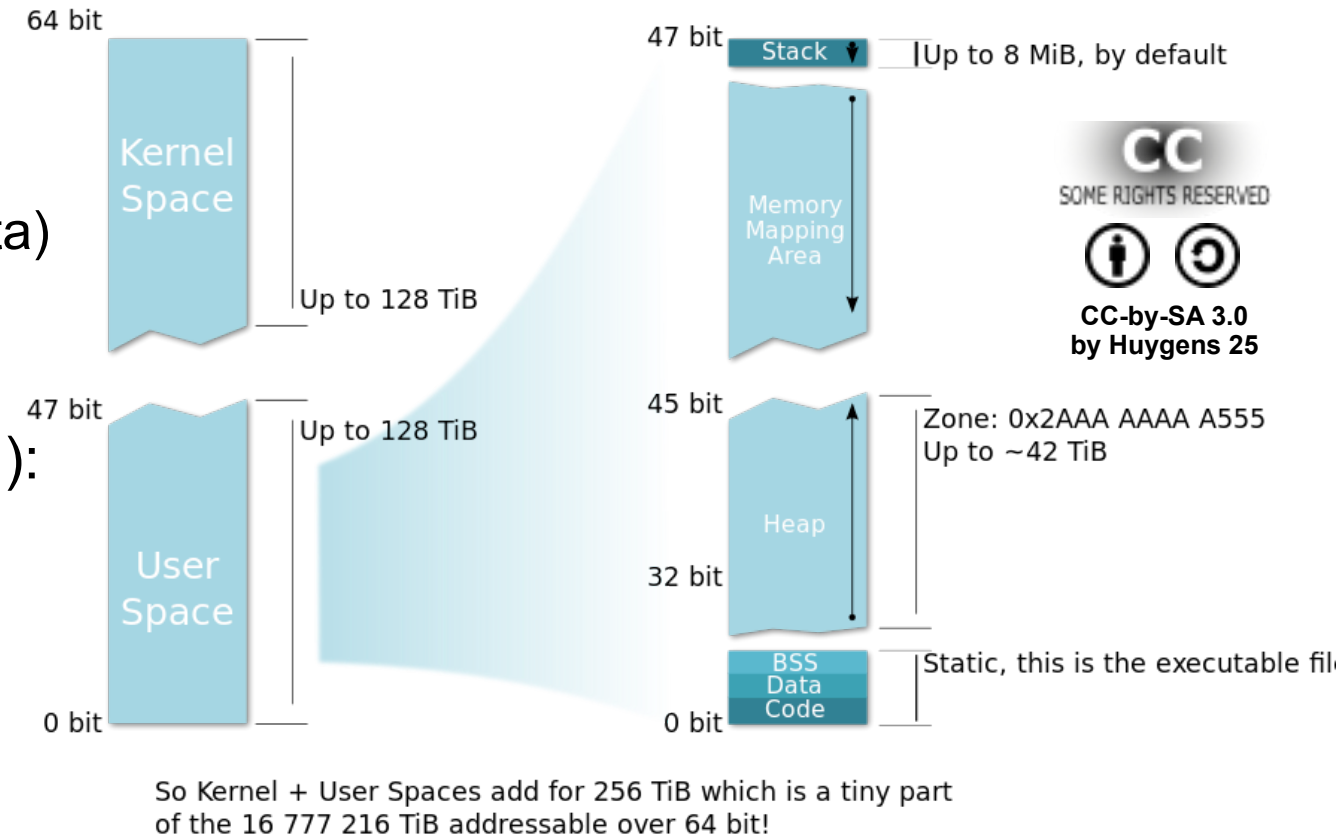
- What goes where in our object file?

```
$ gcc -c foo.c
$ readelf -W -S foo.o
There are 13 section headers, starting at offset 0x2d0:
Section Headers:
  [Nr] Name          Type      Address           Off    Size   ES Flg Lk Inf Al
  [ 0]               NULL      0000000000000000 000000 000000 00      0   0  0
  [ 1] .text         PROGBITS  0000000000000000 000040 00001f 00  AX  0   0  1
  [ 2] .rela.text    RELA      0000000000000000 000208 000048 18   I 10   1  8
  [ 3] .data         PROGBITS  0000000000000000 000060 000004 00  WA  0   0  4
  [ 4] .bss          NOBITS    0000000000000000 000064 000000 00  WA  0   0  1
  [ 5] .rodata       PROGBITS  0000000000000000 000064 000004 00   A  0   0  4
```

Section	Function
Text (.text)	Machine code, entry point
Read-only data (.rodata)	Pre-initialized constants
Read-write data (.data)	Pre-initialized variables
Base storage segment (.bss)	Uninitialized data
Symbols (.symtab)	Addresses for symbolic names
others	Debug information, C++ constructors, …

Code/data storage – from disk to RAM

- What happens when the OS loads a program?
  - ELF gets loaded into ***virtual address space*** (47 bit = 128 TB)
- **Low addresses** (0x0 upwards):
  - Code (.text)
  - Data (.rodata, .data)
  - BSS
- **High addresses** (0x7fff ffff ffff down):
  - Stack
- **In between:**
  - Heap (`malloc`)
  - Shared libraries



CC
SOME RIGHTS RESERVED
CC-by-SA 3.0
by Huygens 25

The 64-bit x86 processors

- Probably what's running in your PC / notebook…
- Evolutionary design, started with the 8086 in 1978
  - Still backwards compatible… with more and more features added
- Complex Instruction Set Computer (CISC)
  - Lots of different instructions
  - Compilers only use a small subset!
- 1985: Intel 80386: first 32-bit x86 CPU
- 2003: AMD Opteron: first 64-bit x86 CPU ("AMD64" = x86-64)
- Almost all modern x86 processors support the x86-64 mode
  - But there is lots of 32 bit code still used
- We will concentrate on 64 bit only here

CC
SOME RIGHTS RESERVED
CC-by-SA 3.0
by Konstantin Lanzet/
Ruben de Rijcke

x86-64 instruction set architecture

- The x86 architecture made the transition from a single-core 16-bit processor to multicore 64-bit processors
    - Basis of the x86-64 **Instruction Set Architecture** (**ISA**)
- Retained compatibility,
    - extended the features and data sizes
- Data types encoded in instruction names:
    - b = byte (8 bit): `movb`
    - w = word (16 bit): `movw`
    - l = long word (32 bit): `movl`
    - q = quad word (64 bit): `movq`

C type/size	x86-32	x86-64
unsigned	4	4
int	4	4
long int	4	8
char	1	1
short	2	2
float	4	4
double	8	8
long dbl.	10/12	16
pointer	4	8

x86-64 registers

x86-64 processors provide the programmer with 16 64-bit registers

Lower 32 bit (LSBs) accessible as eax, ebx… resp. r8d, r9d, ...

63	31 0
%rax	%eax
%rbx	%ebx
%rcx	%ecx
%rdx	%edx
%rsi	%esi
%rdi	%edi
%rsp	%esp
%rbp	%ebp

63	31 0
%r8	%r8d
%r9	%r9d
%r10	%r10d
%r11	%r11d
%r12	%r12d
%r13	%r13d
%r14	%r14d
%r15	%r15d

x86-64 registers

For historical reasons, parts of some of the registers can also be accessed byte- (%al/%ah) and 16 bit (%ax) wise

Historical origin of register name:

accumulator

base register

counter

data

source **i**ndex

destination **i**ndex

stack **p**ointer

base **p**ointer

Bits 0-7: al, bl, cl, dl
Bits 8-15: ah, bh, ch, dh
Bits 0-15: ax, bx, cd, dx, si, di, sp, bp
Bits 0-31: eax, ebx, ecx, … r8d, r9d,…
Bits 0-63: rax, rbx, rcx, …, r8, r9, …

x86-64 assembler operations

Types of operations:

- Transfer data between memory and register
    - Load data from memory into register
    - Store register data into memory
- Perform arithmetic or logic function on register or memory data
- Transfer control
    - Unconditional jumps
    - Jumps to/from procedures
    - Conditional branches
    - Indirect branches

x86-64: moving data

- Moving (in fact: copying) data: `movl source, dest`
- Operand types
  - **Immediate**: constant integer data
    - Example: `$0x400`, `$-533`
    - Like a C constant, but prefixed with "`$`"
    - Encoded using 1, 2, 4 or 8 bytes
  - **Register**: one of 16 integer registers
    - Example `%eax`, `%edx`
    - `%esp` and `%ebp` reserved for stack handling
  - **Memory**: data in memory at address given by register
    - Simple example: **(%eax)**

x86-64: movl operand combinations

	Source	Dest	Src,Dest	C Analog
movl	*Imm*	*Reg*	`movl $0x4,%eax`	`temp = 0x4;`
		Mem	`movl $-147,(%eax)`	`*p = -147;`
	Reg	*Reg*	`movl %eax,%edx`	`temp2 = temp1;`
		Mem	`movl %eax,(%edx)`	`*p = temp;`
	Mem	*Reg*	`movl (%eax),%edx`	`temp = *p;`

- Memory-to-memory transfers are **not** possible with a single instruction

x86-64: memory addressing modes

- Normal `( R )`: `Mem[Reg[R]]` – `movl (%rcx), %eax`
  - Register `R` specifies memory address
  - e.g. used for pointers

```
int r, *x;  // r=%ecx, x=%rax
r = *x;  -> movl (%rax), %ecx
*x = 42; -> movl $42, (%rax)
```

- Displacement `D( R )`: `Mem[Reg[R]+D]` – `movq 8(%rbp), %rdx`
  - Register `R`: start of memory region
  - Constant "displacement" `D`: offset
  - e.g. accesses to struct elements

```
struct { int x; int y; int z; } s;
int r;     // r=%ecx, &s=%rax
r = s.y;   -> movl 4(%rax), %ecx
s.z = 42;  -> movl $42, 8(%rax)
```

- General form `D(Rb,Ri,S)`: `Mem[Reg[Rb]+S*Reg[Ri]]`
  - `D`: constant displacement, `Rb`: base register, `Ri`: index register
  - `S`: scale factor 1, 2, 4 or 8

x86-64: memory addressing modes

- Examples for address calculations
    - It is often possible to omit a register
        - e.g. the one in the last line of the table
- A good overview and much more detail in the "x64 cheat sheet" [4]

%rdx	0xf000
%rcx	0x0100

Expression	Address calculation	Address
0x8(%rdx)	0xf000+0x8	0xf008
(%rdx,%rcx)	0xf000+0x0100	0xf100
(%rdx,%rcx,4)	0xf000+4*0x0100	0xf400
0x80(,%rdx,2)	2*0xf000+0x80	0x1e080

x86-64: arithmetic and logic operations

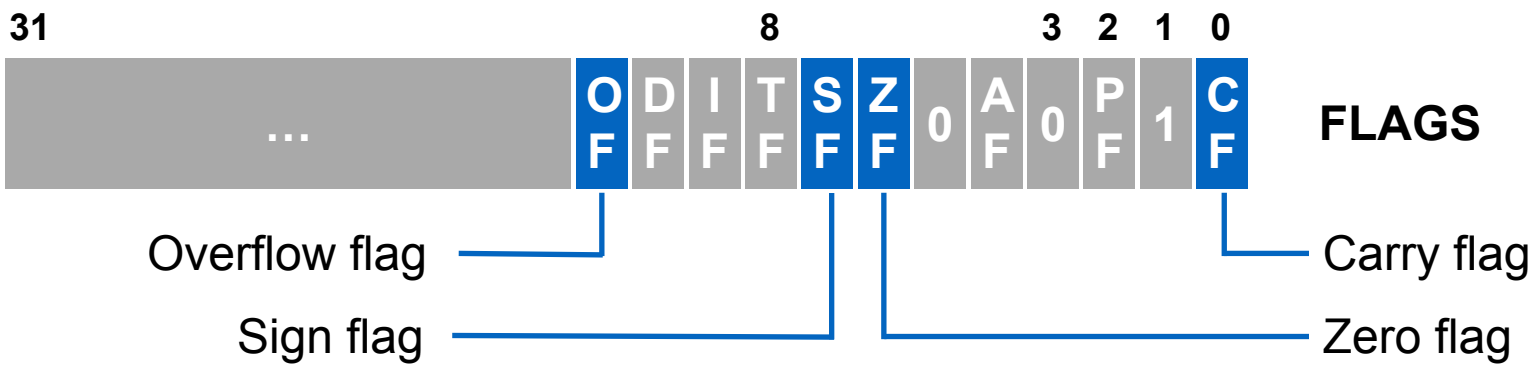
- ALU operations
  - binary: **`add`**, **`sub`**, **`imul`**, **`xor`**, **`or`**, **`and`**
  - unary: **`inc`**, **`dec`**, **`neg`**, **`not`**
  - shifts: **`sal`**/**`shl`**, **`sar`**, **`shr`**

```
addq $42, %rax // %rax = %rax + 42
```

```
incq %rcx      // %rcx = %rcx + 1
```

```
shlq $2, %rcx  // %rcx = %rcx * 4
```

- Result of arithmetic instructions:
  - computed result + meta information recorded in FLAGS register: **sign** (negative), **zero**, **carry**, arithmetic **overflow**



x86-64: control flow

Conditional jumps: `j<cond> dest`

- check state of one flag: `Z`, `C`, `S`, `O`
- jump if check succeeds

Example: compare a constant to a value in register eax:

```
cmpl $42, %eax
```

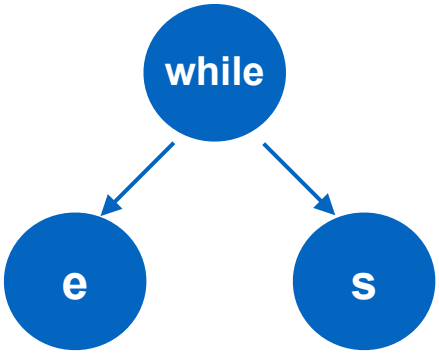
- Compare is a subtraction that does not store the result, but sets flags:

```
%eax==42? → 42-eax==0 → ZF=1
```

Instruction	Description	flags
`JO`	Jump if overflow	`OF=1`
`JNO`	Jump if not overflow	`OF=0`
`JS`	Jump if sign	`SF=1`
`JNS`	Jump if not sign	`SF=0`
`JE` `JZ`	Jump if equal Jump if zero	`ZF=1`
`JNE` `JNZ`	Jump if not equal Jump if not zero	`ZF=0`
…	…and many more…	

```
.LBB0_1:
    cmpl $0, -8(%rbp)   ; set flags
    je   .LBB0_3        ; ZF==1? → jump
    …                   ; ZF!=1? → continue
.LBB0_3:
    …
```

Example: while loop




```
int i;
i=10;
while(i) {
    i--;
}
```

C source
to IR
to x86-64 assembly

`T[while(e) do s]` **becomes**

```
Ltest:
    t1 = T[e]
    ifFALSE t1 goto Lend
    T[s]
    jump Ltest
Lend:
```

`t1` = **true**

`t1` = **false**

```
    movl $10, -8(%rbp)
.LBB0_1:
    cmpl $0, -8(%rbp)
    je   .LBB0_3
# %bb.2:
    movl -8(%rbp), %eax
    addl $-1, %eax
    movl %eax, -8(%rbp)
    jmp  .LBB0_1
.LBB0_3:
    …
```

`ZF` = **false**

`ZF` = **true**

x86-64: control flow

- Unconditional jumps: `j <target>`
  - continues program execution at indicated location
  - target can be constant or using addressing modes
  - Example: jump table for `switch`

Jump table (in data segment):

```
.section rodata
   .align 4
.L7:
   .long L2 ; x=0
   .long L3 ; x=1
   .long L4 ; x=2
```

```
int switchme(int x) {
  int w;
  switch(x) {
    case 0: w=1; break;
    case 1: w=2; break;
    case 2: w=4; break;
  }
  return w;
}
```

```
switchme:                ; x=%eax, w=%ecx
     …
     cmpl $2, %eax       ; x > 2?
     jgt  .L5            ; yes -> exit
     jmp  *.L7(,%eax,8)  ; goto *jtab[x]

.L2: movl $1, %ecx       ; w=1
     jmp  .L5            ; break

.L3: movl $2, %ecx       ; w=2
     jmp  .L5            ; break

.L4: movl $4, %ecx       ; w=4

.L5: …          ; after the end of switch
```

x86-64: call and return

```
0x1000    callq .L42
0x1005
…
.L42:
0x2000    retq
```

- Procedure calls: **`callq`**/**`retq`**

- **`callq`** performs two actions:
  - push **return address** (address of the instruction following **`callq`**) on the stack
  - jump to given target address

push 0x1005 on stack

jump to 0x2000

- **`retq`** reverses this:
  - pop **return address** from the stack
  - jump to this address

pop topmost value from stack → 0x1005

jump to 0x1005

Procedures on x86-64

- x86 has only minimal support for procedures: `call` and `return`

- Compilers have to implement all other features
  - e.g. parameter passing and saving of context
  - conventions for these features defined in the **Application Binary Interface** (**ABI**) [5]
  - required for interoperability
    - e.g., linking to code compiled with a different compiler
- Example conventions:
  - how to pass parameters and in which order
  - how to return values
  - how to pass complex parameters (e.g., structs)
  - which register contents need to be preserved across procedure calls

Procedure context

- The compiler cannot know which registers can be safely used (=overwritten) in a procedure
    - since a procedure can be called from arbitrary locations
- Solution: save all register values → too much overhead

Instead: use a register usage convention

- Registers `%rbp`, `%rbx` and `%r12` through `%r15` "belong" to the calling function
  - the called function is required to preserve their values
  - In other words, a called function must preserve these registers' values for its caller
- Remaining registers "belong" to the called function
  - If a calling function wants to preserve such a register value across a function call, it must save the value in its local stack frame

Procedure parameters

- Again, just a convention of the ABI
- Two general methods exist
  - on the stack: uses space and time (32 bit x86)
  - in registers: restrictions in the number of params (x86-64)
- Convention for x86-64:
  - first six integer or pointer parameters passed in registers **`%rdi`**, **`%rsi`**, **`%rdx`**, **`%rcx`**, **`%r8`**, and **`%r9`**
  - subsequent parameters (or parameters larger than 64 bits) pushed onto the stack, with the first argument topmost
  - return value in **`%rax`**

```
; Call foo(1, 15)
movq  $1, %rdi
movq  $15, %rsi
call  foo
```

Local variables

```
int foo(int x, int y) {
  return x+y;
}
```

- The stack is not only used for storing return addresses
- Local variables are also stored there, including formal parameters
- This function-local information is called a **stack frame**
- Often, a **base pointer** is used to indicate the stack frame start
    - offsets from the base pointer are used to access variables

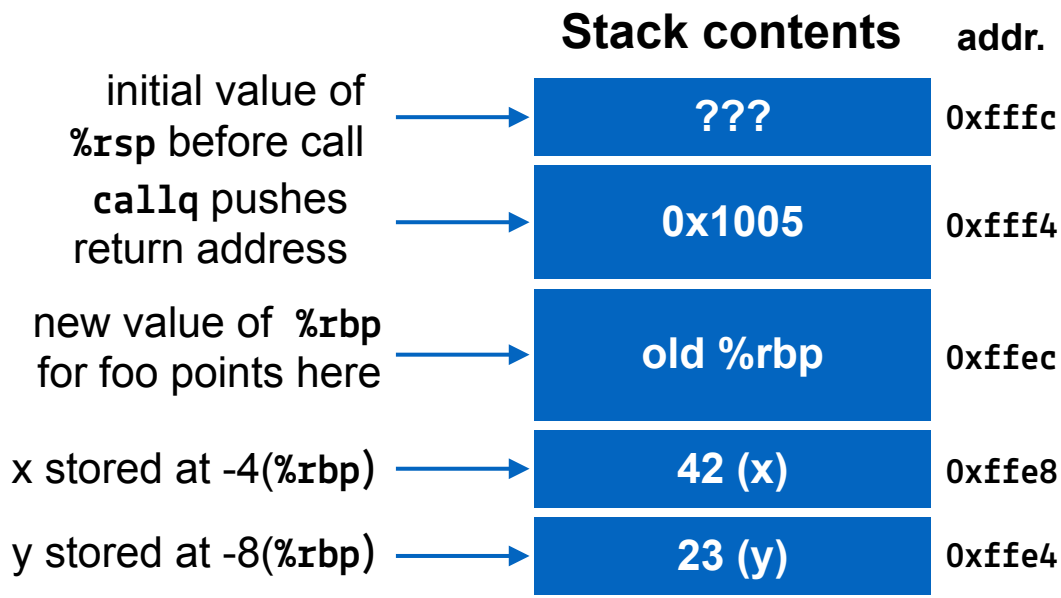
```
foo:
   push   %rbp              ; save old base pointer to stack
   mov    %rsp,%rbp         ; current stack location: new base
   mov    %edi,-0x4(%rbp)   ; 1st actual param (in %edi) -> x
   mov    %esi,-0x8(%rbp)   ; 2nd actual param (in %esi) -> y
   mov    -0x4(%rbp),%esi   ; read value of x -> %esi
   add    -0x8(%rbp),%esi   ; add  value of y to %esi
   mov    %esi,%eax         ; return value is returned in %eax
   pop    %rbp              ; restore previous base pointer
   retq
```

The x86-64 stack

```
bar:     ; calls foo(42, 23);
   mov    $42, %edi
   mov    $23, %esi
   callq  foo
   …      ; this is address 0x1005
```

```
foo:
   push   %rbp
   mov    %rsp,%rbp
   mov    %edi,-0x4(%rbp)
   mov    %esi,-0x8(%rbp)
   mov    -0x4(%rbp),%esi
   add    -0x8(%rbp),%esi
   mov    %esi,%eax
   pop    %rbp
   retq
```

- The stack on x86-64 grows **downwards** (from high to low addresses)
- **push** decrements `%rsp`, then writes value to stack (`%rsp`)
- **pop** gets value from (`%rsp`), then increments `%rsp`
- In/decrement depends on data type
  - addresses/pointers use 8 bytes
  - ints use 4 bytes



Example disassembly

```
$ objdump -S w.o   ; -S prints C source code if compiled with -g
0000000000000000 <foo>:
int foo(void) {
   0:     55                          push   %rbp
   1:     48 89 e5                    mov    %rsp,%rbp
    int i;
    i=10;
   4:     c7 45 f8 0a 00 00 00        movl   $0xa,-0x4(%rbp)
    while(i) {
   b:     83 7d f8 00                 cmpl   $0x0,-0x4(%rbp)
   f:     0f 84 0e 00 00 00           je     23 <foo+0x23>
          i--;
  15:     8b 45 f8                    mov    -0x4(%rbp),%eax
  18:     83 c0 ff                    add    $0xffffffff,%eax
  1b:     89 45 f8                    mov    %eax,-0x4(%rbp)
    while(i) {
  1e:     e9 e8 ff ff ff              jmpq   b <foo+0xb>
    }
    return i;
  23:     8b 45 fc                    mov    -0x4(%rbp),%eax
  26:     5d                          pop    %rbp
  27:     c3                          retq
```

```
int foo(void) {
    int i;
    i=10;

    while(i) {
      i--;
    }
    return i;
}
```

i (stored at address %rbp-4) is initialized with 0xa=10

i == 0? jump to address 0x23

copy i into register %eax
subtract 1 from register %eax
copy register %eax back to i

jump back to address 0xb

copy i into register %eax (register for return value)

Working with x86-64 assembler

- **Tips and tricks**
  - Output compiled assembler source: `gcc/clang -S file.c`
  - Disassemble binary file: `objdump /bin/ls`
  - Get addresses and names of symbols: `nm /bin/ls`
- **Caution**
  - There are two different styles of x86 assembler syntax: intel and AT&T (Unix)
    - These differ in the order of parameters (& other things)!
  - We use AT&T syntax (since that's what Linux understands)
  - Take care when looking at examples from the Internet
- **Take a look at compilation results**: Godbolt compiler explorer [6]

What's next?

- Optimizations, optimizations…

References

[1] Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2 (May 1995)

[2] Microsoft Corp., Microsoft Portable Executable and Common Object File Format Specification", Revision 6, Februar 1999 http://www.osdever.net/documents/PECOFF

[3] Apple, Inc., Mach-O Runtime Architecture, 2004

[4] Tom Doeppner, Brown University x64 Cheat Sheet https://cs.brown.edu/courses/cs033/docs/guides/x64_cheatsheet.pdf

[5] System V Application Binary Interface Edition 4.1 (1997-03-18)

[6] Matt Godbolt's Compiler Explorer: https://godbolt.org/